A
Problem sets

A.1 The prisoners' dilemma

I'm an apple farmer who hates apples but loves broccoli. You're a broccoli farmer who hates broccoli but loves apples. The obvious solution to this sad state of affairs is for us to trade – I ship you a box of my apples and you ship me a box of your broccoli. Win-win.

But I might try to get clever by shipping an empty box. Instead of cooperating, I "defect". I still get my broccoli from you (assuming you don't defect) and get to keep my apples. You, thinking through this scenario, realize that you're better off defecting as well; at least you'll get to keep your broccoli. But then, nobody gets what we want; we're both worse off. The best thing to do in this DONATION GAME seems to be to defect.

It's a bit of a mystery, then, why people cooperate at all. The answer may lie in the fact that we engage in many rounds of the game. If you get a reputation for cooperating, others may be willing to cooperate as well, leading to overall better outcomes for all involved.

The donation game is an instance of a classic game-theory thought experiment called the PRISONER'S DILEMMA. A prisoner's dilemma is a type of game involving two players in which each player is individually incentivized to choose a particular action, even though it may not result in the best global outcome for both players. The outcomes are commonly specified through a payoff matrix, such as the one in Table A.1.

To read the matrix, Player 1's actions are outlined at the left and Player 2's actions at the top. The entry in each box corresponds to a payoff to each player, depending on their respective actions. For instance, the top-right box indicates the payoff when Player 1 cooperates and Player 2 defects. Player 1 receives a payoff of −2 and Player 2 receives a payoff of 5 in that case.

To see why a dilemma arises, consider the possible actions taken

		Player 2	
		Cooperate	*Defect*
Player 1	*Cooperate*	(3, 3)	(−2, 5)
	Defect	(5, −2)	(0, 0)

Table A.1: Example payoff matrix for a prisoner's dilemma. This particular payoff matrix corresponds to a donation game in which providing the donation (of apples or broccoli, say) costs 2 unit and receiving the donation provides a benefit of 5 units.

by Player 1. If Player 2 cooperates, then Player 1 should defect rather than cooperating, since the payoff from defecting is higher ($5 > 3$). If Player 2 defects, then Player 1 should again defect since the payoff from defecting is higher ($0 > -2$). The same analysis applies to Player 2. Therefore, both players are incentivized to defect. However, the payoff from both players defecting (each getting 0) is objectively worse for both players than the payoff from both players cooperating (each getting 3).

An ITERATED PRISONER'S DILEMMA is a multi-round prisoner's dilemma, where the number of rounds is not known.[1] A STRATEGY specifies what action to take based on a history of past rounds of a game. We can (and will) represent a history as a list of pairs of actions (cooperate or defect) taken in the past, and a strategy as a function from histories to actions.

[1] If the number of rounds is known by the players ahead of time, players are again incentivized to defect for all rounds. We will not delve into the reasoning here, as that is outside the scope of this course, but it is an interesting result!

For example, a simple strategy is to ignore the histories and always defect. We call that the "nasty" strategy. More optimistic is the "patsy" strategy, which always cooperates.

Whereas the above analysis showed both players are incentivized to defect in a single-round prisoner's dilemma (leading to the nasty strategy), that is no longer necessarily the case if there are multiple rounds. Instead, more complicated strategies can emerge as players can take into account the history of their opponent's plays and their own. A particularly effective strategy – effective because it leads to cooperation, with its larger payoffs – is TIT-FOR-TAT. In the tit-for-tat strategy, the player starts off by cooperating in the first round, and then in later rounds chooses the action that the other player just played, rewarding the other player's cooperation by cooperating and punishing the other player's defection by defecting.

In this problem set, you'll complete a simulation of the iterated prisoner's dilemma that allows for testing different payoff matrices and strategies.

A.2 Higher-order functional programming

This assignment focuses on programming in the higher-order functional programming paradigm, with special attention to the idiomatic use of higher-order functions like `map`, `fold`, and `filter`. In doing so, you will exercise important features of functional languages, such as recursion, pattern matching, and list processing.

A.3 Bignums and RSA encryption

Cryptography is the science of methods for storing or transmitting messages securely and privately.

Cryptographic systems typically use *keys* for encryption and decryption. An encryption key is used to convert the original message (the plaintext) to coded form (the ciphertext). A corresponding decryption key is used to convert the ciphertext back to the original plaintext.

In traditional cryptographic systems, the same key is used for both encryption and decryption, which must be kept secret. Two parties can exchange coded messages only if they share the secret key. Since anyone who learned that key would be able to decode the messages, keys must be carefully guarded and transmitted only under tight security, for example, couriers handcuffed to locked, tamper-resistant briefcases!

Figure A.1: Whitfield Diffie (1944–) and Martin Hellman (1948–), co-inventors of public-key cryptography, for which they received the Turing Award in 2015.

In 1976, Diffie and Hellman initiated a new era in cryptography with their discovery of a new approach: public-key cryptography. In this approach, the encryption and decryption keys are different from each other. Knowing the encryption key cannot help you find the decryption key. Thus, you can publish your encryption key publicly – on the web, say – and anyone who wants to send you a secret message can use it to encode a message to send to you. You do not have to worry about key security at all, for even if everyone in the world knew your encryption key, no one could decrypt messages sent to you without knowing your decryption key, which you keep private to yourself. You used public-key encryption when you set up your CS51 git repositories: the command `ssh-keygen` generated a public encryption key and private decryption key for you. You uploaded the public key and (hopefully) kept the private key to yourself.

The best known public-key cryptosystem is due to computer scientists Rivest, Shamir, and Adelman, and is known by their initials, RSA. The security of your web browsing probably depends on RSA encryption. The system relies on the fact that there are fast algorithms for exponentiation and for testing prime numbers, but no known fast algorithms for factoring extremely large numbers. In this problem set you will complete an implementation of a version of the RSA system. (If you're interested in some of the mathematics behind RSA, see Section **??**. However, an understanding of that material is not needed to complete the problem set.)

Crucially, RSA requires manipulation of very large integers, much larger than can be stored, for instance, as an OCaml `int` value. OCaml's `int` type has a size of 63 bits, and therefore can represent integers between -2^{62} and $2^{62}-1$. These limits are available as OCaml constants

`min_int` and `max_int`:

```
# min_int, max_int ;;
- : int * int = (-4611686018427387904, 4611686018427387903)
```

The `int` type can then represent integers with up to 18 or so digits, that is, integers in the quintillions, but RSA needs integers with hundreds of digits.

Computer representations for arbitrary size integers are traditionally referred to as BIGNUMS. In this assignment, you will be implementing bignums, along with several operations on bignums, including addition and multiplication. We provide code that will use your bignum implementation to implement the RSA cryptosystem. Once you complete your bignum implementation, you'll be able to encrypt and decrypt messages using this public-key cryptosystem, and discover a hidden message that we've provided encoded in this way.

A.4 Symbolic differentiation

Solving an equation like $x^2 = x + 1$ NUMERICALLY yields a particular number as an approximation to the solution for x, for instance, 1.618. Solving the equation SYMBOLICALLY yields an expression representing the solution exactly, for instance, $\frac{1+\sqrt{5}}{2}$. (The golden ratio! See Exercise 8.) The earliest computing devices were used to calculate numerically. Charles Babbage envisioned his analytical engine as a device for calculating numeric tables, and Ada Lovelace's famous program for Babbage's analytical engine numerically calculated Bernoulli numbers.

But Lovelace (Figure A.2) was perhaps the first computer scientist to have the revolutionary idea that computers could be used for much more than numerical calculations.

Figure A.2: A rare daguerrotype of Ada Lovelace (Augusta Ada King, Countess of Lovelace, 1815–1852) by Antoine Claudet, taken c. 1843, around the time she was engaged in writing her notes on the Babbage analytical engine. (Menabrea and Lovelace, 1843)

> The operating mechanism... might act upon other things besides *number*, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent. (Menabrea and Lovelace, 1843, page 694)

One of the applications of the power of computers to transcend numerical calculation, which Lovelace immediately saw, was to engage in mathematics symbolically rather than numerically.

> It seems to us obvious, however, that where operations are so independent in their mode of acting, it must be easy by means of a few simple

> provisions and additions in arranging the mechanism, to bring out a *double* set of *results*, viz. – 1st, the *numerical magnitudes* which are the results of operations performed on *numerical data*. (These results are the *primary* object of the engine). 2ndly, the symbolical results to be attached to those numerical results, which symbolical results are not less the necessary and logical consequences of operations performed upon *symbolical data*, than are numerical results when the data are numerical. (Menabrea and Lovelace, 1843, page 694–5)

The first carrying out of symbolic mathematics by computer arose over a hundred years later, in the work of Turing-Award-winning computer scientist John McCarthy (Figure A.3). In the summer of 1958, McCarthy made a major contribution to the field of programming languages. With the objective of writing a program that performed symbolic differentiation (that is, the process of finding the derivative of a function) of algebraic expressions in an effective way, he noticed that some features that would have helped him to accomplish this task were absent in the programming languages of that time. This led him to the invention of the programming language LISP (McCarthy, 1960) and other ideas, such as the concept of list processing (from which LISP derives its name), recursion, and garbage collection, which are essential to modern programming languages.

McCarthy saw that the power of higher-order functional programming, together with the ability to manipulate structured data, make it possible to carry out such symbolic mathematics in an especially elegant manner. However, it was Jean Sammet (Figure A.4) who first envisioned a full system devoted to symbolic mathematics more generally. Her FORMAC system (Sammet, 1993) ushered in a wave of symbolic mathematics systems that have made good on Lovelace's original observation. Nowadays, symbolic differentiation of algebraic expressions is a task that can be conveniently accomplished on modern mathematical packages, such as Mathematica and Maple.

This assignment focuses on using abstract data types to design your own mini-language – a mathematical expression language over which you'll perform symbolic mathematics by computing derivatives symbolically.

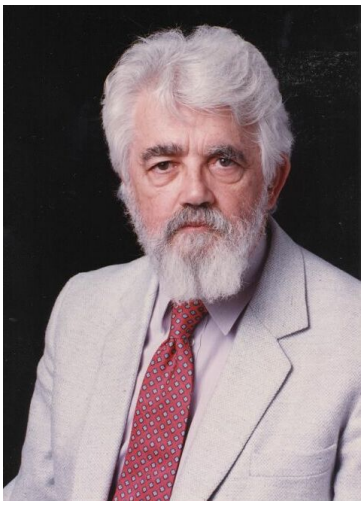


Figure A.3: John McCarthy (1927–2011), one of the founders of (and coiner of the term) artificial intelligence. His LISP programming language was widely influential in the history of programming languages. He was awarded the Turing Award in 1971.

Figure A.4: Jean Sammet (1928–2017), head of the FORMAC project to build "the first widely available programming language for symbolic mathematical computation to have significant practical usage" (Sammet, 1993). She was awarded the Augusta Ada Lovelace Award in 1999 and the Computer Pioneer Award in 2009 for her work on FORMAC and (with Admiral Grace Hopper) the programming language COBOL.

A.5 Ordered collections

In this assignment you will use modules to define several useful abstract data types (ADT). The particular ADTs that you'll be implementing are ordered collections (as implemented through binary search trees) and priority queues (as implemented through binary search trees and binary heaps).

An ordered collection is a collection of elements that have an in-

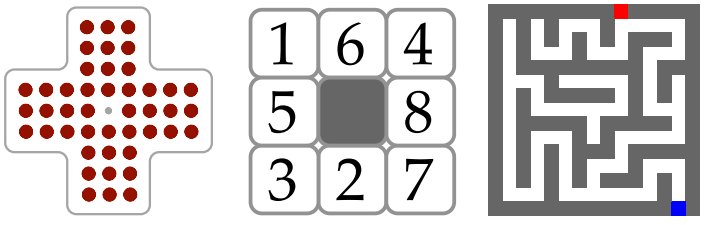
trinsic ordering to them. Natural operations on ordered collections include insertion of an element, deletion of an element, searching for an element, and access to the minimum and maximum elements. Priority queues constitute a special case of ordered collection in which the only operations are insertion of an element and extraction of the minimum element.

A.6 The search for intelligent solutions

In this assignment, you will apply your knowledge of OCaml modules and functors to complete the implementation of a program for solving search problems, a core problem in the field of artificial intelligence. In the course of working on this assignment, you'll implement a more efficient *queue module* using two stacks; create a *higher-order functor* that abstracts away details of search algorithms and puzzle implementations; and compare, visualize, and analyze the *performance* of various search algorithms on different puzzles.

A.6.1 Search problems

The field of ARTIFICIAL INTELLIGENCE pursues the computational emulation of behaviors that in humans are indicative of intelligence. A hallmark of intelligent behavior is the ability to figure out how to achieve some desired goal. Let's consider an idealized version of this behavior – puzzle solving. A puzzle can be in any of a variety of STATES. The puzzle starts in a specially designated INITIAL STATE, and we desire to reach a GOAL STATE by finding a sequence of MOVES that, when executed starting in the initial state, reach the goal state. Figure A.5 provides some examples of this sort of puzzle – peg solitaire, the 8-puzzle, and a maze puzzle.



(a) (b) (c)

Figure A.5: Some puzzles based on search for a goal state. (a) the peg solitaire puzzle; (b) the sliding-tile 8 puzzle; (c) a maze puzzle.

A good example is the 8 puzzle, depicted in Figure A.6. (You may know it better as the 15 puzzle, its larger 4 by 4 version.) A 3 by 3 grid of numbered tiles, with one tile missing, allows sliding of a tile adjacent to the empty space. The goal state is to be reached by repeated moves of this sort. But which moves should you make?

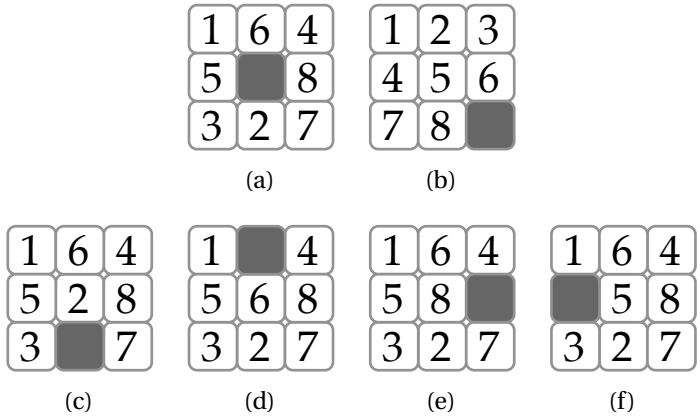


Figure A.6: The 8 puzzle: (a) an initial state, (b) the goal state, (c-f) the states resulting from moving up, down, left, and right from the initial state, respectively.

Solving goal-directed problems of this sort requires a SEARCH among all the possible move sequences for one that achieves the goal. You can think of this search process as a walk of a SEARCH TREE, where the nodes in the tree are the possible states of the puzzle and the directed edges correspond to moves that change the state from one to another. Figure A.7 depicts a small piece of the tree corresponding to the 8 puzzle.

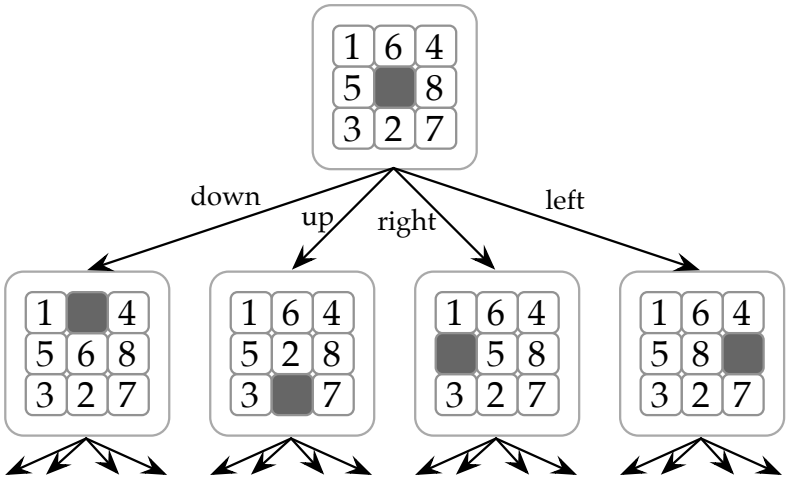


Figure A.7: A snippet from the search tree for the 8 puzzle.

To solve a puzzle of this sort, you maintain a collection of states to be searched, which we will call the *pending* collection. The pending collection is initialized with just the initial state. You can then take a state from the pending collection and test it to see if it is a goal state. If so, the puzzle has been solved. But if not, this state's NEIGHBOR states – states that are reachable in one move from the current state – are added to the pending collection (or at least those that have not been visited before) and the search continues.

To avoid adding states that have already been visited before, you'll

need to keep track of a set of states that have already been visited, which we'll call the *visited* set, so you don't revisit one that has already been visited. For instance, in the 8 puzzle, after a down move, you don't want to then perform an up move, which would just take you back to where you started. (The standard OCaml `Set` library will be useful here to keep track of the set of visited states.)

Of course, much of the effectiveness of this process depends on the order in which states are taken from the collection of pending states as the search proceeds. If the states taken from the collection are those most recently added to the collection (last-in, first-out, that is, as a stack), the tree is being explored in a DEPTH-FIRST manner. If the states taken from the collection are those least recently added (first-in, first-out, as a queue), the exploration is BREADTH-FIRST. Other orders are possible, for instance, the states might be taken from the collection in order of how closely they match the goal state (using some metric of closeness). This regime corresponds to BEST-FIRST or GREEDY SEARCH.

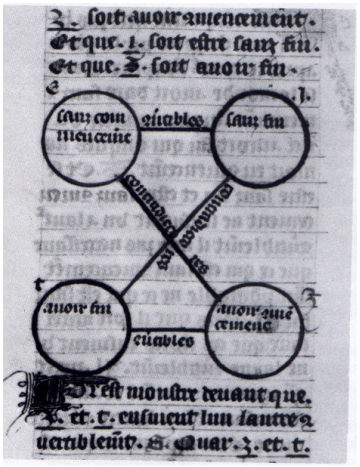
A.7 Refs, streams, and music

In this problem set you will work with two new ideas: First, we provide a bit of practice with imperative programming, emphasizing mutable data structures and the interaction between assignment and lexical scoping. Since this style of programming is probably most familiar to you, this portion of the problem set is brief. Second, we introduce lazy programming and its use in modeling infinite data structures. This part of the problem set is more extensive, and culminates in a project to generate infinite streams of music.

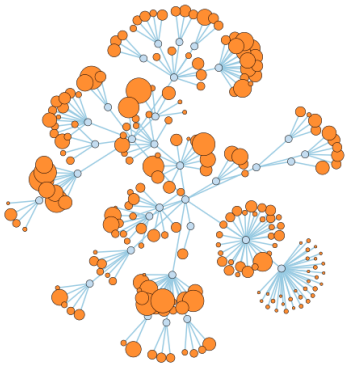
A.8 Force-directed graph drawing

You'll be familiar with graph drawings, those renderings of nodes and edges between them that depict all kinds of networks – both physical and virtual. These drawings are ubiquitous, in large part because of their fabulous utility. Examples date from as early as the Middle Ages (see Figure A.8(a)), when they were used to depict family trees and categorizations of vices and virtues. These days, they are used to depict everything from molecular interactions to social networks.

To gain the best benefit from visualizing graphs through a graph drawing, the nodes and edges must be laid out well. In this problem set, you'll complete the implementation of a system for *force-directed graph layout*. A modern example of what can be done with force-directed graph drawing is provided in Figure A.8(b). If you'd like to get



(a)



(b)

Figure A.8: Two sample graph drawings several hundred years apart. (a) A graph drawing from the 14th century with nodes depicting logical propositions in an argument and edges depicting relations among them. From Kruja et al. (2001). (b) Snapshot of a dynamic interactive force-directed graph drawing built using D3 (https://mbostock.github.io/d3/talk/20111116/force-collapsible.html), from the D3 gallery.

a sense of what can be done with force-directed graph drawing, you can play around with the graph visualization from which this snapshot came. In carrying out this project, you'll be making use of the object-oriented programming paradigm supported by OCaml.

A note of assuagement: Although this problem set document uses a lot of physics terminology, you really don't need to know any physics whatsoever to do the problem set. All of the physics-related code is in portions of the code-base (`graphdraw.ml` and `controls.ml`) that we have provided for you and that you won't need to modify.

A.8.1 Background

A GRAPH is a mathematical object defined as a set of NODES and EDGES connecting the nodes. As an example, consider a set of four nodes (numbered 0 to 3) connected with edges cyclically, 0 to 1, 1 to 2, 2 to 3, and 3 to 0, plus an extra edge from 0 to 2. A GRAPH DRAWING is a depiction of a graph in two (or sometimes three) dimensions indicating the nodes in the graph by graphical symbols of various sorts (circles, squares, and the like) and edges by lines drawn between the nodes. Other aspects of the graph are also typically manifested in graphical properties. For instance, groups of nodes might be aligned horizontally or vertically, or grouped with a zone box surrounding them, or laid out symmetrically or in a hub-and-spoke motif.

For the example four-node graph just presented, if we depict the nodes as small circles, placed more or less randomly on a drawing "canvas", we might get a graph drawing like Figure A.9(a). It's not particularly visually pleasing.

Much more attractive layouts can be generated by thinking of the positions at which the nodes are to be placed as physical MASSES sub-

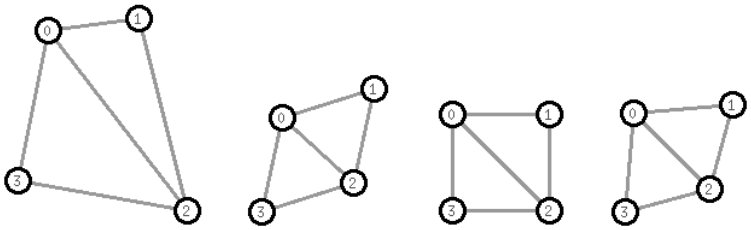
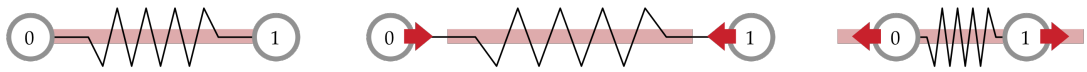


Figure A.9: Four different drawings of the same graph. (a) Nodes randomly placed. (b) With fixed length spring constraints between nodes connected by edges. (c) With fixed length spring constraints between nodes connected by outside edges, plus a horizontal alignment constraint on nodes 0 and 1 and a vertical alignment constraint on nodes 0 and 3. (d) An overconstrained layout with the constraints from (c) but with all of the edge constraints from (b), including the fixed length constraint between 0 and 2.

ject to various kinds of FORCES. The forces encourage the satisfying of graphical constraints, such as nodes being a particular distance from each other, or far away from each other, or horizontally or vertically aligned. For instance, if we imagine a spring with a certain REST LENGTH connecting two masses, those masses will have forces pushing them towards each other if they are farther apart than the rest length or away from each other if they are closer together than the rest length. (See Figure A.10 for a visual depiction.) According to Hooke's law, the force applied is directly proportional to the difference between the current distance and the rest length.

We can use this kind of mass-spring physical system to help with graph layout. We imagine that there is a mass for each node initially placed at the locations shown in Figure A.9(a), and for each edge in the graph there is a Hooke's law spring of a given rest length, 80 pixels, say, connecting the masses representing the nodes at the end of the edge. We refer to a force-generating element like the Hooke's law spring as a CONTROL. If we physically simulate how the forces on the masses generated by the controls would work, eventually the masses will come to rest at locations different from where they started, and indeed, if we place the graph nodes at those locations, we get exactly the layout in Figure A.9(b). Notice how all of the edge-connected nodes are the same length apart from each other – as it turns out, 80 pixels apart.

