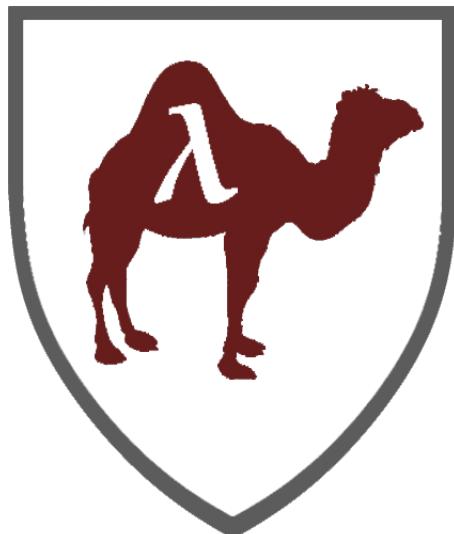


STUART M. SHIEBER

PROGRAMMING  
WELL:  
ABSTRACTION AND  
DESIGN IN  
COMPUTATION





©2025 Stuart M. Shieber. All rights reserved for the time being, though the intention is for this document to eventually be licensed under a CC license. In the meantime, please do not cite, quote, or redistribute.

*CI Build: 89-1eee1c9 (Mon Jan 13 22:22:06 UTC 2025)*

*Commit 1eee1c9 from Mon Jan 13 17:08:39 2025 -0500 by CS51 Bot.*



# *Contents*

<b>Preface</b>	<b>13</b>
<b>1 Introduction</b>	<b>19</b>
1.1 An extended example: greatest common divisor . . . . .	21
1.2 Programming as design . . . . .	24
1.3 The OCaml programming language . . . . .	26
1.4 Tools and skills for design . . . . .	28
<b>2 A Cook's tour of OCaml</b>	<b>29</b>
<b>3 Expressions and the linguistics of programming languages</b>	<b>31</b>
3.1 Specifying syntactic structure with rules . . . . .	31
3.2 Disambiguating ambiguous expressions . . . . .	34
3.3 Abstract and concrete syntax . . . . .	36
3.4 Expressing your intentions . . . . .	37
3.4.1 Commenting . . . . .	38
<b>4 Values and types</b>	<b>41</b>
4.1 OCaml expressions have values . . . . .	41
4.1.1 Integer values and expressions . . . . .	41
4.1.2 Floating point values and expressions . . . . .	42
4.1.3 Character and string values . . . . .	43
4.1.4 Truth values and expressions . . . . .	43
4.2 OCaml expressions have types . . . . .	44
4.2.1 Type expressions and typings . . . . .	46
4.3 The unit type . . . . .	48
4.4 Functions are themselves values . . . . .	48
<b>5 Naming and scope</b>	<b>51</b>
5.1 Variables are names for values . . . . .	51
5.2 The type of a let-bound variable can be inferred . . . . .	52
5.3 let expressions are expressions . . . . .	52
5.4 Naming to avoid duplication . . . . .	53
5.5 Scope . . . . .	55

5.6 Global naming and top-level let . . . . .	57
<b>6 Functions</b>	<b>59</b>
6.1 Function application . . . . .	60
6.2 Multiple arguments and currying . . . . .	61
6.3 Defining anonymous functions . . . . .	62
6.4 Named functions . . . . .	63
6.4.1 Compact function definitions . . . . .	64
6.4.2 Providing typings for function arguments and outputs . . . . .	65
6.5 Function abstraction and irredundancy . . . . .	67
6.6 Defining recursive functions . . . . .	69
6.7 Unit testing . . . . .	72
6.8 Supplementary material . . . . .	76
<b>7 Structured data and composite types</b>	<b>77</b>
7.1 Tuples . . . . .	77
7.2 Pattern matching for decomposing data structures . . . . .	79
7.2.1 Advanced pattern matching . . . . .	82
7.3 Lists . . . . .	83
7.3.1 Some useful list functions . . . . .	85
7.4 Records . . . . .	90
7.4.1 Field selection . . . . .	92
7.5 Comparative summary . . . . .	92
<b>8 Higher-order functions and functional programming</b>	<b>95</b>
8.1 The map abstraction . . . . .	95
8.2 Partial application . . . . .	97
8.3 The fold abstraction . . . . .	100
8.4 The filter abstraction . . . . .	102
8.5 Problem section: Credit card numbers and the Luhn check . . . . .	103
8.6 Supplementary material . . . . .	105
<b>9 Polymorphism and generic programming</b>	<b>107</b>
9.1 Polymorphism . . . . .	108
9.2 Polymorphic map . . . . .	109
9.3 Regaining explicit types . . . . .	110
9.4 The List library . . . . .	112
9.5 Problem section: Function composition . . . . .	113
9.6 Weak type variables . . . . .	114
9.7 Supplementary material . . . . .	115
<b>10 Handling anomalous conditions</b>	<b>117</b>

10.1 A non-solution: Error values . . . . .	118
10.2 Option types . . . . .	119
10.2.1 Option poisoning . . . . .	121
10.3 Exceptions . . . . .	122
10.3.1 Handling exceptions . . . . .	125
10.3.2 Zipping lists . . . . .	126
10.3.3 Declaring new exceptions . . . . .	130
10.4 Options or exceptions? . . . . .	131
10.5 Unit testing with exceptions . . . . .	132
10.6 Supplementary material . . . . .	134
<b>11 Algebraic data types</b>	<b>137</b>
11.1 Built-in composite types as algebraic types . . . . .	139
11.2 Example: Boolean document search . . . . .	140
11.3 Example: Dictionaries . . . . .	146
11.4 Example: Arithmetic expressions . . . . .	149
11.5 Problem section: Binary trees . . . . .	151
11.6 Supplementary material . . . . .	153
<b>12 Abstract data types and modular programming</b>	<b>155</b>
12.1 Modules . . . . .	158
12.2 A queue module . . . . .	159
12.3 Signatures hide extra components . . . . .	162
12.4 Modules with polymorphic components . . . . .	165
12.5 Abstract data types and programming for change . . . . .	166
12.5.1 A string set module . . . . .	169
12.5.2 A generic set signature . . . . .	172
12.5.3 A generic set implementation . . . . .	176
12.6 A dictionary module . . . . .	181
12.7 Alternative methods for defining signatures and modules	185
12.7.1 Set and dictionary modules . . . . .	186
12.8 Library Modules . . . . .	188
12.9 Problem section: Image manipulation . . . . .	189
12.10 Problem section: An abstract data type for intervals . . .	190
12.11 Problem section: Mobiles . . . . .	191
12.12 Supplementary material . . . . .	194
<b>13 Semantics: The substitution model</b>	<b>195</b>
13.1 Semantics of arithmetic expressions . . . . .	197
13.2 Semantics of local naming . . . . .	201
13.3 Defining substitution . . . . .	204
13.3.1 A problem with variable scope . . . . .	204
13.3.2 Free and bound occurrences of variables . . . . .	205
13.3.3 Handling variable scope properly . . . . .	206

13.4	Implementing a substitution semantics . . . . .	207
13.4.1	Implementing substitution . . . . .	208
13.4.2	Implementing evaluation . . . . .	209
13.5	Problem section: Semantics of booleans and conditionals	212
13.6	Semantics of function application . . . . .	212
13.6.1	More on capturing free variables . . . . .	214
13.7	Substitution semantics of recursion . . . . .	218
13.8	Supplementary material . . . . .	221
<b>14</b>	<b>Efficiency, complexity, and recurrences</b>	<b>223</b>
14.1	The need for an abstract notion of efficiency . . . . .	224
14.2	Two sorting functions . . . . .	225
14.3	Empirical efficiency . . . . .	227
14.4	Big- $O$ notation . . . . .	229
14.4.1	Informal function notation . . . . .	231
14.4.2	Useful properties of $O$ . . . . .	232
14.4.3	Big- $O$ as the metric of relative growth . . . . .	233
14.5	Recurrence equations . . . . .	234
14.5.1	Solving recurrences by unfolding . . . . .	236
14.5.2	Complexity of reversing a list . . . . .	237
14.5.3	Complexity of reversing a list with accumulator .	239
14.5.4	Complexity of inserting in a sorted list . . . . .	240
14.5.5	Complexity of insertion sort . . . . .	241
14.5.6	Complexity of merging lists . . . . .	242
14.5.7	Complexity of splitting lists . . . . .	243
14.5.8	Complexity of divide and conquer algorithms .	243
14.5.9	Complexity of mergesort . . . . .	244
14.5.10	Basic Recurrence patterns . . . . .	245
14.6	Problem section: Complexity of the Luhn check . . .	246
14.7	Supplementary material . . . . .	246
<b>15</b>	<b>Mutable state and imperative programming</b>	<b>247</b>
15.1	References . . . . .	249
15.1.1	Reference operator types . . . . .	250
15.1.2	Boxes and arrows . . . . .	251
15.1.3	References and pointers . . . . .	252
15.2	Other primitive mutable data types . . . . .	254
15.2.1	Mutable record fields . . . . .	254
15.2.2	Arrays . . . . .	255
15.3	References and mutation . . . . .	255
15.4	Mutable lists . . . . .	258
15.5	Imperative queues . . . . .	260
15.5.1	Method 1: List references . . . . .	262
15.5.2	Method 2: Two stacks . . . . .	262

15.5.3 Method 3: Mutable lists . . . . .	264
15.6 Hash tables . . . . .	266
15.7 Conclusion . . . . .	270
15.8 Supplementary material . . . . .	270
<b>16 Loops and procedural programming</b>	<b>271</b>
16.1 Loops require impurity . . . . .	272
16.2 Recursion versus iteration . . . . .	273
16.2.1 Saving stack space . . . . .	273
16.2.2 Tail recursion . . . . .	274
16.3 Saving data structure space . . . . .	275
16.3.1 Problem section: Metering allocations . . . . .	276
16.3.2 Reusing space through mutable data structures .	277
16.4 In-place sorting . . . . .	278
16.5 Supplementary material . . . . .	283
<b>17 Infinite data structures and lazy programming</b>	<b>285</b>
17.1 Delaying computation . . . . .	285
17.2 Streams . . . . .	287
17.2.1 Operations on streams . . . . .	288
17.3 Lazy recomputation and thunks . . . . .	291
17.3.1 The Lazy Module . . . . .	293
17.4 Application: Approximating $\pi$ . . . . .	294
17.5 Problem section: Circuits and boolean streams . . . . .	296
17.6 A unit testing framework . . . . .	297
17.7 A brief history of laziness . . . . .	301
17.8 Supplementary material . . . . .	302
<b>18 Extension and object-oriented programming</b>	<b>303</b>
18.1 Drawing graphical elements . . . . .	304
18.2 Objects introduced . . . . .	308
18.3 Object-oriented terminology and syntax . . . . .	311
18.4 Inheritance . . . . .	313
18.4.1 Overriding . . . . .	315
18.5 Subtyping . . . . .	316
18.6 Problem section: Object-oriented counters . . . . .	319
18.7 Supplementary material . . . . .	320
<b>19 Semantics: The environment model</b>	<b>321</b>
19.1 Review of substitution semantics . . . . .	321
19.2 Environment semantics . . . . .	322
19.2.1 Dynamic environment semantics . . . . .	323
19.2.2 Lexical environment semantics . . . . .	330
19.3 Conditionals and booleans . . . . .	331

19.4	Recursion . . . . .	332
19.5	Implementing environment semantics . . . . .	334
19.6	Semantics of mutable storage . . . . .	335
19.6.1	Lexical environment semantics of recursion . . .	339
19.7	Supplementary material . . . . .	340
<b>20</b>	<b>Concurrency</b>	<b>341</b>
20.1	Sequential, concurrent, and parallel computation . . . . .	342
20.2	Dependencies . . . . .	343
20.3	Threads . . . . .	344
20.4	Interthread communication . . . . .	347
20.5	Futures . . . . .	350
20.6	Futures are not enough . . . . .	352
20.7	Locks . . . . .	356
20.7.1	Abstracting lock usage . . . . .	358
20.8	Deadlock . . . . .	359
<b>A</b>	<b>Final project: Implementing MiniML</b>	<b>361</b>
A.1	Overview . . . . .	361
A.1.1	Grading and collaboration . . . . .	362
A.1.2	A digression: How is this project different from a problem set? . . . . .	362
A.2	Implementing a substitution semantics for MiniML . . . . .	363
A.3	Implementing an environment semantics for MiniML . . . . .	368
A.4	Extending the language . . . . .	371
A.4.1	Extension ideas . . . . .	371
A.4.2	A lexically scoped environment semantics . . . . .	372
A.4.3	The MiniML parser . . . . .	375
A.5	Submitting the project . . . . .	375
A.6	Alternative final projects . . . . .	376
<b>A</b>	<b>Problem sets</b>	<b>377</b>
A.1	The prisoners' dilemma . . . . .	377
A.2	Higher-order functional programming . . . . .	378
A.3	Bignums and RSA encryption . . . . .	379
A.4	Symbolic differentiation . . . . .	380
A.5	Ordered collections . . . . .	381
A.6	The search for intelligent solutions . . . . .	382
A.6.1	Search problems . . . . .	382
A.7	Refs, streams, and music . . . . .	384
A.8	Force-directed graph drawing . . . . .	384
A.8.1	Background . . . . .	385
A.9	Simulating an infectious process . . . . .	387
A.9.1	The simulation . . . . .	387

<b>B Mathematical background and notations</b>	<b>389</b>
B.1 Functions . . . . .	389
B.1.1 Defining functions with equations . . . . .	389
B.1.2 Notating function application . . . . .	390
B.1.3 Alternative mathematical notations for functions and their application . . . . .	390
B.1.4 The lambda notation for functions . . . . .	393
B.2 Summation . . . . .	394
B.3 Logic . . . . .	395
B.4 Geometry . . . . .	395
B.5 Sets . . . . .	396
B.6 Equality and identity . . . . .	397
<b>C A style guide</b>	<b>399</b>
C.1 Formatting . . . . .	400
C.1.1 No tab characters . . . . .	400
C.1.2 80 column limit . . . . .	400
C.1.3 No needless blank lines . . . . .	400
C.1.4 Use parentheses sparingly . . . . .	400
C.1.5 Delimiting code used for side effects . . . . .	401
C.1.6 Spacing for operators and delimiters . . . . .	402
C.1.7 Indentation . . . . .	403
C.2 Documentation . . . . .	404
C.2.1 Comments before code . . . . .	404
C.2.2 Comment length should match abstraction level	405
C.2.3 Multi-line commenting . . . . .	405
C.3 Naming and declarations . . . . .	405
C.3.1 Naming conventions . . . . .	405
C.3.2 Use meaningful names . . . . .	406
C.3.3 Constants and magic numbers . . . . .	407
C.3.4 Function declarations and type annotations . .	407
C.3.5 Avoid global mutable variables . . . . .	408
C.3.6 When to rename variables . . . . .	408
C.3.7 Order of declarations in a module . . . . .	408
C.4 Pattern matching . . . . .	409
C.4.1 No incomplete pattern matches . . . . .	409
C.4.2 Pattern match in the function arguments when possible . . . . .	409
C.4.3 Pattern match with as few <code>match</code> expressions as necessary . . . . .	410
C.4.4 Misusing <code>match</code> expressions . . . . .	410
C.4.5 Avoid using too many projection functions . .	411
C.5 Verbosity . . . . .	411

C.5.1 Reuse code where possible . . . . .	411
C.5.2 Do not abuse <code>if</code> expressions . . . . .	412
C.5.3 Don't rewrap functions . . . . .	412
C.5.4 Avoid computing values twice . . . . .	413
C.6 Other common infelicities . . . . .	413
<b>D Solutions to selected exercises</b>	<b>415</b>
<b>Bibliography</b>	<b>475</b>
<b>Index</b>	<b>479</b>
<b>Image Credits</b>	<b>484</b>

# Preface

This book began as the notes for Computer Science 51, a second semester course in programming at Harvard College, which follows the legendary CS50 course that ably introduces some half of all Harvard undergraduate students to computer programming, and in its online HarvardX version [CS50x](#) has benefited hundreds of thousands of other students.

Students just learning to program, like those in CS50, typically view the end product of programming as a program that works – that “gets the right answer”. Once such a program is in hand, the student thinks, the programmer’s job is done. This book was developed to move students past this view of programming, to focus on programming *well*, regarding programming not as a transaction but as an art and a craft.

The book emphasizes the role of abstraction and abstraction mechanisms in engendering a design space in which good programs can be constructed. These abstraction mechanisms are associated with and enable the major programming paradigms – first- and higher-order functional programming, structure-driven programming, generic programming, modular programming, imperative programming, procedural programming, lazy programming, object-oriented programming, and concurrent programming. By expanding the student’s armamentarium of abstraction mechanisms, this design space grows as well, making possible programs that are better along multiple dimensions – readability, maintainability, succinctness, efficiency, testability, and, most importantly but ineffably, beauty.

## Aims

In developing the book, I had in mind several aims.

*Explicit presentation of general principles.* I introduce a small set of very general software engineering principles – presented as “edicts” in the text – and make frequent reference to them throughout the text to tie together more particular software engineering ideas.

The programming edicts:

- *Edict of intention:* Make your intentions clear.
- *Edict of irredundancy:* Never write the same code twice.
- *Edict of decomposition:* Carve software at its joints.
- *Edict of prevention:* Make the illegal inexpressible.
- *Edict of compartmentalization:* Limit information to those with a need to know.

I emphasize other general principles, such as the separation of concepts and paradigms from languages, and programming as art and craft, not a science.

*Use of formal methods and notations.* Facility with notation is the essence of mathematical maturity, and a strong correlate to computational thinking. I explicitly motivate the use of formal notation, and introduce notations for many of the core ideas in the book – syntax, semantics, complexity – both to emphasize rigorous thinking and to provide practice in handling notations. Use of this kind of notation is ubiquitous in computer science (Guy Steele has referred to this kind of notation, which he calls “computer science metanotation”, as “the most popular programming language in computer science”) though it is rarely introduced explicitly. For that reason alone, an introductory presentation of these notations is valuable for the early computer science student.

*Provenance of ideas.* Rather than presenting computational ideas or techniques as disconnected from history, I emphasize the provenance of these ideas, highlighting the role of real people in their development and promulgation and providing acculturation into some of the intellectual history of computer science. Special attention is given wherever appropriate to the role of women in developing the ideas.

*Emphasis on reliable methods.* Emphasis is placed on using modern methods for generating reliable programs by having the computer take on much of the work, in particular, strong static typing (and the polymorphic type inference that makes it practical), unit testing, and compartmentalization.

*Pedagogical structure.* The textbook contains a variety of components in keeping with its pedagogical goals.

- My intention is for the text to be self-contained. Little background is assumed beyond basic programming of the sort learned in a first-semester programming course. Any mathematical ideas that arise in examples or assignments are explained in an appendix.
- Code examples in the text are often developed step-wise, rather than being presented as whole and complete, reflecting how code is typically constructed. Similarly, examples are often revisited as new concepts are introduced that can be used to implement the examples in novel ways.

- The text is tightly connected to a series of pedagogical activities for students. Throughout the text, exercises test understanding of the just presented material; solutions to the exercises, often with extensive further explanations and descriptions of alternatives, are available in an appendix. Supplementary materials tightly connected with the book include labs, problem sets, and a project. Labs, intended to be done individually or synchronously in pairs or groups, provide a series of small and carefully graduated problems that build up practice with the programming concepts introduced in the texts. Lab solutions, again providing alternatives and cross-references to previous and upcoming discussions, are provided. Problem sets provide for more open-ended work on larger-scale but still self-contained problems, and relate to topical issues such as public-key encryption, symbolic math, artificial intelligence search, music composition, and epidemic simulation. The culmination is a project implementing a small run-time-typed subset of OCaml, synthesizing ideas from throughout the book, especially the presentations of formal syntax and semantics.

*Openness.* The text and related materials are intended to be openly available, allowing widespread adoption, including in venues, like MOOCs, where closed materials aren't appropriate.

### *Use of OCaml*

It is typical in courses that introduce multiple programming paradigms to introduce different programming languages geared towards one or another of the paradigms. This language profligacy has the effect of dramatically increasing the amount of language syntax that needs to be introduced and misleadingly implies that the paradigms are coincident with or require different languages. By contrast, I make use of a single well-designed and well-supported language, OCaml, whose relatively simple core allows development and exposition of all of these paradigms and the abstraction mechanisms they rely on. OCaml is introduced and used not for its own sake but as a vehicle for conveying the wide range of programming and computational concepts.

OCaml is an ideal language for pedagogical purposes for the following reasons:

*Simple core.* The language is designed based on a relatively simple core set of orthogonal constructs, which are extended via syntactic sugar. This sparseness means that students can get to the level of implementing an interpreter for a nontrivial subset of the language

by the end of the book.

*Clean semantics.* The language has quite clean semantics, which aids understanding.

*Type discipline.* Programs are strongly statically typed, so that students are confronted from the start with thinking in terms of always and only using values consistently with their types. Experience with reasoning about the types of expressions can inform better programming practice even when programming later in languages with weaker type systems or dynamic typing.

*Multi-paradigm.* Although the core of the language is relatively spare, built on top of the core is syntactic support for multiple paradigms including functional, modular, imperative, lazy, and object-oriented programming.

*Nonproprietary.* The language is supported by an open-source, non-proprietary, cross-platform toolset.

The primary disadvantage of using OCaml is that the language is little known and not widely used in the software industry. It is generally viewed as an “academic language”, of interest to computer scientists rather than mainstream software developers. Nonetheless, the general approach of strongly statically typed languages based on a functional foundation is gaining currency through languages like F#, Reason, Rust, and Elm. More importantly, the goal of the textbook is not to teach a particular language so as to improve employability; rather, it is to teach a range of programming concepts that will be of use whatever language one programs in.

### *Limitations*

The book is intentionally limited in certain ways.

- It does not cover the OCaml language exhaustively, and does not serve as a language reference. This is in keeping with the use of OCaml as a vehicle for presenting concepts. Just enough OCaml is presented to make possible the implementations of the presented concepts. (Cf. Minsky et al.’s *Real World OCaml*.)
- It does not cover formal proofs of correctness (though there is limited and informal discussion of invariants). The importance of correct code is highlighted in a focus on unit testing. (Indeed, a recurring thematic example is the building up of a simple unit testing framework for OCaml.)

- There is no coverage of interactive systems, graphics, or user interface design and implementation. (Cf. Stein's text *Interactive Programming In Java*.)
- No large application examples are given in their entirety. (Cf. the Whitington or Cousineau texts.) However, the problem sets provide opportunity for working with larger-scale examples.

### *Acknowledgements*

The nature of the course – introducing a wide range of programming abstractions and paradigms with an eye toward developing a large design space of programs, using functional programming as its base and OCaml as the delivery vehicle – is shared with similar courses at a number of colleges. The instructors in those courses have for many years informally shared ideas, examples, problems, and notes in an open and free-flowing manner. When I took over the course from Greg Morrisett (now Dean and Vice Provost of Cornell Tech), I became the beneficiary of all of this collaboration, including source materials from these courses – handouts, notes, lecture material, and problem sets – which have been influential in the structure and design of these notes, and portions of which have thereby inevitably become intermixed with my own contributions in a way that would be impossible to disentangle. I owe a debt of gratitude to all of the faculty who have been engaged in this informal sharing, especially,

- Dan Grossman, University of Washington
- Michael Hicks, University of Maryland
- Greg Morrisett, Cornell University
- Benjamin Pierce, University of Pennsylvania
- David Walker, Princeton University
- Stephanie Weirich, University of Pennsylvania
- Steve Zdancewic, University of Pennsylvania

All of these faculty have kindly agreed to allow their contributions to be used here and distributed openly.

