OCaml Programming: Correct + Efficient + Beautiful

Michael R. Clarkson et al.

Jan 29, 2022

CONTENTS

A textbook on functional programming and data structures in OCaml, with an emphasis on semantics and software engineering. This book is the textbook for CS 3110 Data Structures and Functional Programming at Cornell University. A past title of this book was "Functional Programming in OCaml".

Fall 2021 Edition. For the most recent version of this work, see the most recent CS 3110 course website.

Videos. There are over 200 YouTube videos embedded in this book. They can be watched independently of reading the book. Start with this YouTube playlist.

Authors. This book is based on courses taught by Michael R. Clarkson, Robert L. Constable, Nate Foster, Michael D. George, Dan Grossman, Daniel P. Huttenlocher, Dexter Kozen, Greg Morrisett, Andrew C. Myers, Radu Rugina, and Ramin Zabih. Together they have created over 20 years worth of course notes and intellectual contributions. Teasing out who contributed what is, by now, not an easy task. The primary compiler and author of this work in its form as a unified textbook is Michael R. Clarkson, who as of this edition is the author of about 40% of the words and code tokens.

Part I

Preface

CHAPTER

ONE

ABOUT THIS BOOK

Reporting Errors. If you find an error, please report it! Or if you have a suggestion about how to rewrite some part of the book, let us know. Just go to the page of the book for which you'd like to make a suggestion, click on the Github icon (it looks like a cat) near the top right of the page, and click "open issue" or "suggest edit". The latter is a little heavier weight, because it requires you to fork the textbook repository with Github. But for minor edits that will be appreciated and lead to much quicker uptake of suggestions.

Background. This book is used at Cornell for a third-semester programming course. Most students have had one semester of introductory programming in Python, followed by one semester of object-oriented programming in Java. Frequent comparisons are therefore made to those two languages. Readers who have studied similar languages should have no difficulty following along. The book does not assume any prior knowledge of functional programming, but it does assume that readers have prior experience programming in some mainstream imperative language. Knowledge of discrete mathematics at the level of a standard first-semester CS course is also assumed.

Videos. You will find over 200 YouTube videos embedded throughout this book. The videos usually provide an introduction to material, upon which the textbook then expands. These videos were produced during pandemic when the Cornell course that uses this textbook, CS 3110, had to be asynchronous. The student response to them was overwhelmingly positive, so they are now being made public as part of the textbook. But just so you know, they were not produced by a professional A/V team—just a guy in his basement who was learning as he went.

The videos mostly use the versions of OCaml and its ecosystem that were current in Fall 2020. Current versions you are using are likely to look different from the videos, but don't be alarmed: the underlying ideas are the same. The most visible difference is likely to be the VS Code plugin for OCaml. In Fall 2020 the badly-aging "OCaml and Reason IDE" plugin was still being used. It has since been superseded by the "OCaml Platform" plugin.

The order that the textbook covers topics sometimes differs from the order that the videos cover the topics, simply because the videos originate from lectures. The videos are placed in the textbook nearest to the topic they cover, but that does mean sometimes the videos are not in chronological order. To watch them in their original order, start with this YouTube playlist.

Collaborative Annotations. At the right margin of each page, you will find an annotation feature provided by hypothes.is. You can use this to highlight and make private notes as you study the text. You can form study groups to share your annotations, or share them publicly. Check out these tips for how to annotate effectively.

Executable Code. Many pages of this book have OCaml code embedded in them. The output of that code is already shown in the book. Here's an example:

```
print_endline "Hello world!"
```

You can also edit and re-run the code yourself to experiment and check your understanding. Look for the icon near the top right of the page that looks like a rocket ship. In the drop-down menu you'll find two ways to interact with the code:

- *Binder* will launch the site mybinder.org, which is a free cloud-based service for "reproducible, interactive, sharable environments for science at scale." All the computation happens in their cloud servers, but the UI is provided through your browser. It will take a little while for the textbook page to open in Binder. Once it does, you can

edit and run the code in a *Jupyter notebook*. Jupyter notebooks are documents (usually ending in the `.ipynb` extension) that can be viewed in web browsers and used to write narrative content as well as code. They became popular in data science communities (especially Python, R, and Julia) as a way of sharing analyses. Now many languages can run in Jupyter notebooks, including OCaml. Code and text are written in *cells* in a Jupyter notebook. Look at the "Cell" menu in it for commands to run cells. Note that Shift-Enter is usually a hotkey for running the cell that has focus.

- *Live code* will actually do about the same thing, except that instead of leaving the current textbook page and taking you off to Binder, it will modify the code cells on the page to be editable. It takes some time for the connection to be made behind the scenes, during which you will see "Waiting for kernel". After the connection has been made, you can edit all the code cells on the page and re-run them.

Try interacting with the cell above now to make it print a string of your choice. How about: `"Camels are bae."`

Tip: When you write "real" OCaml code, this is not the interface you'll be using. You'll write code in an editor such as Visual Studio Code or Emacs, and you'll compile it from a terminal. Binder and Live Code are just for interacting seamlessly with the textbook.

Downloadable Pages. Each page of this book is downloadable in a variety of formats. The download icon is at the top right of each page. You'll always find the original source code of the page, which is usually Markdown—or more precisely MyST Markdown, which is an extension of Markdown for technical writing. Each page is also individually available as PDF, which simply prints from your browser. For the entire book as a PDF, see the paragraph about that below.

Pages with OCaml code cells embedded in them can also be downloaded as Jupyter notebooks. To run those locally on your own machine (instead of in the cloud on Binder), you'll need to install Jupyter. The easiest way of doing that is typically to install Anaconda. Then you'll need to install OCaml Jupyter, which requires that you already have OCaml installed. To be clear, there's no need to install Jupyter or to use notebooks. It's just another way to interact with this textbook beyond reading it.

Exercises and Solutions. At the end of each chapter except the first, you will find a section of exercises. The exercises are annotated with a difficulty rating:

- One star [★]: easy exercises that should take only a minute or two.
- Two stars [★★]: straightforward exercises that should take a few minutes.
- Three stars [★★★]: exercises that might require anywhere from five to twenty minutes or so.
- Four [★★★★] or more stars: challenging or time-consuming exercises provided for students who want to dig deeper into the material.

It's possible we've misjudged the difficulty of a problem from time to time. Let us know if you think an annotation is off.

Please do not post your solutions to the exercises anywhere, especially not in public repositories where they could be found by search engines. Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

PDF. A full PDF version of this book is available. It does not contain the embedded videos, annotations, or other features that the HTML version has. It might also have typesetting errors. At this time, no tablet (ePub, etc.) version is available, but most tablets will let you import PDFs.

CHAPTER

TWO

INSTALLING OCAML

If all you need is a way to follow along with the code examples in this book, you don't actually have to install OCaml! The code on each page is executable in your browser, as described earlier in this *Preface*.

If you want to take it a step further but aren't ready to spend time installing OCaml yourself, we provide a *virtual machine* with OCaml pre-installed inside a Linux OS.

But if you want to do OCaml development on your own, you'll need to install it on your machine. There's no universally "right" way to do that. The instructions below are for Cornell's CS 3110 course, which has goals and needs beyond just OCaml. Nonetheless, you might find them to be useful even if you're not a student in the course.

Here's what we're going to install:

- A Unix development environment
- OPAM, the OCaml Package Manager
- An OPAM *switch* with the OCaml compiler and some packages
- The Visual Studio Code editor, with OCaml support

The installation process will rely heavily on the *terminal*, or text interface to your computer. If you're not too familiar with it, you might want to brush up with a terminal tutorial.

Tip: If this is your first time installing development software, it's worth pointing out that "close doesn't count": trying to proceed past an error usually just leads to worse errors, and sadness. That's because we're installing a kind of tower of software, with each level of the tower building on the previous. If you're not building on a solid foundation, the whole thing might collapse. The good news is that if you do get an error, you're probably not alone. A quick google search will often turn up solutions that others have discovered. Of course, do think critically about suggestions made by random strangers on the internet.

Let's get started!

2.1 Unix Development Environment

First, upgrade your OS. If you've been intending to make any major OS upgrades, do them now. Otherwise when you do get around to upgrading, you might have to repeat some or all of this installation process. Better to get it out of the way beforehand.

Linux. If you're already running Linux, you're done with this step. Proceed to OPAM, below.

Mac. Beneath the surface, macOS is already a Unix-based OS. But you're going to need some developer tools and a Unix package manager. There are two to pick from: Homebrew and MacPorts. From the perspective of this textbook and CS

3110, it doesn't matter which you choose. So if you're already accustomed to one, feel free to keep using it. Make sure to run its update command before continuing with these instructions.

Otherwise, pick one and follow the installation instructions on its website. The installation process for Homebrew is typically easier and faster, which might nudge you in that direction. If you do choose MacPorts, make sure to follow *all* the detailed instructions on its page, including XCode and an X11 server. **Do not install both Homebrew and MacPorts**; they aren't meant to co-exist. If you change your mind later, make sure to uninstall one before installing the other. After you've finished installing either Homebrew or MacPorts, you can proceed to OPAM, below.

Windows. Unix development in Windows 10 is made possible by the Windows Subsystem for Linux (WSL). Follow Microsoft's install instructions for WSL. Here are a few notes on Microsoft's instructions:

- From the perspective of this textbook and CS 3110, it doesn't matter whether you join Windows Insider.
- WSL2 is preferred over WSL1 by OCaml (and WSL2 offers performance and functionality improvements), so install WSL2 if you can.
- To open Windows PowerShell as Administrator, click Start, type PowerShell, and it should come up as the best match. Click "Run as Administrator", and click Yes to allow changes.
- To use WSL2 (rather than WSL1) you might need to enable virtualization in your machine's BIOS; some laptop manufacturers disable it before shipping machines from the factory. The instructions for that are dependent on the manufacturer of your machine. Try googling "enable virtualization ", substituting for the manufacturer and model of your machine. This Red Hat Linux page might also help.
- These instructions assume that you install Ubuntu (20.04) as the Linux distribution from the Microsoft Store. In principle other distributions should work, but might require different commands from this point forward.
- You will be prompted to create a Unix username and password. You can use any username and password you wish. It has no bearing on your Windows username and password (though you are free to re-use those). Do not put a space in your username. Do not forget your password. You will need it in the future.
- To enable copy-and-paste, click on the icon on the top left of the shell window, click Properties, and make sure "Use Ctrl+Shift+C/V as Copy/Paste" is checked. Now Ctrl+Shift+C will copy and Ctrl+Shift+V will paste into the terminal. Note that you have to include Shift as part of that keystroke.

When you've finished installing WSL, open the Ubuntu app. You will be at the *Bash prompt*, which looks something like this:

```
user@machine:$
```

Run the following command to update the *APT package manager*, which is what helps to install Unix packages:

```
sudo apt update
```

You will be prompted for the Unix password you chose. The prefix `sudo` means to run the command as the administrator, aka "super user". In other words, do this command as super user, hence, "sudo".

> **Warning:** Running commands with `sudo` is potentially dangerous and should not be done lightly. Do not get into the habit of putting `sudo` in front of commands, and do not randomly try it without reason.

Now run this command to upgrade all the APT software packages:

```
sudo apt upgrade -y
```

WSL has its own filesystem that is distinct from the Windows filesystem, though there are ways to access each from the other. This is a potentially tricky concept to master.

- When you launch Ubuntu and get the $ prompt, you are in the WSL filesystem. Your home directory there is named `~`, which is a built-in alias for `/home/your_user_name`.
- When you use Windows, you are in the Windows filesystem. Microsoft issued one hard-and-fast rule: "Do not under any circumstances access [WSL filesystem] files using Windows." You can corrupt the files.

We recommend storing your OCaml development work in the WSL filesystem, not the Windows filesystem.

2.2 Install OPAM

Linux. Follow the instructions for your distribution.

Mac. If you're using Homebrew, run these commands:

```
brew install gpatch
brew install opam
```

If you're using MacPorts, run this command:

```
sudo port install opam
```

Windows. Run this command from Ubuntu:

```
sudo apt install -y m4 zip unzip bubblewrap build-essential opam
```

2.3 Initialize OPAM

> **Warning:** Do not put `sudo` in front of any `opam` commands. That would break your OCaml installation.

Linux, Mac, and WSL2. Run:

```
opam init --bare -a -y
```

WSL1. Run:

```
opam init --bare -a -y --disable-sandboxing
```

It is necessary to disable sandboxing because of an issue involving OPAM and WSL1.

2.4 Create an OPAM Switch

A *switch* is a named installation of OCaml with a particular compiler version and set of packages. You can have many switches and, well, switch between them —whence the name. Create a switch for this semester's CS 3110 by running this command:

```
opam switch create cs3110-2021fa ocaml-base-compiler.4.12.0
```

Tip: If that command fails saying that the 4.12.0 compiler can't be found, you probably installed OPAM sometime back in the past and now need to update it. Do so with `opam update`.

You might be prompted to run the next command. If so, do it. If not, don't.

```
eval $(opam env)
```

Regardless, continue:

```
opam install -y utop odoc ounit2 qcheck bisect_ppx menhir ocaml-lsp-server␣
↪ocamlformat ocamlformat-rpc
```

(Make sure to grab that whole line above when you copy it.)

You should now be able to launch utop, the OCaml Universal Toplevel.

```
utop
```

Enter 3110 followed by two semi-colons. Press return. The # is the utop prompt.

```
# 3110;;
- : int = 3110
```

Stop to appreciate how lovely `3110` is. Then quit utop. Note that you must enter the extra # before the quit directive.

```
# #quit;;
```

2.5 Double Check OCaml

Let's pause to double check whether your installation has been successful. It's worth the effort!

First, **reboot your computer**. (Really! No matter how silly it might seem, we want a clean slate for this test.) Second, run utop, and make sure it still works. If it does not, here are some common issues:

- **Are you in the right Unix prompt?** On Mac, make sure you are in whatever Unix shell is the default for your Terminal: don't run bash or zsh or anything else manually to change the shell. On Windows, make sure you are in the Ubuntu app, not PowerShell or Cmd.
- **Is the OPAM environment set?** If utop isn't a recognized command, run `eval $(opam env)` then try running utop again. If utop now works, your login shell is somehow not running the right commands to automatically activate the OPAM environment; you shouldn't have to manually activate the environment with the `eval` command. Probably something went wrong earlier when you ran the `opam init` command. To fix it, follow the "redo" instructions below.
- **Is your switch listed?** Run `opam switch list` and make sure a switch named `cs3110-2021fa` is listed, that it has the 4.12.0 compiler, and that it is the active switch (which is indicated with an arrow beside it). If that switch is present but not active, run `opam switch cs3110-2021fa` then see whether utop works. If that switch is not present, follow the "redo" instructions below.

Redo Instructions: Remove the OPAM directory by running `rm -r ~/.opam`, then re-run the OPAM initialization command for your OS (given above), then re-run the switch creation and package installation commands above. Finally, redo the double check: reboot and see whether utop still works. You want to get to the point where utop "just works" after a reboot.

2.6 Visual Studio Code

Visual Studio Code is a great choice as a code editor for OCaml. (Though if you are already a power user of Emacs or Vim those are great, too.)

- Download and install Visual Studio Code (henceforth, VS Code). Launch VS Code. Find the extensions pane, either by going to View -> Extensions, or by clicking on the icon for it in the column of icons on the left.
- **Windows only:** Install the "Remote - WSL" extension. Second, open a WSL window by using the command "Remote-WSL: New Window" or by running `code .` in Ubuntu. Either way, make sure you see the green "WSL" indicator in the bottom-left of the VS Code window. Follow the rest of the instructions in that window.
- **Mac only:** Open the Command Palette and type "shell command" to find the "Shell Command: Install 'code' command in PATH" command. Run it. Then close any open terminals to let the new path settings take effect.
- **For everyone:** In the extensions pane, search for and install the "OCaml Platform" extension. Be careful to use the extension with *exactly the right name* - the correct extension will also have an icon of a camel on a black background, as opposed to a red or orange background. **Windows only:** make sure you install the extension with the button that says "Install on WSL: …", not with a button that says only "Install". The latter will not work.

> **Warning:** The extensions named simply "OCaml" or "OCaml and Reason IDE" are not the right ones. (They are both old and no longer maintained by their developers.)

2.7 Double Check VS Code

Let's make sure VS Code's OCaml support is working.

- Reboot your computer again. (Yeah, that really shouldn't be necessary. But it will detect so many potential mistakes now that it's worth the effort.)
- Open a fresh new Unix shell. **Windows**: remember that's the Ubuntu, not PowerShell or Cmd. **Mac**: remember that you shouldn't be manually switching to a different shell.
- Navigate to a directory of your choice, preferably a subdirectory of your home directory. For example, you might create a directory for your 3110 work inside your home directory:

```
mkdir ~/3110
cd ~/3110
```

  In that directory open VS Code by running:

```
code .
```

  Go to File -> New File. Save the file with the name `test.ml`. VS Code should give it an orange camel icon.

- Type the following OCaml code then press Return:

```
let x : int = 3110
```

  As you type, VS Code should colorize the syntax, suggest some completions, and add a little annotation above the line of code. Try changing the `int` you typed to `string`. A squiggle should appear under `3110`. Hover over it to see the error message. Go to View -> Problems to see it there, too. Add double quotes around the integer to make it a string, and the problem will go away.

If you don't observe those behaviors, something is wrong with your install. Here's how to proceed:

- Do not hardcode any paths in the VS Code settings file, despite any advice you might find online. That is a band-aid, not a cure of whatever the underlying problem really is.
- Make sure that, from the same Unix prompt as which you launched VS Code, you can successfully complete the double-check instructions for your OPAM switch: can you run utop? is the right switch active? If not, that's the problem you need to solve first. Then return to the VS Code issue. It might be fixed now.
- If you're on WSL and VS Code does add syntax highlighting but does not add squiggles as described above, and/or you get an error about "Sandbox initialization failed", then double-check that you see a green "WSL" indicator in the bottom left of the VS Code window. If you do not, make sure you installed the "Remote - WSL" extension as described above, and that you are launching VS Code from Ubuntu rather than PowerShell or from the Windows GUI.

If you're still stuck with an issue, try uninstalling VS Code, rebooting, and re-doing all the install instructions above from scratch. Pay close attention to any warnings or errors.

2.8 VS Code Settings

We recommend tweaking a few editor settings. Open the user settings JSON file by (i) going to View → Command Palette, (ii) typing "settings json", and (iii) selecting Open Settings (JSON). Copy and paste these settings into the window:

```
{
    "editor.tabSize": 2,
    "editor.rulers": [
        80
    ],
    "editor.formatOnSave": true,
}
```

Save the file and close the tab.

2.9 Using VS Code Collaboratively

VS Code's Live Share extension makes it easy and fun to collaborate on code with other humans. You can edit code together like collaborating inside a Google Doc. It even supports a shared voice channel, so there's no need to spin up a separate Zoom call. To install Live Share:

- Open the Extensions page in VS Code. Search for "Live Share Extension Pack". Install it.
- The first time you use Live Share, you will be prompted to login. If you are a Cornell student, choose to login with your Microsoft account, not Github. Enter your Cornell NetID email, e.g., your_netid@cornell.edu. That will take you to Cornell's login site. Use the password associated with your NetID.

To collaborate with Live Share:

- The *host* starts the Live Share session. That generates a URL. Send the URL to the *guests* however you like (DM, email, etc.).
- The guest puts that URL into a browser or directly into VS Code, and connects to the shared programming session.

Part II

Introduction

CHAPTER

THREE

BETTER PROGRAMMING THROUGH OCAML

Do you already know how to program in a mainstream language like Python or Java? Good. This book is for you. It's time to learn how to program better. It's time to learn a functional language, OCaml.

Functional programming provides a different perspective on programming than what you have experienced so far. Adapting to that perspective requires letting go of old ideas: assignment statements, loops, classes and objects, among others. That won't be easy.

> Nan-in, a Japanese master during the Meiji era (1868-1912), received a university professor who came to inquire about Zen. Nan-in served tea. He poured his visitor's cup full, and then kept on pouring. The professor watched the overflow until he no longer could restrain himself. "It is overfull. No more will go in!" "Like this cup," Nan-in said, "you are full of your own opinions and speculations. How can I show you Zen unless you first empty your cup?"

I believe that learning OCaml will make you a better programmer. Here's why:

- You will experience the freedom of *immutability*, in which the values of so-called "variables" cannot change. Goodbye, debugging.
- You will improve at *abstraction*, which is the practice of avoiding repetition by factoring out commonality. Goodbye, bloated code.
- You will be exposed to a *type system* that you will at first hate because it rejects programs you think are correct. But you will come to love it, because you will humbly realize it was right and your programs were wrong. Goodbye, failing tests.
- You will be exposed to some of the *theory and implementation of programming languages*, helping you to understand the foundations of what you are saying to the computer when you write code. Goodbye, mysterious and magic incantations.

All of those ideas can be learned in other contexts and languages. But OCaml provides an incredible opportunity to bundle them all together. **OCaml will change the way you think about programming.**

> "A language that doesn't affect the way you think about programming is not worth knowing."
>
> ---Alan J. Perlis (1922-1990), first recipient of the Turing Award

Moreover, OCaml is beautiful. OCaml is elegant, simple, and graceful. Aesthetics do matter. Code isn't written just to be executed by machines. It's also written to communicate to humans. Elegant code is easier to read and maintain. It isn't necessarily easier to write.

The OCaml code you write can be stylish and tasteful. At first, this might not be apparent. You are learning a new language after all—you wouldn't expect to appreciate Sanskrit poetry on day 1 of Introductory Sanskrit. In fact, you'll likely feel frustrated for awhile as you struggle to express yourself in a new language. So give it some time. After you've mastered OCaml, you might be surprised at how ugly those other languages you already know end up feeling when you return to them.

3.1 The Past of OCaml

Genealogically, OCaml comes from the line of programming languages whose grandfather is Lisp and includes other modern languages such as Clojure, F#, Haskell, and Racket.

OCaml originates from work done by Robin Milner and others at the Edinburgh Laboratory for Computer Science in Scotland. They were working on theorem provers in the late 1970s and early 1980s. Traditionally, theorem provers were implemented in languages such as Lisp. Milner kept running into the problem that the theorem provers would sometimes put incorrect "proofs" (i.e., non-proofs) together and claim that they were valid. So he tried to develop a language that only allowed you to construct valid proofs. ML, which stands for "Meta Language", was the result of that work. The type system of ML was carefully constructed so that you could only construct valid proofs in the language. A theorem prover was then written as a program that constructed a proof. Eventually, this "Classic ML" evolved into a full-fledged programming language.

In the early '80s, there was a schism in the ML community with the French on one side and the British and US on another. The French went on to develop CAML and later Objective CAML (OCaml) while the Brits and Americans developed Standard ML. The two dialects are quite similar. Microsoft introduced its own variant of OCaml called F# in 2005.

Milner received the Turing Award in 1991 in large part for his work on ML. The ACM website for his award includes this praise:

> ML was way ahead of its time. It is built on clean and well-articulated mathematical ideas, teased apart so that they can be studied independently and relatively easily remixed and reused. ML has influenced many practical languages, including Java, Scala, and Microsoft's F#. Indeed, no serious language designer should ignore this example of good design.

3.2 The Present of OCaml

OCaml is a functional programming language. The key linguistic abstraction of functional languages is the mathematical function. A function maps an input to an output; for the same input, it always produces the same output. That is, mathematical functions are *stateless*: they do not maintain any extra information or *state* that persists between usages of the function. Functions are *first-class*: you can use them as input to other functions, and produce functions as output. Expressing everything in terms of functions enables a uniform and simple programming model that is easier to reason about than the procedures and methods found in other families of languages.

Imperative programming languages such as C and Java involve *mutable* state that changes throughout execution. *Commands* specify how to compute by destructively changing that state. Procedures (or methods) can have *side effects* that update state in addition to producing a return value.

The **fantasy of mutability** is that it's easy to reason about: the machine does this, then this, etc.

The **reality of mutability** is that whereas machines are good at complicated manipulation of state, humans are not good at understanding it. The essence of why that's true is that mutability breaks *referential transparency*: the ability to replace an expression with its value without affecting the result of a computation. In math, if $f(x) = y$, then you can substitute y anywhere you see $f(x)$. In imperative languages, you cannot: f might have side effects, so computing $f(x)$ at time t might result in a different value than at time t'.

It's tempting to believe that there's a single state that the machine manipulates, and that the machine does one thing at a time. Computer systems go to great lengths in attempting to provide that illusion. But it's just that: an illusion. In reality, there are many states, spread across threads, cores, processors, and networked computers. And the machine does many things concurrently. Mutability makes reasoning about distributed state and concurrent execution immensely difficult.

Immutability, however, frees the programmer from these concerns. It provides powerful ways to build correct and concurrent programs. OCaml is primarily an immutable language, like most functional languages. It does support imperative programming with mutable state, but we won't use those features until many chapters into the book—in part because we

simply won't need them, and in part to get you to quit "cold turkey" from a dependence you might not have known that you had. This freedom from mutability is one of the biggest changes in perspective that OCaml can give you.

3.2.1 The Features of OCaml

OCaml is a *statically-typed* and *type-safe* programming language. A statically-typed language detects type errors at compile time, so that programs with type errors cannot be executed. A type-safe language limits which kinds of operations can be performed on which kinds of data. In practice, this prevents a lot of silly errors (e.g., treating an integer as a function) and also prevents a lot of security problems: over half of the reported break-ins at the Computer Emergency Response Team (CERT, a US government agency tasked with cybersecurity) were due to buffer overflows, something that's impossible in a type-safe language.

Some functional languages, like Python and Racket, are type-safe but *dynamically typed*. That is, type errors are caught only at run time. Other languages, like C and C++, are statically typed but not type safe. There's no guarantee that a type error won't occur at run time. And still other languages, like Java, use a combination of static and dynamic typing to achieve type safety.

OCaml supports a number of advanced features, some of which you will have encountered before, and some of which are likely to be new:

- **Algebraic datatypes:** You can build sophisticated data structures in OCaml easily, without fussing with pointers and memory management. *Pattern matching*—a feature we'll soon learn about that enables examining the shape of a data structure—makes them even more convenient.
- **Type inference:** You do not have to write type information down everywhere. The compiler automatically figures out most types. This can make the code easier to read and maintain.
- **Parametric polymorphism:** Functions and data structures can be parameterized over types. This is crucial for being able to re-use code.
- **Garbage collection:** Automatic memory management relieves you from the burden of memory allocation and deallocation, a common source of bugs in languages such as C.
- **Modules:** OCaml makes it easy to structure large systems through the use of modules. Modules are used to encapsulate implementations behind interfaces. OCaml goes well beyond the functionality of most languages with modules by providing functions (called *functors*) that manipulate modules.

3.2.2 OCaml in Industry

OCaml and other functional languages are nowhere near as popular as Python, C, or Java. OCaml's real strength lies in language manipulation (i.e., compilers, analyzers, verifiers, provers, etc.). This is not surprising, because OCaml evolved from the domain of theorem proving.

That's not to say that functional languages aren't used in industry. There are many industry projects using OCaml and Haskell, among other languages. Yaron Minsky (Cornell PhD '02) even wrote a paper about using OCaml in the financial industry. It explains how the features of OCaml make it a good choice for quickly building complex software that works.

3.3 Look to Your Future

General-purpose languages come and go. In your life you'll likely learn a handful. Today, it's Python and Java. Yesterday, it was Pascal and Cobol. Before that, it was Fortran and Lisp. Who knows what it will be tomorrow? In this fast-changing field you need to be able to rapidly adapt. A good programmer has to learn the principles behind programming that transcend the specifics of any specific language. There's no better way to get at these principles than to approach programming from a functional perspective. Learning a new language from scratch affords the opportunity to reflect along the way about the difference between *programming* and programming *in a language.*

If after OCaml you want to learn more about functional programming, you'll be well prepared. OCaml does a great job of clarifying and simplifying the essence of functional programming in a way that other languages that blend functional and imperative programming (like Scala) or take functional programming to the extreme (like Haskell) do not.

And even if you never code in OCaml again after learning it, you'll still be better prepared for the future. Advanced features of functional languages have a surprising tendency to predict new features of more mainstream languages. Java brought garbage collection into the mainstream in 1995; Lisp had it in 1958. Java didn't have generics until version 5 in 2004; the ML family had it in 1990. First-class functions and type inference have been incorporated into mainstream languages like Java, C#, and C++ over the last 10 years, long after functional languages introduced them.

News Flash!

Python just announced plans to support pattern matching in February 2021.

3.4 A Brief History of CS 3110

This book is the primary textbook for CS 3110 at Cornell University. The course has existed for over two decades and has always taught functional programming, but it has not always used OCaml.

Once upon a time, there was a course at MIT known as 6.001 *Structure and Interpretation of Computer Programs* (SICP). It had a textbook by the same name, and it used Scheme, a functional programming language. Tim Teitelbaum taught a version of the course at Cornell in Fall 1988, following the book rather closely and using Scheme.

CS 212. Dan Huttenlocher had been a TA for 6.001 at MIT; he later became faculty at Cornell. In Fall 1989, he inaugurated CS 212 Modes of Algorithm Expression. Basing the course on SICP, he infused a more rigorous approach to the material. Huttenlocher continued to develop CS 212 through the mid 1990s, using various homegrown dialects of Scheme.

Other faculty began teaching the course regularly. Ramin Zabih had taken 6.001 as a first-year student at MIT. In Spring 1994, having become faculty at Cornell, he taught CS 212. Dexter Kozen (Cornell PhD 1977) first taught the course in Spring 1996. The earliest surviving online record of the course seems to be Spring 1998, which was taught by Greg Morrisett in Dylan; the name of the course had become Structure and Interpretation of Computer Programs.

By Fall 1999, CS 212 had its own lecture notes. As CS 3110 still does, that instance of CS 212 covered functional programming, the substitution and environment models, some data structures and algorithms, and programming language implementation.

CS 312. At that time, the CS curriculum had two introductory programming courses, CS 211 Computers and Programming, and CS 212. Students took one or the other, similar to how students today take either CS 2110 or CS 2112. Then they took CS 410 Data Structures. The earliest surviving online record of CS 410 seems to be from Spring 1998. It covered many data structures and algorithms not covered by CS 212, including balanced trees and graphs, and it used Java as the programming language.

Depending on which course they took, CS 211 or 212, students were entering upper-level courses with different skill sets. After extensive discussions, the faculty chose to make CS 211 required, to rename CS 212 into CS 312 Data Structures

and Functional Programming, and to make CS 211 a prerequisite for CS 312. At the same time, CS 410 was eliminated from the curriculum and its contents parceled out to CS 312 and CS 482 Introduction to Analysis of Algorithms. Dexter Kozen taught the final offering of CS 410 in Fall 1999.

Greg Morrisett inaugurated the new CS 312 in Spring 2001. He switched from Scheme to Standard ML. Kozen first taught it in Fall 2001, and Andrew Myers in Fall 2002. Myers began to incorporate material on modular programming from another MIT textbook, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design* by Barbara Liskov and John Guttag. Huttenlocher first taught the course in Spring 2006.

CS 3110. In Fall 2008 two big changes came: the language switched to OCaml, and the university switched to four-digit course numbers. CS 312 became CS 3110. Myers, Huttenlocher, Kozen, and Zabih first taught the revised course in Fall 2008, Spring 2009, Fall 2009, and Fall 2010, respectively. Nate Foster first taught the course in Spring 2012; and Bob Constable and Michael George co-taught for the first time in Fall 2013.

Michael Clarkson (Cornell PhD 2010) first taught the course in Fall 2014, after having first TA'd the course as a PhD student back in Spring 2008. He began to revise the presentation of the OCaml programming material to incorporate ideas by Dan Grossman (Cornell PhD 2003) about a principled approach to learning a programming language by decomposing it into syntax, dynamic, and static semantics. Grossman uses that approach in CSE 341 Programming Languages at the University of Washington and in his popular Programming Languages MOOC.

In Fall 2018 the compilation of this textbook began. It synthesizes the work of over two decades of functional programming instruction at Cornell. In the words of the Cornell Evening Song,

> 'Tis an echo from the walls Of our own, our fair Cornell.

3.5 Summary

This book is about becoming a better programmer. Studying functional programming will help with that. The biggest obstacle in our way is the frustration of speaking a new language, particularly letting go of mutable state. But the benefits will be great: a discovery that programming transcends programming in any particular language or family of languages, an exposure to advanced language features, and an appreciation of beauty.

3.5.1 Terms and Concepts

- dynamic typing
- first-class functions
- functional programming languages
- immutability
- Lisp
- ML
- OCaml
- referential transparency
- side effects
- state
- static typing
- type safety

3.5.2 Further Reading

- Introduction to Objective Caml, chapters 1 and 2, a freely available textbook that is recommended for this course
- OCaml from the Very Beginning, chapter 1, a relatively inexpensive PDF textbook that is very gentle and recommended for this course
- A guided tour [of OCaml]: chapter 1 of *Real World OCaml*, a book written by some Cornellians that some students might enjoy reading
- The history of Standard ML: though it focuses on the SML variant of the ML language, it's relevant to OCaml
- The value of values: a lecture by the designer of Clojure (a modern dialect of Lisp) on how the time of imperative programming has passed
- Teach yourself programming in 10 years: an essay by a Director of Research at Google that puts the time required to become an educated programmer into perspective

CHAPTER

FOUR

THE BASICS OF OCAML

This chapter will cover some of the basic features of OCaml. But before we dive in to learning OCaml, let's first talk about a bigger idea: learning languages in general.

One of the secondary goals of this course is not just for you to learn a new programming language, but to improve your skills at learning *how to learn* new languages.

There are five essential components to learning a language: syntax, semantics, idioms, libraries, and tools.

Syntax. By *syntax*, we mean the rules that define what constitutes a textually well-formed program in the language, including the keywords, restrictions on whitespace and formatting, punctuation, operators, etc. One of the more annoying aspects of learning a new language can be that the syntax feels odd compared to languages you already know. But the more languages you learn, the more you'll become used to accepting the syntax of the language for what it is, rather than wishing it were different. (If you want to see some languages with really unusual syntax, take a look at APL, which needs its own extended keyboard, and Whitespace, in which programs consist entirely of spaces, tabs, and newlines.) You need to understand syntax just to be able to speak to the computer at all.

Semantics. By *semantics*, we mean the rules that define the behavior of programs. In other words, semantics is about the meaning of a program—what computation a particular piece of syntax represents. Note that although "semantics" is plural in form, we use it as singular. That's similar to "mathematics" or "physics".

There are two pieces to semantics, the *dynamic* semantics of a language and the *static* semantics of a language. The dynamic semantics define the run-time behavior of a program as it is executed or evaluated. The static semantics define the compile-time checking that is done to ensure that a program is legal, beyond any syntactic requirements. The most important kind of static semantics is probably *type checking*: the rules that define whether a program is well typed or not. Learning the semantics of a new language is usually the real challenge, even though the syntax might be the first hurdle you have to overcome. You need to understand semantics to say what you mean to the computer, and you need to say what you mean so that your program performs the right computation.

Idioms. By *idioms*, we mean the common approaches to using language features to express computations. Given that you might express one computation in many ways inside a language, which one do you choose? Some will be more natural than others. Programmers who are fluent in the language will prefer certain modes of expression over others. We could think of this in terms of using the dominant paradigms in the language effectively, whether they are imperative, functional, object oriented, etc. You need to understand idioms to say what you mean not just to the computer, but to other programmers. When you write code idiomatically, other programmers will understand your code better.

Libraries. *Libraries* are bundles of code that have already been written for you and can make you a more productive programmer, since you won't have to write the code yourself. (It's been said that laziness is a virtue for a programmer.) Part of learning a new language is discovering what libraries are available and how to make use of them. A language usually provides a *standard library* that gives you access to a core set of functionality, much of which you would be unable to code up in the language yourself, such as file I/O.

Tools. At the very least any language implementation provides either a compiler or interpreter as a tool for interacting with the computer using the language. But there are other kinds of tools: debuggers; integrated development environments (IDE); and analysis tools for things like performance, memory usage, and correctness. Learning to use tools that are associated with a language can also make you a more productive programmer. Sometimes it's easy to confuse the tool

itself for the language; if you've only ever used Eclipse and Java together for example, it might not be apparent that Eclipse is an IDE that works with many languages, and that Java can be used without Eclipse.

When it comes to learning OCaml in this book, our focus is primarily on semantics and idioms. We'll have to learn syntax along the way, of course, but it's not the interesting part of our studies. We'll get some exposure to the OCaml standard library and a couple other libraries, notably OUnit (a unit testing framework similar to JUnit, HUnit, etc.). Besides the OCaml compiler and build system, the main tool we'll use is the toplevel, which provides the ability to interactively experiment with code.

4.1 The OCaml Toplevel

The *toplevel* is like a calculator or command-line interface to OCaml. It's similar to JShell for Java, or the interactive Python interpreter. The toplevel is handy for trying out small pieces of code without going to the trouble of launching the OCaml compiler. But don't get too reliant on it, because creating, compiling, and testing large programs will require more powerful tools. Some other languages would call the toplevel a *REPL*, which stands for read-eval-print-loop: it reads programmer input, evaluates it, prints the result, and then repeats.

In a terminal window, type `utop` to start the toplevel. Press Control-D to exit the toplevel. You can also enter `#quit;;` and press return. Note that you must type the `#` there: it is in addition to the `#` prompt you already see.

4.1.1 Types and values

You can enter expressions into the OCaml toplevel. End an expression with a double semi-colon `;;` and press the return key. OCaml will then evaluate the expression, tell you the resulting value, and the value's type. For example:

```
# 42;;
- : int = 42
```

Let's dissect that response from utop, reading right to left:

- `42` is the value.
- `int` is the type of the value.
- The value was not given a name, hence the symbol `-`.

That utop interaction was "hardcoded" as part of this book. We had to type in all the characters: the `#`, the `-`, etc. But the infrastructure used to write this book actually enables us to write code that is evaluated by OCaml at the time the book is translated into HTML or PDF. From now on, that's usually what we will do. It looks like this:

```
42
```

The first code block with the `42` in it is the code we asked OCaml to run. If you want to enter that into utop, you can copy and paste it. There's an icon in the top right of the block to do that easily. Just remember to add the double semicolon at the end. The second code block, which is indented a little, is the output from OCaml as the book was being translated.

Tip: If you're viewing this in a web browser, look to the top right for a download icon. Choose the `.md` option, and you'll see the original MyST Markdown source code for this page of the book. You'll see that the output from the second example above is not actually present in the source code. That's good! It means that the output stays consistent with whatever current version of the OCaml compiler we use to build the book. It also means that any compilation errors can be detected as part of building the book, instead of lurking for you, dear reader, to find them.

You can bind values to names with a `let` definition, as follows:

```
let x = 42
```

Again, let's dissect that response, this time reading left to right:

- A value was bound to a name, hence the `val` keyword.
- `x` is the name to which the value was bound.
- `int` is the type of the value.
- `42` is the value.

You can pronounce the entire output as "`x` has type `int` and equals `42`."

4.1.2 Functions

A function can be defined at the toplevel using syntax like this:

```
let increment x = x + 1
```

Let's dissect that response:

- `increment` is the identifier to which the value was bound.
- `int -> int` is the type of the value. This is the type of functions that take an `int` as input and produce an `int` as output. Think of the arrow `->` as a kind of visual metaphor for the transformation of one value into another value—which is what functions do.
- The value is a function, which the toplevel chooses not to print (because it has now been compiled and has a representation in memory that isn't easily amenable to pretty printing). Instead, the toplevel prints `<fun>`, which is just a placeholder.

Note: `<fun>` itself is not a value. It just indicates an unprintable function value.

You can "call" functions with syntax like this:

```
increment 0
```

```
increment(21)
```

```
increment (increment 5)
```

But in OCaml the usual vocabulary is that we "apply" the function rather than "call" it.

Note how OCaml is flexible about whether you write the parentheses or not, and whether you write whitespace or not. One of the challenges of first learning OCaml can be figuring out when parentheses are actually required. So if you find yourself having problems with syntax errors, one strategy is to try adding some parentheses. The preferred style, though, is usually to omit parentheses when they are not needed. So, `increment 21` is better than `increment(21)`.

4.1.3 Loading code in the toplevel

In addition to allowing you to define functions, the toplevel will also accept *directives* that are not OCaml code but rather tell the toplevel itself to do something. All directives begin with the `#` character. Perhaps the most common directive is `#use`, which loads all the code from a file into the toplevel, just as if you had typed the code from that file into the toplevel.

For example, suppose you create a file named `mycode.ml`. In that file put the following code:

```
let inc x = x + 1
```

Start the toplevel. Try entering the following expression, and observe the error:

```
inc 3
```

The error occurs because the toplevel does not yet know anything about a function named `inc`. Now issue the following directive to the toplevel:

```
# #use "mycode.ml";;
```

Note that the first `#` character above indicates the toplevel prompt to you. The second `#` character is one that you type to tell the toplevel that you are issuing a directive. Without that character, the toplevel would think that you are trying to apply a function named `use`.

Now try again:

```
inc 3
```

4.1.4 Workflow in the toplevel

The best workflow when using the toplevel with code stored in files is:

- Edit the code in the file.
- Load the code in the toplevel with `#use`.
- Interactively test the code.
- Exit the toplevel. **Warning:** do not skip this step.

Tip: Suppose you wanted to fix a bug in your code. It's tempting to not exit the toplevel, edit the file, and re-issue the `#use` directive into the same toplevel session. Resist that temptation. The "stale code" that was loaded from an earlier `#use` directive in the same session can cause surprising things to happen—surprising when you're first learning the language, anyway. So **always exit the toplevel before re-using a file.**

4.2 Compiling OCaml Programs

Using OCaml as a kind of interactive calculator can be fun, but we won't get very far with writing large programs that way. We instead need to store code in files and compile them.

4.2.1 Storing code in files

Open a terminal, create a new directory, and open VS Code in that directory. For example, you could use the following commands:

```
$ mkdir hello-world
$ code hello-world
```

Warning: Do not use the root of your Unix home directory as the place you store the file. The build system we are going to use very soon, dune, might not work right in the root of your home directory. Instead, you need to use a subdirectory of your home directory.

Use VS Code to create a new file named `hello.ml`. Enter the following code into the file:

```
let _ = print_endline "Hello world!"
```

Note: There is no double semicolon `;;` at the end of that line of code. The double semicolon is intended for interactive sessions in the toplevel, so that the toplevel knows you are done entering a piece of code. There's usually no reason to write it in a .ml file.

The `let _ =` above means that we don't care to give a name (hence the "blank" or underscore) to code on the right-hand side of the =.

Save the file and return to the command line. Compile the code:

```
$ ocamlc -o hello.byte hello.ml
```

The compiler is named `ocamlc`. The `-o hello.byte` option says to name the output executable `hello.byte`. The executable contains compiled OCaml bytecode. In addition, two other files are produced, `hello.cmi` and `hello.cmo`. We don't need to be concerned with those files for now. Run the executable:

```
$ ./hello.byte
```

It should print `Hello world!` and terminate.

Now change the string that is printed to something of your choice. Save the file, recompile, and rerun. Try making the code print multiple lines.

This edit-compile-run cycle between the editor and the command line is something that might feel unfamiliar if you're used to working inside IDEs like Eclipse. Don't worry; it will soon become second nature.

Now let's clean up all those generated files:

```
$ rm hello.byte hello.cmi hello.cmo
```

4.2.2 What about Main?

Unlike C or Java, OCaml programs do not need to have a special function named `main` that is invoked to start the program. The usual idiom is just to have the very last definition in a file serve as the main function that kicks off whatever computation is to be done.

4.2.3 Dune

In larger projects, we don't want to run the compiler or clean up manually. Instead, we want to use a *build system* to automatically find and link in libraries. OCaml has a legacy build system called ocamlbuild, and a newer build system called Dune. Similar systems include `make`, which has long been used in the Unix world for C and other languages; and Gradle, Maven, and Ant, which are used with Java.

A Dune *project* is a directory (and its subdirectories) that contain OCaml code you want to compile. The *root* of a project is the highest directory in its hierarchy. A project might rely on external *packages* providing additional code that is already compiled. Usually, packages are installed with OPAM, the OCaml Package Manager.

Each directory in your project can contain a file named `dune`. That file describes to Dune how you want the code in that directory (and subdirectories) to be compiled. Dune files use a functional-programming syntax descended from LISP called *s-expressions*, in which parentheses are used to show nested data that form a tree, much like HTML tags do. The syntax of Dune files is documented in the Dune manual.

Here is a small example of how to use Dune. In the same directory as `hello.ml`, create a file named `dune` and put the following in it:

```
(executable
 (name hello))
```

That declares an *executable* (a program that can be executed) whose main file is `hello.ml`. Then run this command from the terminal:

```
$ dune build hello.exe
```

Note that the `.exe` extension is used on all platforms by dune, not just on Windows. That causes dune to build a *native* executable rather than a bytecode executable. The first time you run that command, dune will automatically create another file `dune-project` that marks the root directory of the project. You don't need to do anything more with that file.

Dune will create a directory `_build` and compile our program inside it. That's one benefit of the build system over directly running the compiler: instead of polluting your source directory with a bunch of generated files, they get cleanly created in a separate directory. Inside `_build` there are many files that get created by dune. Our executable is buried rather deep:

```
$ _build/default/hello.exe
Hello world!
```

But dune provides a shortcut to having to remember and type all of that. To build and execute the program in one step, we can simply run:

```
$ dune exec ./hello.exe
Hello world!
```

Finally, to clean up all the compiled code we just run:

```
$ dune clean
```

That removes the `_build` directory, leaving just your source code.

Tip: When dune compiles your program, it caches a copy of your source files in `_build/default`. If you ever accidentally make a mistake that results in loss of a source file, you might be able to recover it from inside `_build`. Of course, using source control like git is also advisable.

4.3 Expressions

The primary piece of OCaml syntax is the *expression*. Just like programs in imperative languages are primarily built out of *commands*, programs in functional languages are primarily built out of expressions. Examples of expressions include `2+2` and `increment 21`.

The OCaml manual has a complete definition of all the expressions in the language. Though that page starts with a rather cryptic overview, if you scroll down, you'll come to some English explanations. Don't worry about studying that page now; just know that it's available for reference.

The primary task of computation in a functional language is to *evaluate* an expression to a *value*. A value is an expression for which there is no computation remaining to be performed. So, all values are expressions, but not all expressions are values. Examples of values include `2`, `true`, and `"yay!"`.

The OCaml manual also has a definition of all the values, though again, that page is mostly useful for reference rather than study.

Sometimes an expression might fail to evaluate to a value. There are two reasons that might happen:

1. Evaluation of the expression raises an exception.
2. Evaluation of the expression never terminates (e.g., it enters an "infinite loop").

4.3.1 Primitive Types and Values

The *primitive* types are the built-in and most basic types: integers, floating-point numbers, characters, strings, and booleans. They will be recognizable as similar to primitive types from other programming languages.

Type `int`: Integers. OCaml integers are written as usual: `1`, `2`, etc. The usual operators are available: `+`, `-`, `*`, `/`, and `mod`. The latter two are integer division and modulus:

```
65 / 60
```

```
65 mod 60
```

```
65 / 0
```

OCaml integers range from -2^{62} to $2^{62} - 1$ on modern platforms. They are implemented with 64-bit machine *words*, which is the size of a register on 64-bit processor. But one of those bits is "stolen" by the OCaml implementation, leading to a 63-bit representation. That bit is used at run time to distinguish integers from pointers. For applications that need true 64-bit integers, there is an `Int64` module in the standard library. And for applications that need arbitrary-precision integers, there is a separate `Zarith` library. But for most purposes, the built-in `int` type suffices and offers the best performance.

Type `float`: Floating-point numbers. OCaml floats are IEEE 754 double-precision floating-point numbers. Syntactically, they must always contain a dot—for example, `3.14` or `3.0` or even `3.`. The last is a `float`; if you write it as `3`, it is instead an `int`:

```
3.
```

```
3
```

OCaml deliberately does not support operator overloading, Arithmetic operations on floats are written with a dot after them. For example, floating-point multiplication is written `*.` not `*`:

```
3.14 *. 2.
```

```
3.14 * 2.
```

OCaml will not automatically convert between `int` and `float`. If you want to convert, there are two built-in functions for that purpose: `int_of_float` and `float_of_int`.

```
3.14 *. (float_of_int 2)
```

As in any language, the floating-point representation is approximate. That can lead to rounding errors:

```
0.1 +. 0.2
```

The same behavior can be observed in Python and Java, too. If you haven't encountered this phenomenon before, here's a basic guide to floating-point representation that you might enjoy reading.

Type `bool`: Booleans. The boolean values are written `true` and `false`. The usual short-circuit conjunction `&&` and disjunction `||` operators are available.

Type `char`: Characters. Characters are written with single quotes, such as `'a'`, `'b`, and `'c'`. They are represented as bytes —that is, 8-bit integers— in the ISO 9958-1 "Latin-1" encoding. The first half of the characters in that range are the standard ASCII characters. You can convert characters to and from integers with `char_of_int` and `int_of_char`.

Type `string`: Strings. Strings are sequences of characters. They are written with double quotes, such as `"abc"`. The string concatenation operator is `^`:

```
"abc" ^ "def"
```

Object-oriented languages often provide an overridable method for converting objects to strings, such as `toString()` in Java or `__str__()` in Python. But most OCaml values are not objects, so another means is required to convert to strings. For three of the primitive types, there are built-in functions: `string_of_int`, `string_of_float`, `string_of_bool`. Strangely, there is no `string_of_char`, but the library function `String.make` can be used to accomplish the same goal.

```
string_of_int 42
```

```
String.make 1 'z'
```

Likewise, for the same three primitive types, there are built-in functions to convert from a string if possible: `int_of_string`, `float_of_string`, and `bool_of_string`.

```
int_of_string "123"
```

```
int_of_string "not an int"
```

There is no `char_of_string`, but the individual characters of a string can be accessed by a 0-based index. The indexing operator is written with a dot and square brackets:

```
"abc".[0]
```

```
"abc".[1]
```

```
"abc".[3]
```

4.3.2 More Operators

We've covered most of the built-in operators above, but there are a few more that you can see in the OCaml manual.

There are two equality operators in OCaml, = and ==, with corresponding inequality operators <> and `!=`. Operators = and <> examine *structural* equality whereas == and `!=` examine *physical* equality. Until we've studied the imperative features of OCaml, the difference between them will be tricky to explain. See the documentation of `Stdlib.(==)` if you're curious now.

Important: Start training yourself now to use = and not to use ==. This will be difficult if you're coming from a language like Java where == is the usual equality operator.

4.3.3 Assertions

The expression `assert e` evaluates `e`. If the result is `true`, nothing more happens, and the entire expression evaluates to a special value called *unit*. The unit value is written `()` and its type is `unit`. But if the result is `false`, an exception is raised.

4.3.4 If Expressions

The expression `if e1 then e2 else e3` evaluates to `e2` if `e1` evaluates to `true`, and to `e3` otherwise. We call `e1` the *guard* of the `if` expression.

```
if 3 + 5 > 2 then "yay!" else "boo!"
```

Unlike `if-then-else` *statements* that you may have used in imperative languages, `if-then-else` *expressions* in OCaml are just like any other expression; they can be put anywhere an expression can go. That makes them similar to the ternary operator `? :` that you might have used in other languages.

```
4 + (if 'a' = 'b' then 1 else 2)
```

`If` expressions can be nested in a pleasant way:

```
if e1 then e2
else if e3 then e4
else if e5 then e6
...
else en
```

You should regard the final `else` as mandatory, regardless of whether you are writing a single `if` expression or a highly nested `if` expression. If you omit it you'll likely get an error message that, for now, is inscrutable:

```
if 2 > 3 then 5
```

Syntax. The syntax of an `if` expression:

```
if e1 then e2 else e3
```

The letter `e` is used here to represent any other OCaml expression; it's an example of a *syntactic variable* aka *metavariable*, which is not actually a variable in the OCaml language itself, but instead a name for a certain syntactic construct. The numbers after the letter `e` are being used to distinguish the three different occurrences of it.

Dynamic semantics. The dynamic semantics of an `if` expression:

- If `e1` evaluates to `true`, and if `e2` evaluates to a value `v`, then `if e1 then e2 else e3` evaluates to `v`
- If `e1` evaluates to `false`, and if `e3` evaluates to a value `v`, then `if e1 then e2 else e3` evaluates to `v`.

We call these *evaluation rules*: they define how to evaluate expressions. Note how it takes two rules to describe the evaluation of an `if` expression, one for when the guard is true, and one for when the guard is false. The letter `v` is used here to represent any OCaml value; it's another example of a metavariable. Later we will develop a more mathematical way of expressing dynamic semantics, but for now we'll stick with this more informal style of explanation.

Static semantics. The static semantics of an `if` expression:

- If `e1` has type `bool` and `e2` has type `t` and `e3` has type `t` then `if e1 then e2 else e3` has type `t`

We call this a *typing rule*: it describes how to type check an expression. Note how it only takes one rule to describe the type checking of an `if` expression. At compile time, when type checking is done, it makes no difference whether the guard is true or false; in fact, there's no way for the compiler to know what value the guard will have at run time. The letter `t` here is used to represent any OCaml type; the OCaml manual also has definition of all types (which curiously does not name the base types of the language like `int` and `bool`).

We're going to be writing "has type" a lot, so let's introduce a more compact notation for it. Whenever we would write "`e` has type `t`", let's instead write `e : t`. The colon is pronounced "has type". This usage of colon is consistent with how the toplevel responds after it evaluates an expression that you enter:

```
let x = 42
```

In the above example, variable `x` has type `int`, which is what the colon indicates.

4.3.5 Let Expressions

In our use of the word `let` thus far, we've been making definitions in the toplevel and in `.ml` files. For example,

```
let x = 42;;
```

defines `x` to be 42, after which we can use `x` in future definitions at the toplevel. We'll call this use of `let` a *let definition*.

There's another use of `let` which is as an expression:

```
let x = 42 in x + 1
```

Here we're *binding* a value to the name `x` then using that binding inside another expression, `x+1`. We'll call this use of `let` a *let expression*. Since it's an expression it evaluates to a value. That's different than definitions, which themselves do not evaluate to any value. You can see that if you try putting a let definition in place of where an expression is expected:

```
(let x = 42) + 1
```

Syntactically, a `let` definition is not permitted on the left-hand side of the + operator, because a value is needed there, and definitions do not evaluate to values. On the other hand, a `let` expression would work fine:

```
(let x = 42 in x) + 1
```

Another way to understand let definitions at the toplevel is that they are like let expression where we just haven't provided the body expression yet. Implicitly, that body expression is whatever else we type in the future. For example,

```
# let a = "big";;
# let b = "red";;
# let c = a ^ b;;
# ...
```

is understand by OCaml in the same way as

```
let a = "big" in
let b = "red" in
let c = a ^ b in
...
```

That latter series of `let` bindings is idiomatically how several variables can be bound inside a given block of code.

Syntax.

```
let x = e1 in e2
```

As usual, `x` is an identifier. These identifiers must begin with lower-case, not upper, and idiomatically are written with `snake_case` not `camelCase`. We call `e1` the *binding expression*, because it's what's being bound to `x`; and we call `e2` the *body expression*, because that's the body of code in which the binding will be in scope.

Dynamic semantics.

To evaluate `let x = e1 in e2`:

- Evaluate `e1` to a value `v1`.
- Substitute `v1` for `x` in `e2`, yielding a new expression `e2'`.
- Evaluate `e2'` to a value `v2`.
- The result of evaluating the let expression is `v2`.

Here's an example:

```
    let x = 1 + 4 in x * 3
-->   (evaluate e1 to a value v1)
    let x = 5 in x * 3
-->   (substitute v1 for x in e2, yielding e2')
    5 * 3
-->   (evaluate e2' to v2)
    15
      (result of evaluation is v2)
```

Static semantics.

- If `e1 : t1` and if under the assumption that `x : t1` it holds that `e2 : t2`, then `(let x = e1 in e2) : t2`.

We use the parentheses above just for clarity. As usual, the compiler's type inferencer determines what the type of the variable is, or the programmer could explicitly annotate it with this syntax:

```
let x : t = e1 in e2
```

4.3.6 Scope

`Let` bindings are in effect only in the block of code in which they occur. This is exactly what you're used to from nearly any modern programming language. For example:

```
let x = 42 in
  (* y is not meaningful here *)
  x + (let y = "3110" in
         (* y is meaningful here *)
         int_of_string y)
```

The *scope* of a variable is where its name is meaningful. Variable `y` is in scope only inside of the `let` expression that binds it above.

It's possible to have overlapping bindings of the same name. For example:

```
let x = 5 in
  ((let x = 6 in x) + x)
```

But this is darn confusing, and for that reason, it is strongly discouraged style—much like ambiguous pronouns are discouraged in natural language. Nonetheless, let's consider what that code means.

To what value does that code evaluate? The answer comes down to how `x` is replaced by a value each time it occurs. Here are a few possibilities for such *substitution*:

```
(* possibility 1 *)
let x = 5 in
  ((let x = 6 in 6) + 5)

(* possibility 2 *)
let x = 5 in
  ((let x = 6 in 5) + 5)

(* possibility 3 *)
let x = 5 in
  ((let x = 6 in 6) + 6)
```

The first one is what nearly any reasonable language would do. And most likely it's what you would guess But, **why?**

The answer is something we'll call the *Principle of Name Irrelevance*: the name of a variable shouldn't intrinsically matter. You're used to this from math. For example, the following two functions are the same:

$$f(x) = x^2$$
$$f(y) = y^2$$

It doesn't intrinsically matter whether we call the argument to the function x or y; either way, it's still the squaring function. Therefore, in programs, these two functions should be identical:

```
let f x = x * x
let f y = y * y
```

This principle is more commonly known as *alpha equivalence*: the two functions are equivalent up to renaming of variables, which is also called *alpha conversion* for historical reasons that are unimportant here.

According to the Principle of Name Irrelevance, these two expressions should be identical:

```
let x = 6 in x
let y = 6 in y
```

Therefore, the following two expressions, which have the above expressions embedded in them, should also be identical:

```
let x = 5 in (let x = 6 in x) + x
let x = 5 in (let y = 6 in y) + x
```

But for those to be identical, we **must** choose the first of the three possibilities above. It is the only one that makes the name of the variable be irrelevant.

There is a term commonly used for this phenomenon: a new binding of a variable *shadows* any old binding of the variable name. Metaphorically, it's as if the new binding temporarily casts a shadow over the old binding. But eventually the old binding could reappear as the shadow recedes.

Shadowing is not mutable assignment. For example, both of the following expressions evaluate to 11:

```
let x = 5 in ((let x = 6 in x) + x)
let x = 5 in (x + (let x = 6 in x))
```

Likewise, the following utop transcript is not mutable assignment, though at first it could seem like it is:

```
# let x = 42;;
val x : int = 42
# let x = 22;;
val x : int = 22
```

Recall that every `let` definition in the toplevel is effectively a nested `let` expression. So the above is effectively the following:

```
let x = 42 in
  let x = 22 in
    ... (* whatever else is typed in the toplevel *)
```

The right way to think about this is that the second `let` binds an entirely new variable that just happens to have the same name as the first `let`.

Here is another utop transcript that is well worth studying:

```
# let x = 42;;
val x : int = 42
# let f y = x + y;;
val f : int -> int = <fun>
# f 0;;
: int = 42
# let x = 22;;
val x : int = 22
# f 0;;
- : int = 42   (* x did not mutate! *)
```

To summarize, each let definition binds an entirely new variable. If that new variable happens to have the same name as an old variable, the new variable temporarily shadows the old one. But the old variable is still around, and its value is

immutable: it never, ever changes. So even though `let` expressions might superficially look like assignment statements from imperative languages, they are actually quite different.

4.3.7 Type Annotations

OCaml automatically infers the type of every expression, with no need for the programmer to write it manually. Nonetheless, it can sometimes be useful to manually specify the desired type of an expression. A *type annotation* does that:

```
(5 : int)
```

An incorrect annotation will produce a compile-time error:

```
(5 : float)
```

And that example shows why you might use manual type annotations during debugging. Perhaps you had forgotten that `5` cannot be treated as a `float`, and you tried to write:

```
5 +. 1.1
```

You might try manually specifying that `5` was supposed to be a `float`:

```
(5 : float) +. 1.1
```

It's clear that the type annotation has failed. Although that might seem silly for this tiny program, you might find this technique to be effective as programs get larger.

Important: Type annotations are **not** *type casts*, such as might be found in C or Java. They do not indicate a conversion from one type to another. Rather they indicate a check that the expression really does have the given type.

Syntax. The syntax of a type annotation:

```
(e : t)
```

Note that the parentheses are required.

Dynamic semantics. There is no run-time meaning for a type annotation. It goes away during compilation, because it indicates a compile-time check. There is no run-time conversion.

Static semantics. If `e` has type `t` then `(e : t)` has type `t`.

4.4 Functions

Since OCaml is a functional language, there's a lot to cover about functions. Let's get started.

Important: Methods and functions are not the same idea. A method is a component of an object, and it implicitly has a receiver that is usually accessed with a keyword like `this` or `self`. OCaml functions are not methods: they are not components of objects, and they do not have a receiver.

Some might say that all methods are functions, but not all functions are methods. Some might even quibble with that, making a distinction between functions and procedures. The latter would be functions that do not return any meaningful value, such as a `void` return type in Java or `None` return value in Python.

So if you're coming from an object-oriented background, be careful about the terminology. **Everything here is a strictly a function, not a method.**

4.4.1 Function Definitions

The following code

```
let x = 42
```

has an expression in it (`42`) but is not itself an expression. Rather, it is a *definition*. Definitions bind values to names, in this case the value `42` being bound to the name `x`. The OCaml manual describes definitions (see the third major grouping titled "*definition*" on that page), but that manual page is again primarily for reference not for study. Definitions are not expressions, nor are expressions definitions—they are distinct syntactic classes.

For now, let's focus on one particular kind of definition, a *function definition*. Non-recursive functions are defined like this:

```
let f x = ...
```

Recursive functions are defined like this:

```
let rec f x = ...
```

The difference is just the `rec` keyword. It's probably a bit surprising that you explicitly have to add a keyword to make a function recursive, because most languages assume by default that they are. OCaml doesn't make that assumption, though. (Nor does the Scheme family of languages.)

One of the best known recursive functions is the factorial function. In OCaml, it can be written as follows:

```
(** [fact n] is [n]!.
    Requires: [n >= 0]. *)
let rec fact n = if n = 0 then 1 else n * fact (n - 1)
```

We provided a specification comment above the function to document the precondition (`Requires`) and postcondition (`is`) of the function.

Note that, as in many languages, OCaml integers are not the "mathematical" integers but are limited to a fixed number of bits. The manual specifies that (signed) integers are at least 31 bits, but they could be wider. As architectures have grown, so has that size. In current implementations, OCaml integers are 63 bits. So if you test on large enough inputs, you might begin to see strange results. The problem is machine arithmetic, not OCaml. (For interested readers: why 31 or 63 instead of 32 or 64? The OCaml garbage collector needs to distinguish between integers and pointers. The runtime representation of these therefore steals one bit to flag whether a word is an integer or a pointer.)

Here's another recursive function:

```
(** [pow x y] is [x] to the power of [y].
     Requires: [y >= 0]. *)
let rec pow x y = if y = 0 then 1 else x * pow x (y - 1)
```

Note how we didn't have to write any types in either of our functions: the OCaml compiler infers them for us automatically. The compiler solves this *type inference* problem algorithmically, but we could do it ourselves, too. It's like a mystery that can be solved by our mental power of deduction:

- Since the `if` expression can return `1` in the `then` branch, we know by the typing rule for `if` that the entire `if` expression has type `int`.

- Since the `if` expression has type `int`, the function's return type must be `int`.
- Since `y` is compared to `0` with the equality operator, `y` must be an `int`.
- Since `x` is multiplied with another expression using the `*` operator, `x` must be an `int`.

If we wanted to write down the types for some reason, we could do that:

```
let rec pow (x : int) (y : int) : int = ...
```

The parentheses are mandatory when we write the *type annotations* for `x` and `y`. We will generally leave out these annotations, because it's simpler to let the compiler infer them. There are other times when you'll want to explicitly write down types. One particularly useful time is when you get a type error from the compiler that you don't understand. Explicitly annotating the types can help with debugging such an error message.

Syntax. The syntax for function definitions:

```
let rec f x1 x2 ... xn = e
```

The `f` is a metavariable indicating an identifier being used as a function name. These identifiers must begin with a lowercase letter. The remaining rules for lowercase identifiers can be found in the manual. The names `x1` through `xn` are metavariables indicating argument identifiers. These follow the same rules as function identifiers. The keyword `rec` is required if `f` is to be a recursive function; otherwise it may be omitted.

Note that syntax for function definitions is actually simplified compared to what OCaml really allows. We will learn more about some augmented syntax for function definition in the next couple weeks. But for now, this simplified version will help us focus.

Mutually recursive functions can be defined with the `and` keyword:

```
let rec f x1 ... xn = e1
and g y1 ... yn = e2
```

For example:

```
(** [even n] is whether [n] is even.
    Requires: [n >= 0]. *)
let rec even n =
  n = 0 || odd (n - 1)

(** [odd n] is whether [n] is odd.
    Requires: [n >= 0]. *)
and odd n =
  n <> 0 && even (n - 1);;
```

The syntax for function types is:

```
t -> u
t1 -> t2 -> u
t1 -> ... -> tn -> u
```

The `t` and `u` are metavariables indicating types. Type `t -> u` is the type of a function that takes an input of type `t` and returns an output of type `u`. We can think of `t1 -> t2 -> u` as the type of a function that takes two inputs, the first of type `t1` and the second of type `t2`, and returns an output of type `u`. Likewise for a function that takes `n` arguments.

Dynamic semantics. There is no dynamic semantics of function definitions. There is nothing to be evaluated. OCaml just records that the name `f` is bound to a function with the given arguments `x1..xn` and the given body `e`. Only later, when the function is applied, will there be some evaluation to do.

Static semantics. The static semantics of function definitions:

- For non-recursive functions: if by assuming that `x1 : t1` and `x2 : t2` and … and `xn : tn`, we can conclude that `e : u`, then `f : t1 -> t2 -> ... -> tn -> u`.
- For recursive functions: if by assuming that `x1 : t1` and `x2 : t2` and … and `xn : tn` and `f : t1 -> t2 -> ... -> tn -> u`, we can conclude that `e : u`, then `f : t1 -> t2 -> ... -> tn -> u`.

Note how the type checking rule for recursive functions assumes that the function identifier `f` has a particular type, then checks to see whether the body of the function is well-typed under that assumption. This is because `f` is in scope inside the function body itself (just like the arguments are in scope).

4.4.2 Anonymous Functions

We already know that we can have values that are not bound to names. The integer `42`, for example, can be entered at the toplevel without giving it a name:

```
42
```

Or we can bind it to a name:

```
let x = 42
```

Similarly, OCaml functions do not have to have names; they may be *anonymous*. For example, here is an anonymous function that increments its input: `fun x -> x + 1`. Here, `fun` is a keyword indicating an anonymous function, `x` is the argument, and `->` separates the argument from the body.

We now have two ways we could write an increment function:

```
let inc x = x + 1
let inc = fun x -> x + 1
```

They are syntactically different but semantically equivalent. That is, even though they involve different keywords and put some identifiers in different places, they mean the same thing.

Anonymous functions are also called *lambda expressions*, a term that comes from the *lambda calculus*, which is a mathematical model of computation in the same sense that Turing machines are a model of computation. In the lambda calculus, `fun x -> e` would be written $\lambda x.e$. The λ denotes an anonymous function.

It might seem a little mysterious right now why we would want functions that have no names. Don't worry; we'll see good uses for them later in the course, especially when we study so-called "higher-order programming". In particular, we will often create anonymous functions and pass them as input to other functions.

Syntax.

```
fun x1 ... xn -> e
```

Static semantics.

- If by assuming that `x1 : t1` and `x2 : t2` and … and `xn : tn`, we can conclude that `e : u`, then `fun x1 ... xn -> e : t1 -> t2 -> ... -> tn -> u`.

Dynamic semantics. An anonymous function is already a value. There is no computation to be performed.

4.4.3 Function Application

Here we cover a somewhat simplified syntax of function application compared to what OCaml actually allows.

Syntax.

```
e0 e1 e2 ... en
```

The first expression `e0` is the function, and it is applied to arguments `e1` through `en`. Note that parentheses are not required around the arguments to indicate function application, as they are in languages in the C family, including Java.

Static semantics.

- If `e0 : t1 -> ... -> tn -> u` and `e1 : t1` and ... and `en : tn` then `e0 e1 ... en : u`.

Dynamic semantics.

To evaluate `e0 e1 ... en`:

1. Evaluate `e0` to a function. Also evaluate the argument expressions `e1` through `en` to values `v1` through `vn`.

   For `e0`, the result might be an anonymous function `fun x1 ... xn -> e` or a name `f`. In the latter case, we need to find the definition of `f`, which we can assume to be of the form `let rec f x1 ... xn = e`. Either way, we now know the argument names `x1` through `xn` and the body `e`.

2. Substitute each value `vi` for the corresponding argument name `xi` in the body `e` of the function. That substitution results in a new expression `e'`.

3. Evaluate `e'` to a value `v`, which is the result of evaluating `e0 e1 ... en`.

If you compare these evaluation rules to the rules for `let` expressions, you will notice they both involve substitution. This is not an accident. In fact, anywhere `let x = e1 in e2` appears in a program, we could replace it with `(fun x -> e2) e1`. They are syntactically different but semantically equivalent. In essence, `let` expressions are just syntactic sugar for anonymous function application.

4.4.4 Pipeline

There is a built-in infix operator in OCaml for function application called the *pipeline* operator, written `|>`. Imagine that as depicting a triangle pointing to the right. The metaphor is that values are sent through the pipeline from left to right. For example, suppose we have the increment function `inc` from above as well as a function `square` that squares its input:

```
let square x = x * x
```

Here are two equivalent ways of squaring `6`:

```
square (inc 5);;
5 |> inc |> square;;
```

The latter uses the pipeline operator to send `5` through the `inc` function, then send the result of that through the `square` function. This is a nice, idiomatic way of expressing the computation in OCaml. The former way is arguably not as elegant: it involves writing extra parentheses and requires the reader's eyes to jump around, rather than move linearly from left to right. The latter way scales up nicely when the number of functions being applied grows, where as the former way requires more and more parentheses:

```
5 |> inc |> square |> inc |> inc |> square;;
square (inc (inc (square (inc 5))));;
```

It might feel weird at first, but try using the pipeline operator in your own code the next time you find yourself writing a big chain of function applications.

Since `e1 |> e2` is just another way of writing `e2 e1`, we don't need to state the semantics for `|>`: it's just the same as function application. These two programs are another example of expressions that are syntactically different but semantically equivalent.

4.4.5 Polymorphic Functions

The *identity* function is the function that simply returns its input:

```
let id x = x
```

Or equivalently as an anonymous function:

```
let id = fun x -> x
```

The `'a` is a *type variable*: it stands for an unknown type, just like a regular variable stands for an unknown value. Type variables always begin with a single quote. Commonly used type variables include `'a`, `'b`, and `'c`, which OCaml programmers typically pronounce in Greek: alpha, beta, and gamma.

We can apply the identity function to any type of value we like:

```
id 42;;
id true;;
id "bigred";;
```

Because you can apply `id` to many types of values, it is a *polymorphic* function: it can be applied to many (*poly*) forms (*morph*).

With manual type annotations, it's possible to give a more restrictive type to a polymorphic function than the type the compiler automatically infers. For example:

```
let id_int (x : int) : int = x
```

That's the same function as `id`, except for the two manual type annotations. Because of those, we cannot apply `id_int` to a bool like we did `id`:

```
id_int true
```

Another way of writing `id_int` would be in terms of `id`:

```
let id_int' : int -> int = id
```

In effect we took a value of type `'a -> 'a`, and we bound it to a name whose type was manually specified as being `int -> int`. You might ask, why does that work? They aren't the same types, after all.

One way to think about this is in terms of **behavior.** The type of `id_int` specifies one aspect of its behavior: given an `int` as input, it promises to produce an `int` as output. It turns out that `id` also makes the same promise: given an `int` as input, it too will return an `int` as output. Now `id` also makes many more promises, such as: given a `bool` as input, it will return a `bool` as output. So by binding `id` to a more restrictive type `int -> int`, we have thrown away all those additional promises as irrelevant. Sure, that's information lost, but at least no promises will be broken. It's always going to be safe to use a function of type `'a -> 'a` when what we needed was a function of type `int -> int`.

The converse is not true. If we needed a function of type `'a -> 'a` but tried to use a function of type `int -> int`, we'd be in trouble as soon as someone passed an input of e.g. type `bool`. Here are two concrete examples, which of course do not compile:

```
let id' : 'a -> 'a = fun x -> x + 1;;
let id'' : 'a -> 'a = id_int;;
```

The first is maybe more obviously wrong than the second. Function `id'` is actually the increment function, not the identify function. So passing it a `bool` or `string` or some complicated data structure is not safe; the only data `+` can safely manipulate are integers.

Function `id''` is maybe less obviously wrong. If follow the chain of definition backwards, we discover that `id''` is really going to be just `fun x -> x`. There's nothing wrong with that code as the implementation of `id''`: it's guaranteed to return an output of the same type as its input, because it *does* return its input. But the compiler doesn't go back and peek "under the covers" like that. Function `id_int` has type `int -> int` at the point `id''` is being defined, and that's all that matters. The compiler has to assume it could be *any* function of that type, including the increment function.

That leads us to another, more mechanical, way to think about all of this in terms of **application**. By that we mean the very same notion of how a function is applied to arguments: when evaluating the application `id 5`, the argument `x` is *instantiated* with value `5`. Likewise, the `'a` in the type of `id` is being instantiated with type `int` at that application. So if we write

```
let id_int' : int -> int = id
```

we are in fact instantiating the `'a` in the type of `id` with the type `int`. And just as there is no way to "unapply" a function—for example, given `6` we can't compute backwards to `id 5`—we can't unapply that type instantiation and change `int` back to `'a`.

To make that precise, suppose we have a `let` definition [or expression]:

```
let x = e [in e']
```

and that OCaml infers `x` has a type `t` that includes some type variables `'a`, `'b`, etc. Then we are permitted to instantiate those type variables. We can do that by applying the function to arguments that reveal what the type instantiations should be (as in `id 5`) or by a type annotation (as in `id_int'`), among other ways. But we have to be consistent with the instantiation. For example, we cannot take a function of type `'a -> 'b -> 'a` and instantiate it at type `int -> 'b -> string`, because the instantiation of `'a` is not the same type in each of the two places it occurred:

```
let first x y = x;;
let first_int : int -> 'b -> int = first;;
let bad_first : int -> 'b -> string = first;;
```

4.4.6 Labeled and Optional Arguments

The type and name of a function usually give you a pretty good idea of what the arguments should be. However, for functions with many arguments (especially arguments of the same type), it can be useful to label them. For example, you might guess that the function `String.sub` returns a substring of the given string (and you would be correct). You could type in `String.sub` to find its type:

```
String.sub;;
```

But it's not clear from the type how to use it—you're forced to consult the documentation.

OCaml supports labeled arguments to functions. You can declare this kind of function using the following syntax:

```
let f ~name1:arg1 ~name2:arg2 = arg1 + arg2;;
```

This function can be called by passing the labeled arguments in either order:

```
f ~name2:3 ~name1:4
```

Labels for arguments are often the same as the variable names for them. OCaml provides a shorthand for this case. The following are equivalent:

```
let f ~name1:name1 ~name2:name2 = name1 + name2
let f ~name1 ~name2 = name1 + name2
```

Use of labeled arguments is largely a matter of taste. They convey extra information, but they can also add clutter to types.

The syntax to write both a labeled argument and an explicit type annotation for it is:

```
let f ~name1:(arg1 : int) ~name2:(arg2 : int) = arg1 + arg2
```

It is also possible to make some arguments optional. When called without an optional argument, a default value will be provided. To declare such a function, use the following syntax:

```
let f ?name:(arg1=8) arg2 = arg1 + arg2
```

You can then call a function with or without the argument:

```
f ~name:2 7
```

```
f 7
```

4.4.7 Partial Application

We could define an addition function as follows:

```
let add x y = x + y
```

Here's a rather similar function:

```
let addx x = fun y -> x + y
```

Function `addx` takes an integer `x` as input and returns a *function* of type `int -> int` that will add `x` to whatever is passed to it.

The type of `addx` is `int -> int -> int`. The type of `add` is also `int -> int -> int`. So from the perspective of their types, they are the same function. But the form of `addx` suggests something interesting: we can apply it to just a single argument.

```
let add5 = addx 5
```

```
add5 2
```

It turns out the same can be done with `add`:

```
let add5 = add 5
```

```
add5 2;;
```

What we just did is called *partial application*: we partially applied the function `add` to one argument, even though you would normally think of it as a multi-argument function. This works because the following three functions are *syntactically different* but *semantically equivalent*. That is, they are different ways of expressing the same computation:

```
let add x y = x + y
let add x = fun y -> x + y
let add = fun x -> (fun y -> x + y)
```

So `add` is really a function that takes an argument `x` and returns a function `(fun y -> x + y)`. Which leads us to a deep truth…

4.4.8 Function Associativity

Are you ready for the truth? Take a deep breath. Here goes…

Every OCaml function takes exactly one argument.

Why? Consider `add`: although we can write it as `let add x y = x + y`, we know that's semantically equivalent to `let add = fun x -> (fun y -> x + y)`. And in general,

```
let f x1 x2 ... xn = e
```

is semantically equivalent to

```
let f =
  fun x1 ->
    (fun x2 ->
       (...
          (fun xn -> e)...))
```

So even though you think of `f` as a function that takes `n` arguments, in reality it is a function that takes 1 argument and returns a function.

The type of such a function

```
t1 -> t2 -> t3 -> t4
```

really means the same as

```
t1 -> (t2 -> (t3 -> t4))
```

That is, function types are *right associative*: there are implicit parentheses around function types, from right to left. The intuition here is that a function takes a single argument and returns a new function that expects the remaining arguments.

Function application, on the other hand, is *left associative*: there are implicit parenthesis around function applications, from left to right. So

```
e1 e2 e3 e4
```

really means the same as

```
((e1 e2) e3) e4
```

The intuition here is that the left-most expression grabs the next expression to its right as its single argument.

4.4.9 Operators as Functions

The addition operator + has type `int -> int -> int`. It is normally written *infix*, e.g., `3 + 4`. By putting parentheses around it, we can make it a *prefix* operator:

```
( + )
```

```
( + ) 3 4;;
```

```
let add3 = ( + ) 3
```

```
add3 2
```

The same technique works for any built-in operator.

Normally the spaces are unnecessary. We could write `(+)` or `( + )`, but it is best to include them. Beware of multiplication, which *must* be written as `( * )`, because `(*)` would be parsed as beginning a comment.

We can even define our own new infix operators, for example:

```
let ( ^^ ) x y = max x y
```

And now `2 ^^ 3` evaluates to `3`.

The rules for which punctuation can be used to create infix operators are not necessarily intuitive. Nor is the relative precedence with which such operators will be parsed. So be careful with this usage.

4.4.10 Tail Recursion

Consider the following seemingly uninteresting function, which counts from 1 to `n`:

```
(** [count n] is [n], computed by adding 1 to itself [n] times.  That is,
    this function counts up from 1 to [n]. *)
let rec count n =
  if n = 0 then 0 else 1 + count (n - 1)
```

Counting to 10 is no problem:

```
count 10
```

Counting to 100,000 is no problem either:

```
count 100_000
```

But try counting to 1,000,000 and you'll get the following error:

```
Stack overflow during evaluation (looping recursion?).
```

What's going on here?

The Call Stack. The issue is that the *call stack* has a limited size. You probably learned in one of your introductory programming classes that most languages implement function calls with a stack. That stack contains one element for each function call that has been started but has not yet completed. Each element stores information like the values of local variables and which instruction in the function is currently being executed. When the evaluation of one function body calls another function, a new element is pushed on the call stack, and it is popped off when the called function completes.

The size of the stack is usually limited by the operating system. So if the stack runs out of space, it becomes impossible to make another function call. Normally this doesn't happen, because there's no reason to make that many successive function calls before returning. In cases where it does happen, there's good reason for the operating system to make that program stop: it might be in the process of eating up *all* the memory available on the entire computer, thus harming other programs running on the same computer. The `count` function isn't likely to do that, but this function would:

```
let rec count_forever n = 1 + count_forever n
```

So the operating system for safety's sake limits the call stack size. That means eventually `count` will run out of stack space on a large enough input. Notice how that choice is really independent of the programming language. So this same issue can and does occur in languages other than OCaml, including Python and Java. You're just less likely to have seen it manifest there, because you probably never wrote quite as many recursive functions in those languages.

Tail Recursion. There is a solution to this issue that was described in a 1977 paper about LISP by Guy Steele. The solution, *tail-call optimization*, requires some cooperation between the programmer and the compiler. The programmer does a little rewriting of the function, which the compiler then notices and applies an optimization. Let's see how it works.

Suppose that a recursive function `f` calls itself then returns the result of that recursive call. Our `count` function does *not* do that:

```
let rec count n =
  if n = 0 then 0 else 1 + count (n - 1)
```

Rather, after the recursive call `count (n - 1)`, there is computation remaining: the computer still needs to add `1` to the result of that call.

But we as programmers could rewrite the `count` function so that it does *not* need to do any additional computation after the recursive call. The trick is to create a helper function with an extra parameter:

```
let rec count_aux n acc =
  if n = 0 then acc else count_aux (n - 1) (acc + 1)

let count_tr n = count_aux n 0
```

Function `count_aux` is almost the same as our original `count`, but it adds an extra parameter named `acc`, which is idiomatic and stands for "accumulator". The idea is that the value we want to return from the function is slowly, with each recursive call, being accumulated in it. The "remaining computation" —the addition of 1— now happens *before* the recursive call not *after*. When the base case of the recursion finally arrives, the function now returns `acc`, where the answer has been accumulated.

But the original base case of 0 still needs to exist in the code somwhere. And it does, as the original value of `acc` that is passed to `count_aux`. Now `count_tr` (we'll get to why the name is "tr" in just a minute) works as a replacement for our original `count`.

At this point we've completed the programmer's responsibility, but it's probably not clear why we went through this effort. After all `count_aux` will still call itself recursively too many times as `count` did, and eventually overflow the stack.

That's where the compiler's responsibility kicks in. A good compiler (and the OCaml compiler is good this way) can notice when a recursive call is in *tail position*, which is a technical way of saying "there's no more computation to be done after it returns". The recursive call to `count_aux` is in tail position; the recursive call to `count` is not. Here they are again so you can compare them:

```
let rec count n =
  if n = 0 then 0 else 1 + count (n - 1)

let rec count_aux n acc =
  if n = 0 then acc else count_aux (n - 1) (acc + 1)
```

Here's why tail position matters: **A recursive call in tail position does not need a new stack frame. It can just reuse the existing stack frame.** That's because there's nothing left of use in the existing stack frame! There's no computation left to be done, so none of the local variables, or next instruction to execute, etc. matter any more. None of that memory ever needs to be read again, because that call is effectively already finished. So instead of wasting space by allocating another stack frame, the compiler "recycles" the space used by the previous frame.

This is the *tail-call optimization*. It can even be applied in cases beyond recursive functions if the calling function's stack frame is suitably compatible with the callee. And, it's a big deal. The tail-call optimization reduces the stack space requirements from linear to constant. Whereas `count` needed $O(n)$ stack frames, `count_aux` needs only $O(1)$, because the same frame gets reused over and over again for each recursive call. And that means `count_tr` actually can count to 1,000,000:

```
count_tr 1_000_000
```

Finally, why did we name this function `count_tr`? The "tr" stands for *tail recursive*. A tail recursive function is a recursive function whose recursive calls are all in tail position. In other words, it's a function that (unless there are other pathologies) will not exhaust the stack.

The Importance of Tail Recursion. Sometimes beginning functional programmers fixate a bit too much upon it. If all you care about is writing the first draft of a function, you probably don't need to worry about tail recursion. It's pretty easy to make it tail recursive later if you need to, just by adding an accumulator argument. Or maybe you should rethink how you have designed the function. Take `count`, for example: it's kind of dumb. But later we'll see examples that aren't dumb, such as iterating over lists with thousands of elements.

It is important that the compiler support the optimization. Otherwise, the transformation you do to the code as a programmer makes no difference. Indeed, most compilers do support it, at least as an option. Java is a notable exception.

The Recipe for Tail Recursion. In a nutshell, here's how we made a function be tail recursive:

1. Change the function into a helper function. Add an extra argument: the accumulator, often named `acc`.
2. Write a new "main" version of the function that calls the helper. It passes the original base case's return value as the initial value of the accumulator.
3. Change the helper function to return the accumulator in the base case.
4. Change the helper function's recursive case. It now needs to do the extra work on the accumulator argument, before the recursive call. This is the only step that requires much ingenuity.

An Example: Factorial. Let's tranform this factorial function to be tail recursive:

```
(* [fact n] is [n] factorial *)
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
```

First, we change its name and add an accumulator argument:

```
let rec fact_aux n acc = ...
```

Second, we write a new "main" function that calls the helper with the original base case as the accumulator:

```
let rec fact_tr n = fact_aux n 1
```

Third, we change the helper function to return the accumulator in the base case:

```
if n = 0 then acc ...
```

Finally, we change the recursive case:

```
else fact (n - 1) (n * acc)
```

Putting it all together, we have:

```
let rec fact_aux n acc =
  if n = 0 then acc else fact_aux (n - 1) (n * acc)

let fact_tr n = fact_aux n 1
```

It was a nice exercise, but maybe not worthwhile. Even before we exhaust the stack space, the computation suffers from integer overflow:

```
fact 50
```

To solve that problem, we turn to OCaml's big integer library, Zarith. Here we use a few OCaml features that are beyond anything we've seen so far, but hopefully nothing terribly surprising.

```
#require "zarith.top";;

let rec zfact_aux n acc =
  if Z.equal n Z.zero then acc else zfact_aux (Z.pred n) (Z.mul acc n);;

let zfact_tr n = zfact_aux n Z.one;;

zfact_tr (Z.of_int 50)
```

If you want you can use that code to compute `zfact_tr 1_000_000` without stack or integer overflow, though it will take several minutes.

The chapter on modules will explain the OCaml features we used above in detail, but for now:

- `#require` loads the library, which provides a module named `Z`. Recall that $\mathbb{Z}$ is the symbol used in mathematics to denote the integers.
- `Z.n` means the name `n` defined inside of `Z`.
- The type `Z.t` is the library's name for the type of big integers.
- We use library values `Z.equal` for equality comparison, `Z.zero` for 0, `Z.pred` for predecessor (i.e., subtracting 1), `Z.mul` for multiplication, `Z.one` for 1, and `Z.of_int` to convert a primitive integer to a big integer.

4.5 Documentation

OCaml provides a tool called OCamldoc that works a lot like Java's Javadoc tool: it extracts specially formatted comments from source code and renders them as HTML, making it easy for programmers to read documentation.

4.5.1 How to Document

Here's an example of an OCamldoc comment:

```
(** [sum lst] is the sum of the elements of [lst]. *)
let rec sum lst = ...
```

- The double asterisk is what causes the comment to be recognized as an OCamldoc comment.
- The square brackets around parts of the comment mean that those parts should be rendered in HTML as `type-writer font` rather than the regular font.

Also like Javadoc, OCamldoc supports *documentation tags*, such as `@author`, `@deprecated`, `@param`, `@return`, etc. For example, in the first line of most programming assignments, we ask you to complete a comment like this:

```
(** @author Your Name (your netid) *)
```

For the full range of possible markup inside a OCamldoc comment, see the OCamldoc manual. But what we've covered here is good enough for most documentation that you'll need to write.

4.5.2 What to Document

The documentation style we favor in this book resembles that of the OCaml standard library: concise and declarative. As an example, let's revisit the documentation of `sum`:

```
(** [sum lst] is the sum of the elements of [lst]. *)
let rec sum lst = ...
```

That comment starts with `sum lst`, which is an example application of the function to an argument. The comment continues with the word "is", thus declaratively describing the result of the application. (The word "returns" could be used instead, but "is" emphasizes the mathematical nature of the function.) That description uses the name of the argument, `lst`, to explain the result.

Note how there is no need to add tags to redundantly describe parameters or return values, as is often done with Javadoc. Everything that needs to be said has already been said. We strongly discourage documentation like the following:

```
(** Sum a list.
    @param lst The list to be summed.
    @return The sum of the list. *)
let rec sum lst = ...
```

That poor documentation takes three needlessly hard-to-read lines to say the same thing as the limpid one-line version.

There is one way we might improve the documentation we have so far, which is to explicitly state what happens with empty lists:

```
(** [sum lst] is the sum of the elements of [lst].
    The sum of an empty list is 0. *)
let rec sum lst = ...
```

4.5.3 Preconditions and Postconditions

Here are a few more examples of comments written in the style we favor.

```
(** [lowercase_ascii c] is the lowercase ASCII equivalent of
    character [c]. *)

(** [index s c] is the index of the first occurrence of
    character [c] in string [s].  Raises: [Not_found]
    if [c] does not occur in [s]. *)

(** [random_int bound] is a random integer between 0 (inclusive)
    and [bound] (exclusive).  Requires: [bound] is greater than 0
    and less than 2^30. *)
```

The documentation of `index` specifies that the function raises an exception, as well as what that exception is and the condition under which it is raised. (We will cover exceptions in more detail in the next chapter.) The documentation of `random_int` specifies that the function's argument must satisfy a condition.

In previous courses, you were exposed to the ideas of *preconditions* and *postconditions*. A precondition is something that must be true before some section of code; and a postcondition, after.

The "Requires" clause above in the documentation of `random_int` is a kind of precondition. It says that the client of the `random_int` function is responsible for guaranteeing something about the value of `bound`. Likewise, the first sentence of that same documentation is a kind of postcondition. It guarantees something about the value returned by the function.

The "Raises" clause in the documentation of `index` is another kind of postcondition. It guarantees that the function raises an exception. Note that the clause is not a precondition, even though it states a condition in terms of an input.

Note that none of these examples has a "Requires" clause that says something about the type of an input. If you're coming from a dynamically-typed language, like Python, this could be a surprise. Python programmers frequently document preconditions regarding the types of function inputs. OCaml programmers, however, do not. That's because the compiler itself does the type checking to ensure that you never pass a value of the wrong type to a function. Consider `lowercase_ascii` again: although the English comment helpfully identifies the type of `c` to the reader, the comment does not state a "Requires" clause like this:

```
(** [lowercase_ascii c] is the lowercase ASCII equivalent of [c].
    Requires: [c] is a character. *)
```

Such a comment reads as highly unidiomatic to an OCaml programmer, who would read that comment and be puzzled, perhaps thinking: "Well of course `c` is a character; the compiler will guarantee that. What did the person who wrote that really mean? Is there something they or I am missing?"

4.6 Printing

OCaml has built-in printing functions for a few of the built-in primitive types: `print_char`, `print_string`, `print_int`, and `print_float`. There's also a `print_endline` function, which is like `print_string`, but also outputs a newline.

```
print_endline "Camels are bae"
```

4.6.1 Unit

Let's look at the types of a couple of those functions:

```
print_endline
```

```
print_string
```

They both take a string as input and return a value of type `unit`, which we haven't seen before. There is only one value of this type, which is written `()` and is also pronounced "unit". So `unit` is like `bool`, except there is one fewer value of type `unit` than there is of `bool`.

Unit is used when you need to take an argument or return a value, but there's no interesting value to pass or return. It is the equivalent of `void` in Java, and is similar to `None` in Python. Unit is often used when you're writing or using code that has side effects. Printing is an example of a side effect: it changes the world and can't be undone.

4.6.2 Semicolon

If you want to print one thing after another, you could sequence some print functions using nested `let` expressions:

```
let _ = print_endline "Camels" in
let _ = print_endline "are" in
print_endline "bae"
```

The `let _ = e` syntax above is a way of evaluating `e` but not binding its value to any name. Indeed, we know the value each of those `print_endline` functions will return: it will always be `()`, the unit value. So there's no good reason to bind it to a variable name. We could also write `let () = e` to indicate we know it's just a unit value that we don't care about:

```
let () = print_endline "Camels" in
let () = print_endline "are" in
print_endline "bae"
```

But either way the boilerplate of all the `let..in` is annoying to have to write! So there's a special syntax that can be used to chain together multiple functions that return unit. The expression `e1; e2` first evaluates `e1`, which should evaluate to `()`, then discards that value, and evaluates `e2`. So we could rewrite the above code as:

```
print_endline "Camels";
print_endline "are";
print_endline "bae"
```

That is more idiomatic OCaml code, and it also looks more natural to imperative programmers.

Warning: There is no semicolon after the final `print_endline` in that example. A common mistake is to put a semicolon *after* each print statement. Instead, the semicolons go strictly *between* statements. That is, semicolon is a statement *separator* not a statement *terminator*. If you were to add a semicolon at the end, you could get a syntax error depending on the surrounding code.

4.6.3 Ignore

If `e1` does not have type `unit`, then `e1; e2` will give a warning, because you are discarding a potentially useful value. If that is truly your intent, you can call the built-in function `ignore : 'a -> unit` to convert any value to `()`:

```
(ignore 3); 5
```

Actually `ignore` is easy to implement yourself:

```
let ignore x = ()
```

Or you can even write underscore to indicate the function takes in a value but does not bind that value to a name. That means the function can never use that value in its body. But that's okay: we want to ignore it.

```
let ignore _ = ()
```

4.6.4 Printf

For complicated text outputs, using the built-in functions for primitive type printing quickly becomes tedious. For example, suppose you wanted to write a function to print a statistic:

```
(** [print_stat name num] prints [name: num]. *)
let print_stat name num =
  print_string name;
  print_string ": ";
  print_float num;
  print_newline ()
```

```
print_stat "mean" 84.39
```

How could we shorten `print_stat`? In Java you might use the overloaded + operator to turn all objects into strings:

```
void print_stat(String name, double num) {
   System.out.println(name + ": " + num);
}
```

But OCaml values are not objects, and they do not have a `toString()` method they inherit from some root `Object` class. Nor does OCaml permit overloading of operators.

Long ago though, FORTRAN invented a different solution that other languages like C and Java and even Python support. The idea is to use a *format specifier* to —as the name suggest— specify how to format output. The name this idea is best known under is probably "printf", which refers to the name of the C library function that implemented it. Many other languages and libraries still use that name, including OCaml's `Printf` module.

Here's how we'd use `printf` to re-implement `print_stat`:

```
let print_stat name num =
  Printf.printf "%s: %F\n%!" name num
```

```
print_stat "mean" 84.39
```

The first argument to function `Printf.printf` is the format specifier. It *looks* like a string, but there's more to it than that. It's actually understood by the OCaml compiler in quite a deep way. Inside the format specifier there are:

- plain characters, and
- conversion specifiers, which begin with `%`.

There are about two dozen conversion specifiers available, which you can read about in the documentation of `Printf`. Let's pick apart the format specifier above as an example.

- It starts with `"%s"`, which is the conversion specifier for strings. That means the next argument to `printf` must be a `string`, and the contents of that string will be output.
- It continues with `":  "`, which are just plain characters. Those are inserted into the output.
- It then has another conversion specifier, `%F`. That means the next argument of `printf` must have type `float`, and will be output in the same format that OCaml uses to print floats.
- The newline `"\n"` after that is another plain character sequence.
- Finally the conversion specifier `"%!"` means to *flush the output buffer*. As you might have learned in earlier programming classes, output is often *buffered*, meaning that it doesn't all happen at once or right away. Flushing the buffer ensures that anything still sitting in the buffer gets output immediately. This specifier is special in that it doesn't actually need another argument to `printf`.

If the type of an argument is incorrect with respect to the conversion specifier, OCaml will detect that. Let's add a type annotation to force `num` to be an `int`, and see what happens with the float conversion specifier `%F`:

```
let print_stat name (num : int) =
  Printf.printf "%s: %F\n%!" name num
```

To fix that, we can change to the conversion specifier for `int`, which is `%i`:

```
let print_stat name num =
  Printf.printf "%s: %i\n%!" name num
```

Another very useful variant of `printf` is `sprintf`, which collects the output in string instead of printing it:

```
let string_of_stat name num =
  Printf.sprintf "%s: %F" name num
```

```
string_of_stat "mean" 84.39
```

4.7 Debugging

Debugging is a last resort when everything else has failed. Let's take a step back and think about everything that comes *before* debugging.

4.7.1 Defenses against Bugs

According to Rob Miller, there are four defenses against bugs:

1. **The first defense against bugs is to make them impossible.**

   Entire classes of bugs can be eradicated by choosing to program in languages that guarantee *memory safety* (that no part of memory can be accessed except through a *pointer* (or reference) that is valid for that region of memory) and *type safety* (that no value can be used in a way inconsistent with its type). The OCaml type system, for example, prevents programs from buffer overflows and meaningless operations (like adding a boolean to a float), whereas the C type system does not.

2. **The second defense against bugs is to use tools that find them.**

   There are automated source-code analysis tools, like FindBugs, which can find many common kinds of bugs in Java programs, and SLAM, which is used to find bugs in device drivers. The subfield of CS known as *formal methods* studies how to use mathematics to specify and verify programs, that is, how to prove that programs have no bugs. We'll study verification later in this course.

   Social methods such as code reviews and pair programming are also useful tools for finding bugs. Studies at IBM in the 1970s-1990s suggested that code reviews can be remarkably effective. In one study (Jones, 1991), code inspection found 65% of the known coding errors and 25% of the known documentation errors, whereas testing found only 20% of the coding errors and none of the documentation errors.

3. **The third defense against bugs is to make them immediately visible.**

   The earlier a bug appears, the easier it is to diagnose and fix. If computation instead proceeds past the point of the bug, then that further computation might obscure where the failure really occurred. *Assertions* in the source code make programs "fail fast" and "fail loudly", so that bugs appear immediately, and the programmer knows exactly where in the source code to look.

4. **The fourth defense against bugs is extensive testing.**

   How can you know whether a piece of code has a particular bug? Write tests that would expose the bug, then confirm that your code doesn't fail those tests. *Unit tests* for a relatively small piece of code, such as an individual function or module, are especially important to write at the same time as you develop that code. Running of those tests should be automated, so that if you ever break the code, you find out as soon as possible. (That's really Defense 3 again.)

After all those defenses have failed, a programmer is forced to resort to debugging.

4.7.2 How to Debug

So you've discovered a bug. What next?

1. **Distill the bug into a small test case.** Debugging is hard work, but the smaller the test case, the more likely you are to focus your attention on the piece of code where the bug lurks. Time spent on this distillation can therefore be time saved, because you won't have to re-read lots of code. Don't continue debugging until you have a small test case!

2. **Employ the scientific method.** Formulate a hypothesis as to why the bug is occurring. You might even write down that hypothesis in a notebook, as if you were in a Chemistry lab, to clarify it in your own mind and keep track

of what hypotheses you've already considered. Next, design an experiment to affirm or deny that hypothesis. Run your experiment and record the result. Based on what you've learned, reformulate your hypothesis. Continue until you have rationally, scientifically determined the cause of the bug.

3. **Fix the bug.** The fix might be a simple correction of a typo. Or it might reveal a design flaw that causes you to make major changes. Consider whether you might need to apply the fix to other locations in your code base—for example, was it a copy and paste error? If so, do you need to refactor your code?
4. **Permanently add the small test case to your test suite.** You wouldn't want the bug to creep back into your code base. So keep track of that small test case by keeping it as part of your unit tests. That way, any time you make future changes, you will automatically be guarding against that same bug. Repeatedly running tests distilled from previous bugs is a part of *regression testing*.

4.7.3 Debugging in OCaml

Here are a couple tips on how to debug—if you are forced into it—in OCaml.

- **Print statements.** Insert a print statement to ascertain the value of a variable. Suppose you want to know what the value of `x` is in the following function:

```
let inc x = x + 1
```

  Just add the line below to print that value:

```
let inc x =
  let () = print_int x in
  x + 1
```

- **Function traces.** Suppose you want to see the *trace* of recursive calls and returns for a function. Use the `#trace` directive:

```
# let rec fib x = if x <= 1 then 1 else fib (x - 1) + fib (x - 2);;
# #trace fib;;
```

  If you evaluate `fib 2`, you will now see the following output:

```
fib <-- 2
fib <-- 0
fib --> 1
fib <-- 1
fib --> 1
fib --> 2
```

  To stop tracing, use the `#untrace` directive.

- **Debugger.** OCaml has a debugging tool `ocamldebug`. You can find a tutorial on the OCaml website. Unless you are using Emacs as your editor, you will probably find this tool to be harder to use than just inserting print statements.

4.7.4 Defensive Programming

As we discussed earlier in the section on debugging, one defense against bugs is to make any bugs (or errors) immediately visible. That idea connects with idea of preconditions.

Consider this specification of `random_int`:

```
(** [random_int bound] is a random integer between 0 (inclusive)
    and [bound] (exclusive).  Requires: [bound] is greater than 0
    and less than 2^30. *)
```

If the client of `random_int` passes a value of `bound` that violates the "Requires" clause, such as `-1`, the implementation of `random_int` is free to do anything whatsoever. All bets are off when the client violates the precondition.

But the most helpful thing for `random_int` to do is to immediately expose the fact that the precondition was violated. After all, chances are that the client didn't *mean* to violate it.

So the implementor of `random_int` would do well to check whether the precondition is violated, and if so, raise an exception. Here are three possibilities of that kind of *defensive programming:*

```
(* possibility 1 *)
let random_int bound =
  assert (bound > 0 && bound < 1 lsl 30);
  (* proceed with the implementation of the function *)

(* possibility 2 *)
let random_int bound =
  if not (bound > 0 && bound < 1 lsl 30)
  then invalid_arg "bound";
  (* proceed with the implementation of the function *)

(* possibility 3 *)
let random_int bound =
  if not (bound > 0 && bound < 1 lsl 30)
  then failwith "bound";
  (* proceed with the implementation of the function *)
```

The second possibility is probably the most informative to the client, because it uses the built-in function `invalid_arg` to raise the well-named exception `Invalid_argument`. In fact, that's exactly what the standard library implementation of this function does.

The first possibility is probably most useful when you are trying to debug your own code, rather than choosing to expose a failed assertion to a client.

The third possibility differs from the second only in the name (`Failure`) of the exception that is raised. It might be useful in situations where the precondition involves more than just a single invalid argument.

In this example, checking the precondition is computationally cheap. In other cases, it might require a lot of computation, so the implementer of the function might prefer not to check the precondition, or only to check some inexpensive approximation to it.

Sometimes programmers worry unnecessarily that defensive programming will be too expensive—either in terms of the time it costs them to implement the checks initially, or in the run-time costs that will be paid in checking assertions. These concerns are far too often misplaced. The time and money it costs society to repair faults in software suggests that we could all afford to have programs that run a little more slowly.

Finally, the implementer might even choose to eliminate the precondition and restate it as a postcondition:

```
(** [random_int bound] is a random integer between 0 (inclusive)
    and [bound] (exclusive).  Raises: [Invalid_argument "bound"]
    unless [bound] is greater than 0 and less than 2^30. *)
```

Now instead of being free to do whatever when `bound` is too big or too small, `random_int` must raise an exception. For this function, that's probably the best choice.

In this course, we're not going to force you to program defensively. But if you're savvy, you'll start (or continue) doing it anyway. The small amount of time you spend coding up such defenses will save you hours of time in debugging, making you a more productive programmer.

4.8 Summary

Syntax and semantics are a powerful paradigm for learning a programming language. As we learn the features of OCaml, we're being careful to write down their syntax and semantics. We've seen that there can be multiple syntaxes for expressing the same semantic idea, that is, the same computation.

The semantics of function application is the very heart of OCaml and of functional programming, and it's something we will come back to several times throughout the course to deepen our understanding.

4.8.1 Terms and Concepts

- anonymous functions
- assertions
- binding
- binding expression
- body expression
- debugging
- defensive programming
- definitions
- documentation
- dynamic semantics
- evaluation
- expressions
- function application
- function definitions
- identifiers
- idioms
- if expressions
- lambda expressions
- let definition
- let expression

- libraries
- metavariables
- mutual recursion
- pipeline operator
- postcondition
- precondition
- printing
- recursion
- semantics
- static semantics
- substitution
- syntax
- tools
- type checking
- type inference
- values

4.8.2 Further Reading

- *Introduction to Objective Caml*, chapter 3
- *OCaml from the Very Beginning*, chapter 2
- *Real World OCaml*, chapter 2
- *Tail Recursion, The Musical*. Tail-call optimization explained in the context of JavaScript with cute 8-bit animations, and Disney songs!

4.9 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Exercise: values [★]

What is the type and value of each of the following OCaml expressions?

- `7 * (1 + 2 + 3)`
- `"CS " ^ string_of_int 3110`

Hint: type each expression into the toplevel and it will tell you the answer. Note: ^ is not exponentiation.

Exercise: operators [★★]

Examine the table of all operators in the OCaml manual (you will have to scroll down to find it on that page).

- Write an expression that multiplies `42` by `10`.
- Write an expression that divides `3.14` by `2.0`. *Hint: integer and floating-point operators are written differently in OCaml.*
- Write an expression that computes `4.2` raised to the seventh power. *Note: there is no built-in integer exponentiation operator in OCaml (nor is there in C, by the way), in part because it is not an operation provided by most CPUs.*

Exercise: equality [★]

- Write an expression that compares `42` to `42` using structural equality.
- Write an expression that compares `"hi"` to `"hi"` using structural equality. What is the result?
- Write an expression that compares `"hi"` to `"hi"` using physical equality. What is the result?

Exercise: assert [★]

- Enter `assert true;;` into utop and see what happens.
- Enter `assert false;;` into utop and see what happens.
- Write an expression that asserts 2110 is not (structurally) equal to 3110.

Exercise: if [★]

Write an if expression that evaluates to `42` if `2` is greater than `1` and otherwise evaluates to `7`.

Exercise: double fun [★]

Using the increment function from above as a guide, define a function `double` that multiplies its input by 2. For example, `double 7` would be `14`. Test your function by applying it to a few inputs. Turn those test cases into assertions.

Exercise: more fun [★★]

- Define a function that computes the cube of a floating-point number. Test your function by applying it to a few inputs.
- Define a function that computes the sign (1, 0, or -1) of an integer. Use a nested if expression. Test your function by applying it to a few inputs.
- Define a function that computes the area of a circle given its radius. Test your function with `assert`.

For the latter, bear in mind that floating-point arithmetic is not exact. Instead of asserting an exact value, you should assert that the result is "close enough", e.g., within 1e-5. If that's unfamiliar to you, it would be worthwhile to read up on floating-point arithmetic.

A function that take multiple inputs can be defined just by providing additional names for those inputs as part of the let definition. For example, the following function computes the average of three arguments:

```
let avg3 x y z = (x +. y +. z) /. 3.
```

Exercise: RMS [★★]

Define a function that computes the *root mean square* of two numbers—i.e., $\sqrt{(x^2 + y^2)/2}$. Test your function with `assert`.

Exercise: date fun [★★★]

Define a function that takes an integer `d` and string `m` as input and returns `true` just when `d` and `m` form a *valid date*. Here, a valid date has a month that is one of the following abbreviations: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept, Oct, Nov, Dec. And the day must be a number that is between 1 and the minimum number of days in that month, inclusive. For example, if the month is Jan, then the day is between 1 and 31, inclusive, whereas if the month is Feb, then the day is between 1 and 28, inclusive.

How terse (i.e., few and short lines of code) can you make your function? You can definitely do this in fewer than 12 lines.

Exercise: fib [★★]

Define a recursive function `fib : int -> int`, such that `fib n` is the `n`th number in the Fibonacci sequence, which is 1, 1, 2, 3, 5, 8, 13, … That is:

- `fib 1 = 1`,
- `fib 2 = 1`, and
- `fib n = fib (n-1) + fib (n-2)` for any `n > 2`.

Test your function in the toplevel.

Exercise: fib fast [★★★]

How quickly does your implementation of `fib` compute the 50th Fibonacci number? If it computes nearly instantaneously, congratulations! But the recursive solution most people come up with at first will seem to hang indefinitely. The problem is that the obvious solution computes subproblems repeatedly. For example, computing `fib 5` requires computing both `fib 3` and `fib 4`, and if those are computed separately, a lot of work (an exponential amount, in fact) is being redone.