This methodology for graph layout is called FORCE-DIRECTED GRAPH LAYOUT based on its use of simulated forces to move the nodes and edges around. The method can be generalized to much more expressive graphical constraints than just establishing fixed distances between nodes with Hooke's-law springs. For instance, we can have force-generating controls that push masses to be in horizontal alignment, or vertical alignment. Using these controls, we can generate layouts like the one in Figure A.9(c). Care must be taken however. If we add too many controls in ways that overconstrain the physical system, the result of finding the resting positions may not fully satisfy any of the constraints, leading to unattractive layouts as in Figure A.9(d).



(a) (b) (c)

Figure A.10: A Hooke's law spring connecting two masses (labeled 0 and 1) and its generated forces. The pale red bar indicates the spring's rest length. (a) The spring at rest. No forces on the masses. (b) When the spring is stretched (the masses are farther apart than the spring's rest length), forces (red arrows) are applied to the two masses pushing them towards each other. (c) Conversely, when the spring is compressed (the masses are closer together than the spring's rest length), forces are applied to the two masses pushing them away from each other.

A.9 Simulating an infectious process

Imagine an infection among a population of people where the agent is transmitted from infected people to susceptible people nearby. The time course of such a process depends on many factors: How infectious is the agent? How much mixing is there of the population? How nearby must people get to be subject to infection? How long does recovery take? Is immunity conferred?

To get a sense of how such factors affect the overall course of the infection, we can simulate the process, with configurable parameters to control these and other aspects of the simulation.

A.9.1 The simulation

In this simulation, a population of people can be in one of several states:

- Susceptible – The person has not been infected or has been infected but is no longer immune.
- Infected – The person is infected and is therefore infectious and can pass the infection on to susceptibles nearby.
- Recovered – The person was infected but recovered and has immunity from further infection for a period of time.
- Deceased – The person was infected but did not recover.

(In the field of epidemiology, this kind of simulation is known as an SIRD model for obvious reasons.)

The simulation proceeds through a series of time steps. At each time step members of the population move on a two-dimensional grid to nearby squares. (How far they move – how many squares in each direction – is a configurable parameter.) Each person's status updates after they've moved. A susceptible person in the vicinity of infecteds may become infected. (This depends on how large a vicinity is considered to be "nearby" and how infectious each of the people in that vicinity are.) An infected person after a certain number of time steps may recover or die. (The relative proportion depends on a mortality parameter.) A recovered person after a certain number of time steps may lose immunity, becoming susceptible again.

B
Mathematical background and notations

In this book, we make free use of a wide variety of mathematical concepts and associated notations, some of which may be unfamiliar to readers. Facility with learning and using notation is an important skill to develop. In this chapter, we describe some of the notations we use, both for reference and to help build this facility.

B.1 Functions

Mathematics is full of functions, and of notations for defining them. In this section we present a menagerie of function-related notations.

B.1.1 Defining functions with equations

A standard technique is to define functions using a set of equations. Each of the equations provides a part of the definition based on a particular subset of the possible argument values of the function. For instance, consider the factorial function, which we'll denote with the symbol "*fact*". It is defined by these two equations:

$$fact(0) = 1$$
$$fact(n) = n \cdot fact(n-1) \quad \text{for } n > 0$$

Sometimes the cases are depicted overtly using a large brace:

$$fact(n) = \begin{cases} 1 & \text{for } n = 0 \\ n \cdot fact(n-1) & \text{for } n > 0 \end{cases}$$

A 'for' or 'where' clause after an equation provides further constraint on the applicability of that equation. In the case at hand, the second equation applies only when the argument n is greater than 0. In equational definitions, each equation must apply disjointly. If there were two equations that applied to a particular input, it would be unclear which of the two to use. These further constraints can guarantee disjointness and remove ambiguity.

B.1.2 Notating function application

In the factorial example, we used the familiar mathematical notation for applying a function to an argument – naming the function followed by its argument in parentheses: $fact(n)$.

> If we call this profit function P, we can use **arrow notation** and write the rule
> $$P\colon n \longrightarrow 5n - 500,$$
> which is read ''the function P that assigns $5n - 500$ to n'' or ''the function P that pairs n with $5n - 500$.'' We could also use **functional notation:**
> $$P(n) = 5n - 500$$
> which is read ''P of n equals $5n - 500$'' or ''the value of P at n is $5n - 500$.''

Figure B.1: A snippet from a typical middle school algebra textbook (Brown et al., 2000, page 379), introducing standard mathematical function application notation.

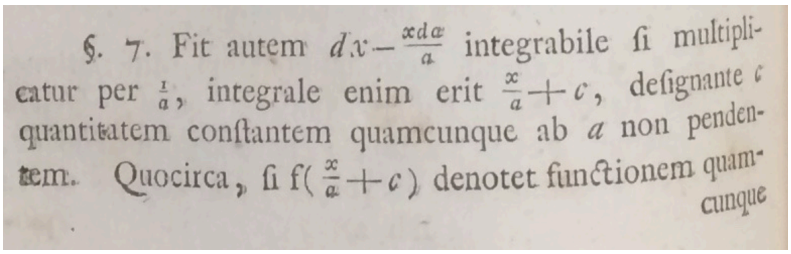
Some time in your primary education, perhaps in middle school, you were taught this standard mathematical notation for applying a function to one or more arguments. In Figure B.1, a snapshot from a middle school algebra textbook shows where this notation is first taught: "We could also use functional notation: $P(n) = 5n - 500$, which is read 'P of n equals $5n - 500$.'" In this notation, functions can take one or more arguments, notated by placing the arguments in parentheses and separated by commas following the function name. This notation is so familiar that it's hard to imagine that someone had to invent it. But someone did. In fact, it was the 18th century Swiss mathematician Leonhard Euler (Figure B.2) who in 1734 first used this notation (Figure B.3). Since then, it has become universal. At this point, the notation is so familiar that it is impossible to see $f(1,2,3)$ without immediately interpreting it as the application of the function f to arguments 1, 2, and 3.

Figure B.2: Leonhard Euler (1707–1783) invented the familiar parenthesized notation for function application.

It is thus perhaps surprising that OCaml doesn't use this notation for function application. Instead, it follows the notational convention proposed by the Princeton mathematician and logician Alonzo Church in his so-called lambda calculus (Section B.1.4), a logic of functions. In the lambda calculus, functions and their application are so central (indeed, there's basically nothing else in the logic) that the addition of the parentheses in the function application notation is too onerous. Instead, Church proposed merely prefixing the function to its argument. Instead of $f(1)$, Church's notation would have $f\,1$. Instead of $f(g(1))$, $f\,(g\,1)$.

B.1.3 Alternative mathematical notations for functions and their application

Despite the ubiquity of Euler's notation, mathematicians use a variety of different notations for functions and their application.



§. 7. Fit autem $dx - \frac{xda}{a}$ integrabile ſi multiplicatur per $\frac{1}{a}$, integrale enim erit $\frac{x}{a} + c$, deſignante c quantitatem conſtantem quamcunque ab a non pendentem. Quocirca, ſi $f(\frac{x}{a} + c)$ denotet functionem quamcunque

Figure B.3: The first known instance of the now standard function application notation, in a 1734 paper by Leonhard Euler. Note the $f\left(\frac{x}{a} + c\right)$. The function is even named f!

Certainly, mathematics uses different conventions for denoting operations than any given programming language. In the second *fact* equation, for instance, a center dot · is used for multiplication instead of the $*$ more common in programming languages. In other cases, simple juxtaposition is used for multiplication, as in $3x^2$ where the juxtaposition of the 3 and the x^2 indicates that they are to be multiplied. The details of these notations are often left unspecified in mathematical writing, reflecting the reality that mathematics is written to be read by *people*, people with sufficient common knowledge with the author to know the background assumptions or to figure them out from context. We don't have such a privilege with computers, so notations are typically more carefully explicated in programming language documentation.

The kind of thing that the argument must be (what computer scientists would call its "type") is often left implicit in mathematical notation. In the factorial example, we didn't state explicitly that the argument of factorial must be a nonnegative integer, yet the definition is only appropriate for that case. Negative integers are not provided a well-founded definition for instance, nor are noninteger numbers. Again, the omission of these requirements is based on an assumption of shared context with the reader. So as not to have to make that assumption, computer programs that implement function definitions make use of type constraints (whether explicit or inferred) or invariant assertions or (as a last resort) documentation to capture these assumptions.

The entire set of equations defines a single function, so that in converting definitions of this sort to code, they will typically end up in a single function definition. The individual equations correspond to different cases, which will likely be manifest by conditionals or case statements (such as OCaml `match` expressions).

Of course, the more standard notation for the factorial function is a

$$(f(x)+g(x))' = f'(x)+g'(x)$$
$$(f(x)-g(x))' = f'(x)-g'(x)$$
$$(f(x)\cdot g(x))' = f'(x)\cdot g(x)+f(x)\cdot g'(x)$$
$$\left(\frac{f(x)}{g(x)}\right)' = \frac{(f'(x)\cdot g(x)-f(x)\cdot g'(x))}{g(x)^2}$$
$$(\sin f(x))' = f'(x)\cdot\cos f(x)$$
$$(\cos f(x))' = f'(x)\cdot\sim\sin f(x)$$
$$(\ln f(x))' = \frac{f'(x)}{f(x)}$$
$$(f(x)^h)' = h\cdot f'(x)\cdot f(x)^{h-1}$$
where h contains no variables
$$(f(x)^{g(x)})' = f(x)^{g(x)}\cdot\left(g'(x)\cdot\ln f(x)+\frac{f'(x)\cdot g(x)}{f(x)}\right)$$
$$(n)' = 0 \qquad \text{where } n \text{ is any constant}$$
$$(x)' = 1$$

Figure B.4: Rules for taking derivatives for a variety of expression types. (Reproduced from Figure **??**.)

postfix exclamation mark (!):

$$0! = 1$$
$$n! = n\cdot(n-1)! \quad \text{for } n>0$$

The point is that the Euler notation is not the only one that can be or is used for function application. Here are some more examples:

- Frequently, superscripts are used to denote function application, for instance, as in Figure **??** (reproduced here as Figure B.4), where a superscript prime symbol specifies the derivative function.
- Newton's notation for derivatives, for example, $\frac{d}{dx}x^3$, provides yet another example of a nonstandard notation for a function application. Here, the function being applied is again the derivative function, this time as depicted by the compound notation $\frac{d}{dx}$, its argument the expression x^3.[1]
- In Chapter 13, a specific notation is used to express the substitution function, a function over a variable (x) and two expressions (P and Q) that returns the expression P with all free occurrences of x replaced by Q. That function is not notated by the Euler notation

[1] For the notation cognoscenti, what's really going on in this notation is that the $\frac{d}{dx}$ is both a binding construct, binding the x as the argument to an anonymous function that is (in Church's lambda calculus notation) $\lambda x.x^3$ and a function application of the derivative function.

(say, $subst(x, P, Q)$) but rather with a special notation employing brackets and arrows $P[x \mapsto Q]$. Nonetheless, it's still just a function applied to some arguments.

The notational profligacy of mathematics – especially having many different notations for functions – hides a lot of commonality shared among mathematical processes. Don't be confused; despite all the notations, they're all just functions.

B.1.4 The lambda notation for functions

Part of the notation for defining functions equationally involves giving them a name. For instance, the ABSOLUTE VALUE function can be defined equationally as

$$abs(n) = \sqrt{n^2}$$

One of the contributions of Church's lambda calculus is a notation for defining functions directly, without bestowing a name. In fact, the expression on the right hand side of the equation, $\sqrt{n^2}$, almost serves this purpose already, by specifying the function from n to $\sqrt{n^2}$. There are two problems in using bare expressions like $\sqrt{n^2}$ to specify functions. First, how is the reader to know that the expression is intended to specify a *function* rather than a *number*? That is, how are we to realize that the use of n is meant generically, and not as standing for some particular number? Second, if the expression makes use of multiple variables, how is the reader supposed to determine which variable represents the *input* to the function? In the case of $\sqrt{n^2}$, there is only one option, since the expression makes use of only one variable. But for other expressions, like $m \cdot n^2$, it is unclear if the input is intended to be m or n.

Church introduced his lambda notation to solve these problems. He prefixes the expression with a Greek lambda (λ), followed by the variable that is serving as the input to the function, followed by a period. Table B.1 provides some examples.

The lambda notation for specifying anonymous functions will be familiar to OCaml programmers; it appears in OCaml as well, though under a different concrete syntax. The keyword `fun` plays the role of λ and the operator `->` plays the role of the period. In fact, the ability to define anonymous functions, so central to functional programming languages, is inherited directly from the lambda notation that gives its name to Church's calculus.

As shown in Table B.1, each of the examples above could be rephrased in OCaml. You may recognize the last of these as an example of a curried function (Section 6.2).

$\lambda n.\sqrt{n^2}$	The function from n to $\sqrt{n^2}$, that is, the absolute value function, or, in OCaml: `fun n -> sqrt(n *. n)`
$\lambda n.(m \cdot n^2)$	the function from n to $m \cdot n^2$, so that m is implicitly being viewed as a constant: `fun n -> m *. (n *. n)`
$\lambda m.(m \cdot n^2)$	the function from m to $m \cdot n^2$, so that n is implicitly being viewed as a constant: `fun m -> m *. (n *. n)`
$\lambda m.\lambda n.(m \cdot n^2)$	the function from m to a function from n to $m \cdot n^2$: `fun m -> fun n -> m *. (n *. n)`

Table B.1: A few functions in lambda notation, with their English glosses and their approximate OCaml equivalents.

When there's a need for specifying mathematical functions directly, unnamed, we will take advantage of Church's lambda notation, especially in Chapter 14.

B.2 Summation

In Section 14.5.2, we make use of the following identity for calculating the sum of all integers from 1 to n

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

which was graphically demonstrated to hold in Figure 14.6. Here we provide a more traditional algebraic proof.

Define the sum in question to be S:

$$S = \sum_{i=1}^{n} i$$

We can think of this sum as adding all the values from 1 to n, or conversely, all the numbers from n to 1, that is all the values of $(n - i + 1)$:

$$S = \sum_{i=1}^{n} (n - i + 1)$$

Adding these two together,

$$2S = \sum_{i=1}^{n} i + \sum_{i=1}^{n} (n - i + 1)$$

but the two sums can be brought together as a single sum and simplified:

$$\begin{aligned} 2S &= \sum_{i=1}^{n} (i + (n - i + 1)) \\ &= \sum_{i=1}^{n} (n + 1) \end{aligned}$$

Now we're just summing up n instances of $n+1$, that is, multiplying n and $n+1$:

$$2S = n \cdot (n+1)$$

so that

$$S = \frac{n \cdot (n+1)}{2}$$

For Gauss's problem, where n is 100, he presumably calculated $\frac{100 \cdot 101}{2} = 5050$.

B.3 Logic

The logic of propositions, boolean logic, underlies the `bool` type. Informally, propositions are conceptual objects that can be either true or false. Propositions can be combined or transformed with various operations. The CONJUNCTION of two propositions p and q is true just in case both p and q are true, and false otherwise. The DISJUNCTION is true just in case either p or q (or both) are true. The NEGATION of p is true just in case p is not true (that is, p is false). Conjunction, disjunction, and negation thus correspond roughly to the English words "and", "or", and "not", respectively, and for that reason, we sometimes speak of the "and" of two boolean values, or their "or". (See Figure B.5.)

There are other operations on boolean values considered in logic – for instance, the conditional, glossed by "if . . . then . . . "; or the exclusive "or" – but these three are sufficient for our purposes. For more background on propositional logic, see Chapter 9 of the text by Lewis and Zax (2019).

p	q	p and q	p or q	not p
`true`	`true`	`true`	`true`	`false`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`false`	`false`	`false`	`false`	`true`

Figure B.5: The three boolean operators defined.

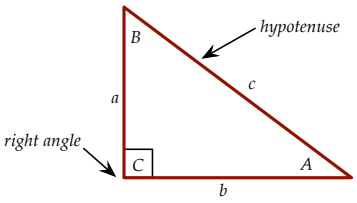


Figure B.6: A right triangle. Angle C is a right angle. The opposite side, of length c, is the hypotenuse. By Pythagorus's theorem, $a^2 + b^2 = c^2$.

B.4 Geometry

The SLOPE of a line between two points x_1, y_1 and x_2, y_2 is the ratio of their vertical difference and their horizontal difference, $\frac{y_2 - y_1}{x_2 - x_1}$. (See Figure B.7.)

A RIGHT TRIANGLE is a triangle one of whose edges is a right (90°) angle. (See Figure B.7.) The side opposite the right angle is called the HYPOTENUSE. PYTHAGORUS'S THEOREM holds that the sum of the squares of the adjacent sides' lengths is the square of the length of the hypotenuse.

Pythagorus's theorem can be used to determine the DISTANCE between two points specified with Cartesian (x-y) coordinates. As depicted in Figure B.7, by Pythagorus's theorem, we can square the differences in each dimension, sum the squares, and take the square root.

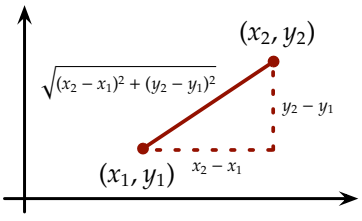


Figure B.7: Two points, given by a pair of their x (horizontal) and y (vertical) coordinates. The slope of the line between them is $\frac{y_2 - y_1}{x_2 - x_1}$. The distance between them, as per the Pythagorean theorem, is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

The ratio of the circumference of a circle and its diameter is (non-trivially, and perhaps surprisingly) a constant, conventionally called π (read, "pi"), and approximately 3.1416. This constant is also the ratio of the area of a circle to the area of a square whose side is the circle's radius. Thus, using the nomenclature of Figure B.8, $c = \pi d = 2\pi r$ and $A = \pi r^2$.

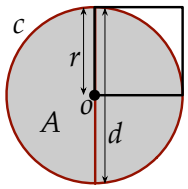


Figure B.8: Geometry of the circle at origin o of radius r, diameter $d = 2r$, circumference c, and area A.

The area of a rectangle is the product of its width w and height h, that is, $A = wh$. The area of a triangle (Figure B.9) is half the area of its circumscribing rectangle, that is, $\frac{1}{2}wh$. Alternatively, if we know the lengths of its three sides (a, b, and c), but not its width and height, we can use HERON'S FORMULA, which makes use of the SEMIPERIMETER s of the triangle, a length that is half of its perimeter: $s = \frac{1}{2}(a+b+c)$. The area is then

$$A = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$$

B.5 Sets

A set is a collection of distinct (physical or mathematical) objects.

An EXTENSIONAL set definition (given by an explicit list of its members) is notated by listing the elements in braces separated by commas, as, for instance, $\{1,2,3,4\}$. Obviously, this notation only works for finite sets, although infinite sets can be informally indicated with ellipses (as $\{1,2,3,\ldots\}$) in cases where the rule for filling in the remaining elements is sufficiently obvious to the reader.

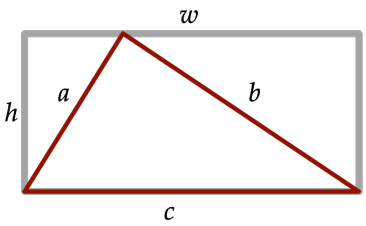


Figure B.9: A triangle and a circumscribing rectangle, with labeled edge lengths.

An INTENSIONAL set definition (given by describing all members of the set rather than listing them) is notated by placing in braces a schematic element of the set, followed by a vertical bar, followed by a description of the range of any variables in the schema. For instance, the set of all even numbers might be $\{x \mid x \bmod 2 = 0\}$, read "the set of all x such that x is evenly divisible by 2." Similarly, the set of all squares of prime numbers would be $\{x^2 \mid x \text{ is prime}\}$. (Note the combination of mathematical notation and natural language, a typical instance of "code switching" in mathematical writing.)

The EMPTY SET, notated $\varnothing$ or $\{\}$, is the set containing no members.

Certain standard operations on sets are notated with infix operators:

Union: $s \cup t$ is the UNION of sets s and t, that is, the set containing all the elements that are in either of the two sets;

Intersection: $s \cap t$ is the INTERSECTION, containing just the elements that are in both of the sets;

Difference: $s - t$ is the set DIFFERENCE, all elements in s except for those in t; and

Membership: $x \in s$ specifies MEMBERSHIP, stating that x is a member of the set s.

By way of example, the following are all true statements, expressed in this notation:

$$\{1,2,3\} \cup \{3,4\} = \{1,2,3,4\}$$
$$\{1,2,3\} \cap \{3,4\} = \{3\}$$
$$\{1,2,3\} - \{3,4\} = \{1,2\}$$
$$3 \in \{1,2,3\}$$
$$3 \notin \{2,4,6\}$$

Note the use of a slash through a symbol to indicate its NEGATION: $\notin$ for 'is not a member of'.

B.6 Equality and identity

There are different notions of IDENTITY used in mathematical notation. The = symbol typically connotes two values being the same "semantically". The ≡ symbol connotes a stronger notion of syntactic identity, so that $x \equiv y$ means that x and y are (that is, represent) the same syntactic entity (variable say) rather than that they have the same value (in whatever context that might be appropriate). For instance, consider these equations found in the definition of substitution:

$$x[x \mapsto P] = P$$
$$y[x \mapsto P] = y \qquad \text{where } x \not\equiv y$$

Recall that $P[x \mapsto Q]$ specifies the expression P with all free occurrences of x replaced by the expression Q (with care taken not to capture any free occurrences of x in Q). Here x and y are variables (metavariables) ranging over expressions that may themselves be (object-level) variables. The notation $x \not\equiv y$ indicates that the variable y that constitutes the expression being substituted into is a different variable from the variable x that is being substituted for.

C
A style guide

This guide provides some simple rules of good programming style, both general and OCaml-specific, developed for the Harvard course CS51. The rules presented here tend to follow from a small set of underlying principles.[1]

Consistency Similar decisions should be made within similar contexts.

Brevity "Everything should be made as simple as possible, but no simpler." (attr. Albert Einstein)

Clarity Code should be chosen so as to communicate clearly to the human reader.

Transparency Appearance should summarize and reflect structure.

Like all rules, those below are not to be followed slavishly. Rather, they should be seen as instances of these underlying principles. These principles may sometimes be in conflict, in which case judgement is required in finding the best way to write the code. This is one of the many ways in which programming is an art, not (just) a science.

This guide is not complete. For more recommendations, from the OCaml developers themselves, see the official OCaml guidelines.

[1] This style guide is reworked from a long line of style guides for courses at Princeton, University of Pennsylvania, and Cornell, including Cornell CS 312, U Penn CIS 500 and CIS 120, and Princeton COS 326. All this shows the great power of recursion. (Also, the joke about recursion was stolen from COS 326. (Also, the joke about the joke about recursion was stolen from Greg Morrisett. I think. See the Preface.))

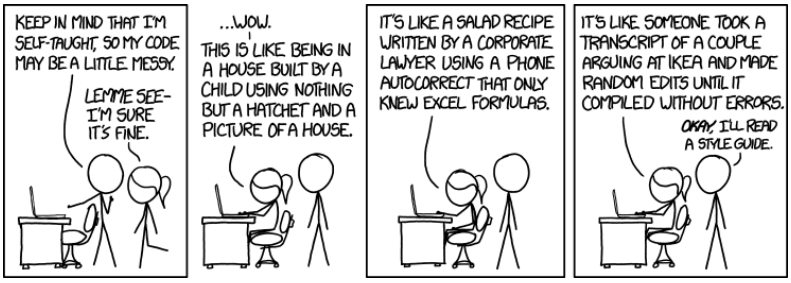


Figure C.1: Yes, coding style is important.

C.1 Formatting

Formatting concerns the layout of the text of a program on the screen or page, such issues as vertical alignments and indentation, line breaks, and whitespace. To allow for repeatable formatting, code is typically presented with a fixed-width font in which all characters including spaces take up the same horizontal pitch.