In addition, the course and this text have benefited immensely from the large crew of teaching staff of CS51 throughout the years. These include the head teaching fellows(list goes here tbd) as well as Sam Green and Serina Hu for help developing the `caml-tex` system that allows running the code examples as part of the typesetting process.



# 1

## *Introduction*

We forget how incredible the computer is. The modern computer executes billions of operations *each second*, every one of which must work perfectly – accurately performing the right operation at each and every cycle. How is this even possible? How can we, mere humans with our cognitive limitations, manage to build devices that work at this pace with this level of fidelity? Each of the billions of instructions executed per second on a modern computer is another detail to be managed. How can we gain control over this mass of detail?

In Jorge Luis Borges's 1944 short story *Funes the Memorious*, the protagonist, Ireneo Funes, experiences what it is like to perceive the world at this level of streaming detail. After being thrown from a wild horse and severely crippled, he develops a prodigious memory. He recalls, perfectly and instantaneously, every moment of his life.

He knew by heart the forms of the southern clouds at dawn on the 30th of April, 1882, and could compare them in his memory with the mottled streaks on a book in Spanish binding he had only seen once.... Two or three times he had reconstructed a whole day; he never hesitated, but each reconstruction had required a whole day. (Borges, 1962)

Yet, each of his memories was individual, disconnected, divorced of any higher structural patterns. Borges relates,

With no effort, he had learned English, French, Portuguese and Latin. I suspect, however, that he was not very capable of thought. To think is to forget differences, generalize, make abstractions. In the teeming world of Funes, there were only details, almost immediate in their presence.

Without abstraction, there are only details. And it is through abstraction – forgetting differences, generalizing – that we can get control of the sheer daunting complexity of controlling a computer.

What is abstraction? ABSTRACTION is the process of *viewing a set of apparently dissimilar things as instantiating an underlying identity*. Funes sees a field of flowers, hundreds of blooms. To him, they are each

individuals, but to the botanist, these apparently dissimilar individuals are all instances of a type, the genus *Tulipa*, the tulips. By capturing innumerable individual plants into a hierarchy of abstract families, genera, and species, the bewildering complexity of plant life on the planet becomes more manageable.

Programming computers is a battle against the sheer daunting complexity of the task. The chief weapon in the battle is abstraction. The first objective of this book is to introduce you to a broad variety of abstraction mechanisms and their uses, providing you with an appropriate armamentarium. The second objective is to open your eyes to the beauty that computer programming can manifest when those tools are elegantly applied.

You are already familiar with some of the primary abstraction mechanisms used in programming computers. (I assume throughout this book that you've had some experience programming computers using an imperative programming language, of the sort, for instance, acquired in Harvard's CS50 or CS50x course.)

Let's take as an example the problem of generating a table of logarithms. The choice is not random. The building of tables of mathematical functions like the logarithm was the motivating task for the earliest computer designs, those of Charles Babbage in the 1820s and 1830s (Figure 1.1). In the margin (Figure 1.2) is the beginning of such a table.

A program to print out this kind of table might look like this:

```
printf "1 0.0000\n";
printf "2 1.0000\n";
printf "3 1.5850\n";
printf "4 2.0000\n" ;;
```

and when the program is executed, it prints the table:

```
# printf "1 0.0000\n";
# printf "2 1.0000\n";
# printf "3 1.5850\n";
# printf "4 2.0000\n" ;;
1 0.0000
2 1.0000
3 1.5850
4 2.0000
- : unit = ()
```

(For the moment, the details of the language in which this computation is written are immaterial. We'll get to all that in a bit. The idea is just to get the gist of the argument. In the meantime, you can just let the code waft over you like a warm summer breeze.)

Now of course this code is hopelessly written. Why? Because it treats each line of the table as an individual specimen, missing the abstract view. The first step in viewing the lines abstractly is to note

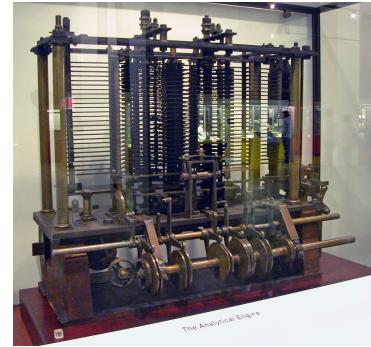


Figure 1.1: A model of a part of Charles Babbage's analytical engine, intended for the calculation of tables of mathematical functions such as the trigonometric functions like sine and cosine, the Bernoulli numbers, or logarithms, as in Figure 1.2 below.

$x$	$\log_2 x$
1	0.0000
2	1.0000
3	1.5850
4	2.0000
	...

Figure 1.2: A small table of logarithms

that they are actually instances of an underlying uniformity: Each string is of the form of an integer (call it  $x$ ) and the log (with base 2) of  $x$ . They are instances of the underlying pattern

```
printf "%2d %2.4f\n" x (log2 x);
```

for each of several values of the variable  $x$ . (Again, the details of the language being used are postponed, but you hopefully get the idea.) This mechanism, the STATE VARIABLE, is thus a mechanism for abstraction – for making apparently dissimilar computations manifest an underlying identity. To take full advantage of this type of variable, we'll need to specify the sequential values, 1 through 4 say, that the variable takes on, using a LOOP.

```
for x = 1 to 4 do
    printf "%2d %2.4f\n" x (log2 x)
done
```

Like [Monsieur Jourdain](#), who discovered he'd been speaking prose his whole life, you've been using abstraction mechanisms without realizing it. Without them, programming is impossible.

This particular style of programming, imperative programming, is undoubtedly most familiar to you. Its most basic abstraction mechanisms are the state variable and the loop. It is the style seen in some of the earliest, most influential programming languages, from FORTRAN to the ALGOL family of languages, to C, to Python, and beyond. And it is the style of programming captured by the first universal model of computation, the TURING MACHINE of Alan Turing (Figure 1.3).

But there are many other abstraction mechanisms than state variables and loops, underpinning many other programming paradigms than imperative programming, and allowing many other ways of designing computations. It is the goal of this book to introduce several such abstraction mechanisms, provide practice in their use and application, and thereby open up a broad range of programming possibilities not otherwise available.

An especially important abstraction mechanism is the FUNCTION, a mapping from inputs to outputs. The idea of the function gives its name to the paradigm of *functional programming*, and we will begin with functions and functional programming ideas. But functional programming is only one of several paradigms that we will discuss.

## 1.1 An extended example: greatest common divisor

By way of example of the distinction between imperative and functional programming, consider the very practical question of tiling a bathroom floor of size 28 by 20 units. We can tile such a floor with tiles

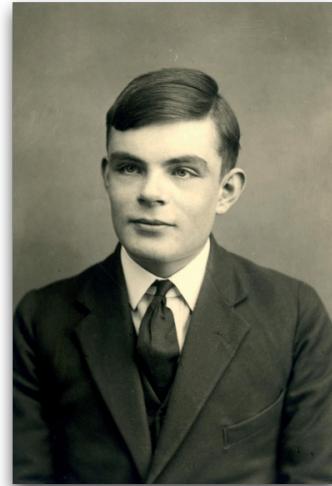


Figure 1.3: Alan Turing (1912–1954), whose Turing machine provided the first universal model of computation, based on imperative programming notions of state and state change. Turing is rightfully credited with fundamental contributions to essentially all areas of computer science: the theory of computing, hardware, software, artificial intelligence, computational biology, and much more. His premature death by suicide at 41 after undergoing “therapy” at the hands of the British government following his conviction for the “crime” of homosexuality is certainly one of the great intellectual tragedies of the twentieth century. (The British government got around to apologizing for his treatment some 50 years later.) The highest award in computing, the Turing Award, is appropriately named after him.

that are 2 by 2, since both 28 and 20 are evenly divisible by 2, but 3 by 3 tiles don't work, since neither 28 nor 20 are divisible by 3. If we want to use the fewest tiles, it would be useful to know the largest number that divides both dimensions evenly, their GREATEST COMMON DIVISOR (GCD).

Here is how we might program a calculation of GCD in an imperative style:

```
let gcd_down a b =
  let guess = ref (min a b) in
  while (a mod !guess > 0) || (b mod !guess > 0) do
    guess := !guess - 1
  done;
  !guess ;;
```

This procedure works by counting down from the smaller of the two numbers, one by one, until a common divisor is found. Since the search for the common divisor is from the largest to the smallest possibility, the greatest common divisor is found.

In the functional style, this same “countdown” algorithm might be coded like this:

```
let gcd_func a b =
  let rec downfrom guess =
    if (a mod guess > 0) || (b mod guess > 0) then
      downfrom (guess - 1)
    else guess
  in
  downfrom (min a b) ;;
```

Here, in the context of calculating the GCD of *a* and *b*, a new function *downfrom* is introduced to check a particular guess of the GCD of the two numbers. The *downfrom* function takes an input *guess* and checks whether it is the GCD of *a* and *b*. If so, the output value of the function is the *guess* *guess* itself, but if not, a one-smaller *guess* is tried. Having defined this counting-down function, the calculation of the GCD itself proceeds just by guessing the minimum of the two numbers.

You may find unusual some of the properties of this latter implementation of what is essentially the identical algorithm – counting down one by one from the minimum of the two numbers until a common divisor is found. First, there are no overt loops, and no assignments to variables that change the state of the computation by changing the value of a variable. It's just functions and their application. Second, the function *downfrom* defined in the code appeals to *downfrom* itself as part of the calculation of its output. It is defined by RECURSION, that is, in terms of itself. Such functions are *recursive*, and when they invoke themselves for a computation are said to *recur*.<sup>1</sup> You may wonder whether this is quite kosher. Isn't defining something in terms of itself a bad idea? But in this case at least, the definition works

<sup>1</sup> Not *recurve* please. To *recurve* is to curse again, not the kind of thing a program – or a person – should be doing.

fine, because the value of `downfrom guess` depends not on the value of `downfrom guess` itself but of `downfrom (guess - 1)`, a different value. This may itself depend on `downfrom (guess - 2)`, and so on, but eventually one of the inputs to `downfrom` will be a common divisor, and in that case, the output value of `downfrom` does not depend on `downfrom` itself. The recursion “bottoms out” and the GCD is returned.

This style of programming – by defining and applying functions – has a certain elegance, which can be seen already in the distinction between the two versions of the GCD computation already provided. But as it turns out, the algorithm underlying both of these implementations is a truly bad one. Counting down is just not the right way to calculate the GCD of two numbers. As far back as 300 BCE, Euclid of Alexandria provided a far better algorithm in Proposition 1 of Book 7 (Figure 1.4) of his treatise on mathematics, *Elements*. Euclid’s algorithm for GCD is based on the following insight: Any square tiling of a 20 by 28 area will tile both a 20 by 20 square and the 8 by 20 remainder. More generally, any square tiling of an  $a$  by  $b$  area (where  $a$  is greater than  $b$ ) will tile both a  $b$  by  $b$  square and the  $b$  by  $a - b$  remainder. Thus, to calculate the GCD of  $a$  and  $b$ , it suffices to calculate the GCD of  $b$  and  $a - b$ . Eventually, we’ll be looking for the GCD of two instances of the same number (that is,  $a$  and  $b$  will be the same; we’ll be looking to tile a square area) in which case we know the GCD; it is  $a$  (or  $b$ ) itself. Figure 1.5 shows the succession of smaller and smaller rectangles explored by Euclid’s algorithm for the 20 by 28 case.

An initial presentation of Euclid’s algorithm is this:

```
let rec gcd_euclid a b =
  if a < b then gcd_euclid b a
  else if a = b then a
  else gcd_euclid b (a - b) ;;
```

Now, in the case that  $a = b$ , were we to continue on one more round of checking the GCD of  $b$  and  $a - b$ , the difference  $a - b$  would simply be 0. Thus, we can check for this condition instead.

```
let rec gcd_euclid a b =
  if a < b then gcd_euclid b a
  else if b = 0 then a
  else gcd_euclid b (a - b) ;;
```

We can simplify further. When subtracting off  $b$  from  $a$ , the remainder may still be greater than  $b$ , in which case, we’ll want to subtract  $b$  again, continuing to subtract  $b$  until, eventually, the remainder is less than  $b$ . Thus, instead of using the difference  $a - b$  as the new second argument of the recursive call, we can use the remainder  $a$  modulo  $b$ .

```
let rec gcd_euclid a b =
  if a < b then gcd_euclid b a
```

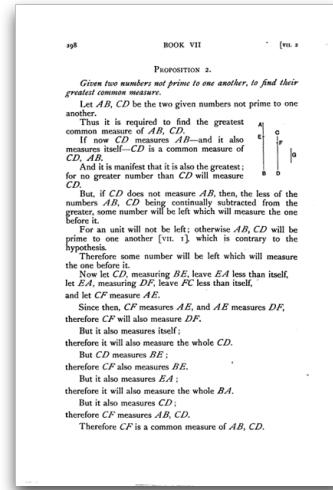
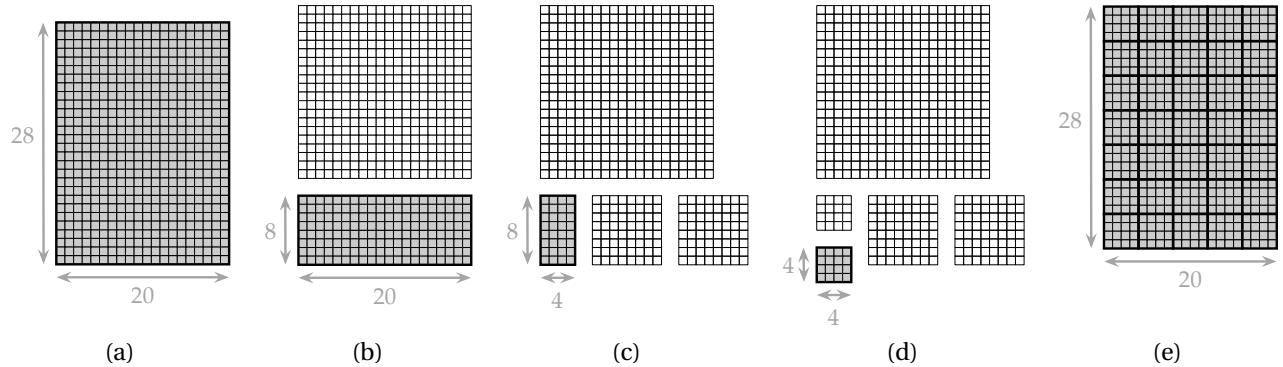


Figure 1.4: Proposition 1 of Book 7 of Euclid’s *Elements*, providing his algorithm for calculating the greatest common divisor of two numbers.



```
else if b = 0 then a
else gcd_euclid b (a mod b) ;;
```

Finally, notice that if  $a < b$ , then the values  $b$  and  $a \text{ mod } b$  are just  $b$  and  $a$ , respectively – exactly the values we want to use for the recursive call in that case. We can therefore drop the test for  $a < b$  entirely.

```
let rec gcd_euclid a b =
  if b = 0 then a
  else gcd_euclid b (a mod b) ;;
```

This is EUCLID'S ALGORITHM. Compare it to the countdown algorithm above. The difference is stark. Euclid's method is beautiful in its simplicity.

It is also, as it turns out, much more efficient. This can be determined analytically or experienced empirically.

## 1.2 Programming as design

Euclid's algorithm for GCD shows us that there is more than one way to solve a problem, and *some ways are better than others*. The dimensions along which programmed solutions can be better or worse are manifold. They include

- succinctness,
- efficiency,
- readability,
- maintainability,
- provability,
- testability,

Figure 1.5: Euclid's algorithm for GCD starting (a) with a  $20 \times 28$  rectangle to be tiled. Removing the  $20 \times 20$  square (b) leaves a  $20 \times 8$  remainder to be tiled. From that rectangle, we remove, successively, two  $8 \times 8$  squares (c), leaving a  $4 \times 8$  remainder. Finally, removing a  $4 \times 4$  square (d) leaves a  $4 \times 4$  square, the largest square that can tile the whole (e).

and, most importantly but ineffably,

- beauty.

Computer programming is not the only practice where practitioners may generate multiple ways of satisfying a goal, which can be evaluated along multiple independent and perhaps conflicting metrics. Architects, engineers, illustrators, industrial designers may generate wildly different plans in response to a client's constraints and desires. All live in a space of possibilities from which they choose solutions that vary along multiple, often competing, criteria.

What all of these practices have in common is DESIGN – the navigation of a space of options, generated by applicable tools, in search of the good, as measured along multiple dimensions. In the case of computer programming, the tools are exactly the abstraction mechanisms provided by a programming language.

A crucial consideration in teaching programming from this perspective is what abstraction mechanisms to concentrate on, as these define the space of options within which we can navigate. As discussed above, the most important of these abstraction mechanisms is the function. In addition to being a fantastic method for abstracting computation (which will become clear some time around Chapter 8), functions also serve as a platform upon which many other abstraction mechanisms can be deployed and combined. It may be difficult at first to see the incredible utility of the function as a unifying abstraction mechanism, but hopefully, as you see more and more examples of their use in combination with other techniques, you will come to appreciate the function's centrality in the design of programs.

Indeed, functions and their application are such a powerful computational tool that they constitute, by themselves, a complete universal computational mechanism. The Princeton mathematician and logician Alonzo Church (1936) developed a “calculus” of functions alone, the so-called LAMBDA CALCULUS (see Section B.1.4), a logical system that includes functions and their application and *literally nothing else* – no data objects or data structures of any kind, neither atomic (like integers) nor composite (like lists); no mutable state (like variable assignment); no control structures (like conditionals or loops). Astoundingly, Turing (1937) was then able to show that anything that can be computed by his universal model of computation, the Turing machine, can also be computed in Church's lambda calculus. Thus, the lambda calculus – comprised only of functions and their applications remember – is itself a universal model of computation. This argument for the universality of Turing's and Church's computation models is now known as the CHURCH-TURING THESIS. (The close connection



Figure 1.6: Princeton professor Alonzo Church (1903–1995), inventor of the lambda calculus, the foundation of all functional programming languages; PhD adviser of Alan Turing.

between the lambda calculus and the Turing machine mirrors the close relationship between Church and Turing; Church was Turing's PhD adviser at Princeton.)

In this book, we concentrate on the following abstraction mechanisms, listed with the style of programming they are associated with:

<i>Abstraction</i>	<i>Programming paradigm</i>
functions	functional programming
algebraic data types	structure-driven programming
polymorphism	generic programming
abstract data types	modular programming
mutable state	imperative programming
loops	procedural programming
lazy evaluation	programming with infinite data structures
object dispatch	object-oriented programming
concurrency	concurrent programming

Of course, there are many other abstraction mechanisms and programming paradigms, but these should both give you a good sense of the importance of a variety of abstractions and provide an excellent base on which to build.

As with any design practice, computer programming is best learned by seeing a range of examples of the space of options – examples that are better or worse along one dimension or another – with attention paid to the process of developing, modifying, and improving such solutions. For that reason, we will often show computer programs being built up in stages and being modified to demonstrate alternative designs (as we did with the GCD example above), and programming problems will be revisited as new abstraction mechanisms open further parts of the design space. You may find the multiple variations on a theme redundant – as indeed they are – but we know of no better way to get across the idea of programming as a design practice than the careful development and exploration of a significant program design space.

### 1.3 *The OCaml programming language*

In order that we can introduce multiple abstraction mechanisms and programming paradigms with a minimum of programming language detail, we use a multi-paradigm programming language called OCAML. (The examples above were written in OCaml.) OCaml is a member of the ML family of programming languages first developed

Table 1.1: Some abstraction mechanisms and the programming paradigms they allow.



Figure 1.7: Robin Milner (1934–2010), developer of the ML programming language, the first functional language with type inference, and the programming language from which OCaml derives. He received the Turing Award in 1991 for his work on ML and other innovations.

at University of Edinburgh by Robin Milner (Figure 1.7) in the 1970's. The OCaml dialect of ML itself was developed at the French national research lab Institut National de Recherche en Informatique et en Automatique (INRIA), where it continues to be developed and maintained. OCaml is a multi-paradigm programming language in that it provides support not only for functional programming, but also imperative programming, object-oriented programming, and all the other mechanisms and paradigms listed in Table 1.1.

OCaml is especially attractive from a pedagogical standpoint because it provides these capabilities on the basis of a relatively small foundation of well-designed orthogonal primitive language constructs, so that programming concepts can be introduced and experimented with, without the need for learning a huge set of syntactic idiosyncrasies. Nonetheless, as with learning any new programming language, it will take a bit of getting used to the ideas and notations of OCaml, and in fact getting practice with learning new notations is a useful skill in its own right.

I emphasize that this book is not a book about OCaml programming. (For instance, this book doesn't pretend to present the language comprehensively, instead covering only those parts of the language needed to present the principles being taught. For that reason, you will want to get at least a bit familiar with [the reference documentation of the language](#).) Rather, it is a book about the role of abstraction in the design of software, which uses the OCaml language as the medium in which to express these ideas. But in order to get these ideas across, we need *some* language, and it turns out that OCaml is an ideal language for this pedagogical purpose. Of course, we'll have to spend some time going over the particularities of the OCaml language, which may seem odd mostly because of their unfamiliarity. The text may have a bit of a disjointed quality to it, bouncing back and forth between details of OCaml and higher-level concepts. But the time spent learning the details of the language isn't time wasted. It pays off in lessons that generalize to any programming you will do in the future. You may discover, like many do, that once you've gained some proficiency with OCaml, you find its charms irresistible, and continue to use it (or its close derivatives like Microsoft's F#, Facebook's Reason, Apple's Swift, or Mozilla's Rust) when appropriate – as many companies including those mentioned do. But whether you continue to program in OCaml or not, the patterns of thinking and the sophistication of your understanding will be the payoff of this process, translatable to any programming you'll do in the future. In fact, the market for software developers reflects this payoff as well. As shown in Figure 1.8, the market rewards software developers fluent in the kinds of technologies and ideas fea-

tured in this book; their salaries are substantially higher on average. Although such pecuniary benefits aren't the point of this book, they certainly don't hurt.

### 1.4 Tools and skills for design

The space of design options available to you is enabled by the palette of abstraction mechanisms that you can fluently deploy. Navigating the design space to find the best solutions is facilitated by a set of skills and analytic tools, which we will also introduce throughout the following chapters as they become pertinent. These include more precise notions of the syntax and semantics of programming languages, facility with notations, sensitivity to programming style (see especially Appendix C), programming interface design, unit testing, tools (big-O notation, recurrence equations) for analyzing efficiency of code. Having these tools and skills at your disposal will add to your computational tool-box and stretch your thinking about what it means to write good code. I expect, based on my own experiences, that learning to develop, analyze, and express your software ideas with precision will also benefit your abilities to develop, analyze, and express ideas more generally.