Create a function `fib_fast` that requires only a linear amount of work. *Hint:* write a recursive helper function `h : int -> int -> int -> int`, where `h n pp p` is defined as follows:

- `h 1 pp p = p`, and
- `h n pp p = h (n-1) p (pp+p)` for any `n > 1`.

The idea of `h` is that it assumes the previous two Fibonacci numbers were `pp` and `p`, then computes forward `n` more numbers. Hence, `fib n = h n 0 1` for any `n > 0`.

What is the first value of `n` for which `fib_fast n` is negative, indicating that integer overflow occurred?

Exercise: poly types [★★★]

What is the type of each of the functions below? You can ask the toplevel to check your answers

```
let f x = if x then x else x
let g x y = if y then x else x
let h x y z = if x then y else z
let i x y z = if x then y else y
```

Exercise: divide [★★]

Write a function `divide : numerator:float -> denominator:float -> float`. Apply your function.

Exercise: associativity [★★]

Suppose that we have defined `let add x y = x + y`. Which of the following produces an integer, which produces a function, and which produces an error? Decide on an answer, then check your answer in the toplevel.

- `add 5 1`
- `add 5`
- `(add 5) 1`
- `add (5 1)`

Exercise: average [★★]

Define an infix operator `+/.` to compute the average of two floating-point numbers. For example,

- `1.0 +/. 2.0 = 1.5`
- `0. +/. 0. = 0.`

Exercise: hello world [★]

Type the following in utop:

- `print_endline "Hello world!";;`
- `print_string "Hello world!";;`

Notice the difference in output from each.

Part III

OCaml Programming

CHAPTER

FIVE

DATA AND TYPES

In this chapter, we'll examine some of OCaml's built-in data types, including lists, variants, records, tuples, and options. Many of those are likely to feel familiar from other programming languages. In particular,

- **lists** and **tuples**, might feel similar to Python; and
- **records** and **variants**, might feel similar to `struct` and `enum` types from C or Java.

Because of that familiarity, we call these *standard* data types. We'll learn about *pattern matching*, which is a feature that's less likely to be familiar.

Almost immediately after we learn about lists, we'll pause our study of standard data types to learn about unit testing in OCaml with OUnit, a unit testing framework similar to those you might have used in other languages. OUnit relies on lists, which is why we couldn't cover it before now.

Later in the chapter, we study some OCaml data types that are unlikely to be as familiar from other languages. They include:

- **options**, which are loosely related to `null` in Java;
- **association lists**, which are an amazingly simple implementation of maps (aka dictionaries) based on lists and tuples;
- **algebraic data types**, which are arguably the most important kind of type in OCaml, and indeed are the power behind many of the other built-in types; and
- **exceptions**, which are a special kind of algebraic data type.

5.1 Lists

An OCaml list is a sequence of values all of which have the same type. They are implemented as singly-linked lists. These lists enjoy a first-class status in the language: there is special support for easily creating and working with lists. That's a characteristic that OCaml shares with many other functional languages. Mainstream imperative languages, like Python, have such support these days too. Maybe that's because programmers find it so pleasant to work directly with lists as a first-class part of the language, rather than having to go through a library (as in C and Java).

5.1.1 Building Lists

Syntax. There are three syntactic forms for building lists:

```
[]
e1 :: e2
[e1; e2; ...; en]
```

The empty list is written `[]` and is pronounced "nil", a name that comes from Lisp. Given a list `lst` and element `elt`, we can prepend `elt` to `lst` by writing `elt :: lst`. The double-colon operator is pronounced "cons", a name that comes from an operator in Lisp that constructs objects in memory. "Cons" can also be used as a verb, as in "I will cons an element onto the list." The first element of a list is usually called its *head* and the rest of the elements (if any) are called its *tail*.

The square bracket syntax is convenient but unnecessary. Any list `[e1; e2; ...; en]` could instead be written with the more primitive nil and cons syntax: `e1 :: e2 :: ... :: en :: []`. When a pleasant syntax can be defined in terms of a more primitive syntax within the language, we call the pleasant syntax *syntactic sugar*: it makes the language "sweeter". Transforming the sweet syntax into the more primitive syntax is called *desugaring*.

Because the elements of the list can be arbitrary expressions, lists can be nested as deeply as we like, e.g., `[[[]]; [[1; 2; 3]]]`.

Dynamic semantics.

- `[]` is already a value.
- If `e1` evaluates to `v1`, and if `e2` evaluates to `v2`, then `e1 :: e2` evaluates to `v1 :: v2`.

As a consequence of those rules and how to desugar the square-bracket notation for lists, we have the following derived rule:

- If `ei` evaluates to `vi` for all `i` in `1..n`, then `[e1; ...; en]` evaluates to `[v1; ...; vn]`.

It's starting to get tedious to write "evaluates to" in all our evaluation rules. So let's introduce a shorter notation for it. We'll write `e ==> v` to mean that `e` evaluates to `v`. Note that `==>` is not a piece of OCaml syntax. Rather, it's a notation we use in our description of the language, kind of like metavariables. Using that notation, we can rewrite the latter two rules above:

- If `e1 ==> v1`, and if `e2 ==> v2`, then `e1 :: e2 ==> v1 :: v2`.
- If `ei ==> vi` for all `i` in `1..n`, then `[e1; ...; en] ==> [v1; ...; vn]`.

Static semantics.

All the elements of a list must have the same type. If that element type is `t`, then the type of the list is `t list`. You should read such types from right to left: `t list` is a list of `t`'s, `t list list` is a list of list of `t`'s, etc. The word `list` itself here is not a type: there is no way to build an OCaml value that has type simply `list`. Rather, `list` is a *type constructor*: given a type, it produces a new type. For example, given `int`, it produces the type `int list`. You could think of type constructors as being like functions that operate on types, instead of functions that operate on values.

The type-checking rules:

- `[] : 'a list`
- If `e1 : t` and `e2 : t list` then `e1 :: e2 : t list`. In case the colons and their precedence is confusing, the latter means `(e1 :: e2) : t list`.

In the rule for `[]`, recall that `'a` is a type variable: it stands for an unknown type. So the empty list is a list whose elements have an unknown type. If we cons an `int` onto it, say `2 :: []`, then the compiler infers that for that particular list, `'a` must be `int`. But if in another place we cons a `bool` onto it, say `true :: []`, then the compiler infers that for that particular list, `'a` must be `bool`.

5.1.2 Accessing Lists

Note: The video linked above also uses records and tuples as examples. Those are covered in the *next section* of this book.

There are really only two ways to build a list, with nil and cons. So if we want to take apart a list into its component pieces, we have to say what to do with the list if it's empty, and what to do if it's non-empty (that is, a cons of one element onto some other list). We do that with a language feature called *pattern matching*.

Here's an example of using pattern matching to compute the sum of a list:

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h :: t -> h + sum t
```

This function says to take the input `lst` and see whether it has the same shape as the empty list. If so, return 0. Otherwise, if it has the same shape as the list `h :: t`, then let `h` be the first element of `lst`, and let `t` be the rest of the elements of `lst`, and return `h + sum t`. The choice of variable names here is meant to suggest "head" and "tail" and is a common idiom, but we could use other names if we wanted. Another common idiom is:

```
let rec sum xs =
  match xs with
  | [] -> 0
  | x :: xs' -> x + sum xs'
```

That is, the input list is a list of xs (pronounced EX-uhs), the head element is an x, and the tail is xs' (pronounced EX-uhs prime).

Syntactically it isn't necessary to use so many lines to define `sum`. We could do it all on one line:

```
let rec sum xs = match xs with | [] -> 0 | x :: xs' -> x + sum xs'
```

Or, noting that the first `|` after `with` is optional regardless of how many lines we use, we could also write:

```
let rec sum xs = match xs with [] -> 0 | x :: xs' -> x + sum xs'
```

The multi-line format is what we'll usually use in this book, because it helps the human eye understand the syntax a bit better. OCaml code formatting tools, though, are moving toward the single-line format whenever the code is short enough to fit on just one line.

Here's another example of using pattern matching to compute the length of a list:

```
let rec length lst =
  match lst with
  | [] -> 0
  | h :: t -> 1 + length t
```

Note how we didn't actually need the variable `h` in the right-hand side of the pattern match. When we want to indicate the presence of some value in a pattern without actually giving it a name, we can write _ (the underscore character):

```
let rec length lst =
  match lst with
```

(continues on next page)

(continued from previous page)

```
  | [] -> 0
  | _ :: t -> 1 + length t
```

That function is actually built-in as part of the OCaml standard library `List` module. Its name there is `List.length`. That "dot" notation indicates the function named `length` inside the module named `List`, much like the dot notation used in many other languages.

And here's a third example that appends one list onto the beginning of another list:

```
let rec append lst1 lst2 =
  match lst1 with
  | [] -> lst2
  | h :: t -> h :: append t lst2
```

For example, `append [1; 2] [3; 4]` is `[1; 2; 3; 4]`. That function is actually available as a built-in operator `@`, so we could instead write `[1; 2] @ [3; 4]`.

As a final example, we could write a function to determine whether a list is empty:

```
let empty lst =
  match lst with
  | [] -> true
  | h :: t -> false
```

But there a much better way to write the same function without pattern matching:

```
let empty lst =
  lst = []
```

Note how all the recursive functions above are similar to doing proofs by induction on the natural numbers: every natural number is either 0 or is 1 greater than some other natural number n, and so a proof by induction has a base case for 0 and an inductive case for $n + 1$. Likewise all our functions have a base case for the empty list and a recursive case for the list that has one more element than another list. This similarity is no accident. There is a deep relationship between induction and recursion; we'll explore that relationship in more detail later in the book.

By the way, there are two library functions `List.hd` and `List.tl` that return the head and tail of a list. It is not good, idiomatic OCaml to apply these directly to a list. The problem is that they will raise an exception when applied to the empty list, and you will have to remember to handle that exception. Instead, you should use pattern matching: you'll then be forced to match against both the empty list and the non-empty list (at least), which will prevent exceptions from being raised, thus making your program more robust.

5.1.3 (Not) Mutating Lists

Lists are immutable. There's no way to change an element of a list from one value to another. Instead, OCaml programmers create new lists out of old lists. For example, suppose we wanted to write a function that returned the same list as its input list, but with the first element (if there is one) incremented by one. We could do that:

```
let inc_first lst =
  match lst with
  | [] -> []
  | h :: t -> h + 1 :: t
```

Now you might be concerned about whether we're being wasteful of space. After all, there are at least two ways the compiler could implement the above code:

1. Copy the entire tail list `t` when the new list is created in the pattern match with cons, such that the amount of memory in use just increased by an amount proportionate to the length of `t`.
2. Share the tail list `t` between the old list and the new list, such that the amount of memory in use does not increase—beyond the one extra piece of memory needed to store `h + 1`.

In fact, the compiler does the latter. So there's no need for concern. The reason that it's quite safe for the compiler to implement sharing is exactly that list elements are immutable. If they were instead mutable, then we'd start having to worry about whether the list I have is shared with the list you have, and whether changes I make will be visible in your list. So immutability makes it easier to reason about the code, and makes it safe for the compiler to perform an optimization.

5.1.4 Pattern Matching with Lists

We saw above how to access lists using pattern matching. Let's look more carefully at this feature.

Syntax.

```
match e with
| p1 -> e1
| p2 -> e2
| ...
| pn -> en
```

Each of the clauses `pi -> ei` is called a *branch* or a *case* of the pattern match. The first vertical bar in the entire pattern match is optional.

The `p`'s here are a new syntactic form called a *pattern*. For now, a pattern may be:

- a variable name, e.g. `x`
- the underscore character _, which is called the *wildcard*
- the empty list `[]`
- `p1 :: p2`
- `[p1; ...; pn]`

No variable name may appear more than once in a pattern. For example, the pattern `x :: x` is illegal. The wildcard may occur any number of times.

As we learn more of data structures available in OCaml, we'll expand the possibilities for what a pattern may be.

Dynamic semantics.

Pattern matching involves two inter-related tasks: determining whether a pattern matches a value, and determining what parts of the value should be associated with which variable names in the pattern. The former task is intuitively about determining whether a pattern and a value have the same *shape*. The latter task is about determining the *variable bindings* introduced by the pattern. For example, consider the following code:

```
match 1 :: [] with
| [] -> false
| h :: t -> h >= 1 && List.length t = 0
```

When evaluating the right-hand side of the second branch, `h` is bound to `1` and `t` is bound to `[]`. Let's write `h->1` to mean the variable binding saying that `h` has value `1`; this is not a piece of OCaml syntax, but rather a notation we use to reason about the language. So the variable bindings produced by the second branch would be `h->1, t->[]`.

Using that notation, here is a definition of when a pattern matches a value and the bindings that match produces:

- The pattern `x` matches any value `v` and produces the variable binding `x->v`.

- The pattern _ matches any value and produces no bindings.
- The pattern `[]` matches the value `[]` and produces no bindings.
- If `p1` matches `v1` and produces a set b_1 of bindings, and if `p2` matches `v2` and produces a set b_2 of bindings, then `p1 :: p2` matches `v1 :: v2` and produces the set $b_1 \cup b_2$ of bindings. Note that `v2` must be a list (since it's on the right-hand side of `::`) and could have any length: 0 elements, 1 element, or many elements. Note that the union $b_1 \cup b_2$ of bindings will never have a problem where the same variable is bound separately in both b_1 and b_2 because of the syntactic restriction that no variable name may appear more than once in a pattern.
- If for all `i` in `1..n`, it holds that `pi` matches `vi` and produces the set b_i of bindings, then `[p1; ...; pn]` matches `[v1; ...; vn]` and produces the set $\bigcup_i b_i$ of bindings. Note that this pattern specifies the exact length the list must be.

Now we can say how to evaluate `match e with p1 -> e1 | ... | pn -> en`:

- Evaluate `e` to a value `v`.
- Match `v` against `p1`, then against `p2`, and so on, in the order they appear in the match expression.
- If `v` does not match against any of the patterns, then evaluation of the match expression raises a `Match_failure` exception. We haven't yet discussed exceptions in OCaml, but you're surely familiar with them from other languages. We'll come back to exceptions near the end of this chapter, after we've covered some of the other built-in data structures in OCaml.
- Otherwise, stop trying to match at the first time a match succeeds against a pattern. Let `pi` be that pattern and let b be the variable bindings produced by matching `v` against `pi`.
- Substitute those bindings inside `ei`, producing a new expression `e'`.
- Evaluate `e'` to a value `v'`.
- The result of the entire match expression is `v'`.

For example, here's how this match expression would be evaluated:

```
match 1 :: [] with
| [] -> false
| h :: t -> h = 1 && t = []
```

- `1 :: []` is already a value.
- `[]` does not match `1 :: []`.
- `h :: t` does match `1 :: []` and produces variable bindings {`h->1`,`t->[]`}, because:
  - `h` matches `1` and produces the variable binding `h->1`.
  - `t` matches `[]` and produces the variable binding `t->[]`.
- Substituting {`h->1`,`t->[]`} inside `h = 1 && t = []` produces a new expression `1 = 1 && [] = []`.
- Evaluating `1 = 1 && [] = []` yields the value `true`. We omit the justification for that fact here, but it follows from other evaluation rules for built-in operators and function application.
- So the result of the entire match expression is `true`.

Static semantics.

- If `e : ta` and for all `i`, it holds that `pi : ta` and `ei : tb`, then `(match e with p1 -> e1 | ... | pn -> en) : tb`.

That rule relies on being able to judge whether a pattern has a particular type. As usual, type inference comes into play here. The OCaml compiler infers the types of any pattern variables as well as all occurrences of the wildcard pattern. As for the list patterns, they have the same type-checking rules as list expressions.

Additional Static Checking.

In addition to that type-checking rule, there are two other checks the compiler does for each match expression.

First, **exhaustiveness:** the compiler checks to make sure that there are enough patterns to guarantee that at least one of them matches the expression `e`, no matter what the value of that expression is at run time. This ensures that the programmer did not forget any branches. For example, the function below will cause the compiler to emit a warning:

```
let head lst = match lst with h :: _ -> h
```

By presenting that warning to the programmer, the compiler is helping the programmer to defend against the possibility of `Match_failure` exceptions at runtime.

Note: Sorry about how the output from the cell above gets split into many lines in the HTML. That is currently an open issue with JupyterBook, the framework used to build this book.

Second, **unused branches:** the compiler checks to see whether any of the branches could never be matched against because one of the previous branches is guaranteed to succeed. For example, the function below will cause the compiler to emit a warning:

```
let rec sum lst =
  match lst with
  | h :: t -> h + sum t
  | [ h ] -> h
  | [] -> 0
```

The second branch is unused because the first branch will match anything the second branch matches.

Unused match cases are usually a sign that the programmer wrote something other than what they intended. So by presenting that warning, the compiler is helping the programmer to detect latent bugs in their code.

Here's an example of one of the most common bugs that causes an unused match case warning. Understanding it is also a good way to check your understanding of the dynamic semantics of match expressions:

```
let length_is lst n =
  match List.length lst with
  | n -> true
  | _ -> false
```

The programmer was thinking that if the length of `lst` is equal to `n`, then this function will return `true`, and otherwise will return `false`. But in fact this function *always* returns `true`. Why? Because the pattern variable `n` is distinct from the function argument `n`. Suppose that the length of `lst` is 5. Then the pattern match becomes: `match 5 with n -> true | _ -> false`. Does `n` match 5? Yes, according to the rules above: a variable pattern matches any value and here produces the binding `n->5`. Then evaluation applies that binding to `true`, substituting all occurrences of `n` inside of `true` with 5. Well, there are no such occurrences. So we're done, and the result of evaluation is just `true`.

What the programmer really meant to write was:

```
let length_is lst n =
  match List.length lst with
  | m -> m = n
```

or better yet:

```
let length_is lst n =
  List.length lst = n
```

5.1.5 Deep Pattern Matching

Patterns can be nested. Doing so can allow your code to look deeply into the structure of a list. For example:

- `_ :: []` matches all lists with exactly one element
- `_ :: _` matches all lists with at least one element
- `_ :: _ :: []` matches all lists with exactly two elements
- `_ :: _ :: _ :: _` matches all lists with at least three elements

5.1.6 Immediate Matches

When you have a function that immediately pattern-matches against its final argument, there's a nice piece of syntactic sugar you can use to avoid writing extra code. Here's an example: instead of

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h :: t -> h + sum t
```

you can write

```
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t
```

The word `function` is a keyword. Notice that we're able to leave out the line containing `match` as well as the name of the argument, which was never used anywhere else but that line. In such cases, though, it's especially important in the specification comment for the function to document what that argument is supposed to be, since the code no longer gives it a descriptive name.

5.1.7 OCamldoc and List Syntax

OCamldoc is a documentation generator similar to Javadoc. It extracts comments from source code and produces HTML (as well as other output formats). The standard library web documentation for the List module is generated by OCamldoc from the standard library source code for that module, for example.

> **Warning:** There is a syntactic convention with square brackets in OCamldoc that can be confusing with respect to lists.
>
> In an OCamldoc comment, source code is surrounded by square brackets. That code will be rendered in typewriter face and syntax-highlighted in the output HTML. The square brackets in this case do not indicate a list.

For example, here is the comment for `List.hd` in the standard library source code:

```
(** Return the first element of the given list. Raise
   [Failure "hd"] if the list is empty. *)
```

The `[Failure "hd"]` does not mean a list containing the exception `Failure "hd"`. Rather it means to typeset the expression `Failure "hd"` as source code, as you can see here.

This can get especially confusing when you want to talk about lists as part of the documentation. For example, here is a way we could rewrite that comment:

```
(** [hd lst] returns the first element of [lst].
    Raises [Failure "hd"] if [lst = []]. *)
```

In `[lst = []]`, the outer square brackets indicate source code as part of a comment, whereas the inner square brackets indicate the empty list.

5.1.8 List Comprehensions

Some languages, including Python and Haskell, have a syntax called *comprehension* that allows lists to be written somewhat like set comprehensions from mathematics. The earliest example of comprehensions seems to be the functional language NPL, which was designed in 1977.

OCaml doesn't have built-in syntactic support for comprehensions. Though some extensions were developed, none seem to be supported any longer. The primary tasks accomplished by comprehensions (filtering out some elements, and transforming others) are actually well-supported already by *higher-order programming*, which we'll study in a later chapter, and the pipeline operator, which we've already learned. So an additional syntax for comprehensions was never really needed.

5.1.9 Tail Recursion

Recall that a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call. Consider these two implementations, `sum` and `sum_tr` of summing a list:

```
let rec sum (l : int list) : int =
  match l with
  | [] -> 0
  | x :: xs -> x + (sum xs)

let rec sum_plus_acc (acc : int) (l : int list) : int =
  match l with
  | [] -> acc
  | x :: xs -> sum_plus_acc (acc + x) xs

let sum_tr : int list -> int =
  sum_plus_acc 0
```

Observe the following difference between the `sum` and `sum_tr` functions above: In the `sum` function, which is not tail recursive, after the recursive call returned its value, we add `x` to it. In the tail recursive `sum_tr`, or rather in `sum_plus_acc`, after the recursive call returns, we immediately return the value without further computation.

If you're going to write functions on really long lists, tail recursion becomes important for performance. So when you have a choice between using a tail-recursive vs. non-tail-recursive function, you are likely better off using the tail-recursive function on really long lists to achieve space efficiency. For that reason, the List module documents which functions are tail recursive and which are not.

But that doesn't mean that a tail-recursive implementation is strictly better. For example, the tail-recursive function might be harder to read. (Consider `sum_plus_acc`.) Also, there are cases where implementing a tail-recursive function entails having to do a pre- or post-processing pass to reverse the list. On small to medium sized lists, the overhead of reversing the list (both in time and in allocating memory for the reversed list) can make the tail-recursive version less time efficient. What constitutes "small" vs. "big" here? That's hard to say, but maybe 10,000 is a good estimate, according to the standard library documentation of the `List` module.

Here is a useful tail-recursive function to produce a long list:

```
(** [from i j l] is the list containing the integers from [i] to [j],
    inclusive, followed by the list [l].
    Example:  [from 1 3 [0] = [1; 2; 3; 0]] *)
let rec from i j l = if i > j then l else from i (j - 1) (j :: l)

(** [i -- j] is the list containing the integers from [i] to [j], inclusive. *)
let ( -- ) i j = from i j []

let long_list = 0 -- 1_000_000
```

It would be worthwhile to study the definition of `--` to convince yourself that you understand (i) how it works and (ii) why it is tail recursive.

You might in the future decide you want to create such a list again. Rather than having to remember where this definition is, and having to copy it into your code, here's an easy way to create the same list using a built-in library function:

```
List.init 1_000_000 Fun.id
```

Expression `List.init len f` creates the list `[f 0; f 1; ...; f (len - 1)]`, and it does so tail recursively if `len` is bigger than 10,000. Function `Fun.id` is simply the identify function `fun x -> x`.

5.2 Variants

A *variant* is a data type representing a value that is one of several possibilities. At their simplest, variants are like enums from C or Java:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
let d = Tue
```

The individual names of the values of a variant are called *constructors* in OCaml. In the example above, the constructors are `Sun`, `Mon`, etc. This is a somewhat different use of the word constructor than in C++ or Java.

For each kind of data type in OCaml, we've been discussing how to build and access it. For variants, building is easy: just write the name of the constructor. For accessing, we use pattern matching. For example:

```
let int_of_day d =
  match d with
  | Sun -> 1
  | Mon -> 2
  | Tue -> 3
  | Wed -> 4
  | Thu -> 5
  | Fri -> 6
  | Sat -> 7
```

There isn't any kind of automatic way of mapping a constructor name to an `int`, like you might expect from languages with enums.

Syntax.

Defining a variant type:

```
type t = C1 | ... | Cn
```

The constructor names must begin with an uppercase letter. OCaml uses that to distinguish constructors from variable identifiers.

The syntax for writing a constructor value is simply its name, e.g., `C`.

Dynamic semantics.

- A constructor is already a value. There is no computation to perform.

Static semantics.

- If `t` is a type defined as `type t = ... | C | ...`, then `C : t`.

5.2.1 Scope

Suppose there are two types defined with overlapping constructor names, for example,

```
type t1 = C | D
type t2 = D | E
let x = D
```

When `D` appears after these definitions, to which type does it refer? That is, what is the type of `x` above? The answer is that the type defined later wins. So `x : t2`. That is potentially surprising to programmers, so within any given scope (e.g., a file or a module, though we haven't covered modules yet) it's idiomatic whenever overlapping constructor names might occur to prefix them with some distinguishing character. For example, suppose we're defining types to represent Pokémon:

```
type ptype =
  TNormal | TFire | TWater

type peff =
  ENormal | ENotVery | ESuper
```

Because "Normal" would naturally be a constructor name for both the type of a Pokémon and the effectiveness of a Pokémon attack, we add an extra character in front of each constructor name to indicate whether it's a type or an effectiveness.

5.2.2 Pattern Matching

Each time we introduced a new kind of data type, we need to introduce the new patterns associated with it. For variants, this is easy. We add the following new pattern form to the list of legal patterns:

- a constructor name `C`

And we extend the definition of when a pattern matches a value and produces a binding as follows:

- The pattern `C` matches the value `C` and produces no bindings.

Note: Variants are considerably more powerful than what we have seen here. We'll return to them again soon.

5.3 Unit Testing with OUnit

Note: This section is a bit of a detour from our study of data types, but it's a good place to take the detour: we now know just enough to understand how unit testing can be done in OCaml, and there's no good reason to wait any longer to learn about it.

Using the toplevel to test functions will only work for very small programs. Larger programs need *test suites* that contain many *unit tests* and can be re-run every time we update our code base. A unit test is a test of one small piece of functionality in a program, such as an individual function.

We've now learned enough features of OCaml to see how to do unit testing with a library called OUnit. It is a unit testing framework similar to JUnit in Java, HUnit in Haskell, etc. The basic workflow for using OUnit is as follows:

- Write a function in a file `f.ml`. There could be many other functions in that file too.
- Write unit tests for that function in a separate file `test.ml`. That exact name is not actually essential.
- Build and run `test` to execute the unit tests.

The OUnit documentation is available on Github.

5.3.1 An Example of OUnit

The following example shows you how to create an OUnit test suite. There are some things in the example that might at first seem mysterious; they are discussed in the next section.

Create a new directory. In that directory, create a file named `sum.ml`, and put the following code into it:

```
let rec sum = function
  | [] -> 0
  | x :: xs -> x + sum xs
```

Now create a second file named `test.ml`, and put this code into it:

```
open OUnit2
open Sum

let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []));
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]));
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]

let _ = run_test_tt_main tests
```

Depending on your editor and its configuration, you probably now see some "Unbound module" errors about OUnit2 and Sum. Don't worry; the code is actually correct. We just need to set up dune and tell it to link OUnit. Create a `dune` file and put this in it:

```
(executable
 (name test)
 (libraries ounit2))
```

Now build the test suite:

```
$ dune build test.exe
```

Go back to your editor and do anything that will cause it to revisit `test.ml`. You can close and re-open the window, or make a trivial change in the file (e.g., add then delete a space). Now the errors should all disappear.

Finally, you can run the test suite:

```
$ dune exec ./test.exe
```

You will get a response something like this:

```
...
Ran: 3 tests in: 0.12 seconds.
OK
```

Now suppose we modify `sum.ml` to introduce a bug by changing the code in it to the following:

```
let rec sum = function
  | [] -> 1 (* bug *)
  | x :: xs -> x + sum xs
```

If rebuild and re-execute the test suite, all test cases now fail. The output tells us the names of the failing cases. Here's the beginning of the output, in which we've replaced some strings that will be dependent on your own local computer with `...`:

```
FFF
============================================================================
Error: test suite for sum:2:two_elements.

File ".../_build/oUnit-test suite for sum-...#01.log", line 9, characters 1-1:
Error: test suite for sum:2:two_elements (in the log).

Raised at OUnitAssert.assert_failure in file "src/lib/ounit2/advanced/oUnitAssert.ml",
 ↪ line 45, characters 2-27
Called from OUnitRunner.run_one_test.(fun) in file "src/lib/ounit2/advanced/
 ↪oUnitRunner.ml", line 83, characters 13-26

not equal
----------------------------------------------------------------------------
```

The first line of that output

```
FFF
```

tells us that OUnit ran three test cases and all three failed.

The next interesting line

```
Error: test suite for sum:2:two_elements.
```

tells us that in the test suite named `test suite for sum` the test case at index 2 named `two_elements` failed. The rest of the output for that test case is not particularly interesting; let's ignore it for now.

5.3.2 Explanation of the OUnit Example

Let's study more carefully what we just did in the previous section. In the test file, `open OUnit2` brings into scope the many definitions in OUnit2, which is version 2 of the OUnit framework. And `open Sum` brings into scope the definitions from `sum.ml`. We'll learn more about scope and the `open` keyword later in a later chapter.

Then we created a list of test cases:

```
[
  "empty"  >:: (fun _ -> assert_equal 0 (sum []));
  "one"    >:: (fun _ -> assert_equal 1 (sum [1]));
  "onetwo" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]
```

Each line of code is a separate test case. A test case has a string giving it a descriptive name, and a function to run as the test case. In between the name and the function we write `>::`, which is a custom operator defined by the OUnit framework. Let's look at the first function from above:

```
fun _ -> assert_equal 0 (sum [])
```

Every test case function receives as input a parameter that OUnit calls a *test context*. Here (and in many of the test cases we write) we don't actually need to worry about the context, so we use the underscore to indicate that the function ignores its input. The function then calls `assert_equal`, which is a function provided by OUnit that checks to see whether its two arguments are equal. If so the test case succeeds. If not, the test case fails.

Then we created a test suite:

```
let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []));
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]));
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]
```

The `>:::` operator is another custom OUnit operator. It goes between the name of the test suite and the list of test cases in that suite.

Then we ran the test suite:

```
let _ = run_test_tt_main tests
```

The function `run_test_tt_main` is provided by OUnit. It runs a test suite and prints the results of which test cases passed vs. which failed to standard output. The use of `let _ =` here indicates that we don't care what value the function returns; it just gets discarded.

5.3.3 Improving OUnit Output

In our example with the buggy implementation of `sum`, we got the following output:

```
==============================================================================
Error: test suite for sum:2:two_elements.
...
not equal
------------------------------------------------------------------------------
```

The `not equal` in the OUnit output means that `assert_equal` discovered the two values passed to it in that test case were not equal. That's not so informative: we'd like to know *why* they're not equal. In particular, we'd like to know what the actual output produced by `sum` was for that test case. To find out, we need to pass an additional argument to `assert_equal`. That argument, whose label is `printer`, should be a function that can transform the outputs to strings. In this case, the outputs are integers, so `string_of_int` from the Stdlib module will suffice. We modify the test suite as follows:

```
let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []) ~printer:string_of_int);
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]) ~printer:string_of_int);
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]) ~printer:string_of_int);
]
```

And now we get more informative output:

```
==============================================================================
Error: test suite for sum:2:two_elements.
...
expected: 3 but got: 4
------------------------------------------------------------------------------
```

That output means that the test named `two_elements` asserted the equality of `3` and `4`. The expected output was `3` because that was the first input to `assert_equal`, and that function's specification says that in `assert_equal x y`, the output you (as the tester) are expecting to get should be `x`, and the output the function being tested actually produces should be `y`.

Notice how our test suite is accumulating a lot of redundant code. In particular, we had to add the `printer` argument to several lines. Let's improve that code by factoring out a function that constructs test cases:

```
let make_sum_test name expected_output input =
  name >:: (fun _ -> assert_equal expected_output (sum input) ~printer:string_of_int)

let tests = "test suite for sum" >::: [
  make_sum_test "empty" 0 [];
  make_sum_test "singleton" 1 [1];
  make_sum_test "two_elements" 3 [1; 2];
]
```

For output types that are more complicated than integers, you will end up needing to write your own functions to pass to `printer`. This is similar to writing `toString()` methods in Java: for complicated types you invent yourself, the language doesn't know how to render them as strings. You have to provide the code that does it.

5.3.4 Testing for Exceptions

We have a little more of OCaml to learn before we can see how to test for exceptions. You can peek ahead to *the section on exceptions* if you want to know now.

5.3.5 Test-Driven Development

Testing doesn't have to happen strictly after you write code. In *test-driven development* (TDD), testing comes first! It emphasizes *incremental* development of code: there is always something that can be tested. Testing is not something that happens after implementation; instead, *continuous testing* is used to catch errors early. Thus, it is important to develop unit tests immediately when the code is written. Automating test suites is crucial so that continuous testing requires essentially no effort.

Here's an example of TDD. We deliberately choose an exceedingly simple function to implement, so that the process is clear. Suppose we are working with a datatype for days:

```
type day = Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday
```

And we want to write a function `next_weekday : day -> day` that returns the next weekday after a given day. We start by writing the most basic, broken version of that function we can:

```
let next_weekday d = failwith "Unimplemented"
```

Note: The built-in function `failwith` raises an exception along with the error message passed to the function.

Then we write the simplest unit test we can imagine. For example, we know that the next weekday after Monday is Tuesday. So we add a test:

```
let tests = "test suite for next_weekday" >::: [
  "tue_after_mon"  >:: (fun _ -> assert_equal Tuesday (next_weekday Monday));
]
```

Then we run the OUnit test suite. It fails, as expected. That's good! Now we have a concrete goal, to make that unit test pass. We revise `next_weekday` to make that happen:

```
let next_weekday d =
  match d with
  | Monday -> Tuesday
  | _ -> failwith "Unimplemented"
```

We compile and run the test; it passes. Time to add some more tests. The simplest remaining possibilities are tests involving just weekdays, rather than weekends. So let's add tests for weekdays.

```
let tests = "test suite for next_weekday" >::: [
  "tue_after_mon"  >:: (fun _ -> assert_equal Tuesday (next_weekday Monday));
  "wed_after_tue"  >:: (fun _ -> assert_equal Wednesday (next_weekday Tuesday));
  "thu_after_wed"  >:: (fun _ -> assert_equal Thursday(next_weekday Wednesday));
  "fri_after_thu"  >:: (fun _ -> assert_equal Friday (next_weekday Thursday));
]
```

We compile and run the tests; many fail. That's good! We add new functionality:

```
  let next_weekday d =
    match d with
    | Monday -> Tuesday
    | Tuesday -> Wednesday
    | Wednesday -> Thursday
    | Thursday -> Friday
    | _ -> failwith "Unimplemented"
```

We compile and run the tests; they pass. At this point we could move on to handling weekends, but we should first notice something about the tests we've written: they involve repeating a lot of code. In fact, we probably wrote them by copying-and-pasting the first test, then modifying it for the next three. That's a sign that we should *refactor* the code. (As we did before with the `sum` function we were testing.)

Let's abstract a function that creates test cases for `next_weekday`:

```
let make_next_weekday_test name expected_output input=
  name >:: (fun _ -> assert_equal expected_output (next_weekday input))

let tests = "test suite for next_weekday" >::: [
  make_next_weekday_test "tue_after_mon" Tuesday Monday;
  make_next_weekday_test "wed_after_tue" Wednesday Tuesday;
  make_next_weekday_test "thu_after_wed" Thursday Wednesday;
  make_next_weekday_test "fri_after_thu" Friday Thursday;
]
```

Now we finish the testing and implementation by handling weekends. First we add some test cases:

```
  ...
  make_next_weekday_test "mon_after_fri" Monday Friday;
  make_next_weekday_test "mon_after_sat" Monday Saturday;
  make_next_weekday_test "mon_after_sun" Monday Sunday;
  ...
```

Then we finish the function:

```
let next_weekday d =
  match d with
  | Monday -> Tuesday
  | Tuesday -> Wednesday
  | Wednesday -> Thursday
  | Thursday -> Friday
  | Friday -> Monday
  | Saturday -> Monday
  | Sunday -> Monday
```

Of course, most people could write that function without errors even if they didn't use TDD. But rarely do we implement functions that are so simple.

Process. Let's review the process of TDD:

- Write a failing unit test case. Run the test suite to prove that the test case fails.
- Implement just enough functionality to make the test case pass. Run the test suite to prove that the test case passes.
- Improve code as needed. In the example above we refactored the test suite, but often we'll need to refactor the functionality being implemented.
- Repeat until you are satisfied that the test suite provides evidence that your implementation is correct.

5.4 Records and Tuples

Singly-linked lists are a great data structure, but what if you want a fixed number of elements, instead of an unbounded number? Or what if you want the elements to have distinct types? Or what if you want to access the elements by name instead of by number? Lists don't make any of those possibilities easy. Instead, OCaml programmers use records and tuples.

5.4.1 Records

A *record* is a composite of other types of data, each of which is named. OCaml records are much like structs in C. Here's an example of a record type definition `mon` for a Pokémon, re-using the `ptype` definition from the *variants* section:

```
type ptype = TNormal | TFire | TWater
type mon = {name : string; hp : int; ptype : ptype}
```

This type defines a record with three *fields* named `name`, `hp` (hit points), and `ptype`. The type of each of those fields is also given. Note that `ptype` can be used as both a type name and a field name; the *namespace* for those is distinct in OCaml.

To build a value of a record type, we write a record expression, which looks like this:

```
{name = "Charmander"; hp = 39; ptype = TFire}
```

So in a type definition we write a colon between the name and the type of a field, but in an expression we write an equals sign.

To access a record and get a field from it, we use the dot notation that you would expect from many other languages. For example:

```
let c = {name = "Charmander"; hp = 39; ptype = TFire};;
c.hp
```

It's also possible to use pattern matching to access record fields:

```
match c with {name = n; hp = h; ptype = t} -> h
```

The `n`, `h`, and `t` here are pattern variables. There is a syntactic sugar provided if you want to use the same name for both the field and a pattern variable:

```
match c with {name; hp; ptype} -> hp
```

Here, the pattern `{name; hp; ptype}` is sugar for `{name = name; hp = hp; ptype = ptype}`. In each of those subexpressions, the identifier appearing on the left-hand side of the equals is a field name, and the identifier appearing on the right-hand side is a pattern variable.

Syntax.

A record expression is written:

```
{f1 = e1; ...; fn = en}
```

The order of the `fi=ei` inside a record expression is irrelevant. For example, `{f = e1; g = e2}` is entirely equivalent to `{g = e2; f = e1}`.

A field access is written:

```
e.f
```

where `f` must be an identifier of a field name, not an expression. That restriction is the same as in any other language with similar features——for example, Java field names. If you really do want to *compute* which identifier to access, then actually you want a different data structure: a *map* (also known by many other names: a *dictionary* or *association list* or *hash table* etc., though there are subtle differences implied by each of those terms.)

Dynamic semantics.

- If for all `i` in `1..n`, it holds that `ei ==> vi`, then `{f1 = e1; ...; fn = en} ==> {f1 = v1; ...; fn = vn}`.
- If `e ==> {...; f = v; ...}` then `e.f ==> v`.

Static semantics.

A record type is written:

```
{f1 : t1; ...; fn : tn}
```

The order of the `fi:ti` inside a record type is irrelevant. For example, `{f : t1; g : t2}` is entirely equivalent to `{g:t2;f:t1}`.

Note that record types must be defined before they can be used. This enables OCaml to do better type inference than would be possible if record types could be used without definition.

The type checking rules are:

- If for all `i` in `1..n`, it holds that `ei : ti`, and if `t` is defined to be `{f1 : t1; ...; fn : tn}`, then `{f1 = e1; ...; fn = en} : t`. Note that the set of fields provided in a record expression must be the full set of fields defined as part of the record's type (but see below regarding *record copy*).
- If `e : t1` and if `t1` is defined to be `{...; f : t2; ...}`, then `e.f : t2`.

Record copy.

Another syntax is also provided to construct a new record out of an old record:

```
{e with f1 = e1; ...; fn = en}
```

This doesn't mutate the old record. Rather, it constructs a new record with new values. The set of fields provided after the `with` does not have to be the full set of fields defined as part of the record's type. In the newly-copied record, any field not provided as part of the `with` is copied from the old record.

Record copy is syntactic sugar. It's equivalent to writing

```
{ f1 = e1;   ...; fn = en;
  g1 = e.g1; ...; gn = e.gn }
```

where the set of `gi` is the set of all fields of the record's type minus the set of `fi`.

Pattern matching.

We add the following new pattern form to the list of legal patterns:

- `{f1 = p1; ...; fn = pn}`

And we extend the definition of when a pattern matches a value and produces a binding as follows:

- If for all `i` in `1..n`, it holds that `pi` matches `vi` and produces bindings b_i, then the record pattern `{f1 = p1; ...; fn = pn}` matches the record value `{f1 = v1; ...; fn = vn; ...}` and produces the set $\bigcup_i b_i$ of bindings. Note that the record value may have more fields than the record pattern does.

As a syntactic sugar, another form of record pattern is provided: `{f1; ...; fn}`. It is desugared to `{f1 = f1; ...; fn = fn}`.

5.4.2 Tuples

Like records, *tuples* are a composite of other types of data. But instead of naming the *components*, they are identified by position. Here are some examples of tuples:

```
(1, 2, 10)
(true, "Hello")
([1; 2; 3], (0.5, 'X'))
```

A tuple with two components is called a *pair*. A tuple with three components is called a *triple*. Beyond that, we usually just use the word *tuple* instead of continuing a naming scheme based on numbers.

Tip: Beyond about three components, it's arguably better to use records instead of tuples, because it becomes hard for a programmer to remember which component was supposed to represent what information.

Building of tuples is easy: just write the tuple, as above. Accessing again involves pattern matching, for example:

```
match (1, 2, 3) with (x, y, z) -> x + y + z
```

Syntax.

A tuple is written

```
(e1, e2, ..., en)
```

The parentheses are not entirely mandatory —often your code can successfully parse without them— but they are usually considered to be good style to include.

Dynamic semantics.

- If for all `i` in `1..n` it holds that `ei ==> vi`, then `(e1, ..., en) ==> (v1, ..., vn)`.

Static semantics.

Tuple types are written using a new type constructor `*`, which is different than the multiplication operator. The type `t1 * ... * tn` is the type of tuples whose first component has type `t1`, ..., and nth component has type `tn`.

- If for all `i` in `1..n` it holds that `ei : ti`, then `(e1, ..., en) : t1 * ... * tn`.

Pattern matching.

We add the following new pattern form to the list of legal patterns:

- `(p1, ..., pn)`

The parentheses are again not entirely mandatory but usually are idimoatic to include.

And we extend the definition of when a pattern matches a value and produces a binding as follows:

- If for all `i` in `1..n`, it holds that `pi` matches `vi` and produces bindings b_i, then the tuple pattern `(p1, ..., pn)` matches the tuple value `(v1, ..., vn)` and produces the set $\bigcup_i b_i$ of bindings. Note that the tuple value must have exactly the same number of components as the tuple pattern does.

5.4.3 Variants vs. Tuples and Records

Note: The second video above uses more advanced examples of variants that will be studied in a *later section*.

The big difference between variants and the types we just learned (records and tuples) is that a value of a variant type is *one of* a set of possibilities, whereas a value of a tuple or record type provides *each of* a set of possibilities. Going back to our examples, a value of type `day` is **one of** `Sun` or `Mon` or etc. But a value of type `mon` provides **each of** a `string` and an `int` and `ptype`. Note how, in those previous two sentences, the word "or" is associated with variant types, and the word "and" is associated with tuple and record types. That's a good clue if you're ever trying to decide whether you want to use a variant, or a tuple or record: if you need one piece of data *or* another, you want a variant; if you need one piece of data *and* another, you want a tuple or record.

One-of types are more commonly known as *sum types*, and each-of types as *product types*. Those names come from set theory. Variants are like disjoint union, because each value of a variant comes from one of many underlying sets (and thus far each of those sets is just a single constructor hence has cardinality one). Disjoint union is indeed sometimes written with a summation operator Σ. Tuples/records are like Cartesian product, because each value of a tuple or record contains a value from each of many underlying sets. Cartesian product is usually written with a product operator, $\times$ or Π.

5.5 Advanced Pattern Matching

Here are some additional pattern forms that are useful:

- `p1 | ... | pn`: an "or" pattern; matching against it succeeds if a match succeeds against any of the individual patterns `pi`, which are tried in order from left to right. All the patterns must bind the same variables.
- `(p : t)`: a pattern with an explicit type annotation.
- `c`: here, `c` means any constant, such as integer literals, string literals, and booleans.
- `'ch1'..'ch2'`: here, `ch` means a character literal. For example, `'A'..'Z'` matches any uppercase letter.
- `p when e`: matches `p` but only if `e` evaluates to `true`.

You can read about all the pattern forms in the manual.

5.5.1 Pattern Matching with Let

The syntax we've been using so far for let expressions is, in fact, a special case of the full syntax that OCaml permits. That syntax is:

```
let p = e1 in e2
```

That is, the left-hand side of the binding may in fact be a pattern, not just an identifier. Of course, variable identifiers are on our list of valid patterns, so that's why the syntax we've studied so far is just a special case.

Given this syntax, we revisit the semantics of let expressions.

Dynamic semantics.

To evaluate `let p = e1 in e2`:

1. Evaluate `e1` to a value `v1`.
2. Match `v1` against pattern `p`. If it doesn't match, raise the exception `Match_failure`. Otherwise, if it does match, it produces a set b of bindings.

3. Substitute those bindings b in `e2`, yielding a new expression `e2'`.
4. Evaluate `e2'` to a value `v2`.
5. The result of evaluating the let expression is `v2`.

Static semantics.

- If all the following hold then `(let p = e1 in e2) : t2`:
  - `e1 : t1`
  - the pattern variables in `p` are `x1..xn`
  - `e2 : t2` under the assumption that for all `i` in `1..n` it holds that `xi : ti`,

Let definitions.

As before, a let definition can be understood as a let expression whose body has not yet been given. So their syntax can be generalized to

```
let p = e
```

and their semantics follow from the semantics of let expressions, as before.

5.5.2 Pattern Matching with Functions

The syntax we've been using so far for functions is also a special case of the full syntax that OCaml permits. That syntax is:

```
let f p1 ... pn = e1 in e2    (* function as part of let expression *)
let f p1 ... pn = e           (* function definition at toplevel *)
fun p1 ... pn -> e            (* anonymous function *)
```

The truly primitive syntactic form we need to care about is `fun p -> e`. Let's revisit the semantics of anonymous functions and their application with that form; the changes to the other forms follow from those below:

Static semantics.

- Let `x1..xn` be the pattern variables appearing in `p`. If by assuming that `x1 : t1` and `x2 : t2` and … and `xn : tn`, we can conclude that `p : t` and `e :u`, then `fun p -> e : t -> u`.
- The type checking rule for application is unchanged.

Dynamic semantics.

- The evaluation rule for anonymous functions is unchanged.
- To evaluate `e0 e1`:
  1. Evaluate `e0` to an anonymous function `fun p -> e`, and evaluate `e1` to value `v1`.
  2. Match `v1` against pattern `p`. If it doesn't match, raise the exception `Match_failure`. Otherwise, if it does match, it produces a set b of bindings.
  3. Substitute those bindings b in `e`, yielding a new expression `e'`.
  4. Evaluate `e'` to a value `v`, which is the result of evaluating `e0 e1`.

5.5.3 Pattern Matching Examples

Here are several ways to get a Pokemon's hit points:

```
(* Pokemon types *)
type ptype = TNormal | TFire | TWater

(* A record to represent Pokemon *)
type mon = { name : string; hp : int; ptype : ptype }

(* OK *)
let get_hp m = match m with { name = n; hp = h; ptype = t } -> h

(* better *)
let get_hp m = match m with { name = _; hp = h; ptype = _ } -> h

(* better *)
let get_hp m = match m with { name; hp; ptype } -> hp

(* better *)
let get_hp m = match m with { hp } -> hp

(* best *)
let get_hp m = m.hp
```

Here's how to get the first and second components of a pair:

```
let fst (x, _) = x

let snd (_, y) = y
```

Both `fst` and `snd` are actually already defined for you in the standard library.

Finally, here are several ways to get the 3rd component of a triple:

```
(* OK *)
let thrd t = match t with x, y, z -> z

(* good *)
let thrd t =
  let x, y, z = t in
  z

(* better *)
let thrd t =
  let _, _, z = t in
  z

(* best *)
let thrd (_, _, z) = z
```

The standard library does not define any functions for triples, quadruples, etc.

5.6 Type Synonyms

A *type synonym* is a new name for an already existing type. For example, here are some type synonyms that might be useful in representing some types from linear algebra:

```
type point = float * float
type vector = float list
type matrix = float list list
```

Anywhere that a `float * float` is expected, you could use `point`, and vice-versa. The two are completely exchangeable for one another. In the following code, `get_x` doesn't care whether you pass it a value that is annotated as one vs. the other:

```
let get_x = fun (x, _) -> x

let p1 : point = (1., 2.)
let p2 : float * float = (1., 3.)

let a = get_x p1
let b = get_x p2
```

Type synonyms are useful because they let us give descriptive names to complex types. They are a way of making code more self-documenting.

5.7 Options

Suppose you want to write a function that *usually* returns a value of type `t`, but *sometimes* returns nothing. For example, you might want to define a function `list_max` that returns the maximum value in a list, but there's not a sensible thing to return on an empty list:

```
let rec list_max = function
  | [] -> ???
  | h :: t -> max h (list_max t)
```

There are a couple possibilities to consider:

- Return `min_int`? But then `list_max` will only work for integers— not floats or other types.
- Raise an exception? But then the user of the function has to remember to catch the exception.
- Return `null`? That works in Java, but by design OCaml does not have a `null` value. That's actually a good thing: null pointer bugs are not fun to debug.

Note: Sir Tony Hoare calls his invention of `null` a "billion-dollar mistake".

In addition to those possibilities, OCaml provides something even better called an *option*. (Haskellers will recognize options as the Maybe monad.)

You can think of an option as being like a closed box. Maybe there's something inside the box, or maybe box is empty. We don't know which until we open the box. If there turns out to be something inside the box when we open it, we can take that thing out and use it. Thus, options provide a kind of "maybe type," which ultimately is a kind of one-of type: the box is in one of two states, full or empty.

In `list_max` above, we'd like to metaphorically return a box that's empty if the list is empty, or a box that contains the maximum element of the list if the list is non empty.

Here's how we create an option that is like a box with `42` inside it:

```
Some 42
```

And here's how we create an option that is like an empty box:

```
None
```

The `Some` means there's something inside the box, and it's `42`. The `None` means there's nothing inside the box.

As for the types we see above, `t option` is a type for every type `t`, much like `t list` is a type for every type `t`. Values of type `t option` might contain a value of type `t`, or they might contain nothing. `None` has type `'a option` because it's unknown what the type is of the thing inside, as there isn't anything inside.

You can access the contents of an option value `e` using pattern matching. Here's a function that extracts an `int` from an option, if there is one inside, and converts it to a string:

```
let extract o =
  match o with
  | Some i -> string_of_int i
  | None -> "";;
```

And here are a couple of example usages of that function:

```
extract (Some 42);;
extract None;;
```

Here's how we can write `list_max` with options:

```
let rec list_max = function
  | [] -> None
  | h :: t -> begin
      match list_max t with
        | None -> Some h
        | Some m -> Some (max h m)
      end
```

Tip: The `begin`..`end` wrapping the nested pattern match above is not strictly required here but is not a bad habit, as it will head off potential syntax errors in more complicated code. The keywords `begin` and `end` are equivalent to `(` and `)`.

In Java, every object reference is implicitly an option. Either there is an object inside the reference, or there is nothing there. That "nothing" is represented by the value `null`. Java does not force programmers to explicitly check for the null case, which leads to null pointer exceptions. OCaml options force the programmer to include a branch in the pattern match for `None`, thus guaranteeing that the programmer thinks about the right thing to do when there's nothing there. So we can think of options as a principled way of eliminating `null` from the language. Using options is usually considered better coding practice than raising exceptions, because it forces the caller to do something sensible in the `None` case.

Syntax and semantics of options.

- `t option` is a type for every type `t`.
- `None` is a value of type `'a option`.

- `Some e` is an expression of type `t option` if `e : t`. If `e ==> v` then `Some e ==> Some v`

5.8 Association Lists

A *map* is a data structure that maps *keys* to *values*. Maps are also known as *dictionaries*. One easy implementation of a map is an *association list*, which is a list of pairs. Here, for example, is an association list that maps some shape names to the number of sides they have:

```
let d = [("rectangle", 4); ("nonagon", 9); ("icosagon", 20)]
```

Note that an association list isn't so much a built-in data type in OCaml as a combination of two other types: lists and pairs.

Here are two functions that implement insertion and lookup in an association list:

```
(** [insert k v lst] is an association list that binds key [k] to value [v]
    and otherwise is the same as [lst] *)
let insert k v lst = (k, v) :: lst

(** [find k lst] is [Some v] if association list [lst] binds key [k] to
    value [v]; and is [None] if [lst] does not bind [k]. *)
let rec lookup k = function
| [] -> None
| (k', v) :: t -> if k = k' then Some v else lookup k t
```

The `insert` function simply adds a new map from a key to a value at the front of the list. It doesn't bother to check whether the key is already in the list. The `lookup` function looks through the list from left to right. So if there did happen to be multiple maps for a given key in the list, only the most recently inserted one would be returned.

Insertion in an association list is therefore constant time, and lookup is linear time. Although there are certainly more efficient implementations of dictionaries—and we'll study some later in this course—association lists are a very easy and useful implementation for small dictionaries that aren't performance critical. The OCaml standard library has functions for association lists in the `List` module; look for `List.assoc` and the functions below it in the documentation. What we just wrote as `lookup` is actually already defined as `List.assoc_opt`. There is no pre-defined `insert` function in the library because it's so trivial just to cons a pair on.

5.9 Algebraic Data Types

Thus far, we have seen variants simply as enumerating a set of constant values, such as:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

type ptype = TNormal | TFire | TWater

type peff = ENormal | ENotVery | Esuper
```

But variants are far more powerful than this.

5.9.1 Variants that Carry Data

As a running example, here is a variant type `shape` that does more than just enumerate values:

```
type point = float * float
type shape =
  | Point of point
  | Circle of point * float (* center and radius *)
  | Rect of point * point (* lower-left and upper-right corners *)
```

This type, `shape`, represents a shape that is either a point, a circle, or a rectangle. A point is represented by a constructor `Point` that *carries* some additional data, which is a value of type `point`. A circle is represented by a constructor `Circle` that carries a pair of type `point * float`, which according to the comment represents the center of the circle and its radius. A rectangle is represented by a constructor `Rect` that carries a pair of type `point*point`.

Here are a couple functions that use the `shape` type:

```
let area = function
  | Point _ -> 0.0
  | Circle (_, r) -> Float.pi *. (r ** 2.0)
  | Rect ((x1, y1), (x2, y2)) ->
      let w = x2 -. x1 in
      let h = y2 -. y1 in
      w *. h

let center = function
  | Point p -> p
  | Circle (p, _) -> p
  | Rect ((x1, y1), (x2, y2)) -> ((x2 +. x1) /. 2.0, (y2 +. y1) /. 2.0)
```

The `shape` variant type is the same as those we've seen before in that it is defined in terms of a collection of constructors. What's different than before is that those constructors carry additional data along with them. Every value of type `shape` is formed from exactly one of those constructors. Sometimes we call the constructor a *tag*, because it tags the data it carries as being from that particular constructor.

Variant types are sometimes called *tagged unions*. Every value of the type is from the set of values that is the union of all values from the underlying types that the constructor carries. For the `shape` type, every value is tagged with either `Point` or `Circle` or `Rect` and carries a value from the set of all `point` valued unioned with the set of all `point * float` values unioned with the set of all `point * point` values.

Another name for these variant types is an *algebraic data type*. "Algebra" here refers to the fact that variant types contain both sum and product types, as defined in the previous lecture. The sum types come from the fact that a value of a variant is formed by *one of* the constructors. The product types come from that fact that a constructor can carry tuples or records, whose values have a sub-value from *each of* their component types.

Using variants, we can express a type that represents the union of several other types, but in a type-safe way. Here, for example, is a type that represents either a `string` or an `int`:

```
type string_or_int =
  | String of string
  | Int of int
```

If we wanted to, we could use this type to code up lists (e.g.) that contain either strings or ints:

```
type string_or_int_list = string_or_int list
```

(continues on next page)

(continued from previous page)

```
let rec sum : string_or_int list -> int = function
  | [] -> 0
  | String s :: t -> int_of_string s + sum t
  | Int i :: t -> i + sum t

let lst_sum = sum [String "1"; Int 2]
```

Variants thus provide a type-safe way of doing something that might before have seemed impossible.

Variants also make it possible to discriminate which tag a value was constructed with, even if multiple constructors carry the same type. For example:

```
type t = Left of int | Right of int
let x = Left 1
let double_right = function
  | Left i -> i
  | Right i -> 2 * i
```

5.9.2 Syntax and Semantics

Syntax.

To define a variant type:

```
type t = C1 [of t1] | ... | Cn [of tn]
```

The square brackets above denote that `of ti` is optional. Every constructor may individually either carry no data or carry data. We call constructors that carry no data *constant*; and those that carry data, *non-constant*.

To write an expression that is a variant:

```
C e
```

Or:

```
C
```

depending on whether the constructor name `C` is non-constant or constant.

Dynamic semantics.

- If `e==>v` then `C e ==> C v`, assuming `C` is non-constant.
- `C` is already a value, assuming `C` is constant.

Static semantics.

- If `t = ... | C | ...` then `C : t`.
- If `t = ... | C of t' | ...` and if `e : t'` then `C e : t`.

Pattern matching.

We add the following new pattern form to the list of legal patterns:

- `C p`

And we extend the definition of when a pattern matches a value and produces a binding as follows:

- If `p` matches `v` and produces bindings b, then `C p` matches `C v` and produces bindings b.

5.9.3 Catch-all Cases

One thing to beware of when pattern matching against variants is what *Real World OCaml* calls "catch-all cases". Here's a simple example of what can go wrong. Let's suppose you write this variant and function:

```
type color = Blue | Red

(* a thousand lines of code in between *)

let string_of_color = function
  | Blue -> "blue"
  | _ -> "red"
```

Seems fine, right? But then one day you realize there are more colors in the world. You need to represent green. So you go back and add green to your variant:

```
type color = Blue | Red | Green

(* a thousand lines of code in between *)

let string_of_color = function
  | Blue -> "blue"
  | _ -> "red"
```

But because of the thousand lines of code in between, you forget that `string_of_color` needs updating. And now, all the sudden, you are red-green color blind:

```
string_of_color Green
```

The problem is the *catch-all* case in the pattern match inside `string_of_color`: the final case that uses the wildcard pattern to match anything. Such code is not robust against future changes to the variant type.

If, instead, you had originally coded the function as follows, life would be better:

```
let string_of_color = function
  | Blue -> "blue"
  | Red  -> "red"
```

The OCaml type checker now alerts you that you haven't yet updated `string_of_color` to account for the new constructor.

The moral of the story is: catch-all cases lead to buggy code. Avoid using them.

5.9.4 Recursive Variants

Variant types may mention their own name inside their own body. For example, here is a variant type that could be used to represent something similar to `int list`:

```
type intlist = Nil | Cons of int * intlist

let lst3 = Cons (3, Nil)  (* similar to 3 :: [] or [3]*)
let lst123 = Cons(1, Cons(2, lst3)) (* similar to [1; 2; 3] *)

let rec sum (l : intlist) : int=
  match l with
  | Nil -> 0
  | Cons (h, t) -> h + sum t

let rec length : intlist -> int = function
  | Nil -> 0
  | Cons (_, t) -> 1 + length t

let empty : intlist -> bool = function
  | Nil -> true
  | Cons _ -> false
```

Notice that in the definition of `intlist`, we define the `Cons` constructor to carry a value that contains an `intlist`. This makes the type `intlist` be *recursive*: it is defined in terms of itself.

Types may be mutually recursive if you use the `and` keyword:

```
type node = {value : int; next : mylist}
and mylist = Nil | Node of node
```

Any such mutual recursion must involve at least one variant or record type that the recursion "goes through". For example, the following is not allowed:

```
type t = u and u = t
```

But this is:

```
type t = U of u and u = T of t
```

Record types may also be recursive:

```
type node = {value : int; next : node}
```

But plain old type synonyms may not be:

```
type t = t * t
```

Although `node` is a legal type definition, there is no way to construct a value of that type because of the circularity involved: to construct the very first `node` value in existence, you would already need a value of type `node` to exist. Later, when we cover imperative features, we'll see a similar idea used (but successfully) for mutable linked lists.

5.9.5 Parameterized Variants

Variant types may be *parameterized* on other types. For example, the `intlist` type above could be generalized to provide lists (coded up ourselves) over any type:

```
type 'a mylist = Nil | Cons of 'a * 'a mylist

let lst3 = Cons (3, Nil)   (* similar to [3] *)
let lst_hi = Cons ("hi", Nil)   (* similar to ["hi"] *)
```

Here, `mylist` is a *type constructor* but not a type: there is no way to write a value of type `mylist`. But we can write value of type `int mylist` (e.g., `lst3`) and `string mylist` (e.g., `lst_hi`). Think of a type constructor as being like a function, but one that maps types to types, rather than values to value.

Here are some functions over `'a mylist`:

```
let rec length : 'a mylist -> int = function
  | Nil -> 0
  | Cons (_, t) -> 1 + length t

let empty : 'a mylist -> bool = function
  | Nil -> true
  | Cons _ -> false
```

Notice that the body of each function is unchanged from its previous definition for `intlist`. All that we changed was the type annotation. And that could even be omitted safely:

```
let rec length = function
  | Nil -> 0
  | Cons (_, t) -> 1 + length t

let empty = function
  | Nil -> true
  | Cons _ -> false
```

The functions we just wrote are an example of a language feature called **parametric polymorphism**. The functions don't care what the `'a` is in `'a mylist`, hence they are perfectly happy to work on `int mylist` or `string mylist` or any other `(whatever) mylist`. The word "polymorphism" is based on the Greek roots "poly" (many) and "morph" (form). A value of type `'a mylist` could have many forms, depending on the actual type `'a`.

As soon, though, as you place a constraint on what the type `'a` might be, you give up some polymorphism. For example,

```
let rec sum = function
  | Nil -> 0
  | Cons (h, t) -> h + sum t
```

The fact that we use the `( + )` operator with the head of the list constrains that head element to be an `int`, hence all elements must be `int`. That means `sum` must take in an `int mylist`, not any other kind of `'a mylist`.

It is also possible to have multiple type parameters for a parameterized type, in which case parentheses are needed:

```
type ('a, 'b) pair = {first : 'a; second : 'b}
let x = {first = 2; second = "hello"}
```

5.9.6 Polymorphic Variants

Thus far, whenever you've wanted to define a variant type, you have had to give it a name, such as `day`, `shape`, or `'a mylist`:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

type shape =
  | Point of point
  | Circle of point * float
  | Rect of point * point

type 'a mylist = Nil | Cons of 'a * 'a mylist
```

Occasionally, you might need a variant type only for the return value of a single function. For example, here's a function `f` that can either return an `int` or ∞; you are forced to define a variant type to represent that result:

```
type fin_or_inf = Finite of int | Infinity

let f = function
  | 0 -> Infinity
  | 1 -> Finite 1
  | n -> Finite (-n)
```

The downside of this definition is that you were forced to defined `fin_or_inf` even though it won't be used throughout much of your program.

There's another kind of variant in OCaml that supports this kind of programming: *polymorphic variants*. Polymorphic variants are just like variants, except:

1. You don't have declare their type or constructors before using them.
2. There is no name for a polymorphic variant type. (So another name for this feature could have been "anonymous variants".)
3. The constructors of a polymorphic variant start with a backquote character.

Using polymorphic variants, we can rewrite `f`:

```
let f = function
  | 0 -> `Infinity
  | 1 -> `Finite 1
  | n -> `Finite (-n)
```

This type says that `f` either returns `` `Finite n `` for some `n : int` or `` `Infinity ``. The square brackets do not denote a list, but rather a set of possible constructors. The > sign means that any code that pattern matches against a value of that type must *at least* handle the constructors `` `Finite `` and `` `Infinity ``, and possibly more. For example, we could write:

```
match f 3 with
  | `NegInfinity -> "negative infinity"
  | `Finite n -> "finite"
  | `Infinity -> "infinite"
```

It's perfectly fine for the pattern match to include constructors other than `` `Finite `` or `` `Infinity ``, because `f` is guaranteed never to return any constructors other than those.

There are other, more compelling uses for polymorphic variants that we'll see later in the course. They are particularly useful in libraries. For now, we generally will steer you away from extensive use of polymorphic variants, because their types can become difficult to manage.

5.9.7 Built-in Variants

OCaml's built-in list data type is really a recursive, parameterized variant. It is defined as follows:

```
type 'a list = [] | ( :: ) of 'a * 'a list
```

So `list` is really just a type constructor, with (value) constructors `[]` (which we pronounce "nil") and `::` (which we pronounce "cons").

OCaml's built-in option data type is also really a parameterized variant. It's defined as follows:

```
type 'a option = None | Some of 'a
```

So `option` is really just a type constructor, with (value) constructors `None` and `Some`.

You can see both `list` and `option` defined in the core OCaml library.

5.10 Exceptions

OCaml has an exception mechanism similar to many other programming languages. A new type of OCaml exception is defined with this syntax:

```
exception E of t
```

where `E` is a constructor name and `t` is a type. The `of t` is optional. Notice how this is similar to defining a constructor of a variant type. For example:

```
exception A
exception B
exception Code of int
exception Details of string
```

To create an exception value, use the same syntax you would for creating a variant value. Here, for example, is an exception value whose constructor is `Failure`, which carries a `string`:

```
Failure "something went wrong"
```

This constructor is pre-defined in the standard library and is one of the more common exceptions that OCaml programmers use.

To raise an exception value `e`, simply write

```
raise e
```

There is a convenient function `failwith : string -> 'a` in the standard library that raises `Failure`. That is, `failwith s` is equivalent to `raise (Failure s)`.

To catch an exception, use this syntax:

```
try e with
| p1 -> e1
| ...
| pn -> en
```

The expression `e` is what might raise an exception. If it does not, the entire `try` expression evaluates to whatever `e` does. If `e` does raise an exception value `v`, that value `v` is that matched against the provide patterns, exactly like `match` expression.

5.10.1 Exceptions are Extensible Variants

All exception values have type `exn`, which is a variant defined in the core. It's an unusual kind of variant, though, called an *extensible* variant, which allows new constructors of the variant to be defined after the variant type itself is defined. See the OCaml manual for more information about extensible variants if you're interested.

5.10.2 Exception Semantics

Since they are just variants, the syntax and semantics of exceptions is already covered by the syntax and semantics of variants—with one exception (pun intended), which is the dynamic semantics of how exceptions are raised and handled.

Dynamic semantics. As we originally said, every OCaml expression either

- evaluates to a value
- raises an exception
- or fails to terminate (i.e., an "infinite loop").

So far we've only presented the part of the dynamic semantics that handles the first of those three cases. What happens when we add exceptions? Now, evaluation of an expression either produces a value or produces an *exception packet*. Packets are not normal OCaml values; the only pieces of the language that recognizes them are `raise` and `try`. The exception value produced by (e.g.) `Failure "oops"` is part of the exception packet produced by `raise (Failure "oops")`, but the packet contains more than just the exception value; there can also be a stack trace, for example.

For any expression `e` other than `try`, if evaluation of a subexpression of `e` produces an exception packet `P`, then evaluation of `e` produces packet `P`.

But now we run into a problem for the first time: what order are subexpressions evaluated in? Sometimes the answer to that question is provided by the semantics we have already developed. For example, with let expressions, we know that the binding expression must be evaluated before the body expression. So the following code raises `A`:

```
let _ = raise A in raise B;;
```

And with functions, the argument must be evaluated before the function. So the following code also raises `A`, in addition to producing some compiler warnings that the first expression will never actually be applied as a function to an argument:

```
(raise B) (raise A)
```

It makes sense that both those pieces of code would raise the same exception, given that we know `let x = e1 in e2` is syntactic sugar for `(fun x -> e2) e1`.

But what does the following code raise as an exception?

```
(raise A, raise B)
```

The answer is nuanced. The language specification does not stipulate what order the components of pairs should be evaluated in. Nor did our semantics exactly determine the order. (Though you would be forgiven if you thought it was left to right.) So programmers actually cannot rely on that order. The current implementation of OCaml, as it turns out, evaluates right to left. So the code above actually raises `B`. If you really want to force the evaluation order, you need to use let expressions:

```
let a = raise A in
let b = raise B in
(a, b)
```

That code is guaranteed to raise `A` rather than `B`.

One interesting corner case is what happens when a raise expression itself has a subexpression that raises:

```
exception C of string;;
exception D of string;;
raise (C (raise (D "oops")))
```

That code ends up raising `D`, because the first thing that has to happen is to evaluate `C (raise (D "oops"))` to a value. Doing that requires evaluating `raise (D "oops")` to a value. Doing that causes a packet containing `D "oops"` to be produced, and that packet then propagates and becomes the result of evaluating `C (raise (D "oops"))`, hence the result of evaluating `raise (C (raise (D "oops")))`.

Once evaluation of an expression produces an exception packet `P`, that packet propagates until it reaches a `try` expression:

```
try e with
| p1 -> e1
| ...
| pn -> en
```

The exception value inside `P` is matched against the provided patterns using the usual evaluation rules for pattern matching—with one exception (again, pun intended). If none of the patterns matches, then instead of producing `Match_failure` inside a new exception packet, the original exception packet `P` continues propagating until the next `try` expression is reached.

5.10.3 Pattern Matching

There is a pattern form for exceptions. Here's an example of its usage:

```
match List.hd [] with
  | [] -> "empty"
  | _ :: _ -> "nonempty"
  | exception (Failure s) -> s
```

Note that the code is above is just a standard `match` expression, not a `try` expression. It matches the value of `List.hd []` against the three provided patterns. As we know, `List.hd []` will raise an exception containing the value `Failure "hd"`. The *exception pattern* `exception (Failure s)` matches that value. So the above code will evaluate to `"hd"`.

In general, exception patterns are a kind of syntactic sugar. Consider this code:

```
match e with
  | p1 -> e1
  | ...
  | pn -> en
```

Some of the patterns `p1..pn` could be exception patterns of the form `exception q`. Let `q1..qn` be that subsequence of patterns (without the `exception` keyword), and let `r1..rm` be the subsequence of non-exception patterns. Then we can rewrite the code as:

```
match
  try e with
    | q1 -> e1
    | ...
    | qn -> en
with
  | r1 -> e1
  | ...
  | rm -> em
```

Which is to say: try evaluating `e`. If it produces an exception packet, use the exception patterns from the original match expression to handle that packet. If it doesn't produce an exception packet but instead produces a non-exception value, use the non-exception patterns from the original match expression to match that value.

5.10.4 Exceptions and OUnit

If it is part of a function's specification that it raises an exception, you might want to write OUnit tests that check whether the function correctly does so. Here's how to do that:

```
open OUnit2

let tests = "suite" >::: [
    "empty" >:: (fun _ -> assert_raises (Failure "hd") (fun () -> List.hd []));
  ]

let _ = run_test_tt_main tests
```

The expression `assert_raises exc (fun () -> e)` checks to see whether expression `e` raises exception `exc`. If so, the OUnit test case succeeds, otherwise it fails.

Tip: A common error is to forget the `(fun () -> ...)` around `e`. If you do, the OUnit test case will fail, and you will likely be confused as to why. The reason is that, without the extra anonymous function, the exception is raised before `assert_raises` ever gets a chance to handle it.

5.11 Example: Trees

Trees are a very useful data structure. A *binary tree*, as you'll recall from CS 2110, is a node containing a value and two children that are trees. A binary tree can also be an empty tree, which we also use to represent the absence of a child node.

5.11.1 Representation with Tuples

Here is a definition for a binary tree data type:

```
type 'a tree =
| Leaf
| Node of 'a * 'a tree * 'a tree
```

A node carries a data item of type `'a` and has a left and right subtree. A leaf is empty. Compare this definition to the definition of a list and notice how similar their structure is:

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

```
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist
```

The only essential difference is that `Cons` carries one sublist, whereas `Node` carries two subtrees.

Here is code that constructs a small tree:

```
(* the code below constructs this tree:
         4
       /   \
      2     5
     / \   / \
    1   3 6   7
*)
let t =
  Node(4,
    Node(2,
      Node(1, Leaf, Leaf),
      Node(3, Leaf, Leaf)
    ),
    Node(5,
      Node(6, Leaf, Leaf),
      Node(7, Leaf, Leaf)
    )
  )
```

The *size* of a tree is the number of nodes in it (that is, `Node`s, not `Leaf`s). For example, the size of tree `t` above is 7. Here is a function function `size : 'a tree -> int` that returns the number of nodes in a tree:

```
let rec size = function
  | Leaf -> 0
  | Node (_, l, r) -> 1 + size l + size r
```

5.11.2 Representation with Records

Next, let's revise our tree type to use use a record type to represent a tree node. In OCaml we have to define two mutually recursive types, one to represent a tree node, and one to represent a (possibly empty) tree:

```
type 'a tree =
  | Leaf
  | Node of 'a node
```

(continues on next page)

(continued from previous page)

```
and 'a node = {
  value: 'a;
  left: 'a tree;
  right: 'a tree
}
```

Here's an example tree:

```
(* represents
      2
     / \
    1   3  *)
let t =
  Node {
    value = 2;
    left = Node {value = 1; left = Leaf; right = Leaf};
    right = Node {value = 3; left = Leaf; right = Leaf}
  }
```

We can use pattern matching to write the usual algorithms for recursively traversing trees. For example, here is a recursive search over the tree:

```
(** [mem x t] is whether [x] is a value at some node in tree [t]. *)
let rec mem x = function
  | Leaf -> false
  | Node {value; left; right} -> value = x || mem x left || mem x right
```

The function name `mem` is short for "member"; the standard library often uses a function of this name to implement a search through a collection data structure to determine whether some element is a member of that collection.

Here's a function that computes the *preorder* traversal of a tree, in which each node is visited before any of its children, by constructing a list in which the values occur in the order in which they would be visited:

```
let rec preorder = function
  | Leaf -> []
  | Node {value; left; right} -> [value] @ preorder left @ preorder right
```

```
preorder t
```

Although the algorithm is beautifully clear from the code above, it takes quadratic time on unbalanced trees because of the `@` operator. That problem can be solved by introducing an extra argument `acc` to accumulate the values at each node, though at the expense of making the code less clear:

```
let preorder_lin t =
  let rec pre_acc acc = function
    | Leaf -> acc
    | Node {value; left; right} -> value :: (pre_acc (pre_acc acc right) left)
  in pre_acc [] t
```

The version above uses exactly one `::` operation per `Node` in the tree, making it linear time.

5.12 Example: Natural Numbers

We can define a recursive variant that acts like numbers, demonstrating that we don't really have to have numbers built into OCaml! (For sake of efficiency, though, it's a good thing they are.)

A *natural number* is either *zero* or the *successor* of some other natural number. This is how you might define the natural numbers in a mathematical logic course, and it leads naturally to the following OCaml type `nat`:

```
type nat = Zero | Succ of nat
```

We have defined a new type `nat`, and `Zero` and `Succ` are constructors for values of this type. This allows us to build expressions that have an arbitrary number of nested `Succ` constructors. Such values act like natural numbers:

```
let zero = Zero
let one = Succ zero
let two = Succ one
let three = Succ two
let four = Succ three
```

Now we can write functions to manipulate values of this type. We'll write a lot of type annotations in the code below to help the reader keep track of which values are `nat` versus `int`; the compiler, of course, doesn't need our help.

```
let iszero = function
  | Zero -> true
  | Succ _ -> false

let pred = function
  | Zero -> failwith "pred Zero is undefined"
  | Succ m -> m
```

Similarly we can define a function to add two numbers:

```
let rec add n1 n2 =
  match n1 with
  | Zero -> n2
  | Succ pred_n -> add pred_n (Succ n2)
```

We can convert `nat` values to type `int` and vice-versa:

```
let rec int_of_nat = function
  | Zero -> 0
  | Succ m -> 1 + int_of_nat m

let rec nat_of_int = function
  | i when i = 0 -> Zero
  | i when i > 0 -> Succ (nat_of_int (i - 1))
  | _ -> failwith "nat_of_int is undefined on negative ints"
```

To determine whether a natural number is even or odd, we can write a pair of mutually recursive functions:

```
let rec even = function Zero -> true | Succ m -> odd m
and odd = function Zero -> false | Succ m -> even m
```

5.13 Summary

Lists are a highly useful built-in data structure in OCaml. The language provides a lightweight syntax for building them, rather than requiring you to use a library. Accessing parts of a list makes use of pattern matching, a very powerful feature (as you might expect from its rather lengthy semantics). We'll see more uses for pattern matching as the course proceeds.

These built-in lists are implemented as singly-linked lists. That's important to keep in mind when your needs go beyond small to medium sized lists. Recursive functions on long lists will take up a lot of stack space, so tail recursion becomes important. And if you're attempting to process really huge lists, you probably don't want linked lists at all, but instead a data structure that will do a better job of exploiting memory locality.

OCaml provides data types for variants (one-of types), tuples and products (each-of types), and options (maybe types). Pattern matching can be used to access values of each of those data types. And pattern matching can be used in let expressions and functions.

Association lists combine lists and tuples to create a lightweight implementation of dictionaries.

Variants are a powerful language feature. They are the workhorse of representing data in a functional language. OCaml variants actually combine several theoretically independent language features into one: sum types, product types, recursive types, and parameterized (polymorphic) types. The result is an ability to express many kinds of data, including lists, options, trees, and even exceptions.

5.13.1 Terms and Concepts

- algebraic data type
- append
- association list
- binary trees as variants
- binding
- branch
- carried data
- catch-all cases
- cons
- constant constructor
- constructor
- copying
- desugaring
- each-of type
- exception
- exception as variants
- exception packet
- exception pattern
- exception value
- exhaustiveness

- field
- head
- induction
- leaf
- list
- lists as variants
- maybe type
- mutually recursive functions
- natural numbers as variants
- nil
- node
- non-constant constructor
- one-of type
- options
- options as variants
- order of evaluation
- pair
- parameterized variant
- parametric polymorphism
- pattern matching
- prepend
- product type
- record
- recursion
- recursive variant
- sharing
- stack frame
- sum type
- syntactic sugar
- tag
- tail
- tail call
- tail recursion
- test-driven development (TDD)
- triple
- tuple

- type constructor
- type synonym
- variant
- wildcard

5.13.2 Further Reading

- *Introduction to Objective Caml*, chapters 4, 5.2, 5.3, 5.4, 6, 7, 8.1
- *OCaml from the Very Beginning*, chapters 3, 4, 5, 7, 8, 10, 11
- *Real World OCaml*, chapter 3, 5, 6, 7

5.14 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Exercise: list expressions [★]

- Construct a list that has the integers 1 through 5 in it. Use the square bracket notation for lists.
- Construct the same list, but do not use the square bracket notation. Instead use `::` and `[]`.
- Construct the same list again. This time, the following expression must appear in your answer: `[2; 3; 4]`. Use the `@` operator, and do not use `::`.

Exercise: product [★★]

Write a function that returns the product of all the elements in a list. The product of all the elements of an empty list is `1`.

Exercise: concat [★★]

Write a function that concatenates all the strings in a list. The concatenation of all the strings in an empty list is the empty string `""`.

Exercise: product test [★★]

Unit test the function `product` that you wrote in an exercise above.

Exercise: patterns [★★★]

Using pattern matching, write three functions, one for each of the following properties. Your functions should return `true` if the input list has the property and `false` otherwise.

- the list's first element is `"bigred"`
- the list has exactly two or four elements; do not use the `length` function
- the first two elements of the list are equal

Exercise: library [★★★]

Consult the `List` standard library to solve these exercises:

- Write a function that takes an `int list` and returns the fifth element of that list, if such an element exists. If the list has fewer than five elements, return `0`. *Hint: `List.length` and `List.nth`.*
- Write a function that takes an `int list` and returns the list sorted in descending order. *Hint: `List.sort` with `Stdlib.compare` as its first argument, and `List.rev`.*

Exercise: library test [★★★]

Write a couple OUnit unit tests for each of the functions you wrote in the previous exercise.

Exercise: library puzzle [★★★]

- Write a function that returns the last element of a list. Your function may assume that the list is non-empty. *Hint: Use two library functions, and do not write any pattern matching code of your own.*
- Write a function `any_zeroes : int list -> bool` that returns `true` if and only if the input list contains at least one `0`. *Hint: use one library function, and do not write any pattern matching code of your own.*

Your solutions will be only one or two lines of code each.

Exercise: take drop [★★★]

- Write a function `take : int -> 'a list -> 'a list` such that `take n lst` returns the first `n` elements of `lst`. If `lst` has fewer than `n` elements, return all of them.
- Write a function `drop : int -> 'a list -> 'a list` such that `drop n lst` returns all but the first `n` elements of `lst`. If `lst` has fewer than `n` elements, return the empty list.

Exercise: take drop tail [★★★★]

Revise your solutions for `take` and `drop` to be tail recursive, if they aren't already. Test them on long lists with large values of `n` to see whether they run out of stack space. To construct long lists, use the `--` operator from the *lists* section.

Exercise: unimodal [★★★]

Write a function `is_unimodal : int list -> bool` that takes an integer list and returns whether that list is unimodal. A *unimodal list* is a list that monotonically increases to some maximum value then monotonically decreases after that value. Either or both segments (increasing or decreasing) may be empty. A constant list is unimodal, as is the empty list.

Exercise: powerset [★★★]

Write a function `powerset : int list -> int list list` that takes a set S represented as a list and returns the set of all subsets of S. The order of subsets in the powerset and the order of elements in the subsets do not matter.

Hint: Consider the recursive structure of this problem. Suppose you already have `p`, such that `p = powerset s`. How could you use `p` to compute `powerset (x :: s)`?

Exercise: print int list rec [★★]

Write a function `print_int_list : int list -> unit` that prints its input list, one number per line. For example, `print_int_list [1; 2; 3]` should result in this output:

```
1
2
3
```

Here is some code to get you started:

```
let rec print_int_list = function
| [] -> ()
| h :: t -> (* fill in here *); print_int_list t
```

Exercise: print int list iter [★★]

Write a function `print_int_list' : int list -> unit` whose specification is the same as `print_int_list`. Do not use the keyword `rec` in your solution, but instead to use the List module function `List.iter`. Here is some code to get you started:

```
let print_int_list' lst =
  List.iter (fun x -> (* fill in here *)) lst
```

Exercise: student [★★]

Assume the following type definition:

```
type student = {first_name : string; last_name : string; gpa : float}
```

Give OCaml expressions that have the following types:

- `student`
- `student -> string * string` (a function that extracts the student's name)
- `string -> string -> float -> student` (a function that creates a student record)

Exercise: pokerecord [★★]

Here is a variant that represents a few Pokémon types:

```
type poketype = Normal | Fire | Water
```

- Define the type `pokemon` to be a record with fields `name` (a string), `hp` (an integer), and `ptype` (a `poketype`).
- Create a record named `charizard` of type `pokemon` that represents a Pokémon with 78 HP and Fire type.
- Create a record named `squirtle` of type `pokemon` that represents a Pokémon with 44 HP and Water type.

Exercise: safe hd and tl [★★]

Write a function `safe_hd : 'a list -> 'a option` that returns `Some x` if the head of the input list is `x`, and `None` if the input list is empty.

Also write a function `safe_tl : 'a list -> 'a list option` that returns the tail of the list, or `None` if the list is empty.

Exercise: pokefun [★★★]

Write a function `max_hp : pokemon list -> pokemon option` that, given a list of `pokemon`, finds the Pokémon with the highest HP.

Exercise: date before [★★]

Define a *date-like triple* to be a value of type `int * int * int`. Examples of date-like triples include `(2013, 2, 1)` and `(0, 0, 1000)`. A *date* is a date-like triple whose first part is a positive year (i.e., a year in the common era), second part is a month between 1 and 12, and third part is a day between 1 and 31 (or 30, 29, or 28, depending on the month and year). `(2013, 2, 1)` is a date; `(0, 0, 1000)` is not.

Write a function `is_before` that takes two dates as input and evaluates to `true` or `false`. It evaluates to `true` if the first argument is a date that comes before the second argument. (If the two dates are the same, the result is `false`.)

Your function needs to work correctly only for dates, not for arbitrary date-like triples. However, you will probably find it easier to write your solution if you think about making it work for arbitrary date-like triples. For example, it's easier to forget about whether the input is truly a date, and simply write a function that claims (for example) that January 100, 2013 comes before February 34, 2013—because any date in January comes before any date in February, but a function that says that January 100, 2013 comes after February 34, 2013 is also valid. You may ignore leap years.

Exercise: earliest date [★★★]

Write a function `earliest : (int*int*int) list -> (int * int * int) option`. It evaluates to `None` if the input list is empty, and to `Some d` if date `d` is the earliest date in the list. *Hint: use `is_before`.*

As in the previous exercise, your function needs to work correctly only for dates, not for arbitrary date-like triples.

Exercise: assoc list [★]

Use the functions `insert` and `lookup` from the *section on association lists* to construct an association list that maps the integer 1 to the string "one", 2 to "two", and 3 to "three". Lookup the key 2. Lookup the key 4.

Exercise: cards [★★]

- Define a variant type `suit` that represents the four suits, ♣ ♦ ♥ ♠, in a standard 52-card deck. All the constructors of your type should be constant.
- Define a type `rank` that represents the possible ranks of a card: 2, 3, …, 10, Jack, Queen, King, or Ace. There are many possible solutions; you are free to choose whatever works for you. One is to make `rank` be a synonym of `int`, and to assume that Jack=11, Queen=12, King=13, and Ace=1 or 14. Another is to use variants.
- Define a type `card` that represents the suit and rank of a single card. Make it a record with two fields.

- Define a few values of type `card`: the Ace of Clubs, the Queen of Hearts, the Two of Diamonds, the Seven of Spades.

Exercise: matching [★]

For each pattern in the list below, give a value of type `int option list` that does *not* match the pattern and is not the empty list, or explain why that's impossible.

- `Some x :: tl`
- `[Some 3110; None]`
- `[Some x; _]`
- `h1 :: h2 :: tl`
- `h :: tl`

Exercise: quadrant [★★]

Complete the `quadrant` function below, which should return the quadrant of the given `x, y` point according to the diagram on the right (borrowed from Wikipedia). Points that lie on an axis do not belong to any quandrant. *Hints: (a) define a helper function for the sign of an integer, (b) match against a pair.*

```
type quad = I | II | III | IV
type sign = Neg | Zero | Pos

let sign (x:int) : sign =
  ...

let quadrant : int*int -> quad option = fun (x,y) ->
  match ... with
    | ... -> Some I
    | ... -> Some II
    | ... -> Some III
    | ... -> Some IV
    | ... -> None
```

Exercise: quadrant when [★★]

Rewrite the quadrant function to use the `when` syntax. You won't need your helper function from before.

```
let quadrant_when : int*int -> quad option = function
    | ... when ... -> Some I
    | ... when ... -> Some II
    | ... when ... -> Some III
    | ... when ... -> Some IV
    | ... -> None
```

Exercise: depth [★★]

Write a function `depth : 'a tree -> int` that returns the number of nodes in any longest path from the root to a leaf. For example, the depth of an empty tree (simply `Leaf`) is `0`, and the depth of tree `t` above is `3`. *Hint: there is a library function `max : 'a -> 'a -> 'a` that returns the maximum of any two values of the same type.*

Exercise: shape [★★★]

Write a function `same_shape : 'a tree -> 'b tree -> bool` that determines whether two trees have the same shape, regardless of whether the values they carry at each node are the same. *Hint: use a pattern match with three branches, where the expression being matched is a pair of trees.*

Exercise: list max exn [★★]

Write a function `list_max : int list -> int` that returns the maximum integer in a list, or raises `Failure "list_max"` if the list is empty.

Exercise: list max exn string [★★]

Write a function `list_max_string : int list -> string` that returns a string containing the maximum integer in a list, or the string `"empty"` (note, not the exception `Failure "empty"` but just the string `"empty"`) if the list is empty. *Hint: `string_of_int` in the standard library will do what its name suggests.*

Exercise: list max exn ounit [★]

Write two OUnit tests to determine whether your solution to **list max exn**, above, correctly raises an exception when its input is the empty list, and whether it correctly returns the max value of the input list when that list is nonempty.

Exercise: is_bst [★★★★]

Write a function `is_bst : ('a*'b) tree -> bool` that returns `true` if and only if the given tree satisfies the binary search tree invariant. An efficient version of this function that visits each node at most once is somewhat tricky to write. *Hint: write a recursive helper function that takes a tree and either gives you (i) the minimum and maximum value in the tree, or (ii) tells you that the tree is empty, or (iii) tells you that the tree does not satisfy the invariant. Your `is_bst` function will not be recursive, but will call your helper function and pattern match on the result. You will need to define a new variant type for the return type of your helper function.*

Exercise: quadrant poly [★★]

Modify your definition of quadrant to use polymorphic variants. The types of your functions should become these:

```
val sign : int -> [> `Neg | `Pos | `Zero ]
val quadrant : int * int -> [> `I | `II | `III | `IV ] option
```

CHAPTER

SIX

HIGHER-ORDER PROGRAMMING

Functions are values just like any other value in OCaml. What does that mean exactly? This means that we can pass functions around as arguments to other functions, that we can store functions in data structures, that we can return functions as a result from other functions, and so forth.

Higher-order functions either take other functions as input or return other functions as output (or both). Higher-order functions are also known as *functionals*, and programming with them could therefore be called *functional programming*—indicating what the heart of programming in languages like OCaml is all about.

Higher-order functions were one of the more recent adoptions from functional languages into mainstream languages. The Java 8 Streams library and Python 2.3's `itertools` modules are examples of that; C++ has also been increasing its support since at least 2011.

Note: C wizards might object the adoption isn't so recent. After all, C has long had the ability to do higher-order programming through function pointers. But that ability also depends on the programming pattern of passing an additional *environment* parameter to provide the values of variables in the function to be called through the pointer. As we'll see in our later chapter on interpreters, the essence of (higher-order) functions in a functional langauge is that they are really something called a *closure* that obviates the need for that extra parameter. Bear in mind that the issue is not what is *possible* to compute in a language—after all everything is eventually compiled down to machine code, so we could just write in that exclusively—but what is *pleasant* to compute.

In this chapter we will see what all the fuss is about. Higher-order functions enable beautiful, general, reusable code.

6.1 Higher-Order Functions

Consider these functions `double` and `square` on integers:

```
let double x = 2 * x
let square x = x * x
```

Let's use these functions to write other functions that quadruple and raise a number to the fourth power:

```
let quad x = double (double x)
let fourth x = square (square x)
```

There is an obvious similarity between these two functions: what they do is apply a given function twice to a value. By passing in the function to another function `twice` as an argument, we can abstract this functionality:

```
let twice f x = f (f x)
```

The function `twice` is higher-order: its input `f` is a function. And—recalling that all OCaml functions really take only a single argument—its output is technically `fun x -> f (f x)`, so `twice` returns a function hence is also higher-order in that way.

Using `twice`, we can implement `quad` and `fourth` in a uniform way:

```
let quad x = twice double x
let fourth x = twice square x
```

6.1.1 The Abstraction Principle

Above, we have exploited the structural similarity between `quad` and `fourth` to save work. Admittedly, in this toy example it might not seem like much work. But imagine that `twice` were actually some much more complicated function. Then if someone comes up with a more efficient version of it, every function written in terms of it (like `quad` and `fourth`) could benefit from that improvement in efficiency, without needing to be recoded.

Part of being an excellent programmer is recognizing such similarities and *abstracting* them by creating functions (or other units of code) that implement them. Bruce MacLennan names this the **Abstraction Principle** in his textbook *Functional Programming: Theory and Practice* (1990). The Abstraction Principle says to avoid requiring something to be stated more than once; instead, *factor out* the recurring pattern. Higher-order functions enable such refactoring, because they allow us to factor out functions and parameterize functions on other functions.

Besides `twice`, here are some more relatively simple examples, indebted also to MacLennan:

Apply. We can write a function that applies its first input to its second input:

```
let apply f x = f x
```

Of course, writing `apply f` is a lot more work than just writing `f`.

Pipeline. The pipeline operator, which we've previously seen, is a higher-order function:

```
let pipeline x f = f x
let (|>) = pipeline
let x = 5 |> double
```

Compose. We can write a function that composes two other functions:

```
let compose f g x = f (g x)
```

This function would let us create a new function that can be applied many times, such as the following:

```
let square_then_double = compose double square
let x = square_then_double 1
let y = square_then_double 2
```

Both. We can write a function that applies two functions to the same argument and returns a pair of the result:

```
let both f g x = (f x, g x)
let ds = both double square
let p = ds 3
```

Cond. We can write a function that conditionally chooses which of two functions to apply based on a predicate:

```
let cond p f g x =
  if p x then f x else g x
```

6.1.2 The Meaning of "Higher Order"

The phrase "higher order" is used throughout logic and computer science, though not necessarily with a precise or consistent meaning in all cases.

In logic, *first-order quantification* refers primarily to the universal and existential ($\forall$ and $\exists$) quantifiers. These let you quantify over some *domain* of interest, such as the natural numbers. But for any given quantification, say $\forall x$, the variable being quantified represents an individual element of that domain, say the natural number 42.

Second-order quantification lets you do something strictly more powerful, which is to quantify over *properties* of the domain. Properties are assertions about individual elements, for example, that a natural number is even, or that it is prime. In some logics we can equate properties with sets of individual, for example the set of all even naturals. So second-order quantification is often thought of as quantification over *sets*. You can also think of properties as being functions that take in an element and return a Boolean indicating whether the element satisfies the property; this is called the *characteristic function* of the property.

Third-order logic would allow quantification over properties of properties, and *fourth-order* over properties of properties of properties, and so forth. *Higher-order logic* refers to all these logics that are more powerful than first-order logic; though one interesting result in this area is that all higher-order logics can be expressed in second-order logic.

In programming languages, *first-order functions* similarly refer to functions that operate on individual data elements (e.g., strings, ints, records, variants, etc.). Whereas *higher-order function* can operate on functions, much like higher-order logics can quantify over over properties (which are like functions).

6.1.3 Famous Higher-order Functions

In the next few sections we'll dive into three of the most famous higher-order functions: map, filter, and fold. These are functions that can be defined for many data structures, including lists and trees. The basic idea of each is that:

- *map* transforms elements,
- *filter* eliminates elements, and
- *fold* combines elements.

6.2 Map

Here are two functions we might want to write:

```
(** [add1 lst] adds 1 to each element of [lst] *)
let rec add1 = function
  | [] -> []
  | h :: t -> (h + 1) :: add1 t

let lst1 = add1 [1; 2; 3]
```

```
(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let rec concat_bang = function
  | [] -> []
```

(continues on next page)

(continued from previous page)

```
  | h :: t -> (h ^ "!") :: concat_bang t

let lst2 = concat_bang ["sweet"; "salty"]
```

There's a lot of similarity between those two functions:

- They both pattern match against a list.
- They both return the same value for the base case of the empty list.
- They both recurse on the tail in the case of a non-empty list.

In fact the only difference (other than their names) is what they do for the head element: add versus concatenate. Let's rewrite the two functions to make that difference even more explicit:

```
(** [add1 lst] adds 1 to each element of [lst] *)
let rec add1 = function
  | [] -> []
  | h :: t ->
    let f = fun x -> x + 1 in
    f h :: add1 t

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let rec concat_bang = function
  | [] -> []
  | h :: t ->
    let f = fun x -> x ^ "!" in
    f h :: concat_bang t
```

Now the only difference between the two functions (again, other than their names) is the body of helper function `f`. Why repeat all that code when there's such a small difference between the functions? We might as well *abstract* that one helper function out from each main function and make it an argument:

```
let rec add1' f = function
  | [] -> []
  | h :: t -> f h :: add1' f t

(** [add1 lst] adds 1 to each element of [lst] *)
let add1 = add1' (fun x -> x + 1)

let rec concat_bang' f = function
  | [] -> []
  | h :: t -> f h :: concat_bang' f t

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let concat_bang = concat_bang' (fun x -> x ^ "!")
```

But now there really is no difference at all between `add1'` and `concat_bang'` except for their names. They are totally duplicated code. Even their types are now the same, because nothing about them mentions integers or strings. We might as well just keep only one of them and come up with a good new name for it. One possibility could be `transform`, because they tranform a list by applying a function to each element of the list:

```
let rec transform f = function
  | [] -> []
  | h :: t -> f h :: transform f t
```

(continues on next page)

(continued from previous page)

```
(** [add1 lst] adds 1 to each element of [lst] *)
let add1 = transform (fun x -> x + 1)

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let concat_bang = transform (fun x -> x ^ "!")
```

Note: Instead of

```
let add1 lst = transform (fun x -> x + 1) lst
```

above we wrote

```
let add1 = transform (fun x -> x + 1)
```

This is another way of being higher order, but it's one we already learned about under the guise of partial application. The latter way of writing the function partially applies `transform` to just one of its two arguments, thus returning a function. That function is bound to the name `add1`.

Indeed, the C++ library does call the equivalent function `transform`. But OCaml and many other languages (including Java and Python) use the shorter word *map*, in the mathematical sense of how a function maps an input to an output. So let's make one final change to that name:

```
let rec map f = function
  | [] -> []
  | h :: t -> f h :: map f t

(** [add1 lst] adds 1 to each element of [lst] *)
let add1 = map (fun x -> x + 1)

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let concat_bang = map (fun x -> x ^ "!")
```

We have now successfully applied the Abstraction Principle: the common structure has been factored out. What's left clearly expresses the computation, at least to the reader who is familiar with `map`, in a way that the original versions do not as quickly make apparent.

6.2.1 Side Effects

The `map` function exists already in OCaml's standard library as `List.map`, but with one small difference from the implementation we discovered above. First, let's see what's potentially wrong with our own implementation, then we'll look at the standard library's implementation.

We've seen before in our discussion of *exceptions* that the OCaml language specification does not generally specify evaluation order of subexpressions, and that the current language implementation generally evaluates right-to-left. Because of that, the following (rather contrived) code actually causes the list elements to be printed in what might seem like reverse order:

```
let p x = print_int x; print_newline(); x + 1

let lst = map p [1; 2]
```

Here's why:

- Expression `map p [1; 2]` evaluates to `p 1 :: map p [2]`.
- The right-hand side of that expression is then evaluated to `p 1 :: (p 2 :: map p [])`. The application of `p` to `1` has not yet occurred.
- The right-hand side of `::` is again evaluated next, yielding `p 1 :: (p 2 :: [])`.
- Then `p` is applied to `2`, and finally to `1`.

That is likely surprising to anyone who is predisposed to thinking that evaluation would occur left-to-right. The solution is to use a `let` expression to cause the evaluation of the function application to occur before the recursive call:

```
let rec map f = function
  | [] -> []
  | h :: t -> let h' = f h in h' :: map f t

let lst2 = map p [1; 2]
```

Here's why that works:

- Expression `map p [1; 2]` evaluates to `let h' = p 1 in h' :: map p [2]`.
- The binding expression `p 1` is evaluated, causing `1` to be printed and `h'` to be bound to `2`.
- The body expression `h' :: map p [2]` is then evaluated, which leads to `2` being printed next.

So that's how the standard library defines `List.map`. We should use it instead of re-defining the function ourselves from now on. But it's good that we have discovered the function "from scratch" as it were, and that if needed we could quickly re-code it.

The bigger lesson to take away from this discussion is that when evaluation order matters, we need to use `let` to ensure it. When does it matter? Only when there are side effects. Printing and exceptions are the two we've seen so far. Later we'll add mutability.

6.2.2 Map and Tail Recursion

Astute readers will have noticed that the implementation of `map` is not tail recursive. That is to some extent unavoidable. Here's a tempting but awful way to create a tail-recursive version of it:

```
let rec map_tr_aux f acc = function
  | [] -> acc
  | h :: t -> map_tr_aux f (acc @ [f h]) t

let map_tr f = map_tr_aux f []

let lst = map_tr (fun x -> x + 1) [1; 2; 3]
```

To some extent that works: the output is correct, and `map_tr_aux` is tail recursive. The subtle flaw is the subexpression `acc @ [f h]`. Recall that append is a linear-time operation on singly-linked lists. That is, if there are n list elements then append takes time $O(n)$. So at each recursive call we perform a $O(n)$ operation. And there will be n recursive calls, one for each element of the list. That's a total of $n \cdot O(n)$ work, which is $O(n^2)$. So we achieved tail recursion, but at a high cost: what ought to be a linear-time operation became quadratic time.

In an attempt to fix that, we could use the constant-time cons operation instead of the linear-time append operation:

```
let rec map_tr_aux f acc = function
  | [] -> acc
  | h :: t -> map_tr_aux f (f h :: acc) t
```

(continues on next page)

(continued from previous page)

```
let map_tr f = map_tr_aux f []

let lst = map_tr (fun x -> x + 1) [1; 2; 3]
```

And to some extent that works: it's tail recursive and linear time. The not-so-subtle flaw this time is that the output is backwards. As we take each element off the front of the input list, we put it on the front of the output list, but that reverses their order.

Note: To understand why the reversal occurs, it might help to think of the input and output lists as people standing in a queue:

- Input: Alice, Bob.
- Output: empty.

Then we remove Alice from the input and add her to the output:

- Input: Bob.
- Output: Alice.

Then we remove Bob from the input and add him to the output:

- Input: empty.
- Output: Bob, Alice.

The point is that with singly-linked lists, we can only operate on the head of the list and still be constant time. We can't move Bob to the back of the output without making him walk past Alice—and anyone else who might be standing in the output.

For that reason, the standard library calls this function `List.rev_map`, that is, a (tail-recursive) map function that returns its output in reverse order.