C.1.1 No tab characters

You may feel inclined to use tab characters (ASCII 0x09) to align text. Do not do so; use spaces instead. The width of a tab is not uniform across all renderings, and what looks good on your machine may look terrible on another's, especially if you have mixed spaces and tabs. Some text editors map the tab key to a sequence of spaces rather than a tab character; in this case, it's fine to use the tab key.

C.1.2 80 column limit

No line of code should extend beyond 80 characters long. Using more than 80 columns typically causes your code to wrap around to the next line, which is devastating to readability.

C.1.3 No needless blank lines

The obvious way to stay within the 80 character limit imposed by the rule above is to press the enter key every once in a while. However, blank lines should only be used at major logical breaks in a program, for instance, between value declarations, especially between function declarations. Often it is not necessary to have blank lines between other declarations unless you are separating the different types of declarations (such as modules, types, exceptions, and values). Unless function declarations within a `let` block are long, there should be no blank lines within a `let` block. There should absolutely never be a blank line within an expression.

C.1.4 Use parentheses sparely

Parentheses have many purposes in OCaml, including constructing tuples, specifying the unit value, grouping sequences of side-effect expressions, forcing higher precedence on an expression for parsing, and grouping structures for functor arguments. Clearly, parentheses must be used with care, as they force the reader to disambiguate the intended purpose of the parentheses, making code more difficult to

understand. You should therefore only use parentheses when necessary or when doing so improves readability.

✗
```
let x = function1 (arg1) (arg2) (function2 (arg3)) (arg4)
```

✓
```
let x = function1 arg1 arg2 (function2 arg3) arg4
```

On the other hand, it is often useful to add parentheses to help indentation algorithms, as in this example:

✗
```
let x = "Long line ..."
  ^ "Another long line..."
```

✓
```
let x = ("Long line ..."
         ^ "Another long line...")
```

Similarly, wrapping `match` expressions in parentheses helps avoid a common (and confusing) error that you get when you have a nested `match` expression. (See Section 10.3.2 for an example.)

Parentheses should never appear on a line by themselves, nor should they be the first visible character; parentheses do not serve the same purpose as brackets do in C or Java.

C.1.5 Delimiting code used for side effects

Imperative programs will often have sequences of expressions to be evaluated primarily for side effect rather than value. When delimiting the scope of such sequences, use `begin` ⟨⟩ `end` rather than parentheses, for instance,

✗
```
if condition then
  (do this;
   do that;
   do the other)
else
  (do something else entirely;
   do this too);
do in any case
```

✓
```
if condition then begin
  do this;
  do that;
  do the other
end else begin
  do something else entirely;
  do this too
end;
do in any case
```

C.1.6 Spacing for operators and delimiters

Infix operators (arithmetic operators like + and ∗, the typing operator `:`, type forming operators like ∗ and `->`, etc.) should be surrounded by spaces. Delimiters (like the list item delimiter `;` and the tuple element delimiter `,`) are followed but not preceded by a space.

✓
```
let f (x : int) : int * int = 3 * x - 1, 3 * x + 1 ;;
```

✗
```
let f (x: int): int*int = 3* x-1, 3* x+1 ;;
```

Judgement can be applied to vary from these rules for clarity's sake, for instance, when emphasizing precedence.

✓
```
let f (x : int) : int * int = 3*x - 1, 3*x + 1 ;;
```

When expressions with operators get overly long, it may be desirable to add line breaks. Such line breaks should tend to be placed just before, rather than just after, operators, so as to highlight the operator at the beginning of the next line.

✓
```
let price = base * (100 + tax_pct) / 100 ;;
```

✗
```
let price = base *
            (100 + tax_pct) /
            100 ;;
```

✓
```
let price = base
            * (100 + tax_pct)
            / 100 ;;
```

It's better to place breaks at operators higher in the abstract syntax tree, to emphasize the structure.

✗
```
let price = base * (100
            + tax_pct) / 100 ;;
```

In the case of delimiters, however, line breaks should occur after the delimiter.

✗
```
let r = { product = "Dynamite"
        ; company = "Acme"
        ; price = base * (100 + tax_pct) / 100} ;;
```

✓
```
let r = {product = "Dynamite";
         company = "Acme";
         price = base * (100 + tax_pct) / 100} ;;
```

Of course, keep in mind that understanding of the code might be enhanced by restructuring the code and naming partial results:

✓
```
let tax = base * tax_pct / 100 ;;
let price = base + tax ;;
```

C.1.7 Indentation

Indentation should be used to encode the block structure of the code as described in the following sections. It is typical to indent by two xor four spaces. Choose one system for indentation, and be consistent throughout your code.

Indenting `if` expressions Indent `if` expressions using one of the following methods, depending on the sizes of the expressions. For very short *then* and *else* branches, a single line may be sufficient.

✓
```
if exp1 then veryshortexp2 else veryshortexp3
```

When the branches are too long for a single line, move the else onto its own line.

✓
```
if exp1 then exp2
else exp3
```

This style lends itself nicely to nested conditionals.

✓
```
if exp1 then shortexp2
else if exp3 then shortexp4
else if exp5 then shortexp6
else exp8
```

For very long *then* or *else* branches, the branch expression can be indented and use multiple lines.

✓
```
if exp1 then
  longexp2
else shortexp3
```

✓
```
if exp1 then
  longexp2
else
  longexp3
```

Some use an alternative conditional layout, with the `then` and `else` keywords starting their own lines.

✗
```
if exp1
then exp2
else exp3
```

This approach is less attractive for nested conditionals and long branches, though for unnested cases it can be acceptable.

Indenting `let` expressions Indent the body of a `let` expression the same as the `let` keyword itself.

✓
```
let x = definition in
code_that_uses_x
```

This is an exception to the rule of further indenting subexpression blocks to manifest the nesting structure.

✗
```
let x = definition in
  code_that_uses_x
```

The intention is that `let` definitions be thought of like mathematical assumptions that are listed before their use, leading to the following attractive indentation for multiple definitions:

```
let x = x_definition in
let y = y_definition in
let z = z_definition in
block_that_uses_all_the_defined_notions
```

Indenting `match` expressions Indent `match` expressions so that the patterns are aligned with the `match` keyword, always including the initial (optional) `|`, as follows:

```
match expr with
| first_pattern -> ...
| second_pattern -> ...
```

Some disfavor aligning the arrows in a match, arguing that it makes the code harder to maintain. However, where there is strong parallelism among the patterns, this alignment (and others) can make the parallelism easier to see, and hence the code easier to understand. Use your judgement.

C.2 Documentation

C.2.1 Comments before code

Comments go above the code they reference. Consider the following:

✗
```
let sum = List.fold_left (+) 0
(* Sums a list of integers. *)
```

✓
```
(* Sums a list of integers. *)
let sum = List.fold_left (+) 0
```

The latter is the better style, although you may find some source code that uses the first. Comments should be indented to the level of the line of code that follows the comment.

C.2.2 Comment length should match abstraction level

Long comments, usually focused on overall structure and function for a program, tend to appear at the top of a file. In that type of comment, you should explain the overall design of the code and reference any sources that have more information about the algorithms or data structures. Comments can document the design and structure of a class at some length. For individual functions or methods, comments should state the invariants, the non-obvious, or any references that have more information about the code. Avoid comments that merely restate the code they reference or state the obvious. All other comments in the file should be as short as possible; after all, brevity is the soul of wit. Rarely should you need to comment within a function; expressive variable naming should be enough.

C.2.3 Multi-line commenting

There are several styles for demarcating multi-line comments in OCaml. Some use this style:

```
(* This is one of those rare but long comments
 * that need to span multiple lines because
 * the code is unusually complex and requires
 * extra explanation. *)
let complicated_function () = ...
```

arguing that the aligned asterisks demarcate the comment well when it is viewed without syntax highlighting. Others find this style heavy-handed and hard to maintain without good code editor support (for instance, emacs Tuareg mode doesn't support it well), leading to this alternative:

```
(* This is one of those rare but long comments
   that need to span multiple lines because
   the code is unusually complex and requires
   extra explanation.
 *)
let complicated_function () = ...
```

Whichever you use, be consistent.

C.3 Naming and declarations

C.3.1 Naming conventions

Table C.1 provides the naming convention rules that are followed by OCaml libraries. You should follow them too. Some of these naming conventions are enforced by the compiler; these are shown **in boldface** below. For example, it is not possible to have the name of a variable start with an uppercase letter.

Token	*Convention*	*Example*
Variables and functions	**Symbolic or initial lower case.** Use underscores for multiword names.	`get_item`
Constructors	**Initial upper case.** Use embedded caps for multiword names. Historical exceptions are `true` and `false`.	`Node`, `EmptyQueue`
Types	**All lower case.** Use underscores for multiword names.	`priority_queue`
Module Types	**Initial upper case.** Use embedded caps for multiword names, or (as we do here) use all uppercase with underscores.	`PriorityQueue` or `PRIORITY_QUEUE`
Modules	**Initial upper case.** Use embedded caps for multiword names.	`PriorityQueue`
Functors	**Initial upper case.** Use embedded caps for multiword names.	`PriorityQueue`

Table C.1: Naming conventions

C.3.2 Use meaningful names

Variable names should describe what the variables are for, in the form of a word or sequence of words. Proper naming of a variable can be the best form of documentation, obviating the need for any further documentation. By convention (Table C.1) the words in a variable name are separated by underscores (`multi_word_name`), not (ironically) distinguished by camel case (`multiWordName`).

✓
```
let local_date = Unix.localtime (Unix.time ()) ;;
let total_cost = quantity * price_each ;;
```

✗
```
let d = Unix.localtime (Unix.time ()) ;;
let c = n * at ;;
```

The length of a variable name is roughly correlated with how long a reader of the code will have to remember its use. In short `let` blocks, one letter variable names can sometimes be appropriate. The definition

```
fun the_optional_number -> the_optional_number <> None
```

is not better than

```
fun x -> x <> None
```

(Of course, this function can be specified even more compactly as `(<>) None`.)

Often it is the case that a function used in a `fold`, `filter`, or `map` is named `f`. Here is an example with appropriate variable names:

```
let local_date = Unix.localtime (Unix.time ()) in
let minutes = date.Unix.tm_min in
let seconds = date.Unix.tm_min in
let f n = (n mod 3) = 0 in
List.filter f [minutes; seconds]
```

Take advantage of the fact that OCaml allows the prime character ' in variable names. Use it to make clear related functions:

```
let reverse (lst : 'a list) =
  let rec reverse' remaining accum =
    match remaining with
    | [] -> accum
    | hd :: tl -> reverse' tl (hd :: accum) in
  reverse' lst [] ;;
```

C.3.3 Constants and magic numbers

MAGIC NUMBERS are explicit values sprinkled in code that are used without explanation, as `1.0625` in the following code:

✗
```
let total_cost = (quantity *. price_each) *. 1.0625 ;;
```

Magic numbers are inscrutable, a nightmare for readers of the code. Instead, give those constants an expressive name. If these defined constants are global, we use the naming convention of using a variable in all uppercase letters except for an initial lowercase 'c' (for "constant").

✓
```
let cTAX_RATE = .0625 ;;
  (* ... some time later ... *)
  let total_cost = (quantity *. price_each) *. (1. +. cTAX_RATE)
;;
```

Not only is this more explanatory – we understand that the final multiplication is to account for taxes – it allows for a single point of code change if the tax rate changes.

C.3.4 Function declarations and type annotations

Top-level functions and values should be declared with explicit type annotations to allow the compiler to verify the programmer's intentions. Use spaces around `:`, as with all operators.

✗
```
let succ x = x + 1
```

✓
```
let succ (x : int) : int = x + 1
```

When a function being declared has multiple arguments with complicated types, so that the declaration doesn't fit nicely on one line,

✗
```
let rec zip3 (x : 'a list) (y : 'b list) (z : 'c list) : ('a * 'b *
  'c) list option =
  ...
```

one of the following indentation conventions can be used:

✓
```
let rec zip3 (x : 'a list)
             (y : 'b list)
             (z : 'c list)
           : ('a * 'b * 'c) list option =
  ...
```

✓
```
let rec zip3
          (x : 'a list)
          (y : 'b list)
          (z : 'c list)
        : ('a * 'b * 'c) list option =
  ...
```

C.3.5 Avoid global mutable variables

Mutable values, on the rare occasion that they are necessary at all, should be local to functions and almost never declared as a structure's value. Making a mutable value global causes many problems. First, an algorithm that mutates the value cannot be ensured that the value is consistent with the algorithm, as it might be modified outside the function or by a previous execution of the algorithm. Second, having global mutable values makes it more likely that your code is nonreentrant. Without proper knowledge of the ramifications, declaring global mutable values can easily lead not only to bad design but also to incorrect code.

C.3.6 When to rename variables

You should rarely need to rename values: in fact, this is a sure way to obfuscate code. Renaming a value should be backed up with a very good reason. One instance where renaming a variable is both common and reasonable is aliasing modules. In these cases, other modules used by functions within the current module are aliased to one or two letter variables at the top of the `struct` block. This serves two purposes: it shortens the name of the module and it documents the modules you use. Here is an example:

```
module H = Hashtbl
module L = List
module A = Array
...
```

C.3.7 Order of declarations in a module

When declaring elements in a file (or nested module) you first alias the modules you intend to use, then declare the types, then define exceptions, and finally list all the value declarations for the module.

Separating each of these sections with a blank line is good practice unless the whole is quite short. Here is an example:

```
module L = List
type foo = int
exception InternalError
let first list = L.nth list 0
```

Every declaration within the module should be indented the same amount.

C.4 Pattern matching

C.4.1 No incomplete pattern matches

Incomplete pattern matches are flagged with compiler warnings, and you should avoid them. In fact, it's best if your code generates no warnings at all. Even if you "know" that a certain match case can never occur, it's better to record that knowledge by adding the match case with an action that raises an appropriate error.

C.4.2 Pattern match in the function arguments when possible

Tuples, records, and algebraic datatypes can be deconstructed using pattern matching. If you simply deconstruct a function argument before you do anything else substantive, it is better to pattern match in the function argument itself. Consider these examples:

✗
```
let f arg1 arg2 =
  let x = fst arg1 in
  let y = snd arg1 in
  let z = fst arg2 in
  ...
```

✓
```
let f (x, y) (z, _) =
  ...
```

✗
```
let f arg1 =
  let x = arg1.foo in
  let y = arg1.bar in
  let baz = arg1.baz in
  ...
```

✓
```
let f {foo = x; bar = y; baz} =
  ...
```

See also the discussion of extraneous match expressions in `let` definitions in Section C.4.4.

C.4.3 *Pattern match with as few* `match` *expressions as necessary*

Rather than nesting `match` expressions, you can sometimes combine them by pattern matching against a tuple. Of course, this doesn't work if one of the nested `match` expressions matches against a value obtained from a branch in another `match` expression. Nevertheless, if all the values are independent of each other you should combine the values in a tuple and match against that. Here is an example:

✗
```
let d = Date.fromTimeLocal (Unix.time ()) in
match Date.month d with
| Date.Jan -> (match Date.day d with
                | 1 -> print "Happy New Year"
                | _ -> ())
| Date.Mar -> (match Date.day d with
                | 14 -> print "Happy Pi Day"
                | _ -> ())
| Date.Oct -> (match Date.day d with
                | 10 -> print "Happy Metric Day"
                | _ -> ())
```

✓
```
let d = Date.fromTimeLocal (Unix.time ()) in
match Date.month d, Date.day d with
| Date.Jan,  1 -> print "Happy New Year"
| Date.Mar, 14 -> print "Happy Pi Day"
| Date.Oct, 10 -> print "Happy Metric Day"
| _ -> ()
```

(This example also provides a case where aligning arrows improves clarity by emulating a table.)

C.4.4 *Misusing* `match` *expressions*

The `match` expression is misused in two common situations. First, `match` should never be used with single atomic values in place of an `if` expression. (That's why `if` exists.) For instance,

✗
```
match e with
| true -> x
| false -> y
```

✓
```
if e then x else y
```

and

✗
```
match e with
| c -> x (* c is a constant value *)
| _ -> y
```

✓
```
if e = c then x else y
```

(Using a `match` to match against *several* atomic values may, however, be preferable to nested conditionals.)

Second, a separate `match` expression should not be used when an enclosing expression (like a `let`, `fun`, `function`) allows pattern-matching itself:

✗
```
let x = match expr with
        | y, z -> y in
...
```

✓
```
let x, _ = expr in
...
```

C.4.5 Avoid using too many projection functions

Frequently projecting a value from a record or tuple causes your code to become unreadable. This is especially a problem with tuple projection because the value is not documented by a mnemonic name. To prevent projections, you should use pattern matching with a function argument or a value declaration. Of course, using projections is okay as long as use is infrequent and the meaning is clearly understood from the context.

✗
```
let v = some_function () in
let x = fst v in
let y = snd v in
x + y
```

✓
```
let x, y = some_function () in
x + y
```

Don't use `List.hd` or `List.tl` at all The functions `hd` and `tl` are used to deconstruct list types; however, they raise exceptions on certain arguments. You should never use these functions. In the case that you find it absolutely necessary to use these (something that probably won't ever happen), you should explicitly handle any exceptions that can be raised by these functions.

C.5 Verbosity

C.5.1 Reuse code where possible

The OCaml standard library has a great number of functions and data structures. Unless told otherwise, use them! Become familiar with the contents of the `Stdlib` module. Often students will recode `List.filter`, `List.map`, and similar functions. A more subtle situation for recoding is all the fold functions. Functions that recursively walk down lists should make vigorous use of `List.fold_left` or

`List.fold_right`. Other data structures often have a fold function; use them whenever they are available. (In some exercises, we will ask you to implement some constructs yourself rather than relying on a library function. In such cases, we'll specify that using library functions is not allowed.)

C.5.2 Do not abuse `if` expressions

Remember that the type of the condition in an `if` expression is `bool`. There is no reason to compare boolean values against boolean literals.

✗
```
if e = true then x else y
```

✓
```
if e then x else y
```

In general, the type of an `if` expression can be any `'a`, but in the case that the type is `bool`, you should probably not be using `if` at all. Consider the following:

✗	✓
`if e then true else false`	`e`
`if e then false else true`	`not e`
`if e then e else false`	`e`
`if x then true else y`	`x || y`
`if x then y else false`	`x && y`
`if x then false else y`	`not x && y`

Also problematic is overly complex conditions such as extraneous negation.

✗
```
if not e then x else y
```

✓
```
if e then y else x
```

The exception here is if the expression `y` is very long and complex, in which case it may be more readable to have it placed at the end of the `if` expression.

C.5.3 Don't rewrap functions

Don't fall for the misconception that functions passed as arguments have to start with `fun` or `function`, which leads to the extraneous rewrapping of functions like this:

✗
```
List.map (fun x -> sqrt x) [1.0; 4.0; 9.0; 16.0]
```

Instead, just pass the function directly.

✓
```
List.map sqrt [1.0; 4.0; 9.0; 16.0]
```

You can even do this when the function is an infix binary operator, though you'll need to place the operator in parentheses.

✗
```
List.fold_left (fun x y -> x + y) 0
```

✓
```
List.fold_left (+) 0
```

C.5.4 Avoid computing values twice

When computing values more than once, you may be wasting CPU time (a design consideration) and making your program less clear (a style consideration) and harder to maintain (a consideration of both design and style). The best way to avoid computing things twice is to create a `let` expression and bind the computed value to a variable name. This has the added benefit of letting you document the purpose of the value with a well-chosen variable name, which means less commenting. On the other hand, not every computed sub-value needs to be `let`-bound.

✗
```
f (calc_score (if cond then val1 else val2))
  (calc_score (if cond then val1 else val2))
```

✓
```
let score = calc_score (if cond then val1 else val2) in
f score score
```

C.6 Other common infelicities

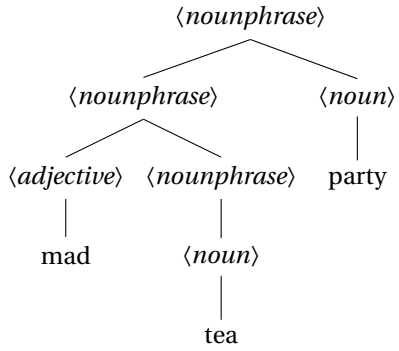
Here is a compilation of some other common infelicities to watch out for:

✗	✓
`x :: []`	`[x]`
`length + 0`	`length`
`length * 1`	`length`
`big_expression * big_expression`	`let x = big_expression in x * x`
`if x then f a b c1 else f a b c2`	`f a b (if x then c1 else c2)`
`String.compare x y = 0`	`x = y`
`String.compare x y < 0`	`x < y`
`String.compare y x < 0`	`x > y`

D

Solutions to selected exercises

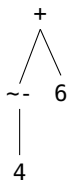
Solution to Exercise 3



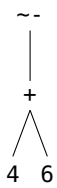
Solution to Exercise 4 There are three structures given the rules provided, corresponding to eaters of flying purple people, flying eaters of purple people, and flying purple eaters of people.

Solution to Exercise 6

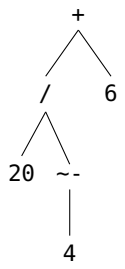
1.



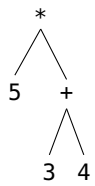
2.



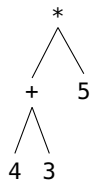
3.



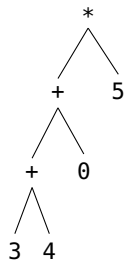
4.



5.



6.



Solution to Exercise 7 Among the concrete expressions of the abstract syntax trees are these, though others are possible.

1. `~- (1 + 42)`
2. `84 / (0 + 42)`
3. `84 + 0 / 42` or `84 + (0 / 42)`

Solution to Exercise 8 The value of the golden ratio is about 1.618. Here's the calculation using OCaml's REPL.

```
# (1. +. sqrt 5.) /. 2. ;;
- : float = 1.6180339887498949
```

Note the consistent use of floating point literals and operators, without which you'd get errors like this:

```
# (1. + sqrt 5.) /. 2. ;;
Line 1, characters 1-3:
```

```
1 | (1. + sqrt 5.) /. 2. ;;
     ^^
Error: This expression has type float but an expression was
  expected of type
         int
```

Solution to Exercise 9 The fourth and seventh might have struck you as unusual.

Why does `3.1416 = 314.16 /. 100.` turn out to be `false`? Floating point arithmetic isn't exact, so that the division `314.16 /. 100.` yields a value that is extremely close to, but not exactly, `3.1416`, as demonstrated here:

```
# 314.16 /. 100. ;;
- : float = 3.14160000000000039
```

Why is `false` less than `true`? It turns out that all values of a type are ordered in this way. The decision to order `false` as less than `true` was arbitrary. Universalizing orderings of values within a type allows for the ordering operators to be polymorphic, which is quite useful, although it does lead to these arbitrary decisions.

Solution to Exercise 10 Only the third of these typings holds, as shown by the REPL.