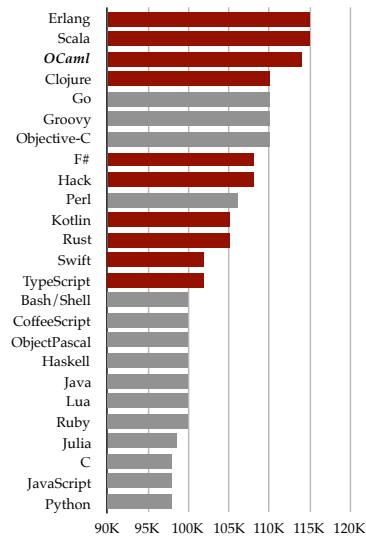


Figure 1.8: United States average salary by technology, from [StackOverflow Developer Survey 2018](#). Highlighted bars correspond to technologies in the typed functional family.

## 2

# *A Cook's tour of OCaml*

To give a flavor of working with the OCaml programming language, we introduce OCaml through an INTERPRETER of the language, called `ocaml`, which is invoked from the command line thus:<sup>1</sup>

```
% ocaml
```

Upon running `ocaml`, you will see a PROMPT ("#") allowing you to type an OCaml expression.

```
% ocaml
          OCaml version 4.14.2
#
```

### **Exercise 1**

The startup of the `ocaml` interpreter indicates that this is version 4.14.2 of the software. What version of `ocaml` are *you* running?

Once the OCaml prompt is available, you can enter a series of OCaml expressions to calculate the values that they specify. Numeric (integer) expressions are a particularly simple case, so we'll start with those. The integer LITERALS – like 3 or 42 or -100 – specify integer values directly, but more complex expressions built by applying arithmetic functions to other values do as well. Consequently, the OCaml interpreter can be used as a kind of calculator.

```
# 42 ;;
- : int = 42
# 3 + 4 * 5 ;;
- : int = 23
# (3 + 4) * 5 ;;
- : int = 35
```

Since this is the first example we've seen of interaction with the OCaml interpreter, some glossing may be useful. The OCaml interactive prompt, '#', indicates that the user can enter an OCaml expression, such as '3 + 4 \* 5'. A double semicolon ';' demarcates the end of the expression. The system *reads* the expression, *evaluates* it (that

<sup>1</sup> We assume that you've already installed the OCaml tools, as described at the [ocaml.org](http://ocaml.org) web site.

is, calculates its value), and *prints* an indication of the result, then *loops* back to provide another prompt for the next expression. For this reason, the OCaml interactive system is referred to as the “READ-EVAL-PRINT LOOP” or REPL.<sup>2</sup> Whenever we show the results of an interaction with the REPL, the interpreter’s output will be shown in a *slanted font* to distinguish it from the input.

You’ll notice that the REPL obeys the standard order of operations, with multiplication before addition for instance. This precedence can be overwritten in the normal manner using parentheses.

#### Exercise 2

Try entering some integer expressions into the OCaml interpreter and verify that appropriate values are returned.

Although we’ll introduce the aspects of the OCaml language incrementally over the next few chapters, to get a general idea of using the language, we demonstrate its use with the GCD algorithm from Chapter 1. We type the definition of the `gcd_euclid` function into the REPL:

```
# let rec gcd_euclid a b =
#   if b = 0 then a
#   else gcd_euclid b (a mod b) ;;
val gcd_euclid : int -> int -> int = <fun>
```

Now we can make use of that definition to calculate the greatest common divisor of 20 and 28

```
# gcd_euclid 20 28 ;;
- : int = 4
```

But we’re getting ahead of ourselves.

<sup>2</sup> To exit the REPL, just enter the end-of-file character, ^d, typed by holding down the control key while pressing the d key.

# 3

## *Expressions and the linguistics of programming languages*

Programming is an expressive activity: We express our intentions to a computer using a language – a programming language – that is in some ways similar to the natural languages that we use to communicate with each other.

One of the deep truths of linguistics, known since the time of the great Sanskrit grammarian Pāṇini in the fourth century BCE, is that the expressive units of natural languages, or EXPRESSIONS as we will call them, have hierarchical structure. (The recovery of that structure used to be a typical subject matter taught to students in “grammar school” through the exercise of sentence diagramming.) Characterizing what are the well-formed and -structured phrases of a language constitutes the realm of SYNTAX.

### *3.1 Specifying syntactic structure with rules*

The expressions of English (and other natural languages) are formed as sequences of words to form expressions of various types. By way of example, noun phrases can be formed in various ways: as a single noun (*party* or *drinker* or *tea*), or by putting together (in sequential order) a noun phrase and a noun (as in *tea party*), or by putting together (again in order) an adjective (*iced* or *mad*) and another noun phrase as in (*iced tea*). We can codify these rules by defining classes of expressions like  $\langle \text{noun} \rangle$  or  $\langle \text{nounphrase} \rangle$  or  $\langle \text{adjective} \rangle$ . We’ll write the rule that allows forming a noun phrase from a single noun as

$\langle \text{nounphrase} \rangle ::= \langle \text{noun} \rangle$

The rules that form a noun phrase from an adjective and a noun phrase or from a noun phrase and a noun are, respectively,

$$\begin{aligned}\langle \text{nounphrase} \rangle &::= \langle \text{adjective} \rangle \langle \text{nounphrase} \rangle \\ \langle \text{nounphrase} \rangle &::= \langle \text{nounphrase} \rangle \langle \text{noun} \rangle\end{aligned}$$

In these rules, we write  $\langle \text{noun} \rangle$  to indicate the class of noun expressions,  $\langle \text{nounphrase} \rangle$  to indicate the class of noun phrases, and in general, put the names of classes of expressions in angle brackets to represent elements of that class. The notation  $::=$  should be read as “can be composed from”, so that expressions of the class on the left of the  $::=$  can be composed by putting together expressions of the classes listed on the right of the  $::=$ , in the order indicated.

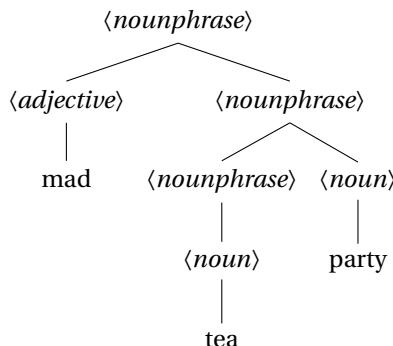
This rule notation for presenting the syntax of languages is called BACKUS-NAUR FORM (BNF), named after John Backus and Peter Naur, who proposed it for specifying the syntax of the ALGOL family of programming languages. But as noted above, the idea goes back much further, at least to Pāṇini.

Putting these rules together, the BNF specification for noun phrases is

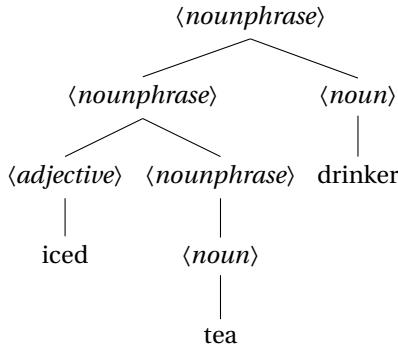
$$\begin{aligned}\langle \text{nounphrase} \rangle &::= \langle \text{noun} \rangle \\ &\quad | \quad \langle \text{adjective} \rangle \langle \text{nounphrase} \rangle \\ &\quad | \quad \langle \text{nounphrase} \rangle \langle \text{noun} \rangle\end{aligned}$$

Here, we've rephrased the three rules as a single rule with three alternative right-hand sides. The BNF notation allows separating alternative right-hand sides with the vertical bar ( $|$ ) as we have done here.

A specification of a language using rules of this sort is referred to as a GRAMMAR. According to this grammar, we can build noun phrases like *mad tea party*

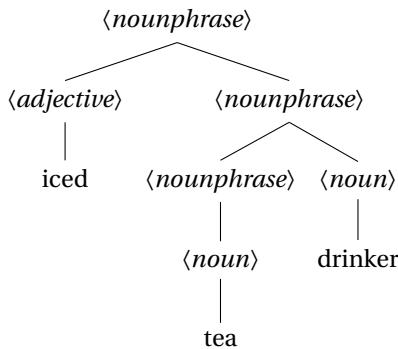


or *iced tea drinker*



Notice the difference in structure. In *mad tea party*, the adjective *mad* is combined with the phrase *tea party*, but in *iced tea drinker*, the adjective *iced* does not combine with *tea drinker*. The drinker isn't iced; the tea is!

But these same rules *can* also be used to build an alternative tree for “iced tea drinker”:



The expression *iced tea drinker* is AMBIGUOUS (as is *mad tea party*); the trees make clear the two syntactic analyses.

Importantly, as shown by these examples, it is the syntactic tree structures that dictate what the expression means. The first tree seems to describe a drinker of cold beverages, the second a cold drinker of beverages. The syntactic structure of an utterance thus plays a crucial role in its meaning. The characterization of the meanings of expressions on the basis of their structure is the realm of SEMANTICS, pertinent to both natural and programming languages. We'll come back to the issue of semantics in detail in Chapters 13 and 19.

#### Exercise 3

Draw a second tree structure for the phrase *mad tea party*, thereby demonstrating that it is also ambiguous.

#### Exercise 4

How many trees can you draw for the noun phrase *flying purple people eater*? Keep in mind that *flying* and *purple* are adjectives and *people* and *eater* are nouns.

The English language, and all natural languages, are ambiguous that way. Fortunately, context, intonation, and other clues disambiguate

these ambiguous constructions so that we are mostly unaware of the ambiguities.<sup>1</sup> In the case of the mad tea party, we understand the phrase as having the syntactic structure as displayed above (as opposed to the one referred to in Exercise 3).

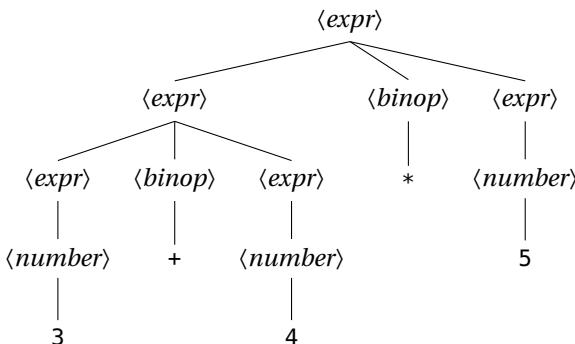
### 3.2 Disambiguating ambiguous expressions

Programming language expressions, like the utterances of natural language, have syntactic structure as well. Without some care, programming languages might be ambiguous too. Consider the following BNF rules for simple arithmetic expressions built out of numbers and BINARY OPERATORS (operators, like +, -, \*, and /, that take two arguments).<sup>2</sup>

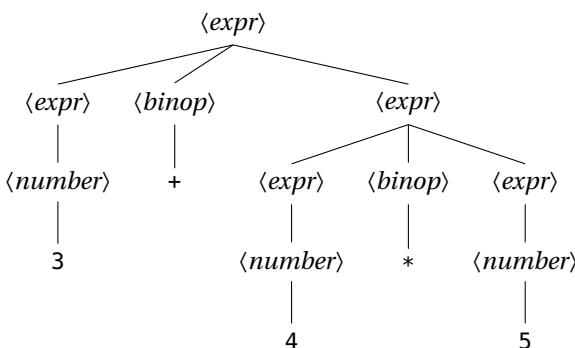
```

⟨expr⟩ ::= ⟨exprleft⟩ ⟨binop⟩ ⟨exprright⟩
         | ⟨number⟩
⟨binop⟩ ::= + | - | * | /
⟨number⟩ ::= 0 | 1 | 2 | 3 | ...
  
```

Using these rules, we can build two trees for the expression  $3 + 4 * 5$ :



or



<sup>1</sup> The rare exceptions where ambiguities are brought to our attention account for the humor (of a sort) found in syntactically ambiguous sentences, as in [the old joke](#) that begins “I shot an elephant in my pajamas.”

<sup>2</sup> In defining expression classes using this notation, we use subscripts to differentiate among different occurrences of the same expression class, such as the two <expr> instances <expr<sub>left</sub>> and <expr<sub>right</sub>> in the first BNF rule.

But in the case of programming languages, we don't have the luxury of access to intonation or shared context to disambiguate expressions.

Instead, we rely on other tools – *conventions* and *annotations*.

In the way of conventions, we rely on a conventional ORDER OF OPERATIONS that dictates which operations we tend to do “first”, that is, lower in the tree. We refer to this kind of priority of operators as their PRECEDENCE, with higher precedence operators appearing lower in the tree than lower precedence operators. By convention, we take the additive operators (+ and -) to have lower precedence than the multiplicative operators (\*, /). Thus, the expression  $3 + 4 * 5$  has the structure shown in the second tree, not the one shown in the first. For that reason, it expresses the value 23 and not 35.

Precedence is not sufficient to disambiguate, for instance, expressions with two binary operators of the same precedence. Precedence alone doesn't disambiguate the structure of  $5 - 4 - 1$ : Is it  $(5 - 4) - 1$ , that is, 0, or  $5 - (4 - 1)$ , that is, 2. Here, we rely on the ASSOCIATIVITY of an operator. We say that subtraction, by convention, is LEFT ASSOCIATIVE, so that the operations are applied starting with the left one. The grouping is  $(5 - 4) - 1$ . Other operators, such as OCaml's exponentiation operator \*\* are RIGHT ASSOCIATIVE, so that  $2. ** 2. ** 3.$  is disambiguated as  $2. ** (2. ** 3.)$ . Its value is 256., not 64..<sup>3</sup>

Associativity and precedence conventions go a long way in picking out the abstract structure of concrete expressions. But what if we want to override those conventions? What if, say, we want to express the left-branching tree for  $3 + 4 * 5$ ? We can use annotations, as indeed, we already have, to enforce a particular structure. This is the role of PARENTHESES, to override conventional rules for disambiguating expressions. In the case at hand, we write  $(3 + 4) * 5$  to obtain the left-branching tree.

#### Exercise 5

What is the structure of the following OCaml expressions? Draw the corresponding tree so that it reflects the actual precedences and associativities of OCaml. Then, try typing the expressions into the REPL to verify that they are interpreted according to the structure you drew.

1.  $10 / 5 / 2$
2.  $5. +. 4. ** 3. /. 2.$
3.  $(5. +. 4.) ** (3. /. 2.)$
4.  $1 - 2 - 3 - 4$

You may have been taught this kind of rule under the mnemonic PEMDAS. But the ideas of precedence, associativity, and annotation are quite a bit broader than the particulars of the PEMDAS convention. They are useful in thinking more generally about the relationship between what we will call concrete syntax and abstract syntax.

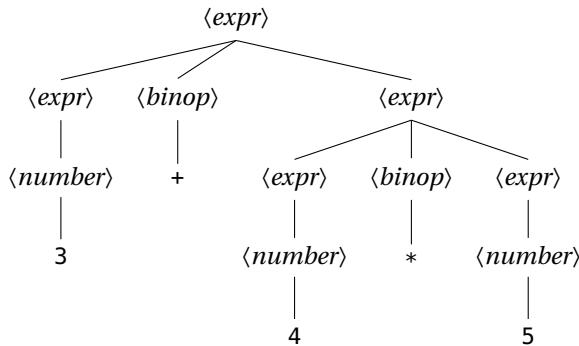
<sup>3</sup>The \*\* operator applies to and returns floating point values, hence the decimal point dots in the arguments and return values.

For a more complete presentation of the precedences and associativities of all of the built-in operators of OCaml, see the [documentation on OCaml's operators](#).

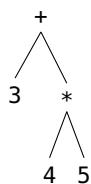
### 3.3 Abstract and concrete syntax

The right way to think of expressions, then, is as hierarchically structured objects, which we have been depicting with trees as specified by BNF grammar rules. From a practical perspective, however, when programming, we are forced to notate these expressions in an unstructured linear form as a sequence of characters, in order to enter them into a computer. We use the term ABSTRACT SYNTAX for expressions viewed as structured objects, and CONCRETE SYNTAX for expressions viewed as unstructured linear text.

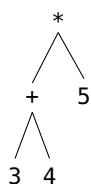
In order to more directly present the abstract syntax that corresponds to a concrete expression, we draw trees as above that depict the structure.<sup>4</sup> So, for instance, the concrete syntax expression  $3 + 4 * 5$  corresponds to the ABSTRACT SYNTAX TREE



We might abbreviate the tree structure to highlight the important aspects by eliding the expression classes as



Then the alternative abstract syntax tree



would correspond to the concrete syntax  $(3 + 4) * 5$ . Parentheses as used for grouping are therefore notions of concrete syntax, not abstract syntax. Similarly, conventions of precedence and associativity have to do with the interpretation of concrete syntax, as opposed to abstract syntax.

<sup>4</sup> The trees shown in Section 3.1, and those shown below, provide more detail than necessary for capturing the structure of the concrete expressions. For that reason, they are, strictly speaking, more like PARSE TREES, rather than abstract syntax trees. (The abbreviated versions introduced below get more to the point of true abstract syntax trees.) But since these parse trees capture structure that the concrete linear forms do not, they will serve our purposes, and we will continue to use these trees to represent abstract syntactic structure and BNF notation to define them. A good course in programming languages will more precisely distinguish parse trees that structure the concrete syntax from abstract syntax trees.

In fact, there are multiple concrete syntax expressions for this abstract syntax, such as  $(3 + 4) * 5$ ,  $((3 + 4) * 5)$ ,  $(3 + ((4))) * 5$ . But certain expressions that may seem related do not have this same abstract syntax:  $5 * (3 + 4)$  or  $((4 + 3) * 5)$  or  $(3 + 4 + 0) * 5$ . Although these expressions specify the same value, they do so in syntactically distinct ways. The fact that multiplication and addition are commutative, or that 0 is an additive identity – these are *semantic* properties, not syntactic.

#### Exercise 6

Draw the (abbreviated) abstract syntax tree for each of the following concrete syntax expressions. Assume the further BNF rule

$\langle \text{expr} \rangle ::= \langle \text{unop} \rangle \langle \text{expr} \rangle$

for unary operators like  $\sim$ , the unary negation operator.

1.  $(\sim 4) + 6$
2.  $\sim (4 + 6)$
3.  $20 / \sim 4 + 6$
4.  $5 * (3 + 4)$
5.  $((4 + 3) * 5)$
6.  $(3 + 4 + 0) * 5$

#### Exercise 7

What concrete syntax corresponds to the following abstract syntax trees? Show as many as you'd like.

1. 
2. 
3. 

### 3.4 Expressing your intentions

It is through the expressions of a programming language – structured as abstract syntax and notated through concrete syntax – that programmers express their intentions to a computer. The computer interprets the expressions in order to carry out those intentions.

Programming is an expressive activity with multiple audiences. Of course, the computer is one audience; a program allows for programmers to express their computational intentions to the computer. But there are human audiences as well. Programs can be used to communicate to other people – those who might be interested in an algorithm for its own sake, or those who are tasked with testing, deploying, or maintaining the programs. One of these latter programmers might even be the future self of the author of the original code. Weeks or even days after writing some code, you might well have already forgotten why you wrote the code a certain way. The following fundamental principle thus follows:

*Edict of intention:  
Make your intentions clear.*

Programmers make mistakes. If their intentions are well expressed, other programmers reviewing the code can notice that those intentions are inconsistent with the code. Even the computer interpreting the program can itself take appropriate action, notifying the programmer with a useful error or warning before the code is executed and the unintended behavior can manifest itself.

Over the next chapters, we'll see many ways that the edict of intention is applied. But one of the most fundamental is through documentation of code.

### 3.4.1 Commenting

One of the most valuable aspects of the concrete syntax of any programming language is the facility to provide elements in a concrete program that have *no correspondence whatsoever* in the abstract syntax, and therefore no effect on the computation expressed by the program. The audience for such COMMENTS is the population of human readers of the program. Comments serve the crucial expressive purpose of documenting the intended workings of a program for those human readers.

In OCaml, comments are marked by surrounding them with special delimiters: `(* ⟨ ⟩ *)`.<sup>5</sup> The primary purpose of comments is satisfying the edict of intention. Comments should therefore describe the *why* rather than the *how* of a program. Section C.2 presents some useful stylistic considerations in providing comments for documenting programs.

There are other aspects of concrete syntax that can be freely deployed because they have no affect on the computation that a program carries out. These too can be judiciously deployed to help express your

<sup>5</sup> We use the symbol `⟨ ⟩` here and throughout the later chapters as a convenient notation to indicate unspecified text of some sort, a textual anonymous variable of a sort. Here, it stands in for the text that forms the comment. In other contexts it stands in for the arguments of an operator, constructor, or subexpression, for instance, in `⟨⟩ + ⟨⟩` or `⟨⟩ list` or `let ⟨⟩ in ⟨⟩`.

intentions. For instance, the particular spacing used in laying out the elements of a program doesn't affect the computation that the program expresses. Spaces, newlines, and indentations can therefore be used to make your intentions clearer to a reader of the code, by laying out the code in a way that emphasizes its structure or internal patterns. Similarly, the choice of variable names is completely up to the programmer. Variables can be consistently renamed without affecting the computation. Programmers can take advantage of this fact by choosing names that make clear their intended use.



Having clarified these aspects of the syntactic structure of programming languages (and OCaml in particular) – distinguishing concrete and abstract syntax; presenting precedence, associativity, and parenthesization for disambiguation – we turn now to begin the discussion of OCaml as a language of types and values.



# 4

## *Values and types*

OCaml is a

- value-based,
- strongly, statically, implicitly typed,
- functional

programming language. In this chapter, we introduce these aspects of the language.

### *4.1 OCaml expressions have values*

The OCaml language is, at its heart, a language for calculating *values*. The expressions of the language specify these values, and the process of calculating the value of an expression is termed EVALUATION. We've already seen examples of OCaml evaluating some simple expressions in Chapter 2:

```
# 3 + 4 * 5 ;;
- : int = 23
# (3 + 4) * 5 ;;
- : int = 35
```

The results of these evaluations are integers, and the output printed by the REPL indicates this by the `int`, about which we'll have more to say shortly.

#### *4.1.1 Integer values and expressions*

Integer values are built using a variety of operators and functions. We've seen the standard arithmetic operators for integer addition (+), subtraction (-), multiplication (\*), and division (/). Integer negation is with the `~-` operator (a tilde followed by a hyphen), which is kept distinct from the subtraction operator for clarity.

A full set of built-in operators is provided in OCaml's **Stdlib module**, one of a large set of OCaml library modules that provide a range of functions. The **Stdlib** module is OCaml's "standard library" in the sense that the values it provides can be referred to anywhere without any additional qualification, whereas values from other modules require a prefix, for example, `List.length` or `Hashtbl.create`.<sup>1</sup> You'll want to look over the **Stdlib** module documentation to get a sense of what is available.

Here are some examples of integer expressions using these operators:

```
# 1001 / 365;;
- : int = 2 (* # of years in 1001 nights *)

# 1001 mod 365;;
- : int = 271 (* # of nights left over *)

# 1001 - (1001 / 365) * 365;;
- : int = 271 (* ...or alternatively *)
```

Notice the use of comments to document the intentions behind the calculations.

#### 4.1.2 Floating point values and expressions

In addition to integers, OCaml provides other kinds of values. Real numbers can be represented using a **floating point** approximation.