```
let rec rev_map_aux f acc = function
  | [] -> acc
  | h :: t -> rev_map_aux f (f h :: acc) t

let rev_map f = rev_map_aux f []

let lst = rev_map (fun x -> x + 1) [1; 2; 3]
```

If you want the output in the "right" order, that's easy: just apply `List.rev` to it:

```
let lst = List.rev (List.rev_map (fun x -> x + 1) [1; 2; 3])
```

Since `List.rev` is both linear time and tail recursive, that yields a complete solution. We get a linear-time and tail-recursive map computation. The expense is that it requires two passes through the list: one to transform, the other to reverse. We're not going to do better than this efficiency with a singly-linked list. Of course, there are other data structures that implement lists, and we'll come to those eventually. Meanwhile, recall that we generally don't have to worry about tail recursion (which is to say, about stack space) until lists have 10,000 or more elements.

Why doesn't the standard library provide this all-in-one function? Maybe it will someday if there's good enough reason. But you might discover in your own programming there's not a lot of need for it. In many cases, we can either do without the tail recursion, or be content with a reversed list.

The bigger lesson to take away from this discussion is that there can be a tradeoff between time and space efficiency for recursive functions. By attempting to make a function more space efficient (i.e., tail recursive), we can accidentally make it asympotically less time efficient (i.e., quadratic instead of linear), or if we're clever keep the asymptotic time efficiency the same (i.e., linear) at the cost of a constant factor (i.e., processing twice).

6.2.3 Map in Other Languages

We mentioned above that the idea of map exists in many programming languages. Here's an example from Python:

```
>>> print(list(map(lambda x: x + 1, [1, 2, 3])))
[2, 3, 4]
```

We have to use the `list` function to convert the result of the `map` back to a list, because Python for sake of efficiency produces each element of the `map` output as needed. Here again we see the theme of "when does it get evaluated?" returning.

In Java, map is part of the `Stream` abstraction that was added in Java 8. Since there isn't a built-in Java syntax for lists or streams, it's a little more verbose to give an example. Here we use a factory method `Stream.of` to create a stream:

```
jshell> Stream.of(1, 2, 3).map(x -> x + 1).collect(Collectors.toList())
$1 ==> [2, 3, 4]
```

Like in the Python example, we have to use something to convert the stream back into a list. In this case it's the `collect` method.

6.3 Filter

Suppose we wanted to filter out only the even numbers from a list, or the odd numbers. Here are some functions to do that:

```
(** [even n] is whether [n] is even. *)
let even n =
  n mod 2 = 0

(** [evens lst] is the sublist of [lst] containing only even numbers. *)
let rec evens = function
  | [] -> []
  | h :: t -> if even h then h :: evens t else evens t

let lst1 = evens [1; 2; 3; 4]
```

```
(** [odd n] is whether [n] is odd. *)
let odd n =
  n mod 2 <> 0

(** [odds lst] is the sublist of [lst] containing only odd numbers. *)
let rec odds = function
  | [] -> []
  | h :: t -> if odd h then h :: odds t else odds t

let lst2 = odds [1; 2; 3; 4]
```

Functions `evens` and `odds` are nearly the same code: the only essential difference is the test they apply to the head element. So as we did with `map` in the previous section, let's factor out that test as a function. Let's name the function `p` as short for "predicate", which is a fancy way of saying that it tests whether something is true or false:

```
let rec filter p = function
  | [] -> []
  | h :: t -> if p h then h :: filter p t else filter p t
```

And now we can reimplement our original two functions:

```
let evens = filter even
let odds = filter odd
```

How simple these are! How clear! (At least to the reader who is familiar with `filter`.)

6.3.1 Filter and Tail Recursion

As we did with `map`, we can create a tail-recursive version of `filter`:

```
let rec filter_aux p acc = function
  | [] -> acc
  | h :: t -> if p h then filter_aux p (h :: acc) t else filter_aux p acc t

let filter p = filter_aux p []

let lst = filter even [1; 2; 3; 4]
```

And again we discover the output is backwards. Here, the standard library makes a different choice than it did with `map`. It builds in the reversal to `List.filter`, which is implemented like this:

```
let rec filter_aux p acc = function
  | [] -> List.rev acc (* note the built-in reversal *)
  | h :: t -> if p h then filter_aux p (h :: acc) t else filter_aux p acc t

let filter p = filter_aux p []
```

Why does the standard library treat `map` and `filter` differently on this point? Good question. Perhaps there has simply never been a demand for a `filter` function whose time efficiency is a constant factor better. Or perhaps it is just historical accident.

6.3.2 Filter in Other Languages

Again, the idea of filter exists in many programming languages. Here it is in Python:

```
>>> print(list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4])))
[2, 4]
```

And in Java:

```
jshell> Stream.of(1, 2, 3, 4).filter(x -> x % 2 == 0).collect(Collectors.toList())
$1 ==> [2, 4]
```

6.4 Fold

The map functional gives us a way to individually transform each element of a list. The filter functional gives us a way to individually decide whether to keep or throw away each element of a list. But both of those are really just looking at a single element at a time. What if we wanted to somehow combine all the elements of a list? That's what the *fold* functional is for. It turns out that there are two versions of it, which we'll study in this section. But to start, let's look at a related function—not actually in the standard library—that we call *combine*.

6.4.1 Combine

Once more, let's write two functions:

```
(** [sum lst] is the sum of all the elements of [lst]. *)
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t

let s = sum [1; 2; 3]
```

```
(** [concat lst] is the concatenation of all the elements of [lst]. *)
let rec concat = function
  | [] -> ""
  | h :: t -> h ^ concat t

let c = concat ["a"; "b"; "c"]
```

As when we went through similar exercises with map and filter, the functions share a great deal of common structure. The differences here are:

- the case for the empty list returns a different initial value, `0` vs `""`
- the case of a non-empty list uses a different operator to combine the head element with the result of the recursive call, `+` vs `^`.

So can we apply the Abstraction Principle again? Sure! But this time we need to factor out *two* arguments: one for each of those two differences.

To start, let's factor out only the initial value:

```
let rec sum' init = function
  | [] -> init
  | h :: t -> h + sum' init t

let sum = sum' 0

let rec concat' init = function
  | [] -> init
  | h :: t -> h ^ concat' init t

let concat = concat' ""
```

Now the only real difference left between `sum'` and `concat'` is the operator used to combine the head with the recursive call on the tail. That operator can also become an argument to a unified function we call `combine`:

```
let rec combine op init = function
  | [] -> init
  | h :: t -> op h (combine op init t)

let sum = combine ( + ) 0
let concat = combine ( ^ ) ""
```

One way to think of `combine` would be that:

- the `[]` value in the list gets replaced by `init`, and
- each `::` constructor gets replaced by `op`.

For example, `[a; b; c]` is just syntactic sugar for `a :: (b :: (c :: []))`. So if we replace `[]` with `0` and `::` with `(+)`, we get `a + (b + (c + 0))`. And that would be the sum of the list.

Once more, the Abstraction Principle has led us to an amazingly simple and succinct expression of the computation.

6.4.2 Fold Right

The `combine` function is the idea underlying an actual OCaml library function. To get there, we need to make a couple of changes to the implementation we have so far.

First, let's rename some of the arguments: we'll change `op` to `f` to emphasize that really we could pass in any function, not just a built-in operator like +. And we'll change `init` to `acc`, which as usual stands for "accumulator". That yields:

```
let rec combine f acc = function
  | [] -> acc
  | h :: t -> f h (combine f acc t)
```

Second, let's make an admittedly less well-motivated change. We'll swap the implicit list argument to `combine` with the `init` argument:

```
let rec combine' f lst acc = match lst with
  | [] -> acc
  | h :: t -> f h (combine' f t acc)

let sum lst = combine' ( + ) lst 0
let concat lst = combine' ( ^ ) lst ""
```

It's a little less convenient to code the function this way, because we no longer get to take advantage of the `function` keyword, nor of partial application in defining `sum` and `concat`. But there's no algorithmic change.

What we now have is the actual implementation of the standard library function `List.fold_right`. All we have left to do is change the function name:

```
let rec fold_right f lst acc = match lst with
  | [] -> acc
  | h :: t -> f h (fold_right f t acc)
```

Why is this function called "fold right"? The intuition is that the way it works is to "fold in" elements of the list from the right to the left, combining each new element using the operator. For example, `fold_right ( + ) [a; b; c] 0` results in evaluation of the expression `a + (b + (c + 0))`. The parentheses associate from the right-most subexpression to the left.

6.4.3 Tail Recursion and Combine

Neither `fold_right` nor `combine` are tail recursive: after the recursive call returns, there is still work to be done in applying the function argument `f` or `op`. Let's go back to `combine` and rewrite it to be tail recursive. All that requires is to change the cons branch:

```
let rec combine_tr f acc = function
  | [] -> acc
  | h :: t -> combine_tr f (f acc h) t   (* only real change *)
```

(Careful readers will notice that the type of `combine_tr` is different than the type of `combine`. We will address that soon.)

Now the function `f` is applied to the head element `h` and the accumulator `acc` *before* the recursive call is made, thus ensuring there's no work remaining to be done after the call returns. If that seems a little mysterious, here's a rewriting of the two functions that might help:

```
let rec combine f acc = function
  | [] -> acc
  | h :: t ->
    let acc' = combine f acc t in
    f h acc'

let rec combine_tr f acc = function
  | [] -> acc
  | h :: t ->
    let acc' = f acc h in
    combine_tr f acc' t
```

Pay close attention to the definition of `acc'`, the new accumulator, in each of those version:

- In the original version, we procrastinate using the head element `h`. First, we combine all the remaining tail elements to get `acc'`. Only then do we use `f` to fold in the head. So the value passed as the initial value of `acc` turns out to be the same for every recursive invocation of `combine`: it's passed all the way down to where it's needed, at the right-most element of the list, then used there exactly once.
- But in the tail recursive version, we "pre-crastinate" by immediately folding `h` in with the old accumulator `acc`. Then we fold that in with all the tail elements. So at each recursive invocation, the value passed as the argument `acc` can be different.

The tail recursive version of combine works just fine for summation (and concatenation, which we elide):

```
let sum = combine_tr ( + ) 0
let s = sum [1; 2; 3]
```

But something possibly surprising happens with subtraction:

```
let sub = combine ( - ) 0
let s = sub [3; 2; 1]

let sub_tr = combine_tr ( - ) 0
let s' = sub_tr [3; 2; 1]
```

The two results are different!

- With `combine` we compute `3 - (2 - (1 - 0))`. First we fold in `1`, then `2`, then `3`. We are processing the list from right to left, putting the initial accumulator at the far right.

- But with `combine_tr` we compute `(((0 - 3) - 2) - 1)`. We are processing the list from left to right, putting the initial accumulator at the far left.

With addition it didn't matter which order we processed the list, because addition is associative and commutative. But subtraction is not, so the two directions result in different answers.

Actually this shouldn't be too surprising if we think back to when we made `map` be tail recursive. Then, we discovered that tail recursion can cause us to process the list in reverse order from the non-tail recursive version of the same function. That's what happened here.

6.4.4 Fold Left

Our `combine_tr` function is also in the standard library under the name `List.fold_left`:

```
let rec fold_left f acc = function
  | [] -> acc
  | h :: t -> fold_left f (f acc h) t

let sum = fold_left ( + ) 0
let concat = fold_left ( ^ ) ""
```

We have once more succeeded in applying the Abstraction Principle.

6.4.5 Fold Left vs. Fold Right

Let's review the differences between `fold_right` and `fold_left`:

- They combine list elements in opposite orders, as indicated by their names. Function `fold_right` combines from the right to the left, whereas `fold_left` proceeds from the left to the right.
- Function `fold_left` is tail recursive whereas `fold_right` is not.
- The types of the functions are different.

Regarding that final point, it can be hard to remember what those types are! Luckily we can always ask the toplevel:

```
List.fold_left;;
List.fold_right;;
```

To understand those types, look for the list argument in each one of them. That tells you the type of the values in the list. Then look for the type of the return value; that tells you the type of the accumulator. From there you can work out everything else.

- In `fold_left`, the list argument is of type `'b list`, so the list contains values of type `'b`. The return type is `'a`, so the accumulator has type `'a`. Knowing that, we can figure out that the second argument is the initial value of the accumulator (because it has type `'a`). And we can figure out that the first argument, the combining operator, takes as its own first argument an accumulator value (because it has type `'a`), as its own second argument a list element (because it has type `'b`), and returns a new accumulator value.
- In `fold_right`, the list argument is of type `'a list`, so the list contains values of type `'a`. The return type is `'b`, so the accumulator has type `'b`. Knowing that, we can figure out that the third argument is the initial value of the accumulator (because it has type `'b`). And we can figure out that the first argument, the combining operator, takes as its own second argument an accumulator value (because it has type `'b`), as its own first argument a list element (because it has type `'a`), and returns a new accumulator value.

Tip: You might wonder why the argument orders are different between the two `fold` functions. Good question. Other libraries do in fact use different argument orders. One way to remember it for OCaml is that in `fold_X` the accumulator argument goes to the `X` of the list argument.

If you find it hard to keep track of all these argument orders, the `ListLabels` module in the standard library can help. It uses labeled arguments to give names to the combining operator (which it calls `f`) and the initial accumulator value (which it calls `init`). Internally, the implementation is actually identical to the `List` module.

```
ListLabels.fold_left;;
ListLabels.fold_left ~f:(fun x y -> x - y) ~init:0 [1;2;3];;
```

```
ListLabels.fold_right;;
ListLabels.fold_right ~f:(fun y x -> x - y) ~init:0 [1;2;3];;
```

Notice how in the two applications of fold above, we are able to write the arguments in a uniform order thanks to their labels. However, we still have to be careful about which argument to the combining operator is the list element vs. the accumulator value.

6.4.6 A Digression on Labeled Arguments and Fold

It's possible to write our own version of the fold functions that would label the arguments to the combining operator, so we don't even have to remember their order:

```
let rec fold_left ~op:(f: acc:'a -> elt:'b -> 'a) ~init:acc lst =
  match lst with
  | [] -> acc
  | h :: t -> fold_left ~op:f ~init:(f ~acc:acc ~elt:h) t

let rec fold_right ~op:(f: elt:'a -> acc:'b -> 'b) lst ~init:acc =
  match lst with
  | [] -> acc
  | h :: t -> f ~elt:h ~acc:(fold_right ~op:f t ~init:acc)
```

But those functions aren't as useful as they might seem:

```
let s = fold_left ~op:( + ) ~init:0 [1;2;3]
```

The problem is that the built-in + operator doesn't have labeled arguments, so we can't pass it in as the combining operator to our labeled functions. We'd have to define our own labeled version of it:

```
let add ~acc ~elt = acc + elt
let s = fold_left ~op:add ~init:0 [1; 2; 3]
```

But now we have to remember that the `~acc` parameter to `add` will become the left-hand argument to `( + )`. That's not really much of an improvement over what we had to remember to begin with.

6.4.7 Using Fold to Implement Other Functions

Folding is so powerful that we can write many other list functions in terms of `fold_left` or `fold_right`. For example,

```
let length lst =
  List.fold_left (fun acc _ -> acc + 1) 0 lst

let rev lst =
  List.fold_left (fun acc x -> x :: acc) [] lst

let map f lst =
  List.fold_right (fun x acc -> f x :: acc) lst []

let filter f lst =
  List.fold_right (fun x acc -> if f x then x :: acc else acc) lst []
```

At this point it begins to become debatable whether it's better to express the computations above using folding or using the ways we have already seen. Even for an experienced functional programmer, understanding what a fold does can take longer than reading the naive recursive implementation. If you peruse the source code of the standard library, you'll see that none of the `List` module internally is implemented in terms of folding, which is perhaps one comment on the readability of fold. On the other hand, using fold ensures that the programmer doesn't accidentally program the recursive traversal incorrectly. And for a data structure that's more complicated than lists, that robustness might be a win.

6.4.8 Fold vs. Recursive vs. Library

We've now seen three different ways for writing functions that manipulate lists:

- directly as a recursive function that pattern matches against the empty list and against cons,
- using `fold` functions, and
- using other library functions.

Let's try using each of those ways to solve a problem, so that we can appreciate them better.

Consider writing a function `lst_and: bool list -> bool`, such that `lst_and [a1; ...; an]` returns whether all elements of the list are `true`. That is, it evaluates the same as `a1 && a2 && ... && an`. When applied to an empty list, it evaluates to `true`.

Here are three possible ways of writing such a function. We give each way a slightly different function name for clarity.

```
let rec lst_and_rec = function
  | [] -> true
  | h :: t -> h && lst_and_rec t

let lst_and_fold =
        List.fold_left (fun acc elt -> acc && elt) true

let lst_and_lib =
        List.for_all (fun x -> x)
```

The worst-case running time of all three functions is linear in the length of the list. But:

- The first function, `lst_and_rec` has the advantage that it need not process the entire list. It will immediately return `false` the first time they discover a `false` element in the list.
- The second function, `lst_and_fold`, will always process every element of the list.

- As for the third function `lst_and_lib`, according to the documentation of `List.for_all`, it returns `(p a1) && (p a2) && ... && (p an)`. So like `lst_and_rec` it need not process every element.

6.5 Beyond Lists

Functionals like map and fold are not restricted to lists. They make sense for nearly any kind of data collection. For example, recall this tree representation:

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

6.5.1 Map on Trees

This one is easy. All we have to do is apply the function `f` to the value `v` at each node:

```
let rec map_tree f = function
  | Leaf -> Leaf
  | Node (v, l, r) -> Node (f v, map_tree f l, map_tree f r)
```

6.5.2 Fold on Trees

This one is only a little harder. Let's develop a fold functional for `'a tree` similar to our `fold_right` over `'a list`. One way to think of `List.fold_right` would be that the `[]` value in the list gets replaced by the `acc` argument, and each `::` constructor gets replaced by an application of the `f` argument. For example, `[a; b; c]` is syntactic sugar for `a :: (b :: (c :: []))`. So if we replace `[]` with `0` and `::` with `( + )`, we get `a + (b + (c + 0))`. Along those lines, here's a way we could rewrite `fold_right` that will help us think a little more clearly:

```
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist

let rec fold_mylist f acc = function
  | Nil -> acc
  | Cons (h, t) -> f h (fold_mylist f acc t)
```

The algorithm is the same. All we've done is to change the definition of lists to use constructors written with alphabetic characters instead of punctuation, and to change the argument order of the fold function.

For trees, we'll want the initial value of `acc` to replace each `Leaf` constructor, just like it replaced `[]` in lists. And we'll want each `Node` constructor to be replaced by the operator. But now the operator will need to be *ternary* instead of *binary*—that is, it will need to take three arguments instead of two—because a tree node has a value, a left child, and a right child, whereas a list cons had only a head and a tail.

Inspired by those observations, here is the fold function on trees:

```
let rec fold_tree f acc = function
  | Leaf -> acc
  | Node (v, l, r) -> f v (fold_tree f acc l) (fold_tree f acc r)
```

If you compare that function to `fold_mylist`, you'll note it very nearly identical. There's just one more recursive call in the second pattern-matching branch, corresponding to the one more occurrence of `'a tree` in the definition of that type.

We can then use `fold_tree` to implement some of the tree functions we've previously seen:

```
let size t = fold_tree (fun _ l r -> 1 + l + r) 0 t
let depth t = fold_tree (fun _ l r -> 1 + max l r) 0 t
let preorder t = fold_tree (fun x l r -> [x] @ l @ r) [] t
```

Why did we pick `fold_right` and not `fold_left` for this development? Because `fold_left` is tail recursive, which is something we're never going to achieve on binary trees. Suppose we process the left branch first; then we still have to process the right branch before we can return. So there will always be work left to do after a recursive call on one branch. Thus on trees an equivalent to `fold_right` is the best which we can hope for.

The technique we used to derive `fold_tree` works for any OCaml variant type `t`:

- Write a recursive `fold` function that takes in one argument for each constructor of `t`.
- That `fold` function matches against the constructors, calling itself recursively on any value of type `t` that it encounters.
- Use the appropriate argument of `fold` to combine the results of all recursive calls as well as all data not of type `t` at each constructor.

This technique constructs something called a *catamorphism*, aka a *generalized fold operation*. To learn more about catamorphisms, take a course on category theory.

6.5.3 Filter on Trees

This one is perhaps the hardest to design. The problem is: if we decide to filter a node, what should we do with its children?

- We could recurse on the children. If after filtering them only one child remains, we could promote it in place of its parent. But what if both children remain, or neither? Then we'd somehow have to reshape the tree. Without knowing more about how the tree is intended to be used—that is, what kind of data it represents—we are stuck.
- Instead, we could just eliminate the children entirely. So the decision to filter a node means pruning the entire subtree rooted at that node.

The latter is easy to implement:

```
let rec filter_tree p = function
  | Leaf -> Leaf
  | Node (v, l, r) ->
    if p v then Node (v, filter_tree p l, filter_tree p r) else Leaf
```

6.6 Pipelining

Suppose we wanted to compute the sum of squares of the numbers from 0 up to n. How might we go about it? Of course (math being the best form of optimization), the most efficient way would be a closed-form formula:

$$\frac{n(n+1)(2n+1)}{6}$$

But let's imagine you've forgotten that formula. In an imperative language you might use a `for` loop:

```
# Python
def sum_sq(n):
        sum = 0
        for i in range(0, n+1):
                sum += i * i
        return sum
```

The equivalent (tail) recursive code in OCaml would be:

```
let sum_sq n =
  let rec loop i sum =
    if i > n then sum
    else loop (i + 1) (sum + i * i)
  in loop 0 0
```

Another, clearer way of producing the same result in OCaml uses higher-order functions and the pipeline operator:

```
let rec ( -- ) i j = if i > j then [] else i :: i + 1 -- j
let square x = x * x
let sum = List.fold_left ( + ) 0

let sum_sq n =
  0 -- n               (* [0;1;2;...;n]   *)
  |> List.map square   (* [0;1;4;...;n*n] *)
  |> sum               (*  0+1+4+...+n*n  *)
```

The function `sum_sq` first constructs a list containing all the numbers `0..n`. Then it uses the pipeline operator `|>` to pass that list through `List.map square`, which squares every element. Then the resulting list is pipelined through `sum`, which adds all the elements together.

The other alternatives that you might consider are somewhat uglier:

```
(* Maybe worse: a lot of extra [let..in] syntax and unnecessary names to
   for intermediate values we don't care about. *)
let sum_sq n =
  let l = 0 -- n in
  let sq_l = List.map square l in
  sum sq_l

(* Maybe worse: have to read the function applications from right to left
   rather than top to bottom, and extra parentheses. *)
let sum_sq n =
  sum (List.map square (0--n))
```

The downside of all of these compared to the original tail recursive version is that they are wasteful of space—linear instead of constant—and take a constant factor more time. So as is so often the case in programming, there is a tradeoff between clarity and efficiency of code.

Note that the inefficiency is *not* from the pipeline operator itself, but from having to construct all those unnecessary intermediate lists. So don't get the idea that pipelining is intrinsically bad. In fact it can be quite useful. When we get to the chapter on modules, we'll use it quite often with some of the data structures we study there.

6.7 Currying

We've already seen that an OCaml function that takes two arguments of types `t1` and `t2` and returns a value of type `t3` has the type `t1 -> t2 -> t3`. We use two variables after the function name in the let expression:

```
let add x y = x + y
```

Another way to define a function that takes two arguments is to write a function that takes a tuple:

```
let add' t = fst t + snd t
```

Instead of using `fst` and `snd`, we could use a tuple pattern in the definition of the function, leading to a third implementation:

```
let add'' (x, y) = x + y
```

Functions written using the first style (with type `t1 -> t2 -> t3`) are called *curried* functions, and functions using the second style (with type `t1 * t2 -> t3`) are called *uncurried*. Metaphorically, curried functions are "spicier" because you can partially apply them (something you can't do with uncurried functions: you can't pass in half of a pair). Actually, the term curry does not refer to spices, but to a logician named Haskell Curry (one of a very small set of people with programming languages named after both their first and last names).

Sometimes you will come across libraries that offer an uncurried version of a function, but you want a curried version of it to use in your own code; or vice versa. So it is useful to know how to convert between the two kinds of functions, as we did with `add` above.

You could even write a couple of higher-order functions to do the conversion for you:

```
let curry f x y = f (x, y)
let uncurry f (x, y) = f x y
```

```
let uncurried_add = uncurry add
let curried_add = curry add''
```

6.8 Summary

This chapter is one of the most important in the book. It didn't cover any new language features. Instead, we learned how to use some of the existing features in ways that might be new, surprising, or challenging. Higher-order programming and the Abstraction Principle are two ideas that will help make you a better programmer in any language, not just OCaml. Of course, languages do vary in the extent to which they support these ideas, with some providing significantly less assistance in writing higher-order code—which is one reason we use OCaml in this course.

Map, filter, fold and other functionals are becoming widely recognized as excellent ways to structure computation. Part of the reason for that is they factor out the *iteration* over a data structure from the *computation* done at each element. Languages such as Python, Ruby, and Java 8 now have support for this kind of iteration.

6.8.1 Terms and Concepts

- Abstraction Principle
- accumulator
- apply
- associative
- compose
- factor
- filter
- first-order function
- fold
- functional
- generalized fold operation
- higher-order function
- map
- pipeline
- pipelining

6.8.2 Further Reading

- *Introduction to Objective Caml*, chapters 3.1.3, 5.3
- *OCaml from the Very Beginning*, chapter 6
- *More OCaml: Algorithms, Methods, and Diversions*, chapter 1, by John Whitington. This book is a sequel to *OCaml from the Very Beginning*.
- *Real World OCaml*, chapter 3 (beware that this book's `Core` library has a different `List` module than the standard library's `List` module, with different types for `map` and `fold` than those we saw here)
- "Higher Order Functions", chapter 6 of *Functional Programming: Practice and Theory*. Bruce J. MacLennan, Addison-Wesley, 1990. Our discussion of higher-order functions and the Abstraction Principle is indebted to this chapter.
- "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." John Backus' 1977 Turing Award lecture in its elaborated form as a published article.
- "Second-order and Higher-order Logic" in *The Stanford Encyclopedia of Philosophy*.

6.9 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Exercise: twice, no arguments [★]

Consider the following definitions:

```
let double x = 2*x
let square x = x*x
let twice f x = f (f x)
let quad = twice double
let fourth = twice square
```

Use the toplevel to determine what the types of `quad` and `fourth` are. Explain how it can be that `quad` is not syntactically written as a function that takes an argument, and yet its type shows that it is in fact a function.

Exercise: mystery operator 1 [★★]

What does the following operator do?

```
let ( $ ) f x = f x
```

Hint: investigate `square $ 2 + 2` vs. `square 2 + 2`.

Exercise: mystery operator 2 [★★]

What does the following operator do?

```
let ( @@ ) f g x = x |> g |> f
```

Hint: investigate `String.length @@ string_of_int` applied to `1`, `10`, `100`, etc.

Exercise: repeat [★★]

Generalize `twice` to a function `repeat`, such that `repeat f n x` applies `f` to `x` a total of `n` times. That is,

- `repeat f 0 x` yields `x`
- `repeat f 1 x` yields `f x`
- `repeat f 2 x` yields `f (f x)` (which is the same as `twice f x`)
- `repeat f 3 x` yields `f (f (f x))`
- …

Exercise: product [★]

Use `fold_left` to write a function `product_left` that computes the product of a list of floats. The product of the empty list is `1.0`. *Hint: recall how we implemented `sum` in just one line of code in lecture.*

Use `fold_right` to write a function `product_right` that computes the product of a list of floats. *Same hint applies.*

Exercise: terse product [★★]

How terse can you make your solutions to the **product** exercise? *Hints: you need only one line of code for each, and you do not need the `fun` keyword. For `fold_left`, your function definition does not even need to explicitly take a list argument. If you use `ListLabels`, the same is true for `fold_right`.*

Exercise: sum_cube_odd [★★]

Write a function `sum_cube_odd n` that computes the sum of the cubes of all the odd numbers between `0` and `n` inclusive. Do not write any new recursive functions. Instead, use the functionals map, fold, and filter, and the `( -- )` operator (defined in the discussion of pipelining).

Exercise: sum_cube_odd pipeline [★★]

Rewrite the function `sum_cube_odd` to use the pipeline operator `|>`.

Exercise: exists [★★]

Consider writing a function `exists: ('a -> bool) -> 'a list -> bool`, such that `exists p [a1; ...; an]` returns whether at least one element of the list satisfies the predicate `p`. That is, it evaluates the same as `(p a1) || (p a2) || ... || (p an)`. When applied to an empty list, it evaluates to `false`.

Write three solutions to this problem, as we did above:

- `exists_rec`, which must be a recursive function that does not use the `List` module,
- `exists_fold`, which uses either `List.fold_left` or `List.fold_right`, but not any other `List` module functions nor the `rec` keyword, and
- `exists_lib`, which uses any combination of `List` module functions other than `fold_left` or `fold_right`, and does not use the `rec` keyword.

Exercise: account balance [★★★]

Write a function which, given a list of numbers representing debits, deducts them from an account balance, and finally returns the remaining amount in the balance. Write three versions: `fold_left`, `fold_right`, and a direct recursive implementation.

Exercise: library uncurried [★★]

Here is an uncurried version of `List.nth`:

```
let uncurried_nth (lst, n) = List.nth lst n
```

In a similar way, write uncurried versions of these library functions:

- `List.append`
- `Char.compare`
- `Stdlib.max`

Exercise: map composition [★★★]

Show how to replace any expression of the form `List.map f (List.map g lst)` with an equivalent expression that calls `List.map` only once.

Exercise: more list fun [★★★]

Write functions that perform the following computations. Each function that you write should use one of `List.fold`, `List.map` or `List.filter`. To choose which of those to use, think about what the computation is doing: combining, transforming, or filtering elements.

- Find those elements of a list of strings whose length is strictly greater than 3.
- Add `1.0` to every element of a list of floats.
- Given a list of strings `strs` and another string `sep`, produce the string that contains every element of `strs` separated by `sep`. For example, given inputs `["hi";"bye"]` and `","`, produce `"hi,bye"`, being sure not to produce an extra comma either at the beginning or end of the result string.

Exercise: association list keys [★★★]

Recall that an association list is an implementation of a dictionary in terms of a list of pairs, in which we treat the first component of each pair as a key and the second component as a value.

Write a function `keys: ('a * 'b) list -> 'a list` that returns a list of the unique keys in an association list. Since they must be unique, no value should appear more than once in the output list. The order of values output does not matter. How compact and efficient can you make your solution? Can you do it in one line and linearithmic space and time? *Hint: `List.sort_uniq`.*

Exercise: valid matrix [★★★]

A mathematical *matrix* can be represented with lists. In *row-major* representation, this matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ 9 & 8 & 7 \end{bmatrix}$$

would be represented as the list `[[1; 1; 1]; [9; 8; 7]]`. Let's represent a *row vector* as an `int list`. For example, `[9; 8; 7]` is a row vector.

A *valid* matrix is an `int list list` that has at least one row, at least one column, and in which every column has the same number of rows. There are many values of type `int list list` that are invalid, for example,

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

Implement a function `is_valid_matrix: int list list -> bool` that returns whether the input matrix is valid. Unit test the function.

Exercise: row vector add [★★★]

Implement a function `add_row_vectors: int list -> int list -> int list` for the element-wise addition of two row vectors. For example, the addition of `[1; 1; 1]` and `[9; 8; 7]` is `[10; 9; 8]`. If the two

vectors do not have the same number of entries, the behavior of your function is *unspecified*—that is, it may do whatever you like. *Hint: there is an elegant one-line solution using* `List.map2`. Unit test the function.

Exercise: matrix add [★★★]

Implement a function `add_matrices: int list list -> int list list -> int list list` for matrix addition. If the two input matrices are not the same size, the behavior is unspecified. *Hint: there is an elegant one-line solution using* `List.map2` *and* `add_row_vectors`. Unit test the function.

Exercise: matrix multiply [★★★★]

Implement a function `multiply_matrices: int list list -> int list list -> int list list` for matrix multiplication. If the two input matrices are not of sizes that can be multiplied together, the behavior is unspecified. Unit test the function. *Hint: define functions for matrix transposition and row vector dot product.*

CHAPTER

SEVEN

MODULAR PROGRAMMING

When a program is small enough, we can keep all of the details of the program in our heads at once. But real-world applications can be many order of magnitude larger than those we write in college classes. They are simply too large and complex to hold all their details in our heads. They are also written by many programmers. To build large software systems requires techniques we haven't talked about so far.

One key solution to managing complexity of large software is *modular programming*: the code is composed of many different code modules that are developed separately. This allows different developers to take on discrete pieces of the system and design and implement them without having to understand all the rest. But to build large programs out of modules effectively, we need to be able to write modules that we can convince ourselves are correct *in isolation* from the rest of the program. Rather than have to think about every other part of the program when developing a code module, we need to be able to use *local reasoning*: that is, reasoning about just the module and the contract it needs to satisfy with respect to the rest of the program. If everyone has done their job, separately developed code modules can be plugged together to form a working program without every developer needing to understand everything done by every other developer in the team. This is the key idea of modular programming.

Therefore, to build large programs that work, we must use *abstraction* to make it manageable to think about the program. Abstraction is simply the removal of detail. A well-written program has the property that we can think about its components (such as functions) abstractly, without concerning ourselves with all the details of how those components are implemented.

Modules are abstracted by giving *specifications* of what they are supposed to do. A good module specification is clear, understandable, and gives just enough information about what the module does for clients to successfully use it. This abstraction makes the programmer's job much easier; it is helpful even when there is only one programmer working on a moderately large program, and it is crucial when there is more than one programmer.

Industrial-strength languages contain mechanisms that support modular programming. In general (i.e. across programming languages), a module specification is known as an *interface*, which provides information to clients about the module's functionality while hiding the *implementation*. Object-oriented languages support modular programming with *classes*. The Java `interface` construct is one example of a mechanism for specifying the interface to a class. A Java `interface` informs clients of the available functionality in any class that implements it without revealing the details of the implementation. But even just the public methods of a class constitute an interface in the more general sense—an abstract description of what the module can do.

Developers working with a module take on distinct roles. Most developers are usually *clients* of the module who understand the interface but do not need to understand the implementation of the module. A developer who works on the module implementation is naturally called an *implementer*. The module interface is a *contract* between the client and the implementer, defining the responsibilities of both. Contracts are very important because they help us to isolate the source of the problem when something goes wrong—and to know who to blame!

It is good practice to involve both clients and implementers in the design of a module's interface. Interfaces designed solely by one or the other can be seriously deficient. Each side will have its own view of what the final product should look like, and these may not align! So mutual agreement on the contract is essential. It is also important to think hard about global module structure and interfaces *early*, because changing an interface becomes more and more difficult as the development proceeds and more of the code comes to depend on it.

Modules should be used only through their declared interfaces, which the language should help to enforce. This is true even when the client and the implementer are the same person. Modules decouple the system design and implementation problem into separate tasks that can be carried out largely independently. When a module is used only through its interface, the implementer has the flexibility to change the module as long as the module still satisfies its interface.

7.1 Module Systems

A programming language's *module system* is the set of features it provides in support of modular programming. Below are some common concerns of module systems. We focus on Java and OCaml in this discussion, mentioning some of the most related features in the two languages.

Namespaces. A *namespace* provides a set of names that are grouped together, are usually logically related, and are distinct from other namespaces. That enables a name `foo` in one namespace to have a distinct meaning from `foo` in another namespace. A namespace is thus a scoping mechanism. Namespaces are essential for modularity. Without them, the names that one programmer chooses could collide with the names another programmer chooses. In Java, classes (and packages) group names. In OCaml, *structures* (which we will soon study) are similar to classes in that they group names — but without any of the added complexity of object-oriented programming that usually accompanies classes (constructors, static vs. instance members, inheritance, overriding, `this`, etc.) Structures are the core of the OCaml module system; in fact, we've been using them all along without thinking too much about them.

Abstraction. An *abstraction* hides some information while revealing other information. Abstraction thus enables *encapsulation*, aka *information hiding*. Usually, abstraction mechanisms for modules allow revealing some names that exist inside the module, but hiding some others. Abstractions therefore describe relationships among modules: there might be many modules that could considered to satisfy a given abstraction. Abstraction is essential for modularity, because it enables implementers of a module to hide the details of the implementation from clients, thus preventing the clients from abusing those details. In a large team, the modules one programmer designs are thereby protected from abuse by another programmer. It also enables clients to be blissfully unaware of those details. So, in a large team, no programmer has to be aware of all the details of all the modules. In Java, interfaces and abstract classes provide abstraction. In OCaml, *signatures* are used to abstract structures by hiding some of the structure's names and definitions. Signatures are essentially the types of structures.

Code reuse. A module system enables *code reuse* by providing features that enable code from one module to be used as part of another module without having to copy that code. Code reuse thereby enables programmers to build on the work of others in a way that is maintainable: when the implementer of one module makes an improvement in that module, all the programmers who are reusing that code automatically get the benefit of that improvement. Code reuse is essential for modularity, because it enables "building blocks" that can be assembled and reassembled to form complex pieces of software. In Java, subtyping and inheritance provide code reuse. In OCaml, *functors* and *includes* enable code reuse. Functors are like functions, in that they produce new modules out of old modules. Includes are like an intelligent form of copy-paste: they include code from one part of a program in another.

> **Warning:** These analogies between Java and OCaml are necessarily imperfect. You might naturally come away from the above discussion thinking either of the following:
>
> - "Structures are like Java classes, and signatures are like interfaces."
> - "Structures are like Java objects, and signatures are like classes."
>
> Both are helpful to a degree, yet both are ultimately wrong. So it might be best to let go of object-oriented programming at this point and come to terms with the OCaml module system in and of itself. Compared to Java, it's just built different.

7.2 Modules

We begin with a couple of examples of the OCaml module system before diving into the details.

A *structure* is simply a collection of definitions, such as:

```
struct
  let inc x = x + 1
  type primary_color = Red | Green | Blue
  exception Oops
end
```

In a way, the structure is like a record: the structure has some distinct components with names. But unlike a record, it can define new types, exceptions, and so forth.

By itself the code above won't compile, because structures do not have the same first-class status as values like integers or functions. You can't just enter that code in utop, or pass that structure to a function, etc. What you can do is bind the structure to a name:

```
module MyModule = struct
  let inc x = x + 1
  type primary_color = Red | Green | Blue
  exception Oops
end
```

The output from OCaml has the form:

```
module MyModule : sig ... end
```

This indicates that `MyModule` has been defined, and that it has been inferred to have the *module type* that appears to the right of the colon. That module type is written as *signature*:

```
sig
  val inc : int -> int
  type primary_color = Red | Green | Blue
  exception Oops
end
```

The signature itself is a collection of *specifications*. The specifications for variant types and exceptions are simply their original definitions, so `primary_color` and `Oops` are no different than they were in the original structure. The specification for `inc` though is written with the `val` keyword, exactly as the toplevel would respond if we defined `inc` in it.

Note: This use of the word "specification" is perhaps confusing, since many programmers would use that word to mean "the comments specifying the behavior of a function." But if we broaden our sight a little, we could allow that the type of a function is part of its specification. So it's at least a related sense of the word.

The definitions in a module are usually more closely related than those in `MyModule`. Often a module will implement some data structure. For example, here is a module for stacks implemented as linked lists:

```
module ListStack = struct
  (** [empty] is the empty stack. *)
  let empty = []
```

(continues on next page)

(continued from previous page)

```
  (** [is_empty s] is whether [s] is empty. *)
  let is_empty = function [] -> true | _ -> false

  (** [push x s] pushes [x] onto the top of [s]. *)
  let push x s = x :: s

  (** [Empty] is raised when an operation cannot be applied
      to an empty stack. *)
  exception Empty

  (** [peek s] is the top element of [s].
      Raises [Empty] if [s] is empty. *)
  let peek = function
    | [] -> raise Empty
    | x :: _ -> x

  (** [pop s] is all but the top element of [s].
      Raises [Empty] if [s] is empty. *)
  let pop = function
    | [] -> raise Empty
    | _ :: s -> s
end
```

Important: The specification of `pop` might surprise you. Note that it does not return the top element. That's the job of `peek`. Instead, `pop` returns all but the top element.

We can then use that module to manipulate a stack:

```
ListStack.push 2 (ListStack.push 1 ListStack.empty)
```

Warning: There's a common confusion lurking here for those programmers coming from object-oriented languages. It's tempting to think of `ListStack` as being an object on which you invoke methods. Indeed `ListStack.push` vaguely looks like we're invoking a `push` method on a `ListStack` object. But that's not what is happening. In an OO language you could instantiate many stack objects. But here, there is only one `ListStack`. Moreover it is not an object, in large part because it has no notion of a `this` or `self` keyword to denote the receiving object of the method call.

That's admittedly rather verbose code. Soon we'll see several solutions to that problem, but for now here's one:

```
ListStack.(push 2 (push 1 empty))
```

By writing `ListStack.(e)`, all the names from `ListStack` become usable in `e` without needing to write the prefix `ListStack.` each time. Another improvement could be using the pipeline operator:

```
ListStack.(empty |> push 1 |> push 2)
```

Now we can read the code left-to-right without having to parse parentheses. Nice.

> **Warning:** There's another common OO confusion lurking here. It's tempting to think of `ListStack` as being a class from which objects are instantiated. That's not the case though. Notice how there is no `new` operator used to create a stack above, nor any constructors (in the OO sense of that word).

Modules are considerably more basic than classes. A module is just a collection of definitions in its own namespace. In `ListStack`, we have some definitions of functions—`push`, `pop`, etc.—and one value, `empty`.

So whereas in Java we might create a couple of stacks using code like this:

```
Stack s1 = new Stack();
s1.push(1);
s1.push(2);
Stack s2 = new Stack();
s2.push(3);
```

In OCaml the same stacks could be created as follows:

```
let s1 = ListStack.(empty |> push 1 |> push 2)
let s2 = ListStack.(empty |> push 3)
```

7.2.1 Module Definitions

The `module` definition keyword is much like the `let` definition keyword that we learned before. (The OCaml designers hypothetically could have chosen to use `let_module` instead of `module` to emphasize the similarity.) The difference is just that:

- `let` binds a value to a name, whereas
- `module` binds a *module value* to a name.

Syntax.

The most common syntax for a module definition is simply:

```
module ModuleName = struct
  module_items
end
```

where `module_items` inside a structure can include `let` definitions, `type` definitions, and `exception` definitions, as well as nested `module` definitions. Module names must begin with an uppercase letter, and idiomatically they use `CamelCase` rather than `Snake_case`.

But a more accurate version of the syntax would be:

```
module ModuleName = module_expression
```

where a `struct` is just one sort of `module_expression`. Here's another: the name of an already defined module. For example, you can write `module L = List` if you'd like a short alias for the `List` module. We'll see other sorts of module expressions later in this section and chapter.

The definitions inside a structure can optionally be terminated by `;;` as in the toplevel:

```
module M = struct
  let x = 0;;
```

(continues on next page)

(continued from previous page)

```
  type t = int;;
end
```

Sometimes that can be useful to add temporarily if you are trying to diagnose a syntax error. It will help OCaml understand that you want two definitions to be syntactically separate. After fixing whatever the underlying error is, though, you can remove the `;;`.

One use case for `;;` is if you want to evaluate an expression as part of a module:

```
module M = struct
  let x = 0;;
  assert (x = 0);;
end
```

But that can be rewritten without `;;` as:

```
module M = struct
  let x = 0
  let _ = assert (x = 0)
end
```

Structures can also be written on a single line, with optional `;;` between items for readability:

```
module N = struct let x = 0 let y = 1 end
module O = struct let x = 0;; let y = 1 end
```

An empty structure is permitted:

```
module E = struct end
```

Dynamic semantics.

We already know that expressions are evaluated to values. Similarly, a module expression is evaluated to a *module value* or just "module" for short. The only interesting kind of module expression we have so far, from the perspective of evaluation anyway, is the structure. Evaluation of structures is easy: just evaluate each definition in it, in the order they occur. Because of that, earlier definitions are therefore in scope in later definitions, but not vice versa. So this module is fine:

```
module M = struct
  let x = 0
  let y = x
end
```

But this module is not, because at the time the `let` definition of `x` is being evaluated, `y` has not yet been bound:

```
module M = struct
  let x = y
  let y = 0
end
```

Of course, mutual recursion can be used if desired:

```
module M = struct
  let rec even = function 0 -> true | n -> odd (n - 1)
  and odd = function 1 -> true | n -> even (n - 1)
end
```

Static semantics.

A structure is well typed if all the definitions in it are themselves well-typed, according to all the typing rules we have already learned.

As we've seen in toplevel output, the module type of a structure is a signature. There's more to module types than that, though. Let's put that off for a moment to first talk about scope.

7.2.2 Scope and Open

After a module `M` has been defined, you can access the names within it using the dot operator. For example:

```
module M = struct let x = 42 end
```

```
M.x
```

Of course from outside the module the name `x` by itself is not meaningful:

```
x
```

But you can bring all of the definitions of a module into the current scope using `open`:

```
open M
```

```
x
```

Opening a module is like writing a local definition for each name defined in the module. For example, `open String` brings all the definitions from the String module into scope, and has an effect similar to the following on the local namespace:

```
let length = String.length
let get = String.get
let lowercase_ascii = String.lowercase_ascii
...
```

If there are types, exceptions, or modules defined in a module, those also are brought into scope with `open`.

The Always-Open Module. There is a special module called `Stdlib` that is automatically opened in every OCaml program. It contains the "built-in" functions and operators. You therefore never need to prefix any of the names it defines with `Stdlib.`, though you could do so if you ever needed to unambiguously identify a name from it. In earlier days, this module was named `Pervasives`, and you might still see that name in some code bases.

Open as a Module Item. An `open` is another sort of `module_item`. So we can open one module inside another:

```
module M = struct
  open List

  (** [uppercase_all lst] upper-cases all the elements of [lst]. *)
```

(continues on next page)

(continued from previous page)

```
  let uppercase_all = map String.uppercase_ascii
end
```

Since `List` is open, the name `map` from it is in scope. But what if we wanted to get rid of the `String.` as well?

```
module M = struct
  open List
  open String

  (** [uppercase_all lst] upper-cases all the elements of [lst]. *)
  let uppercase_all = map uppercase_ascii
end
```

Now we have a problem, because `String` also defines the name `map`, but with a different type than `List`. As usual a later definition shadows an earlier one, so it's `String.map` that gets chosen instead of `List.map` as we intended.

If you're using many modules inside your code, chances are you'll have at least one collision like this. Often it will be with a standard higher-order function like `map` that is defined in many library modules.

Tip: It is therefore generally good practice **not** to `open` all the modules you're going to use at the top of a `.ml` file or structure. This is perhaps different than how you're used to working with languages like Java, where you might `import` many packages with `*`. Instead, it's good to restrict the scope in which you open modules.

Limiting the Scope of Open. We've already seen one way of limiting the scope of an open: `M.(e)`. Inside `e` all the names from module `M` are in scope. This is useful for briefly using `M` in a short expression:

```
(* remove surrounding whitespace from [s] and convert it to lower case *)
let s = "BigRed "
let s' = s |> String.trim |> String.lowercase_ascii (* long way *)
let s'' = String.(s |> trim |> lowercase_ascii) (* short way *)
```

But what if you want to bring a module into scope for an entire function, or some other large block of code? The (admittedly strange) syntax for that is `let open M in e`. It makes all the names from `M` be in scope in `e`. For example:

```
(** [lower_trim s] is [s] in lower case with whitespace removed. *)
let lower_trim s =
  let open String in
  s |> trim |> lowercase_ascii
```

Going back to our `uppercase_all` example, it might be best to eschew any kind of opening and simply to be explicit about which module we are using where:

```
module M = struct
  (** [uppercase_all lst] upper-cases all the elements of [lst]. *)
  let uppercase_all = List.map String.uppercase_ascii
end
```

7.2.3 Module Type Definitions

We've already seen that OCaml will infer a signature as the type of a module. Let's now see how to write those modules types ourselves. As an example, here is a module type for our list-based stacks:

```
module type LIST_STACK = sig
  exception Empty
  val empty : 'a list
  val is_empty : 'a list -> bool
  val push : 'a -> 'a list -> 'a list
  val peek : 'a list -> 'a
  val pop : 'a list -> 'a list
end
```

Now that we have both a module and a module type for list-based stacks, we should move the specification comments from the structure into the signature. Those comments are properly part of the specification of the names in the signature. They specify behavior, thus augmenting the specification of types provided by the `val` declarations.

```
module type LIST_STACK = sig
  (** [Empty] is raised when an operation cannot be applied
      to an empty stack. *)
  exception Empty

  (** [empty] is the empty stack. *)
  val empty : 'a list

  (** [is_empty s] is whether [s] is empty. *)
  val is_empty : 'a list -> bool

  (** [push x s] pushes [x] onto the top of [s]. *)
  val push : 'a -> 'a list -> 'a list

  (** [peek s] is the top element of [s].
      Raises [Empty] if [s] is empty. *)
  val peek : 'a list -> 'a

  (** [pop s] is all but the top element of [s].
      Raises [Empty] if [s] is empty. *)
  val pop : 'a list -> 'a list
end

module ListStack = struct
  let empty = []

  let is_empty = function [] -> true | _ -> false

  let push x s = x :: s

  exception Empty

  let peek = function
    | [] -> raise Empty
    | x :: _ -> x

  let pop = function
    | [] -> raise Empty
    | _ :: s -> s
```

(continues on next page)

(continued from previous page)

```
end
```

Nothing so far, however, tells OCaml that there is a relationship between `LIST_STACK` and `ListStack`. If we want OCaml to ensure that `ListStack` really does have the module type specified by `LIST_STACK`, we can add a type annotation in the first line of the `module` definition:

```
module ListStack : LIST_STACK = struct
  let empty = []

  let is_empty = function [] -> true | _ -> false

  let push x s = x :: s

  exception Empty

  let peek = function
    | [] -> raise Empty
    | x :: _ -> x

  let pop = function
    | [] -> raise Empty
    | _ :: s -> s
end
```

The compiler agrees that the module `ListStack` does define all the items specified by `LIST_STACK` with appropriate types. If we had accidentally omitted some item, the type annotation would have been rejected:

```
module ListStack : LIST_STACK = struct
  let empty = []

  let is_empty = function [] -> true | _ -> false

  let push x s = x :: s

  exception Empty

  let peek = function
    | [] -> raise Empty
    | x :: _ -> x

  (* [pop] is missing *)
end
```

Syntax.

The most common syntax for a module type is simply:

```
module type ModuleTypeName = sig
  specifications
end
```

where `specifications` inside a signature can include `val` declarations, type definitions, exception definitions, and nested `module type` definitions. Like structures, a signature can be written on many lines or just one line, and the empty signature `sig end` is allowed.

But, as we saw with module definitions, a more accurate version of the syntax would be:

```
module type ModuleTypeName = module_type
```

where a signature is just one sort of `module_type`. Another would be the name of an already defined module type—e.g., `module type LS = LIST_STACK`. We'll see other module types later in this section and chapter.

By convention, module type names are usually `CamelCase`, like module names. So why did we use `ALL_CAPS` above for `LIST_STACK`? It was to avoid a possible point of confusion in that example, which we now illustrate. We could instead have used `ListStack` as the name of both the module and the module type:

```
module type ListStack = sig ... end
module ListStack : ListStack = struct ... end
```

In OCaml the namespaces for modules and module types are distinct, so it's perfectly valid to have a module named `ListStack` and and module type named `ListStack`. The compiler will not get confused about which you mean, because they occur in distinct syntactic contexts. But as a human you might well get confused by those seemingly overloaded names.

Note: The use of `ALL_CAPS` for module types was at one point common, and you might see it still. It's an older convention from Standard ML. But the social conventions of all caps have changed since those days. To modern readers, a name like `LIST_STACK` might feel like your code is impolitely shouting at you. That is a connotation that evolved in the 1980s. Older programming languages (e.g., Pascal, COBOL, FORTRAN) commonly used all caps for keywords and even their own names. Modern languages still idiomatically use all caps for constants—see, for example, Java's `Math.PI` or Python's style guide.

More Syntax.

We should also add syntax now for module type annotations. Module definitions may include an optional type annotation:

```
module ModuleName : module_type = module_expression
```

And module expressions may include manual type annotations:

```
(module_expression : module_type)
```

That syntax is analogous to how we can write `(e : t)` to manually specify the type `t` of an expression `e`.

Here are a few examples to show how that syntax can be used:

```
module ListStackAlias : LIST_STACK = ListStack
(* equivalently *)
module ListStackAlias = (ListStack : LIST_STACK)

module M : sig val x : int end = struct let x = 42 end
(* equivalently *)
module M = (struct let x = 42 end : sig val x : int end)
```

And, module types can include nested module specifications:

```
module type X = sig
  val x : int
end

module type T = sig
  module Inner : X
```

(continues on next page)

(continued from previous page)

```
end

module M : T = struct
  module Inner : X = struct
    let x = 42
  end
end
```

In the example above, `T` specifies that there must be an inner module named `Inner` whose module type is `X`. Here, the type annotation is mandatory, because otherwise nothing would be known about `Inner`. In implementing `T`, module `M` therefore has to provide a module (i) with that name, which also (ii) meets the specifications of module type `X`.

Dynamic semantics.

Since module types are in fact types, they are not evaluated. They have no dynamic semantics.

Static semantics.

Earlier in this section we delayed discussing the static semantics of module expressions. Now that we have learned about module types, we can return to that discussion. We do so, next, in its own section, because the discussion will be lengthy.

7.2.4 Module Type Semantics

If `M` is just a `struct` block, its module type is whatever signature the compiler infers for it. But that can be changed by module type annotations. The key question we have to answer is: what does a type annotation mean for modules? That is, what does it mean when we write the `: T` in `module M : T = ...`?

There are two properties the compiler guarantees:

1. *Signature matching:* every name declared in `T` is defined in `M` at the same or a more general type.
2. *Opacity:* any name defined in `M` that does not appear in `T` is not visible to code outside of `M`.

But a more complete answer turns out to involve *subtyping*, which is a concept you've probably seen before in an object-oriented language. We're going to take a brief detour into that realm now, then come back to OCaml and modules.

In Java, the `extends` keyword creates subtype relationships between classes:

```
class C { }
class D extends C { }

D d = new D();
C c = d;
```

Subtyping is what permits the assignment of `d` to `c` on the last line of that example. Because `D` extends `C`, Java considers `D` to be a subtype of `C`, and therefore permits an object instantiated from `D` to be used any place where an object instantiated from `C` is expected. It's up to the programmer of `D` to ensure that doesn't lead to any run-time errors, of course. The methods of `D` have to ensure that class invariants of `C` hold, for example. So by writing `D extends C`, the programmer is taking on some responsibility, and in turn gaining some flexibility by being able to write such assignment statements.

So what is a "subtype"? That notion is in many ways dependent on the language. For a language-independent notion, we turn to Barbara Liskov. She won the Turing Award in 2008 in part for her work on object-oriented language design. Twenty years before that, she invented what is now called the *Liskov Substitution Principle* to explain subtyping. It says that if `S` is a subtype of `T`, then substituting an object of type `S` for an object of type `T` should not change any desirable behaviors of a program. You can see that at work in the Java example above, both in terms of what the language allows and what the programmer must guarantee.

The particular flavor of subtyping in Java is called *nominal subtyping*, which is to say, it is based on names. In our example, `D` is a subtype of `C` just because of the way the names were declared. The programmer decreed that subtype relationship, and the language accepted the decree without question. Indeed the *only* subtype relationships that exist are those that have been decreed by name through such uses of `extends` and `implements`.

Now it's time to return to OCaml. Its module system also uses subtyping, with the same underlying intuition about the Liskov Substitution Principle. But OCaml uses a different flavor called *structural subtyping*. That is, it is based on the structure of modules rather than their names. "Structure" here simply means the definitions contained in the module. Those definitions are used to determine whether `(M : T)` is acceptable as a type annotation, where `M` is a module and `T` is a module type.

Let's play with this idea of structure through several examples, starting with this module:

```
module M = struct
  let x = 0
  let z = 2
end
```

Module `M` contains two definitions. You can see those in the signature for the module that OCaml outputs: it contains `x : int` and `z : int`. Because of the former, the module type annotation below is accepted:

```
module type X = sig
  val x : int
end

module MX = (M : X)
```

Module type `X` requires a module item named `x` with type `int`. Module `M` does contain such an item. So `(M : X)` is valid. The same would work for `z`:

```
module type Z = sig
  val z : int
end

module MZ = (M : Z)
```

Or for both `x` and `z`:

```
module type XZ = sig
  val x : int
  val z : int
end

module MXZ = (M : XZ)
```

But not for `y`, because `M` contains no such item:

```
module type Y = sig
  val y : int
end

module MY = (M : Y)
```

Take a close look at that error message. Learning to read such errors on small examples will help you when they appear in large bodies of code. OCaml is comparing two signatures, corresponding to the two expressions on either side of the colon in `(M : Y)`. The line

```
sig val x : int val z : int end
```

is the signature that OCaml is using for `M`. Since `M` is a module, that signature is just the names and types as they were defined in `M`. OCaml compares that signature to `Y`, and discovers a mismatch:

```
The value `y' is required but not provided
```

That's because `Y` requires `y` but `M` provides no such definition.

Here's another error message to practice reading:

```
module type Xstring = sig
  val x : string
end

module MXstring = (M : Xstring)
```

This time the error is

```
Values do not match: val x : int is not included in val x : string
```

The error changed, because `M` does provide a definition of `x`, but at a different type than `Xstring` requires. That's what "is not included in" means here. So why doesn't OCaml say something a little more straightforward, like "is not the same as"? It's because the types do not have to be exactly the same. If the provided value's type is polymorphic, it suffices for the required value's type to be an instantiation of that polymorphic type.

For example, if a signature requires a type `int -> int`, it suffices for a structure to provide a value of type `'a -> 'a`:

```
module type IntFun = sig
  val f : int -> int
end

module IdFun = struct
  let f x = x
end

module Iid = (IdFun : IntFun)
```

So far all these examples were just a matter of comparing the definitions required by a signature to the definitions provided by a structure. But here's an example that might be surprising:

```
module MXZ' = ((M : X) : Z)
```

Why does OCaml complain that `z` is required but not provided? We know from the definition of `M` that it indeed does have a value `z : int`. Yet the error message perhaps strangely claims:

```
The value `z' is required but not provided.
```

The reason for this error is that we've already supplied the type annotation `X` in the module expression `(M : X)`. That causes the module expression to be known only at the module type `X`. In other words, we've forgotten irrevocably about the existence of `z` after that annotation. All that is known is that the module has items required by `X`.

After all those examples, here are the static semantics of module type annotations:

- Module type annotation `(M : T)` is valid if the module type of `M` is a subtype of `T`. The module type of `(M : T)` is then `T` in any further type checking.
- Module type `S` is a subtype of `T` if the set of definitions in `S` is a superset of those in `T`. Definitions in `T` are permitted to instantiate type variables from `S`.

The "sub" vs. "super" in the second rule is not a typo. Consider these module types and modules:

```
module type T = sig
  val a : int
end

module type S = sig
  val a : int
  val b : bool
end

module A = struct
  let a = 0
end

module AB = struct
  let a = 0
  let b = true
end

module AC = struct
  let a = 0
  let c = 'c'
end
```

Module type `S` provides a *super*set of the definitions in `T`, because it adds a definition of `b`. So why is `S` called a *sub*type of `T`? Think about the set $Type(T)$ of all module values `M` such that `M : T`. That set contains `A`, `AB`, `AC`, and many others. Also think about the set $Type(S)$ of all module values `M` such that `M : S`. That set contains `AB` but not `A` nor `AC`. So $Type(S) \subset Type(T)$, because there are some module values that are in $Type(T)$ but not in $Type(S)$.

As another example, a module type `StackHistory` for stacks might customize our usual `Stack` signature by adding an operation `history : 'a t -> int` to return how many items have ever been pushed on the stack in its history. That `history` operation makes the set of definitions in `StackHistory` bigger than the set in `Stack`, hence the use of "superset" in the rule above. But the set of module values that implement `StackHistory` is smaller than the set of module values that implement `Stack`, hence the use of "subset".

7.2.5 Module Types are Static

Decisions about validity of module type annotations are made at compile time rather than run time.

Important: Module type annotations therefore offer potential confusion to programmers accustomed to object-oriented languages, in which subtyping works differently.

Python programmers, for example, are accustomed to so-called "duck typing". They might expect `((M : X) : Z)` to be valid, because `z` does exist at run-time in `M`. But in OCaml, the compile-time type of `(M : X)` has hidden `z` from view irrevocably.

Java programmers, on the other hand, might expect that module type annotations work like type casts. So it might seem valid to first "cast" `M` to `X` then to `Z`. In Java such type casts are checked, as needed, at run time. But OCaml module type

annotations are static. Once an annotation of `X` is made, there is no way to check at compile time what other items might exist in the module—that would require a run-time check, which OCaml does not permit.

In both cases it might feel as though OCaml is being too restrictive. Maybe. But in return for that restrictiveness, OCaml is guaranteeing an **absence of run-time errors** of the kind that would occur in Java or Python, whether because of a run-time error from a cast, or a run-time error from a missing method.

7.2.6 First-Class Modules

Modules are not as first-class in OCaml as functions. But it is possible to *package* modules as first-class values. Briefly:

- `(module M : T)` packages module `M` with module type `T` into a value.
- `(val e : T)` un-packages `e` into a module with type `T`.

We won't cover this much further, but if you're curious you can have a look at the manual.

7.3 Modules and the Toplevel

Note: The video below uses the legacy build system, ocamlbuild, rather than the new build system, dune. Some of the details change with dune, as described in the text below.

There are several pragmatics involving modules and the toplevel that are important to master to use the two together effectively.

7.3.1 Loading Compiled Modules

Compiling an OCaml file produces a module having the same name as the file, but with the first letter capitalized. These compiled modules can be loaded into the toplevel using `#load`.

For example, suppose you create a file called `mods.ml`, and put the following code in it:

```
let b = "bigred"
let inc x = x + 1
module M = struct
  let y = 42
end
```

Note that there is no `module Mods = struct ... end` around that. The code is at the topmost level of the file, as it were.

Then suppose you type `ocamlc mods.ml` to compile it. One of the newly-created files is `mods.cmo`: this is a compiled module object file, aka bytecode.

You can make this bytecode available for use in the toplevel with the following directives. Recall that the `#` character is required in front of a directive. It is not part of the prompt.

```
# #load "mods.cmo";;
```

That directive loads the bytecode found in `mods.cmo`, thus making a module named `Mods` available to be used. It is exactly as if you had entered this code:

```
module Mods = struct
  let b = "bigred"
  let inc x = x + 1
  module M = struct
    let y = 42
  end
end
```

Both of these expressions will therefore evaluate successfully:

```
Mods.b;;
Mods.M.y;;
```

But this will fail:

```
inc
```

It fails because `inc` is in the namespace of `Mods`.

```
Mods.inc
```

Of course, if you open the module, you can directly name `inc`:

```
open Mods;;
inc;;
```

7.3.2 Dune

Dune provides a command to make it easier to start utop with libraries already loaded. Suppose we add this dune file to the same directory as `mods.ml`:

```
(library
 (name mods))
```

That tells dune to build a library named `Mods` out of `mods.ml` (and any other files in the same directory, if they existed). Then we can run this command to launch utop with that library already loaded:

```
$ dune utop
```

Now right away we can access components of `Mods` without having to issue a `#load` directive:

```
Mods.inc
```

The `dune utop` command accepts a directory name as an argument if you want to load libraries in a particular subdirectory of your source code.

7.3.3 Initializing the Toplevel

If you are doing a lot of testing of a particular module, it can be annoying to have to type directives every time you start utop. You really want to initialize the toplevel with some code as it launches, so that you don't have to keep typing that code.

The solution is to create a file in the working directory and call that file `.ocamlinit`. Note that the `.` at the front of that filename is required and makes it a hidden file that won't appear in directory listings unless explicitly requested (e.g., with `ls -a`). Everything in `.ocamlinit` will be processed by utop when it loads.

For example, suppose you create a file named `.ocamlinit` in the same directory as `mods.ml`, and in that file put the following code:

```
open Mods;;
```

Now restart utop with `dune utop`. All the names defined in `Mods` will already be in scope. For example, these will both succeed:

```
inc;;
M.y;;
```

7.3.4 Requiring Libraries

Suppose you wanted to experiment with some OUnit code in utop. You can't actually open it:

```
open OUnit2;;
```

The problem is that the OUnit library hasn't been loaded into utop yet. It can be with the following directive:

```
#require "ounit2";;
```

Now you can successfully load your own module without getting an error.

```
open OUnit2;;
```

7.3.5 Load vs Use

There is a big difference between `#load`-ing a compiled module file and `#use`-ing an uncompiled source file. The former loads bytecode and makes it available for use. For example, loading `mods.cmo` caused the `Mod` module to be available, and we could access its members with expressions like `Mod.b`. The latter (`#use`) is *textual inclusion*: it's like typing the contents of the file directly into the toplevel. So using `mods.ml` does **not** cause a `Mod` module to be available, and the definitions in the file can be accessed directly, e.g., `b`.

For example, in the following interaction, we can directly refer to `b` but cannot use the qualified name `Mods.b`:

```
# #use "mods.ml"

# b;;
val b : string = "bigred"

# Mods.b;;
Error: Unbound module Mods
```

Whereas in this interaction the situation is reversed:

```
# #directory "_build";;
# #load "mods.cmo";;

# Mods.b;;
- : string = "bigred"

# b;;
Error: Unbound value b
```

So when you're using the toplevel to experiment with your code, it's often better to work with `#load` rather than `#use`. The `#load` directive accurately reflects how your modules interact with each other and with the outside world.

7.4 Encapsulation

One of the main concerns of a module system is to provide *encapsulation*: the hiding of information about implementation behind an interface. OCaml's module system makes this possible with a feature we've already seen: the *opacity* that module type annotations create. One special use of opacity is the declaration of *abstract types*. We'll study both of those ideas in this section.

7.4.1 Opacity

When implementing a module, you might sometimes have helper functions that you don't want to expose to clients of the module. For example, maybe you're implementing a math module that provides a tail-recursive factorial function:

```
module Math = struct
  (** [fact_aux n acc] is [n! * acc]. *)
  let rec fact_aux n acc =
    if n = 0 then acc else fact_aux (n - 1) (n * acc)

  (** [fact n] is [n!]. *)
  let fact n = fact_aux n 1
end
```

You'd like to make `fact` usable by clients of `Math`, but you'd also like to keep `fact_aux` hidden. But in the code above, you can see that `fact_aux` is visible in the signature inferred for `Math`. One way to hide it is simply to nest `fact_aux`:

```
module Math = struct
  (** [fact n] is [n!]. *)
  let fact n =
    (** [fact_aux n acc] is [n! * acc]. *)
    let rec fact_aux n acc =
      if n = 0 then acc else fact_aux (n - 1) (n * acc)
    in
    fact_aux n 1
end
```

Look at the signature, and notice how `fact_aux` is gone. But, that nesting makes `fact` just a little harder to read. It also means `fact_aux` is not available for any other functions *inside* `Math` to use. In this case that's probably fine—there probably aren't any other functions in `Math` that need `fact_aux`. But if there were, we couldn't nest `fact_aux`.

So another way to hide `fact_aux` from clients of `Math`, while still leaving it available for implementers of `Math`, is to use a module type that exposes only those names that clients should see:

```
module type MATH = sig
  (** [fact n] is [n!]. *)
  val fact : int -> int
end

module Math : MATH = struct
  (** [fact_aux n acc] is [n! * acc]. *)
  let rec fact_aux n acc =
    if n = 0 then acc else fact_aux (n - 1) (n * acc)

  let fact n = fact_aux n 1
end
```

Now since `MATH` does not mention `fact_aux`, the module type annotation `Math : MATH` causes `fact_aux` to be hidden:

```
Math.fact_aux
```

In that sense, module type annotations are *opaque*: they can prevent visibility of module items. We say that the module type *seals* the module, making any components not named in the module type be inaccessible.

Important: Remember that module type annotations are therefore not *only* about checking to see whether a module defines certain items. The annotations also hide items.

What if you did want to just check the definitions, but not hide anything? Then don't supply the annotation at the time of module definition:

```
module type MATH = sig
  (** [fact n] is [n!]. *)
  val fact : int -> int
end

module Math = struct
  (** [fact_aux n acc] is [n! * acc]. *)
  let rec fact_aux n acc =
    if n = 0 then acc else fact_aux (n - 1) (n * acc)

  let fact n = fact_aux n 1
end

module MathCheck : MATH = Math
```

Now `Math.fact_aux` is visible, but `MathCheck.fact_aux` is not:

```
Math.fact_aux
```

```
MathCheck.fact_aux
```

You wouldn't even have to give the "check" module a name since you probably never intend to access it; you could instead leave it anonymous:

```
module _ : MATH = Math
```

A Comparison to Visibility Modifiers. The use of sealing in OCaml is thus similar to the use of visibility modifiers such as `private` and `public` in Java. In fact one way to think about Java class definitions is that they simultaneously define multiple signatures.

For example, consider this Java class:

```
class C {
  private int x;
  public int y;
}
```

An analogy to it in OCaml would be the following modules and types:

```
module type C_PUBLIC = sig
  val y : int
end

module CPrivate = struct
  let x = 0
  let y = 0
end

module C : C_PUBLIC = CPrivate
```

With those definitions, any code that uses `C` will have access only to the names exposed in the `C_PUBLIC` module type.

That analogy can be extended to the other visibility modifiers, `protected` and default, as well. Which means that Java classes are effectively defining four related types, and the compiler is making sure the right type is used at each place in the code base `C` is named. No wonder it can be challenging to master visibility in OO languages at first.

7.4.2 Abstract Types

In an earlier section we implemented stacks as lists with the following module and type:

```
module type LIST_STACK = sig
  (** [Empty] is raised when an operation cannot be applied
      to an empty stack. *)
  exception Empty

  (** [empty] is the empty stack. *)
  val empty : 'a list

  (** [is_empty s] is whether [s] is empty. *)
  val is_empty : 'a list -> bool

  (** [push x s] pushes [x] onto the top of [s]. *)
  val push : 'a -> 'a list -> 'a list

  (** [peek s] is the top element of [s].
      Raises [Empty] if [s] is empty. *)
  val peek : 'a list -> 'a

  (** [pop s] is all but the top element of [s].
```

(continues on next page)

(continued from previous page)

```
      Raises [Empty] if [s] is empty. *)
  val pop : 'a list -> 'a list
end

module ListStack : LIST_STACK = struct
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push x s = x :: s
  let peek = function [] -> raise Empty | x :: _ -> x
  let pop = function [] -> raise Empty | _ :: s -> s
end
```

What if we wanted to modify that data structure to add an operation for the size of the stack? The easy way would be to implement it using `List.length`:

```
module type LIST_STACK = sig
  ...
  (** [size s] is the number of elements on the stack. *)
  val size : 'a list -> int
end

module ListStack : LIST_STACK = struct
  ...
  let size = List.length
end
```

That results in a linear-time implementation of `size`. What if we wanted a faster, constant-time implementation? At the cost of a little space, we could cache the size of the stack. Let's now represent the stack as a pair, where the first component of the pair is the same list as before, and the second component of the pair is the size of the stack:

```
module ListStackCachedSize = struct
  exception Empty
  let empty = ([], 0)
  let is_empty = function ([], _) -> true | _ -> false
  let push x (stack, size) = (x :: stack, size + 1)
  let peek = function ([], _) -> raise Empty | (x :: _, _) -> x
  let pop = function
    | ([], _) -> raise Empty
    | (_ :: stack, size) -> (stack, size - 1)
end
```

We have a big problem. `ListStackCachedSize` does not implement the `LIST_STACK` module type, because that module type specifies `'a list` throughout it to represent the stack—not `'a list * int`.

```
module CheckListStackCachedSize : LIST_STACK = ListStackCachedSize
```

Moreover, any code we previously wrote using `ListStack` now has to be modified to deal with the pair, which could mean revising pattern matches, function types, and so forth.

As you no doubt learned in earlier programming courses, the problem we are encountering here is a lack of encapsulation. We should have kept the type that implements `ListStack` hidden from clients. In Java, for example, we might have written:

```
class ListStack<T> {
  private List<T> stack;
  private int size;
  ...
}
```

That way clients of `ListStack` would be unaware of `stack` or `size`. In fact, they wouldn't be able to name those fields at all. Instead, they would just use `ListStack` as the type of the stack:

```
ListStack<Integer> s = new ListStack<>();
s.push(1);
```

So in OCaml, how can we keep the *representation type* of the stack hidden? What we learned about opacity and sealing thus far does not suffice. The problem is that the type `'a list * int` literally appears in the signature of `ListStackCachedSize`, e.g. in `push`:

```
ListStackCachedSize.push
```

A module type annotation could hide one of the values defined in `ListStackCachedSize`, e.g., `push` itself, but that doesn't solve the problem: we need to **hide the type** `'a list * int` while **exposing the operation** `push`. So OCaml has a feature for doing exactly that: *abstract types*. Let's see an example of this feature.

We begin by modifying `LIST_STACK`, replacing `'a list` with a new type `'a stack` everywhere. We won't repeat the specification comments here, so as to keep the example shorter. And while we're at it, let's add the `size` operation.

```
module type LIST_STACK = sig
  type 'a stack
  exception Empty
  val empty : 'a stack
  val is_empty : 'a stack -> bool
  val push : 'a -> 'a stack -> 'a stack
  val peek : 'a stack -> 'a
  val pop : 'a stack -> 'a stack
  val size : 'a stack -> int
end
```

Note how `'a stack` is not actually defined in that signature. We haven't said anything about what it is. It might be `'a list`, or `'a list * int`, or `{stack : 'a list; size : int}`, or anything else. That is what makes it an *abstract* type: we've declared its name but not specified its definition.

Now `ListStackCachedSize` can implement that module type with the addition of just one line of code: the first line of the structure, which defines `'a stack`:

```
module ListStackCachedSize : LIST_STACK = struct
  type 'a stack = 'a list * int
  exception Empty
  let empty = ([], 0)
  let is_empty = function ([], _) -> true | _ -> false
  let push x (stack, size) = (x :: stack, size + 1)
  let peek = function ([], _) -> raise Empty | (x :: _, _) -> x
  let pop = function
    | ([], _) -> raise Empty
    | (_ :: stack, size) -> (stack, size - 1)
  let size = snd
end
```

Take a careful look at the output: nowhere does `'a list` show up in it. In fact, only `LIST_STACK` does. And `LIST_STACK` mentions only `'a stack`. So no one's going to know that internally a list is used. (Ok, they're going to know: the name suggests it. But the point is they can't take advantage of that, because the type is abstract.)

Likewise, our original implementation with linear-time `size` satisfies the module type. We just have to add a line to define `'a stack`:

```
module ListStack : LIST_STACK = struct
  type 'a stack = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push x s = x :: s
  let peek = function [] -> raise Empty | x :: _ -> x
  let pop = function [] -> raise Empty | _ :: s -> s
  let size = List.length
end
```

Note that omitting that added line would result in an error, just as if we had failed to define `push` or any of the other operations from the module type:

```
module ListStack : LIST_STACK = struct
  (* type 'a stack = 'a list *)
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push x s = x :: s
  let peek = function [] -> raise Empty | x :: _ -> x
  let pop = function [] -> raise Empty | _ :: s -> s
  let size = List.length
end
```

Here is a third, custom implementation of `LIST_STACK`. This one is deliberately overly-complicated, in part to illustrate how the abstract type can hide implementation details that are better not revealed to clients:

```
module CustomStack : LIST_STACK = struct
  type 'a entry = {top : 'a; rest : 'a stack; size : int}
  and 'a stack = S of 'a entry option
  exception Empty
  let empty = S None
  let is_empty = function S None -> true | _ -> false
  let size = function S None -> 0 | S (Some {size}) -> size
  let push x s = S (Some {top = x; rest = s; size = size s + 1})
  let peek = function S None -> raise Empty | S (Some {top}) -> top
  let pop = function S None -> raise Empty | S (Some {rest}) -> rest
end
```

Is that really a "list" stack? It satisfies the module type `LIST_STACK`. But upon reflection, that module type never really had anything to do with lists once we made the type `'a stack` abstract. There's really no need to call it `LIST_STACK`. We'd be better off using just `STACK`, since it can be implemented with `list` or without. At that point, we could just go with `Stack` as its name, since there is no module named `Stack` we've written that would be confused with it. That avoids the all-caps look of our code shouting at us.

```
module type Stack = sig
  type 'a stack
  exception Empty
```

(continues on next page)

(continued from previous page)

```
  val empty : 'a stack
  val is_empty : 'a stack -> bool
  val push : 'a -> 'a stack -> 'a stack
  val peek : 'a stack -> 'a
  val pop : 'a stack -> 'a stack
  val size : 'a stack -> int
end

module ListStack : Stack = struct
  type 'a stack = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push x s = x :: s
  let peek = function [] -> raise Empty | x :: _ -> x
  let pop = function [] -> raise Empty | _ :: s -> s
  let size = List.length
end
```

There's one further naming improvement we could make. Notice the type of `ListStack.empty` (and don't worry about the `abstr` part for now; we'll come back to it):

```
ListStack.empty
```

That type, `'a ListStack.stack`, is rather unwieldy, because it conveys the word "stack" twice: once in the name of the module, and again in the name of the representation type inside that module. In places like this, OCaml programmers idiomatically use a standard name, `t`, in place of a longer representation type name:

```
module type Stack = sig
  type 'a t
  exception Empty
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> 'a
  val pop : 'a t -> 'a t
  val size : 'a t -> int
end

module ListStack : Stack = struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push x s = x :: s
  let peek = function [] -> raise Empty | x :: _ -> x
  let pop = function [] -> raise Empty | _ :: s -> s
  let size = List.length
end

module CustomStack : Stack = struct
  type 'a entry = {top : 'a; rest : 'a t; size : int}
  and 'a t = S of 'a entry option
  exception Empty
  let empty = S None
```

(continues on next page)

(continued from previous page)

```
  let is_empty = function S None -> true | _ -> false
  let size = function S None -> 0 | S (Some {size}) -> size
  let push x s = S (Some {top = x; rest = s; size = size s + 1})
  let peek = function S None -> raise Empty | S (Some {top}) -> top
  let pop = function S None -> raise Empty | S (Some {rest}) -> rest
end
```

Now the type of stacks is simpler:

```
ListStack.empty;;
CustomStack.empty;;
```

That idiom is fairly common when there's a single representation type exposed by an interface to a data structure. You'll see it used throughout the standard library.

In informal conversation we would usually pronounce those types without the "dot t" part. For example, we might say "alpha ListStack", simply ignoring the `t`—though it does technically have to be there to be legal OCaml code.

Finally, abstract types are really just a special case of opacity. You actually can expose the definition of a type in a signature if you want to:

```
module type T = sig
  type t = int
  val x : t
end

module M : T = struct
  type t = int
  let x = 42
end

let a : int = M.x
```

Note how we're able to use `M.x` at its type of `int`. That works because the equality of types `t` and `int` has been exposed in the module type. But if we kept `t` abstract, the same usage would fail:

```
module type T = sig
  type t (* = int *)
  val x : t
end

module M : T = struct
  type t = int
  let x = 42
end

let a : int = M.x
```

We're not allowed to use `M.x` at type `int` outside of `M`, because its type `M.t` is abstract. This is encapsulation at work, keeping that implementation detail hidden.

7.4.3 Pretty Printing

In some output above, we observed something curious: the toplevel prints `<abstr>` in place of the actual contents of a value whose type is abstract:

```
ListStack.empty;;
ListStack.(empty |> push 1 |> push 2);;
```

Recall that the toplevel uses this angle-bracket convention to indicate an unprintable value. We've encountered that before with functions and `<fun>`:

```
fun x -> x
```

On the one hand, it's reasonable for the toplevel to behave this way. Once a type is abstract its implementation isn't meant to be revealed to clients. So actually printing out the list `[]` or `[2; 1]` as responses to the above inputs would be revealing more than is intended.

On the other hand, it's also reasonable for implementers to provide clients with a friendly way to view a value of an abstract type. Java programmers, for example, will often write `toString()` methods so that objects can be printed as output in the terminal or in JShell. To support that, the OCaml toplevel has a directive `#install_printer`, which registers a function to print values. Here's how it works.

- You write a *pretty printing* function of type `Format.formatter -> t -> unit`, for whatever type `t` you like. Let's suppose for sake of example that you name that function `pp`.
- You invoke `#install_printer pp` in the toplevel.
- From now on, anytime the toplevel wants to print a value of type `t` it uses your function `pp` to do so.

It probably makes sense the pretty printing function needs to take in a value of type `t` (because that's what it needs to print) and returns `unit` (as other printing functions do). But why does it take the `Format.formatter` argument? It's because of a fairly high-powered feature that OCaml is attempting to provide here: automatic line breaking and indentation in the middle of very large outputs.

Consider the output from this expression, which creates nested lists:

```
List.init 15 (fun n -> List.init n (Fun.const n))
```

Each inner list contains `n` copies of the number `n`. Note how the indentation and line breaks are somewhat sophisticated. All the inner lists are indented one space from the left-hand margin. Line breaks have been inserted to avoid splitting inner lists over multiple lines.

The `Format` module is what provides this functionality, and `Format.formatter` is an abstract type in it. You could think of a formatter as being a place to send output, like a file, and have it be automatically formatted along the way. The typical use of a formatter is as argument to a function such as `Format.fprintf`, which like `Printf` uses format specifiers.

For example, suppose you wanted to change how strings are printed by the toplevel and add " kupo" to the end of each string. Here's code that would do it:

```
let kupo_pp fmt s = Format.fprintf fmt "%s kupo" s;;
#install_printer kupo_pp;;
```

Now you can see that the toplevel adds " kupo" to each string while printing it, even though it's not actual a part of the original string:

```
let h = "Hello"
let s = String.length h
```

To keep ourselves from getting confused about strings in the rest of this section, let's uninstall that pretty printer before going on:

```
#remove_printer kupo_pp;;
```

As a bigger example, let's add pretty printing to `ListStack`:

```
module type Stack = sig
  type 'a t
  exception Empty
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> 'a
  val pop : 'a t -> 'a t
  val size : 'a t -> int
  val pp :
    (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
end
```

First, notice that we have to expose `pp` as part of the module type. Otherwise it would be encapsulated, hence we wouldn't be able to install it. Second, notice that the type of `pp` now takes an extra first argument of type `Format.formatter -> 'a -> unit`. That is itself a pretty printer for type `'a`, on which `t` is parameterized. We need that argument in order to be able to pretty print the values of type `'a`.

```
module ListStack : Stack = struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push x s = x :: s
  let peek = function [] -> raise Empty | x :: _ -> x
  let pop = function [] -> raise Empty | _ :: s -> s
  let size = List.length
  let pp pp_val fmt s =
    let open Format in
    let pp_break fmt () = fprintf fmt "@," in
    fprintf fmt "@[<v 0>top of stack";
    if s <> [] then fprintf fmt "@,";
    pp_print_list ~pp_sep:pp_break pp_val fmt (List.rev s);
    fprintf fmt "@,bottom of stack@]"
end
```

In `ListStack.pp`, we use some of the advanced features of the `Format` module. Function `Format.pp_print_list` does the heavy lifting to print all the elements of the stack. The rest of the code handles the indentation and line breaks. Here's the result:

```
#install_printer ListStack.pp
```

```
ListStack.empty
```

```
ListStack.(empty |> push 1 |> push 2)
```

For more information, see the toplevel manual (search for `#install_printer`), the Format module, and this OCaml Github issue. The latter seems to be the only place that documents the use of extra arguments, as in `pp_val` above, to

print values of polymorphic types.

7.5 Compilation Units

A *compilation unit* is a pair of OCaml source files in the same directory. They share the same base name, call it `x`, but their extensions differ: one file is `x.ml`, the other is `x.mli`. The file `x.ml` is called the *implementation*, and `x.mli` is called the *interface*.

For example, suppose that `foo.mli` contains exactly the following:

```
val x : int
val f : int -> int -> int
```

and `foo.ml`, in the same directory, contains exactly the following:

```
let x = 0
let y = 12
let f x = x + y
```

Then compiling `foo.ml` will have the same effect as defining the module `Foo` as follows:

```
module Foo : sig
  val x : int
  val f : int -> int -> int
end = struct
  let x = 0
  let y = 12
  let f x = x + y
end
```

In general, when the compiler encounters a compilation unit, it treats it as defining a module and a signature like this:

```
module Foo
  : sig (* insert contents of foo.mli here *) end
= struct
  (* insert contents of foo.ml here *)
end
```

The *unit name* `Foo` is derived from the base name `foo` by just capitalizing the first letter. Notice that there is no named module type being defined; the signature of `Foo` is actually anonymous.

The standard library uses compilation units to implement most of the modules we have been using so far, like `List` and `String`. You can see that in the standard library source code.

7.5.1 Documentation Comments

Some documentation comments belong in the interface file, whereas others belong in the implementation file:

- Clients of an abstraction can be expected to read interface files, or rather the HTML documentation generated from them. So the comments in an interface file should be written with that audience in mind. These comments should describe how to use the abstraction, the preconditions for calling its functions, what exceptions they might raise, and perhaps some notes on what algorithms are used to implement operations. The standard library's `List` module contains many examples of these kinds of comments.
- Clients should not be expected to read implementation files. Those files will be read by creators and maintainers of the implementation. The documentation in the implementation file should provide information that explains the internal details of the abstraction, such as how the representation type is used, how the code works, important internal invariants it maintains, and so forth. Maintainers can also be expected to read the specifications in the interface files.

Documentation should **not** be duplicated between the files. In particular, the client-facing specification comments in the interface file should not be duplicated in the implementation file. One reason is that duplication inevitably leads to errors. Another reason is that OCamldoc has the ability to automatically inject the comments from the interface file into the generated HTML from the implementation file.

OCamldoc comments can be placed either before or after an element of the interface. For example, both of these placements are possible:

```
(** The mathematical constant 3.14... *)
val pi : float
```

```
val pi : float
(** The mathematical constant 3.14... *)
```

Tip: The standard library developers apparently prefer the post-placement of the comment, and OCamlFormat seems to work better with that, too.

7.5.2 An Example with Stacks

Put this code in `mystack.mli`, noting that there is no `sig..end` around it or any `module type`:

```
type 'a t
exception Empty
val empty : 'a t
val is_empty : 'a t -> bool
val push : 'a -> 'a t -> 'a t
val peek : 'a t -> 'a
val pop : 'a t -> 'a t
```

We're using the name "mystack" because the standard library already has a `Stack` module. Re-using that name could lead to error messages that are somewhat hard to understand.

Also put this code in `mystack.ml`, nothing that there is no `struct..end` around it or any `module`:

```
type 'a t = 'a list
exception Empty
let empty = []
```

(continues on next page)

(continued from previous page)

```
let is_empty = function [] -> true | _ -> false
let push = List.cons
let peek = function [] -> raise Empty | x :: _ -> x
let pop = function [] -> raise Empty | _ :: s -> s
```

Create a dune file:

```
(library
 (name mystack))
```

Compile the code and launch utop:

```
$ dune utop
```

Your compilation unit is ready for use:

```
# Mystack.empty;;
- : 'a Mystack.t = <abstr>
```

7.5.3 Incomplete Compilation Units

What if either the interface or implementation file is missing for a compilation unit?

Missing Interface Files. Actually this is exactly how we've normally been working up until this point. For example, you might have done some homework in a file named `lab1.ml` but never needed to worry about `lab1.mli`. There is no requirement that every `.ml` file have a corresponding `.mli` file, or in other words, that every compilation unit be complete.

If the `.mli` file is missing there is still a module that is created, as we saw back when we learned about `#load` and modules. It just doesn't have an automatically imposed signature. For example, the situation with `lab1` above would lead to the following module being created during compilation:

```
module Lab1 = struct
  (* insert contents of lab1.ml here *)
end
```

Missing Implementation Files. This case is much rarer, and not one you are likely to encounter in everyday development. But be aware that there is a **misuse** case that Java or C++ programmers sometimes accidentally fall into. Suppose you have an interface for which there will be a few implementation. Thinking back to stacks earlier in this chapter, perhaps you have a module type `Stack` and two modules that implement it, `ListStack` and `CustomStack`:

```
module type Stack = sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  (* etc. *)
end

module type ListStack : Stack = struct
  type 'a t = 'a list
  let empty = []
  let push = List.cons
```

(continues on next page)

(continued from previous page)

```
  (* etc. *)
end

module type CustomStack : Stack = struct
  (* omitted *)
end
```

It's tempting to divide that code up into files as follows:

```
(*********************************)
(* stack.mli *)
type 'a t
val empty : 'a t
val push : 'a -> 'a t -> 'a t
(* etc. *)

(*********************************)
(* listStack.ml *)
type 'a t = 'a list
let empty = []
let push = List.cons
(* etc. *)

(*********************************)
(* customStack.ml *)
(* omitted *)
```

The reason it's tempting is that in Java you might put the `Stack` interface into a `Stack.java` file, the `ListStack` class in a `ListStack.java` file, and so forth. In C++ something similar might be done with `.hpp` and `.cpp` files.

But the OCaml file organization shown above just won't work. To be a compilation unit, the interface for `listStack.ml` **must** be in `listStack.mli`. It can't be in a file with any other name. So there's no way with that code division to stipulate that `ListStack : Stack`.

Instead, the code could be divided like this:

```
(*********************************)
(* stack.ml *)
module type S = sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  (* etc. *)
end

(*********************************)
(* listStack.ml *)
module M : Stack.S = struct
  type 'a t = 'a list
  let empty = []
  let push = List.cons
  (* etc. *)
end

(*********************************)
(* customStack.ml *)
```

(continues on next page)

(continued from previous page)

```
module M : Stack.S = struct
  (* omitted *)
end
```

Note the following about that division:

- The module type goes in a `.ml` file not a `.mli`, because we're not trying to create a compilation unit.
- We give short names to the modules and module types in the files, because they will already be inside a module based on their filename. It would be rather verbose, for example, to name `S` something longer like `Stack`. If we did, we'd have to write `Stack.Stack` in the module type annotations instead of `Stack.S`.

Another possibility for code division would be to put all the code in a single file `stack.ml`. That works if all the code is part of the same library, but not if (e.g.) `ListStack` and `CustomStack` are developed by separate organizations. If it is in a single file, then we could turn it into a compilation unit:

```
(*******************************)
(* stack.mli *)
module type S = sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  (* etc. *)
end

module ListStack : S

module CustomStack : S

(*******************************)
(* stack.ml *)
module type S = sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  (* etc. *)
end

module ListStack : S = struct
  type 'a t = 'a list
  let empty = []
  let push = List.cons
  (* etc. *)
end

module CustomStack : S = struct
  (* omitted *)
end
```

Unfortunately that does mean we've duplicated `Stack.S` in both the interface and implementation files. There's no way to automatically "import" an already declared module type from a `.mli` file into the corresponding `.ml` file.

Code duplication naturally makes us unhappy. Later, with functors, we'll see how to eliminate it.

7.6 Functional Data Structures

A *functional data structure* is one that does not make use of mutability. It's possible to build functional data structures both in functional languages and in imperative languages. For example, you could build a Java equivalent to OCaml's `list` type by creating a `Node` class whose fields are immutable by virtue of using the `const` keyword.

Functional data structures have the property of being *persistent*: updating the data structure with one of its operations does not change the existing version of the data structure but instead produces a new version. Both exist and both can still be accessed. A good language implementation will ensure that any parts of the data structure that are not changed by an operation will be *shared* between the old version and the new version. Any parts that do change will be *copied* so that the old version may persist. The opposite of a persistent data structure is an *ephemeral* data structure: changes are destructive, so that only one version exists at any time. Both persistent and ephemeral data structures can be built in both functional and imperative languages.

7.6.1 Lists

The built-in singly-linked `list` data structure in OCaml is functional. We know that, because we've seen how to implement it with algebraic data types. It's also persistent, which we can demonstrate:

```
let lst = [1; 2];;
let lst' = List.tl lst;;
lst;;
```

Taking the tail of `lst` does not change the list. Both `lst` and `lst'` coexist without affecting one another.

7.6.2 Stacks

We implemented stacks earlier in this chapter. Here's a terse variant of one of those implementations, in which add a `to_list` operation to make it easier to view the contents of the stack in examples:

```
module type Stack = sig
  type 'a t
  exception Empty
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> 'a
  val pop : 'a t -> 'a t
  val size : 'a t -> int
  val to_list : 'a t -> 'a list
end

module ListStack : Stack = struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push = List.cons
  let peek = function [] -> raise Empty | x :: _ -> x
  let pop = function [] -> raise Empty | _ :: s -> s
  let size = List.length
  let to_list = Fun.id
end
```

That implementation is functional, as can be seen above, and also persistent:

```
open ListStack;;
let s = empty |> push 1 |> push 2;;
let s' = pop s;;
to_list s;;
to_list s';;
```

The value `s` is unchanged by the `pop` operation on `s'`. Both versions of the stack coexist.

The `Stack` module type gives us a strong hint that the data structure is persistent in the types it provides for `push` and `pop`:

```
val push : 'a -> 'a t -> 'a t
val pop : 'a t -> 'a t
```

Both of those take a stack as an argument and return a new stack as a result. An ephemeral data structure usually would not bother to return a stack. In Java, for example, similar methods might have a `void` return type; the equivalent in OCaml would be returning `unit`.

7.6.3 Options vs Exceptions

All of our stack implementations so far have raised an exception whenever `peek` or `pop` is applied to the empty stack. Another possibility would be to use an `option` for the return value. If the input stack is empty, then `peek` and `pop` return `None`; otherwise, they return `Some`.

```
module type Stack = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> 'a option
  val pop : 'a t -> 'a t option
  val size : 'a t -> int
  val to_list : 'a t -> 'a list
end

module ListStack : Stack = struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push = List.cons
  let peek = function [] -> None | x :: _ -> Some x
  let pop = function [] -> None | _ :: s -> Some s
  let size = List.length
  let to_list = Fun.id
end
```

But that makes it harder to pipeline:

```
ListStack.(empty |> push 1 |> pop |> peek)
```

The types break down for the pipeline right after the `pop`, because that now returns an `'a t option`, but `peek` expects an input that is merely an `'a t`.

It is possible to define some additional operators to help restore the ability to pipeline. In fact, these functions are already defined in the `Option` module in the standard library, though not as infix operators:

```
(* Option.map aka fmap *)
let ( >>| ) opt f =
  match opt with
  | None -> None
  | Some x -> Some (f x)

(* Option.bind *)
let ( >>= ) opt f =
  match opt with
  | None -> None
  | Some x -> f x
```

We can use those as needed for pipelining:

```
ListStack.(empty |> push 1 |> pop >>| push 2 >>= pop >>| push 3 >>| to_list)
```

But it's not so pleasant to figure out which of the three operators to use where.

There is therefore a tradeoff in the interface design:

- Using options ensures that surprising exceptions regarding empty stacks never occur at run-time. The program is therefore more robust. But the convenient pipeline operator is lost.
- Using exceptions means that programmers don't have to write as much code. If they are sure that an exception can't occur, they can omit the code for handling it. The program is less robust, but writing it is more convenient.

There is thus a tradeoff between writing more code early (with options) or doing more debugging later (with exceptions). The OCaml standard library has recently begun providing both versions of the interface in a data structure, so that the client can make the choice of how they want to use it. For example, we could provide both `peek` and `peek_opt`, and the same for `pop`, for clients of our stack module:

```
module type Stack = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> 'a
  val peek_opt : 'a t -> 'a option
  val pop : 'a t -> 'a t
  val pop_opt : 'a t -> 'a t option
  val size : 'a t -> int
  val to_list : 'a t -> 'a list
end

module ListStack : Stack = struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push = List.cons
  let peek = function [] -> raise Empty | x :: _ -> x
  let peek_opt = function [] -> None | x :: _ -> Some x
  let pop = function [] -> raise Empty | _ :: s -> s
  let pop_opt = function [] -> None | _ :: s -> Some s
  let size = List.length
```

(continues on next page)

(continued from previous page)

```
  let to_list = Fun.id
end
```

One nice thing about this implementation is that it is efficient. All the operations except for `size` are constant time. We saw earlier in the chapter that `size` could be made constant time as well, at the cost of some extra space — though just a constant factor more — by caching the size of the stack at each node in the list.

7.6.4 Queues

Queues and stacks are fairly similar interfaces. We'll stick with exceptions instead of options for now.

```
module type Queue = sig
  (** An ['a t] is a queue whose elements have type ['a]. *)
  type 'a t

  (** Raised if [front] or [dequeue] is applied to the empty queue. *)
  exception Empty

  (** [empty] is the empty queue. *)
  val empty : 'a t

  (** [is_empty q] is whether [q] is empty. *)
  val is_empty : 'a t -> bool

  (** [enqueue x q] is the queue [q] with [x] added to the end. *)
  val enqueue : 'a -> 'a t -> 'a t

  (** [front q] is the element at the front of the queue. Raises [Empty]
      if [q] is empty. *)
  val front : 'a t -> 'a

  (** [dequeue q] is the queue containing all the elements of [q] except the
      front of [q]. Raises [Empty] is [q] is empty. *)
  val dequeue : 'a t -> 'a t

  (** [size q] is the number of elements in [q]. *)
  val size : 'a t -> int

  (** [to_list q] is a list containing the elements of [q] in order from
      front to back. *)
  val to_list : 'a t -> 'a list
end
```

Important: Similarly to `peek` and `pop`, note how `front` and `dequeue` divide the responsibility of getting the first element vs. getting all the rest of the elements.

It's easy to implement queues with lists, just as it was for implementing stacks:

```
module ListQueue : Queue = struct
  (** The list [x1; x2; ...; xn] represents the queue with [x1] at its front,
      followed by [x2], ..., followed by [xn]. *)
  type 'a t = 'a list
```

(continues on next page)

(continued from previous page)

```
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let enqueue x q = q @ [x]
  let front = function [] -> raise Empty | x :: _ -> x
  let dequeue = function [] -> raise Empty | _ :: q -> q
  let size = List.length
  let to_list = Fun.id
end
```

But despite being as easy, this implementation is not as efficient as our list-based stacks. Dequeueing is a constant-time operation with this representation, but enqueueing is a linear-time operation. That's because `dequeue` does a single pattern match, whereas `enqueue` must traverse the entire list to append the new element at the end.

There's a very clever way to do better on efficiency. We can use two lists to represent a single queue. This representation was invented by Robert Melville as part of his PhD dissertation at Cornell (*Asymptotic Complexity of Iterative Computations*, Jan 1981), which was advised by Prof. David Gries. Chris Okasaki (*Purely Functional Data Structures*, Cambridge University Press, 1988) calls these *batched queues*. Sometimes you will see this same implementation referred to as "implementing a queue with two stacks". That's because stacks and lists are so similar (as we've already seen) that you could rewrite `pop` as `List.tl`, and so forth.

The core idea has a Part A and a Part B. Part A is: we use the two lists to split the queue into two pieces, the *inbox* and *outbox*. When new elements are enqueued, we put them in the inbox. Eventually (we'll soon come to how) elements are transferred from the inbox to the outbox. When a dequeue is requested, that element is removed from the outbox; or when the front element is requested, we check the outbox for it. For example, if the inbox currently had `[3; 4; 5]` and the outbox had `[1; 2]`, then the front element would be `1`, which is the head of the outbox. Dequeuing would remove that element and leave the inbox with just `[2]`, which is the tail of the outbox. Likewise, enqueuing `6` would make the inbox become `[3; 4; 5; 6]`.

The efficiency of `front` and `dequeue` is very good so far. We just have to take the head or tail of the outbox, respectively, assuming it is non-empty. Those are constant-time operations. But the efficiency of `enqueue` is still bad. It's linear time, because we have to append the new element to the end of the list. It's too bad we have to use the append operator, which is inherently linear time. It would be much better if we could use cons, which is constant time.

So here's Part B of the core idea: let's keep the inbox in reverse order. For example, if we enqueued `3` then `4` then `5`, the inbox would actually be `[5; 4; 3]`, not `[3; 4; 5]`. Then if `6` were enqueued next, we could cons it onto the beginning of the inbox, which becomes `[6; 5; 4; 3]`. The queue represented by inbox `i` and outbox `o` is therefore `o @ List.rev i`. So `enqueue` can now always be a constant-time operation.

But what about `dequeue` (and `front`)? They're constant time too, **as long as the outbox is not empty.** If it's empty, we have a problem. We need to transfer whatever is in the inbox to the outbox at that point. For example, if the outbox is empty, and the inbox is `[6; 5; 4; 3]`, then we need to switch them around, making the outbox be `[3; 4; 5; 6]` and the inbox be empty. That's actually easy: we just have to reverse the list.

Unfortunately, we just re-introduced a linear-time operation. But with one crucial difference: we don't have to do that linear-time reverse on every `dequeue`, whereas with `ListQueue` above we had to do the linear-time append on every `enqueue`. Instead, we only have to do the reverse on those rare occasions when the outbox becomes empty.

So even though in the worst case `dequeue` (and `front`) will be linear time, most of the time they will not be. In fact, later in this book when we study *amortized analysis* we will show that in the long run they can be understood as constant-time operations. For now, here's a piece of intuition to support that claim: every individual element enters the outbox once (with a cons), moves to the inbox once (with a pattern match then cons), and leaves the inbox once (with a pattern match). Each of those is constant time. So each element only ever experiences constant-time operations from its own perspective.

For now, let's move on to implementing these ideas. In the implementation, we'll add one more idea: the outbox always has to have an element in it, unless the queue is empty. In other words, if the outbox is empty, we're guaranteed the inbox

is too. That requirement isn't necessary for batched queues, but it does keep the code simpler by reducing the number of times we have to check whether a list is empty. The tiny tradeoff is that if the queue is empty, `enqueue` now has to directly put an element into the outbox. No matter, that's still a constant-time operation.

```
module BatchedQueue : Queue = struct
  (** [{o; i}] represents the queue [o @ List.rev i]. For example,
      [{o = [1; 2]; i = [5; 4; 3]}] represents the queue [1, 2, 3, 4, 5],
      where [1] is the front element. To avoid ambiguity about emptiness,
      whenever only one of the lists is empty, it must be [i]. For example,
      [{o = [1]; i = []}] is a legal representation, but [{o = []; i = [1]}]
      is not. This implies that if [o] is empty, [i] must also be empty. *)
  type 'a t = {o : 'a list; i : 'a list}

  exception Empty

  let empty = {o = []; i = []}

  let is_empty = function
    | {o = []} -> true
    | _ -> false

  let enqueue x = function
    | {o = []} -> {o = [x]; i = []}
    | {o; i} -> {o; i = x :: i}

  let front = function
    | {o = []} -> raise Empty
    | {o = h :: _} -> h

  let dequeue = function
    | {o = []} -> raise Empty
    | {o = [_]; i} -> {o = List.rev i; i = []}
    | {o = _ :: t; i} -> {o = t; i}

  let size {o; i} = List.(length o + length i)

  let to_list {o; i} = o @ List.rev i
end
```

The efficiency of batched queues comes at a price in readability. If we compare `ListQueue` and `BatchedQueue`, it's hopefully clear that `ListQueue` is a simple and correct implementation of a queue data structure. It's probably far less clear that `BatchedQueue` is a correct implementation. Just look at how many paragraphs of writing it took to explain it above!

7.6.5 Maps

Recall that a *map* (aka *dictionary*) binds keys to values. Here is a module type for maps. There are many other operations a map might support, but these will suffice for now.

```
module type Map = sig
  (** [('k, 'v) t] is the type of maps that bind keys of type ['k] to
      values of type ['v]. *)
  type ('k, 'v) t

  (** [empty] does not bind any keys. *)
```

(continues on next page)

(continued from previous page)

```
  val empty  : ('k, 'v) t

  (** [insert k v m] is the map that binds [k] to [v], and also contains
      all the bindings of [m].  If [k] was already bound in [m], that old
      binding is superseded by the binding to [v] in the returned map. *)
  val insert : 'k -> 'v -> ('k, 'v) t -> ('k, 'v) t

  (** [lookup k m] is the value bound to [k] in [m]. Raises: [Not_found] if [k]
      is not bound in [m]. *)
  val lookup : 'k -> ('k, 'v) t -> 'v

  (** [bindings m] is an association list containing the same bindings as [m].
      The keys in the list are guaranteed to be unique. *)
  val bindings : ('k, 'v) t -> ('k * 'v) list
end
```

Note how `Map.t` is parameterized on two types, `'k` and `'v`, which are written in parentheses and separated by commas. Although `('k, 'v)` might look like a pair of values, it is not: it is a syntax for writing multiple type variables.

Recall that association lists are lists of pairs, where the first element of each pair is a key, and the second element is the value it binds. For example, here is an association list that maps some well-known names to an approximation of their numeric value:

```
[("pi", 3.14); ("e", 2.718); ("phi", 1.618)]
```

Naturally we can implement the `Map` module type with association lists:

```
module AssocListMap : Map = struct
  (** The list [(k1, v1); ...; (kn, vn)] binds key [ki] to value [vi].
      If a key appears more than once in the list, it is bound to the
      the left-most occurrence in the list. *)
  type ('k, 'v) t = ('k * 'v) list
  let empty = []
  let insert k v m = (k, v) :: m
  let lookup k m = List.assoc k m
  let keys m = List.(m |> map fst |> sort_uniq Stdlib.compare)
  let bindings m = m |> keys |> List.map (fun k -> (k, lookup k m))
end
```

This implementation of maps is persistent. For example, adding a new binding to the map `m` below does not change `m` itself:

```
open AssocListMap
let m = empty |> insert "pi" 3.14 |> insert "e" 2.718
let m' = m |> insert "phi" 1.618
let b = bindings m
let b' = bindings m'
```

The `insert` operation is constant time, which is great. But the `lookup` operation is linear time. It's possible to do much better than that. In a later chapter, we'll see how to do better. Logarithmic-time performance is achievable with balanced binary trees, and something like constant-time performance with hash tables. Neither of those, however, achieves the simplicity of the code above.

The `bindings` operation is complicated by potential duplicate keys in the list. It uses a `keys` helper function to extract the unique list of keys with the help of library function `List.sort_uniq`. That function sorts an input list and in the process discards duplicates. It requires a comparison function as input.

Note: A comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller.

Here we use the standard library's comparison function `Stdlib.compare`, which behaves essentially the same as the built-in comparison operators =, <, >, etc. Custom comparison functions are useful if you want to have a relaxed notion of what being a duplicate means. For example, maybe you'd like to ignore the case of strings, or the sign of a number, etc.

The running time of `List.sort_uniq` is linearithmic, and it produces a linear number of keys as output. For each of those keys, we do a linear-time lookup operation. So the total running time of `bindings` is $O(n \log n) + O(n) \cdot O(n)$, which is $O(n^2)$. We can definitely do better than that with more advanced data structures.

Actually we can have a constant-time `bindings` operation even with association lists, if we are willing to pay for a linear-time `insert` operation:

```
module UniqAssocListMap : Map = struct
  (** The list [(k1, v1); ...; (kn, vn)] binds key [ki] to value [vi].
      No duplicate keys may occur. *)
  type ('k, 'v) t = ('k * 'v) list
  let empty = []
  let insert k v m = (k, v) :: List.remove_assoc k m
  let lookup k m = List.assoc k m
  let bindings m = m
end
```

That implementation removes any duplicate binding of `k` before inserting a new binding.

7.6.6 Sets

Here is a module type for sets. There are many other operations a set data structure might be expected to support, but these will suffice for now.

```
module type Set = sig
  (** ['a t] is the type of sets whose elements are of type ['a]. *)
  type 'a t

  (** [empty] is the empty set *)
  val empty : 'a t

  (** [mem x s] is whether [x] is an element of [s]. *)
  val mem : 'a -> 'a t -> bool

  (** [add x s] is the set that contains [x] and all the elements of [s]. *)
  val add : 'a -> 'a t -> 'a t

  (** [elements s] is a list containing the elements of [s].  No guarantee
      is made about the ordering of that list, but each is guaranteed to
      be unique. *)
  val elements : 'a t -> 'a list
end
```

Here's an implementation of that interface using a list to represent the set. This implementation ensures that the list never contains any duplicate elements, since sets themselves do not:

```
module UniqListSet : Set = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add x s = if mem x s then s else x :: s
  let elements = Fun.id
end
```

Note how `add` ensures that the representation never contains any duplicates, so the implementation of `elements` is easy. Of course, that comes with the tradeoff of `add` being linear time.

Here's a second implementation, which permits duplicates in the list:

```
module ListSet : Set = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add = List.cons
  let elements s = List.sort_uniq Stdlib.compare s
end
```

In that implementation, the `add` operation is now constant time, and the `elements` operation is linearithmic time.

7.7 Module Type Constraints

We have extolled the virtues of encapsulation. Now we're going to do something that might seem counter-intuitive: selectively violate encapsulation.

As a motivating example, here is a module type that represents values that support the usual addition and multiplication operations from arithmetic, or more precisely, a *ring*:

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val ( + ) : t -> t -> t
  val ( * ) : t -> t -> t
  val ( ~- ) : t -> t  (* additive inverse *)
  val to_string : t -> string
end
```

Recall that we must write `( * )` instead of `(*)` because the latter would be parsed as beginning a comment. And we write the `~` in `( ~- )` to indicate a *unary* operator.

This is a bit weird of an example. We don't normally think of numbers as a data structure. But what is a data structure except for a set of values and operations on them? The `Ring` module type makes it clear that's what we have.

Here is a module that implements that module type:

```
module IntRing : Ring = struct
  type t = int
  let zero = 0
  let one = 1
  let ( + ) = Stdlib.( + )
```

(continues on next page)

(continued from previous page)

```
  let ( * ) = Stdlib.( * )
  let ( ~- ) = Stdlib.( ~- )
  let to_string = string_of_int
end
```

Because `t` is abstract, the toplevel can't give us good output about what the sum of one and one is:

```
IntRing.(one + one)
```

But we could convert it to a string:

```
IntRing.(one + one |> to_string)
```

We could even install a pretty printer to avoid having to manually call `to_string`:

```
let pp_intring fmt i =
  Format.fprintf fmt "%s" (IntRing.to_string i);;

#install_printer pp_intring;;

IntRing.(one + one)
```

We could implement other kinds of rings, too:

```
module FloatRing : Ring = struct
  type t = float
  let zero = 0.
  let one = 1.
  let ( + ) = Stdlib.( +. )
  let ( * ) = Stdlib.( *. )
  let ( ~- ) = Stdlib.( ~-. )
  let to_string = string_of_float
end
```

Then we'd have to install a printer for it, too:

```
let pp_floatring fmt f =
  Format.fprintf fmt "%s" (FloatRing.to_string f);;

#install_printer pp_floatring;;

FloatRing.(one + one)
```

Was there really a need to make type `t` abstract in the ring examples above? Arguably not. And if it were not abstract, we wouldn't have to go to the trouble of converting abstract values into strings, or installing printers. Let's pursue that idea, next.

7.7.1 Specializing Module Types

In the past, we've seen that we can leave off the module type annotation, then do a separate check to make sure the structure satisfies the signature:

```
module IntRing = struct
  type t = int
  let zero = 0
  let one = 1
  let ( + ) = Stdlib.( + )
  let ( * ) = Stdlib.( * )
  let ( ~- ) = Stdlib.( ~- )
  let to_string = string_of_int
end

module _ : Ring = IntRing
```

```
IntRing.(one + one)
```

There's a more sophisticated way of accomplishing the same goal. We can specialize the `Ring` module type to specify that `t` must be `int` or `float`. We do that by adding a *constraint* using the `with` keyword:

```
module type INT_RING = Ring with type t = int
```

Note how the `INT_RING` module type now specifies that `t` and `int` are the same type. It exposes or *shares* that fact with the world, so we could call these "sharing constraints."

Now `IntRing` can be given that module type:

```
module IntRing : INT_RING = struct
  type t = int
  let zero = 0
  let one = 1
  let ( + ) = Stdlib.( + )
  let ( * ) = Stdlib.( * )
  let ( ~- ) = Stdlib.( ~- )
  let to_string = string_of_int
end
```

And since the equality of `t` and `int` is exposed, the toplevel can print values of type `t` without any help needed from a pretty printer:

```
IntRing.(one + one)
```

Programmers can even mix and match built-in `int` values with those provided by `IntRing`:

```
IntRing.(1 + one)
```

The same can be done for floats:

```
module type FLOAT_RING = Ring with type t = float

module FloatRing : FLOAT_RING = struct
  type t = float
  let zero = 0.
```

(continues on next page)

(continued from previous page)

```
  let one = 1.
  let ( + ) = Stdlib.( +. )
  let ( * ) = Stdlib.( *. )
  let ( ~- ) = Stdlib.( ~-. )
  let to_string = string_of_float
end
```

It turns out there's no need to separately define `INT_RING` and `FLOAT_RING`. The `with` keyword can be used as part of the `module` definition, though the syntax becomes a little harder to read because of the proximity of the two = signs:

```
module FloatRing : Ring with type t = float = struct
  type t = float
  let zero = 0.
  let one = 1.
  let ( + ) = Stdlib.( +. )
  let ( * ) = Stdlib.( *. )
  let ( ~- ) = Stdlib.( ~-. )
  let to_string = string_of_float
end
```

7.7.2 Constraints

Syntax.

There are two sorts of constraints. One is the sort we saw above, with `type` equations:

- `T with type x = t`, where `T` is a module type, `x` is a type name, and `t` is a type.

The other sort is a `module` equation, which is syntactic sugar for specifying the equality of *all* types in the two modules:

- `T with module M = N`, where `M` and `N` are module names.

Multiple constraints can be added with the `and` keyword:

- `T with constraint1 and constraint2 and ... constraintN`

Static semantics.

The constrained module type `T with type x = t` is the same as `T`, except that the declaration of `type x` inside `T` is replaced by `type x = t`. For example, compare the two signatures output below:

```
module type T = sig type t end
module type U = T with type t = int
```

Likewise, `T with module M = N` is the same as `T`, except that the any declaration `type x` inside the module type of `M` is replaced by `type x = N.x`. (And the same recursively for any nested modules.) It takes more work to give and understand this example:

```
module type XY = sig
  type x
  type y
end

module type T = sig
  module A : XY
end
```

(continues on next page)

(continued from previous page)

```
module B = struct
  type x = int
  type y = float
end

module type U = T with module A = B

module C : U = struct
  module A = struct
    type x = int
    type y = float
    let x = 42
  end
end
```

Focus on the output for module type `U`. Notice that the types of `x` and `y` in it have become `int` and `float` because of the `module A = B` constraint. Also notice how modules `B` and `C.A` are *not* the same module; the latter has an extra item `x` in it. So the syntax `module A = B` is potentially confusing. The constraint is not specifying that the two *modules* are the same. Rather, it specifies that all their *types* are constrained to be equal.