1. ```
   # (3 + 5 : float) ;;
   Line 1, characters 1-6:
   1 | (3 + 5 : float) ;;
        ^^^^^
   Error: This expression has type int but an expression was expected
     of type
            float
   ```

2. ```
   # (3. + 5. : float) ;;
   Line 1, characters 1-3:
   1 | (3. + 5. : float) ;;
        ^^
   Error: This expression has type float but an expression was
     expected of type
            int
   ```

3. ```
   # (3. +. 5. : float) ;;
   - : float = 8.
   ```

4. ```
   # (3 : bool) ;;
   Line 1, characters 1-2:
   1 | (3 : bool) ;;
        ^
   Error: This expression has type int but an expression was expected
     of type
            bool
   ```

5.
```
# (3 || 5 : bool) ;;
Line 1, characters 1-2:
1 | (3 || 5 : bool) ;;
     ^
Error: This expression has type int but an expression was expected
  of type
         bool
```

6.
```
# (3 || 5 : int) ;;
Line 1, characters 1-2:
1 | (3 || 5 : int) ;;
     ^
Error: This expression has type int but an expression was expected
  of type
         bool
```

Solution to Exercise 11 Since the `unit` type has only one value, there is only one such typing:

```
() : unit
```

Solution to Exercise 12 The types of `succ`, `string_of_int`, and `not` are respectively `int -> int`, `int -> string`, and `bool -> bool`. You can verify the typings at the REPL.

```
# succ ;;
- : int -> int = <fun>
# string_of_int ;;
- : int -> string = <fun>
# not ;;
- : bool -> bool = <fun>
```

Solution to Exercise 13 No good comes of applying a function of type `float -> float` to an argument of type `bool`.

```
# sqrt true ;;
Line 1, characters 5-9:
1 | sqrt true ;;
         ^^^^
Error: This expression has type bool but an expression was expected
  of type
         float
```

Solution to Exercise 14 As it turns out, the `let` construct itself has low precedence so that the body of the `let` extends as far as it can. Evaluating the expression without the parentheses demonstrates this, as otherwise it would have generated an unbound variable error for the second `radius`.

```
# 3.1416 *. let radius = 2.
#           in radius *. radius ;;
- : float = 12.5664
```

Nonetheless, the parentheses arguably improve readability, and they can help autoindenters that implement a less nuanced view of OCaml syntax.

Solution to Exercise 15 The most direct approach uses two `let` binding for the two sides:

```
# let side1 = 1.88496 in
# let side2 = 2.51328 in
# sqrt (side1 *. side1 +. side2 *. side2) ;;
- : float = 3.1416
```

However, by taking advantages of pattern-matching over pairs, which will be introduced later in Section 7.2, a single `let` that binds both variables using pattern matching is arguably more elegant:

```
# let side1, side2 = 1.88496, 2.51328 in
# sqrt (side1 *. side1 +. side2 *. side2) ;;
- : float = 3.1416
```

Solution to Exercise 16 Simply dropping the parentheses solves the problem, since `let` has relatively low precedence, as described in Exercise 14.

```
# let s = "hi ho " in
# s ^ s ^ s ;;
- : string = "hi ho hi ho hi ho "
```

Solution to Exercise 17 As shown in the solution to Exercise 16, the REPL infers the type `string` for `s`.

Solution to Exercise 18

1.
```
let x = 3 in
let y = 4 in
y * y ;;
```

2.
```
let x = 3 in
let y = x + 2 in
y * y ;;
```

3.
```
let x = 3 in
let y = 4 + (let z = 5 in z) + x in
y * y ;;
```

Solution to Exercise 19 The value for `price` at the end is 5. Surprise!

```
# let tax_rate = 0.05 ;;
val tax_rate : float = 0.05
# let price = 5. ;;
val price : float = 5.
# let price = price * (1. +. tax_rate) ;;
Line 1, characters 12-17:
```

```
1 | let price = price * (1. +. tax_rate) ;;
                ^^^^^
Error: This expression has type float but an expression was
  expected of type
         int
# price ;;
- : float = 5.
```

What was probably intended was

```
# let tax_rate = 0.05 ;;
val tax_rate : float = 0.05
# let price = 5. ;;
val price : float = 5.
# let price = price *. (1. +. tax_rate) ;;
val price : float = 5.25
# price ;;
- : float = 5.25
```

with a final value of `price` of `5.25`. Thank goodness for strong static typing, so that the REPL was able to warn us of the error, rather than, for instance, silently rounding the result or some such problematic "correction" of the code.

Solution to Exercise 20 You can get the effect of this definition of a global variable `area` by making use of local variables for `pi` and `radius` by making sure to define the local variables within the global definition:

```
# let area =
#   let radius = 4. in
#   let pi = 3.1416 in
#   pi *. radius ** 2. ;;
val area : float = 50.2656
```

This way, the global `let` is at the top level.

Solution to Exercise 21

1. `2 : int`
2. `2 : int`
3. This sequence of tokens doesn't parse, as `-` is a binary infix operator.
4. `"OCaml" : string`
5. `"OCaml" : string`
6. The expression evaluates to a function (unnamed) of type `string -> string`.

7. Again, the expression evaluates to a function of type `float -> float -> float` (the exponentiation function).

Solution to Exercise 22 A function that squares its floating point argument is

```
# fun x -> x *. x ;;
- : float -> float = <fun>
```

and one to repeat a string is

```
# fun s -> s ^ s ;;
- : string -> string = <fun>
```

Solution to Exercise 23

1. `let foo (b : bool) (n : int) : bool = ...`
2. `let foo (f : float -> int) (x : float) : bool = ...`
3. `let foo (b : bool) (f : int -> bool) : int = ...`

Solution to Exercise 24 Typing them into the REPL reveals their types, `string` and `float -> float`, respectively.

1. ```
   # let greet y = "Hello" ^ y in greet "World!" ;;
   - : string = "HelloWorld!"
   ```
2. ```
   # fun x -> let exp = 3. in x ** exp ;;
   - : float -> float = <fun>
   ```

Solution to Exercise 25

```
# let square (x : float) : float =
# x *. x ;;
val square : float -> float = <fun>
```

Solution to Exercise 26

```
# let abs (n : int) : int =
#   if n > 0 then n else ~- n ;;
val abs : int -> int = <fun>
```

Solution to Exercise 27 The type for `string_of_bool` is `bool -> string`. It can be defined as

```
# let string_of_bool (condition : bool) : string =
#   if condition then "true" else "false" ;;
val string_of_bool : bool -> string = <fun>
```

A common stylistic mistake (discussed in Section C.5.2) is to write the test as `if condition = true then...`, but there's no need for the comparison. What goes in the test part of a conditional is a boolean, and `condition` is already one.

Solution to Exercise 28 Using the compact notation:

```
# let even (n : int) : bool =
#   n mod 2 = 0 ;;
val even : int -> bool = <fun>
```

(Did you try

```
# let even (n : int) : bool =
#   if n mod 2 = 0 then true else false ;;
val even : int -> bool = <fun>
```

instead? That works, but the conditional is actually redundant. Remember, boolean expressions aren't limited to use in the test part of conditionals. Such extraneous conditionals are considered poor style.)

Using the explicit, desugared notation:

```
# let even : int -> bool =
#   fun n -> n mod 2 = 0 ;;
val even : int -> bool = <fun>
```

Dropping the typing information, the REPL still infers the correct type.

```
# let even =
#   fun n -> n mod 2 = 0 ;;
val even : int -> bool = <fun>
```

Nonetheless, the edict of intention argues for retaining the explicit typing information.

Solution to Exercise 32 There are many possibilities. Here are some I find especially nice.

1.
```
let rec odd_terminate (n : int) : int =
  if n < 0 then odd_terminate (~- n)
  else if n = 1 then 0
  else odd_terminate (n - 2) ;;
```

2.
```
let rec small_terminate (n : int) : int =
  if n = 5 then 0
  else small_terminate (n + 1) ;;
```

3.
```
let rec zero_terminate (n : int) : int =
  if n = 0 then 0
  else zero_terminate (n * 2) ;;
```

4.
```
let rec true_terminate (b : bool) : bool =
  b || (true_terminate b) ;;
```

Solution to Exercise 33 The most straightforward recursive solution is simply

```
# let rec fib (i : int) : int =
#   if i = 1 then 0
#   else if i = 2 then 1
#   else fib (i - 1) + fib (i - 2) ;;
val fib : int -> int = <fun>
```

Foreshadowing the discussion of error handling in Chapter 10, the following definition verifies an assumption on the argument, before calculating the number recursively.

```
# let rec fib (i : int) : int =
#   assert (i >= 1);
#   if i = 1 then 0
#   else if i = 2 then 1
#   else fib (i - 1) + fib (i - 2) ;;
val fib : int -> int = <fun>
```

As an alternative for the three way condition, a match statement might be clearer:

```
# let rec fib (i : int) : int =
#   match i with
#   | 1 -> 0
#   | 2 -> 1
#   | _ -> assert (i >= 1);
#          fib (i - 1) + fib (i - 2) ;;
val fib : int -> int = <fun>
```

Solution to Exercise 34

```
# let fewer_divisors (n : int) (bound : int) : bool =
#   let rec divisors_from (start : int) : int =
#     if start > n / 2 then 1
#     else divisors_from (start + 1)
#          + (if n mod start = 0 then 1 else 0) in
#   bound > divisors_from 1 ;;
val fewer_divisors : int -> int -> bool = <fun>
```

Solution to Exercise 35

1. `bool * int`
2. `bool * bool`
3. `int * int`
4. `float * int`
5. `float * int`
6. `int * int`
7. `(int -> int) * (int -> int)`

Solution to Exercise 36

```
# true, true ;;
- : bool * bool = (true, true)
# true, 42, 3.14 ;;
- : bool * int * float = (true, 42, 3.14)
# (true, 42), 3.14 ;;
- : (bool * int) * float = ((true, 42), 3.14)
# (1, 2), 3, 4 ;;
- : (int * int) * int * int = ((1, 2), 3, 4)
# succ, 0, 42 ;;
- : (int -> int) * int * int = (<fun>, 0, 42)
# fun (f, n) -> 1 + f (1 + n) ;;
- : (int -> int) * int -> int = <fun>
```

Solution to Exercise 37

```
# let div_mod (x : int) (y : int) : int * int =
#   x / y, x mod y ;;
val div_mod : int -> int -> int * int = <fun>
```

Solution to Exercise 39

```
# let snd (pair : int * int) : int =
#   match pair with
#   | _x, y -> y ;;
val snd : int * int -> int = <fun>
```

Solution to Exercise 40

```
# let addpair (x, y : int * int) : int =
#   x + y ;;
val addpair : int * int -> int = <fun>

# let fst (x, _y : int * int) : int = x ;;
val fst : int * int -> int = <fun>
```

Solution to Exercise 42 Only expressions 1, 3, 6, and 7 are well-formed, as revealed by the REPL.

1. ```
   # 3 :: [] ;;
   - : int list = [3]
   ```

2. ```
   # true :: false ;;
   Line 1, characters 8-13:
   1 | true :: false ;;
               ^^^^^
   Error: This variant expression is expected to have type bool list
          There is no constructor false within type list
   ```

3. ```
   # true :: [false] ;;
   - : bool list = [true; false]
   ```

4. ```
   # [true] :: [false] ;;
   Line 1, characters 11-16:
   1 | [true] :: [false] ;;
                 ^^^^^
   Error: This variant expression is expected to have type bool list
          There is no constructor false within type list
   ```

5.
```
# [1; 2; 3.1416] ;;
Line 1, characters 7-13:
1 | [1; 2; 3.1416] ;;
           ^^^^^^
Error: This expression has type float but an expression was
  expected of type
         int
```

6.
```
# [4; 2; -1; 1_000_000] ;;
- : int list = [4; 2; -1; 1000000]
```

7.
```
# ([true], false) ;;
- : bool list * bool = ([true], false)
```

Solution to Exercise 43 The `length` function is of type `int list -> int`; it expects an `int list` argument. However, we've applied it to an argument of type `int list list`, that is, a list of integer lists. The types are inconsistent, and OCaml reports the type mismatch.

Solution to Exercise 44

```
# let rec sum (lst : int list) : int =
#   match lst with
#   | [] -> 0
#   | hd :: tl -> hd + sum tl ;;
val sum : int list -> int = <fun>
```

It's natural to return the additive identity 0 for the empty list to simplify the recursion.

This function can also be implemented using the techniques of Chapter 8 as a single fold.

Solution to Exercise 45

```
# let rec prod (lst : int list) : int =
#   match lst with
#   | [] -> 1
#   | hd :: tl -> hd * prod tl ;;
val prod : int list -> int = <fun>
```

It's natural to return the multiplicative identity 1 for the empty list to simplify the recursion.

This function can also be implemented using the techniques of Chapter 8 as a single fold.

Solution to Exercise 46

```
# let rec sums (lst : (int * int) list) : int list =
#   match lst with
#   | [] -> []
#   | (x, y) :: tl -> (x + y) :: sums tl ;;
val sums : (int * int) list -> int list = <fun>
```

Solution to Exercise 47

```
# let rec inc_all lst =
#   match lst with
#   | [] -> []
#   | hd :: tl -> (succ hd) :: inc_all tl ;;
val inc_all : int list -> int list = <fun>
```

Solution to Exercise 48

```
# let rec square_all lst =
#   match lst with
#   | [] -> []
#   | hd :: tl -> (hd * hd) :: square_all tl ;;
val square_all : int list -> int list = <fun>
```

Solution to Exercise 49

```
# let rec append (x : int list) (y : int list)
#              : int list =
#   match x with
#   | [] -> y
#   | hd :: tl -> hd :: (append tl y) ;;
val append : int list -> int list -> int list = <fun>
```

Solution to Exercise 50

```
# let rec concat (sep : string) (lst : string list)
#              : string =
#   match lst with
#   | [] -> ""
#   | [hd] -> hd
#   | hd :: tl -> hd ^ sep ^ (concat sep tl) ;;
val concat : string -> string list -> string = <fun>
```

Solution to Exercise 51

```
# let tesseract = power 4 ;;
val tesseract : int -> int = <fun>
```

If your definition was longer, you'll want to review the partial application discussion.

Solution to Exercise 52

```
# let double_all = map (( * ) 2) ;;
val double_all : int list -> int list = <fun>
```

Solution to Exercise 53

```
# let rec fold_left (f : int -> int -> int)
#                   (init : int)
#                   (xs : int list)
#                 : int =
#   match xs with
#   | [] -> init
#   | hd :: tl -> fold_left f (f init hd) tl ;;
val fold_left : (int -> int -> int) -> int -> int list -> int =
  <fun>
```

Solution to Exercise 54 The definition analogous to the one using `fold_right` is

```
#     let length lst = fold_left (fun tlval _hd -> 1 + tlval) 0 lst
# ;;
val length : int list -> int = <fun>
```

but again this can be further simplified by partial application:

```
# let length = fold_left (fun tlval _hd -> 1 + tlval) 0 ;;
val length : int list -> int = <fun>
```

Solution to Exercise 55 A simple solution is to use `fold_left` itself to implement `reduce`:

```
# let reduce (f : int -> int -> int) (list : int list) : int =
#   match list with
#   | hd :: tl -> List.fold_left f hd tl ;;
Lines 2-3, characters 0-36:
2 | match list with
3 | | hd :: tl -> List.fold_left f hd tl...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val reduce : (int -> int -> int) -> int list -> int = <fun>
```

This approach has the disadvantage that applying `reduce` to the empty list yields an unintuitive "Match failure" error message. Looking ahead to Section 10.3 on handling such errors explicitly, we can raise a more appropriate exception, the `Invalid_argument` exception.

```
# let reduce (f : int -> int -> int) (list : int list) : int =
#   match list with
#   | hd :: tl -> List.fold_left f hd tl
#   | [] -> raise (Invalid_argument "reduce: empty list") ;;
val reduce : (int -> int -> int) -> int list -> int = <fun>
```

Solution to Exercise 56 The `filter` function can be implemented directly as a recursive function by extracting the common elements of the four example functions (`evens`, `odds`, `positives`, and `negatives`) and abstracting over their differences:

```
# let rec filter (test : int -> bool)
#                (lst : int list)
#              : int list =
#   match lst with
#   | [] -> []
#   | hd :: tl -> if test hd then hd :: filter test tl
#                 else filter test tl ;;
val filter : (int -> bool) -> int list -> int list = <fun>
```

Looking ahead to the next chapter, it can also be implemented using polymorphic `fold_right` (from the `List` module):

```
# let filter (test : int -> bool)
#          : int list -> int list =
#   List.fold_right (fun elt accum ->
#                      if test elt then elt :: accum
#                      else accum)
#                   [] ;;
val filter : (int -> bool) -> int list -> int list = <fun>
```

(You may want to revisit this latter solution after reading Chapter 9.)

Solution to Exercise 57 A first stab, maximizing partial application:

```
# let evens = filter (fun n -> n mod 2 = 0) ;;
val evens : int list -> int list = <fun>
# let odds = filter (fun n -> n mod 2 <> 0) ;;
val odds : int list -> int list = <fun>
# let positives = filter ((<) 0) ;;
val positives : int list -> int list = <fun>
# let negatives = filter ((>) 0) ;;
val negatives : int list -> int list = <fun>
```

The last two may be a bit confusing: Why `((<) 0)` for the positives? Don't we want to accept only those that are greater than 0? The < function is curried with its left-side argument before its right-side argument, so that the function `((<) 0)` is equivalent to `fun x -> 0 < x`, that is, the function that returns `true` for positive integers. Nonetheless, the expression `((<) 0)` doesn't "read" that way, which is a good argument for not being so cute and using instead the slightly more verbose but transparent

```
# let positives = filter (fun n -> n > 0) ;;
val positives : int list -> int list = <fun>
# let negatives = filter (fun n -> n < 0) ;;
val negatives : int list -> int list = <fun>
```

Clarity trumps brevity.

Solution to Exercise 58 A list can be reversed by repeatedly appending elements at the end of the accumulating reversal. A `fold_right` implements this solution.

```
# let reverse (lst : int list) : int list =
#   List.fold_right (fun elt accum -> accum @ [elt])
#                   lst [] ;;
val reverse : int list -> int list = <fun>
```

Alternatively, we can start at the left.

```
# let reverse (lst : int list) : int list =
#   List.fold_left (fun accum elt -> elt :: accum)
#                  [] lst ;;
val reverse : int list -> int list = <fun>
```

Taking advantage of partial application, we have

```
# let reverse : int list -> int list =
#   List.fold_left (fun accum elt -> elt :: accum)
#                  [] ;;
val reverse : int list -> int list = <fun>

# reverse [1; 2; 3] ;;
- : int list = [3; 2; 1]
```

Solution to Exercise 59 We want to repeatedly "cons" the elements of the first list onto the second. A `fold_right` will work for this purpose. But there's a subtlety here. The `::` to combine an element and a list is a value constructor, not a function. As such, it can't be passed as an argument. We can construct a function that does the same thing, for instance, `fun elt lst -> elt :: lst`, but conveniently, the `List` module already provides such a function, naturally named `cons`, which we use here.

```
# let append (xs : int list) (ys : int list) : int list =
#   List.fold_right List.cons xs ys ;;
val append : int list -> int list -> int list = <fun>
```

Solution to Exercise 60

```
# let rec odds_evens (lst : 'a list) : 'a list * 'a list =
#   match lst with
#   | [] -> [], []
#   | [a] -> [a], []
#   | odds_head :: evens_head :: tail ->
#   let odds_tail, evens_tail = odds_evens tail in
#     (odds_head :: odds_tail), (evens_head :: evens_tail) ;;
val odds_evens : 'a list -> 'a list * 'a list = <fun>
```

Solution to Exercise 61 The `odds_evens` function is typed as `odds_evens : 'a list -> ('a list * 'a list)`. Note the polymorphic type.

Solution to Exercise 62 Taking advantage of partial application:

```
# let sum =
#   List.fold_left (+) 0 ;;
val sum : int list -> int = <fun>
```

Solution to Exercise 63

```
# let luhn (nums : int list) : int =
#   let odds, evens = odds_evens nums in
#   let s = sum ((List.map doublemod9 odds) @ evens) in
#     10 - (s mod 10) ;;
val luhn : int list -> int = <fun>
```

Solution to Exercise 64 Here are some possible solutions, with commentary on how to think through the problems.

1. You were asked to construct an expression that bears a particular type as inferred by OCaml. The constraint that there be "no explicit typing annotations" was intended to prevent trivial solutions such as this:

```
# let (f : bool * bool -> bool) =
#   fun _ -> true in
# f ;;
- : bool * bool -> bool = <fun>
```

   or even

```
# ((fun _ -> failwith "") : bool * bool -> bool) ;;
- : bool * bool -> bool = <fun>
```

   where the explicit type annotation does the work. The structure of the code does little (respectively, nothing) to manifest the requested type.

   A simple solution relies on the insight that the required type is just the uncurried version of the type for the `(&&)` operator.

```
# let f (x, y) =
#   x && y in
# f ;;
- : bool * bool -> bool = <fun>
```

   A typical approach to this problem is to use a top-level `let` definition of a function, such as this:

```
# let f (x, y) = x && y ;;
val f : bool * bool -> bool = <fun>
```

   Strictly speaking, this is not an expression of OCaml that returns a value, but rather a top-level command that names a value, though the value is of the appropriate type. The value itself can be constructed as a self-contained expression either by using a local `let...in` (as above) or an anonymous function:

```
# fun (x, y) -> x && y ;;
- : bool * bool -> bool = <fun>
```

2. In these problems that ask for a *function* of a given type, it makes sense to start by building the first line of a let definition of the function with its arguments: `let f x = ...` and then figure out how to force `x` and the result to be of the right types. Here, `x` should be an `'a list`, so we better not operate nontrivially on any of its elements. Let's match against the list as would typically happen in a recursive function. This provides the skeleton of the code:

```
let f xs =
  match xs with
  | [] -> ...
  | h :: t -> ... in
f ;;
```

Now, we need to make sure the result type is `bool list`, taking care not to further instantiate `'a`. We can insert any values of the right type as return values, but to continue the verisimilitude, we use the empty list for the first case and a recursive call for the second. (Note the added `rec` to allow the recursive call.)

```
# let rec f xs =
#   match xs with
#   | [] -> []
#   | _h :: t -> true :: (f t) in
# f ;;
- : 'a list -> bool list = <fun>
```

Of course, no recursion is really necessary. For instance, even something as simple as the following does the job (ignoring the inexhaustive match warning).

```
# fun [] -> [true] ;;
Line 1, characters 0-16:
1 | fun [] -> [true] ;;
    ^^^^^^^^^^^^^^^^
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
_::_
- : 'a list -> bool list = <fun>
```

3. A natural approach is to apply the first argument (a function) to a pair composed of the second and third arguments, thereby enforcing that the first argument is of type `'a * 'b -> ...`, viz.,

```
# let f g a b =
#   g (a, b) in
# f ;;
- : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

but this by itself does not guarantee that the result type of the function is `'a`. Rather, `f` types as `('a * 'b -> 'c) -> 'a -> 'b -> 'c`. (It's the `curry` function from lab!) We can fix that by, say, comparing the result with a known value of the right type, namely `a`.

```
# let f g a b =
#   if g (a, b) = a then a else a in
# f ;;
- : ('a * 'b -> 'a) -> 'a -> 'b -> 'a = <fun>
```

4. Again, we start with a `let` definition that just lays out the types of the arguments in a pattern, and then make sure that each component has the right type. One of many possibilities is

```
# let f (i, a, b) alst =
#   if i = 0 && (List.hd alst) = a then [b] else []
# in f ;;
- : int * 'a * 'b -> 'a list -> 'b list = <fun>
```

5. We force the argument to be a `bool` by placing it in the test part of a conditional, and return the only value that we can.

```
# fun b -> if b then () else () ;;
- : bool -> unit = <fun>
```

6. We want to construct a polymorphic, higher-order function that takes arguments of type `'a` and `'a -> 'b`; let's call this function `f`. Notice that the argument of type `'a -> 'b` is also a function; let's call this argument function `g`. Conveniently, the input to the argument function `g` is of the same type as the first input to the higher-order function `f`, that is, of type `'a`. Analogously, the output of the argument function `g` is of the same type as the output of the higher-order function `f`, that is, of type `'b`. We can thus simply apply `g` to the first argument of `f` and return the result:

```
# let f x g = g x ;;
val f : 'a -> ('a -> 'b) -> 'b = <fun>
```

The function `f` is the reverse application function!