**Floating point literals** can be expressed in several ways, using decimal notation (3.14), with an exponent (314e-2), and even in hexadecimal (0x1.91eb851eb851fp+1).

```
# 3.14;;
- : float = 3.14

# 314e-2;;
- : float = 3.14

# 0x1.91eb851eb851fp+1;;
- : float = 3.14
```

Floating point expressions can be built up with a variety of operators, including addition (+.), subtraction (-.), multiplication (\*.), division (/.), and negation (~-.). Again, the **Stdlib** module provides a fuller set, including operators for square root (`sqrt`) and various kinds of rounding (`floor` and `ceil`).

```
# 3.14 *. 2. *. 2.;; (* area of circle of radius 2 *)
- : float = 12.56

# ~-. 5e10 /. 2.718;;
- : float = -18395879323.0316429
```

Notice that the floating point operators are distinct from those for integers. Though this will take some getting used to, the reason for this design decision in the language will become apparent shortly.

<sup>1</sup> There is nothing special going on with **Stdlib**. It's just that by default, the **Stdlib** module is "opened", whereas other library modules like `List` and `Hashtbl` are not. The behavior of modules will become clear when they are fully introduced in Chapter 12.

**Exercise 8**

Use the OCaml REPL to calculate the value of the GOLDEN RATIO,

$$\frac{1 + \sqrt{5}}{2}$$

a proportion thought to be especially pleasing to the eye (Figure 4.1).

You'll want to use the built in `sqrt` function for floating point numbers. Be careful to use floating point literals and operators. If you find yourself confronted with errors in solving this exercise, come back to it after reading Section 4.2.

#### 4.1.3 Character and string values

As in many programming languages, text is represented as strings of CHARACTERS. Character literals are given in single quotes, for instance, `'a'`, `'X'`, `'3'`. Certain special characters can be specified only by escaping them with a backslash, for instance, the single-quote character itself `'\ '` and the backslash `'\\'`, as well as certain whitespace characters like newline `'\n'` or tab `'\t'`.

String literals are given in double quotes (with special characters similarly escaped), for instance, `""`, `"first"`, `" and second"`. They can be concatenated with the `^` operator.<sup>2</sup>

```
# "" ^ "first" ^ " and second" ;;
- : string = "first and second"
```

#### 4.1.4 Truth values and expressions

There are two TRUTH VALUES, indicated in OCaml by the literals `true` and `false`. Logical reasoning based on truth values was codified by the British mathematician **George Boole** (1815–1864), leading to the use of the term *boolean* for such values, and the type name `bool` for them in OCaml.

Just as arithmetic values can be operated on with arithmetic operators, the truth values can be operated on with logical operators, such as operators for conjunction (`&&`), disjunction (`||`), and negation (`not`). (See Section B.3 for definitions of these operators.)

```
# false ;;
- : bool = false
# true || false ;;
- : bool = true
# true && false ;;
- : bool = false
# true && not false ;;
- : bool = true
```

The equality operator `=` tests two values<sup>3</sup> for equality, returning `true` if they are equal and `false` otherwise. There are other COMPARISON OPERATORS as well: `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal), `<>` (not equal).



Figure 4.1: A rectangle with width and height in the golden ratio.

<sup>2</sup> A useful trick is to use the escape sequence of a backslash, a newline, and any amount of whitespace, all of which will be ignored, so as to split a string over multiple lines. For instance,

```
# "First, " ^ "second, \
#           third, \
#           and fourth." ;;
- : string = "First, second, third, and fourth."
```

<sup>3</sup> This is the first example of a function that can apply to values of different types, a powerful idea that we will explore in detail in Chapter 9.

```
# 3 = 3 ;;
- : bool = true
# 3 > 4 ;;
- : bool = false
# 1 + 1 = 2 ;;
- : bool = true
# 3.1416 = 314.16 /. 100. ;;
- : bool = false
# true = false ;;
- : bool = false
# true = not false ;;
- : bool = true
# false < true ;;
- : bool = true
```

**Exercise 9**

Are any of the results of these comparisons surprising? See if you can figure out why the results are that way.

Of course, the paradigmatic use of truth values is in the ability to compute different values depending on the truth or falsity of a condition. The OCaml CONDITIONAL expression is structured as follows:<sup>4</sup>

$$\langle \text{expr} \rangle ::= \text{if } \langle \text{expr}_{\text{test}} \rangle \text{ then } \langle \text{expr}_{\text{true}} \rangle \text{ else } \langle \text{expr}_{\text{false}} \rangle$$

The value of such an expression is the value of the  $\langle \text{expr}_{\text{true}} \rangle$  if the value of the test expression  $\langle \text{expr}_{\text{test}} \rangle$  is true and the value of the  $\langle \text{expr}_{\text{false}} \rangle$  if the value of  $\langle \text{expr}_{\text{test}} \rangle$  is false.

```
# if 3 = 3 then 0 else 1 ;;
- : int = 0
# 2 * if 3 > 4 then 3 else 4 + 5 ;;
- : int = 18
# 2 * (if 3 > 4 then 3 else 4) + 5 ;;
- : int = 13
```

## 4.2 OCaml expressions have types

We've introduced these additional values grouped according to their use. Integers are the type of things that integer operations are appropriate for; floating point numbers are the type of things that floating point operations are appropriate for; truth values are the type of things that logical operations are appropriate for. And conversely, it makes no sense to apply operations to values for which they are not appropriate. Therefore, OCaml is a TYPED language. Every expression of the language is associated with a type.

Using values in ways inconsistent with their type is perilous. The maiden flight of the Ariane 5 rocket on June 4, 1996 ended **spectacularly** 37 seconds after launch when the rocket self-destructed. The

<sup>4</sup> We describe the syntax of the construct using the BNF rule notation introduced in Chapter 3. We will continue to do so throughout as we introduce new constructs of the language.

As mentioned in footnote 2 on page 34, in defining expression classes using this notation, we use subscripts to differentiate among different occurrences of the same expression class, as we have done here with the three instances of the  $\langle \text{expr} \rangle$  class –  $\langle \text{expr}_{\text{test}} \rangle$ ,  $\langle \text{expr}_{\text{true}} \rangle$ , and  $\langle \text{expr}_{\text{false}} \rangle$ .



Figure 4.2: Small inconsistencies can lead to major problems: The explosion of the Ariane 5 on June 4, 1996.

reason? A floating-point value was used as an integer, causing an implicit conversion that overflowed. Using values in inappropriate ways is a frequent source of bugs in code, even if not with the dramatic aftermath of the Ariane 5 explosion. As we will see, associating values with types can often prevent these kinds of bugs.

The OCaml language is **STATICALLY TYPED**, in that the type of an expression can be determined just by examining the expression in its context. It is not necessary to run the code in which an expression occurs in order to determine the type of an expression, as might be necessary in a **DYNAMICALLY TYPED** language (Python or JavaScript, for instance).

Types are themselves a powerful abstraction mechanism. Types are essentially *abstract values*. By reasoning about the types of expressions, we can convince ourselves of the correctness of code without having to run it.

Furthermore, OCaml is **STRONGLY TYPED**; values may not be used in ways inappropriate for their type. One of the ramifications of OCaml's strong typing is that functions only apply to values of certain types and only return values of certain types. For instance, the addition function specified by the + operator expects integer arguments and returns an integer result.

By virtue of strong, static typing, the programming system (compiler or interpreter) can tell the programmer when type constraints are violated *even before the program is run*, thereby preventing bugs before they happen. If you attempt to use a value in a manner inconsistent with its type, OCaml will complain with a typing error. For instance, integer multiplication can't be performed on floating point numbers or strings:

```
# 5 * 3 ;;
- : int = 15
# 5 * 3.1416 ;;
Line 1, characters 4-10:
1 | 5 * 3.1416 ;;
          ^^^^^^
Error: This expression has type float but an expression was
expected of type
           int
# "five" * 3 ;;
Line 1, characters 0-6:
1 | "five" * 3 ;;
          ^^^^^^
Error: This expression has type string but an expression was
expected of type
           int
```

Programmers using a language with strong static typing for the first time often find the frequent type errors limiting and even annoying.

Type	Type expression	Example values			An example expression
integers	int	1	-2	42	(3 + 4) * 5
floating point numbers	float	3.14	-2.	2e12	(3.0 +. 4.) *. 5e0
characters	char	'a'	'&'	'\n'	char_of_int (int_of_char 's')
strings	string	"a"	"3 + 4"		"re" ^ "bus"
truth values	bool	true	false		true && not false
unit	unit	()			ignore (3 + 4)

Furthermore, there are some computations that can't be expressed well with such strict limitations, especially low-level systems computations that need access to the underlying memory representation of values. But a type error found at compile time is a warning that data use errors could show up at run time after the code has been deployed – and when it's far too late to repair it. Strong static type constraints are thus an example of a language restraint that frees programmers from verifying that their code does not contain “bad” operations by empowering the language interpreter to do so itself. (Looking ahead to the edict of prevention in Chapter 11, it makes the illegal inexpressible.)

Table 4.1: Some of the atomic OCaml types with example values and an example expression.

#### 4.2.1 Type expressions and typings

In OCaml, every type has a “name”. These names are given as TYPE EXPRESSIONS, a kind of little language for naming types. Just as there are value expressions for specifying values, there are type expressions for specifying types.

In this language of type expressions, each ATOMIC TYPE has its own name. We've already seen the names of the integer, floating point, and truth value types – `int`, `float`, and `bool`, respectively – in the examples earlier in this chapter, because the REPL prints out a type expression for a value's type along with the value itself, for instance,

```
# 42 ;;
- : int = 42
# 3.1416 ;;
- : float = 3.1416
# false ;;
- : bool = false
```

Notice that the REPL presents the type of each computed value after a colon (:). (Why a colon? You'll see shortly.)

Table 4.1 provides a more complete list of some of the atomic types in OCaml (some not yet introduced), along with their type names, some example values, and an example expression that specifies a value of that type using some functions that return values of the given type.

(We'll get to non-atomic (composite) types in Chapter 7.)

It is often useful to note that a certain expression is of a certain type. Such a TYPING is notated in OCaml using the `:` operator, placing the value to the left of the operator and its type to the right. So, for instance, the following typings hold:

- `42 : int`
- `true : bool`
- `3.14 *. 2. *. 2. : float`
- `if 3 > 4 then 3 else 4 : int`

The first states that the expression `42` specifies an integer value, the second that `true` specifies a boolean truth value, and so forth. The `:` operator is sometimes read as “the”, thus “`42`, the integer” or “`true`, the bool”. The typing operator is special in that it combines an expression from the value language (to its left) with an expression from the type language (to its right).

We can test out these typings right in the REPL. (The parentheses are necessary.)

```
# (42 : int) ;;
- : int = 42
# (true : bool) ;;
- : bool = true
# (3.14 *. 2. *. 2. : float) ;;
- : float = 12.56
# (if 3 > 4 then 3 else 4 : int) ;;
- : int = 4
```

The REPL generates an error when a value is claimed to be of an inappropriate type.

```
# (42 : float) ;;
Line 1, characters 1-3:
1 | (42 : float) ;;
^
Error: This expression has type int but an expression was expected
      of type
          float
Hint: Did you mean `42.'?
```

### Exercise 10

Which of the following typings hold?

1. `3 + 5 : float`
2. `3. + 5. : float`
3. `3. +. 5. : float`
4. `3 : bool`

```
5. 3 || 5 : bool
6. 3 || 5 : int
```

Try typing these into the REPL to see what happens. (Remember to surround them with parentheses.)

Finally, in OCaml, expressions are **IMPLICITLY TYPED**. Although all expressions have types, and the types of expressions can be annotated using typings, *the programmer doesn't need to specify those types* in general. Rather, the OCaml interpreter can typically deduce the types of expressions at compile time using a process called **TYPE INFERENCE**. In fact, the examples shown so far depict this inference. The REPL prints not only the value calculated for each expression but also the type that it inferred for the expression.

### 4.3 The unit type

In OCaml, the phrases of the language are expressions, expressing values. In many other programming languages, the phrases of the language are not always used to express values. Rather, they are used as commands. They are of interest because of what they *do*, not what they *are*. This approach is especially prevalent in **imperative programming**, the term ‘imperative’ deriving from the Latin ‘*imperativus*’, meaning ‘pertaining to a command’. But OCaml, like other functional languages, is uniform in privileging expressions over commands.

Occasionally, we have an expression that really need compute no value. But since every expression has to have a value in OCaml, we need to assign a value to such expressions as well. In this case, we use the value `()`, spelled with an open and close parenthesis and pronounced “unit”. This value is the only value of the type `unit`. Since the `unit` type has only one value, that value conveys no information, which is just what we want as the value of an expression whose value is irrelevant. The `unit` type will feature more prominently once we explore imperative programming within OCaml in Chapter 15.

#### Exercise 11

Give a typing for a value of the `unit` type.

### 4.4 Functions are themselves values

Functions play a central role in OCaml. They serve as the primary programming abstraction, as they do in many languages.

In a mathematical sense, a **FUNCTION** is simply a mapping from an input (called the function’s **ARGUMENT**) to an output (the function’s **VALUE**). Some functions that are built into OCaml are depicted in Figure 4.3.

⋮	⋮
<code>-2 → -1</code>	<code>-2 → "-2"</code>
<code>-1 → 0</code>	<code>-1 → "-1"</code>
<code>0 → 1</code>	<code>0 → "0"</code>
<code>1 → 2</code>	<code>1 → "1"</code>
<code>2 → 3</code>	<code>2 → "2"</code>
⋮	⋮
(a)	(b)
	(c)

Figure 4.3: Three example functions:  
 (a) the function from integers to their successors, available in OCaml as the `succ` function; (b) the function from integers to their string representation, available in OCaml as the `string_of_int` function; (c) the function mapping each boolean value onto its negation, available as the `not` function in OCaml.

OCaml is a FUNCTIONAL PROGRAMMING LANGUAGE. By this we mean more than that functions play a central role in the language. We mean that functions are FIRST-CLASS VALUES – they can be passed as arguments to functions or returned as the value of functions. Functions that take functions as arguments or return functions as values are referred to as HIGHER-ORDER FUNCTIONS, and the powerful programming paradigm that makes full use of this capability, which we will introduce in Chapter 8, is HIGHER-ORDER FUNCTIONAL PROGRAMMING.

Related to the idea that functions are values is that they have types as well. In Exercise 8, you used the `sqrt` function to take the square root of a floating point number. This function, `sqrt`, is itself a value and has a type. The type of a function expresses both the type of its argument (in this case, `float`) and the type of its output (again `float`). The type expression for a function (the type's “name”) is formed by placing the symbol `->` (read “arrow” or “to”) between the argument type and the output type. Thus the type for `sqrt` is `float -> float` (read “float arrow float” or “float to float”), or, expressed as a typing, `sqrt : float -> float`.

You can verify this typing yourself, just by evaluating `sqrt`:

```
# sqrt ;;
- : float -> float = <fun>
```

Since functions are themselves values, they can be evaluated, and the REPL performs type inference and provides the type `float -> float` along with a printed representation of the value itself `<fun>`, indicating that the value is a function of some sort.<sup>5</sup>

Because the argument type of `sqrt` is `float`, it can only be applied to values of that type. And since the result type of `sqrt` is `float`, only functions that take `float` arguments can apply to expressions like `sqrt 42..`

#### **Exercise 12**

What are the types of the three functions – `succ`, `string_of_int`, and `not` – from Figure 4.3?

#### **Exercise 13**

Try applying the `sqrt` function to an argument of some type other than `float`, for instance, a value of type `bool`. What happens?

Of course, the real power in functional programming comes from defining your own functions. We'll move to this central topic in Chapter 6, but first, it is useful to provide a means of *naming* values (including functions), to which we turn in the next chapter.

<sup>5</sup>The actual value of a function is a complex data object whose internal structure is not useful to print, so this abstract presentation `<fun>` is printed instead.



# 5

## *Naming and scope*

In this chapter, we introduce the ability to give names to values, an ability with multiple benefits.

### *5.1 Variables are names for values*

We introduced the concept of a variable in Chapter 1 as seen in the imperative programming paradigm – the variable as a locus of mutable state, which takes on different values over time. But in the functional paradigm, variables are better thought of simply as *names* for values. To introduce a name for a value for use in some other expression, OCaml provides the local naming expression, introduced by the keyword `let`:

$$\langle \text{expr} \rangle ::= \text{let } \langle \text{var} \rangle : \langle \text{type} \rangle = \langle \text{expr}_{\text{def}} \rangle \text{ in } \langle \text{expr}_{\text{body}} \rangle$$

In this construct,  $\langle \text{var} \rangle$  is a variable,<sup>1</sup> which will be the name of a value of the given  $\langle \text{type} \rangle$ ;  $\langle \text{expr}_{\text{def}} \rangle$  is an expression defining a value of the given  $\langle \text{type} \rangle$ ; and  $\langle \text{expr}_{\text{body}} \rangle$  is an expression within which the variable can be used as the name for the defined value. The expression as a whole specifies whatever the  $\langle \text{expr}_{\text{body}} \rangle$  evaluates to. We say that the construction BINDS the name  $\langle \text{var} \rangle$  to the value  $\langle \text{expr}_{\text{def}} \rangle$  for use in  $\langle \text{expr}_{\text{body}} \rangle$ .<sup>2</sup> For this reason, the `let` expression is referred to as a BINDING CONSTRUCT. We'll introduce other binding constructs in Chapters 6 and 7.

As an example,<sup>3</sup> we might provide a name for the important constant  $\pi$  in the context of calculating the area of a circle of radius 2:

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

Informally speaking (and we'll provide a more rigorous description in Chapter 13), the construct operates as follows: The  $\langle \text{expr}_{\text{def}} \rangle$  expression

<sup>1</sup> Variables in OCaml are required to be sequences of alphabetic and numeric characters along with the underscore character (`_`) and the prime character (`'`). The first character in the variable name must be alphabetic or an underscore. The special role of the latter case is discussed later in Section 7.2.

<sup>2</sup> The name being defined is sometimes referred to as the DEFINIENDUM, the expression it names being the DEFINIENS.

<sup>3</sup> In these examples, we follow the stylistic guidelines described in Section C.1.7 in indenting the body of a `let` to the same level as the `let` keyword itself. The rationale is provided there.

is evaluated to a value, and then the  $\langle expr_{body} \rangle$  is evaluated, but as if occurrences of the definiendum  $\langle var \rangle$  were first replaced by the value of the definiens  $\langle expr_{def} \rangle$ .

Notice how by naming the value `pi`, we document our intention that the value serves as the mathematical constant,  $\pi$ , consistent with the edict of intention.

## 5.2 *The type of a let-bound variable can be inferred*

It may seem obvious to you that in an expression like

```
let pi : float = 3.1416 in
  pi *. 2. *. 2. ;;
```

the variable `pi` is of type `float`. What else could it be, given that its value is a `float` literal, and it is used as an argument of the `*.` operator, which takes `float` arguments? You would be right, and OCaml itself can make this determination, inferring the type of `pi` without the explicit typing being present. For that reason, the type information in the `let` construct is optional. We can simply write

```
let pi = 3.1416 in
  pi *. 2. *. 2. ;;
```

and the calculation proceeds as usual. This ability to infer types is what we mean when we say (as in Section 4.2.1) that OCaml is implicitly typed.

Although these typings when introducing variables are optional, nonetheless, it can still be useful to provide explicit type information when naming a value. First, (and again following the edict of intention), it allows the programmer to make clear the intended types, so that the OCaml interpreter can verify that the programmer's intention was followed and so that readers of the code are aware of that intention. Second, there are certain (relatively rare) cases (Section 9.6) in which OCaml cannot infer a type for an expression in context; in such cases, the explicit typing is necessary.

## 5.3 *let expressions are expressions*

Remember that all expressions in OCaml have values, even `let` expressions. Thus we can use them as subexpressions of larger expressions.

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

**Exercise 14**

Are the parentheses necessary in this example? Try out the expression without the parentheses and see what happens.

A particularly useful application of the fact that `let` expressions can be used as first-class values is that they may be embedded in other `let` expressions to get the effect of defining multiple names. Here, we define both the constant  $\pi$  and a radius to calculate the area of a circle of radius 4:

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

**Exercise 15**

Use the `let` construct to improve the readability of the following code to calculate the length of the `hypotenuse` of a particular right triangle:

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

## 5.4 Naming to avoid duplication

We introduce an extended example to more crisply demonstrate the advantages of naming. Suppose we wanted to determine the area of the larger of the two triangles in Figure 5.1.

To demonstrate some of the advantages of naming, we attempt to calculate the area of the larger without recourse to the `let` construct. To calculate the areas, we'll use a [method attributed to Heron of Alexandria](#) around 60 CE.

Calculating the area of the larger triangle without defining local names is possible, but ungainly:

```
1 # if sqrt ( ((1. +. 1. +. 1.41) /. 2.)
2 #           *. ((1. +. 1. +. 1.41) /. 2. -. 1.))
3 #           *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
4 #           *. ((1. +. 1. +. 1.41) /. 2. -. 1.41) )
5 #   > sqrt ( ((1.5 +. 0.75 +. 2.) /. 2.)
6 #             *. ((1.5 +. 0.75 +. 2.) /. 2. -. 1.5)
7 #             *. ((1.5 +. 0.75 +. 2.) /. 2. -. 0.55)
8 #             *. ((1.5 +. 0.75 +. 2.) /. 2. -. 2.)) )
9 # then
10 #   sqrt ( ((1. +. 1. +. 1.41) /. 2.)
11 #           *. ((1. +. 1. +. 1.41) /. 2. -. 1.))
12 #           *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
13 #           *. ((1. +. 1. +. 1.41) /. 2. -. 1.41) )
14 # else
15 #   sqrt ( ((1.5 +. 0.75 +. 2.) /. 2.)
16 #           *. ((1.5 +. 0.75 +. 2.) /. 2. -. 1.5)
17 #           *. ((1.5 +. 0.75 +. 2.) /. 2. -. 0.75)
18 #           *. ((1.5 +. 0.75 +. 2.) /. 2. -. 2.)) ;;
- : float = 0.47777651606895504
```

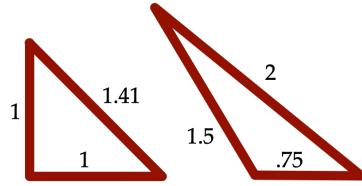


Figure 5.1: Two triangles, the left with sides of length 1, 1, and 1.41, and the right with sides of length 1.75, .75, and 2. Which has the larger area?

It's extraordinarily difficult to tell what's going on in this code. Certainly, the various side lengths appear repeatedly, and in fact, calculations making use of them repeat as well. Lines 1–4 and 10–13 both separately calculate the area of the left triangle in the figure, and lines 5–8 and 15–18 calculate the area of the right triangle. The calculations are redundant, and worse, provide the opportunity for bugs to creep in if the copies aren't kept in perfect synchrony.

Appropriate use of naming can partially remedy these problems. (We'll address their solution more systematically in Chapters 6 and 8.) First, by naming the two area calculations, we need calculate each only once.

```
# let left_area = sqrt ( ((1. +. 1. +. 1.41) /. 2.)
#                      *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
#                      *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
#                      *. ((1. +. 1. +. 1.41) /. 2. -. 1.41) ) in
# let right_area = sqrt ( ((1.5 +. 0.75 +. 2.) /. 2.)
#                        *. ((1.5 +. 0.75 +. 2.) /. 2. -. 1.5)
#                        *. ((1.5 +. 0.75 +. 2.) /. 2. -. 0.75)
#                        *. ((1.5 +. 0.75 +. 2.) /. 2. -. 2.) ) in
# if left_area > right_area then left_area else right_area ;;
- : float = 0.499991149296665216
```

We also correct a bug in line 7, which you may not have noticed, that uses inconsistent values for one of the side lengths in the area calculations. By defining the area once and using the value twice, we remove the possibility for such inconsistencies to even arise.