Dynamic semantics.

There are no dynamic semantics for constraints, because they are only for type checking.

7.8 Includes

Copying and pasting code is almost always a bad idea. Duplication of code causes duplication and proliferation of errors. So why are we so prone to making this mistake? Maybe because it always seems like the easier option — easier and quicker than applying the Abstraction Principle as we should to factor out common code.

The OCaml module system provides a neat feature called *includes* that is like a principled copy-and-paste that is quick and easy to use, but avoids actual duplication. It can be used to solve some of the same problems as *inheritance* in object-oriented languages.

Let's start with an example. Recall this implementation of sets as lists:

```
module type Set = sig
  type 'a t
  val empty : 'a t
  val mem : 'a -> 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val elements : 'a t -> 'a list
end

module ListSet : Set = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add = List.cons
  let elements s = List.sort_uniq Stdlib.compare s
end
```

Suppose we wanted to add a function `of_list : 'a list -> 'a t` that could construct a set out of a list. If we had access to the source code of both `ListSet` and `Set`, and if we were permitted to modify it, this wouldn't be hard.

But what if they were third-party libraries for which we didn't have source code?

In Java, we might use inheritance to solve this problem:

```
interface Set<T> { ... }
class ListSet<T> implements Set<T> { ... }
class ListSetExtended<T> extends ListSet<T> {
  Set<T> ofList(List<T> lst) { ... }
}
```

That helps us to reuse code, because the subclass inherits all the methods of its superclass.

OCaml *includes* are similar. They enable a module to include all the items defined by another module, or a module type to include all the specifications of another module type.

Here's how we can use includes to solve the problem of adding `of_list` to `ListSet`:

```
module ListSetExtended = struct
  include ListSet
  let of_list lst = List.fold_right add lst empty
end
```

This code says that `ListSetExtended` is a module that includes all the definitions of the `ListSet` module, as well as a definition of `of_list`. We don't have to know the source code implementing `ListSet` to make this happen.

Note: You might wonder we why can't simply implement `of_list` as the identity function. See the section below on encapsulation for the answer.

7.8.1 Semantics of Includes

Includes can be used inside of structures and signatures. When we include inside a signature, we must be including another signature. And when we include inside a structure, we must be including another structure.

Including a structure is effectively just syntactic sugar for writing a local definition for each name defined in the module. Writing `include ListSet` as we did above, for example, has an effect similar to writing the following:

```
module ListSetExtended = struct
  (* BEGIN all the includes *)
  type 'a t = 'a ListSet.t
  let empty = ListSet.empty
  let mem = ListSet.mem
  let add = ListSet.add
  let elements = ListSet.elements
  (* END all the includes *)
  let of_list lst = List.fold_right add lst empty
end
```

None of that is actually copying the source code of `ListSet`. Rather, the `include` just creates a new definition in `ListSetExtended` with the same name as each definition in `ListSet`. But if the set of names defined inside `ListSet` ever changed, the `include` would reflect that change, whereas a copy-paste job would not.

Including a signature is much the same. For example, we could write:

```
module type SetExtended = sig
  include Set
  val of_list : 'a list -> 'a t
end
```

Which would have an effect similar to writing the following:

```
module type SetExtended = sig
  (* BEGIN all the includes *)
  type 'a t
  val empty : 'a t
  val mem : 'a -> 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val elements  : 'a t -> 'a list
  (* END all the includes *)
  val of_list : 'a list -> 'a t
end
```

That module type would be suitable for `ListSetExtended`:

```
module ListSetExtended : SetExtended = struct
  include ListSet
  let of_list lst = List.fold_right add lst empty
end
```

7.8.2 Encapsulation and Includes

We mentioned above that you might wonder why we didn't write this simpler definition of `of_list`:

```
module ListSetExtended : SetExtended = struct
  include ListSet
  let of_list lst = lst
end
```

Check out that error message. It looks like `of_list` doesn't have the right type. What if we try adding some type annotations?

```
module ListSetExtended : SetExtended = struct
  include ListSet
  let of_list (lst : 'a list) : 'a t = lst
end
```

Ah, now the problem is clearer: in the body of `of_list`, the equality of `'a t` and `'a list` isn't known. In `ListSetExtended`, we do know that `'a t = 'a ListSet.t`, because that's what the `include` gave us. But the fact that `'a ListSet.t = 'a list` was hidden when `ListSet` was sealed at module type `Set`. So, includes must obey encapsulation, just like the rest of the module system.

One workaround is to rewrite the definitions as follows:

```
module ListSetImpl = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
```

(continues on next page)

(continued from previous page)

```
  let add = List.cons
  let elements s = List.sort_uniq Stdlib.compare s
end

module ListSet : Set = ListSetImpl

module type SetExtended = sig
  include Set
  val of_list : 'a list -> 'a t
end

module ListSetExtendedImpl = struct
  include ListSetImpl
  let of_list lst = lst
end

module ListSetExtended : SetExtended = ListSetExtendedImpl
```

The important change is that `ListSetImpl` is not sealed, so its type `'a t` is not abstract. When we include it in `ListSetExtended`, we can therefore exploit the fact that it's a synonym for `'a list`.

What we just did is effectively the same as what Java does to handle the visibility modifiers `public`, `private`, etc. The "private version" of a class is like the `Impl` version above: anyone who can see that version gets to see all the exposed items (fields in Java, types in OCaml), without any encapsulation. The "public version" of a class is like the sealed version above: anyone who can see that version is forced to treat the items as abstract, hence encapsulated.

With that technique, if we want to provide a new implementation of one of the included functions we *could* do that too:

```
module ListSetExtendedImpl = struct
  include ListSetImpl
  let of_list lst = List.fold_right add lst empty
  let rec elements = function
    | [] -> []
    | h :: t -> if mem h t then elements t else h :: elements t
end
```

But that's a bad idea. First, it's actually a quadratic implementation of `elements` instead of linearithmic. Second, it does not *replace* the original implementation of `elements`. Remember the semantics of modules: all definitions are evaluated from top to bottom, in order. So the new definition of `elements` above won't come into use until the very end of evaluation. If any earlier functions had happened to use `elements` as a helper, they would use the original linearithmic version, not the new quadratic version.

Warning: This differs from what you might expect from Java, which uses a language feature called dynamic dispatch to figure out which method implementation to invoke. Dynamic dispatch is arguably *the* defining feature of object-oriented languages. OCaml functions are not methods, and they do not use dynamic dispatch.

7.8.3 Include vs. Open

The `include` and `open` statements are quite similar, but they have a subtly different effect on a structure. Consider this code:

```
module M = struct
  let x = 0
end

module N = struct
  include M
  let y = x + 1
end

module O = struct
  open M
  let y = x + 1
end
```

Look closely at the values contained in each structure. `N` has both an `x` and `y`, whereas `O` has only a `y`. The reason is that `include M` causes all the definitions of `M` to also be included in `N`, so the definition of `x` from `M` is present in `N`. But `open M` only made those definitions available in the *scope* of `O`; it doesn't actually make them part of the *structure*. So `O` does not contain a definition of `x`, even though `x` is in scope during the evaluation of `O`'s definition of `y`.

A metaphor for understanding this difference might be: `open M` imports definitions from `M` and makes them available for local consumption, but they aren't exported to the outside world. Whereas `include M` imports definitions from `M`, makes them available for local consumption, and additionally exports them to the outside world.

7.8.4 Including Code in Multiple Modules

Recall that we also had an implementation of sets that made sure every element of the underlying list was unique:

```
module UniqListSet : Set = struct
  (** All values in the list must be unique. *)
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add x s = if mem x s then s else x :: s
  let elements = Fun.id
end
```

Suppose we wanted add `of_list` to that module too. One possibility would be to copy and paste that function from `ListSet` into `UniqListSet`. But that's poor software engineering. So let's rule that out right away as a non-solution.

Instead, suppose we try to define the function outside of either module:

```
let of_list lst = List.fold_right add lst empty
```

The problem is we either need to choose which module's `add` and `empty` we want. But as soon as we do, the function becomes useful only with that one module:

```
let of_list lst = List.fold_right ListSet.add lst ListSet.empty
```

We could make `add` and `empty` be parameters instead:

```
let of_list' add empty lst = List.fold_right add lst empty

let of_list lst = of_list' ListSet.add ListSet.empty lst
let of_list_uniq lst = of_list' UniqListSet.add UniqListSet.empty lst
```

But this is annoying in a couple of ways. First, we have to remember which function name to call, whereas all the other operations that are part of those modules have the same name, regardless of which module they're in. Second, the `of_list` functions live outside either module, so clients who open one of the modules won't automatically get the ability to name those functions.

Let's try to use includes to solve this problem. First, we write a module that contains the parameterized implementation:

```
module SetOfList = struct
  let of_list' add empty lst = List.fold_right add lst empty
end
```

Then we include that module to get the helper function:

```
module UniqListSetExtended : SetExtended = struct
  include UniqListSet
  include SetOfList
  let of_list lst = of_list' add empty lst
end

module ListSetExtended : SetExtended = struct
  include ListSet
  include SetOfList
  let of_list lst = of_list' add empty lst
end
```

That works, but we've only partially succeeded in achieving code reuse:

- On the positive side, the code that implements `of_list'` has been factored out into a single location and reused in the two structures.
- But on the negative side, we still had to write an implementation of `of_list` in both modules. Worse yet, those implementations are identical. So there's still code duplication occurring.

Could we do better? Yes. And that leads us to functors, next.

7.9 Functors

The problem we were having in the previous section was that we wanted to add code to two different modules, but that code needed to be parameterized on the details of the module to which it was being added. It's that kind of parameterization that is enabled by an OCaml language feature called *functors*.

Note: Why the name "functor"? In category theory, a *category* contains *morphisms*, which are a generalization of functions as we known them, and a *functor* is map between categories. Likewise, OCaml modules contain functions, and OCaml functors map from modules to modules.

The name is unfortunately intimidating, but **a functor is simply a "function" from modules to modules.** The word "function" is in quotation marks in that sentence only because it's a kind of function that's not interchangeable with the rest of the functions we've already seen. OCaml's type system is *stratified*: module values are distinct from other values,

so functions from modules to modules cannot be written or used in the same way as functions from values to values. But conceptually, functors really are just functions.

Here's a tiny example of a functor:

```
module type X = sig
  val x : int
end

module IncX (M : X) = struct
  let x = M.x + 1
end
```

The functor's name is `IncX`. It's essentially a function from modules to modules. As a function, it takes an input and produces an output. Its input is named `M`, and the type of its input is `X`. Its output is the structure that appears on the right-hand side of the equals sign: `struct let x = M.x + 1`.

Another way to think about `IncX` is that it's a *parameterized structure*. The parameter that it takes is named `M` and has type `X`. The structure itself has a single value named `x` in it. The value that `x` has will depend on the parameter `M`.

Since functors are essentially functions, we *apply* them. Here's an example of applying `IncX`:

```
module A = struct let x = 0 end
```

```
A.x
```

```
module B = IncX (A)
```

```
B.x
```

```
module C = IncX (B)
```

```
C.x
```

Each time, we pass `IncX` a module. When we pass it the module bound to the name `A`, the input to `IncX` is `struct let x = 0 end`. Functor `IncX` takes that input and produces an output `struct let x = A.x + 1 end`. Since `A.x` is `0`, the result is `struct let x = 1 end`. So `B` is bound to `struct let x = 1 end`. Similarly, `C` ends up being bound to `struct let x = 2 end`.

Although the functor `IncX` returns a module that is quite similar to its input module, that need not be the case. In fact, a functor can return any module it likes, perhaps something very different than its input structure:

```
module AddX (M : X) = struct
  let add y = M.x + y
end
```

Let's apply that functor to a module. The module doesn't even have to be bound to a name; we can just write an anonymous structure:

```
module Add42 = AddX (struct let x = 42 end)
```

```
Add42.add 1
```

Note that the input module to `AddX` contains a value named `x`, but the output module from `AddX` does not:

```
Add42.x
```

Warning: It's tempting to think that a functor is the same as `extends` in Java, and that the functor therefore extends the input module with new definitions while keeping the old definitions around too. The example above shows that is not the case. A functor is essentially just a function, and that function can return whatever the programmer wants. In fact the output of the functor could be arbitrarily different than the input.

7.9.1 Functor Syntax and Semantics

In the functor syntax we've been using:

```
module F (M : S) = ...
end
```

the type annotation `: S` and the parentheses around it, `(M : S)` are required. The reason why is that OCaml needs the type information about `S` to be provided in order to do a good job with type inference for `F` itself.

Much like functions, functors can be written anonymously. The following two syntaxes for functors are equivalent:

```
module F (M : S) = ...

module F = functor (M : S) -> ...
```

The second form uses the `functor` keyword to create an anonymous functor, like how the `fun` keyword creates an anonymous function.

And functors can be parameterized on multiple structures:

```
module F (M1 : S1) ... (Mn : Sn) = ...
```

Of course, that's just syntactic sugar for a *higher-order functor* that takes a structure as input and returns an anonymous functor:

```
module F = functor (M1 : S1) -> ... -> functor (Mn : Sn) -> ...
```

If you want to specify the output type of a functor, the syntax is again similar to functions:

```
module F (M : Si) : So = ...
```

As usual, it's also possible to write the output type annotation on the module expression:

```
module F (M : Si) = (... : So)
```

To evaluate an application `module_expression1 (module_expression2)`, the first module expression is evaluated and must produce a functor `F`. The second module expression is then evaluated to a module `M`. The functor is then applied to the module. The functor will be producing a new module `N` as part of that application. That new module is evaluated as always, in order of definition from top to bottom, with the definitions of `M` available for use.

7.9.2 Functor Type Syntax and Semantics

The simplest syntax for functor types is actually the same as for functions:

```
module_type -> module_type
```

For example, `X -> Add` below is a functor type, and it works for the `AddX` module we defined earlier in this section:

```
module type Add = sig val add : int -> int end
module CheckAddX : X -> Add = AddX
```

Functor type syntax becomes more complicated if the output module type is dependent upon the input module type. For example, suppose we wanted to create a functor that pairs up a value from one module with another value:

```
module type T = sig
  type t
  val x : t
end

module Pair1 (M : T) = struct
  let p = (M.x, 1)
end
```

The type of `Pair1` turns out to be:

```
functor (M : T) -> sig val p : M.t * int end
```

So we could also write:

```
module type P1 = functor (M : T) -> sig val p : M.t * int end

module Pair1 : P1 = functor (M : T) -> struct
  let p = (M.x, 1)
end
```

Module type `P1` is the type of a functor that takes an input module named `M` of module type `T`, and returns an output module whose module type is given by the signature `sig..end`. Inside the signature, the name `M` is in scope. That's why we can write `M.t` in it, thereby ensuring that the type of the first component of pair `p` is the type from the *specific* module `M` that is passed into `Pair1`, not any *other* module. For example, here are two different instantiations:

```
module P0 = Pair1 (struct type t = int let x = 0 end)
module PA = Pair1 (struct type t = char let x = 'a' end)
```

Note the difference between `int` and `char` in the resulting module types. It's important that the output module type of `Pair1` can distinguish those. And that's why `M` has to be nameable on the right-hand side of the arrow in `P1`.

Note: Functor types are an example of an advanced programming language feature called *dependent types*, with which the **type** of an output is determined by the **value** of an input. That's different than the normal case of a function, where it's the output **value** that's determined by the input value, and the output **type** is independent of the input value.

Dependent types enable type systems to express much more about the correctness of a program, but type checking and inference for dependent types is much more challenging. Practical dependent type systems are an active area of research. Perhaps someday they will become popular in mainstream languages.

The module type of a functor's actual argument need not be identical to the formal declared module type of the argument; it's fine to be a subtype. For example, it's fine to apply `F` below to either `X` or `Z`. The extra item in `Z` won't cause any difficulty.

```
module F (M : sig val x : int end) = struct let y = M.x end
module X = struct let x = 0 end
module Z = struct let x = 0;; let z = 0 end
module FX = F (X)
module FZ = F (Z)
```

7.9.3 The `Map` Module

The standard library's Map module implements a map (a binding from keys to values) using balanced binary trees. It uses functors in an important way. In this section, we study how to use it. You can see the implementation of that module on GitHub as well as its interface.

The Map module defines a functor `Make` that creates a structure implementing a map over a particular type of keys. That type is the input structure to `Make`. The type of that input structure is `Map.OrderedType`, which are types that support a `compare` operation:

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

The Map module needs ordering, because balanced binary trees need to be able to compare keys to determine whether one is greater than another. The `compare` function's specification is the same as that for the comparison argument to `List.sort_uniq`, which we previously discussed:

- The comparison should return `0` if two keys are equal.
- The comparison should return a strictly negative number if the first key is lesser than the second.
- The comparison should return a strictly positive number if the first key is greater than the second.

Note: Does that specification seem a little strange? Does it seem hard to remember when to return a negative vs. positive number? Why not define a variant instead?

```
type order = LT | EQ | GT
val compare : t -> t -> order
```

Alas, historically many languages have used comparison functions with similar specifications, such as the C standard library's `strcmp` function. When comparing two integers, it does make the comparison easy: just perform a subtraction. It's not necessarily so easy for other data types.

The output of `Map.Make` supports all the usual operations we would expect from a dictionary:

```
module type S = sig
  type key
  type 'a t
  val empty: 'a t
  val mem: key -> 'a t -> bool
  val add: key -> 'a -> 'a t -> 'a t
  val find: key -> 'a t -> 'a
```

(continues on next page)

(continued from previous page)

```
  ...
end
```

The type variable `'a` is the type of values in the map. So any particular map module created by `Map.Make` can handle only one type of key, but is not restricted to any particular type of value.

An Example Map

Here's an example of using the `Map.Make` functor:

```
module IntMap = Map.Make(Int)
```

If you show that output, you'll see the long module type of `IntMap`. The `Int` module is part of the standard library. Conveniently, it already defines the two items required by `OrderedType`, which are `t` and `compare`, with appropriate behaviors. The standard library also already defines modules for the other primitive types (`String`, etc.) that make it convenient to use any primitive type as a key.

Now let's try out that map by mapping an `int` to a `string`:

```
open IntMap;;
let m1 = add 1 "one" empty
```

```
find 1 m1
```

```
mem 42 m1
```

```
find 42 m1
```

```
bindings m1
```

The same `IntMap` module allows us to map an `int` to a `float`:

```
let m2 = add 1 1. empty
```

```
bindings m2
```

But the keys must be `int`, not any other type:

```
let m3 = add true "one" empty
```

That's because the `IntMap` module was specifically created for keys that are integers and ordered accordingly. Again, order is crucial, because the underlying data structure is a binary search tree, which requires key comparisons to figure out where in the tree to store a key. You can even see that in the standard library source code (v4.12), of which the following is a lightly-edited extract:

```
module Make (Ord : OrderedType) = struct
  type key = Ord.t

  type 'a t =
```

(continues on next page)

(continued from previous page)

```
    | Empty
    | Node of {l : 'a t; v : key; d : 'a; r : 'a t; h : int}
      (** left subtree, key, value/data, right subtree, height of node *)

  let empty = Empty

  let rec mem x = function
    | Empty -> false
    | Node {l, v, r} ->
      let c = Ord.compare x v in
      c = 0 || mem x (if c < 0 then l else r)
  ...
end
```

The `key` type is defined to be a synonym for the type `t` inside `Ord`, so `key` values are comparable using `Ord.compare`. The `mem` function uses that to compare keys and decide whether to recurse on the left subtree or right subtree.

Note how the implementor of `Map` had a tricky problem to solve: balanced binary search trees require a way to compare keys, but the implementor can't know in advance all the different types of keys that a client of the data structure will want to use. And each type of key might need its own comparison function. Although `Stdlib.compare` *can* be used to compare any two values of the same type, the result it returns isn't necessarily what a client will want. For example, it's not guaranteed to sort names in the way we wanted above.

So the implementor of `Map` used a functor to solve their problem. They parameterized on a module that bundles together the type of keys with a function that can be used to compare them. It's the client's responsibility to implement that module.

The Java Collections Framework solves a similar problem in the TreeMap class, which has a constructor that takes a Comparator. There, the client has the responsibility of implementing a class for comparisons, rather than a structure. Though the language features are different, the idea is the same.

Maps with Custom Key Types

When the type of a key becomes complicated, we might want to write our own custom comparison function. For example, suppose we want a map in which keys are records representing names, and in which names are sorted alphabetically by last name then by first name. In the code below, we provide a module `Name` that can compare records that way:

```
type name = {first : string; last : string}

module Name = struct
  type t = name
  let compare { first = first1; last = last1 } { first = first2; last = last2 }
      =
    match String.compare last1 last2 with
    | 0 -> String.compare first1 first2
    | c -> c
end
```

The `Name` module can be used as input to `Map.Make` because it satisfies the `Map.OrderedType` signature:

```
module NameMap = Map.Make (Name)
```

Now we could use that map to associate names with birth years:

```
let k1 = {last = "Kardashian"; first = "Kourtney"}
let k2 = {last = "Kardashian"; first = "Kimberly"}
let k3 = {last = "Kardashian"; first = "Khloe"}
let k4 = {last = "West"; first = "Kanye"}

let nm =
  NameMap.(empty |> add k1 1979 |> add k2 1980 |> add k3 1984 |> add k4 1977)

let lst = NameMap.bindings nm
```

Note how the order of keys in that list is not the same as the order in which we added them. The list is sorted according to the `Name.compare` function we wrote. Several of the other functions in the `Map.S` signature will also process map bindings in that sorted order—for example, `map`, `fold`, and `iter`.

How `Map` Uses Module Type Constraints

In the standard library's `map.mli` interface, the specification for `Map.Make` is:

```
module Make (Ord : OrderedType) : S with type key = Ord.t
```

The `with` constraint there is crucial. Recall that type constraints specialize a module type. Here, `S with type key = Ord.t` specializes `S` to expose the equality of `S.key` and `Ord.t`. In other words, the type of keys is the ordered type.

You can see the effect of that sharing constraint by looking at the module type of our `IntMap` example from before. The sharing constraint is what caused the `= Int.t` to be present:

```
module IntMap : sig
  type key = Int.t
  ...
end
```

And the `Int` module contains this line:

```
type t = int
```

So `IntMap.key = Int.t = int`, which is exactly why we're allowed to pass an `int` to the `add` and `mem` functions of `IntMap`.

Without the type constraint, type `key` would have remained abstract. We can simulate that by adding a module type annotation of `Map.S`, thereby resealing the module at that type without exposing the equality:

```
module UnusableMap = (IntMap : Map.S);;
```

Now it's impossible to add a binding to the map:

```
let m = UnusableMap.(empty |> add 0 "zero")
```

This kind of use case is why module type constraints are quite important in effective programming with the OCaml module system. Often it is necessary to specialize the output type of a functor to show a relationship between a type in it and a type in one of the functor's inputs. Thinking through exactly what constraint is necessary can be challenging, though!

7.9.4 Using Functors

With `Map` we saw one use case for functors: producing a data structure that was parameterized on a client-provided ordering. Here are two more use cases.

Test Suites

Here are two implementations of a stack:

```
exception Empty

module type Stack = sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> 'a
  val pop : 'a t -> 'a t
end

module ListStack = struct
  type 'a t = 'a list
  let empty = []
  let push = List.cons
  let peek = function [] -> raise Empty | x :: _ -> x
  let pop = function [] -> raise Empty | _ :: s -> s
end

module VariantStack = struct
  type 'a t = E | S of 'a * 'a t
  let empty = E
  let push x s = S (x, s)
  let peek = function E -> raise Empty | S (x, _) -> x
  let pop = function E -> raise Empty | S (_, s) -> s
end
```

Suppose we wanted to write an OUnit test for `ListStack`:

```
let test = "peek (push x empty) = x" >:: fun _ ->
  assert_equal 1 ListStack.(empty |> push 1 |> peek)
```

Unfortunately, to test a `VariantStack`, we'd have to duplicate that code:

```
let test' = "peek (push x empty) = x" >:: fun _ ->
  assert_equal 1 VariantStack.(empty |> push 1 |> peek)
```

And if we had other stack implementations, we'd have to duplicate the test for them, too. That's not so horrible to contemplate if it's just one test case for a couple implementations, but if it's hundreds of tests for even a couple implementations, that's just too much duplication to be good software engineering.

Functors offer a better solution. We can write a functor that is parameterized on the stack implementation, and produces the test for that implementation:

```
module StackTester (S : Stack) = struct
  let tests = [
    "peek (push x empty) = x" >:: fun _ ->
```

(continues on next page)

(continued from previous page)

```
      assert_equal 1 S.(empty |> push 1 |> peek)
  ]
end

module ListStackTester = StackTester (ListStack)
module VariantStackTester = StackTester (VariantStack)

let all_tests = List.flatten [
  ListStackTester.tests;
  VariantStackTester.tests
]
```

Now whenever we invent a new test we add it to `StackTester`, and it automatically gets run on both stack implementations. Nice!

There is still some objectionable code duplication, though, in that we have to write two lines of code per implementation. We can eliminate that duplication through the use of first-class modules:

```
let stacks = [ (module ListStack : Stack); (module VariantStack) ]

let all_tests =
  let tests m =
    let module S = (val m : Stack) in
    let module T = StackTester (S) in
    T.tests
  in
  let open List in
  stacks |> map tests |> flatten
```

Now it suffices just to add the newest stack implementation to the `stacks` list. Nicer!

Extending Multiple Modules

Earlier, we tried to add a function `of_list` to both `ListSet` and `UniqListSet` without having any duplicated code, but we didn't totally succeed. Now let's really do it right.

The problem we had earlier was that we needed to parameterize the implementation of `of_list` on the `add` function and `empty` value in the set module. We can accomplish that parameterization with a functor:

```
module type Set = sig
  type 'a t
  val empty : 'a t
  val mem : 'a -> 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val elements : 'a t -> 'a list
end

module SetOfList (S : Set) = struct
  let of_list lst = List.fold_right S.add lst S.empty
end
```

Notice how the functor, in its body, uses `S.add`. It takes the implementation of `add` from `S` and uses it to implement `of_list` (and the same for `empty`), thus solving the exact problem we had before when we tried to use includes.

When we apply `SetOfList` to our set implementations, we get modules containing an `of_list` function for each implementation:

```
module ListSet : Set = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add = List.cons
  let elements s = List.sort_uniq Stdlib.compare s
end

module UniqListSet : Set = struct
  (** All values in the list must be unique. *)
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add x s = if mem x s then s else x :: s
  let elements = Fun.id
end
```

```
module OfList = SetOfList (ListSet)
module UniqOfList = SetOfList (UniqListSet)
```

The functor has enabled the code reuse we couldn't get before: we now can implement a single `of_list` function and from it derive implementations for two different sets.

But that's the **only** function those two modules contain. Really what we want is a full set implementation that also contains the `of_list` function. We can get that by combining includes with functors:

```
module SetWithOfList (S : Set) = struct
  include S
  let of_list lst = List.fold_right S.add lst S.empty
end
```

That functor takes a set as input, and produces a module that contains everything from that set (because of the `include`) as well as a new function `of_list`.

When we apply the functor, we get a very nice set module:

```
module SetL = SetWithOfList (ListSet)
module UniqSetL = SetWithOfList (UniqListSet)
```

Notice how the output structure records the fact that its type `t` is the same type as the type `t` in its input structure. They share it because of the `include`.

Stepping back, what we just did bears more than a passing resemblance to class extension in Java. We created a base module and extended its functionality with new code while preserving its old functionality. But whereas class extension necessitates that the newly extended class is a subtype of the old, and that it still has all the old functionality, OCaml functors are more fine-grained in what they can accomplish. We can choose whether they include the old functionality. And no subtyping relationships are necessarily involved. Moreover, the functor we wrote can be used to extend **any** set implementation with `of_list`, whereas class extension applies to just a **single** base class. There are ways of achieving something similar in Java with *mixins*, which were added in Java 1.5.

7.10 Summary

The OCaml module system provides mechanisms for modularity that provide the similar capabilities as mechanisms you will have seen in other languages. But seeing those mechanisms appear in different ways is hopefully helping you understand them better. OCaml abstract types and signatures, for example, provide a mechanism for abstraction that resembles Java visibility modifiers and interfaces. Seeing the same idea embodied in two different languages, but expressed in rather different ways, will hopefully help you recognize that idea when you encounter it in other languages in the future.

Moreover, the idea that a type could be abstract is a foundational notion in programming language design. The OCaml module system makes that idea brutally apparent. Other languages like Java obscure it a bit by coupling it together with many other features all at once. There's a sense in which every Java class implicitly defines an abstract type (actually, four abstract types that are related by subtyping, one for each visibility modifier [`public`, `protected`, `private`, and `default`]), and all the methods of the class are functions on that abstract type.

Functors are an advanced language feature in OCaml that might seem mysterious at first. If so, keep in mind: they're really just a kind of function that takes a structure as input and returns a structure as output. The reason they don't behave quite like normal OCaml functions is that structures are not first-class values in OCaml: you can't write regular functions that take a structure as input or return a structure as output. But functors can do just that.

Functors and includes enable code reuse. The kinds of code reuse that object-oriented features enable can also be achieved with functors and include. That's not to say that functors and includes are exactly equivalent to those object-oriented features: some kinds of code reuse might be easier to achieve with one set of features than the other.

One way to think about this might be that class extension is a very limited, but very useful, combination of functors and includes. Extending a class is like writing a functor that takes the base class as input, includes it, then adds new functions. But functors provide more general capability than class extension, because they can compute arbitrary functions of their input structure, rather than being limited to just certain kinds of extension.

Perhaps the most important idea to get out of studying the OCaml module system is an appreciation for the aspects of modularity that transcend any given language: namespaces, abstraction, and code reuse. Having seen those ideas in a couple very different languages, you're equipped to recognize them more clearly in the next language you learn.

7.10.1 Terms and Concepts

- abstract type
- abstraction
- client
- code reuse
- compilation unit
- declaration
- definition
- encapsulation
- ephemeral data structure
- functional data structure
- functor
- implementation
- implementer
- include

- information hiding
- interface
- local reasoning
- maintainability
- maps
- modular programming
- modularity
- module
- module type
- namespace
- open
- parameterized structure
- persistent data structure
- representation type
- scope
- sealed
- set representations
- sharing constraints
- signature
- signature matching
- specification
- structure

7.10.2 Further Reading

- *Introduction to Objective Caml*, chapters 11, 12, and 13
- *OCaml from the Very Beginning*, chapter 16
- *Real World OCaml*, chapters 4, 9, and 10
- *Purely Functional Data Structures*, chapters 1 and 2, by Chris Okasaki.
- "Design Considerations for ML-Style Module Systems" by Robert Harper and Benjamin C. Pierce, chapter 8 of *Advanced Topics in Types and Programming Languages*, ed. Benjamin C. Pierce, MIT Press, 2005. An advanced treatment of the static semantics of modules.

7.11 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Exercise: complex synonym [★]

Here is a module type for complex numbers, which have a real and imaginary component:

```
module type ComplexSig = sig
  val zero : float * float
  val add : float * float -> float * float -> float * float
end
```

Improve that code by adding `type t = float * float`. Show how the signature can be written more tersely because of the type synonym.

Exercise: complex encapsulation [★★]

Here is a module for the module type from the previous exercise:

```
module Complex : ComplexSig = struct
  type t = float * float
  let zero = (0., 0.)
  let add (r1, i1) (r2, i2) = r1 +. r2, i1 +. i2
end
```

Investigate what happens if you make the following changes (each independently), and explain why any errors arise:

- remove `zero` from the structure
- remove `add` from the signature
- change `zero` in the structure to `let zero = 0, 0`

Exercise: big list queue [★★]

Use the following code to create `ListQueue` of exponentially increasing length: 10, 100, 1000, etc. How big of a queue can you create before there is a noticeable delay? How big until there's a delay of at least 10 seconds? (Note: you can abort utop computations with Ctrl-C.)

```
(** Creates a ListQueue filled with [n] elements. *)
let fill_listqueue n =
  let rec loop n q =
    if n = 0 then q
    else loop (n - 1) (ListQueue.enqueue n q) in
  loop n ListQueue.empty
```

Exercise: big batched queue [★★]

Use the following function to create `BatchedQueue` of exponentially increasing length:

```
let fill_batchedqueue n =
  let rec loop n q =
    if n = 0 then q
    else loop (n - 1) (BatchedQueue.enqueue n q) in
  loop n BatchedQueue.empty
```

Now how big of a queue can you create before there's a delay of at least 10 seconds?

Exercise: queue efficiency [★★★]

Compare the implementations of `enqueue` in `ListQueue` vs. `BatchedQueue`. Explain in your own words why the efficiency of `ListQueue.enqueue` is linear time in the length of the queue. *Hint: consider the `@` operator.* Then explain why adding n elements to the queue takes time that is quadratic in n.

Now consider `BatchedQueue.enqueue`. Suppose that the queue is in a state where it has never had any elements dequeued. Explain in your own words why `BatchedQueue.enqueue` is constant time. Then explain why adding n elements to the queue takes time that is linear in n.

Exercise: binary search tree map [★★★★]

Write a module `BstMap` that implements the `Map` module type using the a binary search tree type. *Binary trees* were covered earlier when we discussed algebraic data types. A binary *search* tree (BST) is a binary tree that obeys the following *BST Invariant*:

> For any node *n*, every node in the left subtree of *n* has a value less than *n*'s value, and every node in the right subtree of *n* has a value greater than *n*'s value.

Your nodes should store pairs of keys and values. The keys should be ordered by the BST Invariant. Based on that invariant, you will always know whether to look left or right in a tree to find a particular key.

Exercise: fraction [★★★]

Write a module that implements the `Fraction` module type below:

```
module type Fraction = sig
  (* A fraction is a rational number p/q, where q != 0.*)
  type t

  (** [make n d] is n/d. Requires d != 0. *)
  val make : int -> int -> t

  val numerator : t -> int
  val denominator : t -> int
  val to_string : t -> string
  val to_float : t -> float

  val add : t -> t -> t
  val mul : t -> t -> t
end
```

Exercise: fraction reduced [★★★]

Modify your implementation of `Fraction` to ensure these invariants hold of every value `v` of type `t` that is returned from `make`, `add`, and `mul`:

1. `v` is in *reduced form*
2. the denominator of `v` is positive

For the first invariant, you might find this implementation of Euclid's algorithm to be helpful:

```
(** [gcd x y] is the greatest common divisor of [x] and [y].
    Requires: [x] and [y] are positive. *)
let rec gcd x y =
  if x = 0 then y
  else if (x < y) then gcd (y - x) x
  else gcd y (x - y)
```

Exercise: make char map [★]

To create a standard library map, we first have to use the `Map.Make` functor to produce a module that is specialized for the type of keys we want. Type the following in utop:

```
# module CharMap = Map.Make(Char);;
```

The output tells you that a new module named `CharMap` has been defined, and it gives you a signature for it. Find the values `empty`, `add`, and `remove` in that signature. Explain their types in your own words.

Exercise: char ordered [★]

The `Map.Make` functor requires its input module to match the `Map.OrderedType` signature. Look at that signature as well as the signature for the `Char` module. Explain in your own words why we are allowed to pass `Char` as an argument to `Map.Make`.

Exercise: use char map [★★]

Using the `CharMap` you just made, create a map that contains the following bindings:

- `'A'` maps to `"Alpha"`
- `'E'` maps to `"Echo"`
- `'S'` maps to `"Sierra"`
- `'V'` maps to `"Victor"`

Use `CharMap.find` to find the binding for `'E'`.

Now remove the binding for `'A'`. Use `CharMap.mem` to find whether `'A'` is still bound.

Use the function `CharMap.bindings` to convert your map into an association list.

Exercise: bindings [★★]

Investigate the documentation of the `Map.S` signature to find the specification of `bindings`. Which of these expressions will return the same association list?

1. `CharMap.(empty |> add 'x' 0 |> add 'y' 1 |> bindings)`
2. `CharMap.(empty |> add 'y' 1 |> add 'x' 0 |> bindings)`

3. `CharMap.(empty |> add 'x' 2 |> add 'y' 1 |> remove 'x' |> add 'x' 0 |> bindings)`

Check your answer in utop.

Exercise: date order [★★]

Here is a type for dates:

```
type date = {month : int; day : int}
```

For example, March 31st would be represented as `{month = 3; day = 31}`. Our goal in the next few exercises is to implement a map whose keys have type `date`.

Obviously it's possible to represent invalid dates with type `date`—for example, `{ month=6; day=50 }` would be June 50th, which is not a real date. The behavior of your code in the exercises below is unspecified for invalid dates.

To create a map over dates, we need a module that we can pass as input to `Map.Make`. That module will need to match the `Map.OrderedType` signature. Create such a module. Here is some code to get you started:

```
module Date = struct
  type t = date
  let compare ...
end
```

Recall the specification of `compare` in `Map.OrderedType` as you write your `Date.compare` function.

Exercise: calendar [★★]

Use the `Map.Make` functor with your `Date` module to create a `DateMap` module. Then define a `calendar` type as follows:

```
type calendar = string DateMap.t
```

The idea is that `calendar` maps a `date` to the name of an event occurring on that date.

Using the functions in the `DateMap` module, create a calendar with a few entries in it, such as birthdays or anniversaries.

Exercise: print calendar [★★]

Write a function `print_calendar : calendar -> unit` that prints each entry in a calendar in a format similar to the inspiring examples in the previous exercise. *Hint: use `DateMap.iter`, which is documented in the `Map.S` signature.*

Exercise: is for [★★★]

Write a function `is_for : string CharMap.t -> string CharMap.t` that given an input map with bindings from k_1 to v_1, …, k_n to v_n, produces an output map with the same keys, but where each key k_i is now bound to the string "k_i is for v_i". For example, if `m` maps `'a'` to `"apple"`, then `is_for m` would map `'a'` to `"a is for apple"`. *Hint: there is a one-line solution that uses a function from the `Map.S` signature. To convert a character to a string, you could use `String.make`. An even fancier way would be to use `Printf.sprintf`.*

Exercise: first after [★★★]

Write a function `first_after : calendar -> Date.t -> string` that returns the name of the first event that occurs strictly after the given date. If there is no such event, the function should raise `Not_found`, which is an exception already defined in the standard library. *Hint: there is a one-line solution that uses two functions from the `Map.S` signature.*

Exercise: sets [★★★]

The standard library `Set` module is quite similar to the `Map` module. Use it to create a module that represents sets of *case-insensitive strings*. Strings that differ only in their case should be considered equal by the set. For example, the sets {"grr", "argh"} and {"aRgh", "GRR"} should be considered the same, and adding "gRr" to either set should not change the set.

Exercise: ToString [★★]

Write a module type `ToString` that specifies a signature with an abstract type `t` and a function `to_string : t -> string`.

Exercise: Print [★★]

Write a functor `Print` that takes as input a module named `M` of type `ToString`. The module returned by your functor should have exactly one value in it, `print`, which is a function that takes a value of type `M.t` and prints a string representation of that value.

Exercise: Print Int [★★]

Create a module named `PrintInt` that is the result of applying the functor `Print` to a new module `Int`. You will need to write `Int` yourself. The type `Int.t` should be `int`. *Hint: do not seal `Int`.*

Experiment with `PrintInt` in utop. Use it to print the value of an integer.

Exercise: Print String [★★]

Create a module named `PrintString` that is the result of applying the functor `Print` to a new module `MyString`. You will need to write `MyString` yourself. *Hint: do not seal `MyString`.*

Experiment with `PrintString` in utop. Use it to print the value of a string.

Exercise: Print Reuse [★]

Explain in your own words how `Print` has achieved code reuse, albeit a very small amount.

Exercise: Print String reuse revisited [★★]

The `PrintString` module you created above supports just one operation: `print`. It would be great to have a module that supports all the `String` module functions in addition to that `print` operation, and it would be super great to derive such a module without having to copy any code.

Define a module `StringWithPrint`. It should have all the values of the built-in `String` module. It should also have the `print` operation, which should be derived from the `Print` functor rather than being copied code. *Hint: use two `include` statements.*

Exercise: implementation without interface [★]

Create a file named `date.ml`. In it put the following code:

```
type date = {month : int; day : int}
let make_date month day = {month; day}
let get_month d = d.month
let get_day d = d.day
let to_string d = (string_of_int d.month) ^ "/" ^ (string_of_int d.day)
```

Also create a dune file:

```
(library
 (name date))
```

Load the library into utop:

```
$ dune utop
```

In utop, open `Date`, create a date, access its day, and convert it to a string.

Exercise: implementation with interface [★]

After doing the previous exercise, also create a file named `date.mli`. In it put the following code:

```
type date = {month : int; day : int}
val make_date : int -> int -> date
val get_month : date -> int
val get_day : date -> int
val to_string : date -> string
```

Then re-do the same work as before in utop.

Exercise: implementation with abstracted interface [★]

After doing the previous two exercises, edit `date.mli` and change the first declaration in it to the following:

```
type date
```

The type `date` is now abstract. Again re-do the same work in utop. Some of the responses will change. Explain in your own words those changes.

Exercise: printer for dat [★★★]

Add a declaration to `date.mli`:

```
val format : Format.formatter -> date -> unit
```

And add a definition of `format` to `date.ml`. *Hint: use `Format.fprintf` and `Date.to_string`.*

Now recompile, load utop, and after loading `date.cmo` install the printer by issuing the directive

```
#install_printer Date.format;;
```

Reissue the other phrases to utop as you did in the exercises above. The response from one phrase will change in a helpful way. Explain why.

Exercise: refactor arith [★★★★]

Download this file: algebra.ml. It contains two signatures and four structures:

- `Ring` is signature that describes the algebraic structure called a *ring*, which is an abstraction of the addition and multiplication operators.
- `Field` is a signature that describes the algebraic structure called a *field*, which is like a ring but also has an abstraction of the division operation.
- `IntRing` and `FloatRing` are structures that implement rings in terms of `int` and `float`.
- `IntField` and `FloatField` are structures that implement fields in terms of `int` and `float`.
- `IntRational` and `FloatRational` are structures that implement fields in terms of ratios (aka fractions)—that is, pairs of `int` and pairs of `float`.

Note: Dear fans of abstract algebra: of course these representations don't necessarily obey all the axioms of rings and fields because of the limitations of machine arithmetic. Also, the division operation in `IntField` is ill-defined on zero. Try not to worry about that.

Refactor the code to improve the amount of code reuse it exhibits. To do that, use `include`, functors, and introduce additional structures and signatures as needed. There isn't necessarily a right answer here, but here's some advice:

- No name should be *directly declared* in more than one signature. For example, `( + )` should not be directly declared in `Field`; it should be reused from an earlier signature. By "directly declared" we mean a declaration of the form `val name : ...`. An indirect declaration would be one that results from an `include`.
- You need only three *direct definitions* of the algebraic operations and numbers (plus, minus, times, divide, zero, one): once for `int`, once for `float`, and once for ratios. For example, `IntField.( + )` should not be directly defined as `Stdlib.( + )`; rather, it should be reused from elsewhere. By "directly defined" we mean a definition of the form `let name = ...`. An indirect definition would be one that results from an `include` or a functor application.
- The rational structures can both be produced by a single functor that is applied once to `IntField` and once to `FloatField`.
- It's possible to eliminate all duplication of `of_int`, such that it is directly defined exactly once, and all structures reuse that definition; and such that it is directly declared in only one signature. This will require the use of functors. It will also require inventing an algorithm that can convert an integer to an arbitrary `Ring` representation, regardless of what the representation type of that `Ring` is.

When you're done, the types of all the modules should remain unchanged. You can easily see those types by running `ocamlc -i algebra.ml`.

Part IV

Correctness and Efficiency

CHAPTER

EIGHT

CORRECTNESS

When we write code, we always hope that we get it right. We *hope* that our code is correct. But how can we *know* it's correct? In this chapter, we'll study three possible answers: documentation, testing, and proof.

Let's be honest: we all at one time or another have thought that documentation or testing was a boring, tedious, and altogether postponable task. But with maturity programmers come to realize that both are essential to writing correct code. Both get at the *truth* of what code really does.

Documentation is the ground truth of what a programmer intended, as opposed to what they actually wrote. It communicates to other humans the ideas the author had in their head. No small amount of the time (even in this book!), we fail at communicating ideas as we intended. Maybe the failure occurs in the code, or maybe in the documentation. But writing documentation forces us to think a second (er, *hopefully* second) time about our intentions. The cognitive task of explaining our ideas to other humans is certainly different than explaining our ideas to the computer. That can expose failures in our thinking.

More importantly, documentation is a message in a time capsule. Imagine this: someone far away and now unreachable has sent that message to you, the programmer. You need that message to interpret the archeological evidence now in front of you—i.e., the otherwise unintelligible source code you have inherited. Your only hope is that the original author, long ago, had enough empathy to commit their thoughts to the written word.

And now imagine this: that author from the distant past? **What if they were YOU?** It might be you from two weeks ago, two months ago, or two years ago. Human memory is fleeting. If you've only been programming for a couple of years yourself, this can be difficult to understand, but give it a generous try: Someday, you're going to come back to the code you're writing today and have no clue what it means. Your only hope is to leave yourself some breadcrumbs at the time you write it. Otherwise, you'll be lost when you circle back.

Testing is the ground truth of what a program actually does, as opposed to what the programmer intended. It provides evidence that the programmer got it right. Good scientists demand evidence. That demand comes not out of arrogance but humility. We human beings are so amazingly good at deluding ourselves. (Consider the echo chamber of modern social media.) You can write a piece of code that you *think* is right. But then you can write a test case that *demonstrates* it's right. Then you can write ten more. The evidence accumulates, and eventually it's enough to be convincing. Is it absolute? Of course not. Maybe there's some test case you weren't clever enough to invent. That's science: new ideas come along to challenge the old.

Even more importantly, testing is *repeatable* science. The ability to replicate experiments is crucial to the truth they establish. By capturing tests as automatically repeatable experiments as unit test suites, we can demonstrate to ourselves and other, now and in the future, that our code is correct.

The challenge of documentation and testing is discipline. It's so tempting, so easy, to care only about writing the code. "That's the fun part", right? But it's like leaving out a third of the letter we intended to write. One part of the letter is to the machine, regarding how to compute. But another part is to other humans, about what we wanted to compute. And another part is to both machines and humans, about what we really did manage to compute. Your job isn't done until all three parts have been written.

If you're not yet convinced about the importance of documentation and testing, no worries. You will be in the future, if you stick with the craft of programming long enough. Meanwhile, let's proceed with learning about how to do it better.

In this chapter, we're going to learn about some successful (and hopefully new-to-you) techniques for both.

Finally, beyond documentation and testing, there is mathematical **proof** of correctness. Techniques from logic and discrete math can be used to formally prove that a program is correct according to a specification. Such proofs aren't necessarily easy—in fact they take even more human discipline and training than documentation and testing do. But they can make sense to apply when programs are used for safety critical tasks where human lives are on the line.

8.1 Specifications

A *specification* is a contract between a *client* of some unit of code and the *implementer* of that code. The most common place we find specifications is as comments in the interface (`.mli`) files for a module. There, the implementer of the module spells out what the client may and may not assume about the module's behavior. This contract makes it clear who to blame if something goes wrong: Did the client misuse the module? Or did the implementer fail to deliver the promised functionality?

Specifications usually involve preconditions and postconditions. The preconditions inform what the client must guarantee about inputs they pass in, and what the implementer may assume about those inputs. The postconditions inform what they client may assume about outputs they receive, and what the implementer must guarantee about those outputs.

An implementation *satisfies* a specification if it provides the behavior described by the specification. There may be many possible implementations of a given specification that are feasible. The client may not assume anything about which of those implementations is actually provided. The implementer, on the other hand, gets to provide one of their choice.

Clear specifications serve many important functions in software development teams. One important one is when something goes wrong, everyone can agree on whose job it is to fix the problem: either the implementer has not met the specification and needs to fix the implementation, or the client has written code that assumes something not guaranteed by the spec, and therefore needs to fix the using code. Or, perhaps the spec is wrong, and then the client and implementer need to decide on a new spec. This ability to decide whose problem a bug is prevents problems from slipping through the cracks.

Writing Specifications. Good specifications have to balance two conflicting goals; they must be

- sufficiently restrictive, ruling out implementations that would be useless to clients, as well as
- sufficiently general, not ruling out implementations that would be useful to clients.

Some common mistakes include not stating enough in preconditions, failing to identify when exceptions will be thrown, failing to specify behavior at boundary cases, writing operational specifications instead of definitional and stating too much in postconditions.

Writing good specifications is hard because the language and compiler do nothing to check the correctness of a specification: there's no type system for them, no warnings, etc. (Though there is ongoing research on how to improve specifications and the writing of them.) The specifications you write will be read by other people, and with that reading can come misunderstanding. Reading specifications requires close attention to detail.

Specifications should be written quite early. As soon as a design decision is made, document it in a specification. Specifications should continue to be updated throughout implementation. A specification becomes obsolete only when the code it specifies becomes obsolete and is removed from the code base.

Abstraction by Specification. Abstraction enables modular programming by hiding the details of implementations. Specifications are a part of that kind of abstraction: they reveal certain information about the behavior of a module without disclosing all the details of the module's implementation.

Locality is one of the benefits of abstraction by specification. A module can be understood without needing to examine its implementation. This locality is critical in implementing large programs, and even in in implementing smaller programs in teams. No one person can keep the entire system in their head at a time.

Modifiability is another benefit. Modules can be reimplemented without changing the implementation of other modules or functions. Software libraries depend upon this to improve their functionality without forcing all their clients to rewrite

code every time the library is upgraded. Modifiability also enables performance enhancements: we can write simple, slow implementations first, then improve bottlenecks as necessary.

The client should not assume more about the implementation than is given in the spec because that allows the implementation to change. The specification forms an *abstraction barrier* that protects the implementer from the client and vice versa. Making assumptions about the implementation that are not guaranteed by the specification is known as *violating the abstraction barrier*. The abstraction barrier enforces local reasoning. Further, it promotes *loose coupling* between different code modules. If one module changes, other modules are less likely to have to change to match.

8.2 Function Documentation

This section continues the discussion of documentation*, which we began in chapter 2.*

A specification is written for humans to read, not machines. "Specs" can take time to write well, and it is time well spent. The main goal is clarity. It is also important to be concise, because client programmers will not always take the effort to read a long spec. As with anything we write, we need to be aware of our audience when writing specifications. Some readers may need a more verbose specification than others.

A well-written specification usually has several parts communicating different kinds of information about the thing specified. If we know what the usual ingredients of a specification are, we are less likely to forget to write down something important. Let's now look at a recipe for writing specifications.

8.2.1 Returns Clause

How might we specify `sqr`, a square-root function? First, we need to describe its result. We will call this description the *returns clause* because it is a part of the specification that describes the result of a function call. It is also known as a *postcondition*: it describes a condition that holds after the function is called. Here is an example of a returns clause:

```
(** returns: [sqr x] is the square root of [x]. *)
```

But we would typically leave out the `returns:`, and simply write the returns clause as the first sentence of the comment:

```
(** [sqr x] is the square root of [x]. *)
```

For numerical programming, we should probably add some information about how accurate it is.

```
(** [sqr x] is the square root of [x]. Its relative accuracy is no worse than
    [1.0e-6]. *)
```

Similarly, here's how we might write a returns clause for a `find` function:

```
(** [find lst x] is the index of [x] in [lst], starting from zero. *)
```

A good specification is concise but clear—it should say enough that the reader understands what the function does, but without extra verbiage to plow through and possibly cause the reader to miss the point. Sometimes there is a balance to be struck between brevity and clarity.

These two specifications use a useful trick to make them more concise: they talk about the result of applying the function being specified to some arbitrary arguments. Implicitly we understand that the stated postcondition holds for all possible values of any unbound variables (the argument variables).

8.2.2 Requires Clause

The specification for `sqr` doesn't completely make sense because the square root does not exist for some `x` of type `real`. The mathematical square root function is a *partial* function that is defined over only part of its domain. A good function specification is complete with respect to the possible inputs; it provides the client with an understanding of what inputs are allowed and what the results will be for allowed inputs.

We have several ways to deal with partial functions. A straightforward approach is to restrict the domain so that it is clear the function cannot be legitimately used on some inputs. The specification rules out bad inputs with a *requires clause* establishing when the function may be called. This clause is also called a *precondition* because it describes a condition that must hold before the function is called. Here is a requires clause for `sqr`:

```
(** [sqr x] is the square root of [x]. Its relative accuracy is no worse
    than [1.0e-6].  Requires: [x >= 0] *)
```

This specification doesn't say what happens when `x < 0`, nor does it have to. Remember that the specification is a contract. This contract happens to push the burden of showing that the square root exists onto the client. If the requires clause is not satisfied, the implementation is permitted to do anything it likes: for example, go into an infinite loop or throw an exception. The advantage of this approach is that the implementer is free to design an algorithm without the constraint of having to check for invalid input parameters, which can be tedious and slow down the program. The disadvantage is that it may be difficult to debug if the function is called improperly, because the function can misbehave and the client has no understanding of how it might misbehave.

8.2.3 Raises Clause

Another way to deal with partial functions is to convert them into total functions (functions defined over their entire domain). This approach is arguably easier for the client to deal with because the function's behavior is always defined; it has no precondition. However, it pushes work onto the implementer and may lead to a slower implementation.

How can we convert `sqr` into a total function? One approach that is (too) often followed is to define some value that is returned in the cases that the requires clause would have ruled; for example:

```
(** [sqr x] is the square root of [x] if [x >= 0],
    with relative accuracy no worse than 1.0e-6.
    Otherwise, a negative number is returned. *)
```

This practice is not recommended because it tends to encourage broken, hard-to-read client code. Almost any correct client of this abstraction will write code like this if the precondition cannot be argued to hold:

```
if sqr(a) < 0.0 then ... else ...
```

The error must still be handled in the `if` expression, so the job of the client of this abstraction isn't any easier than with a requires clause: the client still needs to wrap an explicit test around the call in cases where it might fail. If the test is omitted, the compiler won't complain, and the negative number result will be silently treated as if it were a valid square root, likely causing errors later during program execution. This coding style has been the source of innumerable bugs and security problems in the Unix operating systems and its descendents (e.g., Linux).

A better way to make functions total is to have them raise an exception when the expected input condition is not met. Exceptions avoid the necessity of distracting error-handling logic in the client's code. If the function is to be total, the specification must say what exception is raised and when. For example, we might make our square root function total as follows:

```
(** [sqr x] is the square root of [x], with relative accuracy no worse
    than 1.0e-6. Raises: [Negative] if [x < 0]. *)
```

Note that the implementation of this `sqr` function must check whether `x >= 0`, even in the production version of the code, because some client may be relying on the exception to be raised.

8.2.4 Examples Clause

It can be useful to provide an illustrative example as part of a specification. No matter how clear and well written the specification is, an example is often useful to clients.

```
(** [find lst x] is the index of [x] in [lst], starting from zero.
    Example: [find ["b","a","c"] "a" = 1]. *)
```

8.2.5 The Specification Game

When evaluating specifications, it can be useful to imagine that a game is being played between two people: a *specifier* and a *devious programmer*.

Suppose that the specifier writes the following specification:

```
(** returns a list *)
val reverse : 'a list -> 'a list
```

This spec is clearly incomplete. For example, a devious programmer could meet the spec with an implementation that gives the following output:

```
# reverse [1; 2; 3];;
- : int list = []
```

The specifier, upon realizing this, refines the spec as follows:

```
(** [reverse lst] returns a list that is the same length as [lst] *)
val reverse : 'a list -> 'a list
```

But the devious specifier discovers that the spec still allows broken implementations:

```
# reverse [1; 2; 3];;
- : int list = [0; 0; 0]
```

Finally, the specifier settles on a third spec:

```
(** [reverse lst] returns a list [m] satisfying the following conditions:
    - [length lst = length m]
    - for all [i], [nth m i = nth lst (n - i - 1)],
      where [n] is the length of [lst].
    For example, [reverse [1; 2; 3]] is [3; 2; 1], and [reverse []] is []. *)
val reverse : 'a list -> 'a list
```

With this spec, the devious programmer is forced to provide a working implementation to meet the spec, so the specifier has successfully written her spec.

The point of playing this game is to improve your ability to write specifications. Obviously we're not advocating that you deliberately try to violate the intent of a specification and get away with it. When reading someone else's specification, read as generously as possible. But be ruthless about improving your own specifications.

8.2.6 Comments

In addition to specifying functions, programmers need to provide comments in the body of the functions. In fact, programmers usually do not write enough comments in their code. (For a classic example, check out the actual comment on line 561 of the Quake 3 Arena game engine.)

But this doesn't mean that adding more comments is always better. The wrong comments will simply obscure the code further. Shoveling as many comments into code as possible usually makes the code worse! Both code and comments are precise tools for communication (with the computer and with other programmers) that should be wielded carefully.

It is particularly annoying to read code that contains many interspersed comments (typically of questionable value), e.g.:

```
let y = x + 1 (* make y one greater than x *)
```

For complex algorithms, some comments may be necessary to explain how the code implementing the algorithm works. Programmers are often tempted to write comments about the algorithm interspersed through the code. But someone reading the code will often find these comments confusing because they don't have a high-level picture of the algorithm. It is usually better to write a paragraph-style comment at the beginning of the function explaining how its implementation works. Explicit points in the code that need to be related to that paragraph can then be marked with very brief comments, like `(* case 1 *)`.

Another common but well-intentioned mistake is giving variables long, descriptive names, as in the following verbose code:

```
let number_of_zeros the_list =
  List.fold_left (fun (accumulator : int) (list_element : int) ->
    accumulator + (if list_element = 0 then 1 else 0)) 0 the_list
```

Code using such long names is verbose and hard to read. Instead of trying to embed a complete description of a variable in its name, use a short and suggestive name (e.g., `zeros`), and if necessary, add a comment at its declaration explaining the purpose of the variable.

```
let zeros lst =
  let is0 = function 0 -> 1 | _ -> 0 in
  List.fold_left (fun zs x -> zs + is0 x) 0 lst
```

A similarly bad practice is to encode the type of the variable in its name, e.g. naming a variable `i_count` to show that it's an integer. The type system is going to guarantee that for you, and your editor can provide a hover-over to show the type. If you really want to emphasize the type in the code, add a type annotation at the point where the variable comes into scope.

8.3 Module Documentation

The specification of functions provided by a module can be found in its interface, which is what clients will consult. But what about *internal* documentation, which is relevant to those who implement and maintain a module? The purpose of such implementation comments is to explain to the reader how the implementation correctly implements its interface.

Reminder

It is inappropriate to copy the specifications of functions found in the module interface into the module implementation. Copying runs the risk of introducing inconsistency as the program evolves, because programmers don't keep the copies in sync. Copying code and specifications is a major source (if not *the* major source) of program bugs. In any case, implementers can always look at the interface for the specification.

Implementation comments fall into two categories. The first category arises because a module implementation may define new types and functions that are purely internal to the module. If their significance is not obvious, these types and functions should be documented in much the same style that we have suggested for documenting interfaces. Often, as the code is written, it becomes apparent that the new types and functions defined in the module form an internal data abstraction or at least a collection of functionality that makes sense as a module in its own right. This is a signal that the internal data abstraction might be moved to a separate module and manipulated only through its operations.

The second category of implementation comments is associated with the use of *data abstraction*. Suppose we are implementing an abstraction for a set of items of type `'a`. The interface might look something like this:

```
(** A set is an unordered collection in which multiplicity is ignored. *)
module type Set = sig

  (** ['a t] represents a set whose elements are of type ['a] *)
  type 'a t

  (** [empty] is the set containing no elements *)
  val empty : 'a t

  (** [mem x s] is whether [x] is a member of set [s] *)
  val mem : 'a -> 'a t -> bool

  (** [add x s] is the set containing all the elements of [s]
      as well as [x]. *)
  val add : 'a -> 'a t -> 'a t

  (** [rem x s] is the set containing all the elements of [s],
      minus [x]. *)
  val rem : 'a -> 'a t -> 'a t

  (** [size s] is the cardinality of [s] *)
  val size: 'a t -> int

  (** [union s1 s2] is the set containing all the elements that
      are in either [s1] or [s2]. *)
  val union: 'a t -> 'a t -> 'a t

  (** [inter s1 s2] is the set containing all the elements that
      are in both [s1] and [s2]. *)
  val inter: 'a t -> 'a t -> 'a t
end
```

In a real signature for sets, we'd want operations such as `map` and `fold` as well, but let's omit these for now for simplicity. There are many ways to implement this abstraction.

As we've seen before, one easy way is as a list:

```
(** Implementation of sets as lists with duplicates *)
module ListSet : Set = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add = List.cons
  let rem x = List.filter (( <> ) x)
  let size lst = List.(lst |> sort_uniq Stdlib.compare |> length)
  let union lst1 lst2 = lst1 @ lst2
  let inter lst1 lst2 = List.filter (fun h -> mem h lst2) lst1
end
```

This implementation has the advantage of simplicity. For small sets that tend not to have duplicate elements, it will be a fine choice. Its performance will be poor for large sets or applications with many duplicates but for some applications that's not an issue.

Notice that the types of the functions do not need to be written down in the implementation. They aren't needed because they're already present in the signature, just like the specifications that are also in the signature don't need to be replicated in the structure.

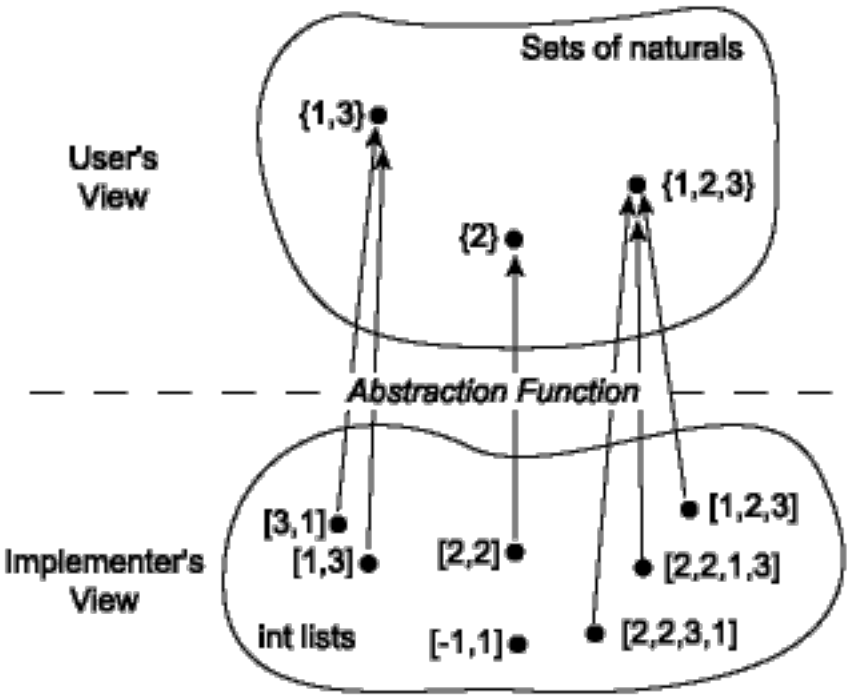
Here is another implementation of `Set` that also uses `'a list` but requires the lists to contain no duplicates. This implementation is also correct (and also slow for large sets). Notice that we are using the same representation type, yet some important aspects of the implementation (`add`, `size`, `union`) are quite different.

```
(** Implementation of sets as lists without duplicates *)
module UniqListSet : Set = struct
  type 'a t = 'a list
  let empty = []
  let mem = List.mem
  let add x lst = if mem x lst then lst else x :: lst
  let rem x = List.filter (( <> ) x)
  let size = List.length
  let union lst1 lst2 = lst1 @ lst2 |> List.sort_uniq Stdlib.compare
  let inter lst1 lst2 = List.filter (fun h -> mem h lst2) lst1
end
```

An important reason why we introduced the writing of function specifications was to enable *local reasoning*: once a function has a spec, we can judge whether the function does what it is supposed to without looking at the rest of the program. We can also judge whether the rest of the program works without looking at the code of the function. However, we cannot reason locally about the individual functions in the three module implementations just given. The problem is that we don't have enough information about the relationship between the concrete type (`int list`) and the corresponding abstract type (`set`). This lack of information can be addressed by adding two new kinds of comments to the implementation: the *abstraction function* and the *representation invariant* for the abstract data type. We turn to discussion of those, next.

8.3.1 Abstraction Functions

The client of any `Set` implementation should not be able to distinguish it from any other implementation based on its functional behavior. As far as the client can tell, the operations act like operations on the mathematical ideal of a set. In the first implementation, the lists `[3; 1]`, `[1; 3]`, and `[1; 1; 3]` are distinguishable to the implementer, but not to the client. To the client, they all represent the abstract set {1, 3} and cannot be distinguished by any of the operations of the `Set` signature. From the point of view of the client, the abstract data type describes a set of abstract values and associated operations. The implementer knows that these abstract values are represented by concrete values that may contain additional information invisible from the client's view. This loss of information is described by the *abstraction function*, which is a mapping from the space of concrete values to the abstract space. The abstraction function for the implementation `ListSet` looks like this:



Notice that several concrete values may map to a single abstract value; that is, the abstraction function may be *many-to-one*. It is also possible that some concrete values do not map to any abstract value; the abstraction function may be *partial*. That is not the case with `ListSet`, but it might be with other implementations.

The abstraction function is important for deciding whether an implementation is correct, therefore it belongs as a comment in the implementation of any abstract data type. For example, in the `ListSet` module, we could document the abstraction function as follows:

```
module ListSet : Set = struct
  (** Abstraction function: The list [[a1; ...; an]] represents the
      set [{b1, ..., bm}], where [[b1; ...; bm]] is the same list as
      [[a1; ...; an]] but with any duplicates removed. The empty list
      [[]] represents the empty set [{}]. *)
  type 'a t = 'a list
  ...
end
```

This comment explicitly points out that the list may contain duplicates, which is helpful as a reinforcement of the first sentence. Similarly, the case of an empty list is mentioned explicitly for clarity, although some might consider it to be redundant.

The abstraction function for the second implementation, which does not allow duplicates, hints at an important difference. We can write the abstraction function for this second representation a bit more simply because we know that the elements are distinct.

```
module UniqListSet : Set = struct
  (** Abstraction function: The list [[a1; ...; an]] represents the set
      [{a1, ..., an}]. The empty list [[]] represents the empty set [{}]. *)
  type 'a t = 'a list
  ...
end
```

8.3.2 Implementing the Abstraction Function

What would it mean to implement the abstraction function for `ListSet`? We'd want a function that took an input of type `'a ListSet.t`. But what should its output type be? The abstract values are mathematical sets, not OCaml types. If we did hypothetically have a type `'a set` that our abstraction function could return, there would have been little point in developing `ListSet`; we could have just used that `'a set` type without doing any work of our own.

On the other hand, we might implement something close to the abstraction function by converting an input of type `'a ListSet.t` to a built-in OCaml type or standard library type:

- We could convert to a `string`. That would have the advantage of being easily readable by humans in the toplevel or in debug output. Java programmers use `toString()` for similar purposes.
- We could convert to `'a list`. (Actually there's little conversion to be done). For data collections this is a convenient choice, since lists can at least approximately represent many data structures: stacks, queues, dictionaries, sets, heaps, etc.

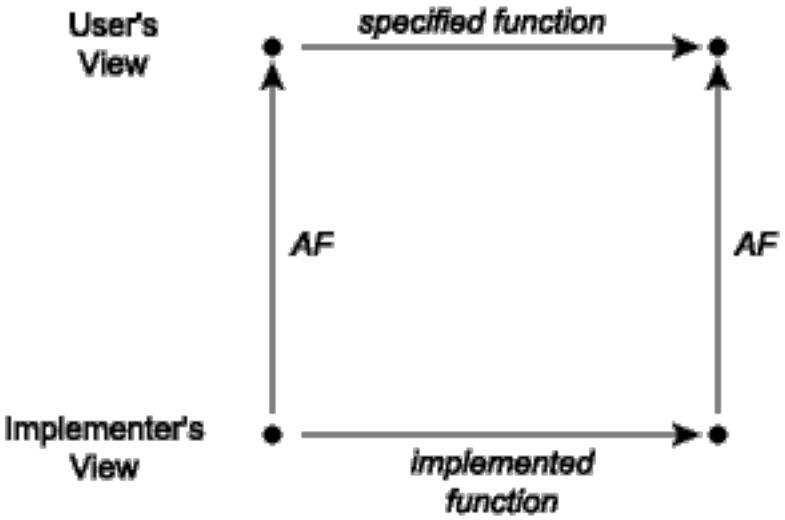
The following functions implement those ideas. Note that `to_string` has to take an additional argument `string_of_val` from the client to convert `'a` to `string`.

```
module ListSet : Set = struct
  ...

  let uniq lst = List.sort_uniq Stdlib.compare lst

  let to_string string_of_val lst =
    let interior =
      lst |> uniq |> List.map string_of_val |> String.concat ", "
    in
    "{" ^ interior ^ "}"

  let to_list = uniq
end
```

Installing a custom formatter, as discussed in the *section on encapsulation*, could also be understood as implementing the abstraction function. But in that case it's usable only by humans at the toplevel rather than other code, programatically.

8.3.3 Commutative Diagrams

Using the abstraction function, we can now talk about what it means for an implementation of an abstraction to be *correct*. It is correct exactly when every operation that takes place in the concrete space makes sense when mapped by the abstraction function into the abstract space. This can be visualized as a *commutative diagram*:



A commutative diagram means that if we take the two paths around the diagram, we have to get to the same place. Suppose that we start from a concrete value and apply the actual implementation of some operation to it to obtain a new concrete value or values. When viewed abstractly, a concrete result should be an abstract value that is a possible result of applying the function *as described in its specification* to the abstract view of the actual inputs. For example, consider the union function from the implementation of sets as lists with repeated elements covered last time. When this function is applied to the concrete pair [1; 3], [2; 2], it corresponds to the lower-left corner of the diagram. The result of this operation is the list [2; 2; 1; 3], whose corresponding abstract value is the list {1, 2, 3}. Note that if we apply the abstraction function AF to the input lists [1; 3] and [2; 2], we have the sets {1, 3} and {2}. The commutative diagram requires that in this instance the union of {1, 3} and {2} is {1, 2, 3}, which is of course true.

8.3.4 Representation Invariants

The abstraction function explains how information within the module is viewed abstractly by module clients. But that is not all we need to know to ensure correctness of the implementation. Consider the `size` function in each of the two implementations. For `ListSet`, which allows duplicates, we need to be sure not to double-count duplicate elements:

```
let size lst = List.(lst |> sort_uniq Stdlib.compare |> length)
```

But for `UniqListSet`, in which the lists have no duplicates, the size is just the length of the list:

```
let size = List.length
```

How do wo know that latter implementation is correct? That is, how do we know that "lists have no duplicates"? It's hinted at by the name of the module, and it can be deduced from the implementation of `add`, but we've never carefully documented it. Right now, the code does not explicitly say that there are no duplicates.

In the `UniqListSet` representation, not all concrete data items represent abstract data items. That is, the *domain* of the abstraction function does not include all possible lists. There are some lists, such as `[1; 1; 2]`, that contain duplicates and must never occur in the representation of a set in the `UniqListSet` implementation; the abstraction function is undefined on such lists. We need to include a second piece of information, the *representation invariant* (or *rep invariant*, or *RI*), to determine which concrete data items are valid representations of abstract data items. For sets represented as lists without duplicates, we write this as part of the comment together with the abstraction function:

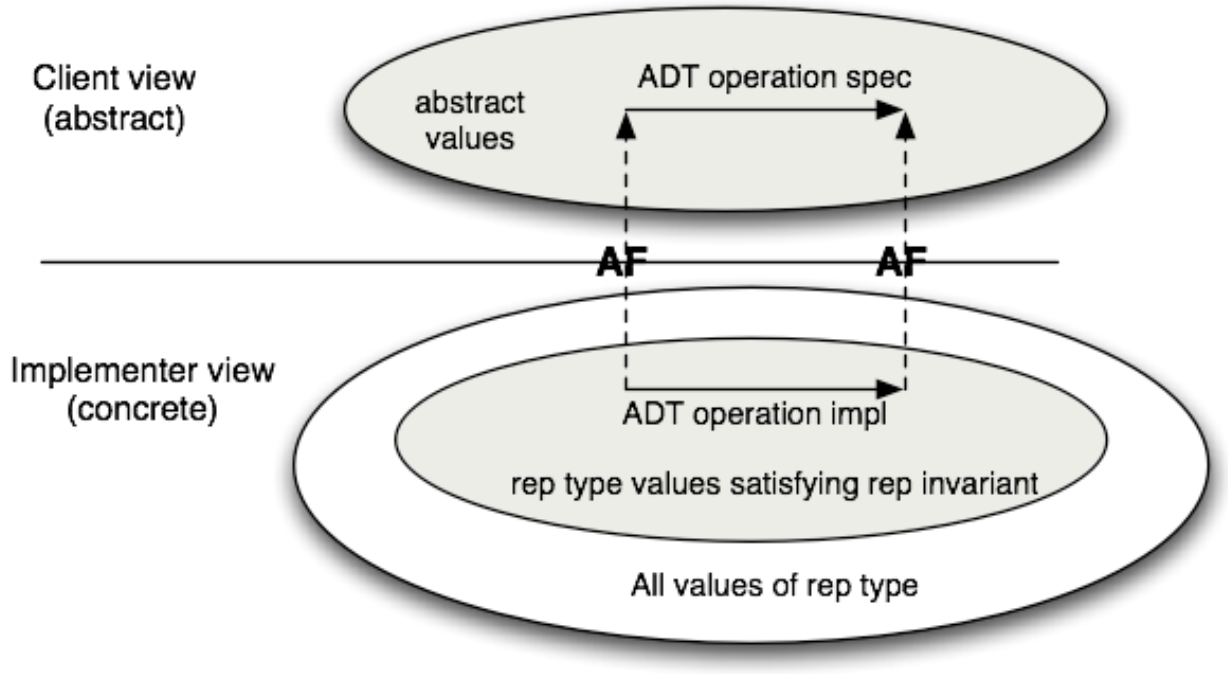
```
module UniqListSet : Set = struct
  (** Abstraction function: the list [[a1; ...; an]] represents the set
      [{a1, ..., an}].  The empty list [[]] represents the empty set [{}].
```

(continues on next page)

(continued from previous page)

```
      Representation invariant: the list contains no duplicates. *)
  type 'a t = 'a list
  ...
end
```

If we think about this issue in terms of the commutative diagram, we see that there is a crucial property that is necessary to ensure correctness: namely, that all concrete operations preserve the representation invariant. If this constraint is broken, functions such as `size` will not return the correct answer. The relationship between the representation invariant and the abstraction function is depicted in this figure:



We can use the rep invariant and abstraction function to judge whether the implementation of a single operation is correct *in isolation from the rest of the functions in the module*. A function is correct if these conditions:

1. The function's preconditions hold of the argument values.
2. The concrete representations of the arguments satisfy the rep invariant.

imply these conditions:

1. All new representation values created satisfy the rep invariant.
2. The commutative diagram holds.

The rep invariant makes it easier to write code that is provably correct, because it means that we don't have to write code that works for all possible incoming concrete representations—only those that satisfy the rep invariant. For example, in the implementation `UniqListSet`, we do not care what the code does on lists that contain duplicate elements. However, we do need to be concerned that on return, we only produce values that satisfy the rep invariant. As suggested in the figure above, if the rep invariant holds for the input values, then it should hold for the output values, which is why we call it an *invariant*.

8.3.5 Implementing the Representation Invariant

When implementing a complex abstract data type, it is often helpful to write an internal function that can be used to check that the rep invariant holds of a given data item. By convention we will call this function `rep_ok`. If the module accepts values of the abstract type that are created outside the module, say by exposing the implementation of the type in the signature, then `rep_ok` should be applied to these to ensure the representation invariant is satisfied. In addition, if the implementation creates any new values of the abstract type, `rep_ok` can be applied to them as a sanity check. With this approach, bugs are caught early, and a bug in one function is less likely to create the appearance of a bug in another.

A convenient way to write `rep_ok` is to make it an identity function that just returns the input value if the rep invariant holds and raises an exception if it fails.

```
(* Checks whether x satisfies the representation invariant. *)
let rep_ok x =
  if (* check the RI holds of x *) then x else failwith "RI violated"
```

Here is an implementation of `Set` that uses the same data representation as `UniqListSet`, but includes copious `rep_ok` checks. Note that `rep_ok` is applied to all input sets and to any set that is ever created. This ensures that if a bad set representation is created, it will be detected immediately. In case we somehow missed a check on creation, we also apply `rep_ok` to incoming set arguments. If there is a bug, these checks will help us quickly figure out where the rep invariant is being broken.

```
(** Implementation of sets as lists without duplicates. *)
module UniqListSet : Set = struct

  (** Abstraction function: The list [[a1; ...; an]] represents the
      set {a1, ..., an}. The empty list [[]] represents the empty set [{}].
      Representation invariant: the list contains no duplicates. *)
  type 'a t = 'a list

  let rep_ok lst =
    let u = List.sort_uniq Stdlib.compare lst in
    match List.compare_lengths lst u with 0 -> lst | _ -> failwith "RI"

  let empty = []

  let mem x lst = List.mem x (rep_ok lst)

  let add x lst = rep_ok (if mem x (rep_ok lst) then lst else x :: lst)

  let rem x lst = rep_ok (List.filter (( <> ) x) (rep_ok lst))

  let size lst = List.length (rep_ok lst)

  let union lst1 lst2 =
    rep_ok
      (List.fold_left
         (fun u x -> if mem x lst2 then u else x :: u)
         (rep_ok lst2) (rep_ok lst1))

  let inter lst1 lst2 = rep_ok (List.filter (fun h -> mem h lst2) (rep_ok lst1))
end
```

Calling `rep_ok` on every argument can be too expensive for the production version of a program. The `rep_ok` above, for example, requires linearithmic time, which destroys the efficiency of all the previously constant time or linear time operations. For production code, it may be more appropriate to use a version of `rep_ok` that only checks the parts of the rep invariant that are cheap to check. When there is a requirement that there be no run-time cost, `rep_ok` can be

changed to an identity function (or macro) so the compiler optimizes away the calls to it. However, it is a good idea to keep around the full code of `rep_ok` so it can be easily reinstated during future debugging:

```
let rep_ok lst = lst

let rep_ok_expensive =
  let u = List.sort_uniq Stdlib.compare lst in
  match List.compare_lengths lst u with 0 -> lst | _ -> failwith "RI"
```

Some languages provide support for *conditional compilation*, which provides some kind of support for compiling some parts of the codebase but not others. The OCaml compiler supports a flag `noassert` that disables assertion checking. So you could implement rep invariant checking with `assert`, and turn it off with `noassert`. The problem with that is that some portions of your codebase might *require* assertion checking to be turned on to work correctly.

8.4 Testing and Debugging

Correct programs behave as we intend them to behave. *Validation* is the process of building our confidence in correct program behavior.

8.4.1 Validation

There are many ways to increase that confidence. Social methods, formal methods, and testing are three. The latter is our main focus, but let's first consider the other two.

Social methods involve developing programs with other people, relying on their assistance to improve correctness. Some good techniques include the following:

- *Code walkthrough.* In the walkthrough approach, the programmer presents the documentation and code to a reviewing team, and the team gives comments. This is an informal process. The focus is on the code rather than the coder, so hurt feelings are easier to avoid. However, the team may not get as much assurance that the code is correct.
- *Code inspection.* Here, the review team drives the code review process. Some, though not necessarily very much, team preparation beforehand is useful. They define goals for the review process and interact with the coder(s) to understand where there may be quality problems. Again, making the process as blameless as possible is important.
- *Pair programming.* The most informal approach to code review is through pair programming, in which code is developed by a pair of engineers: the driver who writes the code, and the observer who watches. The role of the observer is be a critic, to think about potential errors, and to help navigate larger design issues. It's usually better to have the observer be the engineer with the greater experience with the coding task at hand. The observer reviews the code, serving as the devil's advocate that the driver must convince. When the pair is developing specifications, the observer thinks about how to make specs clearer or shorter. Pair programming has other benefits. It is often more fun and educational to work with a partner, and it helps focus both partners on the task. If you are just starting to work with another programmer, pair programming is a good way to understand how your partner thinks and to establish common vocabulary. It is a good idea for partners to trade off roles, too.

These social techniques for *code review* can be remarkably effective. In one study conducted at IBM (Jones, 1991), code inspection found 65% of the known coding errors and 25% of the known documentation errors, whereas testing found only 20% of the coding errors and none of the documentation errors. The code inspection process may be more effective than walkthroughs. One study (Fagan, 1976) found that code inspections resulted in code with 38% fewer failures, compared to code walkthroughs.

Thorough code review can be expensive, however. Jones found that preparing for code inspection took one hour per 150 lines of code, and the actual inspection covered 75 lines of code per hour. Having up to three people on the inspection

team improves the quality of inspection; beyond that, more inspectors doesn't seem to help. Spending a lot of time preparing for inspection did not seem to be useful, either. Perhaps this is because much of the value of inspection lies in the interaction with the coders.

Formal methods use the power of mathematics and logic to validate program behavior. *Verification* uses the program code and its specifications to construct a proof that the program behaves correctly on all possible inputs. There are research tools available to help with program verification, often based on automated theorem provers, as well as research languages that are designed for program verification. Verification tends to be expensive and to require thinking carefully about and deeply understanding the code to be verified. So in practice, it tends to be applied to code that is important and relatively short. Verification is particularly valuable for critical systems where testing is less effective. Because their execution is not deterministic, concurrent programs are hard to test, and sometimes subtle bugs can only be found by attempting to verify the code formally. In fact, tools to help prove programs correct have been getting increasingly effective and some large systems have been fully verified, including compilers, processors and processor emulators, and key pieces of operating systems.

Testing involves actually executing the program on sample inputs to see whether the behavior is as expected. By comparing the actual results of the program with the expected results, we find out whether the program really works on the particular inputs we try it on. Testing can never provide the absolute guarantees that formal methods do, but it is significantly easier and cheaper to do. It is also the validation methodology with which you are probably most familiar. Testing is a good, cost-effective way of building confidence in correct program behavior.

8.4.2 Debugging

When testing reveals an error, we usually say that the program is "buggy". But the word "bug" suggests something that wandered into a program. Better terminology would be that there are

- *faults*, which are the result of human errors in software systems, and
- *failures*, which are violations of requirements.

Some faults might never appear to an end user of a system, but failures are those faults that do. A fault might result because an implementation doesn't match design, or a design doesn't match the requirements.

Debugging is the process of discovering and fixing faults. Testing clearly is the "discovery" part, but fixing can be more complicated. Debugging can be a task that takes even more time than an original implementation itself! So you would do well to make it easy to debug your programs from the start. Write good specifications for each function. Document the AF and RI for each data abstraction. Keep modules small, and test them independently.

Inevitably, though, you will discover faults in your programs. When you do, approach them as a scientist by employing the *scientific method:*

- evaluate the data that are available;
- formulate a hypothesis that might explain the data;
- design a repeatable experiment to test that hypothesis; and
- use the result of that experiment to refine or refute your hypothesis.

Often the crux of this process is finding the simplest, smallest input that triggers a fault. That's not usually the original input for which we discover a fault. So some initial experimentation might be needed to find a *minimal test case*.

Never be afraid to write additional code, even a lot of additional code, to help you find faults. Functions like `to_string` or `format` can be invaluable in understanding computations, so writing them up front before any faults are detected is completely worthwhile.

When you do discover the source of a fault, be extra careful in fixing it. It is tempting to slap a quick fix into the code and move on. This is quite dangerous. Far too often, fixing a fault just introduces a new (and unknown) fault! If a bug is difficult to find, it is often because the program logic is complex and hard to reason about. Think carefully about why the fault could have been introduced in the first place, and about how you might prevent similar faults in the future.

8.5 Black-box and Glass-box Testing

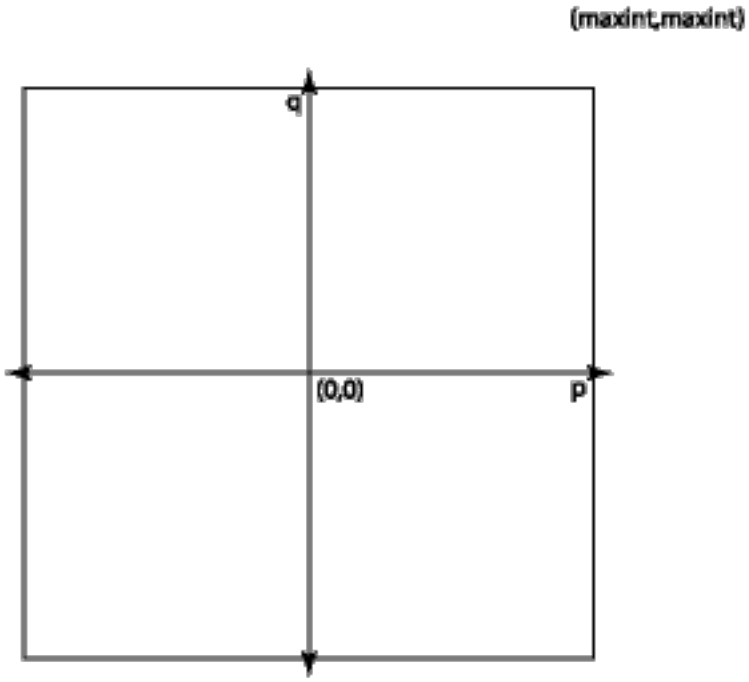
We would like to know that a program works on all possible inputs. The problem with testing is that it is usually infeasible to try all the possible inputs. For example, suppose that we are implementing a module that provides an abstract data type for rational numbers. One of its operations might be an addition function `plus`, e.g.:

```
module type RATIONAL = sig
  (** A [t] is a rational. *)
  type t

  (** [create p q] is the rational number [p/q].
      Raises: [Invalid_argument "0"] if [q] is 0. *)
  val create : int -> int -> t

  (** [plus r1 r2] is [r1 + r2] *)
  val plus : t -> t -> t
end

module Rational : RATIONAL = struct
  (** AF: [(p, q)] represents the rational number [p/q]
      RI: [q] is not 0. *)
  type t = int * int

  let create p q =
    if q = 0 then invalid_arg "0" else (p, q)

  let plus (p1, q1) (p2, q2) =
    (p1 * q2 + p2 * q1, q1 * q2)
end
```

What would it take to exhaustively test just this one function? We'd want to try all possible rationals as both the `r1` and `r2` arguments. A rational is formed from two ints, and there are 2^{63} ints on a modern OCaml implementation. Therefore there are approximately $(2^{63})^4 = 2^{252}$ possible inputs to the `plus` function. Even if we test one addition every nanosecond, it will take about 10^{59} years to finish testing this one function.

Clearly we can't test software exhaustively. But that doesn't mean we should give up on testing. It just means that we need to think carefully about what our test cases should be so that they are as effective as possible at convincing us that the code works.

Consider our `create` function, above. It takes in two integers `p` and `q` as arguments. How should we go about selecting a relatively small number of test cases that will convince us that the function works correctly on all possible inputs? We can visualize the space of all possible inputs as a large square:



There are about 2^{126} points in this square, so we can't afford to test them all. And testing them all is going to mostly be a waste of time—most of the possible inputs provide nothing new. We need a way to find a set of points in this space to test that are interesting and will give a good sense of the behavior of the program across the whole space.

Input spaces generally comprise a number of subsets in which the behavior of the code is similar in some essential fashion across the entire subset. We don't get any additional information by testing more than one input from each such subset.

If we test all the interesting regions of the input space, we have achieved good *coverage*. We want tests that in some useful sense cover the space of possible program inputs.

Two good ways of achieving coverage are *black-box testing* and *glass-box testing*. We discuss those, next.

8.5.1 Black-box Testing

In selecting our test cases for good coverage, we might want to consider both the specification and the implementation of the program or module being tested. It turns out that we can often do a pretty good job of picking test cases by just looking at the specification and ignoring the implementation. This is known as **black-box testing**. The idea is that we think of the code as a black box about which all we can see is its surface: its specification. We pick test cases by looking at how the specification implicitly introduces boundaries that divide the space of possible inputs into different regions.

When writing black-box test cases, we ask ourselves what set of test cases that will produce distinctive behavior as predicted by the specification. It is important to try out both *typical inputs* and inputs that are *boundary cases* aka *corner cases* or *edge cases*. A common error is to only test typical inputs, with the result that the program usually works but fails in less frequent situations. It's also important to identify ways in which the specification creates classes of inputs that should elicit similar behavior from the function, and to test on those *paths through the specification*. Here are some examples.

Example 1.

Here are some ideas for how to test the `create` function:

- Looking at the square above, we see that it has boundaries at `min_int` and `max_int`. We want to try to construct rationals at the corners and along the sides of the square, e.g. `create min_int min_int`, `create`

`max_int 2`, etc.

- The line `p=0` is important because `p/q` is zero all along it. We should try `(0, q)` for various values of `q`.
- We should try some typical `(p, q)` pairs in all four quadrants of the space.
- We should try both `(p, q)` pairs in which `q` divides evenly into `p`, and pairs in which `q` does not divide into `p`.
- Pairs of the form `(1, q)`, `(-1, q)`, `(p, 1)`, `(p, -1)` for various `p` and `q` also may be interesting given the properties of rational numbers.

The specification also says that the code will check that `q` is not zero. We should construct some test cases to ensure this checking is done as advertised. Trying `(1, 0)`, `(max_int, 0)`, `(min_int, 0)`, `(-1, 0)`, `(0, 0)` to see that they all raise the specified exception would probably be an adequate set of black-box tests.

Example 2.

Consider a function `list_max`:

```
(** Return the maximum element in the list. *)
val list_max: int list -> int
```

What is a good set of black-box test cases? Here the input space is the set of all possible lists of ints. We need to try some typical inputs and also consider boundary cases. Based on this spec, boundary cases include the following:

- A list containing one element. In fact, an empty list is probably the first boundary case we think of. Looking at the spec above, we realize that it doesn't specify what happens in the case of an empty list. Thus, thinking about boundary cases is also useful in identifying errors in the specification.
- A list containing two elements.
- A list in which the maximum is the first element. Or the last element. Or somewhere in the middle of the list.
- A list in which every element is equal.
- A list in which the elements are arranged in ascending sorted order, and one in which they are arranged in descending sorted order.
- A list in which the maximum element is `max_int`, and a list in which the maximum element is `min_int`.

Example 3.

Consider the function `sqrt`:

```
(** [sqrt x n] is the square root of [x] computed to an accuracy of [n]
    significant digits.
    Requires: [x >= 0] and [n >= 1]. *)
val sqrt : float -> int -> float
```

The precondition identifies two possibilities for `x` (either it is 0 or greater) and two possibilities for `n` (either it is 1 or greater). That leads to four "paths through the specification", i.e., representative and boundary cases for satisfying the precondition, which we should test:

- `x` is 0 and `n` is 1
- `x` is greater than 0 and `n` is 1
- `x` is 0 and `n` is greater than 1
- `x` is greater than 0 and `n` is greater than 1.

8.5.2 Black-box Testing of Data Abstractions

So far we've been thinking about testing just one function at a time. But data abstractions usually have many operations, and we need to test how those operations interact with one another. It's useful to distinguish *consumer* and *producers* of the data abstraction:

- A consumer is an operation that takes a value of the data abstraction as input.
- A producer is an operation that returns a value of the data abstraction as output.

For example, consider this set abstraction:

```
module type Set = sig

  (** ['a t] is the type of a set whose elements have type ['a]. *)
  type 'a t

  (** [empty] is the empty set. *)
  val empty : 'a t

  (** [size s] is the number of elements in [s]. *
      [size empty] is [0]. *)
  val size : 'a t -> int

  (** [add x s] is a set containing all the elements of
      [s] as well as element [x]. *)
  val add : 'a -> 'a t -> 'a t

  (** [mem x s] is [true] iff [x] is an element of [s]. *)
  val mem : 'a -> 'a t -> bool

end
```

The `empty` and `add` functions are producers; and the `size`, `add` and `mem` functions are consumers.

When black-box testing a data abstraction, we should test how each consumer of the data abstraction handles every path through each producer of it. In the `Set` example, that means testing the following:

- how `size` handles the `empty` set;
- how `size` handles a set produced by `add`, both when `add` leaves the set unchanged as well as when it increases the set;
- how `add` handles sets produced by `empty` as well as `add` itself;
- how `mem` handles sets produced by `empty` as well as `add`, including paths where `mem` is invoked on elements that have been added as well as elements that have not.

8.5.3 Glass-box Testing

Black-box testing is a good place to start when writing test cases, but ultimately it is not enough. In particular, it's not possible to determine how much coverage of the implementation a black-box test suite actually achieves—we actually need to know the implementation source code. Testing based on that code is known as *glass box* or *white box* testing. Glass-box testing can improve on black-box by testing *execution paths* through the implementation code: the series of expressions that is conditionally evaluated based on if-expressions, match-expressions, and function applications. Test cases that collectively exercise all paths are said to be *path-complete*. At a minimum, path-completeness requires that for every line of code, and even for every expression in the program, there should be a test case that causes it to be executed. Any unexecuted code could contain a bug if has never been tested.

For true path completeness we must consider all possible execution paths from start to finish of each function, and try to exercise every distinct path. In general this is infeasible, because there are too many paths. A good approach is to think of the set of paths as the space that we are trying to explore, and to identify boundary cases within this space that are worth testing.

For example, consider the following implementation of a function that finds the maximum of its three arguments:

```
let max3 x y z =
  if x > y then
    if x > z then x else z
  else
    if y > z then y else z
```

Black-box testing might lead us to invent many tests, but looking at the implementation reveals that there are only four paths through the code—the paths that return `x`, `z`, `y`, or `z` (again). We could test each of those paths with representative inputs such as: `max3 3 2 1`, `max3 3 2 4`, `max3 1 2 1`, `max3 1 2 3`.

When doing glass box testing, we should include test cases for each branch of each (nested) if expression, and each branch of each (nested) pattern match. If there are recursive functions, we should include test cases for the base cases as well as each recursive call. Also, we should include test cases to trigger each place where an exception might be raised.

Of course, path complete testing does not guarantee an absence of errors. We still need to test against the specification, i.e., do black-box testing. For example, here is a broken implementation of `max3`:

```
let max3 x y z = x
```

The test `max3 2 1 1` is path complete, but doesn't reveal the error.

8.5.4 Glass-box Testing of Data Abstractions

Look at the abstraction function and representation invariant for hints about what boundaries may exist in the space of values manipulated by a data abstraction. The rep invariant is a particularly effective tool for constructing useful test cases. Looking at the rep invariant of the `Rational` data abstraction above, we see that it requires that `q` is non-zero. Therefore we should construct test cases to see whether it's possible to cause that invariant to be violated.

8.5.5 Black-box vs. Glass-box

Black-box testing has some important advantages:

- It doesn't require that we see the code we are testing. Sometimes code will not be available in source code form, yet we can still construct useful test cases without it. The person writing the test cases does not need to understand the implementation.
- The test cases do not depend on the implementation. They can be written in parallel with or before the implementation. Further, good black-box test cases do not need to be changed, even if the implementation is completely rewritten.
- Constructing black-box test cases causes the programmer to think carefully about the specification and its implications. Many specification errors are caught this way.

The disadvantage of black box testing is that its coverage may not be as high as we'd like, because it has to work without the implementation.

8.5.6 Bisect

Glass-box testing can be aided by *code-coverage tools* that assess how much of the code has been exercised by a test suite. The bisect_ppx tool for OCaml can tell you which expressions in your program have been tested, and which have not. Here's how it works:

- You compile your code using Bisect_ppx (henceforth, just Bisect for short) as part of the compilation process. It *instruments* your code, mainly by inserting additional expressions to be evaluated.
- You run your code. The instrumentation that Bisect inserted causes your program to do something in addition to whatever functionality you programmed yourself: the program will now record which expressions from the source code actually get executed at run time, and which do not. Also, the program will now produce an output file that contains that information.
- You run a tool called `bisect-ppx-report` on that output file. It produces HTML showing you which parts of your code got executed, and which did not.

How does that help with computing coverage of a test suite? If you run your OUnit test suite, the test cases in it will cause the code in whatever functions they test to be executed. If you don't have enough test cases, some code in your functions will never be executed. The report produced by Bisect will show you exactly what code that is. You can then design new glass-box test cases to cause that code to execute, add them to your OUnit suite, and create a new Bisect report to confirm that the code really did get executed.

Bisect Tutorial.

1. Download the file sorts.ml. You will find an implementation of insertion sort and merge sort.
2. Download the file test_sorts.ml. It has the skeleton for an OUnit test suite.
3. Create a `dune` file to execute `test_sorts`:

```
(executable
 (name test_sorts)
 (libraries ounit2)
 (instrumentation
  (backend bisect_ppx)))
```

4. Run:

```
$ dune exec --instrument-with bisect_ppx ./test_sorts.exe
```

   That will execute the test suite with Bisect coverage enabled, causing some files named `bisectNNNN.coverage` to be produced.

5. Run:

```
$ bisect-ppx-report html
```

   to generate the Bisect report from your test suite execution. The report is in a newly-created directory named `_coverage`.

6. Open the file `_coverage/index.html` in a web browser. Look at the per-file coverage; you'll see we've managed to test a few percent of `sorts.ml` with our test suite so far. Click on the link in that report for `sorts.ml`. You'll see that we've managed to cover only one line of the source code.
7. There are some additional tests in the test file. Try un-commenting those, as documented in the test file, and increasing your code coverage. Between each run, you will need to delete the `bisectNNNN.coverage` files, otherwise the report will contain information from those previous runs:

```
$ rm bisect*.coverage
```

By the time you're done un-commenting the provided tests, you should be at 25% coverage, including all of the insertion sort implementation. For fun, try adding more tests to get 100% coverage of merge sort.

Parallelism. OUnit will by default attempt to run some of the tests in parallel, which reduces the time it takes to run a large test suite, at the tradeoff of making it nondeterministic in what order the tests run. It's possible for that to affect coverage if you are testing imperative code. To make the tests run one at a time, in order, you can pass the flag `-runner sequential` to the executable. OUnit will see that flag and cease parallelization:

```
$ dune exec --instrument-with bisect_ppx ./test_sorts.exe -- -runner sequential
```

8.6 Randomized Testing with QCheck

Randomized testing aka *fuzz testing* is the process of generating random inputs and feeding them to a program or a function to see whether the program behaves correctly. The immediate issue is how to determine what the correct output is for a given input. If a *reference implementation* is available—that is, an implementation that is believed to be correct but in some other way does not suffice (e.g., its performance is too slow, or it is in a different language)—then the outputs of the two implementations can be compared. Otherwise, perhaps some *property* of the output could be checked. For example,

- "not crashing" is a property of interest in user interfaces;
- adding n elements to a data collection then removing those elements, and ending up with an empty collection, is a property of interest in data structures; and
- encrypting a string under a key then decrypting it under that key and getting back the original string is a property of interest in an encryption scheme like Enigma.

Randomized testing is an incredibly powerful technique. It is often used in testing programs for security vulnerabilities. The `qcheck` package for OCaml supports randomized testing. We'll look at it, next, after we discuss random number generation.

8.6.1 Random Number Generation

To understand randomized testing, we need to take a brief digression into random number generation.

Most languages provide the facility to generate random numbers. In truth, these generators are usually not truly random (in the sense that they are completely unpredictable) but in fact are *pseudorandom*: the sequence of numbers they generate pass good statistical tests to ensure there is no discernible pattern in them, but the sequence itself is a deterministic function of an initial *seed* value. (Recall that the prefix *pseudo* is from the Greek *pseudēs* meaning "false".) Java and Python both provide pseudorandom number generators (PRNGs). So does OCaml in the standard library's `Random` module.

An Experiment. Start a new session of utop and enter the following:

```
# Random.int 100;;
# Random.int 100;;
# Random.int 100;;
```

Each response will be an integer i such that $0 \leq i < 100$.

Now quit utop and start another new session. Enter the same phrases again. You will get the same responses as last time. In fact, unless your OCaml installation is somehow different than that used to produce this book, you will get the same numbers as those below:

```
Random.int 100;;
Random.int 100;;
Random.int 100;;
```

Not exactly unpredictable, eh?

PRNGs. Although for purposes of security and cryptography a PRNG leads to terrible vulnerabilities, for other purposes—including testing and simulation—PRNGs are just fine. Their predictability can even be useful: given the same initial seed, a PRNG will always produce the same sequence of pseudorandom numbers, leading to the ability to repeat a particular sequence of tests or a particular simulation.

The way a PRNG works in general is that it initializes a *state* that it keeps internally from the initial seed. From then on, each time the PRNG generates a new value, it imperatively updates that state. The `Random` module makes it possible to manipulate that state in limited ways. For example, you can

- get the current state with `Random.get_state`,
- duplicate the current state with `Random.State.copy`,
- request a random int generated from a particular state with `Random.State.int`, and
- initialize the state yourself. The functions `Random.self_init` and `Random.State.make_self_init` will choose a "random" seed to initialize the state. They do so by sampling from a special Unix file named `/dev/urandom`, which is meant to provide as close to true randomness as a computer can.

Repeating the Experiment. Start a new session of utop. Enter the following:

```
# Random.self_init ();;
# Random.int 100;;
# Random.int 100;;
# Random.int 100;;
```

Now do that a second time (it doesn't matter whether you exit utop or not in between). You will notice that you get a different sequence of values. With high probability, what you get will be different than the values below:

```
Random.self_init ();;
Random.int 100;;
Random.int 100;;
Random.int 100;;
```

8.6.2 QCheck Abstractions

QCheck has three abstractions we need to cover before using it for testing: generators, properties, and arbitraries. If you want to follow along in utop, load QCheck with this directive:

```
#require "qcheck";;
```

Generators. One of the key pieces of functionality provided by QCheck is the ability to generate pseudorandom values of various types. Here is some of the signature of the module that does that:

```
module QCheck : sig
  ...
  module Gen :
  sig
    type 'a t = Random.State.t -> 'a
```

(continues on next page)

(continued from previous page)

```
    val int : int t
    val generate : ?rand:Random.State.t -> n:int -> 'a t -> 'a list
    val generate1 : ?rand:Random.State.t -> 'a t -> 'a
    ...
  end
  ...
end
```

An `'a QCheck.Gen.t` is a function that takes in a PRNG state and uses it to produce a pseudorandom value of type `'a`. So `QCheck.Gen.int` produces pseudorandom integers. The function `generate1` actually does the generation of one pseudorandom value. It takes an optional argument that is a PRNG state; if that argument is not supplied, it uses the default PRNG state. The function `generate` produces a list of `n` pseduorandom values.

QCheck implements many producers of pseudorandom values. Here are a few more of them:

```
module QCheck : sig
  ...
  module Gen :
  sig
    val int : int t
    val small_int : int t
    val int_range : int -> int -> int t
    val list : 'a t -> 'a list t
    val list_size : int t -> 'a t -> 'a list t
    val string : ?gen:char t -> string t
    val small_string : ?gen:char t -> string t
    ...
  end
  ...
end
```

You can read the documentation of those and many others.

Properties. It's tempting to think that QCheck would enable us to test a function by generating many pseudorandom inputs to the function, running the function on them, then checking that the outputs are correct. But there's immediately a problem: how can QCheck know what the correct output is for each of those inputs? Since they're randomly generated, the test engineer can't hardcode the right outputs.

So instead, QCheck allows us to check whether a *property* of each output holds. A property is a function of type `t -> bool`, for some type `t`, that tells use whether the value of type `t` exhibits some desired characteristic. Here, for example, are two properties; one that determines whether an integer is even, and another that determines whether a list is sorted in non-decreasing order according to the built-in <= operator:

```
let is_even n = n mod 2 = 0

let rec is_sorted = function
  | [] -> true
  | [ h ] -> true
  | h1 :: (h2 :: t as t') -> h1 <= h2 && is_sorted t'
```

Arbitraries. The way we present to QCheck the outputs to be checked is with a value of type `'a QCheck.Arbitrary`. This type represents an "arbitrary" value of type `'a`—that is, it has been pseudorandomly chosen as a value that we want to check, and more specifically, to check whether it satisfies a property.

We can create *arbitraries* out of generators using the function `QCheck.make : 'a QCheck.Gen.t -> 'a QCheck.arbitrary`. (Actually that function takes some optional arguments that we elide here.) This isn't actually

the normal way to create arbitraries, but it's a simple way that will help us understand them; we'll get to the normal way in a little while. For example, the following expression represents an arbitrary integer:

```
QCheck.make QCheck.Gen.int
```

8.6.3 Testing Properties

To construct a QCheck test, we create an arbitrary and a property, and pass them to `QCheck.Test.make`, whose type can be simplified to:

```
QCheck.Test.make : 'a QCheck.arbitrary -> ('a -> bool) -> QCheck.Test.t
```

In reality, that function also takes several optional arguments that we elide here. The test will generate some number of arbitraries and check whether the property holds of each of them. For example, the following code creates a QCheck test that checks whether an arbitrary integer is even:

```
let t = QCheck.Test.make (QCheck.make QCheck.Gen.int) is_even
```

If we want to change the number of arbitraries that are checked, we can pass an optional integer argument `~count` to `QCheck.Test.make`.

We can run that test with `QCheck_runner.run_tests : QCheck.Test.t list -> int`. (Once more, that function takes some optional arguments that we elide here.) The integer it returns is 0 if all the tests in the list pass, and 1 otherwise. For the test above, running it will output 1 with high probability, because it will generate at least one odd integer.

```
QCheck_runner.run_tests [t]
```

Unfortunately, that output isn't very informative; it doesn't tell us what particular values failed to satisfy the property! We'll fix that problem in a little while.

If you want to make an OCaml program that runs QCheck tests and prints the results, there is a function `QCheck_runner.run_tests_main` that works much like `OUnit2.run_test_tt_main`: just invoke it as the final expression in a test file. For example:

```
let tests = (* code that constructs a [QCheck.Test.t list] *)
let _ = QCheck_runner.run_tests_main tests
```

To compile QCheck code, just add the `qcheck` library to your `dune` file:

```
(executable
 ...
 (libraries ... qcheck))
```

QCheck tests can be converted to OUnit tests and included in the usual kind of OUnit test suite we've been writing all along. The function that does this is:

```
QCheck_runner.to_ounit2_test
```

8.6.4 Informative Output from QCheck

We noted above that the output of QCheck so far has told us only *whether* some arbitraries satisfied a property, but not *which* arbitraries failed to satisfy it. Let's fix that problem.

The issue is with how we constructed an arbitrary directly out of a generator. An arbitrary is properly more than just a generator. The QCheck library needs to know how to print values of the generator, and a few other things as well. You can see that in the definition of `'a QCheck.arbitrary`:

```
#show QCheck.arbitrary;;
```

In addition to the generator field `gen`, there is a field containing an optional function to print values from the generator, and a few other optional fields as well. Luckily, we don't usually have to find a way to complete those fields ourselves; the `QCheck` module provides many arbitraries that correspond to the generators found in `QCheck.Gen`:

```
module QCheck :
  sig
    ...
  val int : int arbitrary
  val small_int : int arbitrary
  val int_range : int -> int -> int arbitrary
  val list : 'a arbitrary -> 'a list arbitrary
  val list_of_size : int Gen.t -> 'a arbitrary -> 'a list arbitrary
  val string : string arbitrary
  val small_string : string arbitrary
    ...
  end
```

Using those arbitraries, we can get improved error messages:

```
let t = QCheck.Test.make ~name:"my_test" QCheck.int is_even;;
QCheck_runner.run_tests [t];;
```

The output tells us the `my_test` failed, and shows us the input that caused the failure.

8.6.5 Testing Functions with QCheck

The final piece of the QCheck puzzle is to use a randomly generated input to test whether a function's output satisfies some property. For example, here is a QCheck test to see whether the output of `double` is correct:

```
let double x = 2 * x;;
let double_check x = double x = x + x;;
let t = QCheck.Test.make ~count:1000 QCheck.int double_check;;
QCheck_runner.run_tests [t];;
```

Above, `double` is the function we are testing. The property we're testing `double_check`, is that `double x` is always `x + x`. We do that by having QCheck create 1000 arbitrary integers and test that the property holds of each of them.