```
# ( |> ) ;;
- : 'a -> ('a -> 'b) -> 'b = <fun>
```

7. This question is deceptively simple. The trick here is that the function is polymorphic in both its inputs and outputs, yet the arguments and return type may be different. In fact, we circumvent this issue by simply not returning a value at all. There are two ways to approach this:

   (a) Raise an exception (to be introduced in Section 10.3) instead of returning:

```
# let f x y =
#   if x = y then failwith "true" else failwith "false" ;;
val f : 'a -> 'a -> 'b = <fun>
```

   (b) Recur indefinitely to prevent a return:

```
# let rec f x y =
#   if x = y then f x y else f x y ;;
val f : 'a -> 'a -> 'b = <fun>
```

   or even more elegantly:

```
# let rec f x y = f y x ;;
val f : 'a -> 'a -> 'b = <fun>
```

Solution to Exercise 65 All that needs to be changed from the monomorphic version in the preceding chapter is the typing information in the header. The definition itself naturally works polymorphically.

```
# let rec fold (f : 'a -> 'b -> 'b)
#              (xs : 'a list)
#              (init : 'b)
#            : 'b =
#   match xs with
#   | [] -> init
#   | hd :: tl -> f hd (fold f tl init) ;;
val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# let rec filter (test : 'a -> bool)
#                (lst : 'a list)
#              : 'a list =
#   match lst with
#   | [] -> []
#   | hd :: tl -> if test hd then hd :: filter test tl
#                 else filter test tl ;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Solution to Exercise 66

1. Since `x` is an argument of a `float` operator, it is of type `float`. The result is also of type `float`. Thus `f` is of function type `float -> float`, as can be easily verified in the REPL:

   ```
   # let f x =
   #   x +. 42. ;;
   val f : float -> float = <fun>
   ```

2. The function `f` is clearly of a function type taking two (curried) arguments, that is, of type `... -> ... -> ...`. The argument `g` is also a function, apparently from integers to some result type `'a`, so `f` is of type `(int -> 'a) -> int -> 'a`.

   ```
   # let f g x =
   #   g (x + 1) ;;
   val f : (int -> 'a) -> int -> 'a = <fun>
   ```

3. The argument type for `f`, that is, the type of `x`, must be a list, say, `'a list`. The result type can be gleaned from the two possible return values `x` and `h`. Since `h` is an element of `x`, it must be of type `'a`. Thus the return type is both `'a` and `'a list`. But there is no type that matches both. Thus, the expression does not type.

```
# let f x =
#   match x with
#   | [] -> x
#   | h :: t -> h ;;
Line 4, characters 12-13:
4 | | h :: t -> h ;;
                ^
Error: This expression has type 'a but an expression was expected
  of type
         'a list
       The type variable 'a occurs inside 'a list
```

4. The result type for `f` must be the same as the type of `a` since it returns `a` in one of the match branches. Since `x` is matched as a list, it must be of list type. So far, then, we have `f` of type `... list -> 'a -> 'a`. The elements of `x` (such as `h`) are apparently functions, as shown in the second match branch where `h` is applied to something of type `'a` and returning also an `'a`; so `h` is of type `'a -> 'a`. The final typing is `f : ('a -> 'a) list -> 'a -> 'a`.

```
# let rec f x a =
#   match x with
#   | [] -> a
#   | h :: t -> h (f t a) ;;
val f : ('a -> 'a) list -> 'a -> 'a = <fun>
```

5. The match tells us that the first argument `x` is a pair, whose element `w` is used as a `bool`; we'll take the type of the element `z` to be `'a`. The second argument `y` is applied to `z` (of type `'a`) and returns a `bool` (since the then and else branches of the conditional tell us that `y z` and `w` are of the same type). Thus the type of `f` is given by the typing `f : bool * 'a -> ('a -> bool) -> bool`.

```
let f x y =
  match x with
  | (w, z) -> if w then y z else w ;;
```

6. We can see that we apply `y` to `x` twice. There's nothing else in this function that would indicate a specific typing, so we know our function is polymorphic. Let's say the type of `y` is `'a`. We know that since we can apply `x` to two arguments of type `'a`, and there are no constraints on the output type of `x`, `x` must be of type `'a -> 'a -> 'b`. Since `f` returns `x y y`, we know the output type of `f` must be the same as the output type of `x`. The final typing is thus `f : ('a -> 'a -> 'b) -> 'a -> 'b`.

```
# let f x y =
#   x y y ;;
val f : ('a -> 'a -> 'b) -> 'a -> 'b = <fun>
```

7. This definition does not type. The argument `y` is here applied as a function, so its type must be of the form `'a -> 'b`. Yet the function `y` can take `y` as an argument. This implies that `'a`, the type of the input to `y`, must be identical to `'a -> 'b`, the type of `y` itself. There is no finite type satisfying that constraint. A type cannot be a subpart of itself.

```
# let f x y =
#   x (y y) ;;
Line 2, characters 5-6:
2 | x (y y) ;;
         ^
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```

8. The code matches `x` with option types formed with `Some` or `None`, so we know that `x` must be of type `'a option` for some `'a`. We also see that when deconstructing `x` into `Some y`, we perform subtraction on `y` in the recursive function call: `f (Some (y - 1))`. We can thus conclude `y` is of type `int`, and can further specify `x` to be of type `int option`. Finally, note that the case `None | Some 0 -> None` is the sole terminal case in this recursive function. Because this case returns `None`, we know that if `f` terminates, f returns `None`. Our function `f` therefore outputs a value of type `'a option`. We cannot infer a more specific type for `'a` because we always return `None` and thus have no constraints on `'a`. The final typing is thus as follows: `f : int option -> 'a option`.

```
# let rec f x =
#   match x with
#   | None
#   | Some 0 -> None
#   | Some y -> f (Some (y - 1)) ;;
val f : int option -> 'a option = <fun>
```

9. Since `x` is in the condition of an `if` statement (`if x then ...`), we know that `x` must be of type `bool`. We also can see that both return paths of the code return a list; these lists contain `x`, so we know `f` returns a `bool list`. Since `y` appears in the list `[not x; y]`, we therefore know `y` must be of type `bool` as well. This gives us the overall typing of `f : bool -> bool -> bool list`.

```
# let f x y =
#   if x then [x]
#   else [not x; y] ;;
val f : bool -> bool -> bool list = <fun>
```

Solution to Exercise 67 To implement `map f lst` with a fold, we can start with the empty list and at each step cons on `f` applied to each

element of the `lst`. Here are two solutions, implemented using `fold_left` and `fold_right`, respectively.

```
let map (f : 'a -> 'b) (lst : 'a list) : 'b list =
  List.fold_right (fun elt accum -> f elt :: accum)
                  lst [] ;;

let map (f : 'a -> 'b) (lst : 'a list) : 'b list =
  List.fold_left (fun accum elt -> accum @ [f elt])
                 [] lst ;;
```

The latter can be simplified through use of partial application to

```
let map (f : 'a -> 'b) : 'a list -> 'b list =
  List.fold_left (fun accum elt -> accum @ [f elt]) [] ;;
```

Solution to Exercise 68 An implementation of `fold_right` solely in terms of a single call to `map` over the same list is not possible. The type of `fold_right` makes clear that the output may be of any type. But `map` always returns a list. So a single call to `map` cannot generate the range of answers that `fold_right` can.

One can use `map` in an implementation of `fold_right` in a trivial way, for instance, by vacuously mapping the identity function over the list argument of `fold_right` before doing the real work, but that misses the point of the question, which asks that the implementation use *only* a call to `List.map`.

Solution to Exercise 69 We approach this problem similarly to how we implemented `filter`. The distinction here is that the base case returns two empty lists rather than one, so we have to deconstruct the tuple created by the recursive function call. This results in two output lists – the `yeses` and the `nos` – so we simply pass the current element into the test function and append to the appropriate output list according to the result.

```
# let rec partition (test : 'a -> bool)
#                   (lst : 'a list)
#                 : 'a list * 'a list =
#   match lst with
#   | [] -> [], []
#   | hd :: tl ->
#       let yeses, nos = partition test tl in
#       if test hd then hd :: yeses,  nos
#       else yeses, hd :: nos ;;
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list =
  <fun>
```

Solution to Exercise 70

```
# let rec interleave (n : 'a) (lst : 'a list)
#                  : 'a list list =
```

```
#   match lst with
#   | [] -> [[n]]
#   | x :: xs -> (n :: x :: xs)
#                :: List.map (fun l -> x :: l)
#                            (interleave n xs) ;;
val interleave : 'a -> 'a list -> 'a list list = <fun>

# let rec permutations (lst : 'a list) : 'a list list =
#   match lst with
#   | [] -> [[]]
#   | x :: xs -> List.concat (List.map (interleave x)
#                                      (permutations xs)) ;;
val permutations : 'a list -> 'a list list = <fun>
```

Solution to Exercise 71 We start by providing implementations for `sum` and `prods` making use of the higher-order polymorphic list processing functions of the `List` module.

```
# let sum = List.fold_left (+) 0 ;;
val sum : int list -> int = <fun>
# let prods = List.map (fun (x, y) -> x * y) ;;
val prods : (int * int) list -> int list = <fun>
```

The composition function `@+` is simply

```
# let (@+) f g x = f (g x) ;;
val ( @+ ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

We can use it to implement the `weighted_sum` function and test it out:

```
# let weighted_sum = sum @+ prods ;;
val weighted_sum : (int * int) list -> int = <fun>
# weighted_sum [(1, 3); (2, 4); (3, 5)] ;;
- : int = 26
```

Solution to Exercise 72 The REPL response in the solution to Exercise 71 reveals the polymorphic type of `compose` as `('a -> 'b) -> ('c -> 'a) -> 'c ->'b`, or equivalently but more intuitively, `('b -> 'c) -> ('a -> 'b) -> ('a -> 'c)`.

Solution to Exercise 73 The typings are `hd : 'a list -> 'a` and `tl : 'a list -> 'a list`, as attested by the REPL:

```
# List.hd ;;
- : 'a list -> 'a = <fun>
# List.tl ;;
- : 'a list -> 'a list = <fun>
```

Solution to Exercise 74 That design decision undoubtedly was based on thinking ahead about partial application.

By placing the list argument first, partial application can be used to generate a function that returns the n-th element of a particular list, for example

```
# let pi_digit = List.nth [3;1;4;1;5;9] ;;
val pi_digit : int -> int = <fun>
# pi_digit 0 ;;
- : int = 3
# pi_digit 2 ;;
- : int = 4
```

Solution to Exercise 75 We can first check for the additional anomalous condition.

```
# let rec nth_opt (lst : 'a list) (n : int) : 'a option =
#   if n < 0 then None
#   else
#     match lst with
#     | [] -> None
#     | hd :: tl ->
#         if n = 0 then Some hd
#         else nth_opt tl (n - 1) ;;
val nth_opt : 'a list -> int -> 'a option = <fun>
```

Alternatively, the check could have been done inside the second match statement. Why might this be the dispreferred choice?

Solution to Exercise 76

```
# let rec last_opt (lst : 'a list) : 'a option =
#   match lst with
#   | [] -> None
#   | [elt] -> Some elt
#   | _ :: tl -> last_opt tl ;;
val last_opt : 'a list -> 'a option = <fun>
```

Solution to Exercise 77 Here's a solution that peforms all list calculations directly, making no use of the `List` library. Can you simplify this using the `List` library?

```
# let variance (lst : float list) : float option =
#   let rec sum_length (lst : float list) : float * int =
#     match lst with
#     | [] -> 0., 0
#     | hd :: tl -> let sum, len = sum_length tl in
#                   hd +. sum, 1 + len in
#   let sum, length = sum_length lst in
#     if length < 2
#     then None
#     else let flength = float length in
#          let mean = sum /. flength in
#          let rec residuals (lst : float list) : float =
#            match lst with
#            | [] -> 0.
#            | hd :: tl -> (hd -. mean) ** 2.
#                          +. residuals tl in
#          Some (residuals lst /. (flength -. 1.)) ;;
val variance : float list -> float option = <fun>
```

Solution to Exercise 79 If the first two patterns are `[], []` and `_, _`, the third branch of the match can never be reached. The REPL gives an appropriate warning to that effect:

```
# let rec zip_opt' (xs : 'a list)
#                  (ys : 'b list)
#                : ('a * 'b) list option =
#   match xs, ys with
#   | [], [] -> Some []
#   | _, _ -> None
#   | xhd :: xtl, yhd :: ytl ->
#       match zip_opt' xtl ytl with
#       | None -> None
#       | Some ztl -> Some ((xhd, yhd) :: ztl) ;;
Line 7, characters 2-24:
7 | | xhd :: xtl, yhd :: ytl ->
      ^^^^^^^^^^^^^^^^^^^^^^
Warning 11 [redundant-case]: this match case is unused.
val zip_opt' : 'a list -> 'b list -> ('a * 'b) list option = <fun>
```

Solution to Exercise 80 The function can be implemented as:

```
# let zip_safe (x : 'a list)
#              (y : 'b list)
#            : ('a * 'b) list =
#   try
#     zip x y
#   with
#     Invalid_argument _msg -> [] ;;
val zip_safe : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

However, this approach to handling anomalous conditions in `zip` uses in-band error signaling, which we'd always want to avoid; the error value also happens to be the value returned by the non-error call `zip [] []`.

Solution to Exercise 81

```
# let rec fact (n : int) : int =
#   if n < 0 then
#     raise (Invalid_argument "fact: arg less than zero")
#   else if n = 0 then 1
#   else n * fact (n - 1) ;;
val fact : int -> int = <fun>
```

Solution to Exercise 83

1. ```
   # let f x y =
   #   Some (x + y) ;;
   val f : int -> int -> int option = <fun>
   ```

2. ```
   # let f g =
   #   Some (1 + g 3) ;;
   val f : (int -> int) -> int option = <fun>
   ```

3.
```
# let f x g = g x ;;
val f : 'a -> ('a -> 'b) -> 'b = <fun>
```

or

```
# let f = ( |> ) ;;
val f : 'a -> ('a -> 'b) -> 'b = <fun>
```

4.
```
# let rec f xl yl =
#   match xl, yl with
#   | (Some xhd :: xtl), (Some yhd :: ytl)
#       -> (xhd, yhd) :: f xtl ytl
#   | (None :: _), _
#   | _, (None :: _)
#   | [], _
#   | _, [] -> [] ;;
val f : 'a option List.t -> 'b option List.t -> ('a * 'b) List.t =
  <fun>
```

Solution to Exercise 84

1. No type exists for `f`. Assume that the type of `f` is some instantiation of the function type `'a -> 'b`. Since the first match clause returns `f`, the result type `'b` of `f` must be the entire type `'a -> 'b` of `f` itself. But a type can't contain itself as a subpart. So no type for `f` exists.

2. The type of `f` is `bool -> bool * bool`. In fact, `f` always returns the same value, the pair `true, true`.

Solution to Exercise 85 Taking advantage of the fact that `f` always returns the same value:

```
let f (b : bool) = true, true ;;
```

Note that the explicit typing of `b` is required to force the function type to be `bool -> bool * bool` instead of `'a -> bool * bool`.

Solution to Exercise 87 The REPL provides the answer:

```
# ( |> ) ;;
- : 'a -> ('a -> 'b) -> 'b = <fun>
```

Solution to Exercise 88

```
# let ( |> ) arg func = func arg ;;
val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>
```

Making the types explicit:

```
# let ( |> ) (arg : 'a) (func : 'a -> 'b) : 'b =
#   func arg ;;
val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>
```

Solution to Exercise 89 There are only six card types, so one might be inclined to just have an enumerated type with six constructors:

```
type card =
  | KSpades
  | QSpades
  | JSpades
  | KDiamonds
  | QDiamonds
  | JDiamonds ;;
```

The inelegance of this approach should be clear.

The crucial point here is that the information be kept in a structured form (as specified in the problem), clearly keeping separate information about the suit and the value of a card. This calls for enumerated types for suits and values.

The type for cards can integrate a suit and a value, either by pairing them or putting them into a record. Here, we take the latter approach.

```
type suit = Spades | Diamonds ;;
type cardval = King | Queen | Jack ;;
type card = {suit : suit; cardval : cardval} ;;
```

Note that the field names and type names can be identical, since they are in different namespaces.

Using `int`s for the suits and card values, for instance,

```
type card = {suit : int; cardval : int} ;;
```

is inferior as the convention for mapping between `int` and card suit or value is obscure. At best it could be made clear in documentation, but the enumerated type makes it clear in the constructors themselves. Further, the `int` approach allows `int`s that don't participate in the mapping, and thus doesn't let the language help with catching errors.

We have carefully ordered the constructors from better to worse and ordered the record components from higher to lower order so that comparisons on the data values will accord with the "better" relation, as seen in the solution to Problem 91.

Solution to Exercise 90 The `better` function is supposed to take two cards and return a truth value, so if the arguments are taken curried, then

```
better : card -> card -> bool
```

Alternatively, but less idiomatically, the function could be uncurried:

```
better : card * card -> bool
```

Solution to Exercise 91 The following oh-so-clever approach works if you carefully order the constructors and fields from best to worst and higher order (suit) before lower order (cardval), as is done in the solution to Problem 89.

```
let better (card1 : card) (card2 : card) : bool =
  card1 < card2 ;;
```

This relies on the fact that the < operator has a kind of ad hoc polymorphism, which works on enumerated and variant types, pairs, and records inductively to define an ordering on values of those types. Relying on this property of variant types behooves you to explicitly document the fact at the type definition so it gets preserved.

To not rely on the ad hoc polymorphism of <, we need a more explicit definition like this:

```
let better ({suit = suit1; cardval = cardval1} : card)
           ({suit = suit2; cardval = cardval2} : card)
         : bool =
  let to_int v = match v with
                 | King -> 3
                 | Queen -> 2
                 | Jack -> 1 in
  if suit1 = suit2 then
    (to_int cardval1) > (to_int cardval2)
  else suit1 = Spades ;;
```

though this is hacky since it doesn't generalize well to adding more suits. Of course, a separate map of suits to an `int` value would solve that problem. Many other approaches are possible.

Solution to Exercise 94

```
# let str_bintree =
#   Node ("red",
#     Node ("orange",
#       Node ("green",
#         Node ("blue", Empty, Empty),
#         Node ("indigo", Empty, Empty)),
#       Empty),
#     Node ("yellow",
#       Node ("violet", Empty, Empty),
#       Empty)) ;;
val str_bintree : string bintree =
  Node ("red",
   Node ("orange",
    Node ("green", Node ("blue", Empty, Empty),
     Node ("indigo", Empty, Empty)),
    Empty),
   Node ("yellow", Node ("violet", Empty, Empty), Empty))
```

Solution to Exercise 95

```
# let rec preorder (t : 'a bintree) : 'a list =
#   match t with
#   | Empty -> []
#   | Node (n, left, right) ->
#       n :: preorder left @ preorder right ;;
val preorder : 'a bintree -> 'a list = <fun>
```

Solution to Exercise 96 Let's start with the tree input and the output of the tree traversal. The third argument to `foldbt` is a binary tree of type `'a bintree`, say. The result of the traversal is a value of type, say, `'b`. Then the first argument, which serves as the return value for empty trees must also be of type `'b` and the function calculating the values for internal nodes is given the value stored at the node (`'a`) and the two recursively returned values and returns a `'b`; it must be of type `'a -> 'b -> 'b -> 'b`. Overall, the appropriate type for `foldbt` is

```
'b -> ('a -> 'b -> 'b -> 'b) -> 'a bintree -> 'b
```

Solution to Exercise 97 A directly recursive implementation looks like

```
# let rec foldbt (emptyval : 'b)
#                (nodefn : 'a -> 'b -> 'b -> 'b)
#                (t : 'a bintree)
#              : 'b =
#   match t with
#   | Empty -> emptyval
#   | Node (value, left, right) ->
#       nodefn value (foldbt emptyval nodefn left)
#                    (foldbt emptyval nodefn right) ;;
val foldbt : 'b -> ('a -> 'b -> 'b -> 'b) -> 'a bintree -> 'b =
  <fun>
```

Notice that each time `foldbt` is recursively called, it passes along the same first two arguments.The following version of `foldbt` uses a local function to avoid this redundancy.

```
# let foldbt (emptyval : 'b)
#            (nodefn : 'a -> 'b -> 'b -> 'b)
#            (t : 'a bintree)
#          : 'b =
#   let rec foldbt' t =
#     match t with
#     | Empty -> emptyval
#     | Node (value, left, right) ->
#         nodefn value (foldbt' left) (foldbt' right) in
#   foldbt' t ;;
val foldbt : 'b -> ('a -> 'b -> 'b -> 'b) -> 'a bintree -> 'b =
  <fun>
```

Here's a third slightly less attractive alternative, which introduces a level of function application indirection and doesn't take advantage of the lexical scoping.

```
# let rec foldbt (emptyval : 'b)
#                (nodefn : 'a -> 'b -> 'b -> 'b)
#                (t : 'a bintree)
#              : 'b =
#   let foldbt' = foldbt emptyval nodefn in
#   match t with
#   | Empty -> emptyval
```

```
#   | Node (value, left, right) ->
#       nodefn value (foldbt' left)
#                    (foldbt' right) ;;
val foldbt : 'b -> ('a -> 'b -> 'b -> 'b) -> 'a bintree -> 'b =
  <fun>
```

At least it uses the partial application of `foldbt` in the definition of `foldbt'`.

Solution to Exercise 98 By abstracting out the generic tree walking, this and other functions can be succinctly implemented. The value of the sum for an empty tree is 0, and the function to be applied to the value at a node and the values of the subtrees should just sum up those three values.

```
# let sum_bintree =
#   foldbt 0 (fun v l r -> v + l + r) ;;
val sum_bintree : int bintree -> int = <fun>
# preorder int_bintree ;;
- : int list = [16; 93; 3; 42]
```

Solution to Exercise 99

```
# let preorder tree =
#   foldbt [] (fun v l r -> v :: l @ r) tree ;;
val preorder : 'a bintree -> 'a list = <fun>
# preorder int_bintree ;;
- : int list = [16; 93; 3; 42]
```

Why not partially apply `foldbt`, as in the `sum_bintree` example? Because of the problem with weak type variables noted in Section 9.6.

Solution to Exercise 100

```
# let find (tree : 'a bintree) (value : 'a) : bool =
#   foldbt false
#          (fun v l r -> (value = v) || l || r)
#          tree ;;
val find : 'a bintree -> 'a -> bool = <fun>
```

You'll want to avoid redundant locutions like `(l = true)` in the second to last line. See Section C.5.2 in the style guide.

Solution to Exercise 101 An implementation with the top element at the end of the list requires walking the whole list to dequeue an element. We add a function `split` to perform the walk. To keep track of the remaining queue elements, `split` uses an accumulator to add the elements we walk past. This implementation is considerably more complicated and requires repeatedly adding elements to the end of the accumulator, making it far less efficient as well.

```
#       (* IntQueue -- An implementation of integer queues as
#          int lists, where the elements are kept with older
#          elements closer to the end of the list. *)
#       module IntQueueAlternate =
#         struct
#           type int_queue = int list
#           let empty_queue : int_queue = []
#           let enqueue (elt : int) (q : int_queue)
#                     : int_queue =
#             elt :: q
#
#           let rec split (q : int_queue) (rest : int_queue) : int *
#   int_queue =
#             match q with
#             | [] -> raise (Invalid_argument
#                                "dequeue: empty queue")
#             | [top] -> top, rest
#             | hd :: tl ->
#                split tl (rest @ [hd])
#
#           let dequeue (q : int_queue) : int * int_queue =
#             split q []
#         end ;;
module IntQueueAlternate :
  sig
    type int_queue = int list
    val empty_queue : int_queue
    val enqueue : int -> int_queue -> int_queue
    val split : int_queue -> int_queue -> int * int_queue
    val dequeue : int_queue -> int * int_queue
  end
```

Solution to Exercise 102 We specify the signature of the dictionary to provide only an abstract type and the types of the functions, along with an exception to raise in case of duplicate keys.