Finally, notice the repeated calculation of, for instance,  $(1. +. 1. +. 1.41) /. 2.$ , which is calculated some four times, and similarly for  $(1.5 +. 0.75 +. 2.) /. 2.$ . Each of these is the SEMIPERIMETER of a triangle (that is, half the perimeter). The semiperimeter features heavily in Heron's method of calculating triangle areas. By naming these two subexpressions, we clarify even further what is going on in the example.

```
# let left_area =
#   let left_sp = (1. +. 1. +. 1.41) /. 2. in
#   sqrt ( left_sp
#          *. (left_sp -. 1.)
#          *. (left_sp -. 1.)
#          *. (left_sp -. 1.41) ) in
# let right_area =
#   let right_sp = (1.5 +. 0.75 +. 2.) /. 2. in
#   sqrt ( right_sp
#          *. (right_sp -. 1.5)
#          *. (right_sp -. 0.75)
#          *. (right_sp -. 2.) ) in
# if left_area > right_area then left_area else right_area ;;
- : float = 0.499991149296665216
```

There's still much room for improvement, but to make further

progress on this example awaits additional techniques beyond naming, as described in Section 6.5.

## 5.5 Scope

The name defined in the `let` expression is available only in the body of the expression. The name is LOCAL to the body, and unavailable outside of the body. We say that the SCOPE of the variable – that is, the code region within which the variable is available as a name of the defined value – is the body of the `let` expression. This explains the following behavior:

```
# (let s = "hi ho " in
#   s ^ s) ^ s ;;
Line 2, characters 9-10:
2 | s ^ s) ^ s ;;
          ^
Error: Unbound value s
```

The body of the `let` expression in this example ends at the closing parenthesis, and thus the variable `s` defined by that construct is unavailable (“unbound”) thereafter.

### Exercise 16

Correct the example to provide the triple concatenation of the defined string.

### Exercise 17

What type do you expect is inferred for `s` in the example?

In particular, the scope of a local `let` naming does not include the definition itself (the  $\langle \text{expr}_{\text{def}} \rangle$  part between the `=` and the `in`). Thus the following expression is ill-formed:

```
# let x = x + 1 in
# x * 2 ;;
Line 1, characters 8-9:
1 | let x = x + 1 in
          ^
Error: Unbound value x
```

And a good thing too, for what would such an expression mean? This kind of recursive definition isn’t well founded. Nonetheless, there *are* useful recursive definitions, as we will see in Section 6.6.

What if we define the same name twice? There are several cases to consider. Perhaps the two uses are disjoint, as in this example:

```
# sqrt ((let x = 3. in x *. x)
#       +. (let x = 4. in x *. x)) ;;
- : float = 5.
```

Since each `x` is introduced with its own `let` and has its own body, the scopes are disjoint. The occurrences of `x` in the first expression name

the number 3 . and in the second name the number 4 .. But in the following case, the scopes are not disjoint:

```
# sqrt (let x = 3. in
#      x *. x +. (let x = 4. in [x] *. [x])) ;;
- : float = 5.
```

The scope of the first `let` encompasses the entire second `let`. Do the highlighted occurrences of `x` in the body of the second `let` name 3 . or 4 . ? The rule used in OCaml (and most modern languages) is that the occurrences are bound by the *nearest enclosing binding construct for the variable*. The same binding relations hold as if the inner `let`-bound variable `x` and the occurrences of `x` in its body were uniformly renamed, for instance, as `y`:

```
# sqrt (let x = 3. in
#      x *. x +. (let y = 4. in y *. y)) ;;
- : float = 5.
```

By virtue of this convention that variables are bound by the closest binder, when an inner binder for a variable falls within the scope of an outer binder for the same variable, the outer variable is inaccessible in the inner scope. We say that the outer variable is SHADOWED by the inner variable. For instance, in

```
# let x = 1 in
# [x] + let x = 2 in
#   [x] + let x = 4 in
#     [x] ;;
- : int = 7
```

the innermost `x` (naming 4) shadows the outer two, and the middle `x` (naming 2) shadows the outer `x` (naming 1). Thus the three highlighted occurrences of `x` name 1, 2, and 4, respectively, which the expression as a whole sums to 7.

Since the scope of a `let`-bound variable is the body of the construct, but not the definition, occurrences of the same variable in the definition must be bound outside of the `let`. Consider the highlighted occurrence of `x` on the second line:

```
let x = 3 in
let x = [x] * 2 in
x + 1 ;;
```

This occurrence is bound by the `let` in line 1, not the one in line 2. That is, it is equivalent to the renaming

```
let x = 3 in
let y = x * 2 in
y + 1 ;;
```

**Exercise 18**

For each occurrence of the variable `x` in the following examples, which `let` construct binds it? Rewrite the expressions by renaming the variables to make them distinct while preserving the bindings.

1. `let x = 3 in  
let x = 4 in  
x * x ;;`
2. `let x = 3 in  
let x = x + 2 in  
x * x ;;`
3. `let x = 3 in  
let x = 4 + (let x = 5 in x) + x in  
x * x ;;`

## 5.6 Global naming and top-level let

The `let` construct introduced above introduces a local name, local in the sense that its scope is just the body of the `let`. OCaml provides a global naming construct as well, defined by this BNF rule:<sup>4</sup>

$\langle \text{definition} \rangle ::= \text{let } \langle \text{var} \rangle : \langle \text{type} \rangle = \langle \text{expr}_{\text{def}} \rangle$

By simply leaving off the ‘`in`  $\langle \text{expr}_{\text{body}} \rangle$ ’ part of the `let` construct, the name can continue to be used thereafter; the scope of the naming extends all the way through the remainder of the REPL session or to the end of the program file.

```
# let pi = 3.1416 ;;
val pi : float = 3.1416
# let radius = 4.0 ;;
val radius : float = 4.
# pi *. radius *. radius ;;
- : float = 50.2656
# 2. *. pi *. radius ;;
- : float = 25.1328
```

The REPL indicates that new names have been introduced by presenting typings for the names (`pi : float` or `radius : float`) as well as displaying their values.

This global naming may look a bit like assignment in imperative languages. We can have, for instance,

```
# let x = 3 ;;
val x : int = 3
# let x = x + 1 ;;
val x : int = 4
# x + x ;;
- : int = 8
```

The second line may look like it is assigning a new value to `x`. But no, all that is happening is that there is a new name (coincidentally the same as a previous name) for a new value. The old name `x` for the value

<sup>4</sup> Unlike the local naming construct, the global naming construct expressed in this BNF rule is not an expression (that is, of syntactic class  $\langle \text{expr} \rangle$ ). Rather, we categorize it as a DEFINITION (of syntactic class  $\langle \text{definition} \rangle$ ). Such definitions are allowed only at the top level of program files or the REPL.

`3` is still around; it's just inaccessible, shadowed by the new name `x`. (In Chapter 15, we provide a demonstration that this is so.)

#### Exercise 19

In the sequence of expressions

```
let tax_rate = 0.05 ;;
let price = 5. ;;
let price = price * (1. +. tax_rate) ;;
price ;;
```

what is the value of the final expression? (You can use the REPL to verify your answer.)

Global naming is available only at the top level. A global name cannot be defined from within another expression, for instance, the body of a local `let`. The following is thus not well-formed:

```
# let radius = 4. in
# let pi = 3.1416 in
# let area = pi *. radius ** 2. ;;
Line 3, characters 30-32:
3 | let area = pi *. radius ** 2. ;;
                           ^^^
Error: Syntax error
```

#### Exercise 20

How might you get the effect of this definition of a global variable `area` by making use of local variables for `pi` and `radius`?



We alluded to the fact that in OCaml, functions are first-class values, and as such they can be named as well. In fact, the ability to name values becomes most powerful when the named values are functions. In the next chapter, we introduce functions and function application in OCaml, and start to demonstrate the power of functions as an abstraction mechanism.

# 6

## *Functions*

Recall that abstraction is the process of viewing a set of apparently dissimilar things as instantiating an underlying identity. Plato in his *Phaedrus* has Socrates adduce two rhetorical principles. The first Socrates describes as

That of perceiving and bringing together in one idea the scattered particulars, that one may make clear by definition the particular thing which he wishes to explain. ([Plato, 1927](#))

that is, a principle of abstraction. (Socrates's second principle shows up in Chapter 8.)

Abstraction in programming is this process applied to code, and can be enabled by appropriate language constructs. Programming abstraction is important because it enables programmers to satisfy perhaps the most important edict of programming:

*Edict of irredundancy:  
Never write the same code twice.*

A standard technique that beginning programmers use is “cut and paste” programming – you find some code that does more or less what you need, perhaps code you’ve written before, and you cut and paste it into your program, adjusting as necessary for the context the code now appears in. There is a high but mostly hidden cost to the cut and paste approach. If you find a bug in one of the copies, it needs to be fixed in all of the copies. If some functionality changes in one of the copies, the other copies don’t benefit unless they are modified too. As documentation is added to clarify one of the copies, it must be maintained for all of them. When one of the copies is tested, no assurance is thereby gained for the other copies. There’s a theme here. Having written the same code twice, all of the problems of debugging, maintaining, documenting, and testing code have been similarly multiplied.

The edict of irredundancy is the principle of avoiding the problems introduced by duplicative code. Rather than write the same code twice, the edict calls for viewing the apparently dissimilar pieces of code as instantiating an underlying identity, and factoring out the common parts using an appropriate abstraction mechanism.

Given the emphasis in the previous chapters, it will be unsurprising to see that the abstraction mechanism we turn to for satisfying the edict of irredundancy is the function itself. but before getting there, there is much to be introduced about how functions are defined and used in OCaml.

We will thus introduce how OCaml supports functions, their application and their definition, including some notational issues that simplify writing functions and connections to the typing constraints that make sure that code works properly. Then, we'll have the tools to provide an example of how functions can factor out redundancies from code in keeping with the edict of irredundancy. Finally, we'll extend the expressivity of functions even further with recursive functions, and introduce the idea of unit testing of functions to help verify their correctness.

## 6.1 *Function application*

We introduced functions in Section 4.4 as mappings from an argument to the function's value at that argument. We can make use of a function by APPLYING it to its argument. You'll be most familiar with the traditional and ubiquitous mathematical notation for function application, in which a symbol naming the function precedes a parenthesized, comma-separated list of the arguments, as, for instance,  $f(1, 2, 3)$ .<sup>1</sup> It is thus perhaps surprising that OCaml doesn't use this notation for function application. Instead, it follows the notational convention proposed by Church in his lambda calculus. (See Section 1.2.) 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 becomes onerous. Instead, Church proposed merely prefixing the function to its argument. Instead of  $f(1)$ , Church's notation would have  $f\ 1$ . Instead of  $f(g(1))$ , he would have  $f\ (g\ 1)$ , using the parentheses for grouping, but not for demarcating the arguments.

Similarly, in OCaml, the function merely precedes its argument. The successor of 41 is thus simply `succ 41`. The square root of two is `sqrt 2.0`.

```
# succ 41 ;;
- : int = 42
```

<sup>1</sup> Some historical background on this notation is provided in Section B.1.2.

```
# sqrt 2.0 ;;
- : float = 1.41421356237309515
```

Syntactically, we can codify that in a simple BNF rule for function application:

$$\langle \text{expr} \rangle ::= \langle \text{expr}_{\text{func}} \rangle \langle \text{expr}_{\text{arg}} \rangle$$

Recall from Section 4.4 that functions (as all values) have types, which can be expressed as type expressions using the `->` operator. For instance, the successor function `succ` has the type given by the type expression `int -> int` and the `string_of_int` function the type `int -> string`.

## 6.2 Multiple arguments and currying

The simple prefix notation for function application is only appropriate when functions take exactly one argument. But it turns out that this is not a substantial limitation in a system (like the lambda calculus and like OCaml) in which functions are themselves values. Suppose we have a function that we think of as taking multiple arguments simultaneously (like  $f(1, 2, 3)$ ). We can reconceptualize  $f$  as taking only one argument (in this case, the argument 1), returning a function that takes the second argument 2, again returning a function that takes the third and final argument 3, returning the final value. The type of such a function, which takes three integers returning an integer result, say, is thus

```
int -> (int -> (int -> int))
```

In essence, the function takes its three arguments *one at a time*, returning a function after each argument before the last. Although this trick was first discussed by Schönfinkel (1924), it is referred to as CURRYING a function, the resulting function being *curried*, so named after Haskell Curry who popularized the approach.

Because in OCaml functions take one argument, the language makes extensive use of currying, and language constructs facilitate its use. For instance, the `->` type expression operator is right associative (see Section 3.2) in OCaml, so that the type of the curried three-argument function above can be expressed as

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

Application, conversely, is left associative, so that applying a curried function  $f$  to its arguments can be notated  $f 1 2 3$  instead of  $((f 1) 2) 3$ .



Figure 6.1: Moses Schönfinkel (1889–1942), Russian logician and mathematician, first specified the use of higher-order functions to mimic the effect of multiple-argument functions.



Figure 6.2: Haskell Curry (1900–1982), American logician, promulgator of the use of higher-order functions to simulate functions of multiple arguments, which is referred to as *currying* in his honor.

We've already used some curried functions without noticing. The two-argument arithmetic and boolean operators, like `+`, `/.`, and `&&`, are curried. As usual, the REPL reveals their type:

```
# (+) ;;
- : int -> int -> int = <fun>
# (/.) ;;
- : float -> float -> float = <fun>
# (&&) ;;
- : bool -> bool -> bool = <fun>
```

Normally, we write these operators INFIX, placing the operator *between* its two arguments, but by placing the operator in parentheses<sup>2</sup> as we've done, the OCaml REPL interprets them as regular PREFIX functions, in which the function appears *before* its argument. Making use of this ability, they can even be applied in the one-by-one manner, as we've done here both parenthesized and unparenthesized:

```
# ((+) 3) 4 ;;
- : int = 7
# (+) 3 4 ;;
- : int = 7
```

### Exercise 21

What (if anything) are the types and values of the following expressions? Try to figure them out yourself before typing them into the REPL to verify your answer.

1. `(-) 5 3`
2. `5 - 3`
3. `- 5 3`
4. `"0" ^ "Caml"`
5. `(^) "0" "Caml"`
6. `(^) "0"`
7. `( ** )` – See footnote 2.

### 6.3 Defining anonymous functions

Now we get to the whole point of functional programming: *defining your own functions*. Suppose we want to specify a function that maps a certain input, call it `x`, to an output, say the doubling of `x`. The following expression does the trick: `fun x : int -> 2 * x`.

```
# fun x : int -> 2 * x ;;
- : int -> int = <fun>
```

The keyword `fun` introduces the function definition. The arrow `->` separates the typing of a variable that represents the input, the integer `x`, from an expression that represents the output value, `2 * x`. The output expression can, of course, make free use of the input variable as part of the computation.

<sup>2</sup> Care must be taken when parenthesizing the multiplication operators `*` and `.*` to convert them to prefix functions. Since OCaml comments are provided as `(* () *)`, parenthesizing as `(*)` will be misinterpreted as the beginning of a comment. To avoid this problem, place spaces between the parentheses and the operator: `( * )`.

We can apply this function to an argument (21, say). We use the usual OCaml prefix function application syntax, placing the function before its argument:

```
# (fun x : int -> 2 * x) 21 ;;
- : int = 42
```

Syntactically, we construct such a “function without a name”, an ANONYMOUS FUNCTION, with the OCaml `fun` construct, given by the following syntactic rule:<sup>3</sup>

$$\langle \text{expr} \rangle ::= \text{fun } \langle \text{var} \rangle : \langle \text{type} \rangle \rightarrow \langle \text{expr} \rangle$$

Here,  $\langle \text{var} \rangle$  is a variable, the name of the argument of the function, and  $\langle \text{expr} \rangle$  is an expression defining the output of the function, which will be of the given  $\langle \text{type} \rangle$ .

The `fun` construct, like the `let` construct, is a binding construct. The `fun` construct introduces a variable and binds occurrences of that variable in its scope. The scope of the variable is the body of the `fun`, the expression  $\langle \text{expr} \rangle$  after the arrow.

As was the case for `let` expressions, when the type of the variable can be inferred from how it is used in the definition part, as is typically the case, the typing part can be left off. So, for instance, the doubling function could be written

```
# fun x -> 2 * x ;;
- : int -> int = <fun>
```

and the same type `int -> int` still inferred.

#### **Exercise 22**

Try defining your own functions, perhaps one that squares a floating point number, or one that repeats a string.

### 6.4 Named functions

Now that we have the ability to define functions (with `fun`) and the ability to name values (with `let`), we can put them together to name newly-defined functions. Here, we give a global naming of the doubling function and use it:

```
# let double = fun x -> 2 * x ;;
val double : int -> int = <fun>
# double 21 ;;
- : int = 42
```

Here are **functions for the circumference and area of circles of given radius**:

<sup>3</sup> Warning: The same arrow symbol  $\rightarrow$  is used in defining both function *values* and function *types*. This sometimes leads to confusion. Be aware that though the same symbol is used for both, the two are quite distinct.

```

# let pi = 3.1416;;
val pi : float = 3.1416

# let area =
#   fun radius ->
#     pi *. radius ** 2. ;;
val area : float -> float = <fun>

# let circumference =
#   fun radius ->
#     2. *. pi *. radius ;;
val circumference : float -> float = <fun>

# area 4. ;;
- : float = 50.2656
# circumference 4. ;;
- : float = 25.1328

```

#### 6.4.1 Compact function definitions

This method for defining named functions, though effective, is a bit cumbersome. For that reason, OCaml provides a simpler syntax for defining functions, in which a definition for the calling pattern itself is provided. Instead of the phrasing

```
let <varfunc> = fun <vararg> -> <expr>
```

OCaml allows the following equivalent phrasing

```
let <varfunc> <vararg> = <expr>
```

This syntax for defining functions may be more familiar from other languages. It is also consistent with a more general pattern-matching syntax that we will come to in Section 7.2.

This compact syntax for function definition is an example of SYNTACTIC SUGAR,<sup>4</sup> a bit of additional syntax that serves to abbreviate a more complex construction. By adding some syntactic sugar, the language can provide simpler expressions without adding underlying constructs to the language; a language with a small core set of constructs can still have a sufficiently expressive concrete syntax that it is pleasant to program in. As we introduce additional syntactic sugar constructs, notice how they allow for idiomatic programming without increasing the core language.

We can use this more compact function definition notation to provide a more elegant definition of the doubling function:

```

# let double x = 2 * x ;;
val double : int -> int = <fun>
# double (double 3) ;;
- : int = 12

```

<sup>4</sup>The term “syntactic sugar” was first used by Landin (1964) (Figure 17.7) to describe just such abbreviatory constructs.

This compact notation applies to local definitions as well.

```
# let double x = 2 * x in
# double (double 3) ;;
- : int = 12
```

It even extends to multiple-argument curried functions. The definition

```
# let hypotenuse x =
#   sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

is syntactic sugar for (and hence completely equivalent to) the definition

```
# let hypotenuse =
#   fun x ->
#     fun y ->
#       sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

#### 6.4.2 Providing typings for function arguments and outputs

As in all definitions, you can provide a typing for the variable being defined, as in

```
# let hypotenuse : float -> float -> float =
#   fun x ->
#     fun y ->
#       sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

and it is good practice to do so for top-level definitions. That way, you are registering your intentions as to the types – remember the edict of intention? – and the language interpreter can verify that those intentions are satisfied. (See Section C.3.4.)

In the compact notation, typings can and should be provided for the application of the function to its arguments, as well as for the arguments itself. In the `hypotenuse` function definition, the application `hypotenuse x y` is of type `float`, which can be recorded as

```
# let hypotenuse x y : float =
#   sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

Each of the arguments can be explicitly typed as well.

```
# let hypotenuse (x : float) (y : float) : float =
#   sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

Here, we have recorded that `x` and `y` are each of `float` type, and the result of an application `hypotenuse x y` is also a `float`, which together

capture the full information about the type of hypotenuse itself. Consequently, the type inferred for the hypotenuse function itself is, as before, `float -> float -> float`, that is, a curried binary function from floats to floats.

### Exercise 23

Consider the following beginnings of function declarations. How would these appear using the compact notation (using whatever argument variable names you prefer)?

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

### Exercise 24

What are the types for the following expressions?

1. `let greet y = "Hello" ^ y in greet "World!" ;;`
2. `fun x -> let exp = 3. in x ** exp ;;`

### Exercise 25

Define a function square, using compact notation, that squares a floating point number.  
For instance,

```
# square 3.14 ;;
- : float = 9.8596
# square 1234567. ;;
- : float = 1524155677489.
```

### Exercise 26

Define a function abs : int -> int, using compact notation, that computes the absolute value of an integer.

```
# abs (-5) ;;
- : int = 5
# abs 0 ;;
- : int = 0
# abs (3 + 4) ;;
- : int = 7
```

### Exercise 27

The Stdlib.string\_of\_bool function returns a string representation of a boolean.

Here it is in operation:

```
# string_of_bool (3 = 3) ;;
- : string = "true"
# string_of_bool (0 = 3) ;;
- : string = "false"
```

What is the type of `string_of_bool`? Provide your own function definition for it.

### Exercise 28

Define a function even : int -> bool that determines whether its integer argument is an even number. It should return true if so, and false otherwise. Try using both the compact notation for the definition and the full desugared notation. Try versions with and without typing information for the function name.

### Exercise 29

Define a function circle\_area : float -> float that returns the area of a circle of a given radius specified by its argument. Try all of the variants described in Exercise 28.

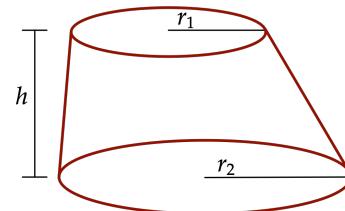


Figure 6.3: The frustum of a cone, with top and bottom radii  $r_1$  and  $r_2$  respectively, and height  $h$ .

**Exercise 30**

A frustum (Figure 6.3) is a three-dimensional solid formed by slicing off the top of a cone parallel to its base. The volume  $V$  of a frustum with radii  $r_1$  and  $r_2$  and height  $h$  is given by the formula

$$V = \frac{\pi h}{3} (r_1^2 + r_1 r_2 + r_2^2)$$

Implement a function to calculate the volume of a frustum given the radii and height.

**Problem 31**

The calculation of the date of Easter, a calculation so important to early Christianity that it was referred to simply as **COMPUTUS** ("the computation"), has been the subject of innumerable algorithms since the early history of the Christian church. An especially simple method, published in *Nature* in 1876 and attributed to "[A New York correspondent](#)" (1876), proceeds by sequentially calculating the following values on the basis of the year  $Y$ :

$$\begin{aligned} a &= Y \bmod 19 & h &= (19a + b - d - g + 15) \bmod 30 \\ b &= \frac{Y}{100} & i &= \frac{c}{4} \\ c &= Y \bmod 100 & k &= c \bmod 4 \\ d &= \frac{b}{4} & l &= (32 + 2e + 2i - h - k) \bmod 7 \\ e &= b \bmod 4 & m &= \frac{a + 11h + 22l}{451} \\ f &= \frac{b+8}{25} & month &= \frac{h + l - 7m + 114}{31} \\ g &= \frac{b-f+1}{3} & day &= ((h + l - 7m + 114) \bmod 31) + 1 \end{aligned}$$

Write two functions, `computus_month` and `computus_day`, which take an integer year argument and return, respectively, the month and day of Easter as calculated by the method above. Use them to verify that the date of Easter in 2018 was April 1.

## 6.5 Function abstraction and irredundancy

We have enough background in place to see directly how functions are key to obeying the edict of irredundancy. Recall the comparison of the areas of two triangles from Section 5.4. By appropriate use of naming subcalculations, the computation was defined as

```
# let left_area =
#   let left_sp = (1. +. 1. +. 1.41) /. 2. in
#     sqrt ( left_sp
#           *. (left_sp -. 1.)
#           *. (left_sp -. 1.)
#           *. (left_sp -. 1.41) ) in
# let right_area =
#   let right_sp = (1.5 +. 0.75 +. 2.) /. 2. in
#     sqrt ( right_sp
#           *. (right_sp -. 1.5)
#           *. (right_sp -. 0.75)
#           *. (right_sp -. 2.) ) in
# if left_area > right_area then left_area else right_area ;;
- : float = 0.499991149296665216
```

But some obvious redundancies remain. The calculation of `left_area` and `right_area` are structured identically, composed of first a

calculation of the semiperimeter for the three sides and then the area calculation itself, again using the three side lengths in corresponding places.

Of course, they are not strictly identical; if they were, we could just use the naming trick (Section 5.4) to remove the redundancy. However, except for the three side lengths, the two calculations are the same. The two area values involve the same computation over the side lengths, the same mapping from side lengths to area, the same *function* of the side lengths so to speak. We can view these two dissimilar expressions as manifesting an underlying identity by thinking of them as applications of one and the same function (call it `area`) to the three side lengths.

We start with a definition of this `area` function.