Here are a couple more examples, drawn from QCheck's own documentation. The first checks that `List.rev` is an *involution*, meaning that applying it twice brings you back to the original list. That is a property that should hold of a correct implementation of list reversal.

```
let rev_involutive lst = List.(lst |> rev |> rev = lst);;
let t = QCheck.(Test.make ~count:1000 (list int) rev_involutive);;
QCheck_runner.run_tests [t];;
```

Indeed, running 1000 random tests reveals that none of them fails. The `int` generator used above generates integers uniformly over the entire range of OCaml integers. The `list` generator creates lists whose elements are individual generated by `int`. According to the documentation of `list`, the length of each list is randomly generated by another generator `nat`, which generates "small natural numbers." What does that mean? It isn't specified. But if we read the current source code, we see that those are integers from 0 to 10,000, and biased toward being smaller numbers in that range.

The second example checks that all lists are sorted. Of course, not all lists *are* sorted! So we should expect this test to fail.

```
let is_sorted lst = lst = List.sort Stdlib.compare lst;;
let t = QCheck.(Test.make ~count:1000 (list small_nat) is_sorted);;
QCheck_runner.run_tests [t];;
```

The output shows an example of a list that is not sorted, hence violates the property. Generator `small_nat` is like `nat` but ranges from 0 to 100.

8.7 Proving Correctness

Testing provides evidence of correctness, but not full assurance. Even after extensive black-box and glass-box testing, maybe there's still some test case the programmer failed to invent, and that test case would reveal a fault in the program.

> Program testing can be used to show the presence of bugs, but never to show their absence.
>
> ---Edsger W. Dijkstra

The point is not that testing is useless! It can be quite effective. But it is a kind of *inductive reasoning*, in which evidence (i.e., passing tests) accumulates in support of a conclusion (i.e., correctness of the program) without absolutely guaranteeing the validity of that conclusion. (Note that the word "inductive" here is being used in a different sense than the proof technique known as induction.) To get that guarantee, we turn to *deductive reasoning*, in which we proceed from premises and rules about logic to a valid conclusion. In other words, we prove the correctness of the program. Our goal, next, is to learn some techniques for such correctness proofs. These techniques are known as *formal methods* because of their use of logical formalism.

Correctness here means that the program produces the right output according to a *specification*. Specifications are usually provided in the documentation of a function (hence the name "specification comment"): they describe the program's precondition and postcondition. Postconditions, as we have been writing them, have the form `[f x] is "...a description of the output in terms of the input [x]..."`. For example, the specification of a factorial function could be:

```
(** [fact n] is [n!]. Requires: [n >= 0]. *)
let rec fact n = ...
```

The postcondition is asserting an equality between the output of the function and some English description of a computation on the input. *Formal verification* is the task for proving that the implementation of the function satisfies its specification.

Equalities are one of the fundamental ways we think about correctness of functional programs. The absence of mutable state makes it possible to reason straightforwardly about whether two expressions are equal. It's difficult to do that in an imperative language, because those expressions might have side effects that change the state.

8.7.1 Equality

When are two expressions equal? Two possible answers are:

- When they are syntactically identical.
- When they are semantically equivalent: they produce the same value.

For example, are `42` and `41+1` equal? The syntactic answer would say they are not, because they involve different tokens. The semantic answer would say they are: they both produce the value `42`.

What about functions: are `fun x -> x` and `fun y -> y` equal? Syntactically they are different. But semantically, they both produce a value that is the identity function: when they are applied to an input, they will both produce the same output. That is, `(fun x -> x) z = z`, and `(fun y -> y) z = z`. If it is the case that for all inputs two functions produce the same output, we will consider the functions to be equal:

```
if (forall x, f x = g x), then f = g.
```

That definition of equality for functions is known as the *Axiom of Extensionality* in some branches of mathematics; henceforth we'll refer to it simply as "extensionality".

Here we will adopt the semantic approach. If `e1` and `e2` evaluate to the same value `v`, then we write `e1 = e2`. We are using = here in a mathematical sense of equality, not as the OCaml polymorphic equality operator. For example, we allow `(fun x -> x) = (fun y -> y)`, even though OCaml's operator would raise an exception and refuse to compare functions.

We're also going to restrict ourselves to expressions that are well typed, pure (meaning they have no side effects), and total (meaning they don't have exceptions or infinite loops).

8.7.2 Equational Reasoning

Consider these functions:

```
let twice f x = f (f x)
let compose f g x = f (g x)
```

We know from the rules of OCaml evaluation that `twice h x = h (h x)`, and likewise, `compose h h x = h (h x)`. Thus we have:

```
twice h x = h (h x) = compose h h x
```

Therefore can conclude that `twice h x = compose h h x`. And by extensionality we can simplify that equality: Since `twice h x = compose h h x` holds for all `x`, we can conclude `twice h = compose h h`.

As another example, suppose we define an infix operator for function composition:

```
let ( << ) = compose
```

Then we can prove that composition is associative, using equational reasoning:

```
Theorem: (f << g) << h  =  f << (g << h)

Proof: By extensionality, we need to show
  ((f << g) << h) x  =  (f << (g << h)) x
for an arbitrary x.
```

(continues on next page)

(continued from previous page)

```
  ((f << g) << h) x
= (f << g) (h x)
= f (g (h x))

and

  (f << (g << h)) x
= f ((g << h) x)
= f (g (h x))

So ((f << g) << h) x = f (g (h x)) = (f << (g << h)) x.

QED
```

All of the steps in the equational proof above follow from evaluation. Another format for writing the proof would provide hints as to why each step is valid:

```
  ((f << g) << h) x
=   { evaluation of << }
  (f << g) (h x)
=   { evaluation of << }
  f (g (h x))

and

  (f << (g << h)) x
=   { evaluation of << }
  f ((g << h) x)
=   { evaluation of << }
  f (g (h x))
```

8.7.3 Induction on Natural Numbers

The following function sums the non-negative integers up to `n`:

```
let rec sumto n =
  if n = 0 then 0 else n + sumto (n - 1)
```

You might recall that the same summation can be expressed in closed form as `n * (n + 1) / 2`. To prove that `forall n >= 0, sumto n = n * (n + 1) / 2`, we will need *mathematical induction*.

Recall that induction on the natural numbers (i.e., the non-negative integers) is formulated as follows:

```
forall properties P,
  if P(0),
  and if forall k, P(k) implies P(k + 1),
  then forall n, P(n)
```

That is called the *induction principle* for natural numbers. The *base case* is to prove `P(0)`, and the *inductive case* is to prove that `P(k + 1)` holds under the assumption of the *inductive hypothesis* `P(k)`.

Let's use induction to prove the correctness of `sumto`.

```
Claim: sumto n = n * (n + 1) / 2

Proof: by induction on n.
P(n) = sumto n = n * (n + 1) / 2

Base case: n = 0
Show: sumto 0 = 0 * (n + 1) / 2

  sumto 0
=   { evaluation }
  0
=   { algebra }
  0 * (n + 1) / 2

Inductive case: n = k + 1
Show: sumto (k + 1) = (k + 1) * ((k + 1) + 1) / 2
IH: sumto k = k * (k + 1) / 2

  sumto (k + 1)
=   { evaluation }
  k + 1 + sumto k
=   { IH }
  k + 1 + k * (k + 1) / 2
=   { algebra }
  (k + 1) * (k + 2) / 2

QED
```

Note that we have been careful in each of the cases to write out what we need to show, as well as to write down the inductive hypothesis. It is important to show all this work.

Suppose we now define:

```
let sumto_closed n = n * (n + 1) / 2
```

Then as a corollary to our previous claim, by extensionality we can conclude

```
sumto_closed = sumto
```

Technically that equality holds only inputs that are natural numbers. But since all our examples henceforth will be for naturals, not integers per se, we will elide stating any preconditions or restrictions regarding natural numbers.

8.7.4 Programs as Specifications

We have just proved the correctness of an efficient implementation relative to an inefficient implementation. The inefficient implementation, `sumto`, serves as a specification for the efficient implementation, `sumto_closed`.

That technique is common in verifying functional programs: write an obviously correct implementation that is lacking in some desired property, such as efficiency, then prove that a better implementation is equal to the original.

Let's do another example of this kind of verification. This time, well use the factorial function.

The simple, obviously correct implementation of factorial would be:

```
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
```

A tail-recursive implementation would be more efficient about stack space:

```
let rec facti acc n =
  if n = 0 then acc else facti (acc * n) (n - 1)

let fact_tr n = facti 1 n
```

The `i` in the name `facti` stands for *iterative*. We call this an iterative implementation because it strongly resembles how the same computation would be expressed using a loop (that is, an iteration construct) in an imperative language. For example, in Java we might write:

```
int facti (int n) {
  int acc = 1;
  while (n != 0) {
    acc *= n;
    n--;
  }
  return acc;
}
```

Both the OCaml and Java implementation of `facti` share these features:

- they start `acc` at `1`
- they check whether `n` is `0`
- they multiply `acc` by `n`
- they decrement `n`
- they return the accumulator, `acc`

Let's try to prove that `fact_tr` correctly implements the same computation as `fact`.

```
Claim: forall n, fact n = fact_tr n

Since fact_tr n = facti 1 n, it suffices to show fact n = facti 1 n.

Proof: by induction on n.
P(n) = fact n = facti 1 n

Base case: n = 0
Show: fact 0 = facti 1 0

  fact 0
=   { evaluation }
  1
=   { evaluation }
  facti 1 0

Inductive case: n = k + 1
Show: fact (k + 1) = facti 1 (k + 1)
IH: fact k = facti 1 k

  fact (k + 1)
=   { evaluation }
  (k + 1) * fact k
=   { IH }
  (k + 1) * facti 1 k
```

(continues on next page)

(continued from previous page)

```
  facti 1 (k + 1)
=   { evaluation }
  facti (1 * (k + 1)) k
=   { evaluation }
  facti (k + 1) k

Unfortunately, we're stuck.  Neither side of what we want to show
can be manipulated any further.

ABORT
```

We know that `facti (k + 1) k` and `(k + 1) * facti 1 k` should yield the same value. But the IH allows us only to use `1` as the second argument to `facti`, instead of a bigger argument like `k + 1`. So our proof went astray the moment we used the IH. We need a stronger inductive hypothesis!

So let's strengthen the claim we are making. Instead of showing that `fact n = facti 1 n`, we'll try to show `forall p, p * fact n = facti p n`. That generalizes the `k + 1` we were stuck on to an arbitrary quantity `p`.

```
Claim: forall n, forall p . p * fact n = facti p n

Proof: by induction on n.
P(n) = forall p, p * fact n = facti p n

Base case:  n = 0
Show: forall p,  p * fact 0 = facti p 0

  p * fact 0
=   { evaluation and algebra }
  p
=   { evaluation }
  facti p 0

Inductive case: n = k + 1
Show: forall p,  p * fact (k + 1) = facti p (k + 1)
IH: forall p,  p * fact k = facti p k

  p * fact (k + 1)
=   { evaluation }
  p * (k + 1) * fact k
=   { IH, instantiating its p as p * (k + 1) }
  facti (p * (k + 1)) k

  facti p (k + 1)
=   { evaluation }
  facti (p * (k + 1)) k

QED

Claim: forall n, fact n = fact_tr n

Proof:

  fact n
=   { algebra }
```

(continues on next page)

(continued from previous page)

```
  1 * fact n
=   { previous claim }
  facti 1 n
=   { evaluation }
  fact_tr n

QED
```

That finishes our proof that the efficient, tail-recursive function `fact_tr` is equivalent to the simple, recursive function `fact`. In essence, we have proved the correctness of `fact_tr` using `fact` as its specification.

8.7.5 Recursion vs. Iteration

We added an accumulator as an extra argument to make the factorial function be tail recursive. That's a trick we've seen before. Let's abstract and see how to do it in general.

Suppose we have a recursive function over integers:

```
let rec f_r n =
  if n = 0 then i else op n (f_r (n - 1))
```

Here, the `r` in `f_r` is meant to suggest that `f_r` is a recursive function. The `i` and `op` are pieces of the function that are meant to be replaced by some concrete value `i` and operator `op`. For example, with the factorial function, we have:

```
f_r = fact
i = 1
op = ( * )
```

Such a function can be made tail recursive by rewriting it as follows:

```
let rec f_i acc n =
  if n = 0 then acc
  else f_i (op acc n) (n - 1)

let f_tr = f_i i
```

Here, the `i` in `f_i` is meant to suggest that `f_i` is an iterative function, and `i` and `op` are the same as in the recursive version of the function. For example, with factorial we have:

```
f_i = fact_i
i = 1
op = ( * )
f_tr = fact_tr
```

We can prove that `f_r` and `f_tr` compute the same function. During the proof, next, we will discover certain conditions that must hold of `i` and `op` to make the transformation to tail recursion be correct.

```
Theorem: f_r = f_tr

Proof:  By extensionality, it suffices to show that forall n, f_r n = f_tr n.

As in the previous proof for fact, we will need a strengthened induction
hypothesis. So we first prove this lemma, which quantifies over all accumulators
```

(continues on next page)

(continued from previous page)

```
that could be input to f_i, rather than only i:

  Lemma: forall n, forall acc, op acc (f_r n) = f_i acc n

  Proof of Lemma: by induction on n.
  P(n) = forall acc, op acc (f_r n) = f_i acc n

  Base: n = 0
  Show: forall acc, op acc (f_r 0) = f_i acc 0

    op acc (f_r 0)
  =   { evaluation }
    op acc i
  =   { if we assume forall x, op x i = x }
    acc

    f_i acc 0
  =   { evaluation }
    acc

  Inductive case: n = k + 1
  Show: forall acc, op acc (f_r (k + 1)) = f_i acc (k + 1)
  IH: forall acc, op acc (f_r k) = f_i acc k

    op acc (f_r (k + 1))
  =   { evaluation }
    op acc (op (k + 1) (f_r k))
  =   { if we assume forall x y z, op x (op y z) = op (op x y) z }
    op (op acc (k + 1)) (f_r k)

    f_i acc (k + 1)
  =   { evaluation }
    f_i (op acc (k + 1)) k
  =   { IH, instantiating acc as op acc (k + 1)}
    op (op acc (k + 1)) (f_r k)

  QED

The proof then follows almost immediately from the lemma:

  f_r n
=   { if we assume forall x, op i x = x }
  op i (f_r n)
=   { lemma, instantiating acc as i }
  f_i i n
=   { evaluation }
  f_tr n

QED
```

Along the way we made three assumptions about i and op:

1. `forall x, op x i = x`
2. `op x (op y z) = op (op x y) z`
3. `forall x, op i x = x`

The first and third say that `i` is an *identity* of `op`: using it on the left or right side leaves the other argument `x` unchanged. The second says that `op` is *associative*. Both those assumptions held for the values we used in the factorial functions:

- `op` is multiplication, which is associative.
- `i` is `1`, which is an identity of multiplication: multiplication by 1 leaves the other argument unchanged.

So our transformation from a recursive to a tail-recursive function is valid as long as the operator applied in the recursive call is associative, and the value returned in the base case is an identity of that operator.

Returning to the `sumto` function, we can apply the theorem we just proved to immediately get a tail-recursive version:

```
let rec sumto_r n =
  if n = 0 then 0 else n + sumto_r (n - 1)
```

Here, the operator is addition, which is associative; and the base case is zero, which is an identity of addition. Therefore our theorem applies, and we can use it to produce the tail-recursive version without even having to think about it:

```
let rec sumto_i acc n =
  if n = 0 then acc else sumto_i (acc + n) (n - 1)

let sumto_tr = sumto_i 0
```

We already know that `sumto_tr` is correct, thanks to our theorem.

8.7.6 Termination

Sometimes correctness of programs is further divided into:

- **partial correctness**: meaning that *if* a program terminates, then its output is correct; and
- **total correctness**: meaning that a program *does* terminate, *and* its output is correct.

Total correctness is therefore the conjunction of partial correctness and termination. Thus far, we have been proving partial correctness.

To prove that a program terminates is difficult. Indeed, it is impossible in general for an algorithm to do so: a computer can't precisely decide whether a program will terminate. (Look up the "halting problem" for more details.) But, a smart human sometimes can do so.

There is a simple heuristic that can be used to show that a recursive function terminates:

- All recursive calls are on a "smaller" input, and
- all base cases are terminating.

For example, consider the factorial function:

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)
```

The base case, `1`, obviously terminates. The recursive call is on `n - 1`, which is a smaller input than the original `n`. So `fact` always terminates (as long as its input is a natural number).

The same reasoning applies to all the other functions we've discussed above.

To make this more precise, we need a notion of what it means to be smaller. Suppose we have a binary relation $<$ on inputs. Despite the notation, this relation need not be the less-than relation on integers—although that will work for `fact`. Also suppose that it is never possible to create an infinite sequence `x0 > x1 > x2 > x3 ...` of elements using this

relation. (Where of course `a > b` iff `b < a`.) That is, there are no infinite descending chains of elements: once you pick a starting element `x0`, there can be only a finite number of "descents" according to the < relation before you bottom out and hit a base case. This property of < makes it a *well-founded relation*.

So, a recursive function terminates if all its recursive calls are on elements that are smaller according to <. Why? Because there can be only a finite number of calls before a base case is reached, and base cases must terminate.

The usual < relation is well-founded on the natural numbers, because eventually any chain must reach the base case of 0. But it is not well-founded on the integers, which can get just keep getting smaller: `-1 > -2 > -3 > ...`.

Here's an interesting function for which the usual < relation doesn't suffice to prove termination:

```
let rec ack = function
  | (0, n) -> n + 1
  | (m, 0) -> ack (m - 1, 1)
  | (m, n) -> ack (m - 1, ack (m, n - 1))
```

This is known as *Ackermann's function*. It grows faster than any exponential function. Try running `ack (1, 1)`, `ack (2, 1)`, `ack (3, 1)`, then `ack (4, 1)` to get a sense of that. It also is a famous example of a function that can be implemented with `while` loops but not with `for` loops. Nonetheless, it does terminate.

To show that, the base case is easy: when the input is `(0, _)`, the function terminates. But in other cases, it makes a recursive call, and we need to define an appropriate < relation. It turns out *lexicograpic ordering* on pairs works. Define `(a, b) < (c, d)` if:

- `a < c`, or
- `a = c` and `b < d`.

The < order in those two cases is the usual < on natural numbers.

In the first recursive call, `(m - 1, 1) < (m, 0)` by the first case of the definition of <, because `m - 1 < m`. In the nested recursive call `ack (m - 1, ack (m, n - 1))`, both cases are needed:

- `(m, n - 1) < (m, n)` because `m = m` and `n - 1 < n`
- `(m - 1, _) < (m, n)` because `m - 1 < m`.

8.8 Structural Induction

So far we've proved the correctness of recursive functions on natural numbers. We can do correctness proofs about recursive functions on variant types, too. That requires us to figure out how induction works on variants. We'll do that, next, starting with a variant type for representing natural numbers, then generalizing to lists, trees, and other variants. This inductive proof technique is sometimes known as *structural induction* instead of *mathematical induction*. But that's just a piece of vocabulary; don't get hung up on it. The core idea is completely the same.

8.8.1 Induction on Naturals

We used OCaml's `int` type as a representation of the naturals. Of course, that type is somewhat of a mismatch: negative `int` values don't represent naturals, and there is an upper bound to what natural numbers we can represent with `int`.

Let's fix those problems by defining our own variant to represent natural numbers:

```
type nat = Z | S of nat
```

The constructor `Z` represents zero; and the constructor `S` represents the successor of another natural number. So,

- 0 is represented by `Z`,
- 1 by `S Z`,
- 2 by `S (S Z)`,
- 3 by `S (S (S Z))`,

and so forth. This variant is thus a *unary* (as opposed to binary or decimal) representation of the natural numbers: the number of times `S` occurs in a value `n : nat` is the natural number that `n` represents.

We can define addition on natural numbers with the following function:

```
let rec plus a b =
  match a with
  | Z -> b
  | S k -> S (plus k b)
```

Immediately we can prove the following rather trivial claim:

```
Claim:  plus Z n = n

Proof:

  plus Z n
=   { evaluation }
  n

QED
```

But suppose we want to prove this also trivial-seeming claim:

```
Claim:  plus n Z = n

Proof:

  plus n Z
=
  ???
```

We can't just evaluate `plus n Z`, because `plus` matches against its first argument, not second. One possibility would be to do a case analysis: what if `n` is `Z`, vs. `S k` for some `k`? Let's attempt that.

```
Proof:

By case analysis on n, which must be either Z or S k.

Case:  n = Z

  plus Z Z
=   { evaluation }
  Z

Case:  n = S k

  plus (S k) Z
=   { evaluation }
  S (plus k Z)
```

(continues on next page)

(continued from previous page)

```
=
  ???
```

We are again stuck, and for the same reason: once more `plus` can't be evaluated any further.

When you find yourself needing to solve the same subproblem in programming, you use recursion. When it happens in a proof, you use induction!

We'll need an induction principle for `nat`. Here it is:

```
forall properties P,
  if P(Z),
  and if forall k, P(k) implies P(S k),
  then forall n, P(n)
```

Compare that to the induction principle we used for natural numbers before, when we were using `int` in place of natural numbers:

```
forall properties P,
  if P(0),
  and if forall k, P(k) implies P(k + 1),
  then forall n, P(n)
```

There's no essential difference between the two: we just use `Z` in place of `0`, and `S k` in place of `k + 1`.

Using that induction principle, we can carry out the proof:

```
Claim:  plus n Z = n

Proof: by induction on n.
P(n) = plus n Z = n

Base case: n = Z
Show: plus Z Z = Z

  plus Z Z
=   { evaluation }
  Z

Inductive case: n = S k
IH: plus k Z = k
Show: plus (S k) Z = S k

  plus (S k) Z
=   { evaluation }
  S (plus k Z)
=   { IH }
  S k

QED
```

8.8.2 Induction on Lists

It turns out that natural numbers and lists are quite similar, when viewed as data types. Here are the definitions of both, aligned for comparison:

```
type    nat  = Z  | S      of nat
type 'a list = [] | ( :: ) of 'a * 'a list
```

Both types have a constructor representing a concept of "nothing". Both types also have a constructor representing "one more" than another value of the type: `S n` is one more than `n`, and `h :: t` is a list with one more element than `t`.

The induction principle for lists is likewise quite similar to the induction principle for natural numbers. Here is the principle for lists:

```
forall properties P,
  if P([]),
  and if forall h t, P(t) implies P(h :: t),
  then forall lst, P(lst)
```

An inductive proof for lists therefore has the following structure:

```
Proof: by induction on lst.
P(lst) = ...

Base case: lst = []
Show: P([])

Inductive case: lst = h :: t
IH: P(t)
Show: P(h :: t)
```

Let's try an example of this kind of proof. Recall the definition of the append operator:

```
let rec append lst1 lst2 =
  match lst1 with
  | [] -> lst2
  | h :: t -> h :: append t lst2

let ( @ ) = append
```

We'll prove that append is associative.

```
Theorem: forall xs ys zs, xs @ (ys @ zs) = (xs @ ys) @ zs

Proof: by induction on xs.
P(xs) = forall ys zs, xs @ (ys @ zs) = (xs @ ys) @ zs

Base case: xs = []
Show: forall ys zs, [] @ (ys @ zs) = ([] @ ys) @ zs

  [] @ (ys @ zs)
=   { evaluation }
  ys @ zs
=   { evaluation }
  ([] @ ys) @ zs
```

(continues on next page)

(continued from previous page)

```
Inductive case: xs = h :: t
IH: forall ys zs, t @ (ys @ zs) = (t @ ys) @ zs
Show: forall ys zs, (h :: t) @ (ys @ zs) = ((h :: t) @ ys) @ zs

  (h :: t) @ (ys @ zs)
=   { evaluation }
  h :: (t @ (ys @ zs))
=   { IH }
  h :: ((t @ ys) @ zs)

  ((h :: t) @ ys) @ zs
=   { evaluation of inner @ }
  (h :: (t @ ys)) @ zs
=   { evaluation of outer @ }
  h :: ((t @ ys) @ zs)

QED
```

8.8.3 A Theorem about Folding

When we studied `List.fold_left` and `List.fold_right`, we discussed how they sometimes compute the same function, but in general do not. For example,

```
  List.fold_left ( + ) 0 [1; 2; 3]
= (((0 + 1) + 2) + 3
= 6
= 1 + (2 + (3 + 0))
= List.fold_right ( + ) lst [1; 2; 3]
```

but

```
  List.fold_left ( - ) 0 [1; 2; 3]
= (((0 - 1) - 2) - 3
= -6
<> 2
= 1 - (2 - (3 - 0))
= List.fold_right ( - ) lst [1; 2; 3]
```

Based on the equations above, it looks like the fact that + is commutative and associative, whereas − is not, explains this difference between when the two fold functions get the same answer. Let's prove it!

First, recall the definitions of the fold functions:

```
let rec fold_left f acc lst =
  match lst with
  | [] -> acc
  | h :: t -> fold_left f (f acc h) t

let rec fold_right f lst acc =
  match lst with
  | [] -> acc
  | h :: t -> f h (fold_right f t acc)
```

Second, recall what it means for a function `f : 'a -> 'a` to be commutative and associative:

```
Commutative:  forall x y, f x y = f y x
Associative:  forall x y z, f x (f y z) = f (f x y) z
```

Those might look a little different than the normal formulations of those properties, because we are using `f` as a prefix operator. If we were to write `f` instead as an infix operator `op`, they would look more familiar:

```
Commutative:  forall x y, x op y = y op x
Associative:  forall x y z, x op (y op z) = (x op y) op z
```

When `f` is both commutative and associative we have this little interchange lemma that lets us swap two arguments around:

```
Lemma (interchange): f x (f y z) = f y (f x z)

Proof:

  f x (f y z)
=   { associativity }
  f (f x y) z
=   { commutativity }
  f (f y x) z
=   { associativity }
  f y (f z x)

QED
```

Now we're ready to state and prove the theorem.

```
Theorem: If f is commutative and associative, then
  forall lst acc,
    fold_left f acc lst = fold_right f lst acc.

Proof: by induction on lst.
P(lst) = forall acc,
  fold_left f acc lst = fold_right f lst acc

Base case: lst = []
Show: forall acc,
  fold_left f acc [] = fold_right f [] acc

  fold_left f acc []
=   { evaluation }
  acc
=   { evaluation }
  fold_right f [] acc

Inductive case: lst = h :: t
IH: forall acc,
  fold_left f acc t = fold_right f t acc
Show: forall acc,
  fold_left f acc (h :: t) = fold_right f (h :: t) acc

  fold_left f acc (h :: t)
=   { evaluation }
  fold_left f (f acc h) t
=   { IH with acc := f acc h }
  fold_right f t (f acc h)
```

(continues on next page)

(continued from previous page)

```
  fold_right f (h :: t) acc
=   { evaluation }
  f h (fold_right f t acc)
```

Now, it might seem as though we are stuck: the left and right sides of the equality we want to show have failed to "meet in the middle." But we're actually in a similar situation to when we proved the correctness of `facti` earlier: there's something (applying `f` to `h` and another argument) that we want to push into the accumulator of that last line (so that we have `f acc h`).

Let's try proving that with its own lemma:

```
Lemma: forall lst acc x,
  f x (fold_right f lst acc) = fold_right f lst (f acc x)

Proof: by induction on lst.
P(lst) = forall acc x,
  f x (fold_right f lst acc) = fold_right f lst (f acc x)

Base case: lst = []
Show: forall acc x,
  f x (fold_right f [] acc) = fold_right f [] (f acc x)

  f x (fold_right f [] acc)
=   { evaluation }
  f x acc

  fold_right f [] (f acc x)
=   { evaluation }
  f acc x
=   { commutativity of f }
  f x acc

Inductive case: lst = h :: t
IH: forall acc x,
  f x (fold_right f t acc) = fold_right f t (f acc x)
Show: forall acc x,
  f x (fold_right f (h :: t) acc) = fold_right f (h :: t) (f acc x)

  f x (fold_right f (h :: t) acc)
=  { evaluation }
  f x (f h (fold_right f t acc))
=  { interchange lemma }
  f h (f x (fold_right f t acc))
=  { IH }
  f h (fold_right f t (f acc x))

  fold_right f (h :: t) (f acc x)
=   { evaluation }
  f h (fold_right f t (f acc x))

QED
```

Now that the lemma is completed, we can resume the proof of the theorem. We'll restart at the beginning of the inductive case:

```
Inductive case: lst = h :: t
IH: forall acc,
  fold_left f acc t = fold_right f t acc
Show: forall acc,
  fold_left f acc (h :: t) = fold_right f (h :: t) acc

  fold_left f acc (h :: t)
=   { evaluation }
  fold_left f (f acc h) t
=   { IH with acc := f acc h }
  fold_right f t (f acc h)

  fold_right f (h :: t) acc
=   { evaluation }
  f h (fold_right f t acc)
=   { lemma with x := h and lst := t }
  fold_right f t (f acc h)

QED
```

It took two inductions to prove the theorem, but we succeeded! Now we know that the behavior we observed with + wasn't a fluke: any commutative and associative operator causes `fold_left` and `fold_right` to get the same answer.

8.8.4 Induction on Trees

Lists and binary trees are similar when viewed as data types. Here are the definitions of both, aligned for comparison:

```
type 'a list = []   | ( :: ) of           'a * 'a list
type 'a tree = Leaf | Node   of 'a tree * 'a * 'a tree
```

Both have a constructor that represents "empty", and both have a constructor that combines a value of type `'a` together with another instance of the data type. The only real difference is that `( :: )` takes just *one* list, whereas `Node` takes *two* trees.

The induction principle for binary trees is therefore very similar to the induction principle for lists, except that with binary trees we get *two* inductive hypotheses, one for each subtree:

```
forall properties P,
  if P(Leaf),
  and if forall l v r, (P(l) and P(r)) implies P(Node (l, v, r)),
  then forall t, P(t)
```

An inductive proof for binary trees therefore has the following structure:

```
Proof: by induction on t.
P(t) = ...

Base case: t = Leaf
Show: P(Leaf)

Inductive case: t = Node (l, v, r)
IH1: P(l)
IH2: P(r)
Show: P(Node (l, v, r))
```

Let's try an example of this kind of proof. Here is a function that creates the mirror image of a tree, swapping its left and right subtrees at all levels:

```
let rec reflect = function
  | Leaf -> Leaf
  | Node (l, v, r) -> Node (reflect r, v, reflect l)
```

For example, these two trees are reflections of each other:

```
     1               1
   /   \           /   \
  2     3         3     2
 / \   / \       / \   / \
4   5 6   7     7   6 5   4
```

If you take the mirror image of a mirror image, you should get the original back. That means reflection is an *involution*, which is any function `f` such that `f (f x) = x`. Another example of an involution is multiplication by negative one on the integers.

Let's prove that `reflect` is an involution.

```
Claim: forall t, reflect (reflect t) = t

Proof: by induction on t.
P(t) = reflect (reflect t) = t

Base case: t = Leaf
Show: reflect (reflect Leaf) = Leaf

  reflect (reflect Leaf)
=   { evaluation }
  reflect Leaf
=   { evaluation }
  Leaf

Inductive case: t = Node (l, v, r)
IH1: reflect (reflect l) = l
IH2: reflect (reflect r) = r
Show: reflect (reflect (Node (l, v, r))) = Node (l, v, r)

  reflect (reflect (Node (l, v, r)))
=   { evaluation }
  reflect (Node (reflect r, v, reflect l))
=   { evaluation }
  Node (reflect (reflect l), v, reflect (reflect r))
=   { IH1 }
  Node (l, v, reflect (reflect r))
=   { IH2 }
  Node (l, v, r)

QED
```

Induction on trees is really no more difficult than induction on lists or natural numbers. Just keep track of the inductive hypotheses, using our stylized proof notation, and it isn't hard at all.

8.8.5 Induction Principles for All Variants

We've now seen induction principles for `nat`, `list`, and `tree`. Generalizing from what we've seen, each constructor of a variant either generates a base case for the inductive proof, or an inductive case. And, if a constructor itself carries values of that data type, each of those values generates an inductive hypothesis. For example:

- `Z`, `[]`, and `Leaf` all generated base cases.
- `S`, `::`, and `Node` all generated inductive cases.
- `S` and `::` each generated one IH, because each carries one value of the data type.
- `Node` generated two IHs, because it carries two values of the data type.

As an example of an induction principle for a more complicated type, let's consider a type that represents the syntax of a mathematical expression. You might recall from an earlier data structures course that trees can be used for that purpose.

Suppose we have the following `expr` type, which is a kind of tree, to represent expressions with integers, Booleans, unary operators, and binary operators:

```
type uop =
  | UMinus

type bop =
  | BPlus
  | BMinus
  | BLeq

type expr =
  | Int of int
  | Bool of bool
  | Unop of uop * expr
  | Binop of expr * bop * expr
```

For example, the expression `5 < 6` would be represented as `Binop (BLeq, Int 5, Int 6)`. We'll see more examples of this kind of representation later in the book when we study interpreters.

The induction principle for `expr` is:

```
forall properties P,
  if forall i, P(Int i)
  and forall b, P(Bool b)
  and forall u e, P(e) implies P(Unop (u, e))
  and forall b e1 e2, (P(e1) and P(e2)) implies P(Binop (e1, b, e2))
  then forall e, P(e)
```

There are two base cases, corresponding to the two constructors that don't carry an `expr`. There are two inductive cases, corresponding to the two constructors that do carry `expr`s. `Unop` gets one IH, whereas `Binop` gets two IHs, because of the number of `expr`s that each carries.

8.8.6 Induction and Recursion

Inductive proofs and recursive programs bear a striking similarity. In a sense, an inductive proof *is* a recursive program that shows how to construct evidence for a theorem involving an algebraic data type (ADT). The **structure** of an ADT determines the structure of proofs and programs:

- The **constructors** of an ADT are the organizational principle of both proofs and programs. In a proof, we have a base or inductive case for each constructor. In a program, we have a pattern-matching case for each constructor.
- The use of **recursive types** in an ADT determine where recursion occurs in both proofs and programs. By "recursive type", we mean the occurrence of the type in its own definition, such as the second `'a list` in `type 'a list = [] | ( :: ) 'a * 'a list`. Such occurrences lead to "smaller" values of a type occurring inside larger values. In a proof, we apply the inductive hypothesis upon reaching such a smaller value. In a program, we recurse on the smaller value.

8.9 Algebraic Specification

Next let's tackle a bigger challenge: proving the correctness of a data structure, such as a stack, queue, or set.

Correctness proofs always need specifications. In proving the correctness of iterative factorial, we used recursive factorial as a specification. By analogy, we could provide two implementations of a data structure—one simple, the other complex and efficient—and prove that the two are equivalent. That would require us to introduce ways to translate between the two implementations. For example, we could prove the correctness of a map implemented with an efficient balanced binary search tree relative to an implementation as an inefficient association list, by defining functions to convert trees to lists. Such an approach is certainly valid, but it doesn't lead to new ideas about verification for us to study.

Instead, we will pursue a different approach based on *equational specifications*, aka *algebraic specifications*. The idea with these is to

- define the types of the data structure operations, and
- to write a set of equations that define how the operations interact with one another.

The reason the word "algebra" shows up here is (in part) that this type-and-equation based approach is something we learned in high-school algebra. For example, here is a specification for some operators:

```
0 : int
1 : int
- : int -> int
+ : int -> int -> int
* : int -> int -> int

(a + b) + c = a + (b + c)
a + b = b + a
a + 0 = a
a + (-a) = 0
(a * b) * c = a * (b * c)
a * b = b * a
a * 1 = a
a * 0 = 0
a * (b + c) = a * b + a * c
```

The types of those operators, and the associated equations, are facts learned when studying algebra.

Our goal is now to write similar specifications for data structures, and use them to reason about the correctness of implementations.

8.9.1 Example: Stacks

Here are a few familiar operations on stacks along with their types.

```
module type Stack = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val peek : 'a t -> 'a
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a t
end
```

As usual, there is a design choice to be made with `peek` etc. about what to do with empty stacks. Here we have not used `option`, which suggests that `peek` will raise an exception on the empty stack. So we are cautiously relaxing our prohibition on exceptions.

In the past we've given these operations specifications in English, e.g.,

```
  (** [push x s] is the stack [s] with [x] pushed on the top. *)
  val push : 'a -> 'a stack -> 'a stack
```

But now, we'll write some equations to describe how the operations work:

```
1. is_empty empty = true
2. is_empty (push x s) = false
3. peek (push x s) = x
4. pop (push x s) = s
```

(Later we'll return to the question of *how* to design such equations.) The variables appearing in these equations are implicitly universally quantified. Here's how to read each equation:

1. `is_empty empty = true`. The empty stack is empty.
2. `is_empty (push x s) = false`. A stack that has just been pushed is non-empty.
3. `peek (push x s) = x`. Pushing then immediately peeking yields whatever value was pushed.
4. `pop (push x s) = s`. Pushing then immediately popping yields the original stack.

Just with these equations alone, we already can deduce a lot about how any sequence of stack operations must work. For example,

```
  peek (pop (push 1 (push 2 empty)))
=   { equation 4 }
  peek (push 2 empty)
=   { equation 3 }
  2
```

And `peek empty` doesn't equal any value according to the equations, since there is no equation of the form `peek empty = ...`. All that is true regardless of the stack implementation that is chosen: any correct implementation must cause the equations to hold.

Suppose we implemented stacks as lists, as follows:

```
module ListStack = struct
  type 'a t = 'a list
  let empty = []
```

(continues on next page)

(continued from previous page)

```
  let is_empty = function [] -> true | _ -> false
  let peek = List.hd
  let push = List.cons
  let pop = List.tl
end
```

Next we could *prove* that each equation holds of the implementation. All these proofs are quite easy by now, and proceed entirely by evaluation. For example, here's a proof of equation 3:

```
  peek (push x s)
=   { evaluation }
  peek (x :: s)
=   { evaluation }
  x
```

8.9.2 Example: Queues

Stacks were easy. How about queues? Here is the specification:

```
module type Queue = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val front : 'a t -> 'a
  val enq : 'a -> 'a t -> 'a t
  val deq : 'a t -> 'a t
end
```

```
1.  is_empty empty = true
2.  is_empty (enq x q) = false
3a. front (enq x q) = x            if is_empty q = true
3b. front (enq x q) = front q      if is_empty q = false
4a. deq (enq x q) = empty          if is_empty q = true
4b. deq (enq x q) = enq x (deq q)  if is_empty q = false
```

The types of the queue operations are actually identical to the types of the stack operations. Here they are, side-by-side for comparison:

```
module type Stack = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val peek : 'a t -> 'a
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a t
end
```

```
module type Queue = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val front : 'a t -> 'a
  val enq : 'a -> 'a t -> 'a t
  val deq : 'a t -> 'a t
end
```

Look at each line: though the operation may have a different name, its type is the same. Obviously, the types alone don't tell us enough about the operations. But the equations do! Here's how to read each equation:

1. The empty queue is empty.
2. Enqueueing makes a queue non-empty.

3. Enqueueing `x` on an empty queue makes `x` the front element. But if the queue isn't empty, enqueueing doesn't change the front element.
4. Enqueueing then dequeueing on an empty queue leaves the queue empty. But if the queue isn't empty, the enqueue and dequeue operations can be swapped.

For example,

```
  front (deq (enq 1 (enq 2 empty)))
=   { equation 4b }
  front (enq 1 (deq (enq 2 empty)))
=   { equation 4a }
  front (enq 1 empty)
=   { equation 3a }
  1
```

And `front empty` doesn't equal any value according to the equations.

Implementing a queue as a list results in an implementation that is easy to verify just with evaluation.

```
module ListQueue : Queue = struct
  type 'a t = 'a list
  let empty = []
  let is_empty q = q = []
  let front = List.hd
  let enq x q = q @ [x]
  let deq = List.tl
end
```

For example, 4a can be verified as follows:

```
  deq (enq x empty)
=   { evaluation of empty and enq}
  deq ([] @ [x])
=   { evaluation of @ }
  deq [x]
=   { evaluation of deq }
  []
=   { evaluation of empty }
  empty
```

And 4b, as follows:

```
  deq (enq x q)
=  { evaluation of enq and deq }
  List.tl (q @ [x])
=  { lemma, below, and q <> [] }
  (List.tl q) @ [x]

  enq x (deq q)
=  { evaluation }
  (List.tl q) @ [x]
```

Here is the lemma:

```
Lemma: if xs <> [], then List.tl (xs @ ys) = (List.tl xs) @ ys.
Proof: if xs <> [], then xs = h :: t for some h and t.
```

(continues on next page)

(continued from previous page)

```
  List.tl ((h :: t) @ ys)
=   { evaluation of @ }
  List.tl (h :: (t @ ys))
=   { evaluation of tl }
  t @ ys

  (List.tl (h :: t)) @ ys
=   { evaluation of tl }
  t @ ys

QED
```

Note how the precondition in 3b and 4b of `q` not being empty ensures that we never have to deal with an exception being raised in the equational proofs.

8.9.3 Example: Batched Queues

Recall that batched queues represent a queue with two lists:

```
module BatchedQueue = struct
  (* AF: [(o, i)] represents the queue [o @ (List.rev i)].
     RI: if [o] is empty then [i] is empty. *)
  type 'a t = 'a list * 'a list

  let empty = ([], [])

  let is_empty (o, i) = if o = [] then true else false

  let enq x (o, i) = if o = [] then ([x], []) else (o, x :: i)

  let front (o, _) = List.hd o

  let deq (o, i) =
    match List.tl o with
    | [] -> (List.rev i, [])
    | t -> (t, i)
end
```

This implementation is superficially different from the earlier implementation we gave, in that it uses pairs instead of records, and it raises the built-in exception `Failure` instead of a custom exception `Empty`.

Is this implementation correct? We need only verify the equations to find out.

First, a lemma:

```
Lemma:  if is_empty q = true, then q = empty.
Proof:  Since is_empty q = true, it must be that q = (f, b) and f = [].
By the RI, it must also be that b = [].  Thus q = ([], []) = empty.
QED
```

Verifying equation 1:

```
  is_empty empty
=   { eval empty }
  is_empty ([], [])
=   { eval is_empty }
  [] = []
=   { eval = }
  true
```

Verifying equation 2:

```
  is_empty (enq x q) = false
=   { eval enq }
  is_empty (if f = [] then [x], [] else f, x :: b)

case analysis: f = []

  is_empty (if f = [] then [x], [] else f, x :: b)
=   { eval if, f = [] }
  is_empty ([x], [])
=   { eval is_empty }
  [x] = []
=   { eval = }
  false

case analysis: f = h :: t

  is_empty (if f = [] then [x], [] else f, x :: b)
=   { eval if, f = h :: t }
  is_empty (h :: t, x :: b)
=   { eval is_empty }
  h :: t = []
=   { eval = }
  false
```

Verifying equation 3a:

```
  front (enq x q) = x
=   { emptiness lemma }
  front (enq x ([], []))
=   { eval enq }
  front ([x], [])
=   { eval front }
  x
```

Verifying equation 3b:

```
  front (enq x q)
=   { rewrite q as (h :: t, b), because q is not empty }
  front (enq x (h :: t, b))
=   { eval enq }
  front (h :: t, x :: b)
=   { eval front }
  h

  front q
=   { rewrite q as (h :: t, b), because q is not empty }
```

(continues on next page)

(continued from previous page)

```
  front (h :: t, b)
=   { eval front }
  h
```

Verifying equation 4a:

```
  deq (enq x q)
=   { emptiness lemma }
  deq (enq x ([], []))
=   { eval enq }
  deq ([x], [])
=   { eval deq }
  List.rev [], []
=   { eval rev }
  [], []
=   { eval empty }
  empty
```

Verifying equation 4b:

```
Show: deq (enq x q) = enq x (deq q)  assuming is_empty q = false.
Proof: Since is_empty q = false, q must be (h :: t, b).

Case analysis:  t = [], b = []

  deq (enq x q)
=   { rewriting q as ([h], []) }
  deq (enq x ([h], []))
=   { eval enq }
  deq ([h], [x])
=   { eval deq }
  List.rev [x], []
=   { eval rev }
  [x], []

  enq x (deq q)
=   { rewriting q as ([h], []) }
  enq x (deq ([h], []))
=   { eval deq }
  enq x (List.rev [], [])
=   { eval rev }
  enq x ([], [])
=   { eval enq }
  [x], []

Case analysis:  t = [], b = h' :: t'

  deq (enq x q)
=   { rewriting q as ([h], h' :: t') }
  deq (enq x ([h], h' :: t'))
=   { eval enq }
  deq ([h], x :: h' :: t')
=   { eval deq }
  List.rev (x :: h' :: t'), []
```

(continues on next page)

(continued from previous page)

```
  enq x (deq q)
=   { rewriting q as ([h], h' :: t') }
  enq x (deq ([h], h' :: t'))
=   { eval deq }
  enq x (List.rev (h' :: t'), [])
=   { eval enq }
  (List.rev (h' :: t'), [x])

STUCK
```

Wait, we just got stuck! `List.rev (x :: h' :: t'), []` and `(List.rev (h' :: t'), [x])` are not the same. But, abstractly, they do represent the same queue: `(List.rev t') @ [h'; x]`. We need to allow an additional equation for the representation type:

```
e = e'   if  AF(e) = AF(e')
```

Using that additional equation, we can continue:

```
  (List.rev (h' :: t'), [x])
=   { AF equation }
  List.rev (x :: h' :: t'), []

The AF equation holds because:

  List.rev (h' :: t') @ [x]
=   { eval rev }
  List.rev (h' :: t') @ List.rev [x]
=   { rev distributes over @, an exercise in the previous lecture }
  List.rev ([x] @ (h' :: t'))
=   { eval @ }
  List.rev (x :: h' :: t'))
=   { lst @ [] = lst, an exercise in the previous lecture }
  List.rev (x :: h' :: t') @ []

Case analysis:  t = h' :: t'

  deq (enq x q)
=   { rewriting q as (h :: h' :: t', b) }
  deq (enq x (h :: h' :: t', b))
=   { eval enq }
  deq (h :: h' :: t, x :: b)
=   { eval deq }
  h' :: t, x :: b

  enq x (deq q)
=   { rewriting q as (h :: h' :: t', b) }
  enq x (deq (h :: h' :: t', b))
=   { eval deq }
  enq x (h' :: t', b)
=   { eval enq }
  h' :: t', x :: b

QED
```

That concludes our verification of the batched queue. Note that we had to add the extra equation involving the abstraction function to get the proofs to go through:

```
e = e'   if  AF(e) = AF(e')
```

and that we made use of the RI during the proof. The AF and RI really are important!

8.9.4 Designing Algebraic Specifications

For both stacks and queues we provided some equations as the specification. Designing those equations is, in part, a matter of thinking hard about the data structure. But there's more to it than that.

Every value of the data structure is constructed with some operations. For a stack, those operations are `empty` and `push`. There might be some `pop` operations involved, but those can be eliminated. For example, `pop (push 1 (push 2 empty))` is really the same stack as `push 2 empty`. The latter is the *canonical form* of that stack: there are many other ways to construct it, but that is the simplest. Indeed, every possible stack value can be constructed just with `empty` and `push`. Similarly, every possible queue value can be constructed just with `empty` and `enq`: if there are `deq` operations involved, those can be eliminated.

Let's categorize the operations of a data structure as follows:

- **Generators** are those operations involved in creating a canonical form. They return a value of the data structure type. For example, `empty`, `push`, `enq`.
- **Manipulators** are operations that create a value of the data structure type, but are not needed to create canonical forms. For example, `pop`, `deq`.
- **Queries** do not return a value of the data structure type. For example, `is_empty`, `peek`, `front`.

Given such a categorization, we can design the equational specification of a data structure by applying non-generators to generators. For example: What does `is_empty` return on `empty`? on `push`? What does `front` return on `enq`? What does `deq` return on `enq`? etc.

So if there are `n` generators and `m` non-generators of a data structure, we would begin by trying to create `n*m` equations, one for each pair of a generator and non-generator. Each equation would show how to simplify an expression. In some cases we might need a couple equations, depending on the result of some comparison. For example, in the queue specification, we have the following equations:

1. `is_empty empty = true`: this is a non-generator `is_empty` applied to a generator `empty`. It reduces just to a Boolean value, which doesn't involve the data structure type (queues) at all.
2. `is_empty (enq x q) = false`: a non-generator `is_empty` applied to a generator `enq`. Again it reduces simply to a Boolean value.
3. There are two subcases.
   - `front (enq x q) = x`, if `is_empty q = true`. A non-generator `front` applied to a generator `enq`. It reduces to `x`, which is a smaller expression than the original `front (enq x q)`.
   - `front (enq x q) = front q`, `if is_empty q = false`. This similarly reduces to a smaller expression.
4. Again, there are two subcases.
   - `deq (enq x q) = empty`, if `is_empty q = true`. This simplifies the original expression by reducing it to `empty`.
   - `deq (enq x q) = enq x (deq q)`, if `is_empty q = false`. This simplifies the original expression by reducing it to an generator applied to a smaller argument, `deq q` instead of `deq (enq x q)`.

We don't usually design equations involving pairs of non-generators. Sometimes pairs of generators are needed, though, as we will see in the next example.

Example: Sets. Here is a small interface for sets:

```
module type Set = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val mem : 'a -> 'a t -> bool
  val remove : 'a -> 'a t -> 'a t
end
```

The generators are `empty` and `add`. The only manipulator is `remove`. Finally, `is_empty` and `mem` are queries. So we should expect at least 2 * 3 = 6 equations, one for each pair of generator and non-generator. Here is an equational specification:

```
1.  is_empty empty = true
2.  is_empty (add x s) = false
3.  mem x empty = false
4a. mem y (add x s) = true                    if x = y
4b. mem y (add x s) = mem y s                 if x <> y
5.  remove x empty = empty
6a. remove y (add x s) = remove y s           if x = y
6b. remove y (add x s) = add x (remove y s)   if x <> y
```

Consider, though, these two sets:

- `add 0 (add 1 empty)`
- `add 1 (add 0 empty)`

They both intuitively represent the set {0,1}. Yet, we cannot prove that those two sets are equal using the above specification. We are missing an equation involving two generators:

```
7.  add x (add y s) = add y (add x s)
```

8.10 Summary

Documentation and testing are crucial to establishing the truth of what a correct program does. Documentation communicates to other humans the intent of the programmer. Testing communicates evidence about the success of the programmer.

Good documentation provides several pieces: a summary, preconditions, postconditions (including errors), and examples. Documentation is written for two different audiences, clients and maintainers. The latter needs to know about abstraction functions and representation invariants.

Testing methodologies include black-box, glass-box, and randomized tests. These are complementary, not orthogonal, approaches to developing correct code.

Formal methods is an important link between mathematics and computer science. We can use techniques from discrete math, such as induction, to prove the correctness of functional programs. Equational reasoning makes the proofs relatively pleasant.

Proving the correctness of imperative programs can be more challenging, because of the need to reason about mutable state. That can break equational reasoning. Instead, *Hoare logic*, named for Tony Hoare, is a common formal method for imperative programs. Dijkstra's *weakest precondition* calculus is another.

8.10.1 Terms and Concepts

- abstract value
- abstraction by specification
- abstraction function
- algebraic specification
- asserting
- associative
- base case
- black box
- boundary case
- bug
- canonical form
- client
- code inspection
- code review
- code walkthrough
- comments
- commutative
- commutative diagram
- concrete value
- conditional compilation
- consumer
- correctness
- data abstraction
- debugging by scientific method
- defensive programming
- equation
- equational reasoning
- example clause
- extensionality
- failure
- fault
- formal methods
- formal methods
- generator
- glass box

- identity
- implementer
- induction
- induction hypothesis
- induction principle
- inductive case
- inputs for classes of output
- inputs that satisfy precondition
- inputs that trigger exceptions
- iterative
- locality
- manipulator
- many to one
- minimal test case
- modifiability
- natural numbers
- pair programming
- partial
- partial correctness
- partial function
- path coverage
- paths through implementation
- paths through specification
- postcondition
- postcondition
- precondition
- precondition
- producer
- query
- raises clause
- randomized testing
- regression testing
- rely
- rep ok
- representation invariant
- representation type

- representative inputs
- requires clause
- returns clause
- satisfaction
- social methods
- specification
- specification
- testing
- total correctness
- total function
- typical input
- validation
- verification
- well-founded

8.10.2 Further Reading

- *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, chapters 3, 5, and 9, by Barbara Liskov with John Guttag.
- *The Functional Approach to Programming*, section 3.4. Guy Cousineau and Michel Mauny. Cambridge, 1998.
- *ML for the Working Programmer*, second edition, chapter 6. L.C. Paulson. Cambridge, 1996.
- *Thinking Functionally with Haskell*, chapter 6. Richard Bird. Cambridge, 2015.
- *Software Foundations*, volume 1, chapters Basic, Induction, Lists, Poly. Benjamin Pierce et al. https://softwarefoundations.cis.upenn.edu/
- "Algebraic Specifications", Robert McCloskey, https://www.cs.scranton.edu/~mccloske/courses/se507/alg_specs_lec.html.
- *Software Engineering: Theory and Practice*, third edition, section 4.5. Shari Lawrence Pfleeger and Joanne M. Atlee. Prentice Hall, 2006.
- "Algebraic Semantics", chapter 12 of *Formal Syntax and Semantics of Programming Languages*, Kenneth Slonneger and Barry L. Kurtz, Addison-Wesley, 1995.
- "Algebraic Semantics", Muffy Thomas. Chapter 6 in *Programming Language Syntax and Semantics*, David Watt, Prentice Hall, 1991.
- *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. H. Ehrig and B. Mahr. Springer-Verlag, 1985.

8.10.3 Acknowledgments

Our treatment of formal methods is inspired by and indebted to course materials for Princeton COS 326 by David Walker et al.

Our example algebraic specifications are based on McCloskey's. The terminology of "generator", "manipulator", and "query" is based on Pfleeger and Atlee.

Many of our exercises on formal methods are inspired by *Software Foundations*, volume 1.

8.11 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Exercise: spec game [★★★]

Pair up with another programmer and play the specification game with them. Take turns being the specifier and the devious programmer. Here are some suggested functions you could use:

- `num_vowels : string -> int`
- `is_sorted : 'a list -> bool`
- `sort : 'a list -> 'a list`
- `max : 'a list -> 'a`
- `is_prime : int -> bool`
- `is_palindrome : string -> bool`
- `second_largest : int list -> int`
- `depth : 'a tree -> int`

Exercise: poly spec [★★★]

Let's create a data abstraction for single-variable integer polynomials of the form

$$c_n x^n + ... + c_1 x + c_0.$$

Let's assume that the polynomials are *dense*, meaning that they contain very few coefficients that are zero. Here is an incomplete interface for polynomials:

```
(** [Poly] represents immutable polynomials with integer coefficients. *)
module type Poly = sig
  (** [t] is the type of polynomials *)
  type t

  (** [eval x p] is [p] evaluated at [x]. Example: if [p] represents
      $3x^3 + x^2 + x$, then [eval 10 p] is [3110]. *)
  val eval : int -> t -> int
end
```

Finish the design of `Poly` by adding more operations to the interface. Consider what operations would be useful to a client of the abstraction:

- How would they create polynomials?
- How would they combine polynomials to get new polynomials?
- How would they query a polynomial to find out what it represents?

Write specification comments for the operations that you invent. Keep in mind the spec game as you write them: could a devious programmer subvert your intentions?

Exercise: poly impl [★★★]

Implement your specification of `Poly`. As part of your implementation, you will need to choose a representation type `t`. *Hint: recalling that our polynomials are dense might guide you in choosing a representation type that makes for an easier implementation.*

Exercise: interval arithmetic [★★★★]

Specify and implement a data abstraction for interval arithmetic. Be sure to include the abstraction function, representation invariant, and `rep_ok`. Also implement a `to_string` function and a `format` that can be installed in the top level with `#install_printer`.

Exercise: function maps [★★★★]

Implement a map (aka dictionary) data structure with abstract type `('k, 'v) t`. As the representation type, use `'k -> 'v`. That is, a map is represented as an OCaml function from keys to values. Document the AF. You do not need an RI. Your solution will make heavy use of higher-order functions. Provide at least these values and operations: `empty`, `mem`, `find`, `add`, `remove`.

Exercise: set black box [★★★]

Go back to the implementation of sets with lists in the previous chapter. Based on the specification comments of `Set`, write an OUnit test suite for `ListSet` that does black-box testing of all its operations.

Exercise: set glass box [★★★]

Achieve as close to 100% code coverage with Bisect as you can for `ListSet` and `UniqListSet`.

Exercise: random lists [★★★]

Use `QCheck.Gen.generate1` to generate a list whose length is between 5 and 10, and whose elements are integers between 0 and 100. Then use `QCheck.Gen.generate` to generate a 3-element list, each element of which is a list of the kind you just created with `generate1`.

Then use `QCheck.make` to create an arbitrary that represents a list whose length is between 5 and 10, and whose elements are integers between 0 and 100. The type of your arbitrary should be `int list QCheck.arbitrary`.

Finally create and run a QCheck test that checks whether at least one element of an arbitrary list (of 5 to 10 elements, each between 0 and 100) is even. You'll need to "upgrade" the `is_even` property to work on a list of integers rather than a single integer.

Each time you run the test, recall that it will generate 100 lists and check the property of them. If you run the test many times, you'll likely see some successes and some failures.

Exercise: qcheck odd divisor [★★★]

Here is a buggy function:

```
(** [odd_divisor x] is an odd divisor of [x].
    Requires: [x >= 0]. *)
let odd_divisor x =
  if x < 3 then 1 else
    let rec search y =
      if y >= x then y  (* exceeded upper bound *)
      else if x mod y = 0 then y  (* found a divisor! *)
      else search (y + 2) (* skip evens *)
    in search 3
```

Write a QCheck test to determine whether the output of that function (on a positive integer, per its precondition; *hint: there is an arbitrary that generates positive integers*) is both odd and is a divisor of the input. You will discover that there is a bug in the function. What is the smallest integer that triggers that bug?

Exercise: qcheck avg [★★★★]

Here is a buggy function:

```
(** [avg [x1; ...; xn]] is [(x1 + ... + xn) / n].
     Requires: the input list is not empty. *)
let avg lst =
  let rec loop (s, n) = function
    | [] -> (s, n)
    | [ h ] -> (s + h, n + 1)
    | h1 :: h2 :: t -> if h1 = h2 then loop (s + h1, n + 1) t
      else loop (s + h1 + h2, n + 2) t
  in
  let (s, n) = loop (0, 0) lst
  in float_of_int s /. float_of_int n
```

Write a QCheck test that detects the bug. For the property that you check, construct your own *reference implementation* of average—that is, a less optimized version of `avg` that is obviously correct.

Exercise: exp [★★]

Prove that `exp x (m + n) = exp x m * exp x n`, where

```
let rec exp x n =
  if n = 0 then 1 else x * exp x (n - 1)
```

Proceed by induction on `m`.

Exercise: fibi [★★★]

Prove that `forall n >= 1, fib n = fibi n (0, 1)`, where

```
let rec fib n =
  if n = 1 then 1
  else if n = 2 then 1
  else fib (n - 2) + fib (n - 1)

let rec fibi n (prev, curr) =
  if n = 1 then curr
  else fibi (n - 1) (curr, prev + curr)
```

Proceed by induction on `n`, rather than trying to apply the theorem about converting recursion into iteration.

Exercise: expsq [★★★]

Prove that `expsq x n = exp x n`, where

```
let rec expsq x n =
  if n = 0 then 1
  else if n = 1 then x
  else (if n mod 2 = 0 then 1 else x) * expsq (x * x) (n / 2)
```

Proceed by *strong induction* on `n`. Function `expsq` implements *exponentiation by repeated squaring*, which results in more efficient computation than `exp`.

Exercise: mult [★★]

Prove that `forall n, mult n Z = Z` by induction on `n`, where:

```
let rec mult a b =
  match a with
  | Z -> Z
  | S k -> plus b (mult k b)
```

Exercise: append nil [★★]

Prove that `forall lst, lst @ [] = lst` by induction on `lst`.

Exercise: rev dist append [★★★]

Prove that reverse distributes over append, i.e., that `forall lst1 lst2, rev (lst1 @ lst2) = rev lst2 @ rev lst1`, where:

```
let rec rev = function
  | [] -> []
  | h :: t -> rev t @ [h]
```

(That is, of course, an inefficient implementation of `rev`.) You will need to choose which list to induct over. You will need the previous exercise as a lemma, as well as the associativity of `append`, which was proved in the notes above.

Exercise: rev involutive [★★★]

Prove that reverse is an involution, i.e., that `forall lst, rev (rev lst) = lst`. Proceed by induction on `lst`. You will need the previous exercise as a lemma.

Exercise: reflect size [★★★]

Prove that `forall t, size (reflect t) = size t` by induction on `t`, where:

```
let rec size = function
  | Leaf -> 0
  | Node (l, v, r) -> 1 + size l + size r
```

Exercise: fold theorem 2 [★★★★]

We proved that `fold_left` and `fold_right` yield the same results if their function argument is associative and commutative. But that doesn't explain why these two implementations of `concat` yield the same results, because `( ^ )` is not commutative:

```
let concat_l lst = List.fold_left ( ^ ) "" lst
let concat_r lst = List.fold_right ( ^ ) lst ""
```

Formulate and prove a new theorem about when `fold_left` and `fold_right` yield the same results, under the relaxed assumption that their function argument is associative but not necessarily commutative. *Hint: make a new assumption about the initial value of the accumulator.*

Exercise: propositions [★★★★]

In propositional logic, we have atomic propositions, negation, conjunction, disjunction, and implication. For example, `raining /\ snowing /\ cold` is a proposition stating that it is simultaneously raining and snowing and cold (a weather condition known as *Ithacating*).

Define an OCaml type to represent propositions. Then state the induction principle for that type.

Exercise: list spec [★★★]

Design an OCaml interface for lists that has `nil`, `cons`, `append`, and `length` operations. Design the equational specification. Hint: the equations will look strikingly like the OCaml implementations of `@` and `List.length`.

Exercise: bag spec [★★★★]

A *bag* or *multiset* is like a blend of a list and a set: like a set, order does not matter; like a list, elements may occur more than once. The number of times an element occurs is its *multiplicity*. An element that does not occur in the bag has multiplicity 0. Here is an OCaml signature for bags:

```
module type Bag = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val insert : 'a -> 'a t -> 'a t
  val mult : 'a -> 'a t -> int
  val remove : 'a -> 'a t -> 'a t
end
```

Categorize the operations in the `Bag` interface as generators, manipulators, or queries. Then design an equational specification for bags. For the `remove` operation, your specification should cause at most one occurrence of an element to be removed. That is, the multiplicity of that value should decrease by at most one.

CHAPTER

NINE

MUTABILITY

OCaml is not a *pure* language: it does admit side effects. We have seen that already with I/O, especially printing. But up till now we have limited ourself to the subset of the language that is *immutable*: values could not change.

Mutability is neither good nor bad. It enables new functionality that we couldn't implement (at least not easily) before, and it enables us to create certain data structures that are asymptotically more efficient than their purely functional analogues. But mutability does make code more difficult to reason about, hence it is a source of many faults in code. One reason for that might be that humans are not good at thinking about change. With immutable values, we're guaranteed that any fact we might establish about them can never change. But with mutable values, that's no longer true. "Change is hard," as they say.

In this short chapter we'll cover the few mutable features of OCaml we've omitted so far, and we'll use them for some simple data structures. The real win, though, will come in the next chapter, where we put the features to more complicated uses.

9.1 Refs

A *ref* is like a pointer or reference in an imperative language. It is a location in memory whose contents may change. Refs are also called *ref cells*, the idea being that there's a cell in memory that can change.

Here's an example of creating a ref, getting the value from inside it, changing its contents, and observing the changed contents:

```
let x = ref 0;;
```

```
!x;;
```

```
x := 1;;
```

```
!x;;
```

The first phrase, `let x = ref 0`, creates a reference using the `ref` keyword. That's a location in memory whose contents are initialized to `0`. Think of the location itself as being an address—for example, 0x3110bae0—even though there's no way to write down such an address in an OCaml program. The keyword `ref` is what causes the memory location to be allocated and initialized.

The first part of the response from OCaml, `val x : int ref`, indicates that `x` is a variable whose type is `int ref`. We have a new type constructor here. Much like `list` and `option` are type constructors, so is `ref`. A `t ref`, for any type `t`, is a reference to a memory location that is guaranteed to contain a value of type `t`. As usual we should read a type from right to left: `t ref` means a reference to a `t`. The second part of the response shows us the contents of the memory location. Indeed, the contents have been initialized to `0`.

The second phrase, `!x`, dereferences `x` and returns the contents of the memory location. Note that `!` is the dereference operator in OCaml, not Boolean negation.

The third phrase, `x := 1`, is an assignment. It mutates the contents `x` to be `1`. Note that `x` itself still points to the same location (i.e., address) in memory. Memory is mutable; variable bindings are not. What changes is the contents. The response from OCaml is simply `()`, meaning that the assignment took place—much like printing functions return `()` to indicate that the printing did happen.

The fourth phrase, `!x` again dereferences `x` to demonstrate that the contents of the memory location did indeed change.

9.1.1 Aliasing

Now that we have refs, we have *aliasing*: two refs could point to the same memory location, hence updating through one causes the other to also be updated. For example,

```
let x = ref 42;;
let y = ref 42;;
let z = x;;
x := 43;;
let w = !y + !z;;
```

The result of executing that code is that `w` is bound to `85`, because `let z = x` causes `z` and `x` to become aliases, hence updating `x` to be `43` also causes `z` to be `43`.

9.1.2 Syntax and Semantics

The semantics of refs is based on *locations* in memory. Locations are values that can be passed to and returned from functions. But unlike other values (e.g., integers, variants), there is no way to directly write a location in an OCaml program. That's different than languages like C, in which programmers can directly write memory addresses and do arithmetic on pointers. C programmers want that kind of low-level access to do things like interface with hardware and build operating systems. Higher-level programmers are willing to forego it to get *memory safety*. That's a hard term to define, but according to Hicks 2014 it intuitively means that

- pointers are only created in a safe way that defines their legal memory region,
- pointers can only be dereferenced if they point to their allotted memory region,
- that region is (still) defined.

Syntax.

- Ref creation: `ref e`
- Ref assignment: `e1 := e2`
- Dereference: `!e`

Dynamic semantics.

- To evaluate `ref e`,
  - Evaluate `e` to a value `v`
  - Allocate a new location `loc` in memory to hold `v`
  - Store `v` in `loc`
  - Return `loc`
- To evaluate `e1 := e2`,

  - Evaluate `e2` to a value `v`, and `e1` to a location `loc`.
  - Store `v` in `loc`.
  - Return `()`, i.e., unit.
- To evaluate `!e`,
  - Evaluate `e` to a location `loc`.
  - Return the contents of `loc`.

Static semantics.

We have a new type constructor, `ref`, such that `t ref` is a type for any type `t`. Note that the `ref` keyword is used in two ways: as a type constructor, and as an expression that constructs refs.

- `ref e : t ref` if `e : t`.
- `e1 := e2 : unit` if `e1 : t ref` and `e2 : t`.
- `!e : t` if `e : t ref`.

9.1.3 Sequencing of Effects

The semicolon operator is used to sequence effects, such as mutating refs. We've seen semicolon occur previously with printing. Now that we're studying mutability, it's time to treat it formally.

- **Syntax:** `e1; e2`
- **Dynamic semantics:** To evaluate `e1; e2`,
  - First evaluate `e1` to a value `v1`.
  - Then evaluate `e2` to a value `v2`.
  - Return `v2`. (`v1` is not used at all.)
  - If there are multiple expressions in a sequence, e.g., `e1; e2; ...; en`, then evaluate each one in order from left to right, returning only `vn`. Another way to think about this is that semicolon is right associative—for example `e1; e2; e3` is the same as `e1; (e2; e3))`.
- **Static semantics:** `e1; e2 : t` if `e1 : unit` and `e2 : t`. Similarly, `e1; e2; ...; en : t` if `e1 : unit`, `e2 : unit`, ... (i.e., all expressions except `en` have type `unit`), and `en : t`.

The typing rule for semicolon is designed to prevent programmer mistakes. For example, a programmer who writes `2+3; 7` probably didn't mean to: there's no reason to evaluate `2+3` then throw away the result and instead return `7`. The compiler will give you a warning if you violate this particular typing rule.

To get rid of the warning (if you're sure that's what you need to do), there's a function `ignore : 'a -> unit` in the standard library. Using it, `ignore(2+3); 7` will compile without a warning. Of course, you could code up `ignore` yourself: `let ignore _ = ()`.

9.1.4 Example: Mutable Counter

Here is code that implements a *counter*. Every time `next_val` is called, it returns one more than the previous time.

```
let counter = ref 0

let next_val =
  fun () ->
    counter := !counter + 1;
    !counter
```

```
next_val ()
```

```
next_val ()
```

```
next_val ()
```

In the implementation of `next_val`, there are two expressions separated by semi-colon. The first expression, `counter := !counter + 1`, is an assignment that increments `counter` by 1. The second expression, `!counter`, returns the newly incremented contents of `counter`.

The `next_val` function is unusual in that every time we call it, it returns a different value. That's quite different than any of the functions we've implemented ourselves so far, which have always been *deterministic*: for a given input, they always produced the same output. On the other hand, we've seen some library functions that are *nondeterministic*, for example, functions in the `Random` module, and `Stdlib.read_line`. It's no coincidence that those happen to be implemented using mutable features.

We could improve our counter in a couple ways. First, there is a library function `incr : int ref -> unit` that increments an `int ref` by 1. Thus it is like the `++` operator that is familiar from many languages in the C family. Using it, we could write `incr counter` instead of `counter := !counter + 1`. (There's also a `decr` function that decrements by 1.)

Second, the way we coded the counter currently exposes the `counter` variable to the outside world. Maybe we're prefer to hide it so that clients of `next_val` can't directly change it. We could do so by nesting `counter` inside the scope of `next_val`:

```
let next_val =
  let counter = ref 0 in
  fun () ->
    incr counter;
    !counter
```

Now `counter` is in scope inside of `next_val`, but not accessible outside that scope.

When we gave the dynamic semantics of let expressions before, we talked about substitution. One way to think about the definition of `next_val` is as follows.

- First, the expression `ref 0` is evaluated. That returns a location `loc`, which is an address in memory. The contents of that address are initialized to `0`.
- Second, everywhere in the body of the let expression that `counter` occurs, we substitute for it that location. So we get:

  ```
  fun () -> incr loc; !loc
  ```

- Third, that anonymous function is bound to `next_val`.

So any time `next_val` is called, it increments and returns the contents of that one memory location `loc`.

Now imagine that we instead had written the following (broken) code:

```
let next_val_broken = fun () ->
  let counter = ref 0 in
  incr counter;
  !counter
```

It's only a little different: the binding of `counter` occurs after the `fun () ->` instead of before. But it makes a huge difference:

```
next_val_broken ();;
next_val_broken ();;
next_val_broken ();;
```

Every time we call `next_val_broken`, it returns `1`: we no longer have a counter. What's going wrong here?

The problem is that every time `next_val_broken` is called, the first thing it does is to evaluate `ref 0` to a new location that is initialized to `0`. That location is then incremented to `1`, and `1` is returned. *Every* call to `next_val_broken` is thus allocating a new ref cell, whereas `next_val` allocates just *one* new ref cell.

9.1.5 Example: Pointers

In languages like C, pointers combine two features: they can be null, and they can be changed. (Java has a similar construct with object references, but that term is confusing in our OCaml context since "reference" currently means a ref cell. So we'll stick with the word "pointer".) Let's code up pointers using OCaml ref cells.

```
type 'a pointer = 'a ref option
```

As usual, read that type right to left. The `option` part of it encodes the fact that a pointer might be null. We're using `None` to represent that possibility.

```
let null : 'a pointer = None
```

The `ref` part of the type encodes the fact that the contents are mutable. We can create a helper function to allocate and initialize the contents of a new pointer:

```
let malloc (x : 'a) : 'a pointer = Some (ref x)
```

Now we could create a pointer to any value we like:

```
let p = malloc 42
```

Dereferencing a pointer is the `*` prefix operator in C. It returns the contents of the pointer, and raises an exception if the pointer is null:

```
exception Segfault

let deref (ptr : 'a pointer) : 'a =
  match ptr with None -> raise Segfault | Some r -> !r
```

```
deref p
```

```
deref null
```

We could even introduce our own OCaml operator for dereference. We have to put ~ in front of it to make it parse as a prefix operator, though.

```
let ( ~* ) = deref;;
~*p
```

In C, an assignment through a pointer is written `*p = x`. That changes the memory to which `p` points, making it contain `x`. We can code up that operator as follows:

```
let assign (ptr : 'a pointer) (x : 'a) : unit =
  match ptr with None -> raise Segfault | Some r -> r := x
```

```
assign p 2;
deref p
```

```
assign null 0
```

Again, we could introduce our own OCaml operator for that, though it's hard to pick a good symbol involving * and = that won't be misunderstood as involving multiplication:

```
let ( =* ) = assign;;
p =* 3;;
~*p
```

The one thing we can't do is treat a pointer as an integer. C allows that, including taking the address of a variable, which enables *pointer arithmetic*. That's great for efficiency, but also terrible because it leads to all kinds of program errors and security vulnerabilities.

Evil Secret

Okay that wasn't actually true what we just said, but this is dangerous knowledge that you really shouldn't even read. There is an undocumented function `Obj.magic` that we could use to get a memory address of a ref:

```
let address (ptr : 'a pointer) : int =
  match ptr with None -> 0 | Some r -> Obj.magic r

let ( ~& ) = address
```

But you have to promise to never, ever use that function yourself, because it completely circumvents the safety of the OCaml type system. All bets are off if you do.

None of this pointer encoding is part of the OCaml standard library, because you don't need it. You can always use refs and options yourself as you need to. Coding as we just did above is not particularly idiomatic. The reason we did it was to illustrate the relationship between OCaml refs and C pointers (equivalently, Java references).

9.1.6 Example: Recursion Without Rec

Here's a neat trick that's possible with refs: we can build recursive functions without ever using the keyword `rec`. Suppose we want to define a recursive function such as `fact`, which we would normally write as follows:

```
let rec fact_rec n = if n = 0 then 1 else n * fact_rec (n - 1)
```

We want to define that function without using `rec`. We can begin by defining a ref to an obviously incorrect version of the function:

```
let fact0 = ref (fun x -> x + 0)
```

The way in which `fact0` is incorrect is actually irrelevant. We just need it to have the right type. We could just as well have used `fun x -> x` instead of `fun x -> x + 0`.

At this point, `fact0` clearly doesn't compute the factorial function. For example, 5! ought to be 120, but that's not what `fact0` computes:

```
!fact0 5
```

Next, we write `fact` as usual, but without `rec`. At the place where we need to make the recursive call, we instead invoke the function stored inside `fact0`:

```
let fact n = if n = 0 then 1 else n * !fact0 (n - 1)
```

Now `fact` does actually get the right answer for `0`, but not for `5`:

```
fact 0;;
fact 5;;
```

The reason it's not right for `5` is that the recursive call isn't actually to the right function. We want the recursive call to go to `fact`, not to `fact0`. **So here's the trick:** we mutate `fact0` to point to `fact`:

```
fact0 := fact
```

Now when `fact` makes its recursive call and dereferences `fact0`, it gets back itself! That makes the computation correct:

```
fact 5
```

Abstracting a little, here's what we did. We started with a function that is recursive:

```
let rec f x = ... f y ...
```

We rewrote it as follows:

```
let f0 = ref (fun x -> x)

let f x = ... !f0 y ...

f0 := f
```

Now `f` will compute the same result as it did in the version where we defined it with `rec`.

What's happening here is sometimes called "tying the recursive knot": we update the reference to `f0` to point to `f`, such that when `f` dereferences `f0`, it gets itself back. The initial function to which we made `f0` point (in this case the identity function) doesn't really matter; it's just there as a placeholder until we tie the knot.

9.1.7 Weak Type Variables

Perhaps you have already tried using the identity function to define `fact0`, as we mentioned above. If so, you will have encountered this rather puzzling output:

```
let fact0 = ref (fun x -> x)
```

What is this strange type for the identity function, `'_weak1 -> '_weak1`? Why isn't it the usual `'a -> 'a`?

The answer has to do with a particularly tricky interaction between polymorphism and mutability. In a later chapter on interpreters, we'll learn how type inference works, and at that point we'll be able to explain the problem in detail. In short, allowing the type `'a -> 'a` for that ref would lead the possibility of programs that crash at run time because of type errors.

For now, think about it this way: although the *value* stored in a ref cell is permitted to change, the *type* of that value is not. And if OCaml gave `ref (fun x -> x)` the type `('a -> 'a) ref`, then that cell could first store `fun x -> x + 1 : int -> int` but later store `fun x -> s ^ "!" : string -> string`. That would be the kind of change in type that is not allowed.

So OCaml uses *weak type variables* to stand for unknown but not polymorphic types. These variables always start with `_weak`. Essentially, type inference for these is just not finished yet. Once you give OCaml enough information, it will finish type inference and replace the weak type variable with the actual type:

```
!fact0
```

```
!fact0 1
```

```
!fact0
```

After the application of `!fact0` to `1`, OCaml now knows that the function is meant to have type `int -> int`. So from then on, that's the only type at which it can be used. It can't, for example, be applied to a string.

```
!fact0 "camel"
```

If you would like to learn more about weak type variables right now, take a look at Section 2 of *Relaxing the value restriction* by Jacques Garrigue, or this section of the OCaml manual.

9.1.8 Physical Equality

OCaml has two equality operators, physical equality and structural equality. The documentation of `Stdlib.(==)` explains physical equality:

> `e1 == e2` tests for physical equality of `e1` and `e2`. On mutable types such as references, arrays, byte sequences, records with mutable fields and objects with mutable instance variables, `e1 == e2` is `true` if and only if physical modification of `e1` also affects `e2`. On non-mutable types, the behavior of `( == )` is implementation-dependent; however, it is guaranteed that `e1 == e2` implies `compare e1 e2 = 0`.

One interpretation could be that == should be used only when comparing refs (and other mutable data types) to see whether they point to the same location in memory. Otherwise, don't use ==.

Structural equality is also explained in the documentation of `Stdlib.(=)`:

> `e1 = e2` tests for structural equality of `e1` and `e2`. Mutable structures (e.g. references and arrays) are equal if and only if their current contents are structurally equal, even if the two mutable objects are not the same physical object. Equality between functional values raises `Invalid_argument`. Equality between cyclic data structures may not terminate.

Structural equality is usually what you want to test. For refs, it checks whether the contents of the memory location are equal, regardless of whether they are the same location.

The negation of physical equality is `!=`, and the negation of structural equality is `<>`. This can be hard to remember.

Here are some examples involving equality and refs to illustrate the difference between structural equality (=) and physical equality (==):

```
let r1 = ref 42
let r2 = ref 42
```

A ref is physically equal to itself, but not to another ref that is a different location in memory:

```
r1 == r1
```

```
r1 == r2
```

```
r1 != r2
```

Two refs that are at different locations in memory but store structurally equal values are themselves structurally equal:

```
r1 = r1
```

```
r1 = r2
```

```
r1 <> r2
```

Two refs that store structurally unequal values are themselves structurally unequal:

```
ref 42 <> ref 43
```

9.1.9 Example: Singly-linked Lists

OCaml's built-in singly-linked lists are functional, not imperative. But we can code up imperative singly-linked lists, of course, with refs. (We could also use the pointers we invented above, but that only makes the code more complicated.)

We start by defining a type `'a node` for nodes of a list that contains values of type `'a`. The `next` field of a node is itself another list.

```
(** An ['a node] is a node of a mutable singly-linked list. It contains a value
    of type ['a] and a link to the [next] node. *)
type 'a node = { next : 'a mlist; value : 'a }

(** An ['a mlist] is a mutable singly-linked list with elements of type ['a].
    The [option] represents the possibility that the list is empty.
```

(continues on next page)

(continued from previous page)

```
    RI: The list does not contain any cycles. *)
and 'a mlist = 'a node option ref
```

To create an empty list, we simply return a ref to `None`:

```
(** [empty ()] is an empty singly-linked list. *)
let empty () : 'a mlist = ref None
```

Note the type of `empty`: instead of being a value, it is now a function. This is typical of functions that create mutable data structures. At the end of this section, we'll return to why `empty` *has* to be a function.

Inserting a new first element just requires creating a new node, linking from it to the original list, and mutating the list:

```
(** [insert_first lst n] mutates mlist [lst] by inserting value [v] as the
    first value in the list. *)
let insert_first (lst : 'a mlist) (v : 'a) : unit =
  lst := Some { next = ref !lst; value = v }
```

Again, note the type of `insert_first`. Rather than returning an `'a mlist`, it returns `unit`. This again is typical of functions that modify mutable data structures.

In both `empty` and `insert_first`, the use of `unit` makes the functions more like their equivalents in an imperative language. The constructor for an empty list in Java, for example, might not take any arguments (which is equivalent to taking `unit`). And the `insert_first` operation for a Java linked list might return `void`, which is equivalent to returning `unit`.

Finally, here's a conversion function from our new mutable lists to OCaml's built-in lists:

```
(** [to_list lst] is an OCaml list containing the same values as [lst]
    in the same order. Not tail recursive. *)
let rec to_list (lst : 'a mlist) : 'a list =
  match !lst with None -> [] | Some { next; value } -> value :: to_list next
```

Now we can see mutability in action:

```
let lst0 = empty ();;
let lst1 = lst0;;
insert_first lst0 1;;
to_list lst1;;
```

The change to `lst0` mutates `lst1`, because they are aliases.

The type of `empty`. Returning to `empty`, why must it be a function? It might seem as though we could define it more simply as follows:

```
let empty = ref None
```

But now there is only ever *one* ref that gets created, hence there is only one list ever in existence:

```
let lst2 = empty;;
let lst3 = empty;;
insert_first lst2 2;;
insert_first lst3 3;;
to_list lst2;;
to_list lst3;;
```

Note how the mutations affect both lists, because they are both aliases for the same ref.

By correctly making `empty` a function, we guarantee that a new ref is returned every time an empty list is created.

```
let empty () = ref None
```

It really doesn't matter what argument that function takes, since it will never use it. We could define it as any of these in principle:

```
let empty _ = ref None
let empty (b : bool) = ref None
let empty (n : int) = ref None
(* etc. *)
```

But the reason we prefer `unit` as the argument type is to indicate to the client that the argument value is not going to be used. After all, there's nothing interesting that the function can do with the unit value. Another way to think about that would be that a function whose input type is `unit` is like a function or method in an imperative language that takes in no arguments. For example, in Java a linked list class could have a constructor that takes no arguments and creates an empty list:

```
class LinkedList {
  /** Returns an empty list. */
  LinkedList() { ... }
}
```

Mutable values. In `mlist`, the nodes of the list are mutable, but the values are not. If we wanted the values also to be mutable, we can make them refs too:

```
type 'a node = { next : 'a mlist; value : 'a ref }
and 'a mlist = 'a node option ref

let empty () : 'a mlist = ref None

let insert_first (lst : 'a mlist) (v : 'a) : unit =
  lst := Some { next = ref !lst; value = ref v }

let rec set (lst : 'a mlist) (n : int) (v : 'a) : unit =
  match (!lst, n) with
  | None, _ -> invalid_arg "out of bounds"
  | Some { value }, 0 -> value := v
  | Some { next }, _ -> set next (n - 1) v

let rec to_list (lst : 'a mlist) : 'a list =
  match !lst with None -> [] | Some { next; value } -> !value :: to_list next
```

Now rather than having to create new nodes if we want to change a value, we can directly mutate the value in a node:

```
let lst = empty ();;
insert_first lst 42;;
insert_first lst 41;;
to_list lst;;
set lst 1 43;;
to_list lst;;
```

9.2 Mutable Fields

The fields of a record can be declared as mutable, meaning their contents can be updated without constructing a new record. For example, here is a record type for two-dimensional colored points whose color field `c` is mutable:

```
type point = {x : int; y : int; mutable c : string}
```

Note that `mutable` is a property of the field, rather than the type of the field. In particular, we write `mutable field : type`, not `field : mutable type`.

The operator to update a mutable field is `<-` which is meant to look like a left arrow.

```
let p = {x = 0; y = 0; c = "red"}
```

```
p.c <- "white"
```

```
p
```

Non-mutable fields cannot be updated that way:

```
p.x <- 3;;
```

- **Syntax:** `e1.f <- e2`
- **Dynamic semantics:** To evaluate `e1.f <- e2`, evaluate `e2` to a value `v2`, and `e1` to a value `v1`, which must have a field named `f`. Update `v1.f` to `v2`. Return `()`.
- **Static semantics:** `e1.f <- e2 : unit` if `e1 : t1` and `t1 = {...; mutable f : t2; ...}`, and `e2 : t2`.

9.2.1 Refs Are Mutable Fields

It turns out that refs are actually implemented as mutable fields. In `Stdlib` we find the following declaration:

```
type 'a ref = { mutable contents : 'a }
```

And that's why when the toplevel outputs a ref it looks like a record: it *is* a record with a single mutable field named `contents`!

```
let r = ref 42
```

The other syntax we've seen for records is in fact equivalent to simple OCaml functions:

```
let ref x = {contents = x}
```

```
let ( ! ) r = r.contents
```

```
let ( := ) r x = r.contents <- x
```

The reason we say "equivalent" is that those functions are actually implemented not in OCaml itself but in the OCaml run-time, which is implemented mostly in C. Nonetheless the functions do behave the same as the OCaml source given above.

9.2.2 Example: Mutable Singly-Linked Lists

Using mutable fields, we can implement singly-linked lists almost the same as we did with references. The types for nodes and lists are simplified:

```
(** An ['a node] is a node of a mutable singly-linked list. It contains a value
    of type ['a] and optionally has a pointer to the next node. *)
type 'a node = {
  mutable next : 'a node option;
  value : 'a
}

(** An ['a mlist] is a mutable singly-linked list with elements of type ['a].
    RI: The list does not contain any cycles. *)
type 'a mlist = {
  mutable first : 'a node option;
}
```

And there is no essential difference in the algorithms for implementing the operations, but the code is slightly simplified because we don't have to use reference operations:

```
(** [insert_first lst n] mutates mlist [lst] by inserting value [v] as the
    first value in the list. *)
let insert_first (lst : 'a mlist) (v : 'a) =
  lst.first <- Some {value = v; next = lst.first}

(** [empty ()] is an empty singly-linked list. *)
let empty () : 'a mlist = {
  first = None
}

(** [to_list lst] is an OCaml list containing the same values as [lst]
    in the same order. Not tail recursive. *)
let to_list (lst : 'a mlist) : 'a list =
  let rec helper = function
    | None -> []
    | Some {next; value} -> value :: helper next
  in
  helper lst.first
```

9.2.3 Example: Mutable Stacks

We already know that lists and stacks can be implemented in quite similar ways. Let's use what we've learned from mutable linked lists to implement mutable stacks. Here is an interface:

```
module type MutableStack = sig
  (** ['a t] is the type of mutable stacks whose elements have type ['a].
      The stack is mutable not in the sense that its elements can
      be changed, but in the sense that it is not persistent:
      the operations [push] and [pop] destructively modify the stack. *)
  type 'a t

  (** Raised if [peek] or [pop] encounter the empty stack. *)
  exception Empty
```

(continues on next page)

(continued from previous page)

```
  (** [empty ()] is the empty stack *)
  val empty : unit -> 'a t

  (** [push x s] modifies [s] to make [x] its top element.
      The rest of the elements are unchanged. *)
  val push : 'a -> 'a t -> unit

  (**[peek s] is the top element of [s].
     Raises: [Empty] is [s] is empty. *)
  val peek : 'a t -> 'a

  (** [pop s] removes the top element of [s].
      Raises: [Empty] is [s] is empty. *)
  val pop : 'a t -> unit
end
```

Now let's implement the mutable stack with a mutable linked list.

```
module MutableRecordStack : MutableStack = struct
  (** An ['a node] is a node of a mutable linked list.  It has
     a field [value] that contains the node's value, and
     a mutable field [next] that is [None] if the node has
     no successor, or [Some n] if the successor is [n]. *)
  type 'a node = {value : 'a; mutable next : 'a node option}

 (** AF: An ['a t] is a stack represented by a mutable linked list.
     The mutable field [top] is the first node of the list,
     which is the top of the stack. The empty stack is represented
     by {top = None}.  The node {top = Some n} represents the
     stack whose top is [n], and whose remaining elements are
     the successors of [n]. *)
  type 'a t = {mutable top : 'a node option}

  exception Empty

  let empty () = {top = None}

  let push x s = s.top <- Some {value = x; next = s.top}

  let peek s =
    match s.top with
    | None -> raise Empty
    | Some {value} -> value

  let pop s =
    match s.top with
    | None -> raise Empty
    | Some {next} -> s.top <- next
end
```

9.3 Arrays and Loops

Arrays are fixed-length mutable sequences with constant-time access and update. So they are similar in various ways to refs, lists, and tuples. Like refs, they are mutable. Like lists, they are (finite) sequences. Like tuples, their length is fixed in advance and cannot be resized.

The syntax for arrays is similar to lists:

```
let v = [|0.; 1.|]
```

That code creates an array whose length is fixed to be 2 and whose contents are initialized to `0.` and `1.`. The keyword `array` is a type constructor, much like `list`.

Later those contents can be changed using the `<-` operator:

```
v.(0) <- 5.
```

```
v
```

As you can see in that example, indexing into an array uses the syntax `array.(index)`, where the parentheses are mandatory.

The `Array` module has many useful functions on arrays.

Syntax.

- Array creation: `[|e0; e1; ...; en|]`
- Array indexing: `e1.(e2)`
- Array assignment: `e1.(e2) <- e3`

Dynamic semantics.

- To evaluate `[|e0; e1; ...; en|]`, evaluate each `ei` to a value `vi`, create a new array of length `n+1`, and store each value in the array at its index.
- To evaluate `e1.(e2)`, evaluate `e1` to an array value `v1`, and `e2` to an integer `v2`. If `v2` is not within the bounds of the array (i.e., `0` to `n-1`, where `n` is the length of the array), raise `Invalid_argument`. Otherwise, index into `v1` to get the value `v` at index `v2`, and return `v`.
- To evaluate `e1.(e2) <- e3`, evaluate each expression `ei` to a value `vi`. Check that `v2` is within bounds, as in the semantics of indexing. Mutate the element of `v1` at index `v2` to be `v3`.

Static semantics.

- `[|e0; e1; ...; en|] : t array` if `ei : t` for all the `ei`.
- `e1.(e2) : t` if `e1 : t array` and `e2 : int`.
- `e1.(e2) <- e3 : unit` if `e1 : t array` and `e2 : int` and `e3 : t`.

Loops.

OCaml has while loops and for loops. Their syntax is as follows:

```
while e1 do e2 done
for x=e1 to e2 do e3 done
for x=e1 downto e2 do e3 done
```

Each of these three expressions evaluates the expression between `do` and `done` for each iteration of the loop; `while` loops terminate when `e1` becomes false; `for` loops execute once for each integer from `e1` to `e2`; `for..to` loops evaluate starting at `e1` and incrementing `x` each iteration; `for..downto` loops evaluate starting at `e1` and decrementing `x` each iteration. All three expressions evaluate to `()` after the termination of the loop. Because they always evaluate to `()`, they are less general than folds, maps, or recursive functions.

Loops are themselves not inherently mutable, but they are most often used in conjunction with mutable features like arrays—typically, the body of the loop causes side effects. We can also use functions like `Array.iter`, `Array.map`, and `Array.fold_left` instead of loops.

9.4 Summary

Mutable data types make programs harder to reason about. For example, before refs, we didn't have to worry about aliasing in OCaml. But mutability does have its uses. I/O is fundamentally about mutation. And some data structures (like arrays and hash tables) cannot be implemented as efficiently without mutability.

Mutability thus offers great power, but with great power comes great responsibility. Try not to abuse your new-found power!

9.4.1 Terms and Concepts

- address
- alias
- array
- assignment
- dereference
- determinstic
- immutable
- index
- loop
- memory safety
- mutable
- mutable field
- nondeterministic
- physical equality
- pointer
- pure
- ref
- ref cell
- reference
- sequencing
- structural equality

9.4.2 Further Reading

- *Introduction to Objective Caml*, chapters 7 and 8.
- *OCaml from the Very Beginning*, chapter 13.
- *Real World OCaml*, chapters 8.

9.5 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Exercise: mutable fields [★]

Define an OCaml record type to represent student names and GPAs. It should be possible to mutate the value of a student's GPA. Write an expression defining a student with name `"Alice"` and GPA `3.7`. Then write an expression to mutate Alice's GPA to `4.0`.

Exercise: refs [★]

Give OCaml expressions that have the following types. Use utop to check your answers.

- `bool ref`
- `int list ref`
- `int ref list`

Exercise: inc fun [★]

Define a reference to a function as follows:

```
let inc = ref (fun x -> x + 1)
```

Write code that uses `inc` to produce the value `3110`.

Exercise: addition assignment [★★]

The C language and many languages derived from it, such as Java, has an *addition assignment* operator written `a += b` and meaning `a = a + b`. Implement such an operator in OCaml; its type should be `int ref -> int -> unit`. Here's some code to get you started:

```
let ( +:= ) x y = ...
```

And here's an example usage:

```
# let x = ref 0;;
# x +:= 3110;;
# !x;;
- : int = 3110
```

Exercise: physical equality [★★]

Define `x`, `y`, and `z` as follows:

```
let x = ref 0
let y = x
let z = ref 0
```

Predict the value of the following series of expressions:

```
# x == y;;
# x == z;;
# x = y;;
# x = z;;
# x := 1;;
# x = y;;
# x = z;;
```

Check your answers in utop.

Exercise: norm [★★]

The Euclidean norm of an n-dimensional vector $x = (x_1, \ldots, x_n)$ is written $|x|$ and is defined to be

$$\sqrt{x_1^2 + \cdots + x_n^2}.$$

Write a function `norm : vector -> float` that computes the Euclidean norm of a vector, where `vector` is defined as follows:

```
(* AF: the float array [| x1; ...; xn |] represents the
 *     vector (x1, ..., xn)
 * RI: the array is non-empty *)
type vector = float array
```

Your function should not mutate the input array. *Hint: although your first instinct might be to reach for a loop, instead try to use `Array.map` and `Array.fold_left` or `Array.fold_right`.*

Exercise: normalize [★★]

Every vector can be *normalized* by dividing each component by $|x|$; this yields a vector with norm 1:

$$\left(\frac{x_1}{|x|}, \ldots, \frac{x_n}{|x|}\right)$$

Write a function `normalize : vector -> unit` that normalizes a vector "in place" by mutating the input array. Here's a sample usage:

```
# let a = [|1.; 1.|];;
val a : float array = [|1.; 1.|]

# normalize a;;
- : unit = ()

# a;;
- : float array = [|0.7071...; 0.7071...|]
```

Hint: `Array.iteri`.

Exercise: norm loop [★★]

Modify your implementation of `norm` to use a loop. Here is pseudocode for what you should do:

```
initialize norm to 0.0
loop through array
  add to norm the square of the current array component
return sqrt of norm
```

Exercise: normalize loop [★★]

Modify your implementation of `normalize` to use a loop.

Exercise: init matrix [★★★]

The `Array` module contains two functions for creating an array: `make` and `init`. `make` creates an array and fills it with a default value, while `init` creates an array and uses a provided function to fill it in. The library also contains a function `make_matrix` for creating a two-dimensional array, but it does not contain an analogous `init_matrix` to create a matrix using a function for initialization.

Write a function `init_matrix : int -> int -> (int -> int -> 'a) -> 'a array array` such that `init_matrix n o f` creates and returns an `n` by `o` matrix `m` with `m.(i).(j) = f i j` for all `i` and `j` in bounds.

See the documentation for `make_matrix` for more information on the representation of matrices as arrays.

Exercise: doubly linked list [★★★★]

Implement a data abstraction for a mutable doubly-linked list. Here is a representation type to get you started:

```
(** An ['a node] is a node of a mutable doubly-linked list.
    It contains a value of type ['a] and optionally has
    pointers to previous and/or next nodes. *)
type 'a node = {
  mutable prev : 'a node option;
  mutable next : 'a node option;
  value : 'a
}

(** An ['a dlist] is a mutable doubly-linked list with elements
    of type ['a].  It is possible to access the first and
    last elements in constant time.
    RI: The list does not contain any cycles. *)
type 'a dlist = {
  mutable first : 'a node option;
  mutable last : 'a node option;
}
```

Implement at least these operations:

- create an empty list

- insert a new first value
- insert a new last value
- insert a new node after a given node
- insert a new node before a given node
- remove a node
- iterate forward through the list applying a function
- iterate backward through the list applying a function

Hint: draw pictures! Reasoning about mutable data structures is typically easier if you draw a picture.

CHAPTER

TEN

DATA STRUCTURES

Efficient data structures are important building blocks for large programs. In this chapter, we'll discuss what it means to be efficient, how to implement some efficient data structures using both imperative and functional programming, and learn about the technique of *amortized analysis*.

Of course, we've already covered quite a few simple data structures, especially in the *modules chapter*, where we used lists to implement stacks, queues, maps, and sets. For stacks and (batched) queues, those implementations were already efficient. But we can do much better for maps (and sets). In this chapter we'll see efficient implementations of maps using hash tables and red-black trees.

We'll also take a look at some cool functional data structures that appear less often in imperative languages: *sequences*, which are infinite lists implemented with functions called *thunks*; *lazy lists*, which are implemented with a language feature (aptly called "laziness") that suspends evaluation; *promises*, which are a way of organizing concurrent computations that has recently become popular in imperative web programming; and *monads*, which are a way of organizing any kind of computation that has (side) effects.

10.1 Hash Tables

The *hash table* is a widely used data structure whose performance relies upon mutability. The implementation of a hash table is quite involved compared to other data structures we've implemented so far. We'll build it up slowly, so that the need for and use of each piece can be appreciated.

10.1.1 Maps

Hash tables implement the *map* data abstraction. A map binds *keys* to *values*. This abstraction is so useful that it goes by many other names, among them *associative array*, *dictionary*, and *symbol table*. We'll write maps abstractly (i.e, mathematically; not actually OCaml syntax) as $\{ k_1 : v_1, k_2 : v_2, \ldots, k_n : v_n \}$. Each $k : v$ is a *binding* of key k to value v. Here are a couple of examples:

- A map binding a course number to something about it: {3110 : "Fun", 2110 : "OO"}.
- A map binding a university name to the year it was chartered: {"Harvard" : 1636, "Princeton" : 1746, "Penn": 1740, "Cornell" : 1865}.

The order in which the bindings are abstractly written does not matter, so the first example might also be written {2110 : "OO", 3110 : "Fun"}. That's why we use set braces—they suggest that the bindings are a set, with no ordering implied.

Note: As that notation suggests, maps and sets are very similar. Data structures that can implement a set can also implement a map, and vice-versa:

- Given a map data structure, we can treat the keys as elements of a set, and simply ignore the values which the keys are bound to. This admittedly wastes a little space, because we never need the values.

- Given a set data structure, we can store key–value pairs as the elements. Searching for elements (hence insertion and removal) might become more expensive, because the set abstraction is unlikely to support searching for keys by themselves.

Here is an interface for maps:

```
module type Map = sig

  (** [('k, 'v) t] is the type of maps that bind keys of type
      ['k] to values of type ['v]. *)
  type ('k, 'v) t

  (** [insert k v m] is the same map as [m], but with an additional
      binding from [k] to [v].  If [k] was already bound in [m],
      that binding is replaced by the binding to [v] in the new map. *)
  val insert : 'k -> 'v -> ('k, 'v) t -> ('k, 'v) t

  (** [find k m] is [Some v] if [k] is bound to [v] in [m],
      and [None] if not. *)
  val find : 'k -> ('k, 'v) t -> 'v option

  (** [remove k m] is the same map as [m], but without any binding of [k].
      If [k] was not bound in [m], then the map is unchanged. *)
  val remove : 'k -> ('k, 'v) t -> ('k, 'v) t

  (** [empty] is the empty map. *)
  val empty : ('k, 'v) t

  (** [of_list lst] is a map containing the same bindings as
      association list [lst].
      Requires: [lst] does not contain any duplicate keys. *)
  val of_list : ('k * 'v) list -> ('k, 'v) t

  (** [bindings m] is an association list containing the same
      bindings as [m]. There are no duplicates in the list. *)
  val bindings : ('k, 'v) t -> ('k * 'v) list
end
```

Next, we're going to examine three implementations of maps based on

- association lists,
- arrays, and
- a combination of the above known as a *hash table with chaining*.

Each implementation will need a slightly different interface, because of constraints resulting from the underlying representation type. In each case we'll pay close attention to the AF, RI, and efficiency of the operations.

10.1.2 Maps as Association Lists

The simplest implementation of a map in OCaml is as an association list. We've seen that representation twice so far *[1]* *[2]*. Here is an implementation of `Map` using it:

```
module ListMap : Map = struct
  (** AF: [[(k1, v1); (k2, v2); ...; (kn, vn)]] is the map {k1 : v1, k2 : v2,
      ..., kn : vn}. If a key appears more than once in the list, then in the
      map it is bound to the left-most occurrence in the list. For example,
      [[(k, v1); (k, v2)]] represents {k : v1}. The empty list represents
      the empty map.
      RI: none. *)
  type ('k, 'v) t = ('k * 'v) list

  (** Efficiency: O(1). *)
  let insert k v m = (k, v) :: m

  (** Efficiency: O(n). *)
  let find = List.assoc_opt

  (** Efficiency: O(n). *)
  let remove k lst = List.filter (fun (k', _) -> k <> k') lst

  (** Efficiency: O(1). *)
  let empty = []

  (** Efficiency: O(1). *)
  let of_list lst = lst

  (** [keys m] is a list of the keys in [m], without
      any duplicates.
      Efficiency: O(n log n). *)
  let keys m = m |> List.map fst |> List.sort_uniq Stdlib.compare

  (** [binding m k] is [(k, v)], where [v] is the value that [k]
       binds in [m].
       Requires: [k] is a key in [m].
       Efficiency: O(n). *)
  let binding m k = (k, List.assoc k m)

  (** Efficiency: O(n log n) + O(n) * O(n), which is O(n^2). *)
  let bindings m = List.map (binding m) (keys m)
end
```

10.1.3 Maps as Arrays

Mutable maps are maps whose bindings may be mutated. The interface for a mutable map therefore differs from a immutable map. Insertion and removal operations for a mutable map therefore return `unit`, because they do not produce a new map but instead mutate an existing map.

An array can be used to represent a mutable map whose keys are integers. A binding from a key to a value is stored by using the key as an index into the array, and storing the binding at that index. For example, we could use an array to map office numbers to their occupants:

Office	Occupant
459	Fan
460	Gries
461	Clarkson
462	Muhlberger
463	*does not exist*

This kind of map is called a *direct address table*. Since arrays have a fixed size, the implementer now needs to know the client's desire for the *capacity* of the table (i.e., the number of bindings that can be stored in it) whenever an empty table is created. That leads to the following interface:

```
module type DirectAddressMap = sig
  (** [t] is the type of maps that bind keys of type int to values of
      type ['v]. *)
  type 'v t

  (** [insert k v m] mutates map [m] to bind [k] to [v]. If [k] was
      already bound in [m], that binding is replaced by the binding to
      [v] in the new map. Requires: [k] is in bounds for [m]. *)
  val insert : int -> 'v -> 'v t -> unit

  (** [find k m] is [Some v] if [k] is bound to [v] in [m], and [None]
      if not. Requires: [k] is in bounds for [m]. *)
  val find : int -> 'v t -> 'v option

  (** [remove k m] mutates [m] to remove any binding of [k]. If [k] was
      not bound in [m], then the map is unchanged. Requires: [k] is in
      bounds for [m]. *)
  val remove : int -> 'v t -> unit

  (** [create c] creates a map with capacity [c]. Keys [0] through [c-1]
      are _in bounds_ for the map. *)
  val create : int -> 'v t

  (** [of_list c lst] is a map containing the same bindings as
      association list [lst] and with capacity [c]. Requires: [lst] does
      not contain any duplicate keys, and every key in [lst] is in
      bounds for capacity [c]. *)
  val of_list : int -> (int * 'v) list -> 'v t

  (** [bindings m] is an association list containing the same bindings
      as [m]. There are no duplicate keys in the list. *)
  val bindings : 'v t -> (int * 'v) list
end
```

Here is an implementation of that interface:

```
module ArrayMap : DirectAddressMap = struct
  (** AF: [[|Some v0; Some v1; ... |]] represents {0 : v0, 1 : v1, ...}.
      If element [i] of the array is instead [None], then [i] is not
      bound in the map.
      RI: None. *)
  type 'v t = 'v option array

  (** Efficiency: O(1) *)
```

(continues on next page)

(continued from previous page)

```
  let insert k v a = a.(k) <- Some v

  (** Efficiency: O(1) *)
  let find k a = a.(k)

  (** Efficiency: O(1) *)
  let remove k a = a.(k) <- None

  (** Efficiency: O(c) *)
  let create c = Array.make c None

  (** Efficiency: O(c) *)
  let of_list c lst =
    (* O(c) *)
    let a = create c in
    (* O(c) * O(1) = O(c) *)
    List.iter (fun (k, v) -> insert k v a) lst;
    a

  (** Efficiency: O(c) *)
  let bindings a =
    let bs = ref [] in
    (* O(1) *)
    let add_binding k v =
      match v with None -> () | Some v -> bs := (k, v) :: !bs
    in
    (* O(c) *)
    Array.iteri add_binding a;
    !bs
end
```

Its efficiency is great! The `insert`, `find`, and `remove` operations are constant time. But that comes at the expense of forcing keys to be integers. Moreover, they need to be small integers (or at least integers from a small range), otherwise the arrays we use will need to be huge.

10.1.4 Maps as Hash Tables

Arrays offer constant time performance, but come with severe restrictions on keys. Association lists don't place those restrictions on keys, but they also don't offer constant time performance. Is there a way to get the best of both worlds? Yes (more or less)! *Hash tables* are the solution.

The key idea is that we assume the existence of a *hash function* `hash : 'a -> int` that can convert any key to a non-negative integer. Then we can use that function to index into an array, as we did with direct address tables. Of course, we want the hash function itself to run in constant time, otherwise the operations that use it would not be efficient.

That leads to the following interface, in which the client of the hash table has to pass in a hash function when a table is created:

```
module type TableMap = sig
  (** [('k, 'v) t] is the type of mutable table-based maps that bind
      keys of type ['k] to values of type ['v]. *)
  type ('k, 'v) t

  (** [insert k v m] mutates map [m] to bind [k] to [v]. If [k] was
      already bound in [m], that binding is replaced by the binding to
```

(continues on next page)

(continued from previous page)

```
      [v]. *)
  val insert : 'k -> 'v -> ('k, 'v) t -> unit

  (** [find k m] is [Some v] if [m] binds [k] to [v], and [None] if [m]
      does not bind [k]. *)
  val find : 'k -> ('k, 'v) t -> 'v option

  (** [remove k m] mutates [m] to remove any binding of [k]. If [k] was
      not bound in [m], the map is unchanged. *)
  val remove : 'k -> ('k, 'v) t -> unit

  (** [create hash c] creates a new table map with capacity [c] that
      will use [hash] as the function to convert keys to integers.
      Requires: The output of [hash] is always non-negative, and [hash]
      runs in constant time. *)
  val create : ('k -> int) -> int -> ('k, 'v) t

  (** [bindings m] is an association list containing the same bindings
      as [m]. *)
  val bindings : ('k, 'v) t -> ('k * 'v) list

  (** [of_list hash lst] creates a map with the same bindings as [lst],
      using [hash] as the hash function. Requires: [lst] does not
      contain any duplicate keys. *)
  val of_list : ('k -> int) -> ('k * 'v) list -> ('k, 'v) t
end
```

One immediate problem with this idea is what to do if the output of the hash is not within the bounds of the array. It's easy to solve this: if `a` is the length of the array then computing `(hash k) mod a` will return an index that is within bounds.

Another problem is what to do if the hash function is not *injective*, meaning that it is not one-to-one. Then multiple keys could *collide* and need to be stored at the same index in the array. That's okay! We deliberately allow that. But it does mean we need a strategy for what to do when keys collide.

There are two well-known strategies for dealing with collisions. One is to store multiple bindings at each array index. The array elements are called *buckets*. Typically, the bucket is implemented as a linked list. This strategy is known by many names, including *chaining*, *closed addressing*, and *open hashing*. We'll use **chaining** as the name. To check whether an element is in the hash table, the key is first hashed to find the correct bucket to look in. Then, the linked list is scanned to see if the desired element is present. If the linked list is short, this scan is very quick. An element is added or removed by hashing it to find the correct bucket. Then, the bucket is checked to see if the element is there, and finally the element is added or removed appropriately from the bucket in the usual way for linked lists.

The other strategy is to store bindings at places other than their proper location according to the hash. When adding a new binding to the hash table would create a collision, the insert operation instead finds an empty location in the array to put the binding. This strategy is (confusingly) known as *probing*, *open addressing*, and *closed hashing*. We'll use **probing** as the name. A simple way to find an empty location is to search ahead through the array indices with a fixed stride (often 1), looking for an unused entry; this *linear probing* strategy tends to produce a lot of clustering of elements in the table, leading to bad performance. A better strategy is to use a second hash function to compute the probing interval; this strategy is called *double hashing*. Regardless of how probing is implemented, however, the time required to search for or add an element grows rapidly as the hash table fills up.

Chaining has often been preferred over probing in software implementations, because it's easy to implement the linked lists in software. Hardware implementations have often used probing, when the size of the table is fixed by circuitry. But some modern software implementations are re-examining the performance benefits of probing.

Chaining Representation

Here is a representation type for a hash table that uses chaining:

```
type ('k, 'v) t = {
  hash : 'k -> int;
  mutable size : int;
  mutable buckets : ('k * 'v) list array
}
```

The `buckets` array has elements that are association lists, which store the bindings. The `hash` function is used to determine which bucket a key goes into. The `size` is used to keep track of the number of bindings currently in the table, since that would be expensive to compute by iterating over `buckets`.

Here are the AF and RI:

```
  (** AF:  If [buckets] is
        [| [(k11,v11); (k12,v12); ...];
           [(k21,v21); (k22,v22); ...];
           ... |]
      that represents the map
        {k11:v11, k12:v12, ...,
         k21:v21, k22:v22, ...,  ...}.
      RI: No key appears more than once in array (so, no
        duplicate keys in association lists).  All keys are
        in the right buckets: if [k] is in [buckets] at index
        [b] then [hash(k) = b]. The output of [hash] must always
        be non-negative. [hash] must run in constant time.*)
```

What would the efficiency of `insert`, `find`, and `remove` be for this rep type? All require

- hashing the key (constant time),
- indexing into the appropriate bucket (constant time), and
- finding out whether the key is already in the association list (linear in the number of elements in that list).

So the efficiency of the hash table depends on the number of elements in each bucket. That, in turn, is determined by how well the hash function distributes keys across all the buckets.

A terrible hash function, such as the constant function `fun k -> 42`, would put all keys into same bucket. Then every operation would be linear in the number n of bindings in the map—that is, $O(n)$. We definitely don't want that.

Instead, we want hash functions that distribute keys more or less randomly across the buckets. Then the expected length of every bucket will be about the same. If we could arrange that, on average, the bucket length were a constant L, then `insert`, `find`, and `remove` would all in expectation run in time $O(L)$.

Resizing

How could we arrange for buckets to have expected constant length? To answer that, let's think about the number of bindings and buckets in the table. Define the *load factor* of the table to be

$$\frac{\text{number of bindings}}{\text{number of buckets}}$$

So a table with 20 bindings and 10 buckets has a load factor of 2, and a table with 10 bindings and 20 buckets has a load factor of 0.5. The load factor is therefore the average number of bindings in a bucket. So if we could keep the load factor constant, we could keep L constant, thereby keeping the performance to (expected) constant time.

Toward that end, note that the number of bindings is not under the control of the hash table implementer—but the number of buckets is. So by changing the number of buckets, the implementer can change the load factor. A common strategy is to keep the load factor from approximately 1/2 to 2. Then each bucket contains only a couple bindings, and expected constant-time performance is guaranteed.

There's no way for the implementer to know in advance, though, exactly how many buckets will be needed. So instead, the implementer will have to *resize* the bucket array whenever the load factor gets too high. Typically the newly allocated bucket will be of a size to restore the load factor to about 1.

Putting those two ideas together, if the load factor reaches 2, then there are twice as many bindings as buckets in the table. So by doubling the size of the array, we can restore the load factor to 1. Similarly, if the load factor reaches 1/2, then there are twice as many buckets as bindings, and halving the size of the array will restore the load factor to 1.

Resizing the bucket array to become larger is an essential technique for hash tables. Resizing it to become smaller, though, is not essential. As long as the load factor is bounded by a constant from above, we can achieve expected constant bucket length. So not all implementations will reduce the size of the array. Although doing so would recover some space, it might not be worth the effort. That's especially true if the size of the hash table cycles over time: although sometimes it becomes smaller, eventually it becomes bigger again.

Unfortunately, resizing would seem to ruin our expected constant-time performance though. Insertion of a binding might cause the load factor to go over 2, thus causing a resize. When the resize occurs, all the existing bindings must be rehashed and added to the new bucket array. Thus, insertion has become a worst-case linear time operation! The same is true for removal, if we resize the array to become smaller when the load factor is too low.

Implementation

The implementation of a hash table, below, puts together all the pieces we discussed above.