```
# module type DICTIONARY = sig
#   exception KeyAlreadyExists of string
#   type ('key, 'value) dictionary
#   val empty : ('key, 'value) dictionary
#   val add : ('key, 'value) dictionary -> 'key -> 'value
#                   -> ('key, 'value) dictionary
#   val lookup : ('key, 'value) dictionary -> 'key -> 'value option
#   end ;;
module type DICTIONARY =
  sig
    exception KeyAlreadyExists of string
    type ('key, 'value) dictionary
    val empty : ('key, 'value) dictionary
    val add :
      ('key, 'value) dictionary ->
      'key -> 'value -> ('key, 'value) dictionary
    val lookup : ('key, 'value) dictionary -> 'key -> 'value option
  end
```

In this implementation of the signature, dictionaries are represented as lists of pairs of keys and values. Unlike the implementation in Section 11.3, here we take advantage of some `List` module functions to simplify the implementation.

```
# module Dictionary : DICTIONARY = struct
#   exception KeyAlreadyExists of string
#   type ('key, 'value) dictionary = ('key * 'value) list
#   let empty = []
#   let add dict key value =
#     if List.exists (fun (k, _) -> k = key) dict then
#       raise (KeyAlreadyExists "add: duplicate key")
#     else
#       (key, value) :: dict
#   let lookup dict key =
#     try Some (snd (List.find (fun (k, _) -> k = key) dict))
#     with Not_found -> None
# end ;;
module Dictionary : DICTIONARY
```

Clearly, dictionaries with duplicate keys canot be constructed using the `Dictionary` module.

Solution to Exercise 103 What we were looking for here is the proper definition of a functor named `MakeImaging` taking an argument, where the functor and argument are appropriately signature-constrained.

```
module MakeImaging (P : PIXEL)
                 : (IMAGING with type pixel = P.t) =
  struct
    (* ... the implementation would go here ... *)
  end ;;
```

Typical problems are to leave out the `: PIXEL`, the `: IMAGING`, or the sharing constraint.

Solution to Exercise 104 Applying the functor to the `IntPixel` module is simply

```
module IntImaging = MakeImaging(IntPixel) ;;
```

Optionally, signature specifications can be added, so long as appropriate sharing constraints are provided.

Solution to Exercise 105 Here, a local open simplifies things.

```
let open IntImaging in
  depict (const (to_pixel 5000) (100, 100)) ;;
```

Solution to Exercise 106

```
module MakeInterval (Point : COMPARABLE)
         : (INTERVAL with type point = Point.t) =
  struct
    (* ... the implementation would go here ... *)
  end ;;
```

Solution to Exercise 107

```
module DiscreteTime : (COMPARABLE with type t = int) =
  struct
    type t = int
    type order = Less | Equal | Greater
    let compare x y = if x < y then Less
                      else if x = y then Equal
                      else Greater
  end ;;
```

Solution to Exercise 108

```
module DiscreteTimeInterval =
  MakeInterval (DiscreteTime) ;;
```

Solution to Exercise 109

```
let intersection i j =
  if relation i j = Disjoint then None
  else let (x, y), (x', y') = endpoints i, endpoints j in
       Some (interval (max x x') (min y y')) ;;
```

Solution to Exercise 110 There are myriad solutions here. The idea is just to establish a few intervals and then test that you can recover some endpoints or relations. Here are a few possibilities:

```
open Absbook ;;
let test () =
  let open DiscreteTimeInterval in
  let i1 = interval 1 3 in
  let i2 = interval 2 6 in
  let i3 = interval 0 7 in
  let i4 = interval 4 5 in
  unit_test (relation i1 i4 = Disjoint) "disjoint\n";
  unit_test (relation i1 i2 = Overlaps) "overlaps\n";
  unit_test (relation i1 i3 = Contains) "contains\n";
  unit_test
(relation (union i1 i2) i4 = Contains) "unioncontains\n";
  let i23 = intersection i1 i2 in
  un
it_test (let
 Some e23 = i23 in endpoints e23 = (2, 3)) "intersection";;
print_endline "tests completed" ;;
```

Solution to Exercise 111 Since we only need the float functionality for `weight`, a simple definition is best.

```
# type weight = float ;;
type weight = float
```

Solution to Exercise 112 Since we want all shapes to be one of three distinct types – either a circle OR an oval OR a fin – we want to use a disjunctive type here. Variant types get the job done.

```
# type shape =
#   | Circle
#   | Oval
#   | Fin ;;
type shape = Circle | Oval | Fin
```

Solution to Exercise 113 Since we want each object to have two attributes – a weight AND a shape – we want to use conjunction here. We can construct a record type `obj` to represent objects. This allows us to ensure each object has a weight and shape that are of the appropriate type.

```
# type obj = { weight : weight; shape : shape } ;;
type obj = { weight : weight; shape : shape; }
```

Solution to Exercise 114 In the header of the functor, we want to explicate the name of the functor and the type of the input module, as well as any sharing constraint. We want to transform any module of type `BINTREEARG` into a module of type `BINTREE`. We also need to add sharing constraints so that the types for `leaft` and `nodet` in the output module of type `BINTREE` are of the same type as the `leaft` and `nodet` in the `Element` module.

```
module MakeBintree (Element : BINTREE_ARG)
  : (BINTREE with
      type leaft = Element.leaft and
      type nodet = Element.nodet) =
  struct
    ..... (* the implementation would go here *)
  end ;;
```

Solution to Exercise 115 To define a `Mobile` with `objs` as leaves and `weights` as nodes, we just need to pass in a module of type `BINTREEARG`. This argument module will also have leaves of type `obj` and nodes of type `weight`:

```
module Mobile = MakeBinTree (struct
                               type leaft = obj
                               type nodet = weight
                             end) ;;
```

An alternative is to explicitly define the argument values:

```
module MobileArg =
  struct
    type leaft = obj
    type nodet = weight
end ;;

module Mobile = MakeBintree (MobileArg) ;;
```

If a module type is given to the argument module, however, there need to be sharing constraints. So the following won't work:

```
module MobileArg : BINTREE_ARG =
  struct
    type leaft = obj
    type nodet = weight
end ;;

module Mobile = MakeBintree (MobileArg) ;;
```

It should be

```
module MobileArg : (BINTREE_ARG with type leaft = obj
                                 and type nodet = weight) =
  struct
    type leaft = obj
    type nodet = weight
  end ;;

module Mobile = MakeBintree (MobileArg) ;;
```

Solution to Exercise 116 The only aspects pertinent to the *use* of a module are manifest in the *signature*. A user need not know *how* a module of type `BINTREE`, say, makes a leaf; a user only needs to know the signature of the `make_leaf` function in order to use it. A user in fact cannot access the implementation details because we've constrained the module to the `BINTREE` interface Similarly, a user need not know how the functor `MakeBintree` works, as implementation details would not be accessible to the user anyway. So long as a user knows the functor's signature, they know if they pass in any module following the `BINTREE_ARG` signature, the functor will return a module follwing the `BINTREE` signature.

Solution to Exercise 117 We construct the mobile using the `make_leaf` and `make_node` functions in the `Mobile` module.

```
let mobile1 =
  let open Mobile in
  make_node
    1.0
    (make_leaf {shape = Oval; weight = 9.0})
    (make_node
      1.0
      (make_leaf {shape = Fin; weight = 3.5})
      (make_leaf {shape = Fin; weight = 4.5})) ;;
```

Solution to Exercise 118 The `size` function takes in a binary tree representing a mobile and returns the number of leaves in that tree. The type is thus `Mobile.tree -> int`.

Solution to Exercise 119 Notice that we pass in `mobile` as an argument to `size`, only to just pass it in again as the last argument to `Mobile.walk`; partial application allows us to simplify as follows:

```
let size =
  Mobile.walk (fun _leaf -> 1)
              (fun _node left_size right_size ->
                 left_size + right_size) ;;
```

Solution to Exercise 120 Let's first think about the signature of `shape_count`. We want `shape_count` to take in a value of type `shape` and output an `int`, so its type is `shape -> int`, leading to a first line of

```
let shape_count (s : shape) : int = ...
```

We're told we want to use the `walk` function here. Since the `walk` function does the hard work of traversing the `Mobile.tree` for us, we just need to pass in the proper arguments to `walk` in order to construct the function `shape_count`. The `walk` function is of type `(leaft -> 'a) -> (nodet -> 'a -> 'a -> 'a) -> tree -> 'a` and takes in two functions, one specifying behavior for leaves and one for nodes. If we can define these two functions, we can easily define shape. Let's start with the function that specifies how we want to count leaves; we need a function of type `leaf -> 'a`. The `shape_count` of a single leaf should be 1 if the leaf matches the desired shape `s` and 0 otherwise. We can construct an anonymous function that achieves this functionality as follows:

```
fun leaf -> if leaf.shape = s then 1 else 0
```

We now want to address the case of nodes. Nodes don't have shapes themselves, but rather connect to other subtrees that might. To find the shape count of a node, we just need to add the shape counts of its subtrees.

```
fun _node l r -> l + r ;;
```

Putting it all together, we get

```
let shape_count (s : shape) =
  Mobile.walk
   (fun leaf -> if leaf.shape = s then 1 else 0)
   (fun _node left_count right_count ->
     left_count + right_count) ;;
```

Solution to Exercise 121 No, this mobile is not balanced. To determine whether the mobile is balanced, we can just sum the total weight on each node. The right subtree connects two submobiles of different weights (3.5 and 4.5).

Solution to Exercise 122 Again, we can use the `walk` function here to avoid traversing the tree directly. We will again need to come up with two functions to pass into `walk`, one for the leaves and one for the nodes. Let's look at the base case, leaves. A leaf is always balanced, so we just ned to return `Some w`, where `w` is the weight of the leaf.

```
fun leaf -> Some leaf.weight
```

Now, let's look at the nodes. We want a function of the form `nodet -> 'a -> 'a -> 'a`, where the first argument is the node itself and the remaining two are the results of `walk` on the left subtree and `walk` on the right subtree, respectively. We want to ensure our node is balanced: this requires that the left and right subtrees are each balanced and are of equal weight. If these conditions are met we want to return If the subtrees aren't balanced or are of unequal weight, we want to return `Some w`, where `w` is the sum of the weights of the connector and its subtrees. We return `None` otherwise.

```
fun node left right ->
  match left, right with
  | Some wt1, Some wt2 ->
      if wt1 = wt2 then Some (node +. wt1 +. wt2)
      else None
  | _, _ -> None) ;;
```

Putting it all together and passing in our mobile as an argument, we get:

```
let balanced (mobile : Mobile.tree) =
  Mobile.walk (fun leaf -> Some leaf.weight)
              (fun node left right ->
                 match left, right with
                 | Some wt1, Some wt2 ->
                     if wt1 = wt2 then
                       Some (node +. wt1 +. wt2)
                     else None
                     | _, _ -> None)
              mobile ;;
```

Note the redundancy of passing in `mobile`. We can use partial application and arrive at the following final solution:

```
let balanced =
  Mobile.walk (fun leaf -> Some leaf.weight)
              (fun node l r ->
               match l, r with
               | Some wt1, Some wt2 ->
                   if wt1 = wt2 then
                     Some (node +. wt1 +. wt2)
                   else None
               | _, _ -> None) ;;
```

Solution to Exercise 124 Since the `+` operator is left-associative, the concrete syntax `3 + 5 + 7` corresponds to the same abstract syntax as `(3 + 5) + 7`. The derivation is structured accordingly. The alternate derivation provided in the exercise corresponds to the evaluation of the concrete expression `3 + (5 + 7)`.

Solution to Exercise 137 R_{fun}: "A function expression of the form `fun` x `->` B evaluates to itself."

R_{app}: "To evaluate an application of the form P Q, first evaluate P to a function value of the form `fun` x `->` B and Q to a value v_Q. Then evaluate the expression resulting from substituting v_Q for free occurrences of x in B to a value v_B. The value of the full expression is then v_B."

Solution to Exercise 139 The derivation starts as usual, until we get to the highlighted derivation of `((fun y -> f 3) 1)`$[$`f`$\mapsto$`fun x -> y`$]$. Our better understanding of how substitution should work, as codified in the new substitution rules, tells us that this substitution uses the third rule, not the second, that is, we get `(fun z -> (fun x-> y) 3) 1`, using the fresh variable `z`. The derivation then continues:

```
let f = fun x -> y in (fun y -> f 3) 1
   ⇓
   │ fun x -> y⇓fun x -> y
   │ (fun z -> (fun x -> y) 3) 1
   │    ⇓
   │    │ (fun z -> (fun x -> y) 3)⇓(fun z -> (fun x -> y) 3)
   │    │ 1⇓1
   │    │ (fun x -> 1) 3⇓
   │    │                │ fun x -> y⇓fun x -> y
   │    │                │ y⇓???
   │    │                ⇓???
   │    ⇓???
   ⇓???
```

At this point, the derivation breaks down, as the variable `y` is unbound.

Solution to Exercise 140

`(let` x `=` Q `in` R`)`$[x\mapsto P] =$ `let` x `=` $Q[x\mapsto P]$ `in` R

`(let` y `=` Q `in` R`)`$[x\mapsto P] =$ `let` y `=` $Q[x\mapsto P]$ `in` $R[x\mapsto P]$
where $x \not\equiv y$ and $y \notin FV(P)$

`(let` y `=` Q `in` R`)`$[x\mapsto P] =$ `let` z `=` $Q[x\mapsto P]$ `in` $R[y\mapsto z][x\mapsto P]$
where $x \not\equiv y$ and $y \in FV(P)$ and z is a fresh variable

Solution to Exercise 145

```
#       module MergeSort : SORT =
#         struct
#           let rec split lst =
#             match lst with
```

```
#             | []
#             | [_] -> lst, []
#             | first :: second :: rest ->
#                 let first', second' = split rest in
#                 first :: first', second :: second'
#
#           let rec merge lt xs ys =
#             match xs, ys with
#             | [], _ -> ys
#             | _, [] -> xs
#             | x :: xst, y :: yst ->
#                if lt x y then x :: (merge lt xst ys)
#                else y :: (merge lt xs yst)
#
#           let rec sort (lt : 'a -> 'a -> bool)
#                        (xs : 'a list)
#                      : 'a list =
#             match xs with
#             | []
#             | [_] -> xs
#             | _ -> let first, second = split xs in
#                    merge lt (sort lt first) (sort lt second)
#         end ;;
module MergeSort : SORT
```

Solution to Exercise 146 The claims in 1, 2, 4, and 5 hold.

Solution to Exercise 147

1. Big-O notation only gives you information about the worst-case performance as the input size becomes very large. Because of this, it ignores lower-order terms and constants that may have a large effect for small inputs. So A may be slower than B for some inputs, and the statement is *false*.

2. Since big-O notation provides worst-case performance, and A is polynomial in big-O, they can be guaranteed that for any input (except for a finite set), A will run in polynomial time, so the statement is *true*.

3. As a worst-case bound, big-O doesn't say anything about average-case performance, so the statement is *false*.

Solution to Exercise 148 Since `length` is linear in the length of its argument, `compare_lengths` is linear in the sum of the lengths of its two arguments. But that sum is less than or equal to twice the length of the longer argument. Thus, `compare_lengths` is in $O(2n)$, where n is the length of the longer argument, hence, dropping multiplicative constants, $O(n)$.

An alternative implementation, which stops the recursion once the shorter list is exhausted, is linear in the length of the shorter list.

```
# let rec compare_lengths xs ys =
#   match xs, ys with
#   | [], [] -> 0
#   | _, [] -> 1
#   | [], _ -> -1
#   | _xhd :: xtl, _yhd :: ytl -> compare_lengths xtl ytl ;;
val compare_lengths : 'a list -> 'b list -> int = <fun>
# compare_lengths [1] [2;3;4] ;;
- : int = -1
# compare_lengths [1;2;3] [4] ;;
- : int = 1
# compare_lengths [1;2] [3;4] ;;
- : int = 0
```

Solution to Exercise 150

$$T_{odds}(n) = c + T_{odds}(n-2)$$

More detail in the equation in terms of constants for different bits is unnecessary, but benign. Note the $n-2$, though $n-1$ yields the same complexity.

Solution to Exercise 151 Linear – $O(n)$.

Solution to Exercise 152 The O classes are independent of multiplication or division by constants, so each "triplet" of answers after the first are equivalent. Since f is $O(n)$, it is also $O(n^2)$ etc. for all higher classes. Thus, all answers from the fifth on are correct.

Solution to Exercise 153 $O(n)$ – linear. The odds and evens function are both linear and return a list of length linear in n. The append is linear in the length of the odds list, so also linear in n. The sum is linear in the length of its argument, which is identical in length to (and thus linear in) n. The `let` body is constant time. Summing these complexities up, we're left with linear and constant terms, which is dominated by the linear term. Hence the function is linear.

Solution to Exercise 154 Let's start with two mutable values of type `int list ref` that are structurally equal but physically distinct:

```
# let lstref1 = ref [1; 2; 3] ;;
val lstref1 : int list ref = {contents = [1; 2; 3]}
# let lstref2 = ref [1; 2; 3] ;;
val lstref2 : int list ref = {contents = [1; 2; 3]}
# lstref1 = lstref2 ;;
- : bool = true
# lstref1 == lstref2 ;;
- : bool = false
```

Modifying one of them makes them both structurally and physically unequal:

```
# lstref1 := [4; 5] ;;
- : unit = ()
# lstref1 = lstref2 ;;
- : bool = false
# lstref1 == lstref2 ;;
- : bool = false
```

Now for two values that are physically equal (that is, aliases), and therefore structurally equal as well:

```
# let lstref3 = ref [1; 2; 3] ;;
val lstref3 : int list ref = {contents = [1; 2; 3]}
# let lstref4 = lstref3 ;;
val lstref4 : int list ref = {contents = [1; 2; 3]}
# lstref3 = lstref4 ;;
- : bool = true
# lstref3 == lstref4 ;;
- : bool = true
```

Modifying one of them retains their physical, and hence structural equality:

```
# lstref3 := [4; 5] ;;
- : unit = ()
# lstref3 = lstref4 ;;
- : bool = true
# lstref3 == lstref4 ;;
- : bool = true
```

Solution to Exercise 155 We evaluate the expressions in the REPL to show their types and values, ignoring the warnings.

1.
```
# let a = ref 3 in
# let b = ref 5 in
# let a = ref b in
# !(!a) ;;
Line 1, characters 4-5:
1 | let a = ref 3 in
        ^
Warning 26 [unused-var]: unused variable a.
- : int = 5
```

2. In this example, `a` is a reference to `b`,which is itself a reference to `a`. If we take the type of `a` to be `'a` then, `b` must be of type `'a ref` and `a` (of type `'a` remember) must also be of type `'a ref ref`, leading to an infinite regress of types. The expression is thus not well-typed. The REPL reports accordingly.

```
# let rec a, b = ref b, ref a in
# !a ;;
Line 1, characters 22-27:
1 | let rec a, b = ref b, ref a in
                          ^^^^^
Error: This expression has type 'a ref ref
```

```
but an expression was expected of type 'a
The type variable 'a occurs inside 'a ref ref
```

3. Note the warning that the *inner* definition of `a` is not used; the `a` used in the definition of `b` is the outer one, as required by lexical scoping. (The REPL even reports that the inner `a` is unused.)

```
# let a = ref 1 in
# let b = ref a in
# let a = ref 2 in
# !(!b) ;;
Line 3, characters 4-5:
3 | let a = ref 2 in
        ^
Warning 26 [unused-var]: unused variable a.
- : int = 1
```

4.
```
# let a = 2 in
# let f = (fun b -> a * b) in
# let a = 3 in
# f (f a) ;;
- : int = 12
```

Solution to Exercise 157

1. Since we've just declared `p` as a reference to the integer `11`, `p` is of type `int ref`

```
# let p = ref 11 ;;
val p : int ref = {contents = 11}
```

2. Our variable `r` is a reference to our variable `p`. We defined `p` as a reference to an integer, so `r` is a reference to this reference, or an `int ref ref`.

```
# let r = ref p ;;
val r : int ref ref = {contents = {contents = 11}}
```

3. (a) False. We know `p` is of type `int ref`. Since we declare `s` as `s = ref !r`, we know that `s` is a reference to the same value that `r` references. Since `!r = p`, we therefore know `s` is also a reference to `p`, and thus also of type `int ref ref`. The types of `p` and `r` are therefore not the same.

```
# let s = ref !r ;;
val s : int ref ref = {contents = {contents = 11}}

# p ;;
- : int ref = {contents = 11}
```

(b) True. The explanation here is the same as for (1): Since `s` is a reference to `!r`, it's of type `int ref ref`, the type of `r`.

```
# r ;;
- : int ref ref = {contents = {contents = 11}}
```

```
# s ;;
- : int ref ref = {contents = {contents = 11}}
```

(c) False. As explained in the solution to (1), `p` is a reference to `11`, whereas `s` contains a reference to that reference.

```
# p ;;
- : int ref = {contents = 11}

# s ;;
- : int ref ref = {contents = {contents = 11}}
```

(d) True. We see `r` and `s` are a reference to the same value – that is, they both are references to `p` – they therefore are structurally equivalent.

```
# r ;;
- : int ref ref = {contents = {contents = 11}}

# s ;;
- : int ref ref = {contents = {contents = 11}}
```

4. We know the starting values of `p`, `r`, and `s`: `p` is a reference to the integer `11`, and `s` and `r` are two different references to `p`.

   To find the value of `t`, let's track the value of each variable at each step in 4–6. We first set the dereference of `s` to equal `14` with the line `!s = 14`. We know that since `s` is a reference to `p`, as found in question (2), `!s` points to the same address as `p`. When we reassign `!s` to `14`, we're thus changing the value at the block of memory to which `p` points to store the value `14`.

   We now set `t` equivalent to the expression `!p + !(!r) + !(!s)`; in order to compute this we must first evaluate each of the addends:

   - `!p`: As described above, since `s` is a reference to `p`, `!s` points to the same address as `p`; by replacing the value at that block with `14`, `p` is now a reference to the value `14`. Dereferencing `p` with `!p` thus gives us the integer `14`.
   - `!(!r)`: As described in the explanation to (3), we know `r` is a reference to `p`, so `!r` points to the same address as `p`. We know `!(!r)`, therefore, is equal to `!p`, which we found above to be `14`.
   - `!(!s)`: Again, `s` is still a reference to `p`, so `!s` would point to the address as `p`. By dereferencing `s` again, with `!(!s)`, we're therefore returning the value to which `p` points, that is, `14`.

   Putting it all together, we can see that this evaluates to `14 + 14 + 14`, so `t = 42`.