```
# let area x y z =
#   let sp = (x +. y +. z) /. 2. in
#   sqrt (sp *. (sp -. x) *. (sp -. y) *. (sp -. z)) ;;
val area : float -> float -> float -> float = <fun>
```

The two original computations of `left_area` and `right_area` match this pattern exactly, just with different values substituted for the three side lengths `x`, `y`, and `z`.

To generate these two instances, we apply the `area` function to the two sets of side lengths and compare the results as before.

```
# let left_area = area 1. 1. 1.41 in
# let right_area = area 1.5 0.75 2. in
# if left_area > right_area then left_area else right_area ;;
- : float = 0.499991149296665216
```

It is worth noting that this solution to the triangle area comparison problem specifies each of the six side lengths exactly once. Compare that with the initial version, in which each of the six side lengths appears ten times in the calculation, providing the risk of accidentally modifying some of the occurrences but not others and introducing bugs that way. Similarly, the definition of semiperimeter occurs once in this version, but 16 times in the original version. The definition of area by Heron's method appears only once here but four times in the original. This is the essence of abstraction, capturing the underlying idea once that unifies many instances.

We've now seen two abstraction techniques for eliminating redundancies. For trivial redundant expressions, exact duplications, it suffices to name the expression once and refer to it by its name multiple times. When the redundancy is a bit more subtle, involving systematic differences as to particular values in particular places, we can introduce a function that abstracts over those places, applying it to the particular values. But there are cases where mere substitution

of simple values (like in the area example) is not sufficient. The true power of functions comes in with these even more sophisticated cases, which we explore in detail in Chapter 8.

To prepare for those abstraction techniques, we extend the expressivity of functions even further by allowing functions to be defined in terms of themselves, recursive functions.

## 6.6 Defining recursive functions

Consider the FACTORIAL function, which maps its integer argument  $n$  onto the product of all the positive integers that are no larger than  $n$ . Thus, the factorial of 3, traditionally notated with a suffixed exclamation mark as  $3!$ , is the product of 1, 2, and 3, that is, 6; and  $4!$  is 24. Notice that  $4!$  is  $4 \cdot 3!$ , which makes sense because  $3!$  has already incorporated all the integers up to 3, so the only remaining integer to multiply in is 4 itself. Indeed, in general,

$$n! = n \cdot (n - 1)!$$

for all integers  $n$  greater than 1, and if we take the value of  $0!$  to be 1, the equation even holds for  $n = 1$ . This serves to completely define the factorial function. We can take its definition to be given by the two equations<sup>5</sup>

$$0! = 1$$

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

We can implement the factorial function directly from this definition. The first line of the definition, setting up the name of the function (`fact`), its single integer argument (`n`), and its output type (`int`) is straightforward.

```
let fact (n : int) : int =
  ...
```

The body of the function starts by distinguishing the two cases, when `n` is zero and when `n` is positive.

```
let fact (n : int) : int =
  if n = 0 then ...
  else ...
```

The zero case is simple; the output value is 1.

```
let fact (n : int) : int =
  if n = 0 then 1
  else ...
```

The non-zero case involves multiplying `n` by the factorial of `n - 1`.

<sup>5</sup> See Section B.1.1 for more background on defining mathematical functions by equations.

```
let fact (n : int) : int =
  if n = 0 then 1
  else n * fact (n - 1) ;;
```

Let's try it.

```
# let fact (n : int) : int =
#   if n = 0 then 1
#   else n * fact (n - 1) ;;
Line 3, characters 9-13:
3 | else n * fact (n - 1) ;;
      ^^^^
Error: Unbound value fact
Hint: If this is a recursive definition,
you should add the 'rec' keyword on line 1
```

There seems to be a problem. Recall from Section 5.5 that the scope of a `let` is the body of the `let` (or the code following a global `let`), but not the definition part of the `let`. Yet we've referred to the name `fact` in the definition of the `fact` function. The scope rules for the `let` constructs (both local and global) disallow this.

In order to extend the scope of the naming to the definition itself, to allow a recursive definition, we add the `rec` keyword after the `let`.

```
# let rec fact (n : int) : int =
#   if n = 0 then 1
#   else n * fact (n - 1) ;;
val fact : int -> int = <fun>
```

The `rec` keyword means that the scope of the `let` includes not only its body but also its definition part. With this change, the definition goes through, and in fact, the function works well:

```
# fact 0 ;;
- : int = 1
# fact 1 ;;
- : int = 1
# fact 4 ;;
- : int = 24
# fact 20 ;;
- : int = 2432902008176640000
```

You may in the past have been admonished against defining something in terms of itself, such as “comb: an object used to comb one's hair; to comb: to run a comb through.” You may therefore find something mysterious about recursive definitions. How can we make use of a function in its own definition? We seem to be using it before it's even fully defined. Isn't that problematic?

Of course, recursive definition *can* be problematic. For instance, consider this recursive definition of a function to add “just one more” to a recursive invocation of itself:

```
# let rec just_one_more (x : int) : int =
#   1 + just_one_more x ;;
val just_one_more : int -> int = <fun>
```

The *definition* works just fine, but any attempt to *use* it fails impressively:

```
# just_one_more 42 ;;
Stack overflow during evaluation (looping recursion?).
```

The error message “Stack overflow during evaluation (looping recursion?)” gives a hint as to what’s gone wrong; there is indeed a looping recursion that would go on forever if the computer didn’t run out of memory (“stack overflow”) first.

A recursion is WELL FOUNDED if it eventually “bottoms out” in a non-recursive computation. Clearly, the recursion in `just_one_more` is not well founded and thus not useful. But a recursion that is well founded can be quite useful.<sup>6</sup> In the case of factorial, each recursive invocation of `fact` is given an argument that is one smaller than the previous invocation, so that eventually an invocation on argument 0 will occur and the recursion will end. Because there are branches of computation (namely, the first arm of the conditional) without recursive invocations of `fact`, and those branches will eventually be taken, all is well.

But will those branches always be eventually taken? Unfortunately not.

```
# fact (~-5) ;;
Stack overflow during evaluation (looping recursion?).
```

This looks familiar. Counting down from any non-negative integer will eventually get us to zero. But counting down from a negative integer won’t. We intended the factorial function to apply only to non-negative integers, the values for which its recursion is well founded, but we didn’t express that intention – the edict of intention again – with this unfortunate result.

You might think that we could solve this problem with types. Instead of specifying the argument as having integer type, perhaps we could specify it as of non-negative integer type. Unfortunately, OCaml does not provide for this more fine-grained type, and in any case, other examples might require different constraints on the type, perhaps odd integers only, or integers larger than 7, or integers within a certain range.

### Exercise 32

For each of the following cases, define a recursive function of a single argument for which the recursion is well founded (and the computation terminates) only when the argument is

<sup>6</sup> In fact, the computer scientist C. A. R. Hoare in his 1981 Turing Award lecture described his own introduction to recursion this way:

Around Easter 1961, a course on ALGOL 60 was offered in Brighton, England, with Peter Naur, Edsger W. Dijkstra, and Peter Landin as tutors. ... It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining. It was there that I wrote the procedure, immodestly named QUICK-SORT, on which my career as a computer scientist is founded. Due credit must be paid to the genius of the designers of ALGOL 60 who included recursion in their language and enabled me to describe my invention so elegantly to the world. I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed. (Hoare, 1981)

1. an odd integer;
2. an integer less than or equal to 5;
3. the integer 0;
4. the truth value `true`.

OCaml's type system isn't expressive enough to capture these fine-grained distinctions.<sup>7</sup> Instead, we'll have to deal with such anomalous conditions using different techniques, which will be the subject of Chapter 10.

#### Exercise 33

Imagine tiling a floor with square tiles of ever-increasing size, each one abutting the previous two, as in Figure 6.4. The sides of the tiles grow according to the FIBONACCI SEQUENCE, in which each number is the sum of the previous two. By convention, the first two numbers in the sequence are 0 and 1. Thus, the third number in the sequence is  $0 + 1 = 1$ , the fourth is  $1 + 1 = 2$ , and so forth.

The first 10 numbers in the Fibonacci sequence are

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$

The Fibonacci sequence has connections to many natural phenomena, from the spiral structure of seashells (as alluded to in the figure) to the arrangement of seeds in a sunflower to the growth rate of rabbits. It even relates to the golden ratio: the tiled area depicted in the figure tends toward a golden rectangle (see Exercise 8) as more tiles are added. (Exercise 172 explores this fact.)

Define a recursive function `fib : integer -> integer` that given an index into the Fibonacci sequence returns the integer at that index. For instance,

```
# fib 1 ;;
- : int = 0
# fib 2 ;;
- : int = 1
# fib 8 ;;
- : int = 13
```

#### Exercise 34

Define a function `fewer_divisors : int -> int -> bool`, which takes two integers, `n` and `bound`, and returns `true` if `n` has fewer than `bound` divisors (including 1 and `n`). For example:

```
# fewer_divisors 17 3 ;;
- : bool = true
# fewer_divisors 4 3 ;;
- : bool = false
# fewer_divisors 4 4 ;;
- : bool = true
```

Do not worry about zero or negative arguments or divisors. Hint: You may find it useful to define an auxiliary function to simplify the definition of `fewer_divisors`.

## 6.7 Unit testing

Having written some functions, how can we have some assurance that our code is correct? Best might be a mathematical proof that the code does what it's supposed to do. Such a proof would guarantee that the code generates the appropriate values regardless of what inputs it is given. This is the domain of FORMAL VERIFICATION of software. Unfortunately, the difficulty of providing formal specifications that can be verified, along with the arduousness of carrying out the necessary

<sup>7</sup> If you are interested in the issue, you might explore the literature on DEPENDENT TYPE SYSTEMS, which provide this expanded expressivity at the cost of much more complex type inference computations.

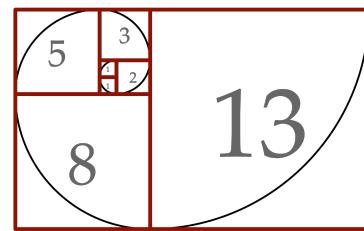


Figure 6.4: A Fibonacci tiling.

proofs, means that this approach to program correctness is used only in rare circumstances. It is, in any case, beyond the scope of this book.

But if we can't have a proof that a program generates the appropriate values on *all* input values, perhaps we can at least verify that it generates the appropriate values on *some* of them – even better if the values we verify are representative of a full range of cases. This leads us to the approach of **UNIT TESTING**, the systematic evaluation of code on known inputs, comparing the actual behavior to the intended behavior.

In this section, we begin the development of a simple unit testing framework for OCaml, continuing the development in Sections 10.5 and 17.6. We do so not because OCaml lacks a good unit testing tool of its own; in fact, there are several such full-featured packages, such as `ounit`, `alcotest`, `qcheck`, `ppl_inline_tests`, `crowbar`, `bun`, and `broken`, providing functionality far beyond what we develop in this running example. Rather, seeing the construction should make clearer what is going on in such unit testing tools, making their utility clearer. In addition, the subtle issues that arise provide a nice opportunity to demonstrate the use of abstractions (exceptions and laziness) that we introduce later. But we start here using only functions.

Consider the `fact` function defined above. It exhibits the following (correct) behavior:

```
# fact 1 ;;
- : int = 1
# fact 2 ;;
- : int = 2
# fact 5 ;;
- : int = 120
# fact 10 ;;
- : int = 3628800
```

We can describe the correctness conditions for these inputs as a series of boolean expressions.

```
# fact 1 = 1 ;;
- : bool = true
# fact 2 = 2 ;;
- : bool = true
# fact 5 = 120 ;;
- : bool = true
# fact 10 = 3628800 ;;
- : bool = true
```

A unit testing function for `fact`, call it `fact_test`, verifies that `fact` calculates the correct values for representative examples. (Let's start with these.) One approach is to simply evaluate each of the conditions and make sure that they are all `true`.

```
# let fact_test () =
#   fact 1 = 1
#   && fact 2 = 2
#   && fact 5 = 120
#   && fact 10 = 3628800 ;;
val fact_test : unit -> bool = <fun>
```

We run the tests by calling the function:

```
# fact_test () ;;
- : bool = true
```

If all of the tests pass (as they do in this case), the testing function returns `true`. If any test fails, it returns `false`. Unfortunately, in the latter case it provides no help in tracking down the tests that fail.

In order to provide information about which tests have failed, we'll print an indicative message associated with the test. An auxiliary function to handle the printing will be helpful:<sup>8</sup>

```
# let unit_test (test : bool) (msg : string) : unit =
#   if test then
#     Printf.printf "%s passed\n" msg
#   else
#     Printf.printf "%s FAILED\n" msg ;;
val unit_test : bool -> string -> unit = <fun>
```

Now the `fact_test` function can call `unit_test` to verify each of the conditions.

```
# let fact_test () =
#   unit_test (fact 1 = 1) "fact 1";
#   unit_test (fact 2 = 2) "fact 2";
#   unit_test (fact 5 = 120) "fact 5";
#   unit_test (fact 10 = 3628800) "fact 10" ;;
val fact_test : unit -> unit = <fun>
```

Running `fact_test` provides a report on the performance of `fact` on each of the unit tests.

```
# fact_test () ;;
fact 1 passed
fact 2 passed
fact 5 passed
fact 10 passed
- : unit = ()
```

We'll want to unit test `fact` as completely as is practicable. We can't test *every* possible input, but we can at least try examples representing as wide a range of cases as possible. We're missing an especially important case, the base case for the recursion, `fact 0`. We'll add a unit test for that case:

```
# let fact_test () =
#   unit_test (fact 0 = 1) "fact 0 (base case)";
```

<sup>8</sup> We're making use here of two language constructs that, strictly speaking, belong in later chapters, as they involve side effects, computational artifacts that don't affect the value expressed: the sequencing operator (`;`) discussed in Section 15.3, and the `printf` function in the [Printf library module](#). Side effects in general are introduced in Chapter 15.

```
#  unit_test (fact 1 = 1) "fact 1";
#  unit_test (fact 2 = 2) "fact 2";
#  unit_test (fact 5 = 120) "fact 5";
#  unit_test (fact 10 = 3628800) "fact 10" ;;
val fact_test : unit -> unit = <fun>
```

We haven't tested the function on negative numbers, and probably should. But `fact` as currently written wasn't intended to handle those cases. We postpone discussion about unit testing in such cases to Section 10.5, when we'll have further tools at hand. (See Exercise 81.)

Testing the `hypotenuse` function presents further issues. We might want to check the simple case of the hypotenuse of a unit triangle, whose hypotenuse ought to be about 1.41421356, as well as the limiting case of a "triangle" with zero-length sides.

```
# let hypotenuse_test () =
#   unit_test (hypotenuse 0. 0. = 0.) "hyp 0 0";
#   unit_test (hypotenuse 1. 1. = 1.41421356) "hyp 1 1" ;;
val hypotenuse_test : unit -> unit = <fun>

# hypotenuse_test () ;;
hyp 0 0 passed
hyp 1 1 FAILED
- : unit = ()
```

The test reveals a problem. The unit triangle test has failed, not because the `hypotenuse` function is wrong but because the value we've proposed isn't exactly the floating point number calculated. The `float` type has a fixed capacity for representing numbers, and can't therefore represent all numbers exactly. The best we can do is check that floating point calculations are approximately correct, within some tolerance. Rather than checking the condition as above, instead we can check that the value is within, say, 0.0001 of the value in the test, a condition like this:

```
# hypotenuse 1. 1. -. 1.41421356 < 0.0001 ;;
- : bool = true
```

Instead of writing out these more complex conditions each time they're needed, we'll devise another unit testing function for approximate floating point calculations:

```
# let unit_test_within (tolerance : float)
#                 (test_value : float)
#                 (expected : float)
#                 (msg : string)
#                 : unit =
#   unit_test (abs_float (test_value -. expected) < tolerance) msg ;;
val unit_test_within : float -> float -> float -> string -> unit =
<fun>
```

We can restate the `hypotenuse_test` function to make use of these approximate tests. (We've added a few more for other conditions.)

```
# let hypotenuse_test () =
#   unit_test_within 0.0001 (hypotenuse 0. 0.) 0.          "hyp 0 0";
#   unit_test_within 0.0001 (hypotenuse 1. 1.) 1.4142      "hyp 1 1";
#   unit_test_within 0.0001 (hypotenuse ~-.1. 1.) 1.4142 "hyp -1 1";
#   unit_test_within 0.0001 (hypotenuse 2. 2.) 2.8284      "hyp 2 2" ;;
val hypotenuse_test : unit -> unit = <fun>
```

Calling the function demonstrates that all of the calculations hold within the required tolerance.

```
# hypotenuse_test () ;;
hyp 0 0 passed
hyp 1 1 passed
hyp -1 1 passed
hyp 2 2 passed
- : unit = ()
```

We'll return to the question of unit testing in Sections 10.5 and 17.6, when we have more advanced tools to use.

### 6.8 Supplementary material

- Lab 1: Basic functional programming
- Problem set A.1: The prisoners' dilemma

# 7

## *Structured data and composite types*

The kinds of data that we've introduced so far have been unstructured. The values are separate atoms,<sup>1</sup> discrete undecomposable units. Each integer is separate and atomic, each floating point number, each truth value. But the power of data comes from the ability to build new data from old by putting together data *structures*.

In this chapter, we'll introduce three quite general ways built into OCaml to structure data: tuples, lists, and records. For each such way, we describe how to *construct* structures from their parts using *value constructors*; what the associated *type* of the structures is and how to construct a type expression for them using *type constructors*; and how to *decompose* the structures into their component parts using *pattern-matching*. (We turn to methods for generating your own composite data structures in Chapter 11.) We start with tuples.

### 7.1 Tuples

The first structured data type is the **TUPLE**, a fixed length sequence of elements. The smallest tuples are pairs, containing two elements, then triples, quadruples, quintuples, sextuples, septuples, and so forth. (The etymology of the term “tuple” derives from this semi-productive suffix.)

In OCaml, a tuple value is formed using the **VALUE CONSTRUCTOR** for tuples, an infix comma. A pair containing the integer 3 and the truth value `true`, for instance, is given by `3, true`. The order is crucial; the pair `true, 3` is a different pair entirely. (Indeed, as we will see, these two pairs are not even of the same type.)

The type of a pair is determined by the types of its parts. We name the type by forming a type expression giving the types of the parts combined using the infix **TYPE CONSTRUCTOR \***, read “cross” (for “**cross product**”). For instance, the pair `3, true` is of type `int * bool` (read, “int cross bool”).

<sup>1</sup> The term “atom” is used here in its sense from Democritus and other classical philosophers, the indivisible units making up the physical universe. Now, of course, we know that though chemical elements are made of atoms, those atoms themselves have substructure and are not indivisible. Unlike the physical world, the world of discrete data can be well thought of as being built from indivisible atoms.

**Exercise 35**

What are the types for the following pair expressions?

1. `false, 5`
2. `false, true`
3. `3, 5`
4. `3.0, 5`
5. `5.0, 3`
6. `5, 3`
7. `succ, pred`

Triples are formed similarly. A triple of the elements 1, 2, and "three" would be `1, 2, "three"`; its type is `int * int * string`. This triple should not be confused with the pair consisting of the integer 1 and the `int * string` pair 2, "three". Such a pair containing a pair is also constructable, as `1, (2, "three")`, and is of type `int * (int * string)`. The parentheses in both the value expression and the type expression make clear that this datum is structured as a pair, not a triple.

**Exercise 36**

Construct a value for each of the following types.

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

**Exercise 37**

Integer division leaves a remainder. It is sometimes useful to calculate both the result of the quotient and the remainder. Define a function `div_mod : int -> int -> (int * int)` that takes two integers and returns a pair of their quotient and the remainder. For instance,

```
# div_mod 40 20 ;;
- : int * int = (2, 0)
# div_mod 40 13 ;;
- : int * int = (3, 1)
# div_mod 0 12 ;;
- : int * int = (0, 0)
```

Using this technique of returning a pair, we can get the effect of a function that returns multiple values.

**Exercise 38**

In Exercise 31, you are asked to implement the *computus* to calculate the month and day of Easter for a given year by defining two functions, one for the day and one for the year. A more natural approach is to define a single function that returns both the month and the day. Use the technique from Exercise 37 to implement a single function for *computus*.

## 7.2 Pattern matching for decomposing data structures

The value constructor is used to construct composite values from parts. How can we do the inverse, extracting the parts from the composite structure? Perhaps surprisingly, we make use of the value constructor for this purpose as well, by matching a template pattern containing the constructor against the structure being decomposed. The `match` construction is used to perform this matching and decomposition. The general form of a `match` is<sup>2</sup>

```
<expr> ::= match <exprvalue> with
  | <pattern1> -> <expr1>
  | <pattern2> -> <expr2>
  ...
  ...
```

Without going into a formal BNF definition of `<pattern>` phrases, they are essentially expressions constructed only of variables, and value constructors (including literals like `true` or `[]`). The structured value given by `<exprvalue>` is pattern-matched against each of the patterns `<pattern1>`, `<pattern2>`, and so on, in that order. Whichever pattern matches first, the variables therein name the corresponding parts of the `<exprvalue>` being matched against. The corresponding `<expri>`, which may use the variables just bound by the `<patterni>`, is evaluated to provide the value of the `match` construction as a whole. The variables in a `<patterni>` are newly introduced names, just like those in `let` and `fun` expressions, and like those variables, they also have a scope, namely, the corresponding `<expri>`.

For example, suppose we want to add the integers in an integer pair. We need to extract the integers in order to operate on them. Here is a function that extracts the two parts of the pair and returns their sum.