```
module HashMap : TableMap = struct

  (** AF and RI: above *)
  type ('k, 'v) t = {
    hash : 'k -> int;
    mutable size : int;
    mutable buckets : ('k * 'v) list array
  }

  (** [capacity tab] is the number of buckets in [tab].
      Efficiency: O(1) *)
  let capacity {buckets} =
    Array.length buckets

  (** [load_factor tab] is the load factor of [tab], i.e., the number of
      bindings divided by the number of buckets. *)
  let load_factor tab =
    float_of_int tab.size /. float_of_int (capacity tab)

  (** Efficiency: O(n) *)
  let create hash n =
    {hash; size = 0; buckets = Array.make n []}

  (** [index k tab] is the index at which key [k] should be stored in the
      buckets of [tab].
      Efficiency: O(1) *)
  let index k tab =
    (tab.hash k) mod (capacity tab)
```

(continues on next page)

(continued from previous page)

```
(** [insert_no_resize k v tab] inserts a binding from [k] to [v] in [tab]
    and does not resize the table, regardless of what happens to the
    load factor.
    Efficiency: expected O(L) *)
let insert_no_resize k v tab =
  let b = index k tab in (* O(1) *)
  let old_bucket = tab.buckets.(b) in
  tab.buckets.(b) <- (k,v) :: List.remove_assoc k old_bucket; (* O(L) *)
  if not (List.mem_assoc k old_bucket) then
    tab.size <- tab.size + 1;
  ()

(** [rehash tab new_capacity] replaces the buckets array of [tab] with a new
    array of size [new_capacity], and re-inserts all the bindings of [tab]
    into the new array.  The keys are re-hashed, so the bindings will
    likely land in different buckets.
    Efficiency: O(n), where n is the number of bindings. *)
let rehash tab new_capacity =
  (* insert (k, v) into tab *)
  let rehash_binding (k, v) =
    insert_no_resize k v tab
  in
  (* insert all bindings of bucket into tab *)
  let rehash_bucket bucket =
    List.iter rehash_binding bucket
  in
  let old_buckets = tab.buckets in
  tab.buckets <- Array.make new_capacity []; (* O(n) *)
  tab.size <- 0;
  (* [rehash_binding] is called by [rehash_bucket] once for every binding *)
  Array.iter rehash_bucket old_buckets (* expected O(n) *)

(* [resize_if_needed tab] resizes and rehashes [tab] if the load factor
   is too big or too small.  Load factors are allowed to range from
   1/2 to 2. *)
let resize_if_needed tab =
  let lf = load_factor tab in
  if lf > 2.0 then
    rehash tab (capacity tab * 2)
  else if lf < 0.5 then
    rehash tab (capacity tab / 2)
  else ()

(** Efficiency: O(n) *)
let insert k v tab =
  insert_no_resize k v tab; (* O(L) *)
  resize_if_needed tab (* O(n) *)

(** Efficiency: expected O(L) *)
let find k tab =
  List.assoc_opt k tab.buckets.(index k tab)

(** [remove_no_resize k tab] removes [k] from [tab] and does not trigger
    a resize, regardless of what happens to the load factor.
    Efficiency: expected O(L) *)
let remove_no_resize k tab =
```

(continues on next page)

(continued from previous page)

```
    let b = index k tab in
    let old_bucket = tab.buckets.(b) in
    tab.buckets.(b) <- List.remove_assoc k tab.buckets.(b);
    if List.mem_assoc k old_bucket then
      tab.size <- tab.size - 1;
    ()

  (** Efficiency: O(n) *)
  let remove k tab =
    remove_no_resize k tab; (* O(L) *)
    resize_if_needed tab (* O(n) *)

  (** Efficiency: O(n) *)
  let bindings tab =
    Array.fold_left
      (fun acc bucket ->
         List.fold_left
           (* 1 cons for every binding, which is O(n) *)
           (fun acc (k,v) -> (k,v) :: acc)
           acc bucket)
      [] tab.buckets

  (** Efficiency: O(n^2) *)
  let of_list hash lst =
    let m = create hash (List.length lst) in  (* O(n) *)
    List.iter (fun (k, v) -> insert k v m) lst; (* n * O(n) is O(n^2) *)
    m
end
```

An optimization of `rehash` is possible. When it calls `insert_no_resize` to re-insert a binding, extra work is being done: there's no need for that insertion to call `remove_assoc` or `mem_assoc`, because we are guaranteed the binding does not contain a duplicate key. We could omit that work. If the hash function is good, it's only a constant amount of work that we save. But if the hash function is bad and doesn't distribute keys uniformly, that could be an important optimization.

10.1.5 Hash Functions

Hash tables are one of the most useful data structures ever invented. Unfortunately, they are also one of the most misused. Code built using hash tables often falls far short of achievable performance. There are two reasons for this:

- Clients choose poor hash functions that do not distribute keys randomly over buckets.
- Hash table abstractions do not adequately specify what is required of the hash function, or make it difficult to provide a good hash function.

Clearly, a bad hash function can destroy our attempts at a constant running time. A lot of obvious hash function choices are bad. For example, if we're mapping names to phone numbers, then hashing each name to its length would be a very poor function, as would a hash function that used only the first name, or only the last name. We want our hash function to use all of the information in the key. This is a bit of an art. While hash tables are extremely effective when used well, all too often poor hash functions are used that sabotage performance.

Hash tables work well when the hash function looks random. If it is to look random, this means that any change to a key, even a small one, should change the bucket index in an apparently random way. If we imagine writing the bucket index as a binary number, a small change to the key should randomly flip the bits in the bucket index. This is called information *diffusion*. For example, a one-bit change to the key should cause every bit in the index to flip with 1/2 probability.

Client vs. implementer. As we've described it, the hash function is a single function that maps from the key type to a bucket index. In practice, the hash function is the composition of *two* functions, one provided by the client and one by the implementer. This is because the implementer doesn't understand the element type, the client doesn't know how many buckets there are, and the implementer probably doesn't trust the client to achieve diffusion.

The client function `hash_c` first converts the key into an integer hash code, and the implementation function `hash_i` converts the hash code into a bucket index. The actual hash function is the composition of these two functions. As a hash table designer, you need to figure out which of the client hash function and the implementation hash function is going to provide diffusion. If clients are sufficiently savvy, it makes sense to push the diffusion onto them, leaving the hash table implementation as simple and fast as possible. The easy way to accomplish this is to break the computation of the bucket index into three steps.

1. Serialization: Transform the key into a stream of bytes that contains all of the information in the original key. Two equal keys must result in the same byte stream. Two byte streams should be equal only if the keys are actually equal. How to do this depends on the form of the key. If the key is a string, then the stream of bytes would simply be the characters of the string.
2. Diffusion: Map the stream of bytes into a large integer x in a way that causes every change in the stream to affect the bits of x apparently randomly. There is a tradeoff in performance versus randomness (and security) here.
3. Compression: Reduce that large integer to be within the range of the buckets. For example, compute the hash bucket index as x mod m. This is particularly cheap if m is a power of two.

Unfortunately, hash table implementations are rarely forthcoming about what they assume of client hash functions. So it can be hard to know, as a client, how to get good performance from a table. The more information the implementation can provide to a client about how well distributed keys are in buckets, the better.

10.1.6 Standard Library `Hashtbl`

Although it's great to know how to implement a hash table, and to see how mutability is used in doing so, it's also great *not* to have to implement a data structure yourself in your own projects. Fortunately the OCaml standard library does provide a module `Hashtbl` [sic] that implements hash tables. You can think of this module as the imperative equivalent of the functional `Map` module.

Hash function. The function `Hashtbl.hash : 'a -> int` takes responsibility for serialization and diffusion. It is capable of hashing any type of value. That includes not just integers but strings, lists, trees, and so forth. So how does it run in constant time, if the length of a tree or size of a tree can be arbitrarily large? It looks only at a predetermined number of *meaningful nodes* of the structure it is hashing. By default, that number is 10. A meaningful node is an integer, floating-point number, string, character, booleans or constant constructor. You can see that as we hash these lists:

```
Hashtbl.hash [1; 2; 3; 4; 5; 6; 7; 8; 9];;
Hashtbl.hash [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
Hashtbl.hash [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11];;
Hashtbl.hash [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12];;
```

The hash values stop changing after the list goes beyond 10 elements. That has implications for how we use this built-in hash function: it will not necessarily provide good diffusion for large data structures, which means performance could degrade as collisions become common. To support clients who want to hash such structures, `Hashtble` provides another function `hash_param` which can be configured to examine more nodes.

Hash table. Here's an abstract of the hash table interface:

```
module type Hashtbl = struct
  type ('a, 'b) t
  val create : int -> ('a, 'b) t
  val add : ('a, 'b) t -> 'a -> 'b -> unit
```

(continues on next page)

(continued from previous page)

```
  val find : ('a, 'b) t -> 'a -> 'b
  val remove : ('a, 'b) t -> 'a -> unit
  ...
end
```

The representation type `('a, 'b) Hashtbl.t` maps keys of type `'a` to values of type `'b`. The `create` function initializes a hash table to have a given capacity, as our implementation above did. But rather than requiring the client to provide a hash function, the module uses `Hashtbl.hash`.

Resizing occurs when the load factor exceeds 2. Let's see that happen. First, we'll create a table and fill it up:

```
open Hashtbl;;
let t = create 16;;
for i = 1 to 16 do
  add t i (string_of_int i)
done;;
```

We can query the hash table to find out how the bindings are distributed over buckets with `Hashtbl.stats`:

```
stats t
```

The number of bindings and number of buckets are equal, so the load factor is 1. The bucket histogram is an array `a` in which `a.(i)` is the number of buckets whose size is `i`.

Let's pump up the load factor to 2:

```
for i = 17 to 32 do
  add t i (string_of_int i)
done;;
stats t;;
```

Now adding one more binding will trigger a resize, which doubles the number of buckets:

```
add t 33 "33";;
stats t;;
```

But `Hashtbl` does not implement resize on removal:

```
for i = 1 to 33 do
  remove t i
done;;
stats t;;
```

The number of buckets is still 32, even though all bindings have been removed.

Note: Java's `HashMap` has a default constructor `HashMap()` that creates an empty hash table with a capacity of 16 that resizes when the load factor exceeds 0.75 rather than 2. So Java hash tables would tend to have a shorter bucket length than OCaml hash tables, but also would tend to take more space to store because of empty buckets.

Client-provided hash functions. What if a client of `Hashtbl` found that the default hash function was leading to collisions, hence poor performance? Then it would make sense to change to a different hash function. To support that, `Hashtbl` provides a functorial interface similar to `Map`. The functor is `Hashtbl.Make`, and it requires an input of the following module type:

```
module type HashedType = sig
  type t
  val equal : t -> t -> bool
  val hash : t -> int
end
```

Type `t` is the key type for the table, and the two functions `equal` and `hash` say how to compare keys for equality and how to hash them. If two keys are equal according to `equal`, they must have the same hash value according to `hash`. If that requirement were violated, the hash table would no longer operate correctly. For example, suppose that `equal k1 k2` holds but `hash k1 <> hash k2`. Then `k1` and `k2` would be stored in different buckets. So if a client added a binding of `k1` to `v`, then looked up `k2`, they would not get `v` back.

Note: That final requirement might sound familiar from Java. There, if you override `Object.equals()` and `Object.hashCode()` you must ensure the same correspondence.

10.2 Amortized Analysis

Our analysis of the efficiency of hash table operations concluded that `find` runs in expected constant time, where the modifier "expected" is needed to express the fact the performance is on average and depends on the hash function satisfying certain properties.

We also concluded that `insert` would usually run in expected constant time, but that in the worst case it would require linear time because of needing to rehash the entire table. That kind of defeats the goal of a hash table, which is to offer constant-time performance, or at least as close to it as we can get.

It turns out there is another way of looking at this analysis that allows us to conclude that `insert` does have "amortized" expected constant time performance—that is, for excusing the occasional worst-case linear performance. Right away, we have to acknowledge this technique is just a change in perspective. We're not going to change the underlying algorithms. The `insert` algorithm will still have worst-case linear performance. That's a fact.

But the change in perspective we now undertake is to recognize that if it's very rare for `insert` to require linear time, then maybe we can "spread out" that cost over all the other calls to `insert`. It's a creative accounting trick!

Sushi vs. Ramen. Let's amuse ourselves with a real-world example for a moment. Suppose that you have $20 to spend on lunches for the week. You like to eat sushi, but you can't afford to have sushi every day. So instead you eat as follows:

- Monday: $1 ramen
- Tuesday: $1 ramen
- Wednesday: $1 ramen
- Thursday: $1 ramen
- Friday: $16 sushi

Most of the time, your lunch was cheap. On a rare occasion, it was expensive. So you could look at it in one of two ways:

- My worst-case lunch cost was $16.
- My average lunch cost was $4.

Both are true statements, but maybe the latter is more helpful in understanding your spending habits.

Back to Hash Tables. It's the same with hash tables. Even though `insert` is occasionally expensive, it's so rarely expensive that the average cost of an operation is actually constant time! But, we need to do more complicated math (or more complicated than our lunch budgeting anyway) to actually demonstrate that's true.

10.2.1 Amortized Analysis of Hash Tables

"Amortization" is a financial term. One of its meanings is to pay off a debt over time. In algorithmic analysis, we use it to refer to paying off the cost of an expensive operation by inflating the cost of inexpensive operations. In effect, we pre-pay the cost of a later expensive operation by adding some additional cost to earlier cheap operations.

The *amortized complexity* or *amortized running time* of a sequence of operations that each have cost $T_1, T_2, ..., T_n$, is just the average cost of each operation:

$$\frac{T_1 + T_2 + ... + T_n}{n}.$$

Thus, even if one operation is especially expensive, we could average that out over a bunch of inexpensive operations.

Applying that idea to a hash table, suppose the table has 8 bindings and 8 buckets. Then 8 more inserts are made. The first 7 are (expected) constant-time, but the 8th insert is linear time: it increases the load factor to 2, causing a resize, thus causing rehashing of all 16 bindings into a new table. The total cost over that series of operations is therefore the cost of 8+16 inserts. For simplicity of calculation, we could grossly round that up to 16+16 = 32 inserts. So the average cost of each operation in the sequence is 32/8 = 4 inserts.

In other words, if we just pretended each insert cost four times its normal price, the final operation in the sequence would have been "pre-paid" by the extra price we paid for earlier inserts. And all of them would be constant-time, since four times a constant is still a constant.

Generalizing from the example above, let's suppose that the the number of buckets currently in a hash table is 2^n, and that the load factor is currently 1. Therefore, there are currently 2^n bindings in the table. Next:

- A series of $2^n - 1$ inserts occurs. There are now $2^n + 2^n - 1$ bindings in the table.
- One more insert occurs. That brings the number of bindings up to $2^n + 2^n$, which is 2^{n+1}. But the number of buckets is 2^n, so the the load factor just reached 2. A resize is necessary.
- The resize occurs. That doubles the number of buckets. All 2^{n+1} bindings have to be reinserted into the new table, which is of size 2^{n+1}. The load factor is back down to 1.

So in total we did $2^n + 2^{n+1}$ inserts, which included 2^n inserts of bindings and 2^{n+1} re-insertions after the resize. We could grossly round that quantity up to 2^{n+2}. Over a series of 2^n insert operations, that's an average cost of $\frac{2^{n+2}}{2^n}$, which equals 4. So if we just pretend each insert costs four times its normal price, every operation in the sequence is amortized (and expected) constant time.

Doubling vs. Constant-size Increasing. Notice that it is crucial that the array size grows by doubling (or at least geometrically). A bad mistake would be to instead grow the array by a fixed increment—for example, 100 buckets at time. Then we'd be in real trouble as the number of bindings continued to grow:

- Start with 100 buckets and 100 bindings. The load factor is 1.
- **Round 1.** Insert 100 bindings. There are now 200 bindings and 100 buckets. The load factor is 2.
- Increase the number of buckets by 100 and rehash. That's 200 more insertions. The load factor is back down to 1.
- The average cost of each insert is so far just 3x the cost of an actual insert (100+200 insertions / 100 bindings inserted). So far so good.
- **Round 2.** Insert 200 more bindings. There are now 400 bindings and 200 buckets. The load factor is 2.
- Increase the number of buckets **by 100** and rehash. That's 400 more insertions. There are now 400 bindings and 300 buckets. The load factor is 400/300 = 4/3, not 1.
- The average cost of each insert is now 100+200+200+400 / 300 = 3. That's still okay.
- **Round 3.** Insert 200 more bindings. There are now 600 bindings and 300 buckets. The load factor is 2.
- Increase the number of buckets **by 100** and rehash. That's 600 more insertions. There are now 600 bindings and 400 buckets. The load factor is 3/2, not 1.

- The average cost of each insert is now 100+200+200+400+200+600 / 500 = 3.2. It's going up.
- **Round 4.** Insert 200 more bindings. There are now 800 bindings and 400 buckets. The load factor is 2.
- Increase the number of buckets **by 100** and rehash. That's 800 more insertions. There are now 800 bindings and 500 buckets. The load factor is 8/5, not 1.
- The average cost of each insert is now 100+200+200+400+200+600+200+800 / 700 = 3.7. It's continuing to go up, not staying constant.

After k rounds we have $200k$ bindings and $100k$ buckets. We have called `insert` to insert $100 + 200k$ bindings, but all the rehashing has caused us to do $100 + 200(k-1) + \sum_{i=1}^{k} 200i$ actual insertions. That last term is the real problem. It's quadratic:

$$\sum_{i=1}^{k} 200i \quad = \quad \frac{200k(200(k+1))}{2} \quad = \quad 20,000(k^2+k)$$

So over a series of n calls to `insert`, we do $O(n^2)$ actual inserts. That makes the amortized cost of `insert` be $O(n)$, which is linear! Not constant.

That's why it's so important to double the size of the array at each rehash. It's what gives us the amortized constant-time performance.

10.2.2 Amortized Analysis of Batched Queues

The implementation of *batched queues* with two lists was in a way more efficient than the implementation with just one list, because it managed to achieve a constant time `enqueue` operation. But, that came at the tradeoff of making the `dequeue` operation sometimes take more than constant time: whenever the outbox became empty, the inbox had to be reversed, which required an additional linear-time operation.

As we observed then, the reversal is relatively rare. It happens only when the outbox gets exhausted. Amortized analysis gives us a way to account for that. We can actually show that the `dequeue` operation is amortized constant time.

To keep the analysis simple at first, let's assume the queue starts off with exactly one element `1` already enqueued, and that we do three `enqueue` operations of `2`, `3`, then `4`, followed by a single `dequeue`. The single initial element would end up in the outbox. All three `enqueue` operations would cons an element onto the inbox. So just before the `dequeue`, the queue looks like:

```
{o = [1]; i = [4; 3; 2]}
```

and after the `dequeue`:

```
{o = [2; 3; 4]; i = []}
```

It required

- 3 cons operations to do the 3 enqueues, and
- another 3 cons operations to finish the dequeue by reversing the list.

That's a total of 6 cons operations to do the 4 `enqueue` and `dequeue` operations. The average cost is therefore 1.5 cons operations per queue operation. There were other pattern matching operations and record constructions, but those all took only constant time, so we'll ignore them.

What about a more complicated situation, where there are `enqueues` and `dequeues` interspersed with one another? Trying to take averages over the series is going to be tricky to analyze. But, inspired by our analysis of hash tables, suppose we pretend that the cost of each `enqueue` is twice its actual cost, as measured in cons operations? Then at the time an

element is enqueued, we could "prepay" the later cost that will be incurred when that element is cons'd onto the reversed list.

The `enqueue` operation is still constant time, because even though we're now pretending its cost is 2 instead of 1, it's still the case that 2 is a constant. And the `dequeue` operation is amortized constant time:

- If `dequeue` doesn't need to reverse the inbox, it really does just constant work, and
- If `dequeue` does need to reverse an inbox with n elements, it already has n units of work "saved up" from each of the enqueues of those n elements.

So if we just pretend each enqueue costs twice its normal price, every operation in a sequence is amortized constant time. Is this just a bookkeeping trick? Absolutely. But it also reveals the deeper truth that on average we get constant-time performance, even though some operations might rarely have worst-case linear-time performance.

10.2.3 Bankers and Physicists

Conceptually, amortized analysis can be understood in three ways:

1. Taking the average cost over a series of operations. This is what we've done so far.
2. Keeping a "bank account" at each individual element of a data structure. Some operations deposit credits, and others withdraw them. The goal is for account totals to never be negative. The amortized cost of any operation is the actual cost, plus any credits deposited, minus any credits spent. So if an operation actually costs n but spends $n-1$ credits, then its amortized cost is just 1. This is called the *banker's method* of amortized analysis.
3. Regarding the entire data structure as having an amount of "potential energy" stored up. Some operations increase the energy, some decrease it. The energy should never be negative. The amortized cost of any operation is its actual cost, plus the change in potential energy. So if an operation actually costs n, and before the operation the potential energy is n, and after the operation the potential energy is 0, then the amortized cost is $n + (0 - n)$, which is just 0. This is called the *physicist's method* of amortized analysis.

The banker's and physicist's methods can be easier to use in many situations than a complicated analysis of a series of operations. Let's revisit our examples so far to illustrate their use:

- **Banker's method, hash tables:** The table starts off empty. When a binding is added to the table, save up 1 credit in its account. When a rehash becomes necessary, every binding is guaranteed to have 1 credit. Use that credit to pay for the rehash. Now all bindings have 0 credits. From now on, when a binding is added to the table, save up 1 credit in its account and 1 credit in the account of any one of the bindings that has 0 credits. At the time the next rehash becomes necessary, the number of bindings has doubled. But since we've saved 2 credits at each insertion, every binding now has 1 credit in its account again. So we can pay for the rehash. The accounts never go negative, because they always have either 0 or 1 credit.
- **Banker's method, batched queues:** When an element is added to the queue, save up 1 credit in its account. When the inbox must be reversed, use the credit in each element to pay for the cons onto the outbox. Since elements enter at the inbox and transition at most once to the outbox, every element will have 0 or 1 credits. So the accounts never go negative.
- **Physicist's method, hash tables:** At first, define the potential energy of the table to be the number of bindings inserted. That energy will therefore never be negative. Each insertion increases the energy by 1 unit. When the first rehash is needed after inserting n bindings, the potential energy is n. The potential goes back down to 0 at the rehash. So the actual cost is n, but the change in potential is n, which makes the amortized cost 0, or constant. From now on, define the potential energy to be twice the number of bindings inserted since the last rehash. Again, the energy will never be negative. Each insertion increases the energy by 2 units. When the next rehash is needed after inserting n bindings, there will be $2n$ bindings that need to be rehashed. Again, the amortized cost will be constant, because the actual cost of $2n$ re-insertions is offset by the $2n$ change in potential.

- **Physicist's method, batched queues:** Define the potential energy of the queue to be the length of the inbox. It therefore will never be negative. When a `dequeue` has to reverse an inbox of length n, there is an actual cost of n but a change in potential of n too, which offsets the cost and makes it constant.

The two methods are equivalent in their analytical power:

- To convert a banker's analysis into a physicist's, just make the potential be the sum of all the credits in the individual accounts.
- To convert a physicist's analysis into a banker's, just designate one distinguished element of the data structure to be the only one that will ever hold any credits, and have each operation deposit or withdraw the change in potential into that element's account.

So, the choice of which to use really just depends on which is easier for the data structure being analyzed, or which is easier for you to wrap your head around. You might find one or the other of the methods easier to understand for the data structures above, and your friend might have a different opinion.

10.2.4 Amortized Analysis and Persistence

Amortized analysis breaks down as a technique when data structures are used persistently. For example, suppose we have a batched queue `q` into which we've inserted $n + 1$ elements. One element will be in the outbox, and the other n will be in the inbox. Now we do the following:

```
# let q1 = dequeue q
# let q2 = dequeue q
...
# let qn = dequeue q
```

Each one of those n `dequeue` operations requires an actual cost of $O(n)$ to reverse the inbox. So the entire series has an actual cost of $O(n^2)$. But the amortized analysis techniques only apply to the first `dequeue`. After that, all the the accounts are empty (banker's method), or the potential is zero (physicist's), which means the remaining operations can't use them to pay for the expensive list reversal. The total cost of the series is therefore $O(n^2 - n)$, which is $O(n^2)$.

The problem with persistence is that it violates the assumption built-in to amortized analysis that credits (or energy units) are spent only once. Every persistent copy of the data structure instead tries to spend them itself, not being aware of all the other copies.

There are more advanced techniques for amortized analysis that can account for persistence. Those techniques are based on the idea of accumulating *debt* that is later paid off, rather than accumulating savings that are later spent. The reason that debt ends up working as an analysis technique can be summed up as: although our banks would never (financially speaking) allow us to spend money twice, they would be fine with us paying off our debt multiple times. Consult Okasaki's *Purely Functional Data Structures* to learn more.

10.3 Red-Black Trees

As we've now seen, hash tables are an efficient data structure for implementing a map ADT. They offer amortized, expected constant-time performance—which is a subtle guarantee because of those "amortized" and "expected" qualifiers we have to add. Hash tables also require mutability to implement. As functional programmers, we prefer to avoid mutability when possible.

So, let's investigate how to implement functional maps. One of the best data structures for that is the *red-black tree*, which is a kind of balanced binary search tree that offers worst-case logarithmic performance. So on one hand the performance is somewhat worse than hash tables (logarithmic vs. constant), but on the other hand we don't have to qualify the performance with words like "amortized" and "expected". Logarithmic is actually still plenty efficient for even very large workloads. And, we get to avoid mutability!

10.3.1 Binary Search Trees

A **binary search tree** (BST) is a binary tree with the following representation invariant:

> For any node *n*, every node in the left subtree of *n* has a value less than *n*'s value, and every node in the right subtree of *n* has a value greater than *n*'s value.

We call that the *BST invariant*.

Here is code that implements a couple of operations on a BST:

```
type 'a tree = Node of 'a * 'a tree * 'a tree | Leaf

(** [mem x t] is [true] iff [x] is a member of [t]. *)
let rec mem x = function
  | Leaf -> false
  | Node (y, l, r) ->
    if x < y then mem x l
    else if x > y then mem x r
    else true

(** [insert x t] is [t] . *)
let rec insert x = function
  | Leaf -> Node (x, Leaf, Leaf)
  | Node (y, l, r) as t ->
    if x < y then Node (y, insert x l, r)
    else if x > y then Node (y, l, insert x r)
    else t
```

What is the running time of those operations? Since `insert` is just a `mem` with an extra constant-time node creation, we focus on the `mem` operation.

The running time of `mem` is $O(h)$, where h is the height of the tree, because every recursive call descends one level in the tree. What's the worst-case height of a tree? It occurs with a tree of n nodes all in a single long branch—imagine adding the numbers 1,2,3,4,5,6,7 in order into the tree. So the worst-case running time of `mem` is still $O(n)$, where n is the number of nodes in the tree.

What is a good shape for a tree that would allow for fast lookup? A *perfect binary tree* has the largest number of nodes n for a given height h, which is $n = 2^{h+1} - 1$. Therefore $h = \log(n+1) - 1$, which is $O(\log n)$.

If a tree with n nodes is kept balanced, its height is $O(\log n)$, which leads to a lookup operation running in time $O(\log n)$.

How can we keep a tree balanced? It can become unbalanced during element insertion or deletion. Most balanced tree schemes involve adding or deleting an element just like in a normal binary search tree, followed by some kind of *tree surgery* to rebalance the tree. Some examples of balanced binary search tree data structures include:

- AVL trees (1962)
- 2-3 trees (1970's)
- Red-black trees (1970's)

Each of these ensures $O(\log n)$ running time by enforcing a stronger invariant on the data structure than just the binary search tree invariant.

10.3.2 Red-Black Trees

Red-black trees are relatively simple balanced binary tree data structure. The idea is to strengthen the representation invariant so a tree has height logarithmic in the number of nodes n. To help enforce the invariant, we color each node of the tree either *red* or *black*. Where it matters, we consider the color of an empty tree to be black.

```
type color = Red | Black
type 'a rbtree = Leaf | Node of color * 'a * 'a rbtree * 'a rbtree
```

Here are the new conditions we add to the binary search tree representation invariant:

1. **Local Invariant:** There are no two adjacent red nodes along any path.
2. **Global Invariant:** Every path from the root to a leaf has the same number of black nodes. This number is called the *black height* (BH) of the tree.

If a tree satisfies these two conditions, it must also be the case that every subtree of the tree also satisfies the conditions. If a subtree violated either of the conditions, the whole tree would also.

Additionally, by convention the root of the tree is colored black. This does not violate the invariants, but it also is not required by them.

With these invariants, the longest possible path from the root to an empty node would alternately contain red and black nodes; therefore it is at most twice as long as the shortest possible path, which only contains black nodes. The longest path cannot have a length greater than twice the length of the paths in a perfect binary tree, which is $O(\log n)$. Therefore, the tree has height $O(\log n)$ and the operations are all asymptotically logarithmic in the number of nodes.

How do we check for membership in red-black trees? Exactly the same way as for general binary trees.

```
let rec mem x = function
  | Leaf -> false
  | Node (_, y, l, r) ->
    if x < y then mem x l
    else if x > y then mem x r
    else true
```

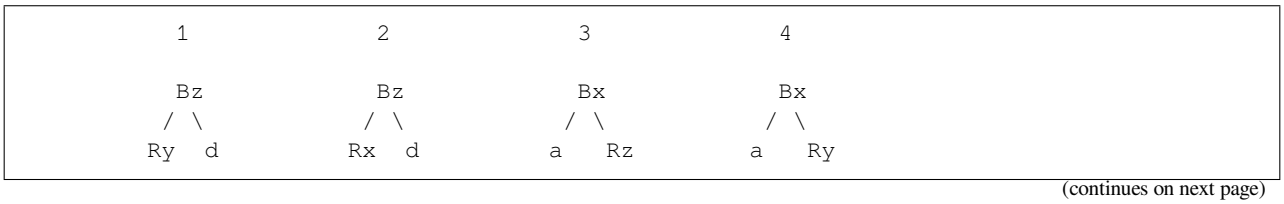
Okasaki's Algorithm. More interesting is the `insert` operation. As with standard binary trees, we add a node by replacing the leaf found by the search procedure. But what can we color that node?

- Coloring it black could increase the black height of that path, violating the Global Invariant.
- Coloring it red could make it adjacent to another red node, violating the Local Invariant.

So neither choice is safe in general. Chris Okasaki (*Purely Functional Data Structures*, 1999) gives an elegant algorithm that solves the problem by opting to violate the Local Invariant, then walk up the tree to repair the violation. Here's how it works.

We always color the new node red to ensure that the Global Invariant is preserved. However, this may destroy the Local Invariant by producing two adjacent red nodes. In order to restore the invariant, we consider not only the new red node and its red parent, but also its (black) grandparent.

The next figure shows the four possible cases that can arise. In it, `a-d` are possibly empty subtrees, and `x-z` are values stored at a node. The nodes colors are indicated with `R` and `B`.



(continues on next page)

(continued from previous page)

```
      /  \          / \              /  \          /  \
     Rx   c        a   Ry           Ry   d        b    Rz
    /  \             /  \         / \              /  \
   a    b           b    c       b   c            c    d
```

Notice that in each of these trees, we've carefully labeled the values and nodes such that the binary search tree invariant ensures the following ordering:

```
all nodes in a
 <
  x
   <
    all nodes in b
     <
      y
       <
        all nodes in c
         <
          z
           <
            all nodes in d
```

Therefore, we can transform the tree to restore the invariant locally by replacing any of the above four cases with:

```
        Ry
       /  \
     Bx    Bz
    / \   / \
   a   b c   d
```

Tip: To really understand Okasaki's algorithm, ensure that the last three diagrams make sense. The choice of which labels are placed where in the first diagram is crucial. That's what guarantees the ordering holds, hence that the final tree is the same in all four cases.

This balance function can be written simply and concisely using pattern matching, where each of the four input cases is mapped to the same output case. In addition, there is the case where the tree is left unchanged locally.

```
let balance = function
  | Black, z, Node (Red, y, Node (Red, x, a, b), c), d
  | Black, z, Node (Red, x, a, Node (Red, y, b, c)), d
  | Black, x, a, Node (Red, z, Node (Red, y, b, c), d)
  | Black, x, a, Node (Red, y, b, Node (Red, z, c, d)) ->
    Node (Red, y, Node (Black, x, a, b), Node (Black, z, c, d))
  | a, b, c, d -> Node (a, b, c, d)
```

This balancing transformation possibly breaks the Local Invariant one level up in the tree, but it can be restored again at that level in the same way, and so on up the tree. In the worst case, the process cascades all the way up to the root, resulting in two adjacent red nodes, one of them the root. But if this happens, we can just recolor the root black, which increases the black height by one. The amount of work is $O(\log n)$. The `insert` code using `balance` is as follows:

```
let insert x s =
  let rec ins = function
```

(continues on next page)

(continued from previous page)

```
    | Leaf -> Node (Red, x, Leaf, Leaf)
    | Node (color, y, a, b) as s ->
      if x < y then balance (color, y, ins a, b)
      else if x > y then balance (color, y, a, ins b)
      else s
  in
  match ins s with
  | Node (_, y, a, b) -> Node (Black, y, a, b)
  | Leaf -> (* guaranteed to be nonempty *)
    failwith "RBT insert failed with ins returning leaf"
```

The remove operation. Removing an element from a red-black tree works analogously. We start with a BST element removal and then do rebalancing. When an interior (nonleaf) node is removed, we simply splice it out if it has fewer than two nonleaf children; if it has two nonleaf children, we find the next value in the tree, which must be found inside its right child.

But, balancing the trees during removal from red-black tree requires considering more cases. Deleting a black element from the tree creates the possibility that some path in the tree has too few black nodes, breaking the Global Invariant.

Germane and Might invented an elegant algorithm to handle that rebalancing Their solution is to create "doubly-black" nodes that count twice in determining the black height. For more, read their paper: [*Deletion: The Curse of the Red-Black Tree Journal of Functional Programming*], volume 24, issue 4, July 2014.

10.3.3 Maps and Sets from BSTs

It's easy to use a BST to implement either a map or a set ADT:

- For a map, just store a binding at each node. The nodes are ordered by the keys. The values are irrelevant to the ordering.
- For a set, just store an element at each node. The nodes are ordered by the elements.

The OCaml standard library does this for the `Map` and `Set` modules. It uses a balanced BST that is a variant of an AVL tree. AVL trees are balanced BSTs in which the height of paths is allowed to vary by at most 1. The OCaml standard library modifies that to allow the height to vary by at most 2. Like red-black trees, they achieve worst-case logarithmic performance.

Now that we have a functional map data structure, how does it compare to our imperative version, the hash table?

- **Persistence:** Our red-black trees are persistent, but hash tables are ephemeral.
- **Performance:** We get guaranteed worst-case logarithmic performance with red-black trees, but amortized, expected constant-time with hash tables. That's somewhat hard to compare given all the modifiers involved. It's also an example of a general phenomenon that persistent data structures often have to pay an extra logarithmic cost over the equivalent ephemeral data structures.
- **Convenience:** We have to provide an ordering function for balanced binary trees, and a hash function for hash tables. Most libraries provide a default hash function for convenience. But the performance of the hash table does depend on that hash function truly distributing keys randomly over buckets. If it doesn't, the "expected" part of the performance guarantee for hash tables is violated. So the convenience is a double-edged sword.

There isn't a clear winner here. Since the OCaml library provides both `Map` and `Hashtbl`, you get to choose.

10.4 Sequences

A *sequence* is an infinite list. For example, the infinite list of all natural numbers would be a sequence. So would the list of all primes, or all Fibonacci numbers. How can we efficiently represent infinite lists? Obviously we can't store the whole list in memory.

We already know that OCaml allows us to create recursive functions—that is, functions defined in terms of themselves. It turns out we can define other values in terms of themselves, too.

```
let rec ones = 1 :: ones
```

```
let rec a = 0 :: b and b = 1 :: a
```

The expressions above create *recursive values*. The list `ones` contains an infinite sequence of `1`, and the lists `a` and `b` alternate infinitely between `0` and `1`. As the lists are infinite, the toplevel cannot print them in their entirety. Instead, it indicates a *cycle*: the list cycles back to its beginning. Even though these lists represent an infinite sequence of values, their representation in memory is finite: they are linked lists with back pointers that create those cycles.

Beyond sequences of numbers, there are other kinds of infinite mathematical objects we might want to represent with finite data structures:

- A stream of inputs read from a file, a network socket, or a user. All of these are unbounded in length, hence we can think of them as being infinite in length. In fact, many I/O libraries treat reaching the end of an I/O stream as an unexpected situation and raise an exception.
- A *game tree* is a tree in which the positions of a game (e.g., chess or tic-tac-toe)_ are the nodes and the edges are possible moves. For some games this tree is in fact infinite (imagine, e.g., that the pieces on the board could chase each other around forever), and for other games, it's so deep that we would never want to manifest the entire tree, hence it is effectively infinite.

10.4.1 How Not to Define a Sequence

Suppose we wanted to represent the first of those examples: the sequence of all natural numbers. Some of the obvious things we might try simply don't work:

```
(** [from n] is the infinite list [[n; n + 1; n + 2; ...]]. *)
let rec from n = n :: from (n + 1)
```

```
(** [nats] is the infinite list of natural numbers [[0; 1; ...]]. *)
let nats = from 0
```

```
Stack overflow during evaluation (looping recursion?).
```

The problem with that attempt is that `nats` attempts to compute the entire infinite sequence of natural numbers. Because the function isn't tail recursive, it quickly overflows the stack. If it were tail recursive, it would go into an infinite loop.

Here's another attempt, using what we discovered above about recursive values:

```
let rec nats = 0 :: List.map (fun x -> x + 1) nats
```

That attempt doesn't work for a more subtle reason. In the definition of a recursive value, we are not permitted to use a value before it is finished being defined. The problem is that `List.map` is applied to `nats`, and therefore pattern matches to extract the head and tail of `nats`. But we are in the middle of defining `nats`, so that use of `nats` is not permitted.

10.4.2 How to Correctly Define a Sequence

We can try to define a sequence by analogy to how we can define (finite) lists. Recall that definition:

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
```

We could try to convert that into a definition for sequences:

```
type 'a sequence = Cons of 'a * 'a sequence
```

Note that we got rid of the `Nil` constructor, because the empty list is finite, but we want only infinite lists.

The problem with that definition is that it's really no better than the built-in list in OCaml, in that we still can't define `nats`:

```
let rec from n = Cons (n, from (n + 1))
```

```
let nats = from 0
```

```
Stack overflow during evaluation (looping recursion?).
```

As before, that definition attempts to go off and compute the entire infinite sequence of naturals.

What we need is a way to *pause* evaluation, so that at any point in time, only a finite approximation to the infinite sequence has been computed. Fortunately, we already know how to do that!

Consider the following definitions:

```
let f1 = failwith "oops"
```

```
let f2 = fun x -> failwith "oops";;
```

```
f2 ();;
```

The definition of `f1` immediately raises an exception, whereas the definition of `f2` does not. Why? Because `f2` wraps the `failwith` inside an anonymous function. Recall that, according to the dynamic semantics of OCaml, **functions are already values**. So no computation is done inside the body of the function until it is applied. That's why `f2 ()` raises an exception.

We can use this property of evaluation—that functions delay evaluation—to our advantage in defining sequences: let's wrap the tail of a sequence inside a function. Since it doesn't really matter what argument that function takes, we might as well let it be unit. A function that is used just to delay computation, and in particular one that takes unit as input, is called a *thunk*.

```
(** An ['a sequence] is an infinite list of values of type ['a].
    AF: [Cons (x, f)] is the sequence whose head is [x] and tail is [f ()].
    RI: none. *)
type 'a sequence = Cons of 'a * (unit -> 'a sequence)
```

This definition turns out to work quite well. We can define `nats`, at last:

```
let rec from n = Cons (n, fun () -> from (n + 1));;
let nats = from 0;;
```

We do not get an infinite loop or a stack overflow. The evaluation of `nats` has paused. Only the first element of it, `0`, has been computed. The remaining elements will not be computed until they are requested. To do that, we can define functions to access parts of a sequence, similarly to how we can access parts of a list:

```
(** [hd s] is the head of [s] *)
let hd (Cons (h, _)) = h
```

```
(** [tl s] is the tail of [s] *)
let tl (Cons (_, t)) = t ()
```

Note how, in the definition of `tl`, we must apply the function `t` to `()` to obtain the tail of the sequence. That is, we must *force* the thunk to evaluate at that point, rather than continue to delay its computation.

For convenience, we can write functions that apply `hd` or `tl` multiple times to take or drop some finite prefix of a sequence:

```
(** [take n s] is the list of the first [n] elements of [s] *)
let rec take n s =
  if n = 0 then [] else hd s :: take (n - 1) (tl s)

(** [drop n s] is all but the first [n] elements of [s] *)
let rec drop n s =
  if n = 0 then s else drop (n - 1) (tl s)
```

For example:

```
take 10 nats
```

10.4.3 Programming with Sequences

Let's write some functions that manipulate sequences. It will help to have a notation for sequences to use as part of documentation. Let's use `<a; b; c; ...>` to denote the sequence that has elements `a`, `b`, and `c` at its head, followed by infinitely many other elements.

Here are functions to square a sequence, and to sum two sequences:

```
(** [square <a; b; c; ...>] is [<a * a; b * b; c * c; ...]. *)
let rec square (Cons (h, t)) =
  Cons (h * h, fun () -> square (t ()))

(** [sum <a1; a2; a3; ...> <b1; b2; b3; ...>] is
    [<a1 + b1; a2 + b2; a3 + b3; ...>] *)
let rec sum (Cons (h1, t1)) (Cons (h2, t2)) =
  Cons (h1 + h2, fun () -> sum (t1 ()) (t2 ()))
```

Note how the basic template for defining both functions is the same:

- Pattern match against the input sequence(s), which must be `Cons` of a head and a tail function (a thunk).
- Construct a sequence as the output, which must be `Cons` of a new head and a new tail function (a thunk).
- In constructing the new tail function, delay the evaluation of the tail by immediately starting with `fun () -> ...`.
- Inside the body of that thunk, recursively apply the function being defined (square or sum) to the result of forcing a thunk (or thunks) to evaluate.

Of course, squaring and summing are just two possible ways of mapping a function across a sequence or sequences. That suggests we could write a higher-order map function, much like for lists:

```
(** [map f <a; b; c; ...>] is [<f a; f b; f c; ...>] *)
let rec map f (Cons (h, t)) =
  Cons (f h, fun () -> map f (t ()))

(** [map2 f <a1; b1; c1;...> <a2; b2; c2; ...>] is
    [<f a1 b1; f a2 b2; f a3 b3; ...>] *)
let rec map2 f (Cons (h1, t1)) (Cons (h2, t2)) =
  Cons (f h1 h2, fun () -> map2 f (t1 ()) (t2 ()))

let square' = map (fun n -> n * n)
let sum' = map2 ( + )
```

Now that we have a map function for sequences, we can successfully define `nats` in one of the clever ways we originally attempted:

```
let rec nats = Cons (0, fun () -> map (fun x -> x + 1) nats)
```

```
take 10 nats
```

Why does this work? Intuitively, `nats` is `<0; 1; 2; 3; ...>`, so mapping the increment function over `nats` is `<1; 2; 3; 4; ...>`. If we cons `0` onto the beginning of `<1; 2; 3; 4; ...>`, we get `<0; 1; 2; 3; ...>`, as desired. The recursive value definition is permitted, because we never attempt to use `nats` until after its definition is finished. In particular, the thunk delays `nats` from being evaluated on the right-hand side of the definition.

Here's another clever definition. Consider the Fibonacci sequence `<1; 1; 2; 3; 5; 8; ...>`. If we take the tail of it, we get `<1; 2; 3; 5; 8; 13; ...>`. If we sum those two sequences, we get `<2; 3; 5; 8; 13; 21; ...>`. That's nothing other than the tail of the tail of the Fibonacci sequence. So if we were to prepend `[1; 1]` to it, we'd have the actual Fibonacci sequence. That's the intuition behind this definition:

```
let rec fibs =
  Cons (1, fun () ->
    Cons (1, fun () ->
      sum fibs (tl fibs)))
```

And it works!

```
take 10 fibs
```

Unfortunately, it's highly inefficient. Every time we force the computation of the next element, it required recomputing all the previous elements, twice: once for `fibs` and once for `tl fibs` in the last line of the definition. Try running the code yourself. By the time we get up to the 30th number, the computation is noticeably slow; by the time of the 100th, it seems to last forever.

Could we do better? Yes, with a little help from a new language feature: laziness. We discuss it, next.

10.4.4 Laziness

The example with the Fibonacci sequence demonstrates that it would be useful if the computation of a thunk happened only once: when it is forced, the resulting value could be remembered, and if the thunk is ever forced again, that value could immediately be returned instead of recomputing it. That's the idea behind the OCaml `Lazy` module:

```
module Lazy :
  sig
    type 'a t = 'a lazy_t
    val force : 'a t -> 'a
    ...
  end
```

A value of type `'a Lazy.t` is a value of type `'a` whose computation has been delayed. Intuitively, the language is being *lazy* about evaluating it: it won't be computed until specifically demanded. The way that demand is expressed with by *forcing* the evaluation with `Lazy.force`, which takes the `'a Lazy.t` and causes the `'a` inside it to finally be produced. The first time a lazy value is forced, the computation might take a long time. But the result is *cached* aka *memoized*, and any subsequent time that lazy value is forced, the memoized result will be returned immediately without recomputing it.

Note: "Memoized" really is the correct spelling of this term. We didn't misspell "memorized", though it might look that way.

The `Lazy` module doesn't contain a function that produces a `'a Lazy.t`. Instead, there is a keyword built-in to the OCaml syntax that does it: `lazy e`.

- **Syntax:** `lazy e`
- **Static semantics:** If `e : u`, then `lazy e : u Lazy.t`.
- **Dynamic semantics:** `lazy e` does not evaluate `e` to a value. Instead it produces a *suspension* that, when later forced, will evaluate `e` to a value `v` and return `v`. Moreover, that suspension remembers that `v` is its forced value. And if the suspension is ever forced again, it immediately returns `v` instead of recomputing it.

Note: OCaml's usual evaluation strategy is *eager* aka *strict*: it always evaluate an argument before function application. If you want a value to be computed lazily, you must specifically request that with the `lazy` keyword. Other function languages, notably Haskell, are lazy by default. Laziness can be pleasant when programming with infinite data structures. But lazy evaluation makes it harder to reason about space and time, and it has unpleasant interactions with side effects.

To illustrate the use of lazy values, let's try computing the 30th Fibonacci number using this definition of `fibs`:

```
let rec fibs =
  Cons (1, fun () ->
    Cons (1, fun () ->
      sum fibs (tl fibs)))
```

Tip: These next few examples will make much more sense if you run them interactively, rather than just reading this page.

If we try to get the 30th Fibonacci number, it will take a long time to compute:

```
let fib30long = take 30 fibs |> List.rev |> List.hd
```

But if we wrap evaluation of that with `lazy`, it will return immediately, because the evaluation of that number has been suspended:

```
let fib30lazy = lazy (take 30 fibs |> List.rev |> List.hd)
```

Later on we could force the evaluation of that lazy value, and that will take a long time to compute, as did `fib30long`:

```
let fib30 = Lazy.force fib30lazy
```

But if we ever try to recompute that same lazy value, it will return immediately, because the result has been memoized:

```
let fib30fast = Lazy.force fib30lazy
```

Nonetheless, we still haven't totally succeeded. That particular computation of the 30th Fibonacci number has been memoized, but if we later define some other computation of another it won't be sped up the first time it's computed:

```
let fib29 = take 29 fibs |> List.rev |> List.hd
```

What we really want is to change the representation of sequences itself to make use of lazy values.

Lazy Sequences

Here's a representation for infinite lists using lazy values:

```
type 'a lazysequence = Cons of 'a * 'a lazysequence Lazy.t
```

We've gotten rid of the thunk, and instead are using a lazy value as the tail of the lazy sequence. If we ever want that tail to be computed, we force it.

For sake of comparison, the following two modules implement the Fibonacci sequence with sequences, then with lazy sequences. Try computing the 30th Fibonacci number with both modules, and you'll see that the lazy-sequence implementation is much faster than the standard-sequence implementation.

```
module SequenceFibs = struct
  type 'a sequence = Cons of 'a * (unit -> 'a sequence)

  let hd : 'a sequence -> 'a =
    fun (Cons (h, _)) -> h

  let tl : 'a sequence -> 'a sequence =
    fun (Cons (_, t)) -> t ()

  let rec take_aux n (Cons (h, t)) lst =
    if n = 0 then lst
    else take_aux (n - 1) (t ()) (h :: lst)

  let take : int -> 'a sequence -> 'a list =
    fun n s -> List.rev (take_aux n s [])

  let nth : int -> 'a sequence -> 'a =
    fun n s -> List.hd (take_aux (n + 1) s [])
```

(continues on next page)

(continued from previous page)

```
  let rec sum : int sequence -> int sequence -> int sequence =
    fun (Cons (h_a, t_a)) (Cons (h_b, t_b)) ->
      Cons (h_a + h_b, fun () -> sum (t_a ()) (t_b ()))

  let rec fibs =
    Cons(1, fun () ->
      Cons(1, fun () ->
        sum (tl fibs) fibs))

  let nth_fib n =
    nth n fibs

end

module LazyFibs = struct

  type 'a lazysequence = Cons of 'a * 'a lazysequence Lazy.t

  let hd : 'a lazysequence -> 'a =
    fun (Cons (h, _)) -> h

  let tl : 'a lazysequence -> 'a lazysequence =
    fun (Cons (_, t)) -> Lazy.force t

  let rec take_aux n (Cons (h, t)) lst =
    if n = 0 then lst else
      take_aux (n - 1) (Lazy.force t) (h :: lst)

  let take : int -> 'a lazysequence -> 'a list =
    fun n s -> List.rev (take_aux n s [])

  let nth : int -> 'a lazysequence -> 'a =
    fun n s -> List.hd (take_aux (n + 1) s [])

  let rec sum : int lazysequence -> int lazysequence -> int lazysequence =
    fun (Cons (h_a, t_a)) (Cons (h_b, t_b)) ->
      Cons (h_a + h_b, lazy (sum (Lazy.force t_a) (Lazy.force t_b)))

  let rec fibs =
    Cons(1, lazy (
      Cons(1, lazy (
        sum (tl fibs) fibs))))

  let nth_fib n =
    nth n fibs
end
```

10.5 Memoization

In the previous section, we saw that the `Lazy` module memoizes the results of computations, so that no time has to be wasted on recomputing them. Memoization is a powerful technique for asymptotically speeding up simple recursive algorithms, without having to change the way the algorithm works.

Let's see apply the Abstraction Principle and invent a way to memoize *any* function, so that the function only had to be evaluated once on any given input. We'll end up using imperative data structures (arrays and hash tables) as part of our solution.

10.5.1 Fibonacci

Let's again consider the problem of computing the nth Fibonacci number. The naive recursive implementation takes exponential time, because of the recomputation of the same Fibonacci numbers over and over again:

```
let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)
```

Note: To be precise, its running time turns out to be $O(\phi^n)$, where ϕ is the golden ratio, $\frac{1+\sqrt{5}}{2}$.

If we record Fibonacci numbers as they are computed, we can avoid this redundant work. The idea is that whenever we compute `f n`, we store it in a table indexed by `n`. In this case the indexing keys are integers, so we can use implement this table using an array:

```
let fibm n =
  let memo : int option array = Array.make (n + 1) None in
  let rec f_mem n =
    match memo.(n) with
    | Some result -> (* computed already *) result
    | None ->
        let result =
          if n < 2 then 1 else f_mem (n - 1) + f_mem (n - 2)
        in
        (* record in table *)
        memo.(n) <- Some result;
        result
  in
  f_mem n
```

The function `f_mem` defined inside `fibm` contains the original recursive algorithm, except before doing that calculation it first checks if the result has already been computed and stored in the table in which case it simply returns the result.

How do we analyze the running time of this function? The time spent in a single call to `f_mem` is $O(1)$ if we exclude the time spent in any recursive calls that it happens to make. Now we look for a way to bound the total number of recursive calls by finding some measure of the progress that is being made.

A good choice of progress measure, not only here but also for many uses of memoization, is the number of nonempty entries in the table (i.e. entries that contain `Some n` rather than `None`). Each time `f_mem` makes the two recursive calls it also increases the number of nonempty entries by one (filling in a formerly empty entry in the table with a new value). Since the table has only `n` entries, there can thus only be a total of $O(n)$ calls to `f_mem`, for a total running time of $O(n)$ (because we established above that each call takes $O(1)$ time). This speedup from memoization thus reduces the running time from exponential to linear, a huge change—e.g., for $n = 4$ the speedup from memoization is more than a factor of a million!

The key to being able to apply memoization is that there are common sub-problems which are being solved repeatedly. Thus we are able to use some extra storage to save on repeated computation.

Although this code uses imperative constructs (specifically, array update), the side effects are not visible outside the function `fibm`. So from a client's perspective, `fibm` is functional. There's not need to mention the imperative implementation (i.e., the benign side effects) that are used internally.

10.5.2 Memoization Using Higher-order Functions

Now that we've seen an example of memoizing one function, let's use higher-order functions to memoize any function. First, consider the case of memoizing a non-recursive function `f`. In that case we simply need to create a hash table that stores the corresponding value for each argument that `f` is called with (and to memoize multi-argument functions we can use currying and uncurrying to convert to a single argument function).

```
let memo f =
  let h = Hashtbl.create 11 in
  fun x ->
    try Hashtbl.find h x
    with Not_found ->
      let y = f x in
      Hashtbl.add h x y;
      y
```

For recursive functions, however, the recursive call structure needs to be modified. This can be abstracted out independent of the function that is being memoized:

```
let memo_rec f =
  let h = Hashtbl.create 16 in
  let rec g x =
    try Hashtbl.find h x
    with Not_found ->
      let y = f g x in
      Hashtbl.add h x y;
      y
  in
  g
```

Now we can slightly rewrite the original `fib` function above using this general memoization technique:

```
let fib_memo =
  let rec fib self n =
    if n < 2 then 1 else self (n - 1) + self (n - 2)
  in
  memo_rec fib
```

10.5.3 Just for Fun: Party Optimization

Suppose we want to throw a party for a company whose org chart is a binary tree. Each employee has an associated "fun value" and we want the set of invited employees to have a maximum total fun value. However, no employee is fun if his superior is invited, so we never invite two employees who are connected in the org chart. (The less fun name for this problem is the maximum weight independent set in a tree.) There are $2n$ possible invitation lists, so the naive algorithm that compares the fun of every invitation list takes exponential time.

We can use memoization to turn this into a linear-time algorithm. We start by defining a variant type to represent the employees. The int at each node is the fun.

```
type tree = Empty | Node of int * tree * tree
```

Now, how can we solve this recursively? One important observation is that in any tree, the optimal invitation list that doesn't include the root node will be the union of optimal invitation lists for the left and right subtrees. And the optimal invitation list that does include the root node will be the union of optimal invitation lists for the left and right children that do not include their respective root nodes. So it seems useful to have functions that optimize the invite lists for the case where the root node is required to be invited, and for the case where the root node is excluded. We'll call these two functions party_in and party_out. Then the result of party is just the maximum of these two functions:

```
module Unmemoized = struct
  type tree =
    | Empty
    | Node of int * tree * tree

  (* Returns optimum fun for t. *)
  let rec party t = max (party_in t) (party_out t)

  (* Returns optimum fun for t assuming the root node of t
   * is included. *)
  and party_in t =
    match t with
    | Empty -> 0
    | Node (v, left, right) -> v + party_out left + party_out right

  (* Returns optimum fun for t assuming the root node of t
   * is excluded. *)
  and party_out t =
    match t with
    | Empty -> 0
    | Node (v, left, right) -> party left + party right
end
```

This code has exponential running time. But notice that there are only n possible distinct calls to party. If we change the code to memoize the results of these calls, the performance will be linear in n. Here is a version that memoizes the result of party and also computes the actual invitation lists. Notice that this code memoizes results directly in the tree.

```
module Memoized = struct
  (* This version memoizes the optimal fun value for each tree node. It
     also remembers the best invite list. Each tree node has the name of
     the employee as a string. *)
  type tree =
    | Empty
    | Node of
        int * string * tree * tree * (int * string list) option ref
```

(continues on next page)

(continued from previous page)

```
  let rec party t : int * string list =
    match t with
    | Empty -> (0, [])
    | Node (v, name, left, right, memo) -> (
        match !memo with
        | Some result -> result
        | None ->
            let infun, innames = party_in t in
            let outfun, outnames = party_out t in
            let result =
              if infun > outfun then (v + infun, name :: innames)
              else (outfun, outnames)
            in
            memo := Some result;
            result)

  and party_in t =
    match t with
    | Empty -> (0, [])
    | Node (v, name, l, r, _) ->
        let lfun, lnames = party_out l and rfun, rnames = party_out r in
        (v + lfun + rfun, name :: lnames @ rnames)

  and party_out t =
    match t with
    | Empty -> (0, [])
    | Node (v, _, l, r, _) ->
        let lfun, lnames = party l and rfun, rnames = party r in
        (lfun + rfun, lnames @ rnames)
end
```

Why was memoization so effective for solving this problem? As with the Fibonacci algorithm, we had the overlapping sub-problems property, in which the naive recursive implementation called the function party many times with the same arguments. Memoization saves all those calls. Further, the party optimization problem has the property of optimal substructure, meaning that the optimal answer to a problem is computed from optimal answers to sub-problems. Not all optimization problems have this property. The key to using memoization effectively for optimization problems is to figure out how to write a recursive function that implements the algorithm and has two properties. Sometimes this requires thinking carefully.

10.6 Promises

So far we have only considered *sequential* programs. Execution of a sequential program proceeds one step at a time, with no choice about which step to take next. Sequential programs are limited in that they are not very good at dealing with multiple sources of simultaneous input and they can only execute on a single processor. Many modern applications are instead *concurrent*.

10.6.1 Concurrency

Concurrent programs enable computations to overlap in duration, instead of being forced to happen sequentially.

- *Graphical user interfaces* (GUIs), for example, rely on concurrency to keep the interface responsive while computation continues in the background. Without concurrency, a GUI would "lock up" until the current action is completed. Sometimes, because of concurrency bugs, that happens anyway—and it's frustrating for the user!
- A spreadsheet needs concurrency to re-compute all the cells while still keeping the menus and editing capabilities available for the user.
- A web browser needs concurrency to read and render web pages incrementally as new data comes in over the network, to run JavaScript programs embedded in the web page, and to enable the user to navigate through the page and click on hyperlinks.

Servers are another example of applications that need concurrency. A web server needs to respond to many requests from clients, and clients would prefer not to wait. If an assignment is released in CMS, for example, you would prefer to be able to view that assignment at the same time as everyone else in the class, rather than having to "take a number" a wait for your number to be called—as at the Department of Motor Vehicles, or at an old-fashioned deli, etc.

One of the primary jobs of an *operating system* (OS) is to provide concurrency. The OS makes it possible for many applications to be executing concurrently: a music player, a web browser, a code editor, etc. How does it do that? There are two fundamental, complementary approaches:

- **Interleaving:** rapidly switch back and forth between computations. For example, execute the music player for 100 milliseconds, then the browser, then the editor, then repeat. That makes it appear as though multiple computations are occurring simultaneously, but in reality, only one is ever occurring at the same time.
- **Parallelism:** use hardware that is capable of performing two or more computations literally at the same time. Many processors these days are *multicore*, meaning that they have multiple central processing units (CPUs), each of which can be executing a program simultaneously.

Regardless of the approaches being used, concurrent programming is challenging. Even if there are multiple cores available for simultaneous use, there are still many other resources that must be shared: memory, the screen, the network interface, etc. Managing that sharing, especially without introducing bugs, is quite difficult. For example, if two programs want to communicate by using the computer's memory, there needs to be some agreement on when each program is allowed to read and write from the memory. Otherwise, for example, both programs might attempt to write to the same location in memory, leading to corrupted data. Those kinds of *race conditions*, where a program races to complete its operations before another program, are notoriously difficult to avoid.

The most fundamental challenge is that concurrency makes the execution of a program become *nondeterministic:* the order in which operations occur cannot necessarily be known ahead of time. Race conditions are an example of nondeterminism. To program correctly in the face of nondeterminism, the programmer is forced to think about *all* possible orders in which operations might execute, and ensure that in *all* of them the program works correctly.

Purely functional programs make nondeterminism easier to reason about, because evaluation of an expression always returns the same value no matter what. For example, in the expression `(2 * 4) + (3 * 5)`, the operations can be executed concurrently (e.g., with the left and right products evaluated simultaneously) without changing the answer. Imperative programming is more problematic. For example, the expressions `!x` and `incr x; !x`, if executed concurrently, could give different results depending on which executes first.

10.6.2 Threads

To make concurrent programming easier, computer scientists have invented many abstractions. One of the best known is *threads*. Abstractly, a thread is a single sequential computation. There can be many threads running at a time, either interleaved or in parallel depending on the hardware, and a *scheduler* handles choosing which threads are running at any given time. Scheduling can either be *preemptive*, meaning that the scheduler is permitted to stop a thread and restart it later without the thread getting a choice in the matter, or *cooperative*, meaning that the thread must choose to relinquish control back to the scheduler. The former can lead to race conditions, and the latter can lead to unresponsive applications.

Concretely, a thread is a set of values that are loaded into the registers of a processor. Those values tell the processor where to find the next instruction to execute, where its stack and heap are located in memory, etc. To implement preemption, a scheduler sets a timer in the hardware; when the timer goes off, the current thread is interrupted and the scheduler gets to run. CS 3410 and 4410 cover those concepts in detail.

10.6.3 Promises

In the functional programming paradigm, one of the best known abstractions for concurrency is *promises*. Other names for this idea include *futures*, *deferreds*, and *delayeds*. All those names refer to the idea of a computation that is not yet finished: it has promised to eventually produce a value in the future, but the completion of the computation has been deferred or delayed. There may be many such values being computed concurrently, and when the value is finally available, there may be computations ready to execute that depend on the value.

This idea has been widely adopted in many languages and libraries, including Java, JavaScript, and .NET. Indeed, modern JavaScript adds an `async` keyword that causes a function to return a promise, and an `await` keyword that waits for a promise to finish computing. There are two widely-used libraries in OCaml that implement promises: Async and Lwt. Async is developed by Jane Street. Lwt is part of the Ocsigen project, which is a web framework for OCaml.

We now take a deeper look at promises in Lwt. The name of the library was an acronym for "light-weight threads." But that was a misnomer, as the Github page admits (as of 10/22/18):

> Much of the current manual refers to … "lightweight threads" or just "threads." This will be fixed in the new manual. [Lwt implements] promises, and has nothing to do with system or preemptive threads.

So don't think of Lwt as having anything to do with threads: it really is a library for promises.

In Lwt, a *promise* is a write-once reference: a value that is permitted to mutate at most once. When created, it is like an empty box that contains nothing. We say that the promise is *pending*. Eventually the promise can be *resolved*, which is like putting something inside the box. Instead of being resolved, the promise can instead be *rejected*, in which case the box is filled with an exception. Regardless of whether the promise is resolved or rejected, once the box is filled, its contents may never change.

For now, we will mostly forget about concurrency. Later we'll come back and add incorporate it. But there is one part of the design for concurrency that we need to address now. When we later start using functions for OS-provided concurrency, such as concurrent reads and writes from files, there will need to be a division of responsibilities:

- The client code that wants to make use of concurrency will need to *access* promises: query whether they are resolved or pending, and make use of the resolved values.
- The library and OS code that implements concurrency will need to *mutate* the promise—that is, to actually resolve or reject it. Client code does not need that ability.

We therefore will introduce one additional abstraction called a *resolver*. There will be a one-to-one association between promises and resolvers. The resolver for a promise will be used internally by the concurrency library but not revealed to clients. The clients will only get access to the promise.

For example, suppose the concurrency library supported a operation to concurrently read a string from the network. The library would implement that operation as follows:

- Create a new promise and its associated resolver. The promise is pending.

- Call an OS function that will concurrently read the string then invoke the resolver on that string.
- Return the promise (but not resolver) to the client. The OS meanwhile continues to work on reading the string.

You might think of the resolver as being a "private and writeable" value used primarily by the library and the promise as being a "public and read-only" value used primarily by the client.

10.6.4 Making Our Own Promises

Here is an interface for our own Lwt-style promises. The names have been changed to make the interface clearer.

```
(** A signature for Lwt-style promises, with better names *)
module type PROMISE = sig
  type 'a state =
    | Pending
    | Resolved of 'a
    | Rejected of exn

  type 'a promise

  type 'a resolver

  (** [make ()] is a new promise and resolver. The promise is pending. *)
  val make : unit -> 'a promise * 'a resolver

  (** [return x] is a new promise that is already resolved with value
      [x]. *)
  val return : 'a -> 'a promise

  (** [state p] is the state of the promise *)
  val state : 'a promise -> 'a state

  (** [resolve r x] resolves the promise [p] associated with [r] with
      value [x], meaning that [state p] will become [Resolved x].
      Requires: [p] is pending. *)
  val resolve : 'a resolver -> 'a -> unit

  (** [reject r x] rejects the promise [p] associated with [r] with
      exception [x], meaning that [state p] will become [Rejected x].
      Requires: [p] is pending. *)
  val reject : 'a resolver -> exn -> unit
end
```

To implement that interface, we can make the representation type of of `'a promise` be a reference to a state:

```
type 'a state = Pending | Resolved of 'a | Rejected of exn
type 'a promise = 'a state ref
```

That way it's possible to mutate the contents of the promise.

For the representation type of the resolver, we'll do something a little clever. It will simply be the same as a promise.

```
type 'a resolver = 'a promise
```

So internally, the two types are exactly the same. But externally no client of the `Promise` module will be able to distinguish them. In other words, we're using the type system to control whether it's possible to apply certain functions (e.g., `state` vs `resolve`) to a promise.

To help implement the rest of the functions, let's start by writing a helper function `update : 'a promise -> 'a state -> unit` to update the reference. This function will implement changing the state of the promise from pending to either resolved or rejected, and once the state has changed, it will not allow it to be changed again. In other words, `update` enforces the "write once" invariant.

```
(** [write_once p s] changes the state of [p] to be [s].  If [p] and [s]
    are both pending, that has no effect.
    Raises: [Invalid_arg] if the state of [p] is not pending. *)
let write_once p s =
  if !p = Pending
  then p := s
  else invalid_arg "cannot write twice"
```

Using that helper, we can implement the `make` function:

```
let make () =
  let p = ref Pending in
  p, p
```

The remaining functions in the interface are trivial to implement. Putting it altogether in a module, we have:

```
module Promise : PROMISE = struct
  type 'a state =
    | Pending
    | Resolved of 'a
    | Rejected of exn

  type 'a promise = 'a state ref

  type 'a resolver = 'a promise

  (** [write_once p s] changes the state of [p] to be [s]. If [p] and
      [s] are both pending, that has no effect. Raises: [Invalid_arg] if
      the state of [p] is not pending. *)
  let write_once p s =
    if !p = Pending then p := s else invalid_arg "cannot write twice"

  let make () =
    let p = ref Pending in
    (p, p)

  let return x = ref (Resolved x)

  let state p = !p

  let resolve r x = write_once r (Resolved x)

  let reject r x = write_once r (Rejected x)
end
```

10.6.5 Lwt Promises

The types and names used in Lwt are a bit more obscure than those we used above. Lwt uses analogical terminology that comes from threads—but since Lwt does not actually implement threads, that terminology is not necessarily helpful. (We don't mean to demean Lwt! It is a library that has been developing and changing over time.)

The Lwt interface includes the following declarations, which we have annotated with comments to compare them to the interface we implemented above:

```
module type Lwt = sig
  (* [Sleep] means pending.  [Return] means resolved.
     [Fail] means rejected. *)
  type 'a state = Sleep | Return of 'a | Fail of exn

  (* a [t] is a promise *)
  type 'a t

  (* a [u] is a resolver *)
  type 'a u

  val state : 'a t -> 'a state

  (* [wakeup] means [resolve] *)
  val wakeup : 'a u -> 'a -> unit

  (* [wakeup_exn] means [reject] *)
  val wakeup_exn : 'a u -> exn -> unit

  (* [wait] means [make] *)
  val wait : unit -> 'a t * 'a u

  val return : 'a -> 'a t
end
```

Lwt's implementation of that interface is much more complex than our own implementation above, because Lwt actually supports many more operations on promises. Nonetheless, the core ideas that we developed above provide sound intuition for what Lwt implements.

Here is some example Lwt code that you can try out in utop:

```
#require "lwt";;
let p, r = Lwt.wait();;
```

To avoid those weak type variables, we can provide a further hint to OCaml as to what type we want to eventually put into the promise. For example, if we wanted to have a promise that will eventually contain an `int`, we could write this code:

```
let (p : int Lwt.t), r = Lwt.wait ()
```

Now we can resolve the promise:

```
Lwt.state p
```

```
Lwt.wakeup r 42
```

```
Lwt.state p;;
```

```
Lwt.wakeup r 42
```

That last exception was raised because we attempted to resolve the promise a second time, which is not permitted.

To reject a promise, we can write similar code:

```
let (p : int Lwt.t), r = Lwt.wait ();;
Lwt.wakeup_exn r (Failure "nope");;
Lwt.state p;;
```

Note that nothing we have implemented so far does anything concurrently. The promise abstraction by itself is not inherently concurrent. It's just a data structure that can be written at most once, and that provides a means to control who can write to it (through the resolver).

10.6.6 Asynchronous I/O

Now that we understand promises as a data abstraction, let's turn to how they can be used for concurrency. The typical way they're used with Lwt is for concurrent input and output (I/O).

The I/O functions that are part of the OCaml standard library are *synchronous* aka *blocking*: when you call such a function, it does not return until the I/O has been completed. "Synchronous" here refers to the synchronization between your code and the I/O function: your code does not get to execute again until the I/O code is done. "Blocking" refers to the fact that your code has to wait—it is blocked—until the I/O completes.

For example, the `Stdlib.input_line : in_channel -> string` function reads characters from an *input channel* until it reaches a newline character, then returns the characters it read. The type `in_channel` is abstract; it represents a source of data that can be read, such as a file, or the network, or the keyboard. The value `Stdlib.stdin : in_channel` represents the *standard input* channel, which is the channel which usually, by default, provides keyboard input.

If you run the following code in utop, you will observe the blocking behavior:

```
# ignore(input_line stdin); print_endline "done";;
<type your own input here>
done
- : unit = ()
```

The string `"done"` is not printed until after the input operation completes, which happens after you type Enter.

Synchronous I/O makes it impossible for a program to carry on other computations while it is waiting for the I/O operation to complete. For some programs that's just fine. A text adventure game, for example, doesn't have any background computations it needs to perform. But other programs, like spreadsheets or servers, would be improved by being able to carry on computations in the background rather than having to completely block while waiting for input.

Asynchronous aka *non-blocking* I/O is the opposite style of I/O. Asynchronous I/O operations return immediately, regardless of whether the input or output has been completed. That enables a program to launch an I/O operation, carry on doing other computations, and later come back to make use of the completed operation.

The Lwt library provides its own I/O functions in the `Lwt_io` module, which is in the `lwt.unix` package. The function `Lwt_io.read_line : Lwt_io.input_channel -> string Lwt.t` is the asynchronous equivalent of `Stdlib.input_line`. Similarly, `Lwt_io.input_channel` is the equivalent of the OCaml standard library's `in_channel`, and `Lwt_io.stdin` represents the standard input channel.

Run this code in utop to observe the non-blocking behavior:

```
# #require "lwt.unix";;
# open Lwt_io;;
# ignore(read_line stdin); printl "done";;
done
- : unit = ()
# <type your own input here>
```

The string `"done"` is printed immediately by `Lwt_io.printl`, which is Lwt's equivalent of `Stdlib.print_endline`, before you even type. Note that it's best to use just one library's I/O functions, rather than mix them together.

When you do type your input, you don't see it echoed to the screen, because it's happening in the background. Utop is still executing—it is not blocked—but your input is being sent to that `read_line` function instead of to utop. When you finally type Enter, the input operation completes, and you are back to interacting with utop.

Now imagine that instead of reading a line asynchronously, the program was a web server reading a file to be served to a client. And instead of printing a string, the server was delivering the contents of a different file that had completed reading to a different client. That's why asynchronous I/O can be so useful: it helps to *hide latency*. Here, "latency" means waiting for data to be transfered from one place to another, e.g., from disk to memory. Latency hiding is an excellent use for concurrency.

Note that all the concurrency here is really coming from the operating system, which is what provides the underlying asynchronous I/O infrastructure. Lwt is just exposing that infrastructure to you through a library.

10.6.7 Promises and Asynchronous I/O

The output type of `Lwt_io.read_line` is `string Lwt.t`, meaning that the function returns a `string` promise. Let's investigate how the state of that promise evolves.

When the promise is returned from `read_line`, it is pending:

```
# let p = read_line stdin in Lwt.state p;;
- : string Lwt.state = Lwt.Sleep
# <now you have to type input and Enter to regain control of utop>
```

When the Enter key is pressed and input is completed, the promise returned from `read_line` should become resolved. For example, suppose you enter "Camels are bae":

```
# let p = read_line stdin;;
val p : string Lwt.t = <abstr>
<now you type Camels are bae followed by Enter>
# p;;
- : string = "Camels are bae"
```

But, if you study that output carefully, you'll notice something very strange just happened! After the `let` statement, `p` had type `string Lwt.t`, as expected. But when we evaluated `p`, it came back as type `string`. It's as if the promise disappeared.

What's actually happening is that utop has some special—and potentially confusing—functionality built into it that is related to Lwt. Specifically, whenever you try to directly evaluate a promise at the top level, *utop will give you the contents of the promise, rather than the promise itself, and if the promise is not yet resolved, utop will block until the promise becomes resolved so that the contents can be returned.*

So the output `- : string = "Camels are bae"` really means that `p` contains a resolved `string` whose value is `"Camels are bae"`, not that `p` itself is a `string`. Indeed, the `#show_val` directive will show us that `p` is a promise:

```
# #show_val p;;
val p : string Lwt.t
```

To disable that feature of utop, or to reenable it, call the function `UTop.set_auto_run_lwt : bool -> unit`, which changes how utop evaluates Lwt promises at the top level. You can see the behavior change in the following code:

```
# UTop.set_auto_run_lwt false;;
- : unit = ()
<now you type Camels are bae followed by Enter>
# p;;
- : string Lwt.state = <abstr>
# Lwt.state p;;
- : string Lwt.state = Lwt.Return "Camels are bae"
```

If you re-enable this "auto run" feature, and directly try to evaluate the promise returned by `read_line`, you'll see that it behaves exactly like synchronous I/O, i.e., `Stdlib.input_line`:

```
# UTop.set_auto_run_lwt true;;
- : unit = ()
# read_line stdin;;
Camels are bae
- : string = "Camels are bae"
```

Because of the potential confusion, we will henceforth assume that auto running is disabled. A good way to make that happen is to put the following line in your `.ocamlinit` file:

```
UTop.set_auto_run_lwt false;;
```

10.6.8 Callbacks

For a program to benefit from the concurrency provided by asynchronous I/O and promises, there needs to be a way for the program to make use of resolved promises. For example, if a web server is asynchronously reading and serving multiple files to multiple clients, the server needs a way to (i) become aware that a read has completed, and (ii) then do a new asynchronous write with the result of the read. In other words, programs need a mechanism for managing the dependencies among promises.

The mechanism provided in Lwt is named *callbacks*. A callback is a function that will be run sometime after a promise has been resolved, and it will receive as input the contents of the resolved promise. Think of it like asking your friend to do some work for you: they promise to do it, and to call you back on the phone with the result of the work sometime after they've finished.

Registering a callback. Here is a function that prints a string using Lwt's version of the `printf` function:

```
let print_the_string str = Lwt_io.printf "The string is: %S\n" str
```

And here, repeated from the previous section, is our code that returns a promise for a string read from standard input:

```
let p = read_line stdin
```

To register the printing function as a callback for that promise, we use the function `Lwt.bind`, which *binds* the callback to the promise:

```
Lwt.bind p print_the_string
```

Sometime after p is resolved, hence contains a string, the callback function will be run with that string as its input. That causes the string to be printed.

Here's a complete utop transcript as an example of that:

```
# let print_the_string str = Lwt_io.printf "The string is: %S\n" str;;
val print_the_string : string -> unit Lwt.t = <fun>
# let p = read_line stdin in Lwt.bind p print_the_string;;
- : unit Lwt.t = <abstr>
  <type Camels are bae followed by Enter>
# The string is: "Camels are bae"
```

Bind. The type of Lwt.bind is important to understand:

```
'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t
```

The bind function takes a promise as its first argument. It doesn't matter whether that promise has been resolved yet or not. As its second argument, bind takes a callback function. That callback takes an input which is the same type 'a as the contents of the promise. It's not an accident that they have the same type: the whole idea is to eventually run the callback on the resolved promise, so the type the promise contains needs to be the same as the type the callback expects as input.

After being invoked on a promise and callback, e.g., bind p c, the bind function does one of three things, depending on the state of p:

- If p is already resolved, then c is run immediately on the contents of p. The promise that is returned might or might not be pending, depending on what c does.
- If p is already rejected, then c does not run. The promise that is returned is also rejected, with the same exception as p.
- If p is pending, then bind does not wait for p to be resolved, nor for c to be run. Rather, bind just registers the callback to eventually be run when (or if) the promise is resolved. Therefore the bind function returns a new promise. That promise will become resolved when (or if) the callback completes running, sometime in the future. Its contents will be whatever contents are contained within the promise that the callback itself returns.

Note: For the first case above: The Lwt source code claims that this behavior might change in a later version: under high load, c might be registered to run later. But as of v4.1.0 that behavior has not yet been activated. So, don't worry about it—this paragraph is just here to future-proof this discussion.

Let's consider that final case in more detail. We have one promise of type 'a Lwt.t and two promises of type 'b Lwt.t:

- The promise of type 'a Lwt.t, call it promise X, is an input to bind. It was pending when bind was called, and when bind returns.
- The first promise of type 'b Lwt.t, call it promise Y, is created by bind and returned to the user. It is pending at that point.
- The second promise of type 'b Lwt.t, call it promise Z, has not yet been created. It will be created later, when promise X has been resolved, and the callback has been run on the contents of X. The callback then returns promise Z. There is no guarantee about the state of Z; it might well still be pending when returned by the callback.
- When Z is finally resolved, the contents of Y are updated to be the same as the contents of Z.

The reason why `bind` is designed with this type is so that programmers can set up a *sequential chain* of callbacks. For example, the following code asynchronously reads one string; then when that string has been read, proceeds to asynchronously read a second string; then prints the concatenation of both strings:

```
Lwt.bind (read_line stdin) (fun s1 ->
  Lwt.bind (read_line stdin) (fun s2 ->
    Lwt_io.printf "%s\n" (s1^s2)));;
```

If you run that in utop, something slightly confusing will happen again: after you press Enter at the end of the first string, Lwt will allow utop to read one character. The problem is that we're mixing Lwt input operations with utop input operations. It would be better to just create a program and run it from the command line.

To do that, put the following code in a file called `read2.ml`:

```
open Lwt_io

let p =
  Lwt.bind (read_line stdin) (fun s1 ->
    Lwt.bind (read_line stdin) (fun s2 ->
      Lwt_io.printf "%s\n" (s1^s2)))

let _ = Lwt_main.run p
```

We've added one new function: `Lwt_main.run : 'a Lwt.t -> 'a`. It waits for its input promise to be resolved, then returns the contents. Typically this function is called only once in an entire program, near the end of the main file; and the input to it is typically a promise whose resolution indicates that all execution is finished.

Create a dune file:

```
(executable
 (name read2)
 (libraries lwt.unix))
```

And run the program, entering a couple strings:

```
dune exec ./read2.exe
My first string
My second string
My first stringMy second string
```

Now try removing the last line of `read2.ml`. You'll see that the program exits immediately, without waiting for you to type.

Bind as an Operator. There is another syntax for bind that is used far more frequently than what we have seen so far. The `Lwt.Infix` module defines an infix operator written `>>=` that is the same as `bind`. That is, instead of writing `bind p c` you write `p >>= c`. This operator makes it much easier to write code without all the extra parentheses and indentations that our previous example had:

```
open Lwt_io
open Lwt.Infix

let p =
  read_line stdin >>= fun s1 ->
  read_line stdin >>= fun s2 ->
  Lwt_io.printf "%s\n" (s1^s2)

let _ = Lwt_main.run p
```

The way to visually parse the definition of `p` is to look at each line as computing some promised value. The first line, `read_line stdin >>= fun s1 ->` means that a promise is created, resolved, and its contents extracted under the name `s1`. The second line means the same, except that its contents are named `s2`. The third line creates a final promise whose contents are eventually extracted by `Lst_main.run`, at which point the program may terminate.

The `>>=` operator is perhaps most famous from the functional language Haskell, which uses it extensively for monads. We'll cover monads as our next major topic.

Bind as Let Syntax. There is a *syntax extension* for OCaml that makes using bind even simpler than the infix operator `>>=`. To install the syntax extension, run the following command:

```
$ opam install lwt_ppx
```

(You might need to `opam update` followed by `opam upgrade` first.)

With that extension, you can use a specialized `let` expression written `let%lwt x = e1 in e2`, which is equivalent to `bind e1 (fun x -> e2)` or `e1 >>= fun x -> e2`. We can rewrite our running example as follows:

```
(* to compile, add lwt_ppx to the libraries in the dune file *)
open Lwt_io

let p =
  let%lwt s1 = read_line stdin in
  let%lwt s2 = read_line stdin in
  Lwt_io.printf "%s\n" (s1^s2)

let _ = Lwt_main.run p
```

Now the code looks pretty much exactly like what its equivalent synchronous version would be. But don't be fooled: all the asynchronous I/O, the promises, and the callbacks are still there. Thus, the evaluation of `p` first registers a callback with a promise, then moves on to the the evaluation of `Lwt_main.run` without waiting for the first string to finish being read. To prove that to yourself, run the following code:

```
open Lwt_io

let p =
  let%lwt s1 = read_line stdin in
  let%lwt s2 = read_line stdin in
  Lwt_io.printf "%s\n" (s1^s2)

let _ = Lwt_io.printf "Got here first\n"

let _ = Lwt_main.run p
```

You'll see that "Got here first" prints before you get a chance to enter any input.

Concurrent Composition. The `Lwt.bind` function provides a way to sequentially compose callbacks: first one callback is run, then another, then another, and so forth. There are other functions in the library for composition of many callbacks as a set. For example,

- `Lwt.join : unit Lwt.t list -> unit Lwt.t` enables waiting upon multiple promises. `Lwt.join ps` returns a promise that is pending until all the promises in `ps` become resolved. You might register a callback on the return promise from the `join` to take care of some computation that needs **all** of a set of promises to be finished.
- `Lwt.pick : 'a Lwt.t list -> 'a Lwt.t` also enables waiting upon multiple promises, but `Lwt.pick ps` returns a promise that is pending until at least one promise in `ps` becomes resolved. You might register a callback on the return promise from the `pick` to take care of some computation that needs just one of a set of promises to be finished, but doesn't care which one.

10.6.9 Implementing Callbacks

When a callback is registered with `bind` or one of the other syntaxes, it is added to a list of callbacks that is stored with the promise. Eventually, when the promise has been resolved, the Lwt *resolution loop* runs the callbacks registered for the promise. There is no guarantee about the execution order of callbacks for a promise. In other words, the execution order is nondeterministic. If the order matters, the programmer needs to use the composition operators (such as `bind` and `join`) to enforce an ordering. If the promise never becomes resolved (or is rejected), none of its callbacks will ever be run.

Once again, it's important to keep track of where the concurrency really comes from: the OS. There might be many asynchronous I/O operations occurring at the OS level. But at the OCaml level, the resolution loop is sequential, meaning that only one callback can ever be running at a time.

Finally, the resolution loop never attempts to interrupt a callback. So if the callback goes into an infinite loop, no other callback will ever get to run. That makes Lwt a cooperative concurrency mechanism, rather than preemptive.

To better understand callback resolution, let's implement it ourselves. We'll use the `Promise` data structure we developed earlier. To start, we add a bind operator to the `Promise` signature:

```
module type PROMISE = sig
  ...

  (** [p >>= c] registers callback [c] with promise [p].
      When the promise is resolved, the callback will be run
      on the promises's contents.  If the promise is never
      resolved, the callback will never run. *)
  val (>>=) : 'a promise -> ('a -> 'b promise) -> 'b promise
end
```

Next, let's re-develop the entire `Promise` structure. We start off just like before:

```
module Promise : PROMISE = struct
  type 'a state = Pending | Resolved of 'a | Rejected of exn
  ...
```

But now to implement the representation type of promises, we use a record with mutable fields. The first field is the state of the promise, and it corresponds to the `ref` we used before. The second field is more interesting and is discussed below.

```
  (** RI: the input may not be [Pending] *)
  type 'a handler = 'a state -> unit

  (** RI: if [state <> Pending] then [handlers = []]. *)
  type 'a promise = {
    mutable state : 'a state;
    mutable handlers : 'a handler list
  }
```

A *handler* is a new abstraction: a function that takes a non-pending state. It will be used to handle resolving and rejecting promises when their state is ready to switch away from pending. The primary use for a handler will be to run callbacks. As a representation invariant, we require that only pending promises may have handlers waiting in their list. Once the state becomes non-pending, i.e., either resolved or rejected, the handlers will all be processed and removed from the list.

This helper function that enqueues a handler on a promise's handler list will be helpful later:

```
  let enqueue
      (handler : 'a state -> unit)
```

(continues on next page)

(continued from previous page)

```
    (promise : 'a promise) : unit
  =
  promise.handlers <- handler :: promise.handlers
```

We continue to pun resolvers and promises internally:

```
type 'a resolver = 'a promise
```

Because we changed the representation type from a `ref` to a record, we have to update a few of the functions in trivial ways:

```
(** [write_once p s] changes the state of [p] to be [s].  If [p] and [s]
    are both pending, that has no effect.
    Raises: [Invalid_arg] if the state of [p] is not pending. *)
let write_once p s =
  if p.state = Pending
  then p.state <- s
  else invalid_arg "cannot write twice"

let make () =
  let p = {state = Pending; handlers = []} in
  p, p

let return x =
  {state = Resolved x; handlers = []}

let state p = p.state
```

Now we get to the trickier parts of the implementation. To resolve or reject a promise, the first thing we need to do is to call `write_once` on it, as we did before. Now we also need to process the handlers. Before doing so, we mutate the handlers list to be empty to ensure that the RI holds.

```
(** requires: [st] may not be [Pending] *)
let resolve_or_reject (r : 'a resolver) (st : 'a state) =
  assert (st <> Pending);
  let handlers = r.handlers in
  r.handlers <- [];
  write_once r st;
  List.iter (fun f -> f st) handlers

let reject r x =
  resolve_or_reject r (Rejected x)

let resolve r x =
  resolve_or_reject r (Resolved x)
```

Finally, the implementation of >>= is the trickiest part. First, if the promise is already resolved, let's go ahead and immediately run the callback on it:

```
let (>>=)
    (input_promise : 'a promise)
    (callback : 'a -> 'b promise) : 'b promise
  =
  match input_promise.state with
  | Resolved x -> callback x
```

Second, if the promise is already rejected, then we return a promise that is rejected with the same exception:

```
    | Rejected exc -> {state = Rejected exc; handlers = []}
```

Third, if the promise is pending, we need to do more work. Here's what we said in our discussion of `bind` in the previous section:

> [T]he bind function returns a new promise. That promise will become resolved when (or if) the callback completes running, sometime in the future. Its contents will be whatever contents are contained within the promise that the callback itself returns.

That's what we now need to implement. So, we create a new promise and resolver called `output_promise` and `output_resolver`. That promise is what `bind` returns. Before returning it, we use a helper function `handler_of_callback` (described below) to transform the callback into a handler, and enqueue that handler on the promise. That ensures the handler will be run when the promise later becomes resolved or rejected:

```
    | Pending ->
      let output_promise, output_resolver = make () in
      enqueue (handler_of_callback callback output_resolver) input_promise;
      output_promise
```

All that's left is to implement that helper function to create handlers from callbacks. The first two cases, below, are simple. It would violate the RI to call a handler on a pending state. And if the state is rejected, then the handler should propagate that rejection to the resolver, which causes the promise returned by bind to also be rejected.

```
let handler_of_callback
    (callback : 'a -> 'b promise)
    (resolver : 'b resolver) : 'a handler
  = function
    | Pending -> failwith "handler RI violated"
    | Rejected exc -> reject resolver exc
```

But if the state is resolved, then the callback provided by the user to bind can—at last!—be run on the contents of the resolved promise. Running the callback produces a new promise. It might already be rejected or resolved, in which case that state again propagates.

```
      | Resolved x ->
        let promise = callback x in
        match promise.state with
        | Resolved y -> resolve resolver y
        | Rejected exc -> reject resolver exc
```

But the promise might still be pending. In that case, we need to enqueue a new handler whose purpose is to do the propagation once the result is available:

```
        | Pending -> enqueue (handler resolver) promise
```

where `handler` is a new helper function that creates a very simple handler to do that propagation:

```
let handler (resolver : 'a resolver) : 'a handler
  = function
    | Pending -> failwith "handler RI violated"
    | Rejected exc -> reject resolver exc
    | Resolved x -> resolve resolver x
```

The Lwt implementation of `bind` follows essentially the same algorithm as we just implemented. Note that there is no concurrency in `bind`: as we said above, everything in Lwt is sequential; it's the OS that provides the concurrency.

10.6.10 The Full Implementation

Here's all of that code in one executable block:

```
(** A signature for Lwt-style promises, with better names *)
module type PROMISE = sig
  type 'a state =
    | Pending
    | Resolved of 'a
    | Rejected of exn

  type 'a promise

  type 'a resolver

  (** [make ()] is a new promise and resolver. The promise is pending. *)
  val make : unit -> 'a promise * 'a resolver

  (** [return x] is a new promise that is already resolved with value
      [x]. *)
  val return : 'a -> 'a promise

  (** [state p] is the state of the promise *)
  val state : 'a promise -> 'a state

  (** [resolve r x] resolves the promise [p] associated with [r] with
      value [x], meaning that [state p] will become [Resolved x].
      Requires: [p] is pending. *)
  val resolve : 'a resolver -> 'a -> unit

  (** [reject r x] rejects the promise [p] associated with [r] with
      exception [x], meaning that [state p] will become [Rejected x].
      Requires: [p] is pending. *)
  val reject : 'a resolver -> exn -> unit

  (** [p >>= c] registers callback [c] with promise [p].
      When the promise is resolved, the callback will be run
      on the promises's contents.  If the promise is never
      resolved, the callback will never run. *)
  val (>>=) : 'a promise -> ('a -> 'b promise) -> 'b promise
end

module Promise : PROMISE = struct
  type 'a state = Pending | Resolved of 'a | Rejected of exn

  (** RI: the input may not be [Pending] *)
  type 'a handler = 'a state -> unit

  (** RI: if [state <> Pending] then [handlers = []]. *)
  type 'a promise = {
    mutable state : 'a state;
    mutable handlers : 'a handler list
  }
```

(continues on next page)

(continued from previous page)

```
let enqueue
    (handler : 'a state -> unit)
    (promise : 'a promise) : unit
  =
  promise.handlers <- handler :: promise.handlers

type 'a resolver = 'a promise

(** [write_once p s] changes the state of [p] to be [s].  If [p] and [s]
    are both pending, that has no effect.
    Raises: [Invalid_arg] if the state of [p] is not pending. *)
let write_once p s =
  if p.state = Pending
  then p.state <- s
  else invalid_arg "cannot write twice"

let make () =
  let p = {state = Pending; handlers = []} in
  p, p

let return x =
  {state = Resolved x; handlers = []}

let state p = p.state

(** requires: [st] may not be [Pending] *)
let resolve_or_reject (r : 'a resolver) (st : 'a state) =
  assert (st <> Pending);
  let handlers = r.handlers in
  r.handlers <- [];
  write_once r st;
  List.iter (fun f -> f st) handlers

let reject r x =
  resolve_or_reject r (Rejected x)

let resolve r x =
  resolve_or_reject r (Resolved x)

let handler (resolver : 'a resolver) : 'a handler
  = function
    | Pending -> failwith "handler RI violated"
    | Rejected exc -> reject resolver exc
    | Resolved x -> resolve resolver x

let handler_of_callback
    (callback : 'a -> 'b promise)
    (resolver : 'b resolver) : 'a handler
  = function
    | Pending -> failwith "handler RI violated"
    | Rejected exc -> reject resolver exc
    | Resolved x ->
      let promise = callback x in
      match promise.state with
      | Resolved y -> resolve resolver y
```

(continues on next page)

(continued from previous page)

```
        | Rejected exc -> reject resolver exc
        | Pending -> enqueue (handler resolver) promise

  let (>>=)
      (input_promise : 'a promise)
      (callback : 'a -> 'b promise) : 'b promise
    =
    match input_promise.state with
    | Resolved x -> callback x
    | Rejected exc -> {state = Rejected exc; handlers = []}
    | Pending ->
      let output_promise, output_resolver = make () in
      enqueue (handler_of_callback callback output_resolver) input_promise;
      output_promise
end
```

10.7 Monads

A *monad* is more of a design pattern than a data structure. That is, there are many data structures that, if you look at them in the right way, turn out to be monads.

The name "monad" comes from the mathematical field of *category theory*, which studies abstractions of mathematical structures. If you ever take a PhD level class on programming language theory, you will likely encounter that idea in more detail. Here, though, we will omit most of the mathematical theory and concentrate on code.

Monads became popular in the programming world through their use in Haskell, a functional programming language that is even more pure than OCaml—that is, Haskell avoids side effects and imperative features even more than OCaml. But no practical language can do without side effects. After all, printing to the screen is a side effect. So Haskell set out to control the use of side effects through the monad design pattern. Since then, monads have become recognized as useful in other functional programming languages, and are even starting to appear in imperative languages.

Monads are used to model *computations*. Think of a computation as being like a function, which maps an input to an output, but as also doing "something more." The something more is an effect that the function has as a result of being computed. For example, the effect might involve printing to the screen. Monads provide an abstraction of effects, and help to make sure that effects happen in a controlled order.

10.7.1 The Monad Signature

For our purposes, a monad is a structure that satisfies two properties. First, it must match the following signature:

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
end
```

Second, a monad must obey what are called the *monad laws*. We will return to those much later, after we have studied the `return` and `bind` operations.

Think of a monad as being like a box that contains some value. The value has type `'a`, and the box that contains it is of type `'a t`. We have previously used a similar box metaphor for both options and promises. That was no accident: options and promises are both examples of monads, as we will see in detail, below.

Return. The `return` operation metaphorically puts a value into a box. You can see that in its type: the input is of type `'a`, and the output is of type `'a t`.

In terms of computations, `return` is intended to have some kind of trivial effect. For example, if the monad represents computations whose side effect is printing to the screen, the trivial effect would be to not print anything.

Bind. The `bind` operation metaphorically takes as input:

- a boxed value, which has type `'a t`, and
- a function that itself takes an *unboxed* value of type `'a` as input and returns a *boxed* value of type `'b t` as output.

The `bind` applies its second argument to the first. That requires taking the `'a` value out of its box, applying the function to it, and returning the result.

In terms of computations, `bind` is intended to sequence effects one after another. Continuing the running example of printing, sequencing would mean first printing one string, then another, and `bind` would be making sure that the printing happens in the correct order.

The usual notation for `bind` is as an infix operator written >>= and still pronounced "bind". So let's revise our signature for monads:

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
end
```

All of the above is likely to feel very abstract upon first reading. It will help to see some concrete examples of monads. Once you understand several >>= and `return` operations, the design pattern itself should make more sense.

So the next few sections look at several different examples of code in which monads can be discovered. Because monads are a design pattern, they aren't always obvious; it can take some study to tease out where the monad operations are being used.

10.7.2 The Maybe Monad

As we've seen before, sometimes functions are partial: there is no good output they can produce for some inputs. For example, the function `max_list : int list -> int` doesn't necessarily have a good output value to return for the empty list. One possibility is to raise an exception. Another possibility is to change the return type to be `int option`, and use `None` to represent the function's inability to produce an output. In other words, *maybe* the function produces an output, or *maybe* it is unable to do so hence returns `None`.

As another example, consider the built-in OCaml integer division function `( / ) : int -> int -> int`. If its second argument is zero, it raises an exception. Another possibility, though, would be to change its type to be `( / ) : int -> int -> int option`, and return `None` whenever the divisor is zero.

Both of those examples involved changing the output type of a partial function to be an option, thus making the function total. That's a nice way to program, until you start trying to combine many functions together. For example, because all the integer operations—addition, subtraction, division, multiplication, negation, etc.—expect an `int` (or two) as input, you can form large expressions out of them. But as soon as you change the output type of division to be an option, you lose that *compositionality*.

Here's some code to make that idea concrete:

```
(* works fine *)
let x = 1 + (4 / 2)
```

```
let div (x:int) (y:int) : int option =
  if y = 0 then None else Some (x / y)

let ( / ) = div

(* won't type check *)
let x = 1 + (4 / 2)
```

The problem is that we can't add an `int` to an `int option`: the addition operator expects its second input to be of type `int`, but the new division operator returns a value of type `int option`.

One possibility would be to re-code all the existing operators to accept `int option` as input. For example,

```
let plus_opt (x:int option) (y:int option) : int option =
  match x,y with
  | None, _ | _, None -> None
  | Some a, Some b -> Some (Stdlib.( + ) a b)

let ( + ) = plus_opt

let minus_opt (x:int option) (y:int option) : int option =
  match x,y with
  | None, _ | _, None -> None
  | Some a, Some b -> Some (Stdlib.( - ) a b)

let ( - ) = minus_opt

let mult_opt (x:int option) (y:int option) : int option =
  match x,y with
  | None, _ | _, None -> None
  | Some a, Some b -> Some (Stdlib.( * ) a b)

let ( * ) = mult_opt

let div_opt (x:int option) (y:int option) : int option =
  match x,y with
  | None, _ | _, None -> None
  | Some a, Some b ->
    if b=0 then None else Some (Stdlib.( / ) a b)

let ( / ) = div_opt
```

```
(* does type check *)
let x = Some 1 + (Some 4 / Some 2)
```

But that's a tremendous amount of code duplication. We ought to apply the Abstraction Principle and deduplicate. Three of the four operators can be handled by abstracting a function that just does some pattern matching to propagate `None`:

```
let propagate_none (op : int -> int -> int) (x : int option) (y : int option) =
  match x, y with
  | None, _ | _, None -> None
  | Some a, Some b -> Some (op a b)

let ( + ) = propagate_none Stdlib.( + )
let ( - ) = propagate_none Stdlib.( - )
let ( * ) = propagate_none Stdlib.( * )
```

Unfortunately, division is harder to deduplicate. We can't just pass `Stdlib.( / )` to `propagate_none`, because neither of those functions will check to see whether the divisor is zero. It would be nice if we could pass our function `div : int -> int -> int option` to `propagate_none`, but the return type of `div` makes that impossible.

So, let's rewrite `propagate_none` to accept an operator of the same type as `div`, which makes it easy to implement division:

```
let propagate_none
  (op : int -> int -> int option) (x : int option) (y : int option)
=
  match x, y with
  | None, _ | _, None -> None
  | Some a, Some b -> op a b

let ( / ) = propagate_none div
```

Implementing the other three operations requires a little more work, because their return type is `int` not `int option`. We need to wrap their return value with `Some`:

```
let wrap_output (op : int -> int -> int) (x : int) (y : int) : int option =
  Some (op x y)

let ( + ) = propagate_none (wrap_output Stdlib.( + ))
let ( - ) = propagate_none (wrap_output Stdlib.( - ))
let ( * ) = propagate_none (wrap_output Stdlib.( * ))
```

Finally, we could re-implement `div` to use `wrap_output`:

```
let div (x : int) (y : int) : int option =
  if y = 0 then None else wrap_output Stdlib.( / ) x y

let ( / ) = propagate_none div
```

Where's the Monad? The work we just did was to take functions on integers and transform them into functions on values that maybe are integers, but maybe are not—that is, values that are either `Some i` where `i` is an integer, or are `None`. We can think of these "upgraded" functions as computations that *may have the effect of producing nothing*. They produce metaphorical boxes, and those boxes may be full of something, or contain nothing.

There were two fundamental ideas in the code we just wrote, which correspond to the monad operations of `return` and `bind`.

The first (which admittedly seems trivial) was upgrading a value from `int` to `int option` by wrapping it with `Some`. That's what the body of `wrap_output` does. We could expose that idea even more clearly by defining the following function:

```
let return (x : int) : int option = Some x
```

This function has the *trivial effect* of putting a value into the metaphorical box.

The second idea was factoring out code to handle all the pattern matching against `None`. We had to upgrade functions whose inputs were of type `int` to instead accept inputs of type `int option`. Here's that idea expressed as its own function:

```
let bind (x : int option) (op : int -> int option) : int option =
  match x with
  | None -> None
```

(continues on next page)

(continued from previous page)

```
  | Some a -> op a

let ( >>= ) = bind
```

The `bind` function can be understood as doing the core work of upgrading `op` from a function that accepts an `int` as input to a function that accepts an `int option` as input. In fact, we could even write a function that does that upgrading for us using `bind`:

```
let upgrade : (int -> int option) -> (int option -> int option) =
  fun (op : int -> int option) (x : int option) -> (x >>= op)
```

All those type annotations are intended to help the reader understand the function. Of course, it could be written much more simply as:

```
let upgrade op x = x >>= op
```

Using just the `return` and >>= functions, we could re-implement the arithmetic operations from above:

```
let ( + ) (x : int option) (y : int option) : int option =
  x >>= fun a ->
  y >>= fun b ->
  return (Stdlib.( + ) a b)

let ( - ) (x : int option) (y : int option) : int option =
  x >>= fun a ->
  y >>= fun b ->
  return (Stdlib.( - ) a b)

let ( * ) (x : int option) (y : int option) : int option =
  x >>= fun a ->
  y >>= fun b ->
  return (Stdlib.( * ) a b)

let ( / ) (x : int option) (y : int option) : int option =
  x >>= fun a ->
  y >>= fun b ->
  if b = 0 then None else return (Stdlib.( / ) a b)
```

Recall, from our discussion of the bind operator in Lwt, that the syntax above should be parsed by your eye as

- take `x` and extract from it the value `a`,
- then take `y` and extract from it `b`,
- then use `a` and `b` to construct a return value.