```
# let t =
#   !s := 14;
#   !p + !(!r) + !(!s) ;;
```

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

5. Note how similar the code in 7–9 looks to the code in 4–6. Yet there is in fact one key difference: we're changing `s` itself rather than `!s`. This means that instead of modifying our reference to `p`, we're replacing it. With the line `s := ref 17`, we're declaring an entirely new reference that points to an instance of the value `17`, and setting `s` to point to that reference. This effectively severs the tie between `s` and `p`: `s` points to a to a completely separate reference to a block of memory containing the value `17`, while `p` continues to point to the value `14`.

   As for `r`, note that while `s` and `r` started out *structurally* equivalent, they were never `physically` equivalent. Think back to when we defined `s`:

   ```
   let s = ref !r ;;
   ```

   When we dereference `r` with `!r`, we lose all association with the specific block of memory to which `r` refers and are only passed along the value contained in that block. Thus while `s` is also a reference to the value `r` references – that is, both `s` and `r` are references to `p` – `s` and `r` are in fact distinct references pointing to distinct blocks in memory. Because `s` and `r` are not structurally equivalent, `s` is still a reference to `p`.

   Putting it all together, we again evaluate each of the addends in the expression `!p + !(!r) + !(!s)`; `!p` and thus `!(!r)` each evaluate to `14`, while `!(!s))` now evaluates to `17`. We're thus left with `14 + 14 + 17`, and `t = 45`.

   ```
   # let t =
   #   s := ref 17;
   #   !p + !(!r) + !(!s) ;;
   val t : int = 45
   # t ;;
   - : int = 45
   ```

Solution to Exercise 158 In this solution, we explicitly raise a `Failure` exception (a la `List.hd` and `List.tl`) when applied to the empty mutable list:

```
# let mhead mlst =
#  match !mlst with
#  | Nil -> raise (Failure "mhead: empty list")
#  | Cons (hd, _tl) -> hd ;;
val mhead : 'a mlist_internal ref -> 'a = <fun>
```

```
# let mtail mlst =
#   match !mlst with
#   | Nil -> raise (Failure "mtail: empty list")
#   | Cons (_hd, tl) -> tl ;;
val mtail : 'a mlist_internal ref -> 'a mlist = <fun>
```

Solution to Exercise 159 We evaluate the expressions in the REPL to show their types and values.

1.
```
# let a = ref (Cons (2, ref (Cons (3, ref Nil)))) ;;
val a : int mlist_internal ref =
  {contents = Cons (2, {contents = Cons (3, {contents = Nil})})}
```

2.
```
# let Cons (_hd, tl) = !a in
# let b = ref (Cons (1, a)) in
# tl := !b ;
# mhead (mtail (mtail b)) ;;
Lines 1-4, characters 0-23:
1 | let Cons (_hd, tl) = !a in
2 | let b = ref (Cons (1, a)) in
3 | tl := !b ;
4 | mhead (mtail (mtail b))...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Nil
- : int = 1
```

Note that the type `int mlist_internal ref` is equivalent to `int mlist`.

Solution to Exercise 160

```
# let mlength (lst : 'a mlist) : int =
#   let rec mlength' lst visited =
#     if List.memq lst visited then 0
#     else
#       match !lst with
#       | Nil -> 0
#       | Cons (_hd, tl) ->
#           1 + mlength' tl (lst :: visited)
#   in mlength' lst [] ;;
val mlength : 'a mlist -> int = <fun>
```

Solution to Exercise 161

```
# let rec mfirst (n: int) (mlst: 'a mlist) : 'a list =
#   if n = 0 then []
#   else match !mlst with
#        | Nil -> []
#        | Cons (hd, tl) -> hd :: mfirst (n - 1) tl ;;
val mfirst : int -> 'a mlist -> 'a list = <fun>
```

Solution to Exercise 162

```
# let rec alternating =
#   ref (Cons (1, ref (Cons (2, alternating)))) ;;
val alternating : int mlist =
  {contents = Cons (1, {contents = Cons (2, <cycle>)})}
```

Solution to Exercise 164 Let's start with insertion. It will be useful to have an auxiliary function that attempts to insert at a particular location, carrying out the probing if that location is already used for a different key.

```
let rec insert' dct target newvalue loc =
  (* fallen off the end of the array; error *)
  if loc >= D.size then raise Exit
  else
    match dct.(loc) with
    | Empty ->
        (* found an empty slot; fill it *)
        dct.(loc) <- Element {key = target;
                              value = newvalue}
    | Element {key; _} ->
        if key = target then
          (* found an existing pair for key; replace it *)
          dct.(loc) <- Element {key = target;
                                value = newvalue}
        else
          (* hash collision; reprobe *)
          insert' (succ loc) ;;
```

Now with this auxiliary function, we can implement insertion just by attempting to insert at the location given by the hash function.

```
let insert dct target newvalue =
  insert' dct target newvalue (D.hash_fn target) ;;
```

Of course, `insert'` is only needed in the context of `insert`. Why not make it a local function? Doing so also puts the definition of `insert'` in the scope of the arguments of `insert`. Since these never change in calls of `insert'`, we can drop them from the arguments list of `insert'`.

```
let insert dct target newvalue =
  let rec insert' loc =
    (* fallen off the end of the array; error *)
    if loc >= D.size then raise Exit
    else
      match dct.(loc) with
      | Empty ->
          (* found an empty slot; fill it *)
          dct.(loc) <- Element {key = target;
                                value = newvalue}
      | Element {key; _} ->
          if key = target then
            (* found an existing pair for key; replace it *)
```

```
            dct.(loc) <- Element {key = target;
                                  value = newvalue}
          else
            (* hash collision; reprobe *)
            insert' (succ loc) in
  insert' (D.hash_fn target) ;;
```

Next, we can look at the `member` function. Using the same approach, we get

```
       let member dct target =
         let rec member' loc =
           (* fallen off the end of the array; not found *)
           if loc >= D.size then false
           else
             match dct.(loc) with
             | Empty ->
                 (* found an empty slot; target not found *)
                 false
             | Element {key; _} ->
                 if key = target then


               (* found an existing pair for this key; target found *)
                  true
                else
                  (* hash collision; reprobe *)
                  member' (succ loc) in
        member' (D.hash_fn target) ;;
```

Perhaps you see the problem. The code is nearly identical, once the putative location for the target key is found. The same will be true for `lookup` and `remove`. Rather than reimplement this search process in each of the functions, we can abstract it into its own function, which we'll call `findloc`. This function returns the (optional) location (index) where a particular target key is already or should go, or `None` if no such location is found.

```
let findloc (dct : dict) (target : key) : int option =
  let rec findloc' loc =
    if loc >= D.size then None
    else
      match dct.(loc) with
      | Empty -> Some loc
      | Element {key; _} ->
         (if key = target then Some loc
          else findloc' (succ loc)) in
  findloc' (D.hash_fn target) ;;
```

Using `findloc`, implementation of the other functions becomes much simpler.

```
let member dct target =
  match findloc dct target with
```

```
    | None -> false
    | Some loc ->
         match dct.(loc) with
         | Empty -> false
         | Element {key; _} ->
            assert (key = target);
            true ;;

let lookup dct target =
  match findloc dct target with
  | None -> None
  | Some loc ->
     match dct.(loc) with
     | Empty -> None
     | Element {key; value} ->
        assert (key = target);
        Some value ;;

let insert dct target newvalue =
  match findloc dct target with
  | None -> raise Exit
  | Some loc ->
     dct.(loc) <- Element {key = target;
                           value = newvalue} ;;

let remove dct target =
  match findloc dct target with
  | None -> ()
  | Some loc -> dct.(loc) <- Empty ;;
```

Furthermore, the code is more maintainable because all of the details of collision handling are localized in the one `findloc` function. If we want to change to, say, quadratic probing, only that one function need be changed.

One might still think that there is more commonality among the hashtable functions than is even getting captured by `findloc`. It seems that in all cases, the result of the call to `findloc` is being tested for three cases: (i) no location is available, (ii) an empty location is found, or (iii) a non-empty location is found with the target key. Rather than perform this triage in all of the various functions, why not do so in `findloc` itself, which can be provided with appropriate functions, called CALLBACKS, for each of the cases. The following version does just this:

```
(* findloc dct key cb_unavailable cb_empty cb_samekey --
   Finds the proper location for the key in the dct, and
   calls the appropriate callback function:
     cb_unavailable -- no element available for this key
     cb_empty loc -- element available is empty at provided
       loc
     cb_samekey loc key value -- element available is non-empty
       at provided loc and has the given key and value
```

```
 *)
let findloc (dct : dict) (target : key)
            (cb_unavailable : unit -> 'a)
            (cb_empty : int -> 'a)
            (cb_samekey : int -> key -> value -> 'a)
          : 'a =
  let rec findloc' loc =
    if loc >= D.size then cb_unavailable ()
    else
      match dct.(loc) with
      | Empty -> cb_empty loc
      | Element {key; value} ->
         (if key = target then cb_samekey loc key value
          else findloc' (succ loc)) in
  findloc' (D.hash_fn target) ;;

let member dct target =
  findloc dct target
          (fun () -> false)
          (fun _ -> false)
          (fun _ _ _ -> true) ;;

let lookup dct target =
  findloc dct target
          (fun () -> None)
          (fun _ -> None)
          (fun _ _ value -> Some value) ;;

let insert dct target newvalue =
  let newelt = Element {key = target;
                        value = newvalue} in
  findloc dct target
          (fun () -> raise Exit)
          (fun loc -> dct.(loc) <- newelt)
          (fun loc _ _ -> dct.(loc) <- newelt) ;;

let remove dct target =
  findloc dct target
          (fun () -> ())
          (fun loc -> ())
          (fun loc _ _ -> dct.(loc) <- Empty) ;;
```

Solution to Exercise 167 Here's a simple implementation keeping an internal counter of allocations since the last reset.

```
#       module Metered : METERED = struct
#         (* internal counter of allocations since last reset *)
#         let constructor_count = ref 0
#
#         let reset () =
#           constructor_count := 0
#
#         let count () =
#           !constructor_count
```

```
#
#         let cons hd tl =
#           incr constructor_count;
#           hd :: tl
#
#         let pair first second =
#           incr constructor_count;
#           first, second
#       end ;;
module Metered : METERED
```

Solution to Exercise 168

```
# let rec zip (xs : 'a list)
#             (ys : 'b list)
#           : ('a * 'b) list =
#   match xs, ys with
#   | [], [] -> []
#   | [], _
#   | _, [] -> raise (Invalid_argument
#                       "zip: unequal length lists")
#   | xhd :: xtl, yhd :: ytl ->
#      Metered.cons (Metered.pair xhd yhd) (zip xtl ytl) ;;
val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

Notice that the constructors in the patterns, which are merely used to deconstruct values, are unchanged. Only the instances used to construct new values are replaced with their metered counterparts.

Solution to Exercise 169 A metered version of quicksort replaces all consing and pairing with the metered version. We've added a metered version of append as well.

```
# module MeteredQuickSort : SORT =
#   struct
#     (* simplify access to the metering *)
#     open Metered
#
#     (* append xs ys -- A metered version of the (@) append
#        function *)
#     let rec append (xs : 'a list) (ys : 'a list) : 'a list =
#       match xs with
#       | [] -> ys
#       | hd :: tl -> cons hd (append tl ys)
#
#     (* partition lt pivot xs -- Returns two lists
#        constituting all elements in `xs` less than (according
#        to `lt`) than the `pivot` value and greater than the
#        pivot `value`, respectively *)
#     let rec partition lt pivot xs =
#       match xs with
#       | [] -> pair [] []
#       | hd :: tl ->
#          let first, second = partition lt pivot tl in
```

```
#          if lt hd pivot then pair (cons hd first) second
#          else pair first (cons hd second)
#
#     (* sort lt xs -- Returns the sorted `xs` according to the
#        comparison function `lt` using the Quicksort algorithm *)
#     let rec sort (lt : 'a -> 'a -> bool)
#                  (xs : 'a list)
#                : 'a list =
#       match xs with
#       | [] -> []
#       | pivot :: rest ->
#          let first, second = partition lt pivot rest in
#          append (sort lt first)
#                 (append (cons pivot [])
#                         (sort lt second))
#   end ;;
module MeteredQuickSort : SORT
```

With the metered version in hand, we can see the allocations more clearly.

```
# Metered.reset () ;;
- : unit = ()
# MeteredQuickSort.sort (<)
#   [1; 3; 5; 7; 9; 2; 4; 6; 8; 10] ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; ...]
# Metered.count () ;;
- : int = 92
```

Solution to Exercise 171 New versions of the functions use `Lazy.force` instead of application to unit and the `lazy` keyword instead of wrapping a function. Notice that `first` is unchanged, as it delays and forces only through its use of the other functions.

```
let tail (s : 'a stream) : 'a stream =
  match Lazy.force s with
  | Cons (_hd, tl) -> tl ;;

let rec smap (f : 'a -> 'b) (s : 'a stream)
           : ('b stream) =
  lazy (Cons (f (head s), smap f (tail s))) ;;

let rec smap2 f s1 s2 =
  lazy (Cons (f (head s1) (head s2),
              smap2 f (tail s1) (tail s2))) ;;

let rec first (n : int) (s : 'a stream) : 'a list =
  if n = 0 then []
  else head s :: first (n - 1) (tail s) ;;
```

Solution to Exercise 172 We start with a function to form ratios of successive stream elements.

```
# let rec ratio_stream (s : float stream) : float stream =
#   lazy (Cons ((head (tail s)) /. (head s),
```

```
#               ratio_stream (tail s))) ;;
val ratio_stream : float stream -> float stream = <fun>
```

Now we can generate the stream of ratios for the Fibonacci sequence and find the required approximation:

```
# let golden_ratio_approx = ratio_stream (to_float fibs) ;;
val golden_ratio_approx : float stream = <lazy>
# within 0.000001 golden_ratio_approx ;;
- : float = 1.61803444782168193
```

Solution to Exercise 173

```
# let rec falses =
#   lazy (Cons (false, falses)) ;;
val falses : bool stream = <lazy>
```

Solution to Exercise 174 As demonstrated by the OCaml REPL type inference in the previous exercise, the type of `falses` is `bool stream`.

Solution to Exercise 175 Here is a recursive implementation of `trueat`:

```
# let rec trueat n =
#   if n = 0 then lazy (Cons (true, falses))
#   else lazy (Cons (false, trueat (n - 1))) ;;
val trueat : int -> bool stream = <fun>
```

Solution to Exercise 176 Here is a recursive implementation of `trueat`:

```
# let circnot : bool stream -> bool stream =
#   smap not ;;
val circnot : bool stream -> bool stream = <fun>
```

Note the use of the `smap` function and the use of partial application.

Solution to Exercise 177

```
# let circand : bool stream -> bool stream -> bool stream =
#   smap2 (&&) ;;
val circand : bool stream -> bool stream -> bool stream = <fun>
```

Solution to Exercise 178

```
# let circnand (s: bool stream) (t: bool stream) : bool stream =
#   circnot (circand s t) ;;
val circnand : bool stream -> bool stream -> bool stream = <fun>
```

Solution to Exercise 179

```
# class text (p : point) (s : string) : display_elt =
#     object (this)
#       inherit shape p
#       method draw = let (w, h) = G.text_size s in
```

```
#                     G.set_color this#get_color ;
#                     G.moveto (this#get_pos.x - w/2)
#                              (this#get_pos.y - h/2);
#                     G.draw_string s
#     end ;;
class text : point -> string -> display_elt
```

Solution to Exercise 180 There are many ways of implementing such functions. Here's one.

```
# let mono x = [x + 1] ;;
val mono : int -> int list = <fun>
# let poly x = [x] ;;
val poly : 'a -> 'a list = <fun>
# let need f =
#   match f 3 with
#   | [] -> []
#   | hd :: tl -> hd + 1 :: tl ;;
val need : (int -> int list) -> int list = <fun>
# need mono ;;
- : int list = [5]
# need poly ;;
- : int list = [4]
```

Solution to Exercise 181 The solution here makes good use of inheritance rather than reimplementation.

```
# class loud_counter : counter_interface =
#   object (this)
#     inherit counter as super
#     method! bump n =
#       super#bump n;
#       Printf.printf "State is now %d\n" this#get_state
#   end ;;
class loud_counter : counter_interface
```

The `bump` method is introduced with a `!` to make clear our intention to override the inherited method, and to avoid a warning.

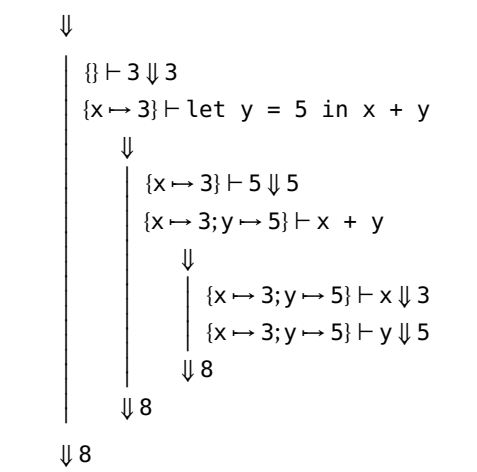
Solution to Exercise 182

```
# class type reset_counter_interface =
#   object
#     inherit counter_interface
#     method reset : unit
#   end ;;
class type reset_counter_interface =
  object
    method bump : int -> unit
    method get_state : int
    method reset : unit
  end
```

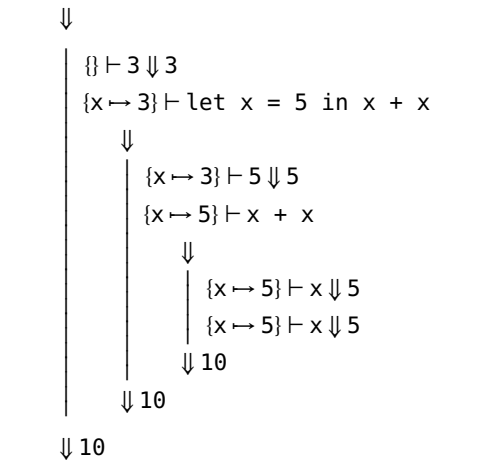
Solution to Exercise 183

```
# class loud_reset_counter : reset_counter_interface =
#   object (this)
#     inherit loud_counter
#     method reset =
#       this#bump (-this#get_state)
#   end ;;
class loud_reset_counter : reset_counter_interface
```

Solution to Exercise 184



Solution to Exercise 185

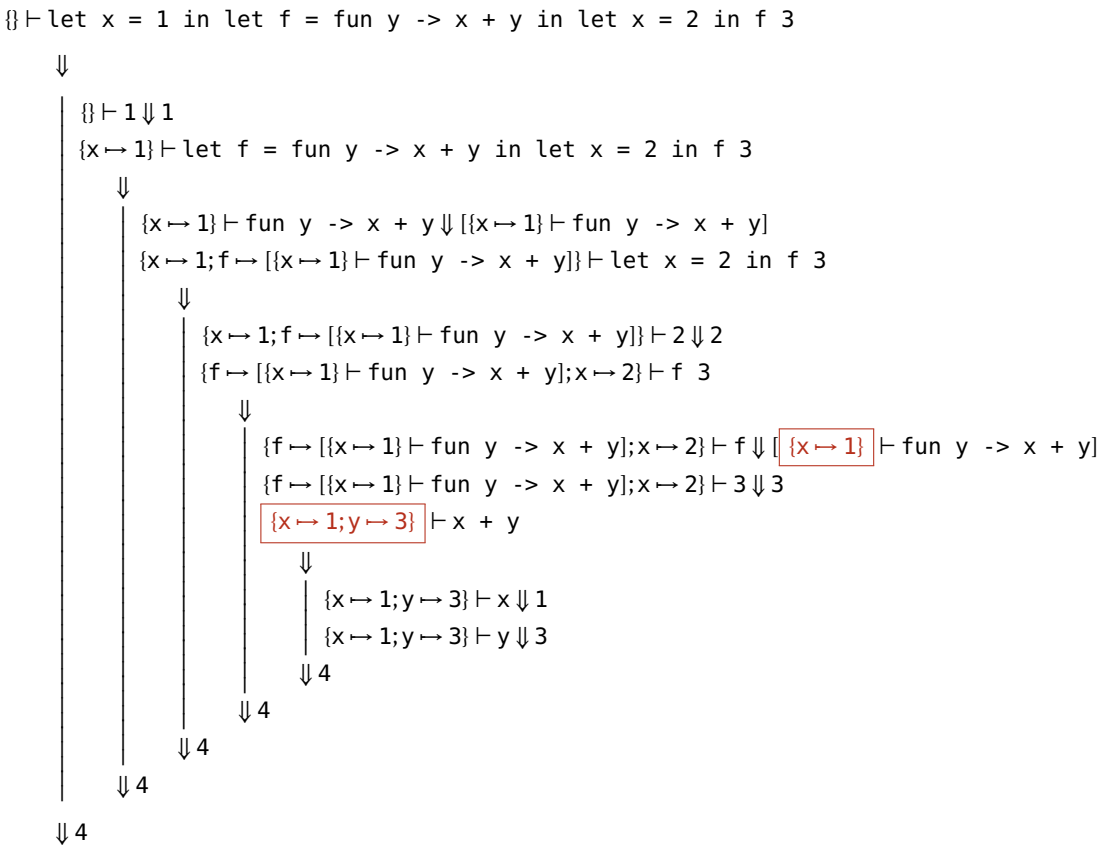


Solution to Exercise 186

- R_{fun}: "A function expression of the form `fun` x `->` B in an environment E evaluates to itself."
- R_{app}: "To evaluate an application of the form P Q in an environment E, first evaluate P in E to a function value of the form `fun` x

`->` B and Q in E to a value v_Q. Then evaluate the expression B in an environment that augments E with a binding of x to v_Q, resulting in a value v_B. The value of the full expression is then v_B."

Solution to Exercise 188 The derivation under a lexical environment semantics is as follows:



Notice that the body of the function is evaluated in an environment constructed by taking the lexical environment of the function (the first highlight in the derivation above) and augmenting it with the argument binding to form the environment in which to evaluate the body (the second highlight). In the lexical environment `x` has the value 1, so the entire expression evaluates to 4 rather than 5 (as under the substitution semantics as well).

Solution to Exercise 190 Only (4) evaluates to a different value under dynamic scoping. Under OCaml's lexical scoping, the `a` in the body of the `f` function is the lexically containing `a` that has value 2. The expression thus has value 12 under lexical scoping:

```
# let a = 2 in
# let f = (fun b -> a * b) in
```

```
# let a = 3 in
# f (f a) ;;
- : int = 12
```

Under dynamic scoping, the `a` in the body of the `f` function is the dynamically more recent `a` with value `3`. The value of the expression is thus `27` under dynamic scoping.

Solution to Exercise 191 Environment semantics rules for `true` and `false`, appropriate for both lexical and dynamic variants, are

$$E \vdash \texttt{true} \Downarrow \texttt{true} \qquad (R_{true})$$

$$E \vdash \texttt{false} \Downarrow \texttt{false} \qquad (R_{false})$$

Solution to Exercise 192 Environment semantics rules for `true` and `false`, appropriate for both lexical and dynamic variants, are

$$\frac{E \vdash C \Downarrow \texttt{true} \quad E \vdash T \Downarrow v_T}{E \vdash \texttt{if } C \texttt{ then } T \texttt{ else } F \Downarrow \Downarrow v_T} \qquad (R_{ifthen})$$

$$\frac{E \vdash C \Downarrow \texttt{false} \quad E \vdash F \Downarrow v_F}{E \vdash \texttt{if } C \texttt{ then } T \texttt{ else } F \Downarrow \Downarrow v_F} \qquad (R_{ifelse})$$

Solution to Exercise 193

$$\frac{E, S \vdash P \Downarrow l, S'}{E, S \vdash \texttt{!}\ P \Downarrow \Downarrow S'(l), S'} \qquad (R_{deref})$$

The rule can be glossed "to evaluate an expression of the form `!` P in environment E and store S, evaluate P in E and S to a location l and new store S'. The result is the value that l maps to in S' and new store S'."

Solution to Exercise 194 The following rule evaluates P to a unit, passing the side-effected store S' on for the evaluation of Q. The result of the sequencing is then the value and store resulting from the evaluation of Q.

$$\frac{E, S \vdash P \Downarrow (), S' \quad E, S' \vdash Q \Downarrow v_Q, S''}{E, S \vdash P\ \texttt{;}\ Q \Downarrow \Downarrow v_Q, S''} \qquad (R_{seq})$$

Solution to Exercise 195 We start by taking

$$\texttt{let rec } x \texttt{ = } D \texttt{ in } B$$

to be equivalent to

$$\texttt{let } x \texttt{ = ref unassigned in (} x \texttt{ := } D'\texttt{); } B'$$

where for brevity we abbreviate $D' \equiv D[x \mapsto !\,x]$, $B' \equiv B[x \mapsto !\,x]$, and $U \equiv \texttt{unassigned}$.

In order to develop the semantic rule for $\texttt{let rec } x \texttt{ = } D \texttt{ in } B$, we carry out a schematic derivation of its desugared equivalent:

$$\begin{array}{l}
E, S \vdash \texttt{let } x \texttt{ = ref } U \texttt{ in (} x \texttt{ := } D'\texttt{); } B' \\
\quad \Downarrow \\
\quad \left|\begin{array}{l}
E, S \vdash \texttt{ref } U \\
\quad \Downarrow \\
\quad \left|\begin{array}{l} E, S \vdash U \Downarrow U, S \end{array}\right. \\
\quad \Downarrow l, S\{l \mapsto U\} \\
E\{x \mapsto l\}, S\{l \mapsto U\} \vdash \texttt{(} x \texttt{ := } D'\texttt{); } B' \\
\quad \Downarrow \\
\quad \left|\begin{array}{l}
E\{x \mapsto l\}, S\{l \mapsto U\} \vdash x \texttt{ := } D' \\
\quad \Downarrow \\
\quad \left|\begin{array}{l}
E\{x \mapsto l\}, S\{l \mapsto U\} \vdash x \Downarrow l, S\{l \mapsto U\} \\
E\{x \mapsto l\}, S\{l \mapsto U\} \vdash \boxed{\text{D'}} \\
\quad \Downarrow \\
\quad \left|\begin{array}{l} \cdots \end{array}\right. \\
\quad \Downarrow v_D, S'
\end{array}\right. \\
\quad \Downarrow \texttt{()}, S'\{l \mapsto v_D\} \\
E\{x \mapsto l\}, S'\{l \mapsto v_D\} \vdash \boxed{\text{B'}} \\
\quad \Downarrow \\
\quad \left|\begin{array}{l} \cdots \end{array}\right. \\
\quad \Downarrow v_B, S''
\end{array}\right. \\
\quad \Downarrow v_B, S''
\end{array}\right. \\
\Downarrow v_B, S''
\end{array}$$

This schematic derivation is complete, except for the two highlighted subderivations for D' and B' respectively. Thus, we can define a semantic rule for the original construct $\texttt{let rec } x \texttt{ = } D \texttt{ in } B$ (now with abbreviations expanded) that incorporates these two subderivations as premises:

$$\begin{array}{l}
E, S \vdash \texttt{let rec } x \texttt{ = } D \texttt{ in } B \Downarrow \\
\qquad\qquad \left|\begin{array}{l}
E\{x \mapsto l\}, S\{l \mapsto \texttt{unassigned}\} \vdash D[x \mapsto !\,x] \Downarrow v_D, S' \\
E\{x \mapsto l\}, S'\{l \mapsto v_D\} \vdash B[x \mapsto !\,x] \Downarrow v_B, S''
\end{array}\right. \\
\qquad\qquad \Downarrow v_B, S''
\end{array}$$

(R_{letrec})

This is just the semantic rule presented in Section 19.6.1.

Solution to Exercise 196 The fold implementation from the solution to Exercise 96 is the following:

```
let rec foldbt (emptyval : 'b)
               (nodefn : 'a -> 'b -> 'b -> 'b)
               (t : 'a bintree)
             : 'b =
  match t with
  | Empty -> emptyval
  | Node (value, left, right) ->
      nodefn value (foldbt emptyval nodefn left)
                   (foldbt emptyval nodefn right) ;;
```

As a first step, let's isolate the two recursive calls.

```
let rec foldbt (emptyval : 'b)
               (nodefn : 'a -> 'b -> 'b -> 'b)
               (t : 'a bintree)
             : 'b =
  match t with
  | Empty -> emptyval
  | Node (value, left, right) ->
      let left' = foldbt emptyval nodefn left in
      let right' = foldbt emptyval nodefn right in
      nodefn value left' right' ;;
```

Now, we can compute the left subtree in a separate thread using `future`, remembering to `force` the value when it's needed.

```
# let rec foldbt_conc (emptyval : 'b)
#                     (nodefn : 'a -> 'b -> 'b -> 'b)
#                     (t : 'a bintree)
#                   : 'b =
#   match t with
#   | Empty -> emptyval
#   | Node (value, left, right) ->
#       let left' =
#         Future.future (foldbt_conc emptyval nodefn) left in
#       let right' = foldbt_conc emptyval nodefn right in
#       nodefn value (Future.force left') right' ;;
val foldbt_conc : 'b -> ('a -> 'b -> 'b -> 'b) -> 'a bintree -> 'b
  = <fun>
```

To demonstrate its operation, we can sum the values in a binary tree as per Exercise 98.

```
# let sum_bintree =
#   foldbt_conc 0 (fun v l r -> v + l + r) ;;
val sum_bintree : int bintree -> int = <fun>

# sum_bintree int_bintree ;;
- : int = 154
```

Solution to Exercise 197 Here's one such interleaving. We adjust the previous interleaving moving thread A's balance update to *after* thread B's update.

	thread A ($75 withdrawal)	*thread B ($50 withdrawal)*
1.	`if balance >= amt then begin`	
2.	`  let updated = balance - amt in`	
3.		`if balance >= amt then begin`
4.		`  let updated = balance - amt in`
5.		`  balance <- updated;`
6.	`  balance <- updated;`	
7.	`  amt`	
8.	…	`  amt`
		…

Solution to Exercise 198 Here's one such interleaving. We adjust the previous interleaving so that thread B verifies the balance adequacy (line 2) before thread A's update (line 3-4), but computes its updated balance afterwards (line 6).

	thread A ($75 withdrawal)	*thread B ($50 withdrawal)*
1.	`if balance >= amt then begin`	
2.		`if balance >= amt then begin`
3.	`  let updated = balance - amt in`	
4.	`  balance <- updated;`	
5.	`  amt`	
6.		`  let updated = balance - amt in`
7.	…	`  balance <- updated;`
8.		`  amt`
		…

Solution to Exercise 199 We wrap the computation of the critical region `f ()` in a `try` ⟨⟩ `with` to trap any exceptions and unlock on the way out.

```
# (* with_lock l f -- Run thunk `f` in context of acquired lock `l`,
#    unlocking on return or exceptions *)
# let with_lock (l : Mutex.t) (f : unit -> 'a) : 'a =
#   Mutex.lock l;
#   let res =
#     try f ()
#     with exn -> Mutex.unlock l;
#                 raise exn in
#   Mutex.unlock l;
#   res ;;
val with_lock : Mutex.t -> (unit -> 'a) -> 'a = <fun>
```

Bibliography

"A New York correspondent". To find Easter. *Nature*, XIII(338): 487, April 20 1876. URL https://books.google.com/books?id=H4ICAAAAIAAJ&pg=PA487#v=onepage&q&f=false.

Marianne Baudinet and David MacQueen. Tree pattern matching for ML (extended abstract). Technical report, Stanford University, 1985. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.9225&rep=rep1&type=pdf.

Jorge Luis Borges. Funes the memorious. In Donald A. Yates and James E. Irby, editors, *Labyrinths*. New Directions, New York, New York, 1962.

Richard G. Brown, Mary P. Dolciani, Robert H. Sorgenfrey, and William L. Cole. *Algebra: Structure and Method: Book 1*. McDougal Littell, Evanston, Illinois, California edition, 2000.

California Utilities Statewide Codes and Standards Team. Guest room occupancy controls: 2013 California building energy efficiency standards. Technical report, California Statewide Utility Codes and Standards Program, October 2011. URL https://www.energy.ca.gov/title24/2013standards/prerulemaking/documents/current/Reports/Nonresidential/Lighting_Controls_Bldg_Power/2013_CASE_NR_Guest_Room_Occupancy_Controls_Oct_2011.pdf.

Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML (Developing Applications With Objective CAML)*. O'Reilly Media, 2000. URL https://caml.inria.fr/pub/docs/oreilly-book/html/book-ora168.html.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936. ISSN 00029327, 10806377. URL http://www.jstor.org/stable/2371045.

Craig Conley. *Wye's Dictionary of Improbable Words: All-Vowel Words and All-Consonant Words*. CreateSpace Independent Publishing

Platform, 2009. ISBN 9781441455277. URL https://books.google.com/books?id=yk1j1WLh80QC.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, pages 137–150, San Francisco, CA, 2004. URL https://research.google.com/archive/mapreduce-osdi04.pdf.

Charles Antony Richard Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, February 1981. DOI: 10.1145/1283920.1283936. URL http://doi.acm.org/10.1145/1283920.1283936.

Donald E. Knuth. Von Neumann's first computer program. *ACM Computing Surveys*, 2(4):247–260, December 1970. ISSN 0360-0300. DOI: 10.1145/356580.356581. URL http://doi.acm.org/10.1145/356580.356581.

Donald E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261–301, December 1974. URL https://dl.acm.org/citation.cfm?id=356640.

Eriola Kruja, Joe Marks, Ann Blair, and Richard Waters. A short note on the history of graph drawing. In *International Symposium on Graph Drawing*, pages 272–286. Springer, 2001.

Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. DOI: 10.1093/comjnl/6.4.308. URL http://dx.doi.org/10.1093/comjnl/6.4.308.

Peter J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965. ISSN 0001-0782. DOI: 10.1145/363744.363749. URL http://doi.acm.org/10.1145/363744.363749.

Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966. ISSN 0001-0782. DOI: 10.1145/365230.365257. URL http://doi.acm.org/10.1145/365230.365257.

Harry Lewis and Rachel Zax. *Essential Discrete Mathematics for Computer Science*. Princeton University Press, Princeton, New Jersey, 2019.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. DOI: 10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.

Luigi Federico Menabrea and Ada Lovelace. Sketch of the analytical engine invented by charles babbage with notes by the translator. translated by ada lovelace. In Richard Taylor, editor, *Scientific Memoirs*, volume 3, pages 666–731. Richard and John E. Taylor, London, 1843. URL http://nrs.harvard.edu/urn-3:FHCL.HOUGH:33047333.

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 19 April 1965.

Christopher Null. Hello, i'm mr. null. my name makes me invisible to computers, November 2015. URL https://www.wired.com/2015/11/null/.

Plato. Phaedrus. In *Plato in Twelve Volumes*, volume I. William Heinemann Ltd., London, 1927. URL https://hdl.handle.net/2027/uc1.32106017211316?urlappend=%3Bseq=563. Translated by Harold N. Fowler.

Jean E. Sammet. The beginning and development of FORMAC (FORmula MAnipulation Compiler). In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 209–230, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. DOI: 10.1145/154766.155372. URL http://doi.acm.org/10.1145/154766.155372.

Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 9(2):305–316, 1924.

Alan Turing. Computability and λ-definability. *J. Symbolic Logic*, 2 (4):153–163, 12 1937. URL https://projecteuclid.org:443/euclid.jsl/1183383711.

Index

Note: The page numbers for primary occurrences of indexed items are typeset as ***123***, other occurrences as 123.

^ (string concatenation), 43

absolute value, ***393***
abstract data type, ***157***
abstract syntax, ***36***
abstract syntax tree, ***36***
abstraction, ***19***
abstraction barrier, ***157***
algebraic data types, ***137***
alias, ***252***
ambiguity, ***33***
anomaly, ***117***
anonymous function, ***63***
anonymous variable, ***80***
application, ***60***
argument, ***48***
artificial intelligence, ***382***
associativity, ***35***
asymptotic, ***230***
atomic type, ***46***

Backus-Naur form, ***32***
backwards application, ***144***
best_first search, ***384***
big-*O* notation, ***230***
bignums, ***380***
binary operators, ***34***
binary tree, ***151***
bind, ***205***
binding, ***51***, ***369***
binding construct, ***51***
bound, ***205***
box and arrow diagrams, ***251***
breadth-first search, ***384***
buffer over-reads, ***253***
buffer overflows, ***253***
busy waiting, ***349***

callbacks, ***462***
characters, ***43***
checksum, ***103***
Church, Alonzo, 25–26, 230, 390
Church-Turing thesis, ***25***, 361
class, ***311***
class interface, ***311***
closed form, ***236***
closed hashing, ***267***
closure, ***330***
comments, ***38***
comparison operators, ***43***
compartmentalization, *see* edict of compartmentalization
composition, ***113***
computus, ***67***
concatenation
  string, 43
concrete syntax, ***36***
concurrent computation, ***343***
conditional, ***44***
conjunction, ***395***
cons, ***83***
control, ***386***
critical region, ***357***
Curry, Haskell, 302
currying, ***61***
cycle, ***260***

decomposition, *see* edict of decomposition
definiendum, ***51***
definiens, ***51***
definition, ***57***
delay, ***286***
denotational semantics, ***197***
dependent type systems, ***72***
depth-first search, ***384***
dequeuing, ***155***
dereference, ***249***
design, ***25***
difference, ***396***
Dikstra, Edsger, 71
disjunction, ***395***
distance, ***395***
divide-and-conquer, ***243***
donation game, ***377***
dynamic environment, ***328***
dynamic environment semantics, ***328***
dynamically typed, ***45***

eager, ***285***

Easter, 67, 71
edges, ***385***
edict
  of compartmentalization, 158, 257, 314
  of decomposition, 105, 303, 310
  of intention, 38, 52, 65, 71, 80, 107, 110, 120, 146, 170, 211, 422
  of irredundancy, 59, 60, 67, 95, 97, 107, 108, 110, 112, 140, 173, 177, 211, 313, 318
  of prevention, 46, 147, 148, 157, 358
empty set, ***396***
enqueueing, ***155***
environment, ***322***
error value, ***118***
Euclid's algorithm, ***24***
evaluation, ***41***
exception, ***122***
expressions, ***31***
extensional, ***396***

factorial, ***69***, 389
Fibonacci sequence, ***72***, 291
field, ***90***
field punning, ***91***
filter, ***102***
first-class values, ***49***
first-in-first-out, ***155***
fold, ***100***
force, ***286***
force-directed graph layout, ***386***
forces, ***386***
fork, ***346***
formal, ***196***
formal verification, ***72***
free, ***205***
free variables, ***206***
function, ***21***, ***48***
  value of, 48
functional programming language, ***49***
functors, ***178***
future, ***351***

garbage, ***253***
garbage collection, ***253***
Gilles Kahn, 196
goal state, ***382***
golden ratio, ***43***, 296, 380
grammar, ***32***
graph, ***385***
graph drawing, ***385***
greatest common divisor, ***22***
greedy search, ***384***

hash collision, ***267***
hash function, ***267***
hash table, ***267***
Haskell, 302
head, ***83***
heartbleed, ***253***
Heron's formula, ***396***
higher-order functional programming, ***49***
higher-order functions, ***49***
Hoare, C. A. R., 71
Hope, 137
hypotenuse, ***395***

identity, ***397***
identity function, ***109***
imperative programming, ***248***
implementation, ***157***
implicitly typed, ***48***
in-band signaling, ***118***
in-place, ***278***
infix, ***62***
inherit, ***313***
initial state, ***382***
insertion sort, ***225***
instance variables, ***311***
instantiation, ***311***
intensional, ***396***
intention, *see* edict of intention
interpreter, ***29***
intersection, ***396***
interval, ***190***
invariant, ***156***
invocation, ***311***
irredundancy, *see* edict of irredundancy
ISWIM, 302
iterated prisoner's dilemma, ***378***

judgement, ***197***

lambda calculus, ***25***, 390, 393
Landin, Peter, 64, 71, 301, 302
large-step, ***197***
lazy evaluation, ***286***
left associative, ***35***
length, ***86***
lexical environment, ***328***
lexical environment semantics, ***328***
library modules, ***188***
linear probing, ***267***
Lists, ***83***
literals, ***29***
local, ***55***
local open, ***160***
location, ***335***
lock, ***356***
loop, ***21***
Luhn check, ***103***

magic number, ***407***
map, ***96***
MapReduce, 102
Marx, Groucho, 34
masses, ***385***
median, ***117***
membership, ***397***
memoization, ***292***

memory corruption, ***253***
memory leak, ***253***
merge sort, ***226***
metacircular interpreter, ***219***, ***361***
metalanguage, ***200***
methods, ***311***
ML, 137
modules, ***158***
monad, ***122***
move, ***382***
mutex locks, ***357***

natural semantics, ***196***
Naur, Peter, 71
negation, ***395***, ***397***
neighbor, ***383***
nil, ***83***
nodes, ***385***
numerical solution, ***380***

object, ***311***
object language, ***200***
object-oriented, ***310***
object-oriented programming, ***304***
OCaml, ***26***
open expressions, ***209***
operational semantics, ***197***
operators
  defining new, 114
option poisoning, ***122***
optional chaining, ***122***
order of operations, ***35***
ordered type, ***162***

parallel computation, ***343***
parentheses, ***35***
parse trees, ***36***
partial application, ***98***
partial sum, ***295***
physical equality, ***252***
π, 51
π, 396
pointers, ***252***
polymorphic function, ***109***
polymorphic type, ***109***
polymorphism, ***108***
postfix, ***83***
precedence, ***35***
prefix, ***62***
premature optimization, ***223***
preorder, ***152***
prevention, *see* edict of prevention
prisoner's dilemma, ***377***
procedural programming, ***271***
procedure, ***251***
prompt, ***29***
pure, ***247***
pyramid of doom, ***122***
Pythagorus's theorem, ***395***

quadratic probing, ***270***
queue, ***155***
quicksort, 71, 227, ***278***

race condition, ***344***
read-after-write dependency, ***344***
read-eval-print loop, ***30***
record, ***90***
recur, 22
recurrence equations, ***235***
recurse, *see* recur
recursion, ***22***
refactoring, ***105***
reference, ***249***
reference types, ***249***
REPL, ***30***, *see* read-eval-print loop
rest length, ***386***
reverse index, ***166***
right associative, ***35***
right triangle, ***395***

scope, ***55***
search, ***383***
search tree, ***383***
semantics, ***33***
semiperimeter, ***54***, ***396***
sequential computation, ***343***
series acceleration, ***296***
shadowing, ***56***
sharing constraints, ***175***
side effect, ***247***
signature, ***157***
slope, ***395***
small-step semantics, ***197***
sorting, ***224***
space efficiency, ***271***
stack frame, ***273***
stages, ***361***
state, ***382***
state variable, ***21***
statically typed, ***45***
store, ***335***
strategy, ***378***
stream, ***287***
strings
  concatenation of, 43
strongly typed, ***45***
structural equality, ***252***
structure-driven programming, ***88***
subclass, ***313***
substitution semantics, ***204***
superclass, ***313***
supertype, ***318***
symbolic solution, ***380***
syntactic sugar, ***64***
syntax, ***31***

tail, ***83***
tail recursion, ***275***
  optimization, ***275***
Taylor series, ***294***
tesseract, ***99***
tesseractic numbers, ***99***
thread, ***344***
thunk, ***292***

timeout, ***298***
tit-for-tat, ***378***
truth values, ***43***
tuple, ***77***
Turing Award, 21, 26, 71, 157, 223, 379, 381
Turing machine, ***21***, 21, 25
Turing, Alan, 21, 25
type constructor, ***77***
type expressions, ***46***
type inference, ***48***
type variables, ***109***
    weak, 114
typed, ***44***
typing, ***47***

unfolding, ***236***
union, ***396***
unit testing, ***73***
University of Edinburgh, 137
update, ***249***

value
    of a function, ***48***
value constructor, ***77***
values, ***198***
variable capture, ***215***
variant type, ***138***

weak type variables, ***114***
weighted sum, ***114***
well-founded recursion, ***71***
widgets, ***303***
wild-card, ***83***
worst-case complexity, ***225***
write-after-read dependency, ***344***
write-after-writedependency, ***344***

Image Credits

Figure 1.1. Trial model of a part of the Analytical Engine, built by Babbage, as displayed at the Science Museum (London), Bruno Barral. CC-BY-SA 2.5, courtesy of Wikipedia. 20

Figure 1.3. Passport photo of Alan Turing. Public domain; courtesy of Wikipedia. 21

Figure 1.4. T. L. Heath, translator. 1908. The Thirteen books of Euclid's Elements. Cambridge: Cambridge University Press, page 298. 23

Figure 1.6. Photo of Alonzo Church, copyright (presumed) Princeton University. Used under fair use, courtesy of Wikipedia. . . 25

Figure 1.7. Photo of Robin Milner, University of Cambridge. 26

Figure 4.2. Explosion of first Ariane 5 flight, June 4, 1996. Copyright European Space Agency; used by permission. 44

Figure 6.1. Photo of Moses Schönfinkel in 1922. CC-BY-SA 4.0, courtesy of Wikimedia Commons. 61

Figure 6.2. Photo of Haskell B. Curry, Gleb.svechnikov. CC-BY-SA 4.0, courtesy of Wikipedia. 61

Figure 7.1. Photo of Marianne Baudinet, courtesy of LinkedIn. . . 79

Figure 9.1. Image of J. Roger Hindley, cached by Internet Archive from University of Swansea Mathematics Department, August 6, 2002. 110

Figure 12.2. Photo of Barbara Liskov by Kenneth C. Zirkel. CC-BY-SA 3.0; courtesy of Wikimedia Commons. 157

Figure 12.4. Alexander Calder, "L'empennage", 1953. Photo by Finlay McWalter, courtesy of Wikimedia Commons, used by permission. CC-by-sa 3.0. 191

Figure 13.1. Portrait of Gottfried Wilhelm Leibniz by Christoph Bernhard Francke in the Herzog Anton Ulrich-Museum Braunschweig. Public domain, courtesy of Wikimedia Commons. 196

Figure 13.2. Photograph of Gilles Kahn by F. Jannin. Copyright INRIA. 196

Figure 14.1. Portrait of Donald Knuth, copyright 2010 Photographic Unit, University of Glasgow. 223
Figure 15.2. Heartbleed logo by Leena Snidate/Codenomicon, in the public domain by CC0 dedication, courtesy of Wikipedia. 253
Figure 17.3. Small copy of a portrait of Brook Taylor, the English mathematician. 1720. National Portrait Gallery (London). Public domain, courtesy of Wikipedia. 294
Figure 17.7. Photo of Peter Landin, courtesy of Wikipedia. 301
Figure 18.3. Photo of Alan Kay, by Alan Kay, used under a CC-by 2.0 Generic license. Courtesy of Wikimedia Commons. Photo of Adele Goldberg, by Terry Hancock, used under a CC-by-sa 2.5 Generic license. Courtesy of Wikimedia Commons. Photo of Dan Ingalls, used under a CC-by-sa 3.0 Unported license. Courtesy of Wikimedia Commons. 304
Figure 20.1. Figure from Moore (1965). 341
Figure 20.1. Computer performance data from Wikipedia. Used by permission (CC-BY-SA). 342
Figure A.1. Portrait of Whitfield Diffie by Duncan Hall, courtesy of Wikimedia, licensed under CC BY-SA 4.0. Portrait of Martin Hellman, courtesy of Wikimedia, licensed under CC BY-SA 3.0. 379
Figure A.2. Daguerrotype of Augusta Ada King, Countess of Lovelace by Antoine Claudet about 1843. Work in the public domain. 380
Figure A.3. Photograph of John McCarthy. Used by implicit permission of the author. 381
Figure A.4. Scanned photograph of Jean Sammet. Copyright 2018 Mount Holyoke College, used under fair use. 381
Figure B.0. Richard G. Brown, Mary P. Dolciani, Robert H. Sorgenfrey, and William L. Cole. Algebra: Structure and Method: Book 1: California Edition. Evanston Illinois: McDougal Littell, 2000. Page 379. 390
Figure B.2. Portrait of Leonhard Euler (1707-1783). Jakob Emanuel Handmann. Public domain; courtesy of Wikipedia. . 390
Figure B.2. Selection from Leonhard Euler [Leonh. Eulero]. 1734-5. Additamentum ad Dissertationem de Infinitis Curvis Eiusdem Generis [An Addition to the Dissertation Concerning an Infinite Number of Curves of the Same Kind]. In *Commentarii Academiae Scientarium Imperialis Petropolitanae [Memoirs of the Imperial Academy of Sciences in St. Petersburg]*, Volume VII (1734-35). Petropoli, Typis Academiae, 1740. Photo of the author. 391
Figure C.0. XKCD comic 1513, “Code Quality”, by Randall Munro. CC-BY-NC 2.5, courtesy of Randall Munro. 399