```
# let addpair (pair : int * int) : int =
#   match pair with
#   | x, y -> x + y ;;
val addpair : int * int -> int = <fun>
# addpair (3, 4) ;;
- : int = 7
```

In the pattern `x, y`, the variables `x` and `y` are names that can be used for the two components of the pair, as they have been in the expression `x + y`. There is nothing special about the names `x` and `y`; any variables could be used.

The `match` used here is especially simple in having just a single pattern/result pair. Only one is needed because there is only one value constructor for pairs. We'll shortly see examples where more than one pattern is used.

<sup>2</sup> The `...` in this BNF rule is intended to indicate that there may be any number of such pattern-expression pairs in the construct. We'll leave this addition to the BNF notation as informal, though precise formulations of the idea can be constructed.

On a stylistic point, the first vertical bar in `match` constructs is, strictly speaking, optional. We uniformly use it for consistency of demarcating the patterns appearing on consecutive lines, as discussed in Section C.1.7.



Figure 7.1: Computer scientist Marianne Baudinet's (1985) work with David MacQueen on compiling ML-style pattern matching constructs to efficient matching code proved to be the breakthrough that made the extensive use of pattern matching in ML-style languages practical.

Notice how the `match` construction allows us to deconstruct a structured datum into its component parts simply by matching against a template that uses the *very same* value constructor that is used to build such data in the first place. This method for decomposition is extremely general. It allows extracting the component parts from arbitrarily structured data.

You might think, for instance, that it would be useful to have a function that directly extracts the first or second element of a pair. But these can be written in terms of the `match` construct.<sup>3</sup>

```
# let fst (pair : int * int) : int =
#   match pair with
#   | x, y -> x ;;
Line 3, characters 5-6:
3 | | x, y -> x ;;
^
Warning 27 [unused-var-strict]: unused variable y.
val fst : int * int -> int = <fun>
# fst (3, 4) ;;
- : int = 3
```

The warning message arises because the variable `y` appears in the pattern, but is never used in the corresponding action. Often this is a sign that something is awry in one's code: Why would you establish a variable only to ignore its value? For that reason, this warning message can be quite useful in catching subtle bugs. But in cases like this, where the value of the variable really is irrelevant, the warning is misleading. To eliminate it, an ANONYMOUS VARIABLE – a variable starting with the underscore character – can be used instead. This codifies the programmer's intention that the variable not be used, and disables the warning message. This is a good example of the edict of intention: by clearly and uniformly expressing our intention not to use a variable, the language interpreter can help find latent bugs where we intended to use a variable but did not (as when a variable name is misspelled).

```
# let fst (pair : int * int) : int =
#   match pair with
#   | x, _y -> x ;;
val fst : int * int -> int = <fun>
# fst (3, 4) ;;
- : int = 3
```

### Exercise 39

Define an analogous function `snd : int * int -> int` that extracts the second element of an integer pair. For instance,

```
# snd (3, 4) ;;
- : int = 4
```

As another example, consider the problem of calculating [the distance between two points](#), where the points are given as pairs of

<sup>3</sup> The functions `fst` and `snd` are available as part of the `Stdlib` module, but it's useful to see how they can be written in terms of the core of the OCaml language.

floats. First, we need to extract the coordinates in each dimension by pattern matching:

```
let distance p1 p2 =
  match p1 with
  | x1, y1 ->
    match p2 with
    | x2, y2 -> ...calculate the distance... ;;
```

Rather than use two separate pattern matches, one for each argument, we can perform both matches at once using a pattern that matches against the pair of points  $p1$ ,  $p2$ .

```
let distance p1 p2 =
  match p1, p2 with
  | (x1, y1), (x2, y2) -> ...calculate the distance... ;;
```

Once the separate components of the points are in hand, the distance can be calculated:

```
# let distance p1 p2 =
#   match p1, p2 with
#   | (x1, y1), (x2, y2) ->
#     sqrt ((x2 -. x1) ** 2. +. (y2 -. y1) ** 2.) ;;
val distance : float * float -> float * float -> float = <fun>
```

The ability to pattern match to extract and name data components is so useful that OCaml provides syntactic sugar to integrate it into other binding constructs, such as the `let` and `fun` constructs. In cases where there is only a single pattern to be matched (as in the examples above), the matching can be performed directly in the `let`. That is, an expression of the form

```
let <var> = <expr> in
  match <var> with
  | <pattern1> -> <expr1>
```

can be “sugared” to<sup>4</sup>

```
let <pattern1> = <expr> in
  <expr1>
```

Using this sugared form further simplifies the `distance` function.

```
let distance p1 p2 =
  let (x1, y1), (x2, y2) = p1, p2 in
    sqrt ((x2 -. x1) ** 2. +. (y2 -. y1) ** 2.) ;;
```

Finally, pattern matching can even be used in global `let` constructs, to further simplify.

```
# let distance (x1, y1) (x2, y2) =
#   sqrt ((x2 -. x1) ** 2. +. (y2 -. y1) ** 2.) ;;
val distance : float * float -> float * float -> float = <fun>

# distance (1., 1.) (2., 2.) ;;
- : float = 1.41421356237309515
```

<sup>4</sup> Anonymous functions can benefit from this syntactic sugar as well, for instance, as in

```
# (fun (x, y) -> x + y) (3, 4) ;;
- : int = 7
```

As usual, it is useful to add typings in the global definition to make clear the intended types of the arguments and the result:<sup>5</sup>

```
# let distance (x1, y1 : float * float)
#           (x2, y2 : float * float)
#           : float =
#   sqrt ((x2 -. x1) ** 2. +. (y2 -. y1) ** 2.) ;;
val distance : float * float -> float * float -> float = <fun>

# distance (1., 1.) (2., 2.) ;;
- : float = 1.41421356237309515
```

#### Exercise 40

Simplify the definitions of `addpair` and `fst` above by taking advantage of this syntactic sugar.

Using this shorthand can make code much more readable, and is thus recommended. See the style guide (Section C.4.2) for further discussion.

#### Exercise 41

Define a function `slope : float * float -> float * float -> float` that returns the `slope` between two points.

### 7.2.1 Advanced pattern matching

It's not only composite types that can be the object of pattern matching. Patterns can match particular values of atomic types as well, such as `int` or `bool`. One could, for instance, write

```
# let int_of_bool (cond : bool) : int =
#   match cond with
#   | true -> 1
#   | false -> 0 ;;
val int_of_bool : bool -> int = <fun>

# int_of_bool true ;;
- : int = 1
# int_of_bool false ;;
- : int = 0
```

For booleans, however, the use of a conditional is considered a better approach:<sup>6</sup>

```
# let int_of_bool (cond : bool) : int =
#   if cond then 1 else 0 ;;
val int_of_bool : bool -> int = <fun>
```

Integers can also be matched against:

```
# let is_small_int (x : int) : bool =
#   match abs x with
#   | 0 -> true
#   | 1 -> true
```

<sup>5</sup>This example provides a good opportunity to mention that for readability code lines should be kept short. We use a convention described in the style guide (Section C.3.4) for breaking up long function definition introductions.

<sup>6</sup>Using `cond = true` as the test part of the conditional is redundant and stylistically poor. See Section C.5.2.

```

#   | 2 -> true
#   | _ -> false ;;
val is_small_int : int -> bool = <fun>

# is_small_int ~1 ;;
- : bool = true
# is_small_int 2 ;;
- : bool = true
# is_small_int 7 ;;
- : bool = false

```

Notice here the use of an anonymous variable `_` as a WILD-CARD pattern that matches any value.

In the `is_small_int` function, the same result is appropriate for multiple patterns. Rather than repeat the result expression in each case, multiple patterns can be associated with a single result, by listing the patterns interspersed with vertical bars (`|`).

```

# let is_small_int (x : int) : bool =
#   match abs x with
#   | 0 | 1 | 2 -> true
#   | _ -> false ;;
val is_small_int : int -> bool = <fun>

```

### 7.3 Lists

Tuples are used for packaging together *fixed-length* sequences of elements of perhaps *differing* type. LISTS, conversely, are used for packaging together *varied-length* sequences of elements all of the *same* type. The type constructor `list` for lists thus operates on a single type, the type of the list elements, and is written in POSTFIX position – that is, following its argument. For instance, the type corresponding to a list of integers is given by the type expression `int list`, a list of booleans as `bool list`, a list of coordinates (pairs of floats, say) as `(float * float) list`.

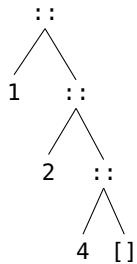
There are two value constructors for lists. The first value constructor, written `[]` and conventionally read as “NIL”, specifies the empty list, that is, the list containing no elements at all. The second value constructor, written with an infix `::` and conventionally read as “CONS”,<sup>7</sup> takes two arguments – a first element and a further list of elements – and specifies the list whose first element is its first argument and whose remaining elements are the second. (The two parts of a non-empty list, the first element and the remaining elements, are called the HEAD and the TAIL of the list, respectively.)

Suppose we want a list of integers containing just the integer 4. Such a list can be constructed by starting with the empty list `[]`, and “consing” 4 to it as `4 :: []`. The list containing, in sequence, 2 and 4

<sup>7</sup> The term “cons” for this constructor derives from the `cons` function in one of the earliest and most influential functional programming languages, Lisp. It reflects the idea of *constructing* a list by adding a new element.

is constructed by consing 2 onto the list containing 4, that is, `2 :: (4 :: [])`. The list of the integers 1, 2, and 4 is analogously `1 :: (2 :: (4 :: []))`.

As usual, some notational cleanup is in order. First, we can take advantage of the fact that the `::` operator is right associative, so that the parentheses in the lists above are not needed. We can simply write `1 :: 2 :: 4 :: []`. Second, OCaml provides a more familiar alternative notation – more sugar – for lists, writing the elements of the list in order within brackets and separated by semicolons, as `[1; 2; 4]`. We can think of all of these as alternative concrete syntaxes for the same underlying abstract syntax, given by



You can verify the equivalence of these notations by entering them into OCaml:

```

# 1 :: (2 :: (4 :: [])) ;;
- : int list = [1; 2; 4]
# 1 :: 2 :: 4 :: [] ;;
- : int list = [1; 2; 4]
# [1; 2; 4] ;;
- : int list = [1; 2; 4]
  
```

Notice that in all three cases, OCaml provides the inferred type `int list` and reports the value using the sugared list notation.<sup>8</sup>

#### Exercise 42

Which of the following expressions are well-formed, and for those that are, what are their types and how would their values be written using the sugared notation?

1. `3 :: []`
2. `true :: false`
3. `true :: [false]`
4. `[true] :: [false]`
5. `[1; 2; 3.1416]`
6. `[4; 2; -1; 1_000_000]`
7. `([true], false)`

Using the `::` and bracketing notations, we can construct lists from their elements. How can we extract those elements from lists? As always in OCaml, decomposing structured data is done with pattern-matching; no new constructs are needed. We'll see examples shortly.

<sup>8</sup> The list containing elements, say, 1 and 2 – written `[1; 2]` – should not be confused with the pair of those same elements – written `(1, 2)`. The concrete syntactic differences may be subtle (semicolon versus comma; brackets versus parentheses) but their respective types make the distinction quite clear.

### 7.3.1 Some useful list functions

To provide some intuition with list processing, we'll construct a few useful functions, starting with a function to determine if an integer list is empty or not. We start with considering the type of the function. Its argument should be an integer list (of type `int list`) and its result a truth value (of type `bool`), so the type of the function itself is `int list -> bool`. This type information is just what we need in order to write the first line of the function definition, naming the function's argument and incorporating the typing information:

```
let is_empty (lst : int list) : bool = ...
```

Now we need to determine whether `lst` is empty or not, that is, what value constructor was used to construct it. We can do so by pattern matching `lst` against a series of patterns. Since lists have only two value constructors, two patterns will be sufficient.

```
let is_empty (lst : int list) : bool =
  match lst with
  | [] -> ...
  | head :: tail -> ...
```

What should we do in these two cases? In the first case, we can conclude that `lst` is empty, and hence, the value of the function should be `true`. In the second case, `lst` must have at least one element (now named `head` by the pattern match), and is thus non-empty; the value of the function should be `false`.<sup>9</sup>

```
# let is_empty (lst : int list) : bool =
#   match lst with
#   | []           -> true
#   | head :: tail -> false ;;
Line 4, characters 2-6:
4 | | head :: tail -> false ;;
      ^^^^

Warning 27 [unused-var-strict]: unused variable head.
Line 4, characters 10-14:
4 | | head :: tail -> false ;;
      ^^^^^

Warning 27 [unused-var-strict]: unused variable tail.
val is_empty : int list -> bool = <fun>
```

Since neither `head` nor `tail` are used in the second pattern match, they should be made anonymous variables to codify that intention (and avoid a warning message).<sup>10</sup>

```
# let is_empty (lst : int list) : bool =
#   match lst with
#   | []           -> true
#   | _ :: _ -> false ;;
val is_empty : int list -> bool = <fun>
```

<sup>9</sup> We've used alignment of the arrows in the pattern match to emphasize the parallelism between these two cases. See the discussion in the style guide (Section C.1.7) for differing views on this practice.

<sup>10</sup> The “wild card” anonymous variable `_` is special in not serving as a name that can be later referred to, and is thus allowed to be used more than once in a pattern.

```
# is_empty [] ;;
- : bool = true
# is_empty [1; 2; 3] ;;
- : bool = false
# is_empty (4 :: []) ;;
- : bool = false
```

Sure enough, the function works well on the test cases.

Let's try another example: calculating the LENGTH of a list, the count of its elements. We use the same approach, starting with the type of the function. The argument is an `int list` as before, but the result type is an `int` providing the count of the elements; overall, the function is of type `int list -> int`. The type of the function in hand, the first line writes itself.

```
let length (lst : int list) : int = ...
```

And again, a pattern match on the sole argument is a natural first step to decide how to proceed in the calculation.

```
let length (lst : int list) : int =
  match lst with
  | [] -> ...
  | hd :: tl -> ...
```

In the first match case, the list is empty; hence its length is 0.

```
let length (lst : int list) : int =
  match lst with
  | [] -> 0
  | hd :: tl -> ...
```

The second case is more subtle, however. The length must be at least 1 (since the list at least has the single element `hd`). But the length of the list overall depends on `tl`, and in particular, the length of `tl`. If only we had a method for calculating the length of `tl`.

But we do; the `length` function itself can be used for this purpose! Indeed, the whole point of `length` is to calculate lengths of `int list`s like `tl`. We can call `length` recursively on `tl`, and add 1 to the result to calculate the length of the full list `lst`.<sup>11</sup>

```
# let rec length (lst : int list) : int =
#   match lst with
#   | [] -> 0
#   | _hd :: tl -> 1 + length tl ;;
val length : int list -> int = <fun>
```

(We've made `_hd` an anonymous variable for the same reasons as above, and also inserted the `rec` keyword to allow the recursive reference to `length` within the definition.)

We can test the function on a few examples to demonstrate it.

<sup>11</sup> As with the definition of the recursive factorial function in Section 6.6, the well-founded basis of this recursive definition depends on the recursive calls heading in the direction of the base case. In this case, the recursive application of the function is to a *smaller* data structure, the tail of the original argument, and all further applications will similarly be to smaller and smaller data structures. This process can't continue indefinitely. Inevitably, it will bottom out in application to the empty list, at which point the computation is non-recursive and terminates. Recursive computation may seem a bit magical when you first confront it, but over time it becomes a powerful tool natural to deploy.

```
# length [1; 2; 4] ;;
- : int = 3
# length [] ;;
- : int = 0
# length [[1; 2; 4]] ;;
Line 1, characters 8-17:
1 | length [[1; 2; 4]] ;;
^~~~~~
Error: This expression has type 'a list
      but an expression was expected of type int
```

**Exercise 43**

Why does this last example cause an error, given that its input is a list of length one?  
Chapter 9 addresses this problem more thoroughly.

As a final example, we'll implement a function that, given a list of pairs of integers, returns the list of products of the pairs. For example, the following behaviors should hold.

```
# prods [2,3; 4,5; 0,10] ;;
- : int list = [6; 20; 0]
# prods [] ;;
- : int list = []
```

By now the process should be familiar. Start with the type of the function: `(int * int) list -> int list`. Use the type to write the function introduction:

```
let rec prods (lst : (int * int) list) : int list = ...
```

Use pattern-matching to decompose the argument:

```
let rec prods (lst : (int * int) list) : int list =
  match lst with
  | [] -> ...
  | hd :: tl -> ...
```

In the first pattern match, the list is empty; we should thus return the empty list of products.

```
let rec prods (lst : (int * int) list) : int list =
  match lst with
  | [] -> []
  | hd :: tl -> ...
```

Finally, we get to the tricky bit. If the list is nonempty, the head will be a pair of integers, which we'll want access to. We could pattern match against `hd` to extract the parts:

```
let rec prods (lst : (int * int) list) : int list =
  match lst with
  | [] -> []
  | hd :: tl ->
    match hd with
    | (x, y) -> ...
```

but it's simpler to fold that pattern match into the list pattern match itself:

```
let rec prods (lst : (int * int) list) : int list =
  match lst with
  | [] -> []
  | (x, y) :: tl -> ...
```

Now, the result in the second pattern match should be a list of integers, the first of which is  $x * y$  and the remaining elements of which are the products of the pairs in  $tl$ . The latter can be computed recursively as  $prods\ tl$ . (It's a good thing we thought ahead to use the `rec` keyword.) Finally, the list whose first element is  $x * y$  and whose remaining elements are  $prods\ tl$  can be *constructed* as  $x * y :: prods\ tl$ .

```
# let rec prods (lst : (int * int) list) : int list =
#   match lst with
#   | [] -> []
#   | (x, y) :: tl -> x * y :: prods tl ;;
val prods : (int * int) list -> int list = <fun>
# prods [2,3; 4,5; 0,10] ;;
- : int list = [6; 20; 0]
# prods [] ;;
- : int list = []
```

You'll have noticed a common pattern to writing these functions, one that is widely applicable.

1. Write down some examples of the function's use.
2. Write down the type of the function.
3. Write down the first line of the function definition, based on the type of the function, which provides the argument and result types.
4. Using information about the argument types, decompose one or more of the arguments.
5. Solve each of the subcases, paying attention to the types, and using recursion where appropriate, to construct the output value.
6. Test the examples from Step 1.

Using this STRUCTURE-DRIVEN PROGRAMMING pattern can make it so that simple functions of this sort almost write themselves. Notice the importance of types in the process. The types constrain so many aspects of the function that they provide a guide to writing the function itself.

#### **Exercise 44**

Define a function `sum : int list -> int` that computes the sum of the integers in its list argument.

```
# sum [1; 2; 4; 8] ;;
- : int = 15
```

What should this function return when applied to the empty list?

#### Exercise 45

Define a function `prod : int list -> int` that computes the product of the integers in its list argument.

```
# prod [1; 2; 4; 8] ;;
- : int = 64
```

What should this function return when applied to the empty list?

#### Exercise 46

Define a function `sums : (int * int) list -> int list`, analogous to `prods` above, which computes the list each of whose elements is the sum of the elements of the corresponding pair of integers in the argument list. For example,

```
# sums [2,3; 4,5; 0,10] ;;
- : int list = [5; 9; 10]
# sums [] ;;
- : int list = []
```

#### Exercise 47

Define a function `inc_all : int list -> int list`, which increments all of the elements in a list.

```
# inc_all [1; 2; 4; 8] ;;
- : int list = [2; 3; 5; 9]
```

#### Exercise 48

Define a function `square_all : int list -> int list`, which squares all of the elements in a list.

```
# square_all [1; 2; 4; 8] ;;
- : int list = [1; 4; 16; 64]
```

#### Exercise 49

Define a function `append : int list -> int list -> int list` to append two integer lists. Some examples:

```
# append [1; 2; 3] [4; 5; 6] ;;
- : int list = [1; 2; 3; 4; 5; 6]
# append [] [4; 5; 6] ;;
- : int list = [4; 5; 6]
# append [1; 2; 3] [] ;;
- : int list = [1; 2; 3]
```

#### Exercise 50

Define a function `concat : string -> string list -> string`, which takes a string `sep` and a string list `lst`, and returns one string with all the elements of `lst` concatenated together but separated by the string `sep`.<sup>12</sup> Some examples:

```
# concat ", " ["first"; "second"; "third"] ;;
- : string = "first, second, third"
# concat "... ["Moo"; "Baa"; "Lalala"] ;;
- : string = "Moo...Baa...Lalala"
# concat ", " [] ;;
- : string = ""
# concat ", " ["Moo"] ;;
- : string = "Moo"
```

We've gone through the valuable exercise of writing a bunch of useful list functions. But list processing is so ubiquitous that OCaml provides a library module for just such functions. We'll discuss the `List` module further in Section 9.4.

<sup>12</sup> The OCaml library module `String` already provides just this function under the same name.

## 7.4 Records

Tuples and lists use the *order* within a sequence to individuate their elements. An alternative, the RECORD, names the elements, providing each with a unique *label*. The type constructor specifies the labels and the type of element associated with each. For instance, suppose we'd like to store information about people: first and last name and year of birth. An appropriate record type would be

```
{lastname : string; firstname : string; birthyear : int}
```

Each of the elements in a record is referred to as a FIELD. Since the fields are individuated by their label, the order in which they occur is immaterial; the same type could have been specified reordering the fields as

```
{firstname : string; birthyear : int; lastname : string}
```

with no difference (except perhaps to add a bit of confusion to a reader expecting a more systematic ordering).

Unlike lists and tuples, which are built-in types in OCaml, particular record types are user-defined. OCaml needs to know about the type – its fields, their labels and types – in order to make use of them. Records are the first of the user-defined types we'll explore in detail in Chapter 11. To define a new type, we use the type construction to give the type a name:<sup>13</sup>

$$\langle \text{definition} \rangle ::= \text{type } \langle \text{typename} \rangle = \langle \text{typeexpr} \rangle$$

We might give the type above the name person:

```
# type person = {lastname : string;
#                 firstname : string;
#                 birthyear : int} ;;
type person = { lastname : string; firstname : string; birthyear :
int; }
```

Now that the type is defined and OCaml is aware of its fields' labels and types, we can start constructing values of that type. To construct a record value, we use the strikingly similar notation of placing the fields, separated by semicolons, within braces. In record value expressions, the label of a field is separated from its value by an =.<sup>14</sup> We define a value of the record type above:

```
# let ac =
#   {firstname = "Alonzo";
#   lastname = "Church";
#   birthyear = 1903} ;;
val ac : person =
{lastname = "Church"; firstname = "Alonzo"; birthyear = 1903}
```

<sup>13</sup> The *<definition>* phrase type was introduced in footnote 4.

<sup>14</sup> A common confusion when first using record types concerns when to use : and when to use = within fields. Here's a way to think about the usages: The use of : in record *type* expressions evokes the use of : in a typing. In a sense, the type constructor provides a typing for each of the fields. The use of = in record *value* expressions evokes the use of = in naming constructs.

Notice that the type inferred for `ac` is `person`, the defined name for the record type.

As usual, we use pattern matching to decompose a record into its constituent parts. A simple example decomposes the `ac` value just created to extract the birth year.

```
# match ac with
# | {lastname = _lname;
#     firstname = _fname;
#     birthyear = byear} -> byear ;;
- : int = 1903
```

We can define a function that takes a value of type `person` and returns the person's full name as a single string by extracting and concatenating the first and last names.