Of course, there's still a fair amount of duplication going on there. We can de-duplicate by using the same techniques as we did before:

```
let upgrade_binary op x y =
  x >>= fun a ->
  y >>= fun b ->
  op a b

let return_binary op x y = return (op x y)
```

(continues on next page)

(continued from previous page)

```
let ( + ) = upgrade_binary (return_binary Stdlib.( + ))
let ( - ) = upgrade_binary (return_binary Stdlib.( - ))
let ( * ) = upgrade_binary (return_binary Stdlib.( * ))
let ( / ) = upgrade_binary div
```

The Maybe Monad. The monad we just discovered goes by several names: the *maybe monad* (as in, "maybe there's a value, maybe not"), the *error monad* (as in, "either there's a value or an error", and error is represented by `None`—though some authors would want an error monad to be able to represent multiple kinds of errors rather than just collapse them all to `None`), and the *option monad* (which is obvious).

Here's an implementation of the monad signature for the maybe monad:

```
module Maybe : Monad = struct
  type 'a t = 'a option

  let return x = Some x

  let (>>=) m f =
    match m with
    | None -> None
    | Some x -> f x
end
```

These are the same implementations of `return` and >>= as we invented above, but without the type annotations to force them to work only on integers. Indeed, we never needed those annotations; they just helped make the code above a little clearer.

In practice the `return` function here is quite trivial and not really necessary. But the >>= operator can be used to replace a lot of boilerplate pattern matching, as we saw in the final implementation of the arithmetic operators above. There's just a single pattern match, which is inside of >>=. Compare that to the original implementations of `plus_opt`, etc., which had many pattern matches.

The result is we get code that (once you understand how to read the bind operator) is easier to read and easier to maintain.

Now that we're done playing with integer operators, we should restore their original meaning for the rest of this file:

```
let ( + ) = Stdlib.( + )
let ( - ) = Stdlib.( - )
let ( * ) = Stdlib.( * )
let ( / ) = Stdlib.( / )
```

10.7.3 Example: The Writer Monad

When trying to diagnose faults in a system, it's often the case that a *log* of what functions have been called, as well as what their inputs and outputs were, would be helpful.

Imagine that we had two functions we wanted to debug, both of type `int -> int`. For example:

```
let inc x = x + 1
let dec x = x - 1
```

(Ok, those are really simple functions; we probably don't need any help debugging them. But imagine they compute something far more complicated, like encryptions or decryptions of integers.)

One way to keep a log of function calls would be to augment each function to return a pair: the integer value the function would normally return, as well as a string containing a log message. For example:

```
let inc_log x = (x + 1, Printf.sprintf "Called inc on %i; " x)
let dec_log x = (x - 1, Printf.sprintf "Called dec on %i; " x)
```

But that changes the return type of both functions, which makes it hard to *compose* the functions. Previously, we could have written code such as

```
let id x = dec (inc x)
```

or even better

```
let id x = x |> inc |> dec
```

or even better still, using the *composition operator* >>,

```
let ( >> ) f g x = x |> f |> g
let id = inc >> dec
```

and that would have worked just fine. But trying to do the same thing with the loggable versions of the functions produces a type-checking error:

```
let id = inc_log >> dec_log
```

That's because `inc_log x` would be a pair, but `dec_log` expects simply an integer as input.

We could code up an upgraded version of `dec_log` that is able to take a pair as input:

```
let dec_log_upgraded (x, s) =
  (x - 1, Printf.sprintf "%s; Called dec on %i; " s x)

let id x = x |> inc_log |> dec_log_upgraded
```

That works fine, but we also will need to code up a similar upgraded version of `f_log` if we ever want to call them in reverse order, e.g., `let id = dec_log >> inc_log`. So we have to write:

```
let inc_log_upgraded (x, s) =
  (x + 1, Printf.sprintf "%s; Called inc on %i; " s x)

let id = dec_log >> inc_log_upgraded
```

And at this point we've duplicated far too much code. The implementations of `inc` and `dec` are duplicated inside both `inc_log` and `dec_log`, as well as inside both upgraded versions of the functions. And both the upgrades duplicate the code for concatenating log messages together. The more functions we want to make loggable, the worse this duplication is going to become!

So, let's start over, and factor out a couple helper functions. The first helper calls a function and produces a log message:

```
let log (name : string) (f : int -> int) : int -> int * string =
  fun x -> (f x, Printf.sprintf "Called %s on %i; " name x)
```

The second helper produces a logging function of type `'a * string -> 'b * string` out of a non-loggable function:

```
let loggable (name : string) (f : int -> int) : int * string -> int * string =
  fun (x, s1) ->
    let (y, s2) = log name f x in
    (y, s1 ^ s2)
```

Using those helpers, we can implement the logging versions of our functions without any duplication of code involving pairs or pattern matching or string concatenation:

```
let inc' : int * string -> int * string =
  loggable "inc" inc

let dec' : int * string -> int * string =
  loggable "dec" dec

let id' : int * string -> int * string =
  inc' >> dec'
```

Here's an example usage:

```
id' (5, "")
```

Notice how it's inconvenient to call our loggable functions on integers, since we have to pair the integer with a string. So let's write one more function to help with that by pairing an integer with the *empty* log:

```
let e x = (x, "")
```

And now we can write `id' (e 5)` instead of `id' (5, "")`.

Where's the Monad? The work we just did was to take functions on integers and transform them into functions on integers paired with log messages. We can think of these "upgraded" functions as computations that log. They produce metaphorical boxes, and those boxes contain function outputs as well as log messages.

There were two fundamental ideas in the code we just wrote, which correspond to the monad operations of `return` and `bind`.

The first was upgrading a value from `int` to `int * string` by pairing it with the empty string. That's what `e` does. We could rename it `return`:

```
let return (x : int) : int * string = (x, "")
```

This function has the *trivial effect* of putting a value into the metaphorical box along with the empty log message.

The second idea was factoring out code to handle pattern matching against pairs and string concatenation. Here's that idea expressed as its own function:

```
let ( >>= ) (m : int * string) (f : int -> int * string) : int * string =
  let (x, s1) = m in
  let (y, s2) = f x in
  (y, s1 ^ s2)
```

Using `return` and >>=, we can re-implement `loggable`, such that no pairs or pattern matching are ever used in its body:

```
let loggable (name : string) (f : int -> int) : int * string -> int * string =
  fun m ->
```

(continues on next page)

(continued from previous page)

```
    m >>= fun x ->
    log name f x >>= fun y ->
    return y
```

The Writer Monad. The monad we just discovered is usually called the *writer monad* (as in, "additionally writing to a log or string"). Here's an implementation of the monad signature for it:

```
module Writer : Monad = struct
  type 'a t = 'a * string

  let return x = (x, "")

  let ( >>= ) m f =
    let (x, s1) = m in
    let (y, s2) = f x in
    (y, s1 ^ s2)
end
```

As we saw with the maybe monad, these are the same implementations of `return` and `>>=` as we invented above, but without the type annotations to force them to work only on integers. Indeed, we never needed those annotations; they just helped make the code above a little clearer.

It's debatable which version of `loggable` is easier to read. Certainly you need to be comfortable with the monadic style of programming to appreciate the version of it that uses `>>=`. But if you were developing a much larger code base (i.e., with more functions involving paired strings than just `loggable`), using the `>>=` operator is likely to be a good choice: it means the code you write can concentrate on the `'a` in the type `'a Writer.t` instead of on the strings. In other words, the writer monad will take care of the strings for you, as long as you use `return` and `>>=`.

10.7.4 Example: The Lwt Monad

By now, it's probably obvious that the Lwt promises library that we discussed is also a monad. The type `'a Lwt.t` of promises has a `return` and `bind` operation of the right types to be a monad:

```
val return : 'a -> 'a t
val bind : 'a t -> ('a -> 'b t) -> 'b t
```

And `Lwt.Infix.( >>= )` is a synonym for `Lwt.bind`, so the library does provide an infix bind operator.

Now we start to see some of the great power of the monad design pattern. The implementation of `'a t` and `return` that we saw before involves creating references, but those references are completely hidden behind the monadic interface. Moreover, we know that `bind` involves registering callbacks, but that functionality (which as you might imagine involves maintaining collections of callbacks) is entirely encapsulated.

Metaphorically, as we discussed before, the box involved here is one that starts out empty but eventually will be filled with a value of type `'a`. The "something more" in these computations is that values are being produced asynchronously, rather than immediately.

10.7.5 Monad Laws

Every data structure has not just a signature, but some expected behavior. For example, a stack has a push and a pop operation, and we expect those operations to satisfy certain algebraic laws. We saw those for stacks when we studied equational specification:

- `peek (push x s) = x`
- `pop (push x empty) = empty`
- etc.

A monad, though, is not just a single data structure. It's a design pattern for data structures. So it's impossible to write specifications of `return` and >>= for monads in general: the specifications would need to discuss the particular monad, like the writer monad or the Lwt monad.

On the other hand, it turns out that we can write down some laws that ought to hold of any monad. The reason for that goes back to one of the intuitions we gave about monads, namely, that they represent computations that have effects. Consider Lwt, for example. We might register a callback C on promise X with `bind`. That produces a new promise Y, on which we could register another callback D. We expect a sequential ordering on those callbacks: C must run before D, because Y cannot be resolved before X.

That notion of *sequential order* is part of what the monad laws stipulate. We will state those laws below. But first, let's pause to consider sequential order in imperative languages.

**Sequential Order.* In languages like Java and C, there is a semicolon that imposes a sequential order on statements, e.g.:

```
System.out.println(x);
x++;
System.out.println(x);
```

First `x` is printed, then incremented, then printed again. The effects that those statements have must occur in that sequential order.

Let's imagine a hypothetical statement that causes no effect whatsoever. For example, `assert true` causes nothing to happen in Java. (Some compilers will completely ignore it and not even produce bytecode for it.) In most assembly languages, there is likewise a "no op" instruction whose mnemonic is usually `NOP` that also causes nothing to happen. (Technically, some clock cycles would elapse. But there wouldn't be any changes to registers or memory.) In the theory of programming languages, statements like this are usually called `skip`, as in, "skip over me because I don't do anything interesting."

Here are two laws that should hold of `skip` and semicolon:

- `skip; s;` should behave the same as just `s;`.
- `s; skip;` should behave the same as just `s;`.

In other words, you can remove any occurrences of `skip`, because it has no effects. Mathematically, we say that `skip` is a *left identity* (the first law) and a *right identity* (the second law) of semicolon.

Imperative languages also usually have a way of grouping statements together into blocks. In Java and C, this is usually done with curly braces. Here is a law that should hold of blocks and semicolon:

- `{s1; s2;} s3;` should behave the same as `s1; {s2; s3;}`.

In other words, the order is always `s1` then `s2` then `s3`, regardless of whether you group the first two statements into a block or the second two into a block. So you could even remove the braces and just write `s1; s2; s3;`, which is what we normally do anyway. Mathematically, we say that semicolon is *associative.*

Sequential Order with the Monad Laws. The three laws above embody exactly the same intuition as the monad laws, which we will now state. The monad laws are just a bit more abstract hence harder to understand at first.

Suppose that we have any monad, which as usual must have the following signature:

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
end
```

The three monad laws are as follows:

- **Law 1:** `return x >>= f` behaves the same as `f x`.
- **Law 2:** `m >>= return` behaves the same as `m`.
- **Law 3:** `(m >>= f) >>= g` behaves the same as `m >>= (fun x -> f x >>= g)`.

Here, "behaves the same as" means that the two expressions will both evaluate to the same value, or they will both go into an infinite loop, or they will both raise the same exception.

These laws are mathematically saying the same things as the laws for `skip`, semicolon, and braces that we saw above: `return` is a left and right identity of `>>=`, and `>>=` is associative. Let's look at each law in more detail.

Law 1 says that having the trivial effect on a value, then binding a function on it, is the same as just calling the function on the value. Consider the maybe monad: `return x` would be `Some x`, and `>>= f` would extract `x` and apply `f` to it. Or consider the Lwt monad: `return x` would be a promise that is already resolved with `x`, and `>>= f` would register `f` as a callback to run on `x`.

Law 2 says that binding on the trivial effect is the same as just not having the effect. Consider the maybe monad: `m >>= return` would depend upon whether `m` is `Some x` or `None`. In the former case, binding would extract `x`, and `return` would just re-wrap it with `Some`. In the latter case, binding would just return `None`. Similarly, with Lwt, binding on `m` would register `return` as a callback to be run on the contents of `m` after it is resolved, and `return` would just take those contents and put them back into an already resolved promise.

Law 3 says that bind sequences effects correctly, but it's harder to see it in this law than it was in the version above with semicolon and braces. Law 3 would be clearer if we could rewrite it as

> `(m >>= f) >>= g` behaves the same as `m >>= (f >>= g)`.

But the problem is that doesn't type check: `f >>= g` doesn't have the right type to be on the right-hand side of `>>=`. So we have to insert an extra anonymous function `fun x -> ...` to make the types correct.

10.7.6 Composition and Monad Laws

There is another monad operator called `compose` that can be used to compose monadic functions. For example, suppose you have a monad with type `'a t`, and two functions:

- `f : 'a -> 'b t`
- `g : 'b -> 'c t`

The composition of those functions would be

- `compose f g : 'a -> 'c t`

That is, the composition would take a value of type `'a`, apply `f` to it, extract the `'b` out of the result, apply `g` to it, and return that value.

We can code up `compose` using `>>=`; we don't need to know anything more about the inner workings of the monad:

```
let compose f g x =
  f x >>= fun y ->
  g y

let ( >=> ) = compose
```

As the last line suggests, `compose` can be expressed as infix operator written `>=>`.

Returning to our example of the maybe monad with a safe division operator, imagine that we have increment and decrement functions:

```
let inc (x : int) : int option = Some (x + 1)
let dec (x : int) : int option = Some (x - 1)
let ( >>= ) x op =
  match x with
  | None -> None
  | Some a -> op a
```

The monadic compose operator would enable us to compose those two into an identity function without having to write any additional code:

```
let ( >=> ) f g x =
  f x >>= fun y ->
  g y

let id : int -> int option = inc >=> dec
```

Using the compose operator, there is a much cleaner formulation of the monad laws:

- **Law 1:** `return >=> f` behaves the same as `f`.
- **Law 2:** `f >=> return` behaves the same as `f`.
- **Law 3:** `(f >=> g) >=> h` behaves the same as `f >=> (g >=> h)`.

In that formulation, it becomes immediately clear that `return` is a left and right identity, and that composition is associative.

10.8 Summary

This chapter has taken a deep dive into some advanced data structures, analysis techniques, and programming patterns. Our goal has been to write correct, efficient, beautiful code. Did we succeed? You can be the judge.

10.8.1 Terms and Concepts

- amortized analysis
- association list
- associative
- associative array
- asymptotic bound
- asynchronous

- banker's method
- big oh
- bind
- binding
- blocking
- brute force
- bucket
- caching
- callback
- chaining
- channel
- collision
- complexity
- computations
- concurrent
- concurrent composition
- cooperative
- credits
- cycle
- delayed evaluation
- deterministic
- dictionary
- diffusion
- direct address table
- eager
- effects
- efficiency
- execution steps
- exponential time
- force
- hash function
- infinite data structure
- injective
- input size
- interleaving
- key

- latency hiding
- lazy
- left identity
- load factor
- Lwt monad
- map
- maybe monad
- memoization
- monads
- monads laws
- mutable map
- non-blocking
- nondeterministic
- parallelism
- pending
- persistent
- physicist's method
- polynomial time
- potential energy
- preemptive
- probing
- promises
- race conditions
- recursive values
- red-black map
- rejected
- resizing
- resolution loop
- resolved
- resolver
- right identity
- sequential
- sequential composition
- serialization
- set
- standard input

- standard output
- stream
- strict
- synchronous
- threads
- thunk
- worst case performance
- writer monad

10.8.2 Further Reading

- *More OCaml: Algorithms, Methods, and Diversions*, chapters 2 and 11, by John Whitington.
- *Introduction to Objective Caml*, chapter 8, section 4
- *Real World OCaml*, chapters 13 and 18
- *Purely Functional Data Structures*, by Chris Okasaki. Cambridge University Press, 1999.

10.9 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Exercise: hash insert [★★]

Suppose we have a hash table on integer keys. The table currently has 7 empty buckets, and the hash function is simply `let hash k = k mod 7`. Draw the hash table that results from inserting the keys 4, 8, 15, 16, 23, and 42 (with whatever values you like).

Exercise: relax bucket RI [★★]

We required that hash table buckets must not contain duplicates. What would happen if we relaxed this RI to allow duplicates? Would the efficiency of any operations (insert, find, or remove) change?

Exercise: strengthen bucket RI [★★]

What would happen if we strengthened the bucket RI to require each bucket to be sorted by the key? Would the efficiency of any operations (insert, find, or remove) change?

Exercise: hash values [★★]

Use `Hashtbl.hash : 'a -> int` to hash several values of different types. Make sure to try at least `()`, false, true, `0`, `1`, `""`, and `[]`, as well as several "larger" values of each type. We saw that lists quickly can create collisions. Try creating binary trees and finding a collision.□

Exercise: hashtbl usage [★★]

Create a hash table `tab` with `Hashtbl.create` whose initial size is 16. Add 31 bindings to it with `Hashtbl.add`. For example, you could add the numbers 1..31 as keys and the strings "1".."31" as their values. Use `Hashtbl.find` to look for keys that are in `tab`, as well as keys that are not.

Exercise: hashtbl stats [★]

Use the `Hashtbl.stats` function to find out the statistics of `tab` (from an exercise above). How many buckets are in the table? How many buckets have a single binding in them?

Exercise: hashtbl bindings [★★]

Define a function `bindings : ('a,'b) Hashtbl.t -> ('a*'b) list`, such that `bindings h` returns a list of all bindings in `h`. Use your function to see all the bindings in `tab` (from an exercise above). *Hint: fold.*

Exercise: hashtbl load factor [★★]

Define a function `load_factor : ('a,'b) Hashtbl.t -> float`, such that `load_factor h` is the load factor of `h`. What is the load factor of `tab`? *Hint: stats.*

Add one more binding to `tab`. Do the stats or load factor change? Now add yet another binding. Now do the stats or load factor change? *Hint: Hashtbl resizes when the load factor goes strictly above 2.*

Exercise: functorial interface [★★★]

Use the functorial interface (i.e., `Hashtbl.Make`) to create a hash table whose keys are strings that are case insensitive. Be careful to obey the specification of `Hashtbl.HashedType.hash`:

> If two keys are equal according to `equal`, then they have identical hash values as computed by `hash`.

Exercise: equals and hash [★★]

The previous exercise quoted the specification of `Hashtbl.HashedType.hash`. Compare that to Java's `Object.hashCode()` specification. Why do they both have this similar requirement?

Exercise: bad hash [★★]

Use the functorial interface to create a hash table with a really bad hash function (e.g., a constant function). Use the `stats` function to see how bad the bucket distribution becomes.

Exercise: linear probing [★★★★]

We briefly mentioned *probing* as an alternative to *chaining*. Probing can be effectively used in hardware implementations of hash tables, as well as in databases. With probing, every bucket contains exactly one binding. In case of a collision, we search forward through the array, as described below.

Your task: Implement a hash table that uses linear probing. The details are below.

Find. Suppose we are trying to find a binding in the table. We hash the binding's key and look in the appropriate bucket. If there is already a different key in that bucket, we start searching forward through the array at the next bucket, then the next bucket, and so forth, wrapping back around to the beginning of the array if necessary. Eventually we will either

- find an empty bucket, in which case the key we're searching for is not bound in the table;
- find the key before we reach an empty bucket, in which case we can return the value; or
- never find the key or an empty bucket, instead wrapping back around to the original bucket, in which case all buckets are full and the key is not bound in the table. This case actually should never occur, because we won't allow the load factor to get high enough for all buckets to be filled.

Insert. Insertion follows the same algorithm as finding a key, except that whenever we first find an empty bucket, we can insert the binding there.

Remove. Removal is more difficult. Once the key is found, we can't just make the bucket empty, because that would affect future searches by causing them to stop early. Instead, we can introduce a special "deleted" value into that bucket to indicate that the bucket does not contain a binding but the searches should not stop at it.

Resizing. Since we never want the array to become completely full, we can keep the load factor near 1/4. When the load factor exceeds 1/2, we can double the array, bringing the load factor back to 1/4. When the load factor goes below 1/8, we can half the array, again bringing the load factor back to 1/4. "Deleted" bindings complicate the definition of load factor:

- When determining whether to double the table size, we calculate the load factor as (# of bindings + # of deleted bindings) / (# of buckets). That is, deleted bindings contribute toward increasing the load factor.
- When determining whether the half the table size, we calculate the load factor as (# of bindings) / (# buckets). That is, deleted bindings do not count toward increasing the load factor.

When rehashing the table, deleted bindings are of course not re-inserted into the new table.

Exercise: functorized BST [★★★]

Our implementation of BSTs assumed that it was okay to compare values using the built-in comparison operators <, =, and >. But what if the client wanted to use their own comparison operators? (e.g., to ignore case in strings, or to have sets of records where only a single field of the record was used for ordering.) Implement a `BstSet` abstraction as a functor parameterized on a structure that enables client-provided comparison operator(s), much like the standard library `Set`.

Exercise: efficient traversal [★★★]

Suppose you wanted to convert a tree to a list. You'd have to put the values stored in the tree in some order. Here are three ways of doing that:

- *preorder*: each node's value appears in the list before the values of its left then right subtrees.
- *inorder*: the values of the left subtree appear, then the value at the node, then the values of the right subtree.
- *postorder*: the values of a node's left then right subtrees appear, followed by the value at the node.

Here is code that implements those *traversals*, along with some example applications:

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree

let rec preorder = function
  | Leaf -> []
  | Node (l,v,r) -> [v] @ preorder l @ preorder r

let rec inorder = function
  | Leaf -> []
  | Node (l,v,r) ->  inorder l @ [v] @ inorder r
```

(continues on next page)

(continued from previous page)

```
let rec postorder = function
  | Leaf -> []
  | Node (l,v,r) ->  postorder l @ postorder r @ [v]

let t =
  Node(Node(Node(Leaf, 1, Leaf), 2, Node(Leaf, 3, Leaf)),
       4,
       Node(Node(Leaf, 5, Leaf), 6, Node(Leaf, 7, Leaf)))

(*
  t is
         4
       /   \
      2     6
     / \   / \
    1   3 5   7
*)

let () = assert (preorder t  = [4;2;1;3;6;5;7])
let () = assert (inorder t   = [1;2;3;4;5;6;7])
let () = assert (postorder t = [1;3;2;5;7;6;4])
```

On unbalanced trees, the traversal functions above require quadratic worst-case time (in the number of nodes), because of the `@` operator. Re-implement the functions without `@`, and instead using `::`, such that they perform exactly one cons per `Node` in the tree. Thus the worst-case execution time will be linear. You will need to add an additional accumulator argument to each function, much like with tail recursion. (But your implementations won't actually be tail recursive.)

Exercise: RB draw complete [★★]

Draw the perfect binary tree on the values 1, 2, …, 15. Color the nodes in three different ways such that (i) each way is a red-black tree (i.e., satisfies the red-black invariants), and (ii) the three ways create trees with black heights of 2, 3, and 4, respectively. Recall that the *black height* of a tree is the maximum number of black nodes along any path from its root to a leaf.

Exercise: RB draw insert [★★]

Draw the red-black tree that results from inserting the characters D A T A S T R U C T U R E into an empty tree. Carry out the insertion algorithm yourself by hand, then check your work with the implementation provided in the book.

Exercise: standard library set [★★]

Read the source code of the standard library `Set` module. Find the representation invariant for the balanced trees that it uses. Which kind of tree does it most resemble: 2-3, AVL, or red-black?

Exercise: pow2 [★★]

Using this type:

```
type 'a sequence = Cons of 'a * (unit -> 'a sequence)
```

Define a value `pow2 : int sequence` whose elements are the powers of two: `<1; 2; 4; 8; 16, ...>`.

Exercise: more sequences [★★]

Define the following sequences:

- the even naturals
- the lower-case alphabet on endless repeat: a, b, c, …, z, a, b, …
- unending pseudorandom coin flips (e.g., booleans or a variant with `Heads` and `Tails` constructors)

Exercise: nth [★★]

Define a function `nth : 'a sequence -> int -> 'a`, such that `nth s n` the element at zero-based position `n` in sequence `s`. For example, `nth pow2 0 = 1`, and `nth pow2 4 = 16`.

Exercise: hd tl [★★]

Explain how each of the following sequence expressions is evaluated:

- `hd nats`
- `tl nats`
- `hd (tl nats)`
- `tl (tl nats)`
- `hd (tl (tl nats))`

Exercise: filter [★★★]

Define a function `filter : ('a -> bool) -> 'a sequence -> 'a sequence`, such that `filter p s` is the sub-sequence of `s` whose elements satisfy the predicate `p`. For example, `filter (fun n -> n mod 2 = 0) nats` would be the sequence `<0; 2; 4; 6; 8; 10; ...>`. If there is no element of `s` that satisfies `p`, then `filter p s` does not terminate.

Exercise: interleave [★★★]

Define a function `interleave : 'a sequence -> 'a sequence -> 'a sequence`, such that `interleave <a1; a2; a3; ...> <b1; b2; b3; ...>` is the sequence `<a1; b1; a2; b2; a3; b3; ...>`. For example, `interleave nats pow2` would be `<0; 1; 1; 2; 2; 4; 3; 8; ...>`

Exercise: sift [★★★]

The *Sieve of Eratosthenes* is a way of computing the prime numbers.

- Start with the sequence `<2; 3; 4; 5; 6; ...>`.
- Take 2 as prime. Delete all multiples of 2, since they cannot be prime. That leaves `<3; 5; 7; 9; 11; ...>`.
- Take 3 as prime and delete its multiples. That leaves `<5; 7; 11; 13; 17; ...>`.
- Take 5 as prime, etc.

Define a function `sift : int -> int sequence -> int sequence`, such that `sift n s` removes all multiples of `n` from `s`. *Hint: filter.*

Exercise: primes [★★★]

Define a sequence `prime : int sequence`, containing all the prime numbers starting with 2.

Exercise: approximately e [★★★★]

The exponential function e^x can be computed by the following infinite sum:

$$e^x = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^k}{k!} + \cdots$$

Define a function `e_terms : float -> float sequence`. Element `k` of the sequence should be term `k` from the infinite sum. For example, `e_terms 1.0` is the sequence `<1.0; 1.0; 0.5; 0.1666...; 0.041666...; ...>`. The easy way to compute that involves a function that computes $f(k) = \frac{x^k}{k!}$.

Define a function `total : float sequence -> float sequence`, such that `total <a; b; c; ...>` is a running total of the input elements, i.e., `<a; a+.b; a+.b+.c; ...>`.

Define a function `within : float -> float sequence -> float`, such that `within eps s` is the first element of `s` for which the absolute difference between that element and the element before it is strictly less than `eps`. If there is no such element, `within` is permitted not to terminate (i.e., go into an "infinite loop"). As a precondition, the *tolerance* `eps` must be strictly positive. For example, `within 0.1 <1.0; 2.0; 2.5; 2.75; 2.875; 2.9375; 2.96875; ...>` is `2.9375`.

Finally, define a function `e : float -> float -> float` such that `e x eps` is e^x computed to within a tolerance of `eps`, which must be strictly positive. Note that there is an interesting boundary case where `x=1.0` for the first two terms of the sum; you could choose to drop the first term (which is always `1.0`) from the sequence before using `within`.

Exercise: better e [★★★★]

Although the idea for computing e^x above through the summation of an infinite series is good, the exact algorithm suggested above could be improved. For example, computing the 20th term in the sequence leads to a very large numerator and denominator if x is large. Investigate that behavior, comparing it to the built-in function `exp : float -> float`. Find a better way to structure the computation to improve the approximations you obtain. *Hint: what if when computing term k you already had term $k-1$? Then you could just do a single multiplication and division.*

Also, you could improve the test that `within` uses to determine whether two values are close. A good one for determining whether a and b are close might be *relative distance*:

$$\left|\frac{a-b}{min(a,b)}\right| < \epsilon.$$

Exercise: different sequence rep [★★★]

Consider this alternative representation of sequences:

```
type 'a sequence = Cons of (unit -> 'a * 'a sequence)
```

How would you code up `hd : 'a sequence -> 'a`, `tl : 'a sequence -> 'a sequence`, `nats : int sequence`, and `map : ('a -> 'b) -> 'a sequence -> 'b sequence` for it? Explain how this representation is even lazier than our original representation.

Exercise: lazy hello [★]

Define a value of type `unit Lazy.t` (which is synonymous with `unit lazy_t`), such that forcing that value with `Lazy.force` causes `"Hello lazy world"` to be printed. If you force it again, the string should not be printed.

Exercise: lazy and [★★]

Define a function `(&&&) : bool Lazy.t -> bool Lazy.t -> bool`. It should behave like a short circuit Boolean AND. That is, `lb1 &&& lb2` should first force `lb1`. If it is `false`, the function should return `false`. Otherwise, it should force `lb2` and return its value.

Exercise: lazy sequence [★★★]

Implement `map` and `filter` for the `'a lazysequence` type provided in the section on laziness.

Exercise: promise and resolve [★★]

Use the finished version of the `Promise` module we developed to do the following: create a integer promise and resolver, bind a function on the promise to print the contents of the promise, then resolve the promise. Only after the promise is resolved should the printing occur.

Exercise: promise and resolve lwt [★★]

Repeat the above exercise, but use the Lwt library instead of our own Promise library. Make sure to use Lwt's I/O functions (e.g., `Lwt_io.printf`).

Exercise: timing challenge 1 [★★]

Here is a function that produces a time delay. We can use it to simulate an I/O call that takes a long time to complete.

```
(** [delay s] is a promise that resolves after about [s] seconds. *)
let delay (sec : float) : unit Lwt.t =
  Lwt_unix.sleep sec
```

Write a function `delay_then_print : unit -> unit Lwt.t` that delays for three seconds then prints `"done"`.

Exercise: timing challenge 2 [★★★]

What happens when `timing2 ()` is run? How long does it take to run? Make a prediction, then run the code to find out.

```
open Lwt.Infix

let timing2 () =
  let _t1 = delay 1. >>= fun () -> Lwt_io.printl "1" in
  let _t2 = delay 10. >>= fun () -> Lwt_io.printl "2" in
  let _t3 = delay 20. >>= fun () -> Lwt_io.printl "3" in
  Lwt_io.printl "all done"
```

Exercise: timing challenge 3 [★★★]

What happens when `timing3 ()` is run? How long does it take to run? Make a prediction, then run the code to find out.

```
open Lwt.Infix

let timing3 () =
  delay 1. >>= fun () ->
  Lwt_io.printl "1" >>= fun () ->
  delay 10. >>= fun () ->
  Lwt_io.printl "2" >>= fun () ->
  delay 20. >>= fun () ->
  Lwt_io.printl "3" >>= fun () ->
  Lwt_io.printl "all done"
```

Exercise: timing challenge 4 [★★★]

What happens when `timing4 ()` is run? How long does it take to run? Make a prediction, then run the code to find out.

```
open Lwt.Infix

let timing4 () =
  let t1 = delay 1. >>= fun () -> Lwt_io.printl "1" in
  let t2 = delay 10. >>= fun () -> Lwt_io.printl "2" in
  let t3 = delay 20. >>= fun () -> Lwt_io.printl "3" in
  Lwt.join [t1; t2; t3] >>= fun () ->
  Lwt_io.printl "all done"
```

Exercise: file monitor [★★★★]

Write an Lwt program that monitors the contents of a file named "log". Specifically, your program should open the file, continually read a line from the file, and as each line becomes available, print the line to stdout. When you reach the end of the file (EOF), your program should terminate cleanly without any exceptions.

Here is starter code:

```
open Lwt.Infix
open Lwt_io
open Lwt_unix

(** [log ()] is a promise for an [input_channel] that reads from
    the file named "log". *)
```

(continues on next page)

(continued from previous page)

```
let log () : input_channel Lwt.t =
  openfile "log" [O_RDONLY] 0 >>= fun fd ->
  Lwt.return (of_fd input fd)

(** [loop ic] reads one line from [ic], prints it to stdout,
    then calls itself recursively. It is an infinite loop. *)
let rec loop (ic : input_channel) =
  failwith "TODO"
  (* hint: use [Lwt_io.read_line] and [Lwt_io.printlf] *)

(** [monitor ()] monitors the file named "log". *)
let monitor () : unit Lwt.t =
  log () >>= loop

(** [handler] is a helper function for [main]. If its input is
    [End_of_file], it handles cleanly exiting the program by
    returning the unit promise. Any other input is re-raised
    with [Lwt.fail]. *)
let handler : exn -> unit Lwt.t =
  failwith "TODO"

let main () : unit Lwt.t =
  Lwt.catch monitor handler

let _ = Lwt_main.run (main ())
```

Complete `loop` and `handler`. You might find the Lwt manual to be useful.

To compile your code, put it in a file named `monitor.ml`. Create a dune file for it:

```
(executable
 (name monitor)
 (libraries lwt.unix))
```

And run it as usual:

```
$ dune exec ./monitor.exe
```

To simulate a file to which lines are being added over time, open a new terminal window and enter the following commands:

```
$ mkfifo log
$ cat >log
```

Now anything you type into the terminal window (after pressing return) will be added to the file named `log`. That will enable you to interactively test your program.

Exercise: add opt [★★]

Here are the definitions for the maybe monad:

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
```

(continues on next page)

(continued from previous page)

```
end

module Maybe : Monad =
struct
  type 'a t = 'a option

  let return x = Some x

  let ( >>= ) m f =
    match m with
    | Some x -> f x
    | None -> None

end
```

Implement `add : int Maybe.t -> int Maybe.t -> int Maybe.t`. If either of the inputs is `None`, then the output should be `None`. Otherwise, if the inputs are `Some a` and `Some b` then the output should be `Some (a+b)`. The definition of `add` must be located outside of `Maybe`, as shown above, which means that your solution may not use the constructors `None` or `Some` in its code.

Exercise: fmap and join [★★]

Here is an extended signature for monads that adds two new operations:

```
module type ExtMonad = sig
  type 'a t
  val return : 'a -> 'a t
  val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
  val ( >>| ) : 'a t -> ('a -> 'b) -> 'b t
  val join : 'a t t -> 'a t
end
```

Just as the infix operator >>= is known as `bind`, the infix operator `>>|` is known as `fmap`. The two operators differ only in the return type of their function argument.

Using the box metaphor, `>>|` takes a boxed value, and a function that only knows how to work on unboxed values, extracts the value from the box, runs the function on it, and boxes up that output as its own return value.

Also using the box metaphor, `join` takes a value that is wrapped in two boxes and removes one of the boxes.

It's possible to implement `>>|` and `join` directly with pattern matching (as we already implemented >>=). It's also possible to implement them without pattern matching.

For this exercise, do the former: implement `>>|` and `join` as part of the `Maybe` monad, and do not use >>= or `return` in the body of `>>|` or `join`.

Exercise: fmap and join again [★★]

Solve the previous exercise again. This time, you must use >>= and `return` to implement `>>|` and `join`, and you may not use `Some` or `None` in the body of `>>|` and `join`.

Exercise: bind from fmap+join [★★★]

The previous exercise demonstrates that `>>|` and `join` can be implemented entirely in terms of `>>=` (and `return`), without needing to know anything about the representation type `'a t` of the monad.

It's actually possible to go the other direction. That is, `>>=` can be implemented using just `>>|` and `join`, without needing to know anything about the representation type `'a t`.

Prove that this is so by completing the following code:

```
module type FmapJoinMonad = sig
  type 'a t
  val ( >>| ) : 'a t -> ('a -> 'b) -> 'b t
  val join : 'a t t -> 'a t
  val return : 'a -> 'a t
end

module type BindMonad = sig
  type 'a t
  val return : 'a -> 'a t
  val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
end

module MakeMonad (M : FmapJoinMonad) : BindMonad = struct
  (* TODO *)
end
```

Hint: let the types be your guide.

Exercise: list monad [★★★]

We've seen three examples of monads already; let's examine a fourth, the *list monad*. The "something more" that it does is to upgrade functions to work on lists instead of just single values. (Note, there is no notion of concurrency intended here. It's not that the list monad runs functions concurrently on every element of a list. The Lwt monad does, however, provide that kind of functionality.)

For example, suppose you have these functions:

```
let inc x = x + 1
let pm x = [x; -x]
```

Then the list monad could be used to apply those functions to every element of a list and return the result as a list. For example,

- `[1; 2; 3] >>| inc` is `[2; 3; 4]`.
- `[1; 2; 3] >>= pm` is `[1; -1; 2; -2; 3; -3]`.
- `[1; 2; 3] >>= pm >>| inc` is `[2; 0; 3; -1; 4; -2]`.

One way to think about this is that the list monad operators take a list of inputs to a function, run the function on all those inputs, and give you back the combined list of outputs.

Complete the following definition of the list monad:

```
module type ExtMonad = sig
  type 'a t
  val return : 'a -> 'a t
  val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
  val ( >>| ) : 'a t -> ('a -> 'b) -> 'b t
```

(continues on next page)

(continued from previous page)

```
  val join : 'a t t -> 'a t
end

module ListMonad : ExtMonad = struct
  type 'a t = 'a list

  (* TODO *)
end
```

Hints: Leave >>= for last. Let the types be your guide. There are two very useful list library functions that can help you.

Exercise: trivial monad laws [★★★]

Here is the world's most trivial monad. All it does is wrap a value inside of a constructor.

```
module Trivial : Monad = struct
  type 'a t = Wrap of 'a
  let return x = Wrap x
  let ( >>= ) (Wrap x) f = f x
end
```

Prove that the three monad laws, as formulated using >>= and `return`, hold for the trivial monad.

Part V

Language Implementation

CHAPTER

ELEVEN

INTERPRETERS

A skilled artisan must understand the tools with which they work. A carpenter needs to understand saws and planes. A chef needs to understand knives and pots. A programmer, among other tools, needs to understand the compilers that implement the programming languages they use.

A full understanding of compilation requires a full course or two. So here, we're going to take a necessarily brief look at how to implement programming languages. The goal is to understand some of the basic implementation techniques, so as to demystify the tools you're using. Although you might never need to implement a full general-purpose programming language, it's highly likely that at some point in your career you will want to design and implement some small, special-purpose language. Sometimes those are called *domain-specific languages* (DSLs). What we cover here should help you with that task.

A *compiler* is a program that implements a programming language. So is an *interpreter*. But they differ in their implementation strategy.

A compiler's primary task is *translation*. It takes as input a *source program* and produces as output a *target program*. The source program is typically expressed in a high-level language, such as Java or OCaml. The target program is typically expressed in a low-level language, such as MIPS or x86 assembly. Then the compiler's job is done, and it is no longer needed. Later the OS helps to load and execute the target program. Typically, a compiler results in higher-performance implementations.

An interpreter's primary task is *execution*. It takes as input a source program and directly executes that program without producing any target program. The OS actually loads and executes the interpreter, and the interpreter is then responsible for executing the program. Typically, an interpreter is easier to implement than a compiler.

It's also possible to implement a language using a mixture of compilation and interpretation. The most common example of that involves *virtual machines* that execute *bytecode*, such as the Java Virtual Machine (JVM) or the OCaml virtual machine (which used to be called the Zinc Machine). With this strategy, a compiler translates the source language into bytecode, and the virtual machine interprets the bytecode.

High-performance virtual machines, such as Java's HotSpot, take this a step further and embed a compiler inside the virtual machine. When the machine notices that a piece of bytecode is being interpreted frequently, it uses the compiler to translate that bytecode into the language of the machine (e.g., x86) on which the machine is running. This is called *just-in-time compilation* (JIT), because code is being compiled just before it is executed.

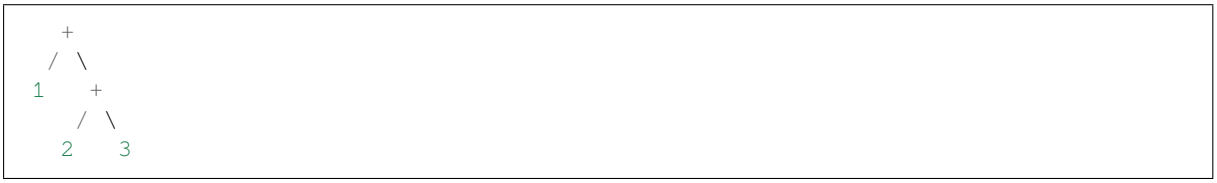
A compiler goes through several phases as it translates a program:

Lexing. During lexing, the compiler transforms the original source code of the program from a sequence of characters to a sequence of *tokens*. Tokens are adjacent characters that have some meaning when grouped together. You might think of them analogously to words in a natural language. Indeed, keywords such as `if` and `match` would be tokens in OCaml. So would constants such as `42` and `"hello"`, variable names such as `x` and `lst`, and punctuation such as `(`, `)`, and `->`. Lexing typically removes whitespace, because it is no longer needed once the tokens have been identified. (Though in a whitespace-sensitive language like Python, it would need to be preserved.)

Parsing. During parsing, the compiler transforms the sequence of tokens into a tree called the *abstract syntax tree* (AST). As the name suggests, this tree abstracts from the *concrete syntax* of the language. Recall that abstraction can mean

"forgetting about details." The AST typically forgets about concrete details. For example:

- In `1 + (2 + 3)` the parentheses group the right-hand addition operation, indicating it should be evaluated first. A tree can represent that as follows:



  Parentheses are no longer needed, because the structure of the tree encodes them.

- In `[1; 2; 3]`, the square brackets delineate the beginning and end of the list, and the semicolons separate the list elements. A tree could represent that as a node with several children:

  The brackets and semicolons are no longer needed.

- In `fun x -> 42`, the `fun` keyword and `->` punctuation mark separate the arguments and body of the function from the surrounding code. A tree can represent that as a node with two children:

  The keyword and punctuation are no longer needed.

An AST thus represents the structure of a program at a level that is easier for the compiler writer to manipulate.

Semantic analysis. During semantic analysis, the compiler checks to see whether the program is meaningful according to the rules of the language that the compiler is implementing. The most common kind of semantic analysis is type checking: the compiler analyzes the types of all the expressions that appear in the program to see whether there is a type error or not. Type checking typically requires producing a data structure called a *symbol table* that maps identifiers (e.g., variable names) to their types. As a new scope is entered, the symbol table is extended with new bindings that might shadow old bindings; and as the scope is exited, the new bindings are removed, thus restoring the old bindings. So a symbol table blends features of a dictionary and a stack data structure.

Besides type checking, there are other kinds of semantic analysis. Examples include the following:

- checking whether the branches of an OCaml pattern match are exhaustive,
- checking whether a C `break` keyword occurs within the body of a loop, and
- checking whether a Java field marked `final` has been initialized by the end of a constructor.

You can think of parsing as "checking to see whether a program is meaningful"—which is how we just defined semantic analysis. So the distinction between parsing and semantic analysis is more about convenience: parsing does enough work to implement the production of an AST, and semantic analysis does the rest of the work.

Sometimes semantic analysis is even necessary to fully determine what the AST should be! Consider, for example, the expression `(foo) - bar` in a C-like language. It might be:

- the unary negation of a variable `bar`, cast to the type `foo`, or
- the binary subtraction operation with operands `foo` and `bar`, where the parentheses were gratuitous.

Until enough semantic analysis has been done to figure out whether `foo` is a variable name or a type name, the compiler doesn't know which AST to generate. In such situations, the parser typically produces an AST in which some tree nodes represent the ambiguous syntax, then the semantic analysis phase rewrites the tree to be unambiguous.

Translation to intermediate representation. After semantic analysis, a compiler *could* immediately translate the AST (augmented with symbol tables) into the target language. But if the same compiler wanted to produce output for multiple targets (e.g., for x86 and ARM and MIPS), that would require defining a translation from the AST to each of the targets. In practice, compilers typically don't do that. Instead, they first translate the AST to an intermediate representation (IR). Think of the IR as a kind of abstraction of many assembly languages. Many source languages (e.g., C, Java, OCaml) could be translated to the same IR, and from that IR, many target language outputs (e.g., x86, ARM, MIPS) could be produced.

An IR language typically has *abstract machine instructions* that accomplish conceptually simple tasks: loading from or storing to memory, performing binary operations, calling and returning, and jumping to other instructions. The abstract machine typically has an unbounded number of registers available for use, much like a source program can have an unbounded number of variables. Real machines, however, have a finite number of registers, which is one way in which the IR is an abstraction.

Target code generation. The final phase of compilation is to generate target code from the IR. This phase typically involves selecting concrete machine instructions (such as x86 opcodes), and determining which variables will be stored in memory (which is slow to access) vs. processor registers (which are fast to access but limited in number). As part of code generation, a compiler therefore attempts to *optimize* the performance of the target code. Some examples of optimizations include:

- eliminating array bounds checks, if they are provably guaranteed to succeed;
- eliminating redundant computations;
- replacing a function call with the body of the function itself, suitably instantiated on the arguments, to eliminate the overhead of calling and returning; and
- re-ordering machine instructions so that (e.g.) slow reads from memory are begun before their results are needed, and doing other instructions in the meanwhile that do not need the result of the read.

Groups of Phases. The phases of compilation can be grouped into two or three pieces:

- The *front end* of the compiler does lexing, parsing, and semantic analysis. It produces an AST and associated symbol tables. It transforms the AST into an IR.
- The *middle end* (if it exists) of the compiler operates on the IR. Usually this involves performing optimizations that are independent of the target language.
- The *back end* of the compiler does code generation, including further optimization.

Interpretation Phases. An interpreter works like the front (and possibly middle) end of a compiler. That is, an interpreter does lexing, parsing, and semantic analysis. It might then immediately begin executing the AST, or it might transform the AST into an IR and begin executing the IR.

In the rest of this book, we are going to focus on interpreters. We'll ignore IRs and code generation, and instead study how to directly execute the AST.

Note: Because of the additional tooling required, the code in this chapter is not runnable in a browser like previous chapters. But we do provide downloadable code for each interpreter implemented here.

11.1 Example: Calculator

Let's start with a video guided tour of implementing an interpreter for a tiny language: just a calculator, essentially, with addition and multiplication. The point of this guided tour is not to go into great detail about any single piece of it. Rather, the goal is to get a little familiarity with the OCaml tools and techniques for lexing, parsing, and evaluation. They are all rather tightly coupled, which makes it challenging to understand one piece without having a high-level understanding of the whole. After we get that understanding from the tour, we'll start over again in the next section (on parsing), and at that time we'll dive into the details.

11.2 Parsing

You *could* code your own lexer and parser from scratch. But many languages include tools for automatically generating lexers and parsers from formal descriptions of the syntax of a language. The ancestors of many of those tools are lex and yacc, which generate lexers and parsers, respectively; lex and yacc were developed in the 1970s for C.

As part of the standard distribution, OCaml provides lexer and parser generators named ocamllex and ocamlyacc. There is a more modern parser generator named menhir available through opam; menhir is "90% compatible" with ocamlyacc and provides significantly improved support for debugging generated parsers.

11.2.1 Lexers

Lexer generators such as lex and ocamllex are built on the theory of deterministic finite automata, which is typically covered in a discrete math or theory of computation course. Such automata accept *regular languages*, which can be described with *regular expressions*. So, the input to a lexer generator is a collection of regular expressions that describe the tokens of the language. The output is an automaton implemented in a high-level language, such as C (for lex) or OCaml (for ocamllex).

That automaton itself takes files (or strings) as input, and each character of the file becomes an input to the automaton. Eventually the automaton either *recognizes* the sequence of characters it has received as a valid token in the language, in which case the automaton produces an output of that token and resets itself to being recognizing the next token, or *rejects* the sequence of characters as an invalid token.

11.2.2 Parsers

Parser generators such as yacc and menhir are similarly built on the theory of automata. But they use *pushdown automata*, which are like finite automata that also maintain a stack onto which they can push and pop symbols. The stack enables them to accept a bigger class of languages, which are known as *context-free languages* (CFLs). One of the big improvements of CFLs over regular languages is that CFLs can express the idea that delimiters must be balanced—for example, that every opening parenthesis must be balanced by a closing parenthesis.

Just as regular languages can be expressed with a special notation (regular expressions), so can CFLs. *Context-free grammars* are used to describe CFLs. A context-free grammar is a set of *production rules* that describe how one symbol can be replaced by other symbols. For example, the language of balanced parentheses, which includes strings such as `(())` and `()()` and `(()())`, but not strings such as `)` or `(()`, is generated by these rules:

- $S \to (S)$
- $S \to SS$
- $S \to \epsilon$

The symbols occurring in those rules are S, (, and). The ϵ denotes the empty string. Every symbol is either a *nonterminal* or a *terminal*, depending on whether it is a token of the language being described. S is a nonterminal in the example above, and (and) are terminals.

In the next section we'll study *Backus-Naur Form* (BNF), which is a standard notation for context-free grammars. The input to a parser generator is typically a BNF description of the language's syntax. The output of the parser generator is a program that recognizes the language of the grammar. As input, that program expects the output of the lexer. As output, the program produces a value of the AST type that represents the string that was accepted. The programs output by the parser generator and lexer generator are thus dependent upon on another and upon the AST type.

11.2.3 Backus-Naur Form

The standard way to describe the syntax of a language is with a mathematical notation called *Backus-Naur form* (BNF), named for its inventors, John Backus and Peter Naur. There are many variants of BNF. Here, we won't be too picky about adhering to one variant or another. Our goal is just to have a reasonably good notation for describing language syntax.

BNF uses a set of *derivation rules* to describe the syntax of a language. Let's start with an example. Here's the BNF description of a tiny language of expressions that include just the integers and addition:

```
e ::= i | e + e
i ::= <integers>
```

These rules say that an expression `e` is either an integer `i`, or two expressions with the symbol + appearing between them. The syntax of "integers" is left unspecified by these rules.

Each rule has the form

```
metavariable ::= symbols | ... | symbols
```

A *metavariable* is variable used in the BNF rules, rather than a variable in the language being described. The `::=` and `|` that appear in the rules are *metasyntax*: BNF syntax used to describe the language's syntax. *Symbols* are sequences that can include metavariables (such as `i` and `e`) as well as tokens of the language (such as +). Whitespace is not relevant in these rules.

Sometimes we might want to easily refer to individual occurrences of metavariables. We do that by appending some distinguishing mark to the metavariable(s). For example, we could rewrite the first rule above as

```
e ::= i | e1 + e2
```

or as

```
e ::= i | e + e'
```

Now we can talk about `e2` or `e'` rather than having to say "the `e` on the right-hand side of +".

If the language itself contains either of the tokens `::=` or `|`—and OCaml does contain the latter—then writing BNF can become a little confusing. Some BNF notations attempt to deal with that by using additional delimiters to distinguish syntax from metasyntax. We will be more relaxed and assume that the reader can distinguish them.

11.2.4 Example: SimPL

As a running example, we'll use a very simple programming language that we call SimPL. Here is its syntax in BNF:

```
e ::= x | i | b | e1 bop e2
    | if e1 then e2 else e3
    | let x = e1 in e2

bop ::= + | * | <=

x ::= <identifiers>

i ::= <integers>

b ::= true | false
```

Obviously there's a lot missing from this language, especially functions. But there's enough in it for us to study the important concepts of interpreters without getting too distracted by lots of language features. Later, we will consider a larger fragment of OCaml.

We're going to develop a complete interpreter for SimPL. You can download the finished interpreter here: simpl.zip. Or, just follow along as we build each piece of it.

The AST

Since the AST is the most important data structure in an interpreter, let's design it first. We'll put this code in a file named `ast.ml`:

```
type bop =
  | Add
  | Mult
  | Leq

type expr =
  | Var of string
  | Int of int
  | Bool of bool
  | Binop of bop * expr * expr
  | Let of string * expr * expr
  | If of expr * expr * expr
```

There is one constructor for each of the syntactic forms of expressions in the BNF. For the underlying primitive syntactic classes of identifiers, integers, and booleans, we're using OCaml's own `string`, `int`, and `bool` types.

Instead of defining the `bop` type and a single `Binop` constructor, we could have defined three separate constructors for the three binary operators:

```
type expr =
  ...
  | Add of expr * expr
  | Mult of expr * expr
  | Leq of expr * expr
  ...
```

But by factoring out the `bop` type we will be able to avoid a lot of code duplication later in our implementation.

The Menhir Parser

Let's start with parsing, then return to lexing later. We'll put all the Menhir code we write below in a file named `parser.mly`. The `.mly` extension indicates that this file is intended as input to Menhir. (The 'y' alludes to yacc.) This file contains the *grammar definition* for the language we want to parse. The syntax of grammar definitions is described by example below. Be warned that it's maybe a little weird, but that's because it's based on tools (like yacc) that were developed quite awhile ago. Menhir will process that file and produce a file named `parser.ml` as output; it contains an OCaml program that parses the language. (There's nothing special about the name `parser` here; it's just descriptive.)

There are four parts to a grammar definition: header, declarations, rules, and trailer.

Header. The *header* appears between `%{` and `%}`. It is code that will be copied literally into the generated `parser.ml`. Here we use it just to open the `Ast` module so that, later on in the grammar definition, we can write expressions like `Int i` instead of `Ast.Int i`. If we wanted we could also define some OCaml functions in the header.

```
%{
open Ast
%}
```

Declarations. The *declarations* section begins by saying what the lexical *tokens* of the language are. Here are the token declarations for SimPL:

```
%token <int> INT
%token <string> ID
%token TRUE
%token FALSE
%token LEQ
%token TIMES
%token PLUS
%token LPAREN
%token RPAREN
%token LET
%token EQUALS
%token IN
%token IF
%token THEN
%token ELSE
%token EOF
```

Each of these is just a descriptive name for the token. Nothing so far says that `LPAREN` really corresponds to `(`, for example. We'll take care of that when we define the lexer.

The `EOF` token is a special *end-of-file* token that the lexer will return when it comes to the end of the character stream. At that point we know the complete program has been read.

The tokens that have a `<type>` annotation appearing in them are declaring that they will carry some additional data along with them. In the case of `INT`, that's an OCaml `int`. In the case of `ID`, that's an OCaml `string`.

After declaring the tokens, we have to provide some additional information about *precedence* and *associativity*. The following declarations say that `PLUS` is left associative, `IN` is not associative, and `PLUS` has higher precedence than `IN` (because `PLUS` appears on a line after `IN`).

```
%nonassoc IN
%nonassoc ELSE
%left LEQ
%left PLUS
%left TIMES
```

Because `PLUS` is left associative, `1 + 2 + 3` will parse as `(1 + 2) + 3` and not as `1 + (2 + 3)`. Because `PLUS` has higher precedence than `IN`, the expression `let x = 1 in x + 2` will parse as `let x = 1 in (x + 2)` and not as `(let x = 1 in x) + 2`. The other declarations have similar effects.

Getting the precedence and associativity declarations correct is one of the trickier parts of developing a grammar definition. It helps to develop the grammar definition incrementally, adding just a couple tokens (and their associated rules, discussed below) at a time to the language. Menhir will let you know when you've added a token (and rule) for which it is confused about what you intend the precedence and associativity should be. Then you can add declarations and test to make sure you've got them right.

After declaring associativity and precedence, we need to declare what the starting point is for parsing the language. The following declaration says to start with a rule (defined below) named `prog`. The declaration also says that parsing a `prog` will return an OCaml value of type `Ast.expr`.

```
%start <Ast.expr> prog
```

Finally, `%%` ends the declarations section.

```
%%
```

Rules. The *rules* section contains production rules that resemble BNF, although where in BNF we would write "::=" these rules simply write ":". The format of a rule is

```
name:
  | production1 { action1 }
  | production2 { action2 }
  | ...
  ;
```

The *production* is the sequence of *symbols* that the rule matches. A symbol is either a token or the name of another rule. The *action* is the OCaml value to return if a *match* occurs. Each production can *bind* the value carried by a symbol and use that value in its action. This is perhaps best understood by example, so let's dive in.

The first rule, named `prog`, has just a single production. It says that a `prog` is an `expr` followed by `EOF`. The first part of the production, `e=expr`, says to match an `expr` and bind the resulting value to `e`. The action simply says to return that value `e`.

```
prog:
  | e = expr; EOF { e }
  ;
```

The second and final rule, named `expr`, has productions for all the expressions in SimPL.

```
expr:
  | i = INT { Int i }
  | x = ID { Var x }
  | TRUE { Bool true }
  | FALSE { Bool false }
  | e1 = expr; LEQ; e2 = expr { Binop (Leq, e1, e2) }
  | e1 = expr; TIMES; e2 = expr { Binop (Mult, e1, e2) }
  | e1 = expr; PLUS; e2 = expr { Binop (Add, e1, e2) }
  | LET; x = ID; EQUALS; e1 = expr; IN; e2 = expr { Let (x, e1, e2) }
  | IF; e1 = expr; THEN; e2 = expr; ELSE; e3 = expr { If (e1, e2, e3) }
  | LPAREN; e=expr; RPAREN {e}
  ;
```

- The first production, `i = INT`, says to match an `INT` token, bind the resulting OCaml `int` value to `i`, and return AST node `Int i`.
- The second production, `x = ID`, says to match an `ID` token, bind the resulting OCaml `string` value to `x`, and return AST node `Var x`.
- The third and fourth productions match a `TRUE` or `FALSE` token and return the corresponding AST node.
- The fifth, sixth, and seventh productions handle binary operators. For example, `e1 = expr; PLUS; e2 = expr` says to match an `expr` followed by a `PLUS` token followed by another `expr`. The first `expr` is bound to `e1` and the second to `e2`. The AST node returned is `Binop (Add, e1, e2)`.
- The eighth production, `LET; x = ID; EQUALS; e1 = expr; IN; e2 = expr`, says to match a `LET` token followed by an `ID` token followed by an `EQUALS` token followed by an `expr` followed by an `IN` token followed by another `expr`. The string carried by the `ID` is bound to `x`, and the two expressions are bound to `e1` and `e2`. The AST node returned is `Let (x, e1, e2)`.
- The last production, `LPAREN; e = expr; RPAREN` says to match an `LPAREN` token followed by an `expr` followed by an `RPAREN`. The expression is bound to `e` and returned.

The final production might be surprising, because it was not included in the BNF we wrote for SimPL. That BNF was intended to describe the *abstract syntax* of the language, so it did not include the concrete details of how expressions can be grouped with parentheses. But the grammer definition we've been writing does have to describe the *concrete syntax*, including details like parentheses.

There can also be a *trailer* section after the rules, which like the header is OCaml code that is copied directly into the output `parser.ml` file.

The Ocamllex Lexer

Now let's see how the lexer generator is used. A lot of it will feel familiar from our discussion of the parser generator. We'll put all the ocamllex code we write below in a file named `lexer.mll`. The `.mll` extension indicates that this file is intended as input to ocamllex. (The 'l' alludes to lexing.) This file contains the *lexer definition* for the language we want to lex. Menhir will process that file and produce a file named `lexer.ml` as output; it contains an OCaml program that lexes the language. (There's nothing special about the name `lexer` here; it's just descriptive.)

There are four parts to a lexer definition: header, identifiers, rules, and trailer.

Header. The *header* appears between `{` and `}`. It is code that will simply be copied literally into the generated `lexer.ml`.

```
{
open Parser
}
```

Here, we've opened the `Parser` module, which is the code in `parser.ml` that was produced by Menhir out of `parser.mly`. The reason we open it is so that we can use the token names declared in it, e.g., `TRUE`, `LET`, and `INT`, inside our lexer definition. Otherwise, we'd have to write `Parser.TRUE`, etc.

Identifiers. The next section of the lexer definition contains *identifiers*, which are named regular expressions. These will be used in the rules section, next.

Here are the identifiers we'll use with SimPL:

```
let white = [' ' '\t']+
let digit = ['0'-'9']
let int = '-'? digit+
let letter = ['a'-'z' 'A'-'Z']
let id = letter+
```

The regular expressions above are for whitespace (spaces and tabs), digits (0 through 9), integers (nonempty sequences of digits, optionally preceded by a minus sign), letters (a through z, and A through Z), and SimPL variable names (nonempty sequences of letters) aka ids or "identifiers"—though we're now using that word in two different senses.

FYI, these aren't exactly the same as the OCaml definitions of integers and identifiers.

The identifiers section actually isn't required; instead of writing `white` in the rules we could just directly write the regular expression for it. But the identifiers help make the lexer definition more self-documenting.

Rules. The rules section of a lexer definition is written in a notation that also resembles BNF. A rule has the form

```
rule name =
  parse
  | regexp1 { action1 }
  | regexp2 { action2 }
  | ...
```

Here, `rule` and `parse` are keywords. The lexer that is generated will attempt to match against regular expressions in the order they are listed. When a regular expression matches, the lexer produces the token specified by its `action`.

Here is the (only) rule for the SimPL lexer:

```
rule read =
  parse
  | white { read lexbuf }
  | "true" { TRUE }
  | "false" { FALSE }
  | "<=" { LEQ }
  | "*" { TIMES }
  | "+" { PLUS }
  | "(" { LPAREN }
  | ")" { RPAREN }
  | "let" { LET }
  | "=" { EQUALS }
  | "in" { IN }
  | "if" { IF }
  | "then" { THEN }
  | "else" { ELSE }
  | id { ID (Lexing.lexeme lexbuf) }
  | int { INT (int_of_string (Lexing.lexeme lexbuf)) }
  | eof { EOF }
```

Most of the regular expressions and actions are self-explanatory, but a couple are not:

- The first, `white { read lexbuf }`, means that if whitespace is matched, instead of returning a token the lexer should just call the `read` rule again and return whatever token results. In other words, whitespace will be skipped.
- The two for ids and ints use the expression `Lexing.lexeme lexbuf`. This calls a function `lexeme` defined in the `Lexing` module, and returns the string that matched the regular expression. For example, in the `id` rule, it would return the sequence of upper and lower case letters that form the variable name.
- The `eof` regular expression is a special one that matches the end of the file (or string) being lexed.

Note that it's important that the `id` regular expression occur nearly last in the list. Otherwise, keywords like `true` and `if` would be lexed as variable names rather than the `TRUE` and `IF` tokens.

Generating the Parser and Lexer

Now that we have completed parser and lexer definitions in `parser.mly` and `lexer.mll`, we can run Menhir and ocamllex to generate the parser and lexer from them. Let's organize our code like this:

```
- <some root folder>
  - dune-project
  - src
    - ast.ml
    - dune
    - lexer.mll
    - parser.mly
```

In `src/dune`, write the following:

```
(library
 (name interp))

(menhir
 (modules parser))

(ocamllex lexer)
```

That organizes the entire `src` folder into a *library* named `Interp`. The parser and lexer will be modules `Interp.Parser` and `Interp.Lexer` in that library.

Run `dune build` to compile the code, thus generating the parser and lexer. If you want to see the generated code, look in `_build/default/src/` for `parser.ml` and `lexer.ml`.

The Driver

Finally, we can pull together the lexer and parser to transform a string into an AST. Put this code into a file named `src/main.ml`:

```
open Ast

let parse (s : string) : expr =
  let lexbuf = Lexing.from_string s in
  let ast = Parser.prog Lexer.read lexbuf in
  ast
```

This function takes a string `s` and uses the standard library's `Lexing` module to create a *lexer buffer* from it. Think of that buffer as the token stream. The function then lexes and parses the string into an AST, using `Lexer.read` and `Parser.prog`. The function `Lexer.read` corresponds to the rule named `read` in our lexer definition, and the function `Parser.prog` to the rule named `prog` in our parser definition.

Note how this code runs the lexer on a string; there is a corresponding function `from_channel` to read from a file.

We could now use `parse` interactively to parse some strings. Start utop and load the library declared in `src` with this command:

```
$ dune utop src
```

Now `Interp.Main.parse` is available for use:

```
# Interp.Main.parse "let x = 3110 in x + x";;
- : Interp.Ast.expr =
Interp.Ast.Let ("x", Interp.Ast.Int 3110,
 Interp.Ast.Binop (Interp.Ast.Add, Interp.Ast.Var "x", Interp.Ast.Var "x"))
```

That completes lexing and parsing for SimPL.

11.3 Substitution Model

After lexing and parsing, the next phase is type checking (and other semantic analysis). We will skip that phase for now and return to it at the end of this chapter.

Instead, let's turn our attention to evaluation. In a compiler, the next phase after semantic analysis would be rewriting the AST into an intermediate representation (IR), in preparation for translating the program into machine code. An interpreter might also rewrite the AST into an IR, or it might directly begin evaluating the AST. One reason to rewrite the AST would be to simplify it: sometimes, certain language features can be implemented in terms of others, and it makes sense to reduce the language to a small core to keep the interpreter implementation shorter. Syntactic sugar is a great example of that idea.

Eliminating syntactic sugar is called *desugaring*. As an example, we know that `let x = e1 in e2` and `(fun x -> e2) e1` are equivalent. So, we could regard let expressions as syntactic sugar.

Suppose we had a language whose AST corresponded to this BNF:

```
e ::= x | fun x -> e | e1 e2
    | let x = e1 in e2
```

Then the interpreter could desugar that into a simpler AST—in a sense, an IR—by transforming all occurrences of `let x = e1 in e2` into `(fun x -> e2) e1`. Then the interpreter would need to evaluate only this smaller language:

```
e ::= x | fun x -> e | e1 e2
```

After having simplified the AST, it's time to evaluate it. *Evaluation* is the process of continuing to simplify the AST until it's just a value. In other words, evaluation is the implementation of the language's dynamic semantics. Recall that a *value* is an expression for which there is no computation remaining to be done. Typically, we think of values as a strict syntactic subset of expressions, though we'll see some exceptions to that later.

Big vs. small step evaluation. We'll define evaluation with a mathematical relation, just as we did with type checking. Actually, we're going to define three relations for evaluation:

- The first, `-->`, will represent how a program takes one single step of execution.
- The second, `-->*`, is the reflexive transitive closure of `-->`, and it represents how a program takes multiple steps of execution.
- The third, `==>`, abstracts away from all the details of single steps and represents how a program reduces directly to a value.

The style in which we are defining evaluation with these relations is known as *operational semantics*, because we're using the relations to specify how the machine "operates" as it evaluates programs. There are two other major styles, known as *denotational semantics* and *axiomatic semantics*, but we won't cover those here.

We can further divide operational semantics into two separate sub-styles of defining evaluation: *small step* vs. *big step* semantics. The first relation, `-->`, is in the small-step style, because it represents execution in terms of individual small steps. The third, `==>`, is in the big-step style, because it represents execution in terms of a big step from an expression

directly to a value. The second relation, `-->*`, blends the two. Indeed, our desire is for it to bridge the gap in the following sense:

Relating big and small steps: For all expressions `e` and values `v`, it holds that `e -->* v` if and only if `e ==> v`.

In other words, if an expression takes many small steps and eventually reaches a value, e.g., `e --> e1 --> .... --> en --> v`, then it ought to be the case that `e ==> v`. So the big step relation is a faithful abstraction of the small step relation: it just forgets about all the intermediate steps.

Why have two different styles, big and small? Each is a little easier to use than the other in certain circumstances, so it helps to have both in our toolkit. The small-step semantics tends to be easier to work with when it comes to modeling complicated language features, but the big-step semantics tends to be more similar to how an interpreter would actually be implemented.

Substitution vs. environment models. There's another choice we have to make, and it's orthogonal to the choice of small vs. big step. There are two different ways to think about the implementation of variables:

- We could eagerly *substitute* the value of a variable for its name throughout the scope of that name, as soon as we finding a binding of the variable.
- We could lazily record the substitution in a dictionary, which is usually called an *environment* when used for this purpose, and we could look up the variable's value in that environment whenever we find its name mentioned in a scope.

Those ideas lead to the *substitution model* of evaluation and the *environment model* of evaluation. As with small step vs. big step, the substitution model tends to be nicer to work with mathematically, whereas the environment model tends to be more similar to how an interpreter is implemented.

Some examples will help to make sense of all this. Let's look, next, at how to define the relations for SimPL.

11.3.1 Evaluating SimPL in the Substitution Model

Let's begin by defining a small-step substitution-model semantics for SimPL. That is, we're going to define a relation `-->` that represents how an expression take a single step at a time, and we'll implement variables using substitution of values for names.

Recall the syntax of SimPL:

```
e ::= x | i | b | e1 bop e2
    | if e1 then e2 else e3
    | let x = e1 in e2

bop ::= + | * | <=
```

We're going to need to know when expressions are done evaluating, that is, when they are considered to be values. For SimPL, we'll define the values as follows:

```
v ::= i | b
```

That is, a value is either an integer constant or a Boolean constant.

For each of the syntactic forms that a SimPL expression could have, we'll now define some *evaluation rules*, which constitute an inductive definition of the `-->` relation. Each rule will have the form `e --> e'`, meaning that `e` takes a single step to `e'`.

Although variables are given first in the BNF, let's pass over them for now, and come back to them after all the other forms.

Constants. Integer and Boolean constants are already values, so they cannot take a step. That might at first seem surprising, but remember that we are intending to also define a `-->*` relation that will permit zero or more steps; whereas, the `-->` relation represents *exactly* one step.

Technically, all we have to do to accomplish this is to just not write any rules of the form `i --> e` or `b --> e` for some `e`. So we're already done, actually: we haven't defined any rules yet.

Let's introduce another notation written `e -/->`, which is meant to look like an arrow with a slash through it, to mean "there does not exist an `e'` such that `e --> e'`. Using that we could write:

- `i -/->`
- `b -/->`

Though not strictly speaking part of the definition of `-->`, those propositions help us remember that constants do not step. In fact, we could more generally write, "for all `v`, it holds that `v -/->`."

Binary operators. A binary operator application `e1 bop e2` has two subexpressions, `e1` and `e2`. That leads to some choices about how to evaluate the expression:

- We could first evaluate the left-hand side `e1`, then the right-hand side `e2`, then apply the operator.
- Or we could do the right-hand side first, then the left-hand side.
- Or we could interleave the evaluation, first doing a step of `e1`, then of `e2`, then `e1`, then `e2`, etc.
- Or maybe the operator is a *short-circuit* operator, in which case one of the subexpressions might never be evaluated.

And there are many other strategies you might be able to invent.

It turns out that the OCaml language definition says that (for non-short-circuit operators) it is unspecified which side is evaluated first. The current implementation happens to evaluate the right-hand side first, but that's not something any programmer should rely upon.

Many people would expect left-to-right evaluation, so let's define the `-->` relation for that. We start by saying that the left-hand side can take a step:

```
e1 bop e2 --> e1' bop e2
  if e1 --> e1'
```

Similarly to the type system for SimPL, this rule says that two expressions are in the `-->` relation if two other (simpler) subexpressions are also in the `-->` relation. That's what makes it an inductive definition.

If the left-hand side is finished evaluating, then the right-hand side may begin stepping:

```
v1 bop e2 --> v1 bop e2'
  if e2 --> e2'
```

Finally, when both sides have reached a value, the binary operator may be applied:

```
v1 bop v2 --> v
  if v is the result of primitive operation v1 bop v2
```

By *primitive operation*, we mean that there is some underlying notion of what `bop` actually means. For example, the character + is just a piece of syntax, but we are conditioned to understand its meaning as an arithmetic addition operation. The primitive operation typically is something implemented by hardware (e.g., an `ADD` opcode), or by a run-time library (e.g., a `pow` function).

For SimPL, let's delegate all primitive operations to OCaml. That is, the SimPL + operator will be the same as the OCaml + operator, as will `*` and `<=`.

Here's an example of using the binary operator rule:

```
    (3*1000) + ((1*100) + ((1*10) + 0))
--> 3000 + ((1*100) + ((1*10) + 0))
--> 3000 + (100 + ((1*10) + 0))
--> 3000 + (100 + (10 + 0))
--> 3000 + (100 + 10)
--> 3000 + 110
--> 3110
```

If expressions. As with binary operators, there are many choices of how to evaluate the subexpressions of an if expression. Nonetheless, most programmers would expect the guard to be evaluated first, then only one of the branches to be evaluated, because that's how most languages work. So let's write evaluation rules for that semantics.

First, the guard is evaluated to a value:

```
if e1 then e2 else e3 --> if e1' then e2 else e3
  if e1 --> e1'
```

Then, based on the guard, the if expression is simplified to just one of the branches:

```
if true then e2 else e3 --> e2

if false then e2 else e3 --> e3
```

Let expressions. Let's make SimPL let expressions evaluate in the same way as OCaml let expressions: first the binding expression, then the body.

The rule that steps the binding expression is:

```
let x = e1 in e2 --> let x = e1' in e2
  if e1 --> e1'
```

Next, if the binding expression has reached a value, we want to substitute that value for the name of the variable in the body expression:

```
let x = v1 in e2 --> e2 with v1 substituted for x
```

For example, `let x = 42 in x + 1` should step to `42 + 1`, because substituting `42` for `x` in `x + 1` yields `42 + 1`.

Of course, the right hand side of that rule isn't really an expression. It's just giving an intuition for the expression that we really want. We need to formally define what "substitute" means. It turns out to be rather tricky. So, rather then getting side-tracked by it right now, let's assume a new notation: `e'{e/x}`, which means, "the expression `e'` with `e` substituted for `x`." We'll come back to that notation in the next section and give it a careful definition.

For now, we can add this rule:

```
let x = v1 in e2 --> e2{v1/x}
```

Variables. Note how the let expression rule eliminates a variable from showing up in the body expression: the variable's name is replaced by the value that variable should have. So, we should *never* reach the point of attempting to step a variable name—assuming that the program was well typed.

Consider OCaml: if we try to evaluate an expression with an unbound variable, what happens? Let's check utop:

```
# x;;
Error: Unbound value x

# let y = x in y;;
Error: Unbound value x
```

It's an error —a type-checking error— for an expression to contain an unbound variable. Thus, any well-typed expression `e` will never reach the point of attempting to step a variable name.

As with constants, we therefore don't need to add any rules for variables. But, for clarity, we could state that `x -/->`.

11.3.2 Implementing the Single-Step Relation

It's easy to turn the above definitions of `-->` into an OCaml function that pattern matches against AST nodes. In the code below, recall that we have yet finished defining substitution (i.e., `subst`); we'll return to that in the next section.

```
(** [is_value e] is whether [e] is a value. *)
let is_value : expr -> bool = function
  | Int _ | Bool _ -> true
  | Var _ | Let _ | Binop _ | If _ -> false

(** [subst e v x] is [e{v/x}]. *)
let subst e v x =
  failwith "See next section"

(** [step] is the [-->] relation, that is, a single step of
    evaluation. *)
let rec step : expr -> expr = function
  | Int _ | Bool _ -> failwith "Does not step"
  | Var _ -> failwith "Unbound variable"
  | Binop (bop, e1, e2) when is_value e1 && is_value e2 ->
    step_bop bop e1 e2
  | Binop (bop, e1, e2) when is_value e1 ->
    Binop (bop, e1, step e2)
  | Binop (bop, e1, e2) -> Binop (bop, step e1, e2)
  | Let (x, e1, e2) when is_value e1 -> subst e2 e1 x
  | Let (x, e1, e2) -> Let (x, step e1, e2)
  | If (Bool true, e2, _) -> e2
  | If (Bool false, _, e3) -> e3
  | If (Int _, _, _) -> failwith "Guard of if must have type bool"
  | If (e1, e2, e3) -> If (step e1, e2, e3)

(** [step_bop bop v1 v2] implements the primitive operation
    [v1 bop v2].  Requires: [v1] and [v2] are both values. *)
and step_bop bop e1 e2 = match bop, e1, e2 with
  | Add, Int a, Int b -> Int (a + b)
  | Mult, Int a, Int b -> Int (a * b)
  | Leq, Int a, Int b -> Bool (a <= b)
  | _ -> failwith "Operator and operand type mismatch"
```

The only new thing we had to deal with in that implementation was the two places where a run-time type error is discovered, namely, in the evaluation of `If (Int _, _, _)` and in the very last line, in which we discover that a binary operator is being applied to arguments of the wrong type. Type checking will guarantee that an exception never gets raised here, but OCaml's exhaustiveness analysis of pattern matching forces us to write a branch nonetheless. Moreover, if it ever turned out that we had a bug in our type checker that caused ill-typed binary operator applications to be evaluated, this exception would help us discover what was going wrong.

11.3.3 The Multistep Relation

Now that we've defined `-->`, there's really nothing left to do to define `-->*`. It's just the reflexive transitive closure of `-->`. In other words, it can be defined with just these two rules:

```
e -->* e

e -->* e''
  if e --> e' and e' -->* e''
```

Of course, in implementing an interpreter, what we really want is to take as many steps as possible until the expression reaches a value. That is, we're interested in the sub-relation `e -->* v` in which the right-hand side is a not just an expression, but a value. That's easy to implement:

```
(** [eval_small e] is the [e -->* v] relation.  That is,
    keep applying [step] until a value is produced.  *)
let rec eval_small (e : expr) : expr =
  if is_value e then e
  else e |> step |> eval_small
```

11.3.4 Defining the Big-Step Relation

Recall that our goal in defining the big-step relation `==>` is to make sure it agrees with the multistep relation `-->*`.

Constants are easy, because they big-step to themselves:

```
i ==> i

b ==> b
```

Binary operators just big-step both of their subexpressions, then apply whatever the primitive operator is:

```
e1 bop e2 ==> v
  if e1 ==> v1
  and e2 ==> v2
  and v is the result of primitive operation v1 bop v2
```

If expressions big step the guard, then big step one of the branches:

```
if e1 then e2 else e3 ==> v2
  if e1 ==> true
  and e2 ==> v2

if e1 then e2 else e3 ==> v3
  if e1 ==> false
  and e3 ==> v3
```

Let expressions big step the binding expression, do a substitution, and big step the result of the substitution:

```
let x = e1 in e2 ==> v2
  if e1 ==> v1
  and e2{v1/x} ==> v2
```

Finally, variables do not big step, for the same reason as with the small step semantics—a well-typed program will never reach the point of attempting to evaluate a variable name:

```
x =/=>
```

11.3.5 Implementing the Big-Step Relation

The big-step evaluation relation is, if anything, even easier to implement than the small-step relation. It just recurses over the tree, evaluating subexpressions as required by the definition of ==>:

```
(** [eval_big e] is the [e ==> v] relation. *)
let rec eval_big (e : expr) : expr = match e with
  | Int _ | Bool _ -> e
  | Var _ -> failwith "Unbound variable"
  | Binop (bop, e1, e2) -> eval_bop bop e1 e2
  | Let (x, e1, e2) -> subst e2 (eval_big e1) x |> eval_big
  | If (e1, e2, e3) -> eval_if e1 e2 e3

(** [eval_bop bop e1 e2] is the [e] such that [e1 bop e2 ==> e]. *)
and eval_bop bop e1 e2 = match bop, eval_big e1, eval_big e2 with
  | Add, Int a, Int b -> Int (a + b)
  | Mult, Int a, Int b -> Int (a * b)
  | Leq, Int a, Int b -> Bool (a <= b)
  | _ -> failwith "Operator and operand type mismatch"

(** [eval_if e1 e2 e3] is the [e] such that [if e1 then e2 else e3 ==> e]. *)
and eval_if e1 e2 e3 = match eval_big e1 with
  | Bool true -> eval_big e2
  | Bool false -> eval_big e3
  | _ -> failwith "Guard of if must have type bool"
```

It's good engineering practice to factor out functions for each of the pieces of syntax, as we did above, unless the implementation can fit on just a single line in the main pattern match inside `eval_big`.

11.3.6 Substitution in SimPL

In the previous section, we posited a new notation `e'{e/x}`, meaning "the expression `e'` with `e` substituted for `x`." The intuition is that anywhere `x` appears in `e'`, we should replace `x` with `e`.

Let's give a careful definition of substitution for SimPL. For the most part, it's not too hard.

Constants have no variables appearing in them (e.g., `x` cannot syntactically occur in `42`), so substitution leaves them unchanged:

```
i{e/x} = i
b{e/x} = b
```

For **binary operators and if expressions**, all that substitution needs to do is to recurse inside the subexpressions:

```
(e1 bop e2){e/x} = e1{e/x} bop e2{e/x}
(if e1 then e2 else e3){e/x} = if e1{e/x} then e2{e/x} else e3{e/x}
```

Variables start to get a little trickier. There are two possibilities: either we encounter the variable `x`, which means we should do the substitution, or we encounter some other variable with a different name, say `y`, in which case we should not do the substitution:

```
x{e/x} = e
y{e/x} = y
```

The first of those cases, `x{e/x} = e`, is important to note: it's where the substitution operation finally takes place. Suppose, for example, we were trying to figure out the result of `(x + 42){1/x}`. Using the definitions from above,

```
  (x + 42){1/x}
= x{1/x} + 42{1/x}   by the bop case
= 1 + 42{1/x}        by the first variable case
= 1 + 42             by the integer case
```

Note that we are not defining the `-->` relation right now. That is, none of these equalities represents a step of evaluation. To make that concrete, suppose we were evaluating `let x = 1 in x + 42`:

```
    let x = 1 in x + 42
--> (x + 42){1/x}
  = 1 + 42
--> 43
```

There are two single steps here, one for the `let` and the other for `+`. But we consider the substitution to happen all at once, as part of the step that `let` takes. That's why we write `(x + 42){1/x} = 1 + 42`, not `(x + 42){1/x} --> 1 + 42`.

Finally, **let expressions** also have two cases, depending on the name of the bound variable:

```
(let x = e1 in e2){e/x}  =  let x = e1{e/x} in e2
(let y = e1 in e2){e/x}  =  let y = e1{e/x} in e2{e/x}
```

Both of those cases substitute `e` for `x` inside the binding expression `e1`. That's to ensure that expressions like `let x = 42 in let y = x in y` would evaluate correctly: `x` needs to be in scope inside the binding `y = x`, so we have to do a substitution there regardless of the name being bound.

But the first case does not do a substitution inside `e2`, whereas the second case does. That's so we *stop* substituting when we reach a shadowed name. Consider `let x = 5 in let x = 6 in x`. We know it would evaluate to `6` in OCaml because of shadowing. Here's how it would evaluate with our definitions of SimPL:

```
    let x = 5 in let x = 6 in x
--> (let x = 6 in x){5/x}
  = let x = 6{5/x} in x      ***
  = let x = 6 in x
--> x{6/x}
  = 6
```

On the line tagged `***` above, we've stopped substituting inside the body expression, because we reached a shadowed variable name. If we had instead kept going inside the body, we'd get a different result:

```
    let x = 5 in let x = 6 in x
--> (let x = 6 in x){5/x}
  = let x = 6{5/x} in x{5/x}      ***WRONG***
  = let x = 6 in 5
--> 5{6/x}
  = 5
```

Example 1:

```
let x = 2 in x + 1
--> (x + 1){2/x}
  = 2 + 1
--> 3
```

Example 2:

```
    let x = 0 in (let x = 1 in x)
--> (let x = 1 in x){0/x}
  = (let x = 1{0/x} in x)
  = (let x = 1 in x)
--> x{1/x}
  = 1
```

Example 3:

```
    let x = 0 in x + (let x = 1 in x)
--> (x + (let x = 1 in x)){0/x}
  = x{0/x} + (let x = 1 in x){0/x}
  = 0 + (let x = 1{0/x} in x)
  = 0 + (let x = 1 in x)
--> 0 + x{1/x}
  = 0 + 1
--> 1
```

11.3.7 Implementing Substitution

The definitions above are easy to turn into OCaml code. Note that, although we write `v` below, the function is actually able to substitute any expression for a variable, not just a value. The interpreter will only ever call this function on a value, though.

```
(** [subst e v x] is [e] with [v] substituted for [x], that
    is, [e{v/x}]. *)
let rec subst e v x = match e with
  | Var y -> if x = y then v else e
  | Bool _ -> e
  | Int _ -> e
  | Binop (bop, e1, e2) -> Binop (bop, subst e1 v x, subst e2 v x)
  | Let (y, e1, e2) ->
    let e1' = subst e1 v x in
    if x = y
    then Let (y, e1', e2)
    else Let (y, e1', subst e2 v x)
  | If (e1, e2, e3) ->
    If (subst e1 v x, subst e2 v x, subst e3 v x)
```

11.3.8 The SimPL Interpreter is Done!

We've completed developing our SimPL interpreter. Recall that the finished interpreter can be downloaded here: simpl.zip. It includes some rudimentary test cases, as well as makefile targets that you will find helpful.

11.3.9 Capture-Avoiding Substitution

The definition of substitution for SimPL was a little tricky but not too complicated. It turns out, though, that for other languages, the definition gets more complicated.

Let's consider this tiny language:

```
e ::= x | e1 e2 | fun x -> e
v ::= fun x -> e
```

It is known as the *lambda calculus*. There are only three kinds of expressions in it: variables, function application, and anonymous functions. The only values are anonymous functions. The language isn't even typed. Yet, one of its most remarkable properties is that it *computationally universal:* it can express any computable function. (To learn more about that, read about the *Church-Turing Hypothesis*.)

Defining a big-step evaluation relation for the lambda calculus is straightforward. In fact, there's only one rule required:

```
e1 e2 ==> v
  if e1 ==> fun x -> e
  and e2 ==> v2
  and e{v2/x} ==> v
```

That rule is named *call by value*, because it requires arguments to be reduced to a value before a function can be applied. If that seems obvious, it's because you're used to it from OCaml. Other languages use other rules. For example, Haskell uses a variant on *call by name*, which is this rule:

```
e1 e2 ==> v
  if e1 ==> fun x -> e
  and e{e2/x} ==> v
```

With call by name, `e2` does not have to be reduced to a value; that can lead to greater efficiency if the value of `e2` is never needed.

Now we need to define the substitution operation for the lambda calculus. We'd like a definition that works for either call by name or call by value. Inspired by our definition for SimPL, here's the beginning of a definition:

```
x{e/x} = e
y{e/x} = y
(e1 e2){e/x} = e1{e/x} e2{e/x}
```

The first two lines are exactly how we defined variable substitution in SimPL. The next line resembles how we defined binary operator substitution; we just recurse into the subexpressions.

What about substitution in a function? In SimPL, we stopped substituting when we reached a bound variable of the same name; otherwise, we proceeded. In the lambda calculus, that idea would be stated as follows:

```
(fun x -> e'){e/x} = fun x -> e'
(fun y -> e'){e/x} = fun y -> e'{e/x}
```

Perhaps surprisingly, that definition turns out to be incorrect. Here's why: it violates the Principle of Name Irrelevance. Suppose we were attempting this substitution:

```
(fun z -> x){z/x}
```

The result would be:

```
  fun z -> x{z/x}
= fun z -> z
```

And, suddenly, a function that was *not* the identity function becomes the identity function. Whereas, if we had attempted this substitution:

```
(fun y -> x){z/x}
```

The result would be:

```
  fun y -> x{z/x}
= fun y -> z
```

Which is not the identity function. So our definition of substitution inside anonymous functions is incorrect, because it *captures* variables. A variable name being substituted inside an anonymous function can accidentally be "captured" by the function's argument name.

Note that we never had this problem in SimPL, in part because SimPL was typed. The function `fun y -> z` if applied to any argument would just return `z`, which is an unbound variable. But the lambda calculus is untyped, so we can't rely on typing to rule out this possibility here. Moreover, with rules such as call by name, we might well end up needing to evaluate such expressions.

So the question becomes, how do we define substitution so that it gets the right answer, without capturing variables? The answer is called *capture-avoiding substitution*, and a correct definition of it eluded mathematicians for centuries.

A correct definition is as follows:

```
(fun x -> e'){e/x} = fun x -> e'
(fun y -> e'){e/x} = fun y -> e'{e/x}  if y is not in FV(e)
```

where `FV(e)` means the "free variables" of `e`, i.e., the variables that are not bound in it, and is defined as follows:

```
FV(x) = {x}
FV(e1 e2) = FV(e1) + FV(e2)
FV(fun x -> e) = FV(e) - {x}
```

and + means set union, and − means set difference.

That definition prevents the substitution `(fun z -> x){z/x}` from occurring, because `z` is in `FV(z)`.

Unfortunately, because of the side-condition `y is not in FV(e)`, the substitution operation is now *partial*: there are times, like the example we just gave, where it cannot be applied.

That problem can be solved by changing the names of variables: if we detect that a partiality has been encountered, we can change the name of the function's argument. For example, when `(fun z -> x){z/x}` is encountered, the function's argument could be replaced with a new name `w` that doesn't occur anywhere else, yielding `(fun w -> x){z/x}`. (And if `z` occurred anywhere in the body, it would be replaced by `w`, too.) This is *replacement*, not substitution: absolutely anywhere we see `z`, we replace it with `w`. Then the substitution may proceed and correctly produce `fun w -> z`.

The tricky part of that is how to pick a new name that doesn't occur anywhere else, that is, how to pick a *fresh* name. Here are three strategies:

1. Pick a new variable name, check whether is fresh or not, and if not, try again, until that succeeds. For example, if trying to replace `z`, you might first try `z'`, then `z''`, etc.
2. Augment the evaluation relation to maintain a stream (i.e., infinite list) of unused variable names. Each time you need a new one, take the head of the stream. But you have to be careful to use the tail of the stream anytime after that. To guarantee that they are unused, reserve some variable names for use by the interpreter alone, and make them illegal as variable names chosen by the programmer. For example, you might decide that programmer variable names may never start with the character `$`, then have a stream `<$x1, $x2, $x3, ...>` of fresh names.
3. Use an imperative counter to simulate the stream from the previous strategy. For example, the following function is guaranteed to return a fresh variable name each time it is called:

```
let gensym =
  let counter = ref 0 in
  fun () -> incr counter; "$x" ^ string_of_int !counter
```

   The name `gensym` is traditional for this kind of function. It comes from LISP, and shows up throughout compiler implementations. It means generate a fresh symbol.

There is a complete implementation of an interpreter for the lambda calculus, including capture-avoiding substitution, that you can download: lambda-subst.zip. It uses the `gensym` strategy from above the generate fresh names. There is a definition named `strategy` in `main.ml` that you can use to switch between call-by-value and call-by-name.

11.3.10 Core OCaml

Let's now upgrade from SimPL and the lambda calculus to a larger language that we call *core OCaml*. Here is its syntax in BNF:

```
e ::= x | e1 e2 | fun x -> e
    | i | b | e1 bop e2
    | (e1, e2) | fst e | snd e
    | Left e | Right e
    | match e with Left x1 -> e1 | Right x2 -> e2
    | if e1 then e2 else e3
    | let x = e1 in e2

bop ::= + | * | < | =

x ::= <identifiers>

i ::= <integers>

b ::= true | false

v ::= fun x -> e | i | b | (v1, v2) | Left v | Right v
```

To keep tuples simple in this core model, we represent them with only two components (i.e., they are pairs). A longer tuple could be coded up with nested pairs. For example, `(1, 2, 3)` in OCaml could be `(1, (2, 3))` in this core language.

Also to keep variant types simple in this core model, we represent them with only two constructors, which we name `Left` and `Right`. A variant with more constructors could be coded up with nested applications of those two constructors. Since we have only two constructors, match expressions need only two branches. One caution in reading the BNF above: the occurrence of `|` in the match expression just before the `Right` constructor denotes syntax, not metasyntax.

There are a few important OCaml constructs omitted from this core language, including recursive functions, exceptions, mutability, and modules. Types are also missing; core OCaml does not have any type checking. Nonetheless, there is enough in this core language to keep us entertained.

11.3.11 Evaluating Core OCaml in the Substitution Model

Let's define the small and big step relations for Core OCaml. To be honest, there won't be much that's surprising at this point; we've seen just about everything already in SimPL and in the lambda calculus.

Small-Step Relation. Here is the fragment of Core OCaml we already know from SimPL:

```
e1 + e2 --> e1' + e2
        if e1 --> e1'

v1 + e2 --> v1 + e2'
        if e2 --> e2'

i1 + i2 --> i3
        where i3 is the result of applying primitive operation +
        to i1 and i2

if e1 then e2 else e3 --> if e1' then e2 else e3
        if e1 --> e1'

if true then e2 else e3 --> e2

if false then e2 else e3 --> e3

let x = e1 in e2 --> let x = e1' in e2
        if e1 --> e1'

let x = v in e2 --> e2{v/x}
```

Here's the fragment of Core OCaml that corresponds to the lambda calculus:

```
e1 e2 --> e1' e2
        if e1 --> e1'

v1 e2 --> v1 e2'
        if e2 --> e2'

(fun x -> e) v2 --> e{v2/x}
```

And here are the new parts of Core OCaml. First, **pairs** evaluate their first component, then their second component:

```
(e1, e2) --> (e1', e2)
        if e1 --> e1'

(v1, e2) --> (v1, e2')
        if e2 --> e2'

fst (v1,v2) --> v1

snd (v1,v2) --> v2
```

Constructors evaluate the expression they carry:

```
Left e --> Left e'
        if e --> e'

Right e --> Right e'
        if e --> e'
```

Pattern matching evaluates the expression being matched, then reduces to one of the branches:

```
match e with Left x1 -> e1 | Right x2 -> e2
--> match e' with Left x1 -> e1 | Right x2 -> e2
        if e --> e'

match Left v with Left x1 -> e1 | Right x2 -> e2
--> e1{v/x1}

match Right v with Left x1 -> e1 | Right x2 -> e2
--> e2{v/x2}
```

Substitution. We also need to define the substitution operation for Core OCaml. Here is what we already know from SimPL and the lambda calculus:

```
i{v/x} = i

b{v/x} = b

(e1 + e2) {v/x} = e1{v/x} + e2{v/x}

(if e1 then e2 else e3){v/x}
 = if e1{v/x} then e2{v/x} else e3{v/x}

(let x = e1 in e2){v/x} = let x = e1{v/x} in e2

(let y = e1 in e2){v/x} = let y = e1{v/x} in e2{v/x}
  if y not in FV(v)

x{v/x} = v

y{v/x} = y

(e1 e2){v/x} = e1{v/x} e2{v/x}

(fun x -> e'){v/x} = (fun x -> e')

(fun y -> e'){v/x} = (fun y -> e'{v/x})
  if y not in FV(v)
```

Note that we've now added the requirement of capture-avoiding substitution to the definitions for `let` and `fun`: they both require `y` not to be in the free variables of `v`. We therefore need to define the free variables of an expression:

```
FV(x) = {x}
FV(e1 e2) = FV(e1) + FV(e2)
FV(fun x -> e) = FV(e) - {x}
FV(i) = {}
FV(b) = {}
FV(e1 bop e2) = FV(e1) + FV(e2)
FV((e1,e2)) = FV(e1) + FV(e2)
```

(continues on next page)

(continued from previous page)

```
FV(fst e1) = FV(e1)
FV(snd e2) = FV(e2)
FV(Left e) = FV(e)
FV(Right e) = FV(e)
FV(match e with Left x1 -> e1 | Right x2 -> e2)
 = FV(e) + (FV(e1) - {x1}) + (FV(e2) - {x2})
FV(if e1 then e2 else e3) = FV(e1) + FV(e2) + FV(e3)
FV(let x = e1 in e2) = FV(e1) + (FV(e2) - {x})
```

Finally, we define substitution for the new syntactic forms in Core OCaml. Expressions that do not bind variables are easy to handle:

```
(e1,e2){v/x} = (e1{v/x}, e2{v/x})

(fst e){v/x} = fst (e{v/x})

(snd e){v/x} = snd (e{v/x})

(Left e){v/x} = Left (e{v/x})

(Right e){v/x} = Right (e{v/x})
```

Match expressions take a little more work, just like let expressions and anonymous functions, to make sure we get capture-avoidance correct:

```
(match e with Left x1 -> e1 | Right x2 -> e2){v/x}
 = match e{v/x} with Left x1 -> e1{v/x} | Right x2 -> e2{v/x}
     if ({x1,x2} intersect FV(v)) = {}

(match e with Left x -> e1 | Right x2 -> e2){v/x}
 = match e{v/x} with Left x -> e1 | Right x2 -> e2{v/x}
     if ({x2} intersect FV(v)) = {}

(match e with Left x1 -> e1 | Right x -> e2){v/x}
 = match e{v/x} with Left x1 -> e1{v/x} | Right x -> e2
      if ({x1} intersect FV(v)) = {}

(match e with Left x -> e1 | Right x -> e2){v/x}
 = match e{v/x} with Left x -> e1 | Right x -> e2
```

We wouldn't actually have to worry about capture-avoiding substitution in all the above rules as long as we are content with call-by-value semantics. But if we ever wanted call-by-name, we'd need all the extra conditions about free variables that we gave above.

11.3.12 Big-Step Relation

At this point there aren't any new concepts remaining to introduce. We can just give the rules:

```
e1 e2 ==> v
  if e1 ==> fun x -> e
  and e2 ==> v2
  and e{v2/x} ==> v
```

(continues on next page)

(continued from previous page)

```
fun x -> e ==> fun x -> e

i ==> i

b ==> b

e1 bop e2 ==> v
  if e1 ==> v1
  and e2 ==> v2
  and v is the result of primitive operation v1 bop v2

(e1, e2) ==> (v1, v2)
  if e1 ==> v1
  and e2 ==> v2

fst e ==> v1
  if e ==> (v1, v2)

snd e ==> v2
  if e ==> (v1, v2)

Left e ==> Left v
  if e ==> v

Right e ==> Right v
  if e ==> v

match e with Left x1 -> e1 | Right x2 -> e2 ==> v
  if e ==> Left v1
  and e1{v1/x1} ==> v

match e with Left x1 -> e1 | Right x2 -> e2 ==> v
  if e ==> Right v2
  and e2{v2/x2} ==> v

if e1 then e2 else e3 ==> v
  if e1 ==> true
  and e2 ==> v

if e1 then e2 else e3 ==> v
  if e1 ==> false
  and e3 ==> v

let x = e1 in e2 ==> v
  if e1 ==> v1
  and e2{v1/x} ==> v
```

11.4 Environment Model

So far we've been using the substitution model to evaluate programs. It's a great mental model for evaluation, and it's commonly used in programming languages theory.

But when it comes to implementation, the substitution model is not the best choice. It's too *eager*: it substitutes for every occurrence of a variable, even if that occurrence will never be needed. For example, `let x = 42 in e` will require crawling over all of `e`, which might be a very large expression, even if `x` never occurs in `e`, or even if `x` occurs only inside a branch of an if expression that never ends up being evaluated.

For sake of efficiency, it would be better to substitute *lazily*: only when the value of a variable is needed should the interpreter have to do the substitution. That's the key idea behind the *environment model*. In this model, there is a data structure called the *dynamic environment*, or just "environment" for short, that is a dictionary mapping variable names to values. Whenever the value of a variable is needed, it's looked up in that dictionary.

To account for the environment, the evaluation relation needs to change. Instead of `e --> e'` or `e ==> v`, both of which are binary relations, we now need a ternary relation, which is either

- `<env, e> --> e`, or
- `<env, e> ==> v`,

where `env` denotes the environment, and `<env, e>` is called a *machine configuration*. That configuration represents the state of the computer as it evaluates a program: `env` represents a part of the computer's memory (the binding of variables to values), and `e` represents the program.

As notation, let:

- `{}` represent the empty environment,
- `{x1:v1, x2:v2, ...}` represent the environment that binds `x1` to `v1`, etc.,
- `env[x -> v]` represent the environment `env` with the variable `x` additionally bound to the value `v`, and
- `env(x)` represent the binding of `x` in `env`.

If we wanted a more mathematical notation we would write $\mapsto$ instead of `->` in `env[x -> v]`, but we're aiming for notation that is easily typed on a standard keyboard.

We'll concentrate in the rest of this chapter on the big-step version of the environment model. It would of course be possible to define a small-step version, too.

11.4.1 Evaluating the Lambda Calculus in the Environment Model

Recall that the lambda calculus is the fragment of a functional language involving functions and application:

```
e ::= x | e1 e2 | fun x -> e

v ::= fun x -> e
```

Let's explore how to define a big-step evaluation relation for the lambda calculus in the environment model. The rule for variables just says to look up the variable name in the environment:

```
<env, x> ==> env(x)
```

This rule for functions says that an anonymous function evaluates just to itself. After all, functions are values:

```
<env, fun x -> e> ==> fun x -> e
```

Finally, this rule for application says to evaluate the left-hand side `e1` to a function `fun x -> e`, the right-hand side to a value `v2`, then to evaluate the body `e` of the function in an extended environment that maps the function's argument `x` to `v2`:

```
<env, e1 e2> ==> v
  if <env, e1> ==> fun x -> e
  and <env, e2> ==> v2
  and <env[x -> v2], e> ==> v
```

Seems reasonable, right? The problem is, **it's wrong.** At least, it's wrong if you want evaluation to behave the same as OCaml. Or, to be honest, nearly any other modern language.

It will be easier to explain why it's wrong if we add two more language feature: let expressions and integer constants. Integer constants would evaluate to themselves:

```
<env, i> ==> i
```

As for let expressions, recall that we don't actually *need* them, because `let x = e1 in e2` can be rewritten as `(fun x -> e2) e1`. Nonetheless, their semantics would be:

```
<env, let x = e1 in e2> ==> v
  if <env, e1> ==> v1
  and <env[x -> v1], e2> ==> v
```

Which is a rule that really just follows from the other rules above, using that rewriting.

What would this expression evaluate to?

```
let x = 1 in
let f = fun y -> x in
let x = 2 in
f 0
```

According to our semantics thus far, it would evaluate as follows:

- `let x = 1` would produce the environment `{x:1}`.
- `let f = fun y -> x` would produce the environment `{x:1, f:(fun y -> x)}`.
- `let x = 2` would produce the environment `{x:2, f:(fun y -> x)}`. Note how the binding of `x` to `1` is shadowed by the new binding.
- Now we would evaluate `<{x:2, f:(fun y -> x)}, f 0>`:

  ```
  <{x:2, f:(fun y -> x)}, f 0> ==> 2
    because <{x:2, f:(fun y -> x)}, f> ==> fun y -> x
    and <{x:2, f:(fun y -> x)}, 0> ==> 0
    and <{x:2, f:(fun y -> x)}[y -> 0], x> ==> 2`
      because <{x:2, f:(fun y -> x), y:0}, x> ==> 2`
  ```

- The result is therefore `2`.

But according to utop (and the substitution model), it evaluates as follows:

```
# let x = 1 in
  let f = fun y -> x in
  let x = 2 in
```

(continues on next page)

(continued from previous page)

```
  f 0;;
- : int = 1
```

And the result is therefore `1`. Obviously, `1` and `2` are different answers!

What went wrong?? It has to do with scope.

11.4.2 Lexical vs. Dynamic Scope

There are two different ways to understand the scope of a variable: variables can be *dynamically* scoped or *lexically* scoped. It all comes down to the environment that is used when a function body is being evaluated:

- With the **rule of dynamic scope**, the body of a function is evaluated in the current dynamic environment at the time the function is applied, not the old dynamic environment that existed at the time the function was defined.
- With the **rule of lexical scope**, the body of a function is evaluated in the old dynamic environment that existed at the time the function was defined, not the current environment when the function is applied.

The rule of dynamic scope is what our semantics, above, implemented. Let's look back at the semantics of function application:

```
<env, e1 e2> ==> v
  if <env, e1> ==> fun x -> e
  and <env, e2> ==> v2
  and <env[x -> v2], e> ==> v
```

Note how the body `e` is being evaluated in the same environment `env` as when the function is applied. In the example program

```
let x = 1 in
let f = fun y -> x in
let x = 2 in
f 0
```

that means that `f` is evaluated in an environment in which `x` is bound to `2`, because that's the most recent binding of `x`.

But OCaml implements the rule of lexical scope, which coincides with the substitution model. With that rule, `x` is bound to `1` in the body of `f` when `f` is defined, and the later binding of `x` to `2` doesn't change that fact.

The consensus after decades of experience with programming language design is that lexical scope is the right choice. Perhaps the main reason for that is that lexical scope supports the Principle of Name Irrelevance. Recall, that principle says that the name of a variable shouldn't matter to the meaning of program, as long as the name is used consistently.

Nonetheless, dynamic scope is useful in some situations. Some languages use it as the norm (e.g., Emacs LISP, LaTeX), and some languages have special ways to do it (e.g., Perl, Racket). But these days, most languages just don't have it.

There is one language feature that modern languages *do* have that resembles dynamic scope, and that is exceptions. Exception handling resembles dynamic scope, in that raising an exception transfers control to the "most recent" exception handler, just like how dynamic scope uses the "most recent" binding of variable.

11.4.3 A Second Attempt at Evaluating the Lambda Calculus in the Environment Model

The question then becomes, how do we implement lexical scope? It seems to require time travel, because function bodies need to be evaluated in old dynamic environment that have long since disappeared.

The answer is that the language implementation must arrange to keep old environments around. And that is indeed what OCaml and other languages must do. They use a data structure called a *closure* for this purpose.

A closure has two parts:

- a *code* part, which contains a function `fun x -> e`, and
- an *environment* part, which contains the environment `env` at the time that function was defined.

You can think of a closure as being like a pair, except that there's no way to directly write a closure in OCaml source code, and there's no way to destruct the pair into its components in OCaml source code. The pair is entirely hidden from you by the language implementation.

Let's notate a closure as `(| fun x -> e, env |)`. The delimiters `(| ... |)` are meant to evoke an OCaml pair, but of course they are not legal OCaml syntax.

Using that notation, we can re-define the evaluation relation as follows:

The rule for functions now says that an anonymous function evaluates to a closure:

```
<env, fun x -> e> ==> (| fun x -> e, env |)
```

That rule saves the defining environment as part of the closure, so that it can be used at some future point.

The rule for application says to use that closure:

```
<env, e1 e2> ==> v
  if <env, e1> ==> (| fun x -> e, defenv |)
  and <env, e2> ==> v2
  and <defenv[x -> v2], e> ==> v
```

That rule uses the closure's environment `defenv` (whose name is meant to suggest the "defining environment") to evaluate the function body `e`.

The derived rule for let expressions remains unchanged:

```
<env, let x = e1 in e2> ==> v
  if <env, e1> ==> v1
  and <env[x -> v1], e2> ==> v
```

That's because the defining environment for the body `e2` is the same as the current environment `env` when the let expression is being evaluated.

11.4.4 An Implementation of SimPL in the Environment Model

You can download a complete implementation of the two semantics above: lambda-env.zip In `main.ml`, there is a definition named `scope` that you can use to switch between lexical and dynamic scope.

11.4.5 Evaluating Core OCaml in the Environment Model

There isn't anything new in the (big step) environment model semantics of Core OCaml, now that we know about closures, but for sake of completeness let's state it anyway.

Syntax.