```
# let fullname (p : person) : string =
#   match p with
#   | {firstname = fname;
#       lastname = lname;
#       birthyear = _byear} ->
#       fname ^ " " ^ lname ;;
val fullname : person -> string = <fun>
```

This function can be used to generate the full name:

```
# fullname ac ;;
- : string = "Alonzo Church"
```

It's a bit cumbersome to have to mention every field in a record pattern match when we are interested in only a subset of the fields. Fortunately, patterns need only specify a subset of the fields, using the notation `_` to stand for any remaining fields.<sup>15</sup>

```
let fullname (p : person) : string =
  match p with
  | {firstname = fname; lastname = lname; _} ->
    fname ^ " " ^ lname ;;
```

Another simplification in record patterns, called FIELD PUNNING, is allowed for fields in which the label and the variable name are identical. In that case, the label alone is all that is required. We can use field punning to simplify `fullname`:

```
let fullname (p : person) : string =
  match p with
  | {firstname; lastname; _} ->
    firstname ^ " " ^ lastname ;;
```

As a final simplification, the syntactic sugar allowing single-pattern matches in `let` constructs allows us to eliminate the explicit `match` entirely:

<sup>15</sup> In fact, the `_` notation isn't necessary, but it performs the useful role of capturing the programmer's intention that the set of fields is not complete. In fact, OCaml will provide a warning (when properly set up) if an incomplete record match isn't marked with this notation.

```
# let fullname ({firstname; lastname; _} : person) : string =
#   firstname ^ " " ^ lastname ;;
val fullname : person -> string = <fun>

# fullname ac ;;
- : string = "Alonzo Church"
```

#### 7.4.1 Field selection

Pattern matching permits extracting the values of all of the fields of a record (or any subset). When only one field value is needed, however, a more succinct technique suffices. The familiar dot notation from many programming languages allows selection of a single field.

```
# ac.firstname ;;
- : string = "Alonzo"
# ac.birthyear ;;
- : int = 1903
```

Thus, the `fullname` function could have been written as

```
# let fullname (p : person) : string =
#   p.firstname ^ " " ^ p.lastname ;;
val fullname : person -> string = <fun>
# fullname ac ;;
- : string = "Alonzo Church"
```

Which notation to use is again a design matter, which will depend on the individual case.

### 7.5 Comparative summary

These three data structuring mechanisms provide three different approaches to the same idea – agglomerating a collection of elements into a single unit. The differences arise in how the elements are individuated. In tuples and lists, an element is individuated by its *index* in an ordered collection. In records, an element is individuated by its *label* in a labeled but unordered collection.

Tuples and records collect a fixed number of elements. Because the number of elements is fixed, they can be of differing type. The type of the tuple or record indicates what type each element has. Lists, on the other hand, collect an arbitrary number of elements. In order to be able to operate on any arbitrary element, the types of all the elements must be indicated in the type of the list itself. This constraint is facilitated by having all elements have the same type, so that they can be operated on uniformly.

Table 7.1 provides a summary of the differing structuring mechanisms.

	<i>Tuples</i>	<i>Records</i>	<i>Lists</i>
element types selected by	differing order	differing label	uniform order
type constructors	$\langle \rangle * \langle \rangle$	$\{a : \langle \rangle ; b : \langle \rangle ; c : \langle \rangle ; \dots\}$	$\langle \rangle \text{ list}$
value constructors	$\langle \rangle , \langle \rangle$	$\{a = \langle \rangle ; b = \langle \rangle ; c = \langle \rangle ; \dots\}$	$[] \quad \langle \rangle :: \langle \rangle$

Table 7.1: Comparison of three structuring mechanisms: tuples, records, and lists.



# 8

## *Higher-order functions and functional programming*

We've laid the groundwork for programming with functions in Chapter 6, and provided some useful structures for data in Chapter 7, especially lists. In this chapter we show how higher-order functions serve as a mechanism to satisfy the edict of irredundancy. By examining some cases of similar code, we will present the use of higher-order functions to achieve the abstraction, in so doing presenting some of the most well known abstractions of higher-order functional programming on lists – map, fold, and filter.

### *8.1 The map abstraction*

In Exercises 47 and 48, you wrote functions to increment and to square all of the elements of a list. After solving the first of these exercises with

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

you may have thought to cut and paste the solution, modifying it slightly to solve the second:

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

These “apparently dissimilar” pieces of code bear a striking resemblance, a result of the cutting and pasting. And to the extent that they echo the same idea, we've written the same code twice, violating the edict of irredundancy. Can we view them abstractly as “instantiating an underlying identity”?

The differences between these functions are localized in their last lines, where they compute the head of the output list from the head

of the input list – in `inc_all` as `1 + hd`, in `square_all` as `hd * hd`. But the redundancies here are not merely the use of different values (as they were in Section 6.5), but different *computations* over values. Do we have a tool to characterize these different computations, what is done to the head of the input list in each case? Yes, the function! In `inc_all`, we are essentially applying the function `fun x -> 1 + x` to the head, and in `square_all`, the function `fun x -> x * x`. We can make this clearer by rewriting the two snippets of code as explicit applications of a function.

```
let rec inc_all (xs : int list) : int list =
  match xs with
  | [] -> []
  | hd :: tl -> (fun x -> 1 + x) hd :: (inc_all tl) ;;

let rec square_all (xs : int list) : int list =
  match xs with
  | [] -> []
  | hd :: tl -> (fun x -> x * x) hd :: (square_all tl) ;;
```

Now, we can take advantage of the fact that in OCaml functions are first-class values, which can be used as arguments or outputs of functions, to construct a single function that performs this general task of applying a function, call it `f`, to each element of a list. We add `f` as a new argument and replace the different functions being applied to `hd` with this `f`. Historically, this abstract pattern of computation – performing an operation on all elements of a list – is called a **MAP**. We capture it in a function named `map` that abstracts both `inc_all` and `square_all`.

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

The `map` function takes two arguments (curried), the first of which is itself a function, to be applied to all elements of its second integer list argument. Its type is thus `(int -> int) -> int list -> int list`. With `map` in hand, we can perform the equivalent of `inc_all` and `square_all` directly.

```
# map (fun x -> 1 + x) [1; 2; 4; 8] ;;
- : int list = [2; 3; 5; 9]
# map (fun x -> x * x) [1; 2; 4; 8] ;;
- : int list = [1; 4; 16; 64]
```

In fact, `map` can even be used to define the functions `inc_all` and `square_all`.

```
# let inc_all (xs : int list) : int list =
#   map (fun x -> 1 + x) xs ;;
val inc_all : int list -> int list = <fun>
# let square_all (xs : int list) : int list =
#   map (fun x -> x * x) xs ;;
val square_all : int list -> int list = <fun>
```

These definitions of `inc_all` and `square_all` don't suffer from the violation of the edict of irredundancy exhibited by our earlier ones. By abstracting out the differences in those functions and capturing them in a single higher-order function `map`, we've simplified each of the definitions considerably.

But making full use of higher-order functions as an abstraction mechanism allows even further simplification, via partial application.

## 8.2 Partial application

Although we traditionally think of functions as being able to take more than one argument, in OCaml functions always take exactly one argument. Here, for instance, is the `power` function, which appears to take two arguments, an exponent  $n$  and a base  $x$ , and returns  $x^n$ .<sup>1</sup>

```
# let rec power (n, x) =
#   if n = 0 then 1
#   else x * power ((n - 1), x) ;;
val power : int * int -> int = <fun>
# power (3, 4) ;;
- : int = 64
```

Though it appears to be a function of two arguments, “desugaring” makes clear that there is really only one argument. First, we desugar the `let`:

```
let rec power =
  fun (n, x) ->
    if n = 0 then 1
    else x * power ((n - 1), x) ;;
```

and then desugar the pattern match in the `fun`:

```
let rec power =
  fun arg ->
    match arg with
    | (n, x) -> if n = 0 then 1
                  else x * power ((n - 1), x) ;;
```

demonstrating that all along, `power` was a function (defined with `fun`) of one argument (now called `arg`).

How about this definition of `power`?

```
# let rec power n x =
#   if n = 0 then 1
```

<sup>1</sup> In general, it's good practice to provide typing information in the header of a function. In this section and the rest of the chapter, however, we leave off types in the headers so as to emphasize the structural relationships among the various versions of the functions we discuss. The typings would be distracting from the point being made.

```
#  else x * power (n - 1) x ;;
val power : int -> int -> int = <fun>
# power 3 4 ;;
- : int = 64
```

Again, desugaring reveals that all of the functions in the definition take a single argument.

```
let rec power =
  fun n ->
    fun x ->
      if n = 0 then 1
      else x * power (n - 1) x;;
```

As described in Section 6.2, we use the term “currying” for encoding a multi-argument function using nested, higher-order functions, as this latter definition of `power`. In OCaml, we tend to use curried functions, rather than uncurried definitions like the first definition of `power` above; the whole language is set up to make that easy to do.

We can use the `power` function to define a function to cube numbers (take numbers to the third power):

```
# let cube x = power 3 x ;;
val cube : int -> int = <fun>
# cube 4 ;;
- : int = 64
```

But since `power` is curried, we can define the `cube` function even more simply, by applying the `power` function to its “first” argument only.

```
# let cube = power 3 ;;
val cube : int -> int = <fun>
# cube 4 ;;
- : int = 64
```

A perennial source of confusion is that in this definition of the `cube` function by partial application, no overt argument of the function appears in its definition. There’s no `let cube x = ...` here. The expression `power 3` is already a function (of type `int -> int`). It *is* the cubing function, not just the result of applying the cubing function. This is PARTIAL APPLICATION: the applying of a curried function to only *some* of its arguments, resulting in a function that takes the remaining arguments.

The order in which a curried function takes its arguments thus becomes an important design consideration, as it determines what partial applications are possible. With partial application at hand, we can define other functions for powers of numbers. Here’s a version of `square`:

```
# let square = power 2 ;;
val square : int -> int = <fun>
```

```
# square 4 ;;
- : int = 16
```

Understanding what's going on in these examples is a good indication that you "get" higher-order functional programming. So we pause for a little practice with partial application.

#### Exercise 51

A TESSERACT is the four-dimensional analog of a cube, so fourth powers of numbers are sometimes referred to as TESSERACTIC NUMBERS. Use the power function to define a function tesseract that takes its integer argument to the fourth power.

Now, `map` is itself a curried function and therefore can itself be partially applied to its first argument. It takes its function argument and its list argument one at a time. Applying it only to its first argument generates a function that applies that argument function to all of the elements of a list. We can partially apply `map` to the successor function to generate the `inc_all` function we had before.

```
# let inc_all = map (fun x -> 1 + x) ;;
val inc_all : int list -> int list = <fun>
```

But there are even further opportunities for partial application.<sup>2</sup> The addition function `(+)` itself is curried, as we noted in Section 6.2. It can thus be partially applied to one argument to form the successor function: `(+)`. 1. (Recall the use of parentheses around the `+` operator in order to allow it to be used as a normal prefix function.) Notice how the types work out: Both `fun x -> 1 + x` and `(+) 1` have the same type, namely, `int -> int`. So the definition of `inc_all` can be expressed simply as is

```
# let inc_all = map ((+) 1) ;;
val inc_all : int list -> int list = <fun>

# inc_all [1; 2; 4; 8] ;;
- : int list = [2; 3; 5; 9]
```

Similarly, `square_all` can be written as the mapping of the `square` function:

```
# let square_all = map square ;;
val square_all : int list -> int list = <fun>

# square_all [1; 2; 4; 8] ;;
- : int list = [1; 4; 16; 64]
```

Compare this to the original definition of `square_all`:

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

<sup>2</sup> Partial application takes full advantage of the first-class nature of functions to enable compact and elegant definitions of functions. However, you should be aware that it does make type inference more difficult in the presence of polymorphism, an advanced topic discussed in Section 9.6 for the adventurous.

**Exercise 52**

Use the `map` function to define a function `double_all` that takes an `int list` argument and returns a list with the elements doubled.

### 8.3 The fold abstraction

Let's take a look at some other functions that bear a striking resemblance. Exercises 44 and 45 asked for definitions of functions that took, respectively, the sum and the product of the elements in a list. Here are some possible solutions, written in the recursive style of Chapter 7:

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

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

As before, note the striking similarity of these two definitions. They differ in just two places (highlighted above): an initial value to return on the empty list and the operation to apply to the next element of the list and the recursively processed suffix of the list.

This abstract pattern of computation – combining all of the elements of a list one at a time with a binary function, starting with an initial value – is called a FOLD. We repeat the abstraction process from the previous section, defining a function called `fold` to capture the abstraction.

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

Notice the two additional arguments – `f` and `init` – which correspond exactly to the two places that `sum` and `prod` differed.<sup>3</sup> In summary, the type of `fold` is `(int -> int -> int) -> int list -> int -> int`.

The `fold` abstraction is simply the repeated embedded application of a binary function, starting with an initial value, to all of the elements of a list. That is, given a list of  $n$  elements  $[x_1, x_2, x_3, \dots, x_n]$ , the `fold` of a binary function `f` with initial value `init` is

<sup>3</sup> Ideally, these two arguments – `f` and `init` – would be placed as the first two arguments of `fold` so that they could be conveniently partially applied. (In fact, the Haskell functional programming language uses that argument order for their `fold` functions.) By convention, however, the argument order for this `fold` operation in OCaml is as provided here, allowing for partially applying the `f` argument but not `init`. The `init` argument is placed at the end to reflect its use as the rightmost element being operated on during the fold. As you'll see later, the alternative `fold_left` function uses the Haskell argument order.

```
f x_1 (f x_2 (f x_3 ( ... (f x_n init) ...))) .
```

Now `sum` can be defined using `fold`:

```
# let sum lst =
#   fold (fun x y -> x + y) lst 0 ;;
val sum : int list -> int = <fun>
```

or, noting that `+` is itself the curried addition function we need as the first argument to `fold`:

```
# let sum lst = fold (+) lst 0 ;;
val sum : int list -> int = <fun>
```

The `prod` function, similarly, is a kind of `fold`, this time of the product function starting with the multiplicative identity 1.

```
# let prod lst = fold ( * ) lst 1 ;;
val prod : int list -> int = <fun>
```

A wide variety of list functions follow this pattern. Consider taking the length of a list, a function from Section 7.3.1.

```
let rec length (lst : int list) : int =
  match lst with
  | [] -> 0
  | _hd :: tl -> 1 + length tl ;;
```

This function matches the `fold` structure as well. The initial value, the length of an empty list, is 0, and the operation to apply to the head of the list and the recursively processed tail is to simply ignore the head and increment the value for the tail.

```
# let length lst = fold (fun _hd tlval -> 1 + tlval) lst 0 ;;
val length : int list -> int = <fun>
#
# length [1; 2; 4; 8] ;;
- : int = 4
```

The function that we've called `fold` operates “right-to-left” producing

```
f x_1 (f x_2 (f x_3 ( ... (f x_n init) ...))) .
```

For this reason, it is sometimes referred to as `fold_right`; in fact, that is the name of the corresponding function in OCaml's `List` module.

The symmetrical function `fold_left` operates left-to-right, calculating

```
(f ... (f (f (f init x_1) x_2) x_3) x_n) .
```

where `init` is as before an initial value, and `f` is a binary function taking as arguments the recursively processed *prefix* and the next element in the list.

**Exercise 53**

Define the higher-order function `fold_left : (int -> int -> int) -> int -> int list -> int`, which performs this left-to-right fold.

Because addition is associative, a list can be summed by either a `fold_right` as above or a `fold_left`. The definition analogous to the one using `fold_right` is

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

but (because the list argument of `fold_left` is the final argument) this can be further simplified by partial application:

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

**Exercise 54**

Define the `length` function that returns the length of a list, using `fold_left`.

**Exercise 55**

A cousin of the `fold_left` function is the function `reduce`,<sup>4</sup> which is like `fold_left` except that it uses the first element of the list as the initial value, calculating

$$(f \dots (f (f x_1 x_2) x_3) x_n) .$$

Define the higher-order function `reduce : (int -> int -> int) -> int list -> int`, which works in this way. You might define `reduce` recursively as we did with `fold` and `fold_left` or nonrecursively by using `fold_left` itself. (By its definition `reduce` is undefined when applied to an empty list, but you needn't deal with this case where it's applied to an invalid argument.)

<sup>4</sup> The higher-order functional programming paradigm founded on functions like `map` and `reduce` inspired the wildly popular Google framework for parallel processing of large data sets called, not surprisingly, MapReduce ([Dean and Ghemawat, 2004](#)).

## 8.4 The filter abstraction

The final list-processing abstraction we look at is the FILTER, which serves as an abstract version of functions that return a subset of elements of a list, such as the following examples, which return the even, odd, positive, and negative elements of an integer list.

```
# let rec evens xs =
#   match xs with
#   | [] -> []
#   | hd :: tl -> if hd mod 2 = 0 then hd :: evens tl
#                   else evens tl ;;
val evens : int list -> int list = <fun>

# let rec odds xs =
#   match xs with
#   | [] -> []
#   | hd :: tl -> if hd mod 2 <> 0 then hd :: odds tl
#                   else odds tl ;;
val odds : int list -> int list = <fun>

# let rec positives xs =
#   match xs with
```

```

# | [] -> []
# | hd :: tl -> if hd > 0 then hd :: positives tl
#           else positives tl ;;
val positives : int list -> int list = <fun>

# let rec negatives xs =
#   match xs with
#   | [] -> []
#   | hd :: tl -> if hd < 0 then hd :: negatives tl
#           else negatives tl ;;
val negatives : int list -> int list = <fun>

```

We leave the definition of an appropriate abstracted function `filter`  
`: (int -> bool) -> int list -> int list` as an exercise.

#### Exercise 56

Define a function `filter : (int -> bool) -> int list -> int list` that returns a list containing all of the elements of its second argument for which its first argument returns `true`.

#### Exercise 57

Provide definitions of `evens`, `odds`, `positives`, and `negatives` in terms of `filter`.

#### Exercise 58

Define a function `reverse : int list -> int list`, which returns the reversal of its argument list. Instead of using explicit recursion, define `reverse` by mapping, folding, or filtering.

#### Exercise 59

Define a function `append : int list -> int list -> int list` (as described in Exercise 49) to calculate the concatenation of two integer lists. Again, avoid explicit recursion, using `map`, `fold`, or `filter` functions instead.

## 8.5 Problem section: Credit card numbers and the Luhn check

Here's an interesting bit of trivia: Not all credit card numbers are well-formed. The final digit in a 16-digit credit card number is in fact a **CHECKSUM**, a digit that is computed from the previous 15 by a simple algorithm.

The algorithm used to generate the checksum is called the **LUHN CHECK**. To calculate the correct final checksum digit used in a 16-digit credit card number, you perform the following computation on the first 15 digits of the credit card number:

- Take all of the digits in an odd-numbered position (the leftmost digit being the first, not the zero-th digit, hence an odd-numbered one) and double them, subtracting nine if the doubling is greater than nine (called "casting out nines").



Figure 8.1: A sample credit card

As an example, we'll use the (partial) credit card number from the card in Figure 8.1:

4275 3156 0372 549*x*

Here, the odd-numbered digits (4, 7, 3, 5, 0, 7, 5, and 9) have been underlined. We double them and cast out nines to get 8, 5, 6, 1, 0, 5, 1, and 9.

2. Add all of the even position digits and the doubled odd position digits together. For the example above, the sum would be

$$(2 + 5 + 1 + 6 + 3 + 2 + 4) + (8 + 5 + 6 + 1 + 0 + 5 + 1 + 9) = 23 + 35 = 58 \quad .$$

3. The checksum is then the digit that when added to this sum makes it a multiple of ten. In the example above the checksum would be 2, since adding 2 to 58 generates 60, which is a multiple of 10. Thus, the sequence 4275 3156 0372 5492 is a valid credit card number, but changing the last digit to any other makes it invalid. (In particular, the final 3 in the card in Figure 8.1 is not the correct checksum!)

#### Problem 60

Define a function `odds_evens` that takes a list of integers and returns a pair of int lists, the list at the odd indices and the list at the even indices, respectively.

```
# let cc = [4; 2; 7; 5; 3; 1; 5; 6; 0; 3; 7; 2; 5; 4; 9] ;;
val cc : int list = [4; 2; 7; 5; 3; 1; 5; 6; 0; 3; 7; 2; 5; 4; 9]
# odds cc ;;
- : int list = [4; 2; 7; 5; 3; 1; 5; 6; 0; 3; 7; 2; 5; 4; 9]
```

#### Exercise 61

What is the type of the `odds_evens` function?

The process of doubling a number and “casting out nines” is easy to implement as well. Here is some code to do that:

```
# let doublemod9 (n : int) : int =
  (n * 2 - 1) mod 9 + 1 ;;
val doublemod9 : int -> int = <fun>
```

Finally, it will be useful to have a function to sum a list of integers.

#### Problem 62

Implement the function `sum` using the `List` module function `fold_left`.

All the parts are now in place to implement the Luhn check algorithm.

#### Problem 63

Implement a function `luhn` that takes a list of integers and returns the check digit for that digit sequence. (You can assume that it is called with a list of 15 integers.) For instance, for the example above

```
# luhn cc ;;
- : int = 2
```

You should feel free to use the functions `odds_evens`, `doublemod9`, `sum`, and any other OCaml library functions that you find useful and idiomatic.

»

We've used the same technique three times in this chapter – noticing redundancies in code and carving out the differing bits to find the underlying commonality. The result is a set of higher-order functions – `map`, `fold_left`, `fold_right`, and `filter` – that are broadly useful.

Determining the best place to carve up code into separate factors to take advantage of the commonalities and maximizing the utility of the factors is an important skill, the basis for **REFACTORING** of code, the name given to exactly this practice. And it turns out to match Socrates's second principle in *Phaedrus*:

PHAEDRUS: And what is the other principle, Socrates?

SOCRATES: That of dividing things again by classes, where the natural joints are, and not trying to break any part after the manner of a bad carver. ([Plato, 1927](#))

This principle deserves its own name:

*Edict of decomposition:  
Carve software at its joints.*

The edict of decomposition arises throughout programming practice, but plays an especial role in Chapter 18, where it motivates the programming paradigm of object-oriented programming. For now, however, we continue in the next chapter our pursuit of mechanisms for capturing more abstractions, by allowing generic programs that operate over various types, a technique called *polymorphism*.

## 8.6 Supplementary material

- Lab 2: Simple data structures and higher-order functions
- Problem set A.2: Higher-order functional programming



# 9

## *Polymorphism and generic programming*

What happens when the edict of intention runs up against the edict of irredundancy? The edict of intention calls for expressing clearly the intended types over which functions operate, so that the language can provide help by checking that the types are used consistently. We've heeded that edict, for example, in our definition of the higher-order function `map` from the previous chapter, repeated here:

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

The `map` function is tremendously useful for a wide variety of operations over integer lists. It seems natural to apply the same idea to other kinds of lists as well. For instance, we may want to define a function to double all of the elements of a `float` list or implement the `prods` function from Section 7.3.1 to take the products of pairs of integers in a list of such pairs. Using `map` we can try

```
# let double = map (fun x -> 2. *. x) ;;
Line 1, characters 33-34:
1 | let double = map (fun x -> 2. *. x) ;;
^
Error: This expression has type int but an expression was expected
of type
  float
# let prods = map (fun (x, y) -> x * y) ;;
Line 