```
e ::= x | e1 e2 | fun x -> e
    | i | b | e1 + e2
    | (e1,e2) | fst e1 | snd e2
    | Left e | Right e
    | match e with Left x1 -> e1 | Right x2 -> e2
    | if e1 then e2 else e3
    | let x = e1 in e2
```

Semantics.

We've already seen the semantics of the lambda calculus fragment of Core OCaml:

```
<env, x> ==> v
  if env(x) = v

<env, e1 e2> ==> v
  if  <env, e1> ==> (| fun x -> e, defenv |)
  and <env, e2> ==> v2
  and <defenv[x -> v2], e> ==> v

<env, fun x -> e> ==> (|fun x -> e, env|)
```

Evaluation of constants ignores the environment:

```
<env, i> ==> i

<env, b> ==> b
```

Evaluation of most other language features just uses the environment without changing it:

```
<env, e1 + e2> ==> n
  if  <env,e1> ==> n1
  and <env,e2> ==> n2
  and n is the result of applying the primitive operation + to n1 and n2

<env, (e1, e2)> ==> (v1, v2)
  if  <env, e1> ==> v1
  and <env, e2> ==> v2

<env, fst e> ==> v1
  if <env, e> ==> (v1, v2)

<env, snd e> ==> v2
```

(continues on next page)

(continued from previous page)

```
  if <env, e> ==> (v1, v2)

<env, Left e> ==> Left v
  if <env, e> ==> v

<env, Right e> ==> Right v
  if <env, e> ==> v

<env, if e1 then e2 else e3> ==> v2
  if <env, e1> ==> true
  and <env, e2> ==> v2

<env, if e1 then e2 else e3> ==> v3
  if <env, e1> ==> false
  and <env, e3> ==> v3
```

Finally, evaluation of binding constructs (i.e., match and let expression) extends the environment with a new binding:

```
<env, match e with Left x1 -> e1 | Right x2 -> e2> ==> v1
  if  <env, e> ==> Left v
  and <env[x1 -> v], e1> ==> v1

<env, match e with Left x1 -> e1 | Right x2 -> e2> ==> v2
  if  <env, e> ==> Right v
  and <env[x2 -> v], e2> ==> v2

<env, let x = e1 in e2> ==> v2
  if  <env, e1> ==> v1
  and <env[x -> v1], e2> ==> v2
```

11.5 Type Checking

Earlier, we skipped over the type checking phase. Let's come back to that now. After lexing and parsing, the next phase of compilation is semantic analysis, and the primary task of semantic analysis is type checking.

A *type system* is a mathematical description of how to determine whether an expression is *ill typed* or *well typed*, and in the latter case, what the type of the expression is. A *type checker* is a program that implements a type system, i.e., that implements the static semantics of the language.

Commonly, a type system is formulated as a ternary relation *HasType*(Γ, e, t), which means that expression e has type t in static environment Γ. A *static environment*, aka *typing context*, is a map from identifiers to types. The static environment is used to record what variables are in scope, and what their types are. The use of the Greek letter Γ for static environments is traditional.

That ternary relation *HasType* is typically written with infix notation, though, as $\Gamma \vdash e : t$. You can read the turnstile symbol $\vdash$ as "proves" or "shows", i.e., the static environment Γ shows that e has type t.

Let's make that notation a little friendlier by eliminating the Greek and the math typesetting. We'll just write `env |- e : t` to mean that static environment `env` shows that `e` has type `t`. We previously used `env` to mean a dynamic environment in the big-step relation `==>`. Since it's always possible to see whether we're using the `==>` or `|-` relation, the meaning of `env` as either a dynamic or static environment is always discernible.

Let's write `{}` for the empty static environment, and `x:t` to mean that `x` is bound to `t`. So, `{foo:int, bar:bool}` would be the static environment is which `foo` has type `int` and `bar` has type `bool`. A static environment may bind an identifier at most once. We'll write `env[x -> t]` to mean a static environment that contains all the bindings of `env`,

and also binds `x` to `t`. If `x` was already bound in `env`, then that old binding is replaced by the new binding to `t` in `env[x -> t]`. As with dynamic environments, if we wanted a more mathematical notation we would write $\mapsto$ instead of `->` in `env[x -> v]`, but we're aiming for notation that is easily typed on a standard keyboard.

With all that machinery, we can at last define what it means to be well typed: An expression `e` is **well typed** in static environment `env` if there exists a type `t` for which `env |- e : t`. The goal of a type checker is thus to find such a type `t`, starting from some initial static environment.

It's convenient to pretend that the initial static environment is empty. But in practice, it's rare that a language truly uses the empty static environment to determine whether a program is well typed. In OCaml, for example, there are many built-in identifiers that are always in scope, such as everything in the `Stdlib` module.

11.5.1 A Type System for SimPL

Recall the syntax of SimPL:

```
e ::= x | i | b | e1 bop e2
    | if e1 then e2 else e3
    | let x = e1 in e2

bop ::= + | * | <=
```

Let's define a type system `env |- e : t` for SimPL. The only types in SimPL are integers and booleans:

```
t ::= int | bool
```

To define `|-`, we'll invent a set of *typing rules* that specify what the type of an expression is based on the types of its subexpressions. In other words, `|-` is an *inductively-defined relation*, as can be learned about in a discrete math course. So, it has some base cases, and some inductive cases.

For the base cases, an integer constant has type `int` in any static environment whatsoever, a Boolean constant likewise always has type `bool`, and a variable has whatever type the static environment says it should have. Here are the typing rules that express those ideas:

```
env |- i : int
env |- b : bool
{x : t, ...} |- x : t
```

The remaining syntactic forms are inductive cases.

Let. As we already know from OCaml, we type check the body of a let expression using a scope that is extended with a new binding.

```
env |- let x = e1 in e2 : t2
  if env |- e1 : t1
  and env[x -> t1] |- e2 : t2
```

The rule says that `let x = e1 in e2` has type `t2` in static environment `env`, but only if certain conditions hold. The first condition is that `e1` has type `t1` in `env`. The second is that `e2` has type `t2` in a new static environment, which is `env` extended to bind `x` to `t1`.

Binary operators. We'll need a couple different rules for binary operators.

```
env |- e1 bop e2 : int
  if bop is + or *
```

(continues on next page)

(continued from previous page)

```
  and env |- e1 : int
  and env |- e2 : int

env |- e1 <= e2 : bool
  if env |- e1 : int
  and env |- e2 : int
```

If. Just like OCaml, an if expression must have a Boolean guard, and its two branches must have the same type.

```
env |- if e1 then e2 else e3 : t
  if env |- e1 : bool
  and env |- e2 : t
  and env |- e3 : t
```

11.5.2 A Type Checker for SimPL

Let's implement a type checker for SimPL, based on the type system we defined in the previous section. You can download the completed type checker as part of the SimPL interpreter: simpl.zip

We need a variant to represent types:

```
type typ =
  | TInt
  | TBool
```

The natural name for that variant would of course have been "type" not "typ", but the former is already a keyword in OCaml. We have to prefix the constructors with "T" to disambiguate them from the constructors of the `expr` type, which include `Int` and `Bool`.

Let's introduce a small signature for static environments, based on the abstractions we've introduced so far: the empty static environment, looking up a variable, and extending a static environment.

```
module type StaticEnvironment = sig
  (** [t] is the type of a static environment. *)
  type t

  (** [empty] is the empty static environment. *)
  val empty : t

  (** [lookup env x] gets the binding of [x] in [env].
      Raises: [Failure] if [x] is not bound in [env]. *)
  val lookup : t -> string -> typ

  (** [extend env x ty] is [env] extended with a binding
      of [x] to [ty]. *)
  val extend : t -> string -> typ -> t
end
```

It's easy to implement that signature with an association list.

```
module StaticEnvironment : StaticEnvironment = struct
  type t = (string * typ) list
```

(continues on next page)

(continued from previous page)

```
  let empty = []

  let lookup env x =
    try List.assoc x env
    with Not_found -> failwith "Unbound variable"

  let extend env x ty =
    (x, ty) :: env
end
```

Now we can implement the typing relation `|-`. We'll do that by writing a function `typeof : StaticEnvironment.t -> expr -> typ`, such that `typeof env e = t` if and only if `env |- e : t`. Note that the `typeof` function produces the type as output, so the function is actually inferring the type! That inference is easy for SimPL; it would be considerably harder for larger languages.

Let's start with the base cases:

```
open StaticEnvironment

(** [typeof env e] is the type of [e] in static environment [env].
    Raises: [Failure] if [e] is not well typed in [env]. *)
let rec typeof env = function
  | Int _ -> TInt
  | Bool _ -> TBool
  | Var x -> lookup env x
  ...
```

Note how the implementation of `typeof` so far is based on the rules we previously defined for `|-`. In particular:

- `typeof` is a recursive function, just as `|-` is an inductive relation.
- The base cases for the recursion of `typeof` are the same as the base cases for `|-`.

Also note how the implementation of `typeof` differs in one major way from the definition of `|-`: error handling. The type system didn't say what to do about errors; rather, it just defined what it meant to be well typed. The type checker, on the other hand, needs to take action and report ill typed programs. Our `typeof` function does that by raising exceptions. The `lookup` function, in particular, will raise an exception if we attempt to lookup a variable that hasn't been bound in the static environment.

Let's continue with the recursive cases:

```
  ...
  | Let (x, e1, e2) -> typeof_let env x e1 e2
  | Binop (bop, e1, e2) -> typeof_bop env bop e1 e2
  | If (e1, e2, e3) -> typeof_if env e1 e2 e3
```

We're factoring out a helper function for each branch for the sake of keeping the pattern match readable. Each of the helpers directly encodes the ideas of the `|-` rules, with error handling added.

```
and typeof_let env x e1 e2 =
  let t1 = typeof env e1 in
  let env' = extend env x t1 in
  typeof env' e2

and typeof_bop env bop e1 e2 =
  let t1, t2 = typeof env e1, typeof env e2 in
```

(continues on next page)

(continued from previous page)

```
  match bop, t1, t2 with
  | Add, TInt, TInt
  | Mult, TInt, TInt -> TInt
  | Leq, TInt, TInt -> TBool
  | _ -> failwith "Operator and operand type mismatch"

and typeof_if env e1 e2 e3 =
  if typeof env e1 = TBool
  then begin
    let t2 = typeof env e2 in
    if t2 = typeof env e3 then t2
    else failwith "Branches of if must have same type"
  end
  else failwith "Guard of if must have type bool"
```

Note how the recursive calls in the implementation of `typeof` occur exactly in the same places where the definition of `|-` is inductive.

Finally, we can implement a function to check whether an expression is well typed:

```
(** [typecheck e] checks whether [e] is well typed in
    the empty static environment. Raises: [Failure] if not. *)
let typecheck e =
  ignore (typeof empty e)
```

11.5.3 Type Safety

What is the purpose of a type system? There might be many, but one of the primary purposes is to ensure that certain run-time errors don't occur. Now that we know how to formalize type systems with the `|-` relation and evaluation with the `-->` relation, we can make that idea precise.

The goals of a language designer usually include ensuring that these two properties, which establish a relationship between `|-` and `-->`, both hold:

- **Progress:** If an expression is well typed, then either it is already a value, or it can take at least one step. We can formalize that as, "for all `e`, if there exists a `t` such that `{} |- e : t`, then `e` is a value, or there exists an `e'` such that `e --> e'`."
- **Preservation:** If an expression is well typed, then if the expression steps, the new expression has the same type as the old expression. Formally, "for all `e` and `t` such that `{} |- e : t`, if there exists an `e'` such that `e --> e'`, then `{} |- e' : t`."

Put together, progress plus preservation imply that that evaluation of a well-typed expression can never *get stuck*, meaning it reaches a non-value that cannot take a step. This property is known as *type safety*.

For example, `5 + true` would get stuck using the SimPL evaluation relation, because the primitive + operation cannot accept a Boolean as an operand. But the SimPL type system won't accept that program, thus saving us from ever reaching that situation.

Looking back at the SimPL we wrote, everywhere in the implementation of `step` where we raised an exception was a place where evaluation would get stuck. But the type system guarantees those exceptions will never occur.

11.6 Type Inference

OCaml and Java are *statically typed* languages, meaning every binding has a type that is determined at *compile time*—that is, before any part of the program is executed. The type-checker is a compile-time procedure that either accepts or rejects a program. By contrast, JavaScript and Ruby are dynamically-typed languages; the type of a binding is not determined ahead of time. Computations like binding 42 to `x` and then treating `x` as a string therefore either result in run-time errors, or run-time conversion between types.

Unlike Java, OCaml is *implicitly typed*, meaning programmers rarely need to write down the types of bindings. This is often convenient, especially with higher-order functions. (Although some people disagree as to whether it makes code easier or harder to read). But implicit typing in no way changes the fact that OCaml is statically typed. Rather, the type-checker has to be more sophisticated because it must infer what the *type annotations* "would have been" had the programmers written all of them. In principle, type inference and type checking could be separate procedures (the inferencer could figure out the types then the checker could determine whether the program is well-typed), but in practice they are often merged into a single procedure called *type reconstruction*.

11.6.1 OCaml Type Reconstruction

At a very high level, OCaml's type reconstruction algorithm works as follows:

- Determine the types of definitions in order, using the types of earlier definitions to infer the types of later ones. (Which is one reason you may not use a name before it is bound in an OCaml program.)
- For each `let` definition, analyze the definition to determine *constraints* about its type. For example, if the inferencer sees `x + 1`, it concludes that `x` must have type `int`. It gathers similar constraints for function applications, pattern matches, etc. Think of these constraints as a system of equations like you might have in algebra.
- Use that system of equations to solve for the type of the name begin defined.

The OCaml type reconstruction algorithm attempts to never reject a program that could type check, if the programmer had written down types. It also attempts never to accept a program that cannot possibly type check. Some more obscure parts of the language can sometimes make type annotations either necessary or at least helpful (see *Real World OCaml* chapter 22, "Type inference", for examples). But for most code you write, type annotations really are completely optional.

Since it would be verbose to keep writing "the type reconstruction algorithm used by OCaml and other functional languages," we'll call the algorithm HM. That name is used throughout the programming languages literature, because the algorithm was independently invented by Roger Hindley and Robin Milner.

HM has been rediscovered many times by many people. Curry used it informally in the 1950's (perhaps even the 1930's). He wrote it up formally in 1967 (published 1969). Hindley discovered it independently in 1969; Morris in 1968; and Milner in 1978. In the realm of logic, similar ideas go back perhaps as far as Tarski in the 1920's. Commenting on this history, Hindley wrote,

> There must be a moral to this story of continual re-discovery; perhaps someone along the line should have learned to read. Or someone else learn to write.

Although we haven't seen the HM algorithm yet, you probably won't be surprised to learn that it's usually very efficient—you've probably never had to wait for the toplevel to print the inferred types of your programs. In practice, it runs in approximately linear time. But in theory, there are some very strange programs that can cause its running-time to blow up. (Technically, it's exponential time.) For fun, try typing the following code in utop:

```
# let b = true;;
# let f0 = fun x -> x + 1;;
# let f = fun x -> if b then f0 else fun y -> x y;;
# let f = fun x -> if b then f else fun y -> x y;;
# let f = fun x -> if b then f else fun y -> x y;;
(* keep repeating that last line *)
```

You'll see the types get longer and longer, and eventually (around 20 repetitions or so) type inference will cause a significant delay.

11.6.2 Constraint-Based Inference

Let's build up to the HM type inference algorithm by starting with this little language:

```
e ::= x | i | b | e1 bop e2
    | if e1 then e2 else e3
    | fun x -> e
    | e1 e2

bop ::= + | * | <=

t ::= int | bool | t1 -> t2
```

That language is SimPL, plus the lambda calculus, minus `let` expressions. It turns out `let` expressions add a extra layer of complication, so we'll come back to them later.

Since anonymous functions in this language do not have type annotations, we have to infer the type of the argument `x`. For example,

- In `fun x -> x + 1`, argument `x` must have type `int` hence the function has type `int -> int`.
- In `fun x -> if x then 1 else 0`, argument `x` must have type `bool` hence the function has type `bool -> int`.
- Function `fun x -> if x then x else 0` is untypeable, because it would require `x` to have both type `int` and `bool`, which isn't allowed.

A Syntactic Simplification. We can treat `e1 bop e2` as syntactic sugar for `( bop ) e1 e2`. That is, we treat infix binary operators as prefix function application. Let's introduce a new syntactic class `n` for *names*, which generalize identifiers and operators. That changes the syntax to:

```
e ::= n | i | b
    | if e1 then e2 else e3
    | fun x -> e
    | e1 e2

n ::= x | bop

bop ::= ( + ) | ( * ) | ( <= )

t ::= int | bool | t1 -> t2
```

We already know the types of those built-in operators:

```
( + ) : int -> int -> int
( * ) : int -> int -> int
( <= ) : int -> int -> bool
```

Those types are given; we don't have to infer them. They are part of the initial static environment. In OCaml those operator names could later be shadowed by values with different types, but here we don't have to worry about that because we don't yet have `let`.

How would *you* mentally infer the type of `fun x -> 1 + x`, or rather, `fun x -> ( + ) 1 x`? It's automatic by now, but we could break it down into pieces:

- Start with `x` having some unknown type `t`.
- Note that `( + )` is known to have type `int -> (int -> int)`.
- So its first argument must have type `int`. Which `1` does.
- And its second argument must have type `int`, too. So `t = int`. That is a *constraint* on `t`.
- Finally the body of the function must also have type `int`, since that's the return type of `( + )`.
- Therefore the type of the entire function must be `t -> int`.
- Since `t = int`, that type is `int -> int`.

The type inference algorithm follows the same idea of generating unknown types, collecting constraints on them, and using the constraints to solve for the type of the expression.

Let's introduce a new quaternary relation `env |- e : t -| C`, which should be read as follows: "in environment `env`, expression `e` is inferred to have type `t` and generates constraint set `C`." A constraint is an equation of the form `t1 = t2` for any types `t1` and `t2`.

If we think of the relation as a type-inference function, the colon in the middle separates the input from the output. The inputs are `env` and `e`: we want to know what the type of `e` is in environment `env`. The function returns as output a type `t` and constraints `C`.

The `e : t` in the middle of the relation is approximately what you see in the toplevel: you enter an expression, and it tells you the type. But around that is an environment and constraint set `env |- ... -| C` that is invisible to you. So, the turnstiles around the outside show the parts of type inference that the toplevel does not.

The easiest parts of inference are constants:

```
env |- i : int -| {}

env |- b : bool -| {}
```

Any integer constant `i`, such as `42`, is known to have type `int`, and there are no constraints generated. Likewise for Boolean constants.

Inferring the type of a name requires looking it up in the environment:

```
env |- n : env(n) -| {}
```

No constraints are generated.

If the name is not bound in the environment, the expression cannot be typed. It's an unbound name error.

The remaining rules are at their core the same as the type-checking rules we saw previously, but they each generate a *type variable* and possibly some constraints on that type variable.

If.

Here's the rule for `if` expressions:

```
env |- if e1 then e2 else e3 : 't -| C1, C2, C3, t1 = bool, 't = t2, 't = t3
  if fresh 't
  and env |- e1 : t1 -| C1
  and env |- e2 : t2 -| C2
  and env |- e3 : t3 -| C3
```

To infer the type of an `if`, we infer the types `t1`, `t2`, and `t3` of each of its subexpressions, along with any constraints on them. We have no control over what those types might be; it depends on what the programmer wrote. But we do know that the type of the guard must be `bool`. So we generate a constraint that `t1 = bool`.

Furthermore, we know that both branches must have the same type—though, we don't know in advance what that type might be. So, we invent a *fresh* type variable `'t` to stand for that type. A type variable is fresh if it has never been used elsewhere during type inference. So, picking a fresh type variable just means picking a new name that can't possibly be confused with any other names in the program. We return `'t` as the type of the `if`, and we record two constraints `'t = t2` and `'t = t3` to say that both branches must have that type.

We therefore need to add type variables to the syntax of types:

```
t ::= 'x | int | bool | t1 -> t2
```

Some example type variables include `'a`, `'foobar`, and `'t`. In the last, `t` is an identifier, not a meta-variable.

Here's an example:

```
{} |- if true then 1 else 0 : 't -| bool = bool, 't = int
  {} |- true : bool -| {}
  {} |- 1 : int -| {}
  {} |- 0 : int -| {}
```

The full constraint set generated is `{}, {}, {}, bool = bool, 't = int, 't = int`, but of course that simplifies to just `bool = bool, 't = int`. From that constraint set we can see that the type of `if true then 1 else 0` must be `int`.

Anonymous functions.

Since there is no type annotation on `x`, its type must be inferred:

```
env |- fun x -> e : 't1 -> t2 -| C
  if fresh 't1
  and env, x : 't1 |- e : t2 -| C
```

We introduce a fresh type variable `'t1` to stand for the type of `x`, and infer the type of body `e` under the environment in which `x : 't1`. Wherever `x` is used in `e`, that can cause constraints to be generated involving `'t1`. Those constraints will become part of `C`.

Here's a function where we can immediately see that `x : bool`, but let's work through the inference:

```
{} |- fun x -> if x then 1 else 0 : 't1 -> 't -| 't1 = bool, 't = int
  {}, x : 't1 |- if x then 1 else 0 : 't -| 't1 = bool, 't = int
    {}, x : 't1 |- x : 't1 -| {}
    {}, x : 't1 |- 1 : int -| {}
    {}, x : 't1 |- 0 : int -| {}
```

The inferred type of the function is `'t1 -> 't`, with constraints `'t1 = bool` and `'t = int`. Simplifying that, the function's type is `bool -> int`.

Function application.

The type of the entire application must be inferred, because we don't yet know anything about the types of either subexpression:

```
env |- e1 e2 : 't -| C1, C2, t1 = t2 -> 't
  if fresh 't
  and env |- e1 : t1 -| C1
  and env |- e2 : t2 -| C2
```

We introduce a fresh type variable `'t` for the type of the application expression. We use inference to determine the types of the subexpressions and any constraints they happen to generate. We add one new constraint, `t1 = t2 -> 't`,

which expresses that the type of the left-hand side `e1` must be a function that takes in an argument of type `t2` and returns a value of type `'t`.

Let `I` be the *initial environment* that binds the boolean operators. Let's infer the type of a partial application of `( + )`:

```
I |- ( + ) 1 : 't -| int -> int -> int = int -> 't
  I |- ( + ) : int -> int -> int -| {}
  I |- 1 : int -| {}
```

From the resulting constraint, we see that

```
int -> int -> int
=
int -> 't
```

Stripping the `int ->` off the left-hand side of each of those function types, we are left with

```
int -> int
=
't
```

Hence the type of `( + ) 1` is `int -> int`.

11.6.3 Solving Constraints

What does it mean to solve a set of constraints? Since constraints are equations on types, it's much like solving a system of equations in algebra. We want to solve for the values of the variables appearing in those equations. By substituting those values for the variables, we should get equations that are identical on both sides. For example, in algebra we might have:

```
5x + 2y =  9
 x -  y = -1
```

Solving that system, we'd get that `x = 1` and `y = 2`. If we substitute `1` for `x` and `2` for `y`, we get:

```
5(1) + 2(2) =  9
  1  -   2  = -1
```

which reduces to

```
 9 =  9
-1 = -1
```

In programming languages terminology (though perhaps not high-school algebra), we say that the substitutions `{1 / x}` and `{2 / y}` together *unify* that set of equations, because they make each equation "unite" such that its left side is identical to its right side.

Solving systems of equations on types is similar. Just as we found numbers to substitute for variables above, we now want to find types to substitute for type variables, and thereby unify the set of equations.

Much like the substitutions we defined before for the substitution model of evaluation, we'll write `{t / 'x}` for the *type substitution* that maps type variable `'x` to type `t`. For example, `{t2/'x} t1` means type `t1` with `t2` substituted for `'x`.

We can define substitution on types as follows:

```
int {t / 'x} = int
bool {t / 'x} = bool
'x {t / 'x} = t
'y {t / 'x} = 'y
(t1 -> t2) {t / 'x} =  (t1 {t / 'x} ) -> (t2 {t / 'x} )
```

Given two substitutions `S1` and `S2`, we write `S1; S2` to mean the substitution that is their *sequential composition*, which is defined as follows:

```
t (S1; S2) = (t S1) S2
```

The order matters. For example, `'x ({('y -> 'y) / 'x}; {bool / 'y})` is `bool -> bool`, not `'y -> 'y`. We can build up bigger and bigger substitutions this way.

A substitution `S` can be applied to a constraint `t = t'`. The result `(t = t') S` is defined to be `t S = t' S`. So we just apply the substitution on both sides of the constraint.

Finally a substitution can be applied to a set `C` of constraints; the result `C S` is the result of applying `S` to each of the individual constraints in `C`.

A substitution *unifies* a constraint `t_1 = t_2` if `t_1 S` results in the same type as `t_2 S`. For example, substitution `S = {int -> int / 'y}; {int / 'x}` unifies constraint `'x -> ('x -> int) = int -> 'y`, because

```
('x -> ('x -> int)) S
=
int -> (int -> int)
```

and

```
(int -> 'y) S
=
int -> (int -> int)
```

A substitution `S` unifies a set `C` of constraints if `S` unifies every constraint in `C`.

At last we can precisely say what it means to solve a set of constraints: we must find a substitution that unifies the set. That is, we need to find a sequence of maps from type variables to types, such that the sequence causes each equation in the constraint set to "unite", meaning that its left-hand side and right-hand side become the same.

To find a substitution that unifies constraint set `C`, we use an algorithm `unify`, which is defined as follows:

- If `C` is the empty set, then `unify(C)` is the empty substitution.
- If `C` contains at least one constraint `t1 = t2` and possibly some other constraints `C'`, then `unify(C)` is defined as follows:
  - If `t1` and `t2` are both the same simple type—i.e. both the same type variable `'x`, or both `int` or both `bool`— then return `unify(C')`. *In this case, the constraint contained no useful information, so we're tossing it out and continuing.*
  - If `t1` is a type variable `'x` and `'x` does not occur in `t2`, then let `S = {t2 / 'x}`, and return `S; unify(C' S)`. *In this case, we are eliminating the variable `'x` from the system of equations, much like Gaussian elimination in solving algebraic equations.*
  - If `t2` is a type variable `'x` and `'x` does not occur in `t1`, then let `S = {t1 / 'x}`, and return `S'; unify(C' S)`. *This is an elimination like the previous case.*

- If `t1 = i1 -> o1` and `t2 = i2 -> o2`, where `i1`, `i2`, `o1`, and `o2` are types, then `unify(i1 = i2, o1 = o2, C')`. *In this case, we break one constraint down into two smaller constraints and add those constraints back in to be further unified.*
- Otherwise, fail. There is no possible unifier.

In the second and third sub-cases, the check that `'x` should not occur in the type ensures that the algorithm is actually eliminating the variable. Otherwise, the algorithm could end up re-introducing the variable instead of eliminating it.

It's possible to prove that the unification algorithm always terminates, and that it produces a result if and only a unifier actually exists—that is, if and only if the set of constraints has a solution. Moreover, the solution the algorithm produces is the *most general unifier*, in the sense that if `S = unify(C)` and `S'` also unifies `C`, then there must exist some `S''` such that `S' = S; S''`. Such an `S'` is less general than `S` because it contains the additional substitutions of `S''`.

11.6.4 Finishing Type Inference

Let's review what we've done so far. We started with this language:

```
e ::= n | i | b
    | if e1 then e2 else e3
    | fun x -> e
    | e1 e2

n ::= x | bop

bop ::= ( + ) | ( * ) | ( <= )

t ::= int | bool | t1 -> t2
```

We then introduced an algorithm for inferring a type of an expression. That type came along with a set of constraints. The algorithm was expressed in the form of a relation `env |- e : t -| C`.

Next, we introduced the unification algorithm for solving constraint sets. That algorithm produces as output a sequence `S` of substitutions, or it fails. If it fails, then `e` is not typeable.

To finish type inference and reconstruct the type of `e`, we just compute `t S`. That is, we apply the solution to the constraints to the type `t` produced by constraint generation.

Let `p` be that type. That is, `p = t S`. It's possible to prove `p` is the *principal* type for the expression, meaning that if `e` also has type `t` for any other `t`, then there exists a substitution `S` such that `t = p S`.

For example, the principal type of the identity function `fun x -> x` would be `'a -> 'a`. But you could also give that function the less helpful type `int -> int`. What we're saying is that HM will produce `'a -> 'a`, not `int -> int`. So in a sense, HM actually infers the most "lenient" type that is possible for an expression.

A Worked Example. Let's infer the type of the following expression:

```
fun f -> fun x -> f (( + ) x 1)
```

It's not much code, but this will get quite involved!

We start in the initial environment `I` that, among other things, maps `( + )` to `int -> int -> int`.

```
I |- fun f -> fun x -> f (( + ) x 1)
```

For now we leave off the `: t -| C`, because that's the output of constraint generation. We haven't figure out the output yet! Since we have a function, we use the function rule for inference to proceed by introducing a fresh type variable for the argument:

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)  <-- Here
```

Again we have a function, hence a fresh type variable:

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1)  <-- Here
```

Now we have an application expression. Before dealing with it, we need to descend into its subexpressions. The first one is easy. It's just a variable. So we finally can finish a judgment with the variable's type from the environment, and an empty constraint set.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1)
      I, f : 'a, x : 'b |- f : 'a -| {}  <-- Here
```

Next is the second subexpression.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1)
      I, f : 'a, x : 'b |- f : 'a -| {}
      I, f : 'a, x : 'b |- ( + ) x 1  <-- Here
```

That is another application, so we need to handle its subexpressions. Recall that `( + ) x 1` is parsed as `(( + ) x) 1`. So the first subexpression is the complicated one to handle.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1)
      I, f : 'a, x : 'b |- f : 'a -| {}
      I, f : 'a, x : 'b |- ( + ) x 1
        I, f : 'a, x : 'b |- ( + ) x  <-- Here
```

Yet another application.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1)
      I, f : 'a, x : 'b |- f : 'a -| {}
      I, f : 'a, x : 'b |- ( + ) x 1
        I, f : 'a, x : 'b |- ( + ) x
          I, f : 'a, x : 'b |- ( + ) : int -> int -> int -| {}  <-- Here
```

That one was easy, because we just had to look up the name `( + )` in the environment. The next is also easy, because we just look up `x`.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1)
      I, f : 'a, x : 'b |- f : 'a -| {}
      I, f : 'a, x : 'b |- ( + ) x 1
```

(continues on next page)

(continued from previous page)

```
        I, f : 'a, x : 'b |- ( + ) x
          I, f : 'a, x : 'b |- ( + ) : int -> int -> int -| {}
          I, f : 'a, x : 'b |- x : 'b -| {}  <-- Here
```

At last, we're ready to resolve a function application! We introduce a fresh type variable and add a constraint. The constraint is that the inferred type `int -> int -> int` of the left-hand subexpression must equal the inferred type `'b` of the right-hand subexpression arrow the fresh type variable `'c`, that is, `'b -> 'c`.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1)
      I, f : 'a, x : 'b |- f : 'a -| {}
      I, f : 'a, x : 'b |- ( + ) x 1
        I, f : 'a, x : 'b |- ( + ) x : 'c -| int -> int -> int = 'b -> 'c  <-- Here
          I, f : 'a, x : 'b |- ( + ) : int -> int -> int -| {}
          I, f : 'a, x : 'b |- x : 'b -| {}
```

Now we're ready for the argument being passed to that function.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1)
      I, f : 'a, x : 'b |- f : 'a -| {}
      I, f : 'a, x : 'b |- ( + ) x 1
        I, f : 'a, x : 'b |- ( + ) x : 'c -| int -> int -> int = 'b -> 'c
          I, f : 'a, x : 'b |- ( + ) : int -> int -> int -| {}
          I, f : 'a, x : 'b |- x : 'b -| {}
        I, f : 'a, x : 'b |- 1 : int -| {}  <-- Here
```

Again we can resolve a function application with a new type variable and constraint.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1)
      I, f : 'a, x : 'b |- f : 'a -| {}
      I, f : 'a, x : 'b |- ( + ) x 1 : 'd -| 'c = int -> 'd, int -> int -> int = 'b ->
↪ 'c  <-- Here
        I, f : 'a, x : 'b |- ( + ) x : 'c -| int -> int -> int = 'b -> 'c
          I, f : 'a, x : 'b |- ( + ) : int -> int -> int -| {}
          I, f : 'a, x : 'b |- x : 'b -| {}
        I, f : 'a, x : 'b |- 1 : int -| {}
```

And once more, a function application, so a new type variable and a new constraint.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1)
    I, f : 'a, x : 'b |- f (( + ) x 1) : 'e -| 'a = 'd -> 'e, 'c = int -> 'd, int ->␣
↪int -> int = 'b -> 'c   <-- Here
      I, f : 'a, x : 'b |- f : 'a -| {}
      I, f : 'a, x : 'b |- ( + ) x 1 : 'd -| 'c = int -> 'd, int -> int -> int = 'b ->
↪ 'c
        I, f : 'a, x : 'b |- ( + ) x : 'c -| int -> int -> int = 'b -> 'c
          I, f : 'a, x : 'b |- ( + ) : int -> int -> int -| {}
          I, f : 'a, x : 'b |- x : 'b -| {}
        I, f : 'a, x : 'b |- 1 : int -| {}
```

Now we finally get to finish off an anonymous function. Its inferred type is the fresh type variable `'b` of its parameter `x`, arrow the inferred type `e` of its body.

```
I |- fun f -> fun x -> f (( + ) x 1)
  I, f : 'a |- fun x -> f (( + ) x 1) : 'b -> 'e -| 'a = 'd -> 'e, 'c = int -> 'd,␣
↪int -> int -> int = 'b -> 'c   <-- Here
    I, f : 'a, x : 'b |- f (( + ) x 1) : 'e -| 'a = 'd -> 'e, 'c = int -> 'd, int ->␣
↪int -> int = 'b -> 'c
      I, f : 'a, x : 'b |- f : 'a -| {}
      I, f : 'a, x : 'b |- ( + ) x 1 : 'd -| 'c = int -> 'd, int -> int -> int = 'b ->
↪ 'c
        I, f : 'a, x : 'b |- ( + ) x : 'c -| int -> int -> int = 'b -> 'c
          I, f : 'a, x : 'b |- ( + ) : int -> int -> int -| {}
          I, f : 'a, x : 'b |- x : 'b -| {}
        I, f : 'a, x : 'b |- 1 : int -| {}
```

And the last anonymous function can now be complete in the same way:

```
I |- fun f -> fun x -> f (( + ) x 1) : 'a -> 'b -> 'e -| 'a = 'd -> 'e, 'c = int ->
↪'d, int -> int -> int = 'b -> 'c  <-- Here
  I, f : 'a |- fun x -> f (( + ) x 1) : 'b -> 'e -| 'a = 'd -> 'e, 'c = int -> 'd,␣
↪int -> int -> int = 'b -> 'c
    I, f : 'a, x : 'b |- f (( + ) x 1) : 'e -| 'a = 'd -> 'e, 'c = int -> 'd, int ->␣
↪int -> int = 'b -> 'c
       I, f : 'a, x : 'b |- f : 'a -| {}
       I, f : 'a, x : 'b |- ( + ) x 1 : 'd -| 'c = int -> 'd, int -> int -> int = 'b -
↪> 'c
         I, f : 'a, x : 'b |- ( + ) x : 'c -| int -> int -> int = 'b -> 'c
           I, f : 'a, x : 'b |- ( + ) : int -> int -> int -| {}
           I, f : 'a, x : 'b |- x : 'b -| {}
         I, f : 'a, x : 'b |- 1 : int -| {}
```

As a result of constraint generation, we know that the type of the expression is `'a -> 'b -> 'e`, where

```
'a = 'd -> 'e
'c = int -> 'd
int -> int -> int = 'b -> 'c
```

To solve that system of equations, we use the unification algorithm:

```
unify('a = 'd -> 'e, 'c = int -> 'd, int -> int -> int = 'b -> 'c)
```

The first constraint yields a substitution `{('d -> 'e) / 'a}`, which we record as part of the solution, and also apply it to the remaining constraints:

```
...
=
{('d -> 'e) / 'a}; unify(('c = int -> 'd, int -> int -> int = 'b -> 'c) {('d -> 'e) /
↪'a}
=
{('d -> 'e) / 'a}; unify('c = int -> 'd, int -> int -> int = 'b -> 'c)
```

The second constraint behaves similarly to the first:

```
...
=
```

(continues on next page)

(continued from previous page)

```
{('d -> 'e) / 'a}; {(int -> 'd) / 'c}; unify((int -> int -> int = 'b -> 'c) {(int ->
 ↪'d) / 'c})
=
{('d -> 'e) / 'a}; {(int -> 'd) / 'c}; unify(int -> int -> int = 'b -> int -> 'd)
```

The function constraint breaks down into two smaller constraints:

```
...
=
{('d -> 'e) / 'a}; {(int -> 'd) / 'c}; unify(int = 'b, int -> int = int -> 'd)
```

We get another substitution:

```
...
=
{('d -> 'e) / 'a}; {(int -> 'd) / 'c}; {int / 'b}; unify((int -> int = int -> 'd)
 ↪{int / 'b})
=
{('d -> 'e) / 'a}; {(int -> 'd) / 'c}; {int / 'b}; unify(int -> int = int -> 'd)
```

Then we get to break down another function constraint:

```
...
=
{('d -> 'e) / 'a}; {(int -> 'd) / 'c}; {int / 'b}; unify(int = int, int = 'd)
```

The first of the resulting new constraints is trivial and just gets dropped:

```
...
=
{('d -> 'e) / 'a}; {(int -> 'd) / 'c}; {int / 'b}; unify(int = 'd)
```

The very last constraint gives us one more substitution:

```
=
{('d -> 'e) / 'a}; {(int -> 'd) / 'c}; {int / 'b}; {int / 'd}
```

To finish, we apply the substitution output by unification to the type inferred by constraint generation:

```
('a -> 'b -> 'e) {('d -> 'e) / 'a}; {(int -> 'd) / 'c}; {int / 'b}; {int / 'd}
=
(('d -> 'e) -> 'b -> 'e) {(int -> 'd) / 'c}; {int / 'b}; {int / 'd}
=
(('d -> 'e) -> 'b -> 'e) {int / 'b}; {int / 'd}
=
(('d -> 'e) -> int -> 'e) {int / 'd}
=
(int -> 'e) -> int -> 'e
```

And indeed that is the same type that OCaml would infer for the original expression:

```
# fun f -> fun x -> f (( + ) x 1);;
- : (int -> 'a) -> int -> 'a = <fun>
```

Except that OCaml uses a different type variable identifier. OCaml is nice to us and "lowers" the type variables down to smaller letters of the alphabet. We could do that too with a little extra work.

Type Errors. In reality there is yet another piece to type inference. If unification fails, the compiler or interpreter needs to produce a helpful error message. That's an important engineering challenge that we won't address here. It requires keeping track of more than just constraints: we need to know why a constraint was introduced, and the ramification of its violation. We also need to track the constraint back to the lexical piece of code that produced it, so that programmers can see where the problem occurs. And since it's possible that constraints can be processed in many different orders, there are many possible error messages that could be produced. Figuring out which one will lead the programmer to the root cause of an error, instead of some downstream consequence of it, is an area of ongoing research.

11.6.5 Let Polymorphism

Now we'll add `let` expressions to our little language:

```
e ::= x | i | b | e1 bop e2
    | if e1 then e2 else e3
    | fun x -> e
    | e1 e2
    | let x = e1 in e2    (* new *)
```

It turns out type inference for them is considerably trickier than might be expected. The naive approach would be to add this constraint generation rule:

```
env |- let x = e1 in e2 : t2 -| C1, C2
  if env |- e1 : t1 -| C1
  and env, x : t1 |- e2 : t2 -| C2
```

From the type-checking perspective, that's the same rule we've always used. And for many `let` expressions it works perfectly fine. For example:

```
{} |- let x = 42 in x : int -| {}
  {} |- 42 : int -| {}
  x : int |- x : int -| {}
```

The problem is that when the value being bound is a polymorphic function, that rule generates constraints that are too restrictive. For example, consider the identity function:

```
let id = fun x -> x in
let a = id 0 in
id true
```

OCaml has no trouble inferring the type of `id` as `'a -> 'a` and permitting it to be applied both to an `int` and a `bool`. But the rule above isn't so permissive about application to both types. When we use it, we generate the following types and constraints:

```
{} |- let id = fun x -> x in (let a = id 0 in id true) : 'c -| 'a -> 'a = int -> 'b,
 ↪'a -> 'a = bool -> 'c
  {} |- fun x -> x : 'a -| {}
    x : 'a |- x : 'a -| {}
  id : 'a -> 'a |- let a = id 0 in id true : 'c -| 'a -> 'a = int -> 'b, 'a -> 'a =␣
 ↪bool -> 'c   <--- POINT 1
    id : 'a -> 'a |- id 0 : 'b -| 'a -> 'a = int -> 'b
      id : 'a -> 'a |- id : 'a -> 'a -| {}
```

(continues on next page)

(continued from previous page)

```
    id : 'a -> 'a |- 0 : int -| {}
  id : 'a -> 'a, a : 'b |- id true : 'c -| 'a -> 'a = bool -> 'c   <--- POINT 2
    id : 'a -> 'a, a : 'b |- id : 'a -> 'a -| {}
    id : 'a -> 'a, a : 'b |- true : bool -| {}
```

Notice that we do infer a type `'a -> 'a` for `id`, which you can see in the environment in later lines of the example. But, at Point 1, we infer a constraint `'a -> 'a = int -> 'b`, and at Point 2, we infer `'a -> 'a = bool -> 'c`. When the unification algorithm encounters those constraints, it will break them down into `'a = int`, `'a = 'b`, `'a = bool`, and `'a = 'c`. The first and third of those are contradictory, because we can't have `'a = int` and `'a = bool`. One or the other will be substituted away during unification, leaving an unsatisfiable constraint `int = bool`. At that point unification will fail, declaring the program to be ill typed.

The problem is that the `'a` type variable in the inferred type of `id` stands for an unknown but **fixed** type. At each application of `id`, we want to let `'a` become a **different** type, instead of forcing it to always be the same type.

The solution to the problem of polymorphism for `let` expressions is not simple. It requires us to introduce a new kind of type: a *type scheme*. Type schemes resemble *universal quantification* from mathematical logic. For example, in logic you might write, "for all natural numbers x, it holds that $0 \cdot x = 0$". The "for all" is the universal quantification: it abstracts away from a particular x and states a property that is true of all natural numbers.

A type scheme is written `'a . t`, where `'a` is a type variable and `t` is a type in which `'a` may appear. For example, `'a . 'a -> 'a` is a type scheme. It is the type of a function that takes in a value of type `'a` and returns a value of type `'a`, for all `'a`. Thus, it is the type of the polymorphic identity function.

We can also have many type variables to the left of the dot in a type scheme. For example, `'a 'b . 'a -> 'b -> 'a` is the type of a function that takes in two arguments and returns the first. In OCaml, we could write that as `fun x y -> x`. Note that utop infers the type of it as we would expect:

```
# let f = fun x y -> x;;
val f : 'a -> 'b -> 'a = <fun>
```

But we could actually manually write down an annotation with a type scheme:

```
# let f : 'a 'b . 'a -> 'b -> 'a = fun x y -> x;;
val f : 'a -> 'b -> 'a = <fun>
```

Note that OCaml accepts our manual type annotation but doesn't include the `'a 'b .` part of it in its output. **But it's implicitly there and always has been.** In general, anytime OCaml has inferred a type `t` and that type has had type variables in it, in reality it's a type scheme. For example, the type of `List.length` is really a type scheme:

```
# let mylen : 'a . 'a list -> int = List.length;;
val mylen : 'a list -> int = <fun>
```

OCaml just doesn't bother outputting the list of type variables that are to the left of the dot in the type scheme. Really they'd just clutter the output, and many programmers never need to know about them. But now that you're learning type inference, it's time for you to know.

Now that we have type schemes, we'll have static environments map names to type schemes. We can think of types as being special cases of type schemes in which the list of type variables is empty. With type schemes, the `let` rule changes in only one way from the naive rule above, which is the `generalize` on the last line:

```
env |- let x = e1 in e2 : t2 -| C1, C2
  if env |- e1 : t1 -| C1
  and generalize(C1, env, x : t1) |- e2 : t2 -| C2
```

The job of `generalize` is to take a type like `'a -> 'a` and *generalize* it into a type scheme like `'a . 'a -> 'a` in an environment `env` against constraints `C1`. Let's come back to how it works in a minute. Before that, there's one other rule that needs to change, which is the name rule:

```
env |- n : instantiate(env(n)) -| {}
```

The only thing that changes there is that use of `instantiate`. Its job is to take a type scheme like `'a . 'a -> 'a` and *instantiate* it into a new type (and here we strictly mean a type, not a type scheme) with fresh type variables. For example, `'a . 'a -> 'a` could be instantiated as `'b -> 'b`, if `'b` isn't yet in use anywhere else as a type variable.

Here's how those two revised rules work together to get our earlier example with the identify function right:

```
{} |- let id = fun x -> x in (let a = id 0 in id true)
  {} |- fun x -> x : 'a -> 'a -| {}
    x : 'a |- x : 'a -| {}
  id : 'a . 'a -> 'a |- let a = id 0 in id true   <--- POINT 1
```

Let's pause there at Point 1. When `id` is put into the environment by the `let` rule, its type is generalized from `'a -> 'a` to `'a . 'a -> 'a`; that is, from a type to a type scheme. That records the fact that each application of `id` should get to use its own value for `'a`. Going on:

```
{} |- let id = fun x -> x in (let a = id 0 in id true)
  {} |- fun x -> x : 'a -> 'a -| {}
    x : 'a |- x : 'a -| {}
  id : 'a . 'a -> 'a |- let a = id 0 in id true   <--- POINT 1
    id : 'a . 'a -> 'a |- id 0
      id : 'a . 'a -> 'a |- id : 'b -> 'b -| {}   <--- POINT 3
```

Pausing here at Point 3, when `id` is applied to `0`, we instantiate its type variable `'a` with a fresh type variable `'b`. Let's finish:

```
{} |- let id = fun x -> x in (let a = id 0 in id true) : 'e -| 'b -> 'b = int -> 'c,
 ↪'d -> 'd = bool -> 'e
  {} |- fun x -> x : 'a -> 'a -| {}
    x : 'a |- x : 'a -| {}
  id : 'a . 'a -> 'a |- let a = id 0 in id true : 'e -| 'b -> 'b = int -> 'c, 'd ->
 ↪'d = bool -> 'e   <--- POINT 1
    id : 'a . 'a -> 'a |- id 0 : 'c -| 'b -> 'b = int -> 'c
      id : 'a . 'a -> 'a |- id : 'b -> 'b -| {}   <--- POINT 3
      id : 'a . 'a -> 'a |- 0 : int -| {}
    id : 'a . 'a -> 'a, a : 'b |- id true : 'e -| 'd -> 'd = bool -> 'e   <--- POINT 2
      id : 'a . 'a -> 'a, a : 'b |- id : 'd -> 'd -| {}   <--- POINT 4
      id : 'a . 'a -> 'a, a : 'b |- true : bool -| {}
```

At Point 4, when `id` is applied to `true`, we again instantiate its type variable `'a` with a fresh type variable, this time `'d`. So the constraints collected at Points 1 and 2 are no longer contradictory, because they are talking about different type variables. Those constraints are:

```
'b -> 'b = int -> 'c
'd -> 'd = bool -> 'e
```

The unification algorithm will therefore conclude:

```
'b = int
'c = int
```

(continues on next page)

(continued from previous page)

```
'd = bool
'e = bool
```

So the entire expression is successfully inferred to have type `bool`.

Instantiation and Generalization. We used two new functions, `instantiate` and `generalize`, to define type inference for `let` expressions. We need to define those functions.

The easy one is `instantiate`. Given a type scheme `'a1 'a2 ... 'an . t`, we instantiate it by:

- choosing `n` fresh type variables, and
- substituting each of those for `'a1` through `'an` in `t`.

Substitution is uncomplicated here, compared to how it was for evaluation in thee substitution model, because there is nothing in a type that can bind variable names.

But `generalize` requires more work. Here's the `let` rule again:

```
env |- let x = e1 in e2 : t2 -| C1, C2
  if env |- e1 : t1 -| C1
  and generalize (C1, env, x : t1) |- e2 : t2 -| C2
```

To generalize `t1`, we do the following.

First, we pretend like `e1` is all that matters, and that the rest of the `let` expression doesn't exist. If `e1` were the entire program, how would we finish type inference? We'd run the unification algorithm on `C1`, get a substitution `S`, and return `t1 S` as the inferred type of `e1`. So, do that now. Let's call that inferred type `u1`. Let's also apply `S` to `env` to get a new environment `env1`, which now reflects all the type information we've gleaned from `e1`.

Second, we figure out which type variables in `u1` should be generalized. Why not all of them? Because some type variables could have been introduced by code that surrounds the `let` expression, e.g.,

```
fun x ->
  (let y = e1 in e2) (let z = e3 in e4)
```

The type variable for `x` should not be generalized in inferring the type of either `y` or `z`, because `x` has to have the same type in all four subexpressions, `e1` through `e4`. Generalizing could mistakenly allow `x` to have one type in `e1` and `e2`, but a different type in `e3` and `e4`.

So instead we generalize only variables that **are** in `u1` but are **not** in `env1`. That way we generalize only the type variables from `e1`, not variables that were already in the environment when we started inferring the `let` expression's type. Suppose those variables are `'a1 ... 'an`. The type scheme we give to `x` is then `'a1 ... 'an . u1`.

Putting all that together, we end up with:

```
generalize(C1, env, x : t1) =
  env1, x : 'a1 ... 'an . u1
```

Returning to our example with the identify function from above, we had `generalize({}, {}, x : 'a -> 'a)`. In that rather simple case, `unify` discovers no new equalities from the environment, so `u1 = 'a -> 'a` and `env1 = {}`. The only type variable in `u1` is `'a`, and it doesn't appear in `env1`. So `'a` is generalized, yielding `'a . 'a -> 'a` as the type scheme for `id`.

11.6.6 Polymorphism and Mutability

There is yet one more complication to type inference for `let` expressions. It appears when we add mutable references to the language. Consider this example code, which does not type check in OCaml:

```
let succ = fun x -> ( + ) 1 x;;
let id = fun x -> x;;
let r = ref id;;
r := succ;;
!r true;;  (* error *)
```

It's clear we should infer `succ : int -> int` and `id : 'a . 'a -> 'a`. But what should the type of `r` be? It's tempting to say we should infer `r : 'a . ('a -> 'a) ref`. That would let us instantiate the type of `r` to be `(int -> int) ref` on line 4 and store `succ` in `r`. But it also would let us instantiate the type of `r` to be `(bool -> bool) ref` on line 5. That's a disaster: it causes the application of `succ` to `true`, which is not type safe.

The solution adopted by OCaml and related languages is called the *value restriction:* the type system is designed to prevent a polymorphic mutable value from ever holding more than one type. Let's redo some of that example again, pausing to look at the toplevel output:

```
# let id = fun x -> x;;
val id : 'a -> 'a = <fun>   (* as expected *)

# let r = ref id;;
val r : ('_weak1 -> '_weak1) ref = { ... }   (* what is _weak? *)

# r;;
- : ('_weak1 -> '_weak1) ref = { ... }   (* it's consistent at least *)

# r := succ;;
- : unit = ()

# r;;
- : (int -> int) ref = { ... }   (* did r just change type ?! *)
```

When the type of `r` is inferred, OCaml gives it a type involving a *weak* type variable. All such variables have a name starting with `'_weak`. A weak type variable is one that has not been generalized hence cannot be instantiated on multiple types. Rather, it indicates a single type that is not yet known. Think of it as type inference for that variable is not yet finished: OCaml is waiting for more information to pin down precisely what it is. When `r := succ` is executed, that information finally becomes available. OCaml infers that `'_weak1 = int` from the type of `succ`. Then OCaml replaces `'_weak1` with `int` everywhere. That's what yields an error on the final line:

```
# !r true;;
Error: This expression has type bool but an expression was expected of type int
```

Since `r : (int -> int) ref`, we cannot apply `!r` to a `bool`.

We won't cover the implementation of weak type variables here.

But, let's not leave this topic of the interaction between polymorphic types and mutability yet. You might be tempted to think that it's a phenomenon that affects only OCaml. But indeed, even Java suffers.

Consider the following class hierarchy:

```
class Animal { }
class Elephant extends Animal { }
class Rabbit extends Animal { }
```

Now suppose we create an array of animals:

```
Animal[] a= new Rabbit[2]
```

Here we are using *subtype polymorphism* to assign an array of `Rabbit` objects to an `Animal[]` reference. That's not the same as *parametric polymorphism* as we've been using in OCaml, but it's nonetheless polymorphism.

What if we try this?

```
a[0]= new Elephant()
```

Since `a` is typed as an `Animal` array, it stands to reason that we could assign an elephant object into it, just as we could assign a rabbit object. And indeed that code is fine according to the Java compiler. But Java gives us a runtime error if we run that code!

```
Exception java.lang.ArrayStoreException
```

The problem is that mutating the first array element to be a rabbit would leave us with a `Rabbit` array in which one element is a `Elephant`. (Ouch! An elephant would sit on a rabbit. Poor bun bun.) But in Java, the type of every object of an array is supposed to be a property of the array as a whole. Every element of the array created by `new Rabbit[2]` therefore must be a `Rabbit`. So Java prevents the assignment above by detecting the error at run time and raising an exception.

This is really the value restriction in another guise! The type of a value stored in a mutable location may not change, according to the value restriction. With arrays, Java implements that with a run-time check, instead of rejecting the program at compile time. This strikes a balance between soundness (preventing errors from happening) and expressivity (allowing more error-free programs to type check).

11.7 Summary

At first it might seem mysterious how a programming language could be implemented. But, after this chapter, hopefully some of that mystery has been revealed. Implementation of a programming language is just a matter of the same studious application of syntax, dynamic semantics, and static semantics that we've studied throughout this book. It also relies heavily on CS theory of the kind studied in discrete mathematics or theory of computation courses.

11.7.1 Terms and Concepts

- abstract syntax
- abstract syntax tree
- associativity
- back end
- Backus-Naur Form (BNF)
- big step
- bytecode
- call by name
- call by value
- capture-avoiding substitution

- closure
- compiler
- concrete syntax
- constraint
- context-free grammar
- context-free language
- desugaring
- dynamic environment
- dynamic scope
- environment model
- evaluation
- fresh
- front end
- generalization
- Hindley–Milner (HM) type inference algorithm
- implicit typing
- instantiation
- intermediate representation
- interpreter
- lambda calculus
- let polymorphism
- lexer
- machine configuration
- metavariable
- nonterminal
- operational semantics
- optimizing compiler
- parser
- precedence
- preliminary type variable
- preservation
- primitive operatiohn
- progress
- pushdown automata
- regular expression
- regular language

- relation
- semantic analysis
- short circuit
- small step
- source program
- static scope
- static typing
- stuck
- substitution
- substitution model
- symbol
- symbol table
- target program
- terminal
- token
- type annotation
- type checking
- type inference
- type reconstruction
- type safety
- type scheme
- type system
- type variable
- typing context
- unification
- unifier
- value
- value restriction
- virtual machine
- weak type variable
- well typed

11.7.2 Further Reading

- *Types and Programming Languages* by Benjamin C. Pierce, chapters 1-14, 22.
- *Modern Compiler Implementation* (in Java or ML) by Andrew W. Appel, chapters 1-5.
- *Automata and Computability* by Dexter C. Kozen, chapters 1-27.
- *Real World OCaml* has a chapter on the OCaml frontend.
- This webpage documents how some of the internals of the OCaml type checker and inferencer.
- The OCaml VM aka the Zinc Machine is described in these papers: 1, 2.

11.7.3 Acknowledgment

Our treatment of type inference is based on Pierce.

11.8 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Many of these exercises rely on the SimPL interpreter as starter code. You can download it here: simpl.zip.

Exercise: parse [★]

Run `make utop` in the SimPL interpreter implementation. It will compile the interpreter and launch utop. Evaluate the following expressions. Note what each returns.

- `parse "22"`
- `parse "1 + 2 + 3"`
- `parse "let x = 2 in 20 + x"`

Also evaluate these expressions, which will raise exceptions. Explain why each one is an error, and whether the error occurs during parsing or lexing.

- `parse "3.14"`
- `parse "3+"`

Exercise: simpl ids [★★]

Examine the definition of the `id` regular expression in the SimPL lexer. Identify at least one way in which it differs from the definition of OCaml identifiers.

Exercise: times parsing [★★]

In the SimPL parser, the `TIMES` token is declared as having higher precedence than `PLUS`, and as being left associative. Let's experiment with other choices.

- Evaluate `parse "1*2*3"`. Note the AST. Now change the declaration of the associativity of `TIMES` in `parser.mly` to be `%right` instead of `%left`. Recompile and reevaluate `parse "1*2*3"`. How did the AST change? Before moving on, restore the declaration to be `%left`.

- Evaluate `parse "1+2*3"`. Note the AST. Now swap the declaration `%left TIMES` in `parser.mly` with the declaration `%left PLUS`. Recompile and reevaluate `parse "1+2*3"`. How did the AST change? Before moving on, restore the original declaration order.

Exercise: infer [★★]

Type inference for SimPL can be done in a much simpler way than for the larger language (with anonymous functions and let expression) that we considered in the section on type inference.

Run `make` in the SimPL interpreter implementation. It will compile the interpreter and launch utop. Now, define a function `infer : string -> typ` such that `infer s` parses `s` into an expression and infers the type of `s` in the empty context. Your solution will make use of the `typeof` function. You don't need constraint collection or unification.

Try out your `infer` function on these test cases:

- `"3110"`
- `"1 <= 2"`
- `"let x = 2 in 20 + x"`

Exercise: subexpression types [★★]

Suppose that a SimPL expression is well typed in a context `ctx`. Are all of its subexpressions also well typed in `ctx`? For every subexpression, does there exist some context in which the subexpression is well typed? Why or why not?

Exercise: typing [★★]

Use the SimPL type system to show that `{} |- let x = 0 in if x <= 1 then 22 else 42 : int`.

Exercise: substitution [★★]

What is the result of the following substitutions?

- `(x + 1){2/x}`
- `(x + y){2/x}{3/y}`
- `(x + y){1/z}`
- `(let x = 1 in x + 1){2/x}`
- `(x + (let x=1 in x+1)){2/x}`
- `((let x=1 in x+1) + x){2/x}`
- `(let x=y in x+1){2/y}`
- `(let x=x in x+1){2/x}`

Exercise: step expressions [★]

Here is an example of evaluating an expression:

```
  7+5*2
-->  (step * operation)
  7+10
-->  (step + operation)
  17
```

There are two steps in that example, and we've annotated each step with a parenthetical comment to hint at which evaluation rule we've used. We stopped evaluating when we reached a value.

Evaluate the following expressions using the small-step substitution model. Use the "long form" of evaluation that we demonstrated above, in which you provide a hint as to which rule is applied at each step.

- `(3 + 5) * 2` (2 steps)
- `if 2 + 3 <= 4 then 1 + 1 else 2 + 2` (4 steps)

Exercise: step let expressions [★★]

Evaluate these expressions, again using the "long form" from the previous exercise.

- `let x = 2 + 2 in x + x` (3 steps)
- `let x = 5 in ((let x = 6 in x) + x)` (3 steps)
- `let x = 1 in (let x = x + x in x + x)` (4 steps)

Exercise: variants [★]

Evaluate these Core OCaml expressions using the small-step substitution model:

- `Left (1+2)` (1 step)
- `match Left 42 with Left x -> x+1 | Right y -> y-1` (2 steps)

Exercise: application [★★]

Evaluate these Core OCaml expressions using the small-step substitution model:

- `(fun x -> 3 + x) 2` (2 steps)
- `let f = (fun x -> x + x) in (f 3) + (f 3)` (6 steps)
- `let f = fun x -> x + x in let x = 1 in let g = fun y -> x + f y in g 3` (7 steps)
- `let f = (fun x -> fun y -> x + y) in let g = f 3 in (g 1) + (f 2 3)` (9 steps)

Exercise: omega [★★★]

Try evaluating `(fun x -> x x) (fun x -> x x)`. This expression, which is usually called Ω, doesn't type check in real OCaml, but we can still use the Core OCaml small-step semantics on it.

Exercise: pair parsing [★★★]

Add pairs (i.e., tuples with exactly two components) to SimPL. Implement lexing and parsing of pairs. Assume that the parentheses around the pair are required (not optional, as they sometimes are in OCaml). Follow this strategy:

- Add a constructor for pairs to the `expr` type.
- Add a comma token to the parser.
- Implement lexing the comma token.
- Implement parsing of pairs.

When you compile, you will get some inexhaustive pattern match warnings, because you have not yet implemented type checking nor interpretation of pairs. But you can still try parsing them in utop with the `parse` function.

Exercise: pair type checking [★★★]

Implement type checking of pairs. Follow this strategy:

- Write down a new typing rule before implementing any code.
- Add a new constructor for pairs to the `typ` type.
- Add a new branch to `typeof`.

Exercise: pair evaluation [★★★]

Implement evaluation of pairs. Follow this strategy:

- Implement `is_value` for pairs. A pair of values (e.g., `(0,1)`) is itself a value, so the function will need to become recursive.
- Implement `subst` for pairs: `(e1, e2){v/x} = (e1{v/x}, e2{v/x})`.
- Implement small-step and big-step evaluation of pairs, using these rules:

```
(e1, e2) --> (e1', e2)
  if e1 --> e1'

(v1, e2) --> (v1, e2')
  if e2 --> e2'

(e1, e2) ==> (v1, v2)
  if e1 ==> v1
  and e2 ==> v2
```

Exercise: desugar list [★]

Suppose we treat list expressions like syntactic sugar in the following way:

- `[]` is syntactic sugar for `Left 0`.
- `e1 :: e2` is syntactic sugar for `Right (e1, e2)`.

What is the core OCaml expression to which `[1; 2; 3]` desugars?

Exercise: list not empty [★★]

Write a core OCaml function `not_empty` that returns `1` if a list is non-empty and `0` if the list is empty. Use the substitution model to check that your function behaves properly on these test cases:

- `not_empty []`

- `not_empty [1]`

Exercise: list not empty [★★★★]

In core OCaml, there are only two patterns: `Left x` and `Right x`, where `x` is a variable name. But in full OCaml, patterns are far more general. Let's see how far we can generalize patterns in core OCaml.

Step 1: Here is a BNF grammar for patterns, and slightly revised BNF grammar for expressions:

```
p ::= i | (p1, p2) | Left p | Right p | x | _

e ::= ...
    | match e with | p1 -> e1 | p2 -> e2 | ... | pn -> en
```

In the revised syntax for `match`, only the very first `|` on the line, immediately before the keyword `match`, is meta-syntax. The remaining four `|` on the line are syntax. Note that we require `|` before the first pattern.

Step 2: A value `v` matches a pattern `p` if by substituting any variables or wildcards in `p` with values, we can obtain exactly `v`. For example:

- `2` matches `x` because `x{2/x}` is `2`.
- `Right(0,Left 0)` matches `Right(x,_)` because `Right(x,_){0/x}{Left 0/_}` is `Right(0, Left 0)`.

Let's define a new ternary relation called `matches`, guided by those examples:

```
v =~ p // s
```

Pronounce this relation as "`v` matches `p` producing substitutions `s`."

Here, `s` is a sequence of substitutions, such as `{0/x}{Left 3/y}{(1,2)/z}`. There is just a single rule for this relation:

```
v =~ p // s
  if v = p s
```

For example,

```
2 =~ x // {2/x}
  because 2 = x{2/x}
```

Step 3: To evaluate a match expression:

- Evaluate the expression being matched to a value.
- If that expression matches the first pattern, evaluate the expression corresponding to that pattern.
- Otherwise, match against the second pattern, the third, etc.
- If none of the patterns matches, evaluation is *stuck*: it cannot take any more steps.

Using those insights, complete the following evaluation rules by filling in the places marked with `???`:

```
(* This rule should implement evaluation of e. *)
match e with | p1 -> e1 | p2 -> e2 | ... | pn -> en
--> ???
  if ???
```

(continues on next page)

(continued from previous page)

```
(* This rule implements moving past p1 to the next pattern. *)
match v with | p1 -> e1 | p2 -> e2 | ... | pn -> en
--> match v with | p2 -> e2 | ... | pn -> en
  if there does not exist an s such that ???

(* This rule implements matching v with p1 then proceeding to evaluate e1. *)
match v with | p1 -> e1 | p2 -> e2 | ... | pn -> en
--> ??? (* something involving e1 *)
  if ???
```

Note that we don't need to write the following rule explicitly:

```
match v with |  -/->
```

Evaluation will get stuck at that point because none of the three other rules above will apply.

Step 4: Double check your rules by evaluating the following expression:

```
match (1 + 2, 3) with | (1,0) -> 4 | (1,x) -> x | (x,y) -> x + y
```

Exercise: let rec [★★★★]

One of the evaluation rules for `let` is

```
let x = v in e --> e{v/x}
```

We could try adapting that to `let rec`:

```
let rec x = v in e --> e{v/x}   (* broken *)
```

But that rule doesn't work properly, as we see in the following example:

```
  let rec fact = fun x ->
        if x <= 1 then 1 else x * (fact (x - 1)) in
  fact 3

-->

  (fun x -> if x <= 1 then 1 else x * (fact (x - 1)) 3

-->

  if 3 <= 1 then 1 else 3 * (fact (3 - 1))

-->

  3 * (fact (3 - 1))

-->

  3 * (fact 2)

-/->
```

We're now stuck, because we need to evaluate `fact`, but it doesn't step. In essence, the semantic rule we used "forgot" the function value that should have been associated with `fact`.

A good way to fix this problem is to introduce a new language construct for recursion called simply `rec`. (Note that OCaml does not have any construct that corresponds directly to `rec`.) Formally, we extend the syntax for expressions as follows:

```
e ::= ...
    | rec f -> e
```

and add the following evaluation rule:

```
rec f -> e  -->  e{(rec f -> e)/f}
```

The intuitive reading of this rule is that when evaluating `rec f -> e`, we "unfold" `f` in the body of `e`. For example, here is an infinite loop coded with `rec`:

```
  rec f -> f

-->  (* step rec *)

  f{(rec f -> f)/f}

= (* substitute *)

  rec f -> f

--> (* step rec *)

  f{(rec f -> f)/f}

...
```

Now we can use `rec` to implement `let rec`. Anywhere `let rec` appears in a program:

```
let rec f = e1 in e2
```

we *desugar* (i.e., rewrite) it to

```
let f = rec f -> e1 in e2
```

Note that the second occurrence of `f` (inside the `rec`) shadows the first one. Going back to the `fact` example, its desugared version is

```
let fact = rec fact -> fun x ->
  if x <= 1 then 1 else x * (fact (x - 1)) in
fact 3
```

Evaluate the following expression (17 steps, we think, though it does get pretty tedious). You may want to simplify your life by writing "F" in place of `(rec fact -> fun x -> if x <= 1 then 1 else x * (fact (x-1)))`

```
let rec fact = fun x ->
  if x <= 1 then 1 else x * (fact (x - 1)) in
fact 3
```

Exercise: simple expressions [★]

In the small-step substitution model, evaluation of an expression was rather *list-like*: we could write an evaluation in a linear form like `e --> e1 --> e2 --> ... --> en --> v`. In the big-step environment model, evaluation is instead rather *tree-like*: evaluations have a nested, recursive structure. Here's an example:

```
<{}, (3 + 5) * 2> ==> 16          (op rule)
    because <{}, (3 + 5)> ==> 8   (op rule)
        because <{},3> ==> 3      (int const rule)
        and     <{},5> ==> 5      (int const rule)
        and 3+5 is 8
    and <{}, 2> ==> 2             (int const rule)
    and 8*2 is 16
```

We've used indentation here to show the shape of the tree, and we've labeled each usage of one of the semantic rules.

Evaluate the following expressions using the big-step environment model. Use the notation for evaluation that we demonstrated above, in which you provide a hint as to which rule is applied at each node in the tree.

- `110 + 3*1000` *hint: three uses of the constant rule, two uses of the op rule*
- `if 2 + 3 < 4 then 1 + 1 else 2 + 2` *hint: five uses of constant, three uses of op, one use of if(else)*

Exercise: let and match expressions [★★]

Evaluate these expressions, continuing to use the tree notation, and continuing to label each usage of a rule.

- `let x=0 in 1` *hint: one use of let, two uses of constant*
- `let x=2 in x+1` *hint: one use of let, two uses of constant, one use of op, one use of variable*
- `match Left 2 with Left x -> x+1 | Right x -> x-1` *hint: one use of match(left), two uses of constant, one use of op, one use of variable*

Exercise: closures [★★]

Evaluate these expressions:

- `(fun x -> x+1) 2` *hint: one use of application, one use of anonymous function, two uses of constant, one use of op, one use of variable*
- `let f = fun x -> x+1 in f 2` *hint: one use of let, one use of anonymous function, one use of application, two uses of variable, one use of op, two uses of constant*

Exercise: lexical scope and shadowing [★★]

Evaluate these expressions:

- `let x=0 in x + (let x=1 in x)` *hint: two uses of let, two uses of variable, one use of op, two uses of constant*
- `let x=1 in let f=fun y -> x in let x=2 in f 0` *hint: three uses of let, one use of anonymous function, one use of application, two uses of variable, three uses of constant*

Exercise: more evaluation [★★]

Evaluate these:

- `let x = 2 + 2 in x + x`
- `let x = 1 in let x = x + x in x + x`
- `let f = fun x -> fun y -> x + y in let g = f 3 in g 2`
- `let f = fst (let x = 3 in fun y -> x, 2) in f 0`

Exercise: dynamic scope [★★★]

Use dynamic scope to evaluate the following expression. You do not need to write down all of the evaluation steps unless you find it helpful. Compare your answer to the answer you would expect from a language with lexical scope.

```
let x = 5 in
let f y = x + y in
let x = 4 in
f 3
```

Exercise: more dynamic scope [★★★]

Use dynamic scope to evaluate the following expressions. Compare your answers to the answers you would expect from a language with lexical scope.

Expression 1:

```
let x = 5 in
let f y = x + y in
let g x = f x in
let x = 4 in
g 3
```

Expression 2:

```
let f y = x + y in
let x = 3 in
let y = 4 in
f 2
```

Exercise: constraints [★★]

Show the derivation of the `env |- e : t -| C` relation for these expressions:

```
1. fun x -> ( + ) 1 x
2. fun b -> if b then false else true
3. fun x -> fun y -> if x <= y then y else x
```

Exercise: unify [★★]

Use the unification algorithm to solve the following system of constraints. Your answer should be a *substitution*, in the sense that the unification algorithm defines that term.

```
X = int
Y = X -> X
```

Exercise: unify more [★★★]

Use the unification algorithm to solve the following system of constraints. Your answer should be a *substitution*, in the sense that the unification algorithm defines that term.

```
X -> Y = Y -> Z
     Z = U -> W
```

Exercise: infer apply [★★★]

Using the HM type inference algorithm, infer the type of the following definition:

```
let apply f x = f x
```

Remember to go through these steps:

- desugar the definition entirely (i.e., construct an AST)
- collect constraints
- solve the constraints with unification

□

Exercise: infer double [★★★]

Using the HM type inference algorithm, infer the type of the following definition:

```
let double f x = f (f x)
```

Exercise: infer S [★★★★]

Using the HM type inference algorithm, infer the type of the following definition:

```
let s x y z = (x z) (y z)
```

Part VI

Lagniappe

CHAPTER

TWELVE

THE CURRY-HOWARD CORRESPONDENCE

Note: A *lagniappe* is a small and unexpected gift — a little "something extra". Please enjoy this little chapter, which contains one of the most beautiful results in the entire book. It is based on the paper Propositions as Types by Philip Wadler. You can watch an entertaining recorded lecture by Prof. Wadler on it, in addition to our lecture below.

As we observed long ago, OCaml is a language in the ML family, and ML was originally designed as the meta language for a theorem prover—that is, a computer program designed to help prove and check the proofs of logical formulas. When constructing proofs, it's desirable to make sure that you can only prove true formulas, to make sure that you don't make incorrect arguments, etc.

The dream would be to have a computer program that can determine the truth or falsity of any logical formula. For some formulas, that is possible. But, one of the groundbreaking results in the early 20th century was that it is *not* possible, in general, for a computer program to do this. Alonzo Church and Alan Turing independently showed this in 1936. Church used the *lambda calculus* as a model for computers; Turing used what we now call *Turing machines*. The *Church-Turing thesis* is a hypothesis that says the lambda calculus and Turing machines both formalize what "computation" informally means.

Instead of focusing on that impossible task, we're going to focus on the relationship between proofs and programs. It turns out the two are deeply connected in a surprising way.

12.1 Computing with Evidence

We're accustomed to OCaml programs that manipulate data, such as integers and variants and functions. Those data values are always typed: at compile time, OCaml infers (or the programmer annotates) the types of expressions. For example, `3110 : int`, and `[] : 'a list`. We long ago learned to read those as "`3110` has type `int`", and "`[]` has type `'a list`".

Let's try a different reading now. Instead of "has type", let's read "is evidence for". So, `3110` is evidence for `int`. What does that mean? Think of a type as a set of values. So, `3110` is evidence that type is not empty. Likewise, `[]` is evidence that the type `'a list` is not empty. We say that the type is *inhabited* if it is not empty.

Are there empty types? There actually is one in OCaml, though we've never had reason to mention it before. It's possible to define a variant type that has no constructors:

```
type empty = |
```

We could have called that type anything we wanted instead of `empty`; the special syntax there is just writing `|` instead of actual constructors. (Note, that syntax might give some editors some trouble. You might need to put double-semicolon after it to get the formatting right.) It is impossible to construct a value of type `empty`, exactly because it has no constructors. So, `empty` is not inhabited.

Under our new reading based on evidence, we could think about functions as ways to manipulate and transform evidence—just as we are already accustomed to thinking about functions as ways to manipulate and transform data. For example, the following functions construct and destruct pairs:

```
let pair x y = (x, y)
let fst (x, y) = x
let snd (x, y) = y
```

We could think of `pair` as a function that takes in evidence for `'a` and evidence for `'b`, and gives us back evidence for `'a * 'b`. That latter piece of evidence is the the pair `(x, y)` containing the individual pieces of evidence, `x` and `y`. Similarly, `fst` and `snd` extract the individual pieces of evidence from the pair. Thus,

- If you have evidence for `'a` and evidence for `'b`, you can produce evidence for `'a` and `'b`.
- If you have evidence for `'a` and `'b`, then you can produce evidence for `'a`.
- If you have evidence for `'a` and `'b`, then you can produce evidence for `'b`.

In learning to do proofs (say, in a discrete mathematics class), you will have learned that in order to prove two statements hold, you individually have to prove that each holds. That is, to prove the conjunction of A and B, you must prove A as well as prove B. Likewise, if you have a proof of the conjunction of A and B, then you can conclude A holds, and you can conclude B holds. We can write those patterns of reasoning as logical formulas, using `/\` to denote conjunction and `->` to denote implication:

```
A -> B -> A /\ B
A /\ B -> A
A /\ B -> B
```

Proofs are a form of evidence: they are logical arguments about the truth of a statement. So another reading of those formulas would be:

- If you have evidence for A and evidence for B, you can produce evidence for A and B.
- If you have evidence for A and B, then you can produce evidence for A.
- If you have evidence for A and B, then you can produce evidence for B.

Notice how we now have given the same reading for programs and for proofs. They are both ways of manipulating and transforming evidence. In fact, take a close look at the types for `pair`, `fst`, and `snd` compared to the logical formulas that describe valid patterns of reasoning:

```
val pair : 'a -> 'b -> 'a * 'b          A -> B -> A /\ B
val fst : 'a * 'b -> 'a                 A /\ B -> A
val snd : 'a * 'b -> 'b                 A /\ B -> B
```

If you replace `'a` with A, and `'b` with B, and `*` with `/\`, **the types of the programs are identical to the formulas!**

12.2 The Correspondence

What we have just discovered is that computing with evidence corresponds to constructing valid logical proofs. This correspondence is not just an accident that occurs with these three specific programs. Rather, it is a deep phenomenon that links the fields of programming and logic. Aspects of it have been discovered by many people working in many areas. So, it goes by many names. One common name is *the Curry-Howard correspondence*, named for logicians Haskell Curry (for whom the functional programming language Haskell is named) and William Howard. This correspondence links ideas from programming to ideas from logic:

- Types correspond to logical formulas (aka *propositions*).

- Programs correspond to logical proofs.
- Evaluation corresponds to simplification of proofs.

We've already seen the first two of those correspondences. The types of our three little programs corresponded to formulas, and the programs themselves corresponded to the reasoning done in proofs involving conjunctions. We haven't seen the third yet; we will later.

Let's dig into each of the correspondences to appreciate them more fully.

12.3 Types Correspond to Propositions

In *propositional logic*, formulas are created with atomic propositions, negation, conjunction, disjunction, and implication. The following BNF describes propositional logic formulas:

```
p ::= atom
    | ~ p       (* negation *)
    | p /\ p    (* conjunction *)
    | p \/ p    (* disjunction *)
    | p -> p    (* implication *)

atom ::= <identifiers>
```

For example, `raining /\ snowing /\ cold` is a proposition stating that it is simultaneously raining and snowing and cold (a weather condition known as *Ithacating*). An atomic proposition might hold of the world, or not. There are two distinguished atomic propositions, written `true` and `false`, which are always hold and never hold, respectively.

All these *connectives* (so-called because they connect formulas together) have correspondences in the types of functional programs.

Conjunction. We have already seen that the `/\` connective corresponds to the `*` type constructor. Proposition `A /\ B` asserts the truth of both `A` and `B`. An OCaml value of type `a * b` contains values both of type `a` and `b`. Both `/\` and `*` thus correspond to the idea of pairing or products.

Implication. The implication connective `->` corresponds to the function type constructor `->`. Proposition `A -> B` asserts that if you can show that `A` holds, then you can show that `B` holds. In other words, by assuming `A`, you can conclude `B`. In a sense, that means you can transform `A` into `B`. An OCaml value of type `a -> b` expresses that idea even more clearly. Such a value is a function that transforms a value of type `a` into a value of type `b`. Thus, if you can show that `a` is inhabited (by exhibiting a value of that type), you can show that `b` is inhabited (by applying the function of type `a -> b` to it). So, `->` corresponds to the idea of transformation.

Disjunction. The disjunction connective `\/` corresponds to something a little more difficult to express concisely in OCaml. Proposition `A \/ B` asserts that either you can show `A` holds or `B` holds. Let's strengthen that to further assert that in addition to showing *one* of them holds, you have to specify *which one* you are showing. Why would that matter?

Suppose we were working on a proof of the *twin prime conjecture*, an unsolved problem that states there are infinitely many twin primes (primes of the form n and $n + 2$, such as 3 and 5, or 5 and 7). Let the atomic proposition `TP` denote that there are infinitely many twin primes. Then the proposition `TP \/ ~ TP` seems reasonable: either there are infinitely many twin primes, or there aren't. We wouldn't even have to figure out how to prove the conjecture! But if we strengthen the meaning of `\/` to be that we have to state *which one* of the sides, left or right, holds, then we would either have to give a proof or disproof of the conjecture. No one knows how to do that currently. So we could not prove `TP \/ ~ TP`.

Henceforth we will use `\/` in that stronger sense of having to identify whether we are giving a proof of the left or the right side proposition. Thus, we can't necessarily conclude `p \/ ~ p` for any proposition `p`: it will matter whether we can prove `p` or `~ p` on their own. Technically, this makes our propositional logic *constructive* rather than *classical*. In constructive logic we must construct the proof of the individual propositions. Classical logic (the traditional way `\/` is understood) does not require that.

Returning to the correspondence between disjunction and variants, consider this variant type:

```
type ('a, 'b) disj = Left of 'a | Right of 'b
```

A value `v` of that type is either `Left a`, where `a : 'a`; or `Right b`, where `b : 'b`. That is, `v` identifies (i) whether it is tagged with the left constructor or the right constructor, and (ii) carries within it exactly one sub-value of type either `'a` or `'b`—not two subvalues of both types, which is what `'a * 'b` would be.

Thus, the (constructive) disjunction connective `\/` corresponds to the `disj` type constructor. Proposition `A \/ B` asserts that either `A` or `B` holds as well as which one, left or right, it is. An OCaml value of type `('a, 'b) disj` similarly contains a value of type either `'a` or `'b` as well as identifying (with the `Left` or `Right` constructor) which one it is. Both `\/` and `disj` therefore correspond to the idea of unions.

Truth and Falsity The atomic proposition `true` is the only proposition that is guaranteed to always hold. There are many types in OCaml that are always inhabited, but the simplest of all of them is `unit`: there is one value `()` of type `unit`. So the proposition `true` (best) corresponds to the type `unit`.

Likewise, the atomic proposition `false` is the only proposition that is guaranteed to never hold. That corresponds to the `empty` type we introduced earlier, which has no constructors. (Other names for that type could include `zero` or `void`, but we'll stick with `empty`.)

There is a subtlety with `empty` that we should address. The type has no constructors, but it is nonetheless possible to write expressions that have type `empty`. Here is one way:

```
let rec loop x = loop x
```

Now if you enter this code in utop you will get no response:

```
let e : empty = loop ()
```

That expression type checks successfully, then enters an infinite loop. So, there is never any value of type `empty` that is produced, even though the expression has that type.

Here is another way:

```
let e : empty = failwith ""
```

Again, the expression type checks, but it never produces an actual value of type `empty`. Instead, this time an exception is produced.

So the type `empty` is not inhabited, even though there are some expressions of that type. But, **if we require programs to be total**, we can rule out those expressions. That means eliminating programs that raise exceptions or go into an infinite loop. We did in fact make that requirement when we started discussing formal methods, and we will continue to assume it.

Negation. This connective is the trickiest. Let's consider negation to actually be syntactic sugar. In particular, let's say that the propositional formula `~ p` actually means this formula instead: `p -> false`. Why? The formula `~ p` should mean that `p` does not hold. So if `p` *did* hold, then it would lead to a contradiction. Thus, given `p`, we could conclude `false`. This is the standard way of understanding negation in constructive logic.

Given that syntactic sugar, `~ p` therefore corresponds to a function type whose return type is `empty`. Such a function could never actually return. Given our ongoing assumption that programs are total, that must mean it's impossible to even call that function. So, it must be impossible to construct a value of the function's input type. Negation therefore corresponds to the idea of impossibility, or contradiction.

Propositions as types. We have now created the following correspondence that enables us to read propositions as types:

- `/\` and `*`

- `->` and `->`
- `\/` and `disj`
- `true` and `unit`
- `false` and `empty`
- `~` and `... -> false`

But that is only the first level of the Curry-Howard correspondence. It goes deeper…

12.4 Programs Correspond to Proofs

We have seen that programs and proofs are both ways to manipulate and transform evidence. In fact, every program **is** a proof that the type of the program is inhabited, since the type checker must verify that the program is well typed.

The details of type checking, though, lead to an even more compelling correspondence between programs and proofs. Let's restrict our attention to programs and proofs involving just conjunction and implication, or equivalently, pairs and functions. (The other propositional connectives could be included as well, but require additional work.)

Type checking rules. For type checking, we gave many *rules* to define when a program is well typed. Here are rules for variables, functions, and pairs:

```
{x : t, ...} |- x : t
```

A variable has whatever type the environment says it has.

```
env |- fun x -> e : t -> t'
if env[x -> t] |- e : t'
```

An anonymous function `fun x -> e` has type `t -> t'` if `e` has type `t'` in a static environment extended to bind `x` to type `t`.

```
env |- e1 e2 : t'
if env |- e1 : t -> t'
and env |- e2 : t
```

An application `e1 e2` has type `t'` if `e1` has type `t -> t'` and `e2` has type `t`.

```
env |- (e1, e2) : t1 * t2
if env |- e1 : t1
and env |- e2 : t2
```

The pair `(e1, e2)` has type `t1 * t2` if `e1` has type `t1` and `e2` has type `t2`.

```
env |- fst e : t1
if env |- e : t1 * t2

env |- snd e : t2
if env |- e : t1 * t2
```

If `e` has type `t1 * t2`, then `fst e` has type `t1`, and `snd e` has type `t2`.

Proof trees. Another way of expressing those rules would be to draw *proof trees* that show the recursive application of rules. Here are those proof trees:

```
--------------------
{x : t, ...} |- x : t


   env[x -> t1] |- e2 : t2
-----------------------------
env |- fun x -> e2 : t1 -> t2


env |- fun x -> e2 : t1 -> t2        env |- e1 : t1
-------------------------------------------------
          env |- (fun x -> e2) e1 : t2


env |- e1 : t1         env |- e2 : t2
-------------------------------------
     env |- (e1, e2) : t1 * t2


 env |- e : t1 * t2
--------------------
 env |- fst e : t1


 env |- e : t1 * t2
--------------------
 env |- snd e : t2
```

Proof trees, logically. Let's rewrite each of those proof trees to eliminate the programs, leaving only the types. At the same time, let's use the propositions-as-types correspondence to re-write the types as propositions:

```
--------
 p |- p


   env, p1 |- p2
-------------------
  env |- p1 -> p2


env |- p1 -> p2     env |- p1
------------------------------
        env |- p2


env |- p1      env |- p2
------------------------
    env |- p1 /\ p2


env |- p1 /\ p2
---------------
   env |- p1


env |- p1 /\ p2
```

(continues on next page)

(continued from previous page)

```
---------------
   env |- p2
```

Each rule can now be read as a valid form of logical reasoning. Whenever we write `env |- t`, it means that "from the assumptions in `env`, we can conclude `p` holds". A rule, as usual, means that from all the premisses above the line, the conclusion below the line holds.

Proofs and programs. Now consider the following proof tree, showing the derivation of the type of a program:

```
------------------------          ------------------------
{p : a * b} |- p : a * b          {p : a * b} |- p : a * b
------------------------          ------------------------
{p : a * b} |- snd p : b          {p : a * b} |- fst p : a
-------------------------------------------------------------
        {p : a * b} |- (snd p, fst p) : b * a
     ---------------------------------------------
     {} |- fun p -> (snd p, fst p) : a * b -> b * a
```

That program shows that you can swap the components of a pair, thus swapping the types involved.

If we erase the program, leaving only the types, and re-write those as propositions, we get this proof tree:

```
----------------           -----------------
a /\ b |- a /\ b           a /\ b |- a /\ b
----------------           -----------------
  a /\ b |- b                 a /\ b |- a
-----------------------------------------
              a /\ b |- b /\ a
           ------------------------
            {} |- a /\ b -> b /\ a
```

And that is a valid proof tree for propositional logic. It shows that you can swap the sides of a conjunction.

What we see from those two proof trees is: **a program is a proof**. A well-typed program corresponds to the proof of a logical proposition. It shows how to compute with evidence, in this case transforming a proof of `a /\ b` into a proof of `b /\ a`.

Programs are proofs. We have now created the following correspondence that enables us to read programs as proofs:

- A program `e : t` corresponds to a proof of the logical formula to which `t` itself corresponds.
- The proof tree of `|- t` corresponds to the proof tree of `{} |- e : t`.
- The proof rules for typing a program correspond to the rules for proving a proposition.

But that is only the second level of the Curry-Howard correspondence. It goes deeper…

12.5 Evaluation Corresponds to Simplification

We will treat this part of the correspondence only briefly. Consider the following program:

```
fst (a, b)
```

That program would of course evaluate to `a`.

Next, consider the typing derivation of that program. The variables `a` and `b` must be bound to some types in the static environment for the program to type check.

```
------------------------          --------------------------
{a : t, b : t'} |- a : t          {a : t, b : t'} |- b : t'
-------------------------------------------------------------
         {a : t, b : t'} |- (a, b) : t * t'
         ----------------------------------
         {a : t, b : t'} |- fst (a, b) : t
```

Erasing that proof tree to just the propositions, per the proofs-as-programs correspondence, we get this proof tree:

```
-----------             -----------
t, t' |-  t             t, t' |- t'
-----------------------------------
         t, t' |- t /\ t'
         ----------------
            t, t' |- t
```

However, there is a much simpler proof tree with the same conclusion:

```
 ------------
  t, t' |- t
```

In other words, we don't need the detour through proving `t /\ t'` to prove `t`, if `t` is already an assumption. We can instead just directly conclude `t`.

Likewise, there is a simpler typing derivation corresponding to that same simpler proof:

```
------------------------
{a : t, b : t'} |- a : t
```

Note that typing derivation is for the program `a`, which is exactly what the bigger program `fst (a, b)` evaluates to.

Thus evaluation of the program causes the proof tree to simplify, and the simplified proof tree is actually (through the proofs-as programs correspondence) a simpler proof of the same proposition. **Evaluation therefore corresponds to proof simplification.** And that is the final level of the Curry-Howard correspondence.

12.6 What It All Means

Logic is a fundamental aspect of human inquiry. It guides us in reasoning about the world, in drawing valid inferences, in deducing what must be true vs. what must be false. Training in logic and argumentation—in various fields and disciplines—is one of the most important parts of a higher education.

The Curry-Howard correspondence shows that logic and computation are fundamentally linked in a deep and maybe even mysterious way. The basic building blocks of logic (propositions, proofs) turn out to correspond to the basic building blocks of computation (types, functional programs). Computation itself, the evaluation or simplification of expressions, turns out to correspond to simplification of proofs. The very task that computers do therefore is the same task that humans do in trying to present a proof in the best possible way.

Computation is thus intrinsically linked to reasoning. And functional programming is a fundamental part of human inquiry.

Could there *be* a better reason to study functional programming?

12.7 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Exercise: propositions as types [★★]

For each of the following propositions, write its corresponding type in OCaml.

- `true -> p`
- `p /\ (q /\ r)`
- `(p \/ q) \/ r`
- `false -> p`

Exercise: programs as proofs [★★★]

For each of the following propositions, determine its corresponding type in OCaml, then write an OCaml `let` definition to give a program of that type. Your program proves that the type is *inhabited*, which means there is a value of that type. It also proves the proposition holds.

- `p /\ q -> q /\ p`
- `p \/ q -> q \/ p`

Exercise: evaluation as simplification [★★★]

Consider the following OCaml program:

```
let f x = snd ((fun x -> x, x) (fst x))
```

- What is the type of that program?
- What is the proposition corresponding to that type?
- How would `f (1,2)` evaluate in the small-step semantics?

- What simplified implementation of `f` does that evaluation suggest? (or perhaps there are several, though one is probably the simplest?)
- Does your simplified `f` still have the same type as the original? (It should.)

Your simplified `f` and the original `f` are both proofs of the same proposition, but evaluation has helped you to produce a simpler proof.

Part VII

Appendix

CHAPTER

THIRTEEN

BIG-OH NOTATION

What does it mean to be *efficient*? Cornell professors Bobby Kleinberg and Eva Tardos have a wonderful explanation in chapter 2 of their textbook, *Algorithm Design* (2006). This appendix is a summary and reinterpretation of that explanation from a functional programming perspective. The ultimate answer will be that an algorithm is *efficient* if its worst-case running time on input size n is $O(n^d)$ for some constant d. But it will take us several steps to build up to that definition.

13.1 Algorithms and Efficiency, Attempt 1

A naive attempt at defining efficiency might proceed as follows:

Attempt 1: An algorithm is efficient if, when implemented, it runs in a small amount of time on particular input instances.

But there are many problems with that definition, such as:

- Inefficient algorithms can run quickly on small test cases.
- Fast processors and optimizing compilers can make inefficient algorithms run quickly.
- Efficient algorithms can run slowly when coded sloppily.
- Some input instances are harder than others.
- Efficiency on small inputs doesn't imply efficiency on large inputs.
- Some clients can afford to be more patient than others; quick for me might be slow for you.

Lesson 1: One lesson learned from that attempt is: time measured by a clock is not the right metric for algorithm efficiency. We need a metric that is reasonably independent of the hardware, compiler, other software that is running, etc. Perhaps a good metric would be to give up on time and instead count the number of *steps* taken during evaluation.

But, now we have a new problem: how should we define a "step"? It needs to be something machine independent. It ought to somehow represent a primitive unit of computation. There's a lot of flexibility. Here are some common choices:

- In pseudocode, we might think of a step as being a single line.
- In imperative languages, we might count assignments, array indexes, pointer dereferences, and arithmetic operations as steps.
- In OCaml, we could count function or operator application, let bindings, and choosing a branch of an `if` or `match` as steps.

In reality, all of those "steps" could really take a different amount of time. But in practice, that doesn't really matter much.

Lesson 2: Another lesson we learned from attempt 1 was: running time on a particular input instance is not the right metric. We need a metric that can predict running time on any input instance. So instead of using the particular input (e.g., a number, or a matrix, or a text document), it might be better to use the *size* of the input (e.g., the number of bits it takes to represent the number, or the number of rows and columns in the matrix, or the number of bytes in the document) as the metric.

But again we have a new problem: how to define "size"? As in the examples we just gave, size should be some measure of how big an input is compared to other inputs. Perhaps the most common representation of size in the context of data structures is just the number of elements maintained by the data structure: the number of nodes in a list, or the number of nodes and edges in a graph, etc.

Could an algorithm run in a different amount of time on two inputs of the same "size"? Sure. For example, multiplying a matrix by all zeroes might be faster than multiplying by arbitrary numbers. But in practice, size matters more than exact inputs.

Lesson 3: A third lesson we learned from attempt 1 was that "small amount of time" is too relative a term. We want a metric that is reasonably objective, rather than relying on subjective notions of what constitutes "small".

One sort-of-okay idea would be that an efficient algorithm needs to beat *brute-force search*. That means enumerating all answers one-by-one, checking each to see whether it's right. For example, a brute-force sorting algorithm would enumerate every possible permutation of a list, checking to see whether it's a sorted version of the input list. That's a terrible sorting algorithm! Certainly quicksort beats it.

Brute-force search is the simple, dumb solution to nearly any algorithmic problem. But it requires enumeration of a huge space. In fact, an exponentially-sized space. So a better idea would be doing less than exponential work. What's less than exponential (e.g., 2^n)? One possibility is polynomial (e.g., n^2).

An immediate objection might be that polynomials come in all sizes. For example, n^{100} is way bigger than n^2. And some non-polynomials, such as $n^{1+.02(\log n)}$, might do an adequate job of beating exponentials. But in practice, polynomials do seem to work fine.

13.2 Algorithms and Efficiency, Attempt 2

Combining lessons 1 through 3 from Attempt 1, we have a second attempt at defining efficiency:

Attempt 2: An algorithm is efficient if its maximum number of execution steps is polynomial in the size of its input.

Note how all three ideas come together there: steps, size, polynomial.

But if we try to put that definition to use, it still isn't perfect. Coming up with an exact formula for the maximum number of execution steps can be insanely tedious. For example, in one other algorithm textbook, the authors develop this following polynomial for the number of execution steps taken by a pseudo-code implementation of insertion sort:

$$c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

No need for us to explain what all the variables mean. It's too complicated. Our hearts go out to the poor grad student who had to work out that one!

Note: That formula for running time of insertion sort is from *Introduction to Algorithms*, 3rd edition, 2009, by Cormen, Leiserson, Rivest, and Stein. We aren't making fun of them. They would also tell you that such formulas are too complicated.

Precise execution bounds like that are exhausting to find and somewhat meaningless. If it takes 25 steps in Java pseudocode, but compiled down to RISC-V would take 250 steps, is the precision useful?

In some cases, yes. If you're building code that flies an airplane or controls a nuclear reactor, you might actually care about precise, real-time guarantees.

But otherwise, it would be better for us to identify broad classes of algorithms with similar performance. Instead of saying that an algorithm runs in

$$1.62n^2 + 3.5n + 8$$

steps, how about just saying it runs in n^2 steps? That is, we could ignore the *low-order terms* and the *constant factor* of the highest-order term.

We ignore low-order terms because we want to THINK BIG. Algorithm efficiency is all about explaining the performance of algorithms when inputs get really big. We don't care so much about small inputs. And low-order terms don't matter when we think big. The following table shows the number of steps as a function of input size N, assuming each step takes 1 microsecond. "Very long" means more than the estimated number of atoms in the universe.

|N|N^2|N^3|2^N :—:|:—:|:—:|:—:|:—:|:—: N=10|< 1 sec|< 1 sec|< 1 sec|< 1 sec N=100|< 1 sec|< 1 sec|1 sec|1017 years N=1,000|< 1 sec|1 sec|18 min|very long N=10,000|< 1 sec|2 min|12 days|very long N=100,000|< 1 sec|3 hours|32 years|very long N=1,000,000|1 sec|12 days|104 years|very long

As you can see, when inputs get big, there's a serious difference between N^3 and N^2 and N. We might as well ignore low-order terms, because they are completely dominated by the highest-order term when we think big.

What about constant factors? My current laptop might be 2x faster (that is, a constant factor of 2) than the one I bought several years ago, but that's not an interesting property of the algorithm. Likewise, $1.62n^2$ steps in pseudocode might be $1620n^2$ steps in assembly (that is, a constant factor of 1000), but it's again not an interesting property of the algorithm. So, should we really care if one algorithm takes 2x or 1000x longer than another, if it's just a constant factor?

The answer is: maybe. Performance tuning in real-world code is about getting the constants to be small. Your employer might be really happy if you make something run twice as fast! But that's not about the **algorithm.** When we're measuring algorithm efficiency, in practice the constant factors just don't matter much.

So all that argues for having an **imprecise abstraction** to measure running time. Instead of $1.62n^2 + 3.5n + 8$, we can just write n^2. Imprecise abstractions are nothing new to you. You might write ± 1 to imprecisely abstract a quantity within 1. In computer science, you already know that we use Big-Oh notation as an imprecise abstraction: $1.62n^2 + 3.5n + 8$ is $O(n^2)$.

13.3 Big-Ell Notation

Before reviewing Big-Oh notation, let's start with something simpler that you might not have seen before: Big-Ell notation.

Big-Ell is an imprecise abstraction of natural numbers less than or equal to another number, hence the L. It's defined as follows:

$$L(n) = \{m \mid 0 \leq m \leq n\}$$

where m and n are natural numbers. That is, $L(n)$ represents all the natural numbers less than or equal to n. For example, $L(5) = \{0, 1, 2, 3, 4, 5\}$.

Could you do arithmetic with Big-Ell? For example, what would $1 + L(5)$ be? It's not a well-posed question, to be honest: addition is an operation we think of being defined on integers, not sets of integers. But a reasonable interpretation of $1 + \{0, 1, 2, 3, 4, 5\}$ could be doing the addition for each element in the set, yielding $\{1, 2, 3, 4, 5, 6\}$. Note that $\{1, 2, 3, 4, 5, 6\}$ is a proper subset of $\{0, 1, 2, 3, 4, 5, 6\}$, and the latter is $L(6)$. So we could say that $1 + L(5) \subseteq L(6)$. We could even say that $1 + L(5) \subseteq L(7)$, but it's not *tight*: the former subset relation included the fewest possible extra elements, whereas the latter was *loose* by needlessly including extra.

For more about Big Ell, see *Concrete Mathematics*, chapter 9, 1989, by Graham, Knuth, and Patashnik.

13.4 Big-Oh Notation

If you understand Big-Ell, and you understand functional programming, here's some good news: you can easily understand Big-Oh.

Let's build up the definition of Big-Oh in a few steps. We'll start with version 1, which we'll write as O_1. It's based on L:

- $L(n)$ represents any **natural number** that is less than or equal to a **natural number** n.
- $O_1(g)$ represents any **natural function** that is less than or equal to a **natural function** g.

A *natural function* is just a function on natural numbers; that is, its type is $\mathbb{N} \to \mathbb{N}$.

All we do with O_1 is upgrade from **natural numbers** to **natural functions**. So Big-Oh version 1 is just the *higher-order* version of Big-Ell. How about that!

Of course, we need to work out what it means for a function to be less than another function. Here's a reasonable formalization:

Big-Oh Version 1: $O_1(g) = \{f \mid \forall n \,.\, f(n) \leq g(n)\}$

For example, consider the function that doubles its input. In math textbooks, that function might be written as $g(n) = 2n$. In OCaml we would write `let g n = 2 * n` or `let g = fun n -> 2 * n` or just anonymously as `fun n -> 2 * n`. In math that same anonymous function would be written with lambda notation as $\lambda n.2n$. Proceeding with lambda notation, we have:

$$O_1(\lambda n.2n) = \{f \mid \forall n.f(n) \leq 2n\}$$

and therefore

- $(\lambda n.n) \in O_1(\lambda n.2n)$,
- $(\lambda n.\frac{n}{2}) \in O_1(\lambda n.2n)$, but
- $(\lambda n.3n) \notin O_1(\lambda n.2n)$.

Next, recall that in defining algorithmic efficiency, we wanted to ignore constant factors. O_1 does not help us with that. We'd really like for all these functions:

- $(\lambda n.n)$
- $(\lambda n.2n)$
- $(\lambda n.3n)$

to be in $O(\lambda n.n)$.

Toward that end, let's define O_2 to ignore constant factors:

Big-Oh Version 2: $O_2(g) = \{f \mid \exists c > 0 \forall n \,.\, f(n) \leq cg(n)\}$

That existentially-quantified positive constant c lets us "bump up" the function g to whatever constant factor we need. For example,

$$O_2(\lambda n.n^3) = \{f \mid \exists c > 0 \forall n.f(n) \leq cn^3\}$$

and therefore $(\lambda n.3n^3) \in O_2(\lambda n.n^3)$, because $3n^3 \leq cn^3$ if we take $c = 3$, or $c = 4$, or any larger c.

Finally, recall that we don't care about small inputs: we want to THINK BIG when we analyze algorithmic efficiency. It doesn't matter whether the running time of an algorithm happens to be a little faster or a little slower for small inputs. In fact, we could just hardcode a lookup table for those small inputs if the algorithm is too slow on them! What matters really is the performance on big-sized inputs.

Toward that end, let's define O_3 to ignore small inputs:

Big-Oh Version 3: $O_3(g) = \{f \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 \ . \ f(n) \leq cg(n)\}$

That existentially quantified positive constant n_0 lets us "ignore" all inputs of that size or smaller. For example,

$$O_3(\lambda n.n^2) = \{f \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 \ . \ f(n) \leq cn^2\}$$

and therefore $(\lambda n.2n) \in O_3(\lambda n.n^2)$, because $2n \leq cn^2$ if we take $c = 2$ and $n_0 = 2$. Note how we get to ignore the fact that $\lambda n.2n$ is temporarily a little too big at $n = 1$ by picking $n_0 = 2$. That's the power of ignoring "small" inputs.

13.5 Big-Oh, Finished

Version 3 is the right definition of Big-Oh. We repeat it here, for real:

Big-Oh: $O(g) = \{f \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 \ . \ f(n) \leq cg(n)\}$

That's the final, important version you should know. But don't just memorize it. If you understand the derivation we gave here, you'll be able to recreate it from scratch anytime you need it.

Big-Oh is called an *asymptotic upper bound*. If $f \in O(g)$, then f is at least as efficient as g, and might be more efficient.

13.6 Big-Oh Notation Warnings

Warning 1. Because it's an upper bound, we can always inflate a Big-Oh statement: for example, if $f \in O(n^2)$, then also $f \in O(n^3)$, and $f \in O(2^n)$, etc. But our goal is always to give *tight* upper bounds, whether we explicitly say that or not. So when asked what the running time of an algorithm is, you must always give the tightest bound you can with Big-Oh.

Warning 2. Instead of $O(g) = \{f \mid ...\}$, most authors instead write $O(g(n)) = \{f(n) \mid ...\}$. They don't really mean g applied to n. They mean a function g parameterized on input n but not yet applied. This is badly misleading and generally a result of not understanding anonymous functions. Moral of that story: more people need to study functional programming.

Warning 3. Instead of $\lambda n.2n \in O(\lambda n.n^2)$ nearly all authors write $2n = O(n^2)$. This is a hideous and inexcusable abuse of notation that should never have been allowed and yet has permanently infected the computer science consciousness. The standard defense is that $=$ here should be read as "is" not as "equals". That is patently ridiculous, and even those who make that defense usually have the good grace to admit it's nonsense. Sometimes we become stuck with the mistakes of our ancestors. This is one of those times. Be careful of this "one-directional equality" and, if you ever have a chance, teach your (intellectual) children to do better.

13.7 Algorithms and Efficiency, Attempt 3

Let's review. Our first attempt at defining efficiency was:

Attempt 1: An algorithm is efficient if, when implemented, it runs in a small amount of time on particular input instances.

By replacing time with steps, particular instances with input size, and small with polynomial, we improved that to:

Attempt 2: An algorithm is efficient if its maximum number of execution steps is polynomial in the size of its input.

And that's really a pretty good definition. But using Big-Oh notation to make it a little more concrete, we can produce our third and final attempt:

Attempt 3: An algorithm is efficient if its worst-case running time on input size n is $O(n^d)$ for some constant d.

By "worst-case running time" we mean the same thing as "maximum number of execution steps", just expressed in different and probably more common words. The worst-case is when execution takes the longest. "Time" is a common euphemism here for execution steps, and is used to emphasize we're thinking about how long a computation takes.

Space is the most common other feature of efficiency to consider. Algorithms can be more or less efficient at requiring constant or linear space, for example. You're already familiar with that from tail recursion and lists in OCaml.

CHAPTER

FOURTEEN

VIRTUAL MACHINE

A *virtual machine* is what the name suggests: a machine running virtually inside another machine. With virtual machines, there are two operating systems involved: the *host* operating system (OS) and the *guest* OS. The host is your own native OS (maybe Windows). The guest is the OS that runs inside the host.

The virtual machine (VM) we provide here has OCaml pre-installed in an Ubuntu guest OS. Ubuntu is a free Linux OS, and is an ancient African word meaning "humanity to others". The process we use to create the VM is documented here.

14.1 Installing the VM

- Download and install Oracle's free VirtualBox for your host OS. Or, if you already had it installed, make sure you update to the latest version of VirtualBox before proceeding.
- Download our VM. Don't worry about the "We're sorry, the preview didn't load" message you see. Just click the Download button and save the `.ova` file wherever you like. It's about a 6GB file, so the download might take awhile.
- Launch VirtualBox, select File → Import Appliance, and choose the `.ova` file you just downloaded. Click Next, then Import.

14.2 Starting the VM

- Select cs3110-2021fa-ubuntu from the list of machines in VirtualBox. Click Start. At this point various errors can occur that depend on your hardware, hence are hard to predict.
  - If you get an error about "VT-x/AMD-V hardware acceleration", you most likely need to access your computer's BIOS settings and enable virtualization. The details of that will vary depending on the model and manufacturer of your computer. Try googling "enable virtualization ", substituting for the manufacturer and model of your machine. This Red Hat Linux page might also help.
  - If the machine just freezes or blacks out or aborts, you might need to adjust the memory provided to it by your host OS. Select the VM in Virtual Box, click Settings, and look at the System and Display settings. You might need to adjust the Base Memory (under System → Motherboard) or the Video Memory (under Display → Screen). Those sliders have color coding underneath them to indicate what good amounts might be on your computer. Make sure nothing is in the red zone, and try some lower or higher settings to see if they help. If the sliders are greyed out and won't permit adjustment, it means the VM is still running: you can't change the amount of memory while the guest OS is active; so, shut down the VM (see below) and try again.
  - If you have a monitor with high pixel density (e.g., an Apple Retina display), the VM window might be incredibly tiny. In VirtualBox go to Settings → Display → Scale Factor and increase it as needed, perhaps to 200%.

- The VM will log you in automatically. The username is `camel` and the password is `camel`. To change your password, run `passwd` from the terminal and follow the prompts. If you'd rather have your own username, you are welcome to go to Settings → Users to create a new account. Just be aware that OPAM and VS Code won't be installed for that user. You'll need to follow the *install instructions* to add them.

14.3 Stopping the VM

You can use Ubuntu's own menus to safely shutdown or reboot the VM. But more often you will likely use VirtualBox to close the VM by clicking the VM window's "X" icon in the host OS. Then you will be presented with three options that VirtualBox doesn't explain very well:

- *Save the machine state.* This option is what you normally want. It's like closing the lid on your laptop: it puts it to sleep, and it can quickly wake.
- *Send the shutdown signal.* This option is like shutting down a machine you don't intend to use for a long time, or before unplugging a desktop machine from the wall. When you start the machine again later, it will have to boot from scratch, which takes longer.
- *Power off the machine.* **This option is dangerous.** It is the equivalent of pulling the power cord of a desktop machine from the wall while the machine is still running: it causes the operating system to suddenly quit without doing any cleanup. Doing this even just a handful of times could cause the file system to become corrupted, which will cause you to lose all your work and have to reinstall the VM from scratch. You will be very unhappy. So, avoid this option.

14.4 Using the VM

- There are **icons** provided for the terminal, VS Code, and the Firefox web browser. They are in the left-hand launcher bar.
- It can be helpful to set up a **shared folder** between the host and guest OS, so that you can easily copy files between them. With the VM shutdown (i.e., select "send the shutdown signal"), click Settings, then click Shared Folders. Click the little icon on the right that looks like a folder with a plus sign. In the dialog box for Folder Path, select Other, then navigate to the folder on your host OS that you want to share with the guest OS. Let's assume you created a new folder named `vmshared` inside your Documents folder, or wherever you like to keep files. The Folder Name in the dialog box will automatically be filled with `vmshared`. This is the name by which the guest OS will know the folder. You can change it if you like. Check Auto-mount; do not check Read-only. Make the Mount Point `/home/camel/vmshared`. Click OK, then click OK again. Start the VM again. You should now have a subdirectory named `vmshared` in your guest OS home directory that is shared between the host OS and the guest OS.
- You might be able to improve the **performance** of your VM by increasing the amount of memory or CPUs allocated to it, though it depends on how much your actual machine has available and what else you have running at the same time. With the VM shut down, try going in Virtual Box to Settings → System, and tinkering with the Base Memory slider on the Motherboard tab, and the Processors slider on the Processor tab. Then bring up the VM again and see how it does. You might have to play around to find a sweet spot. Later, after you are satisfied the VM is working properly hence you won't have to re-import it, you can safely delete the `.ova` file you downloaded to free up some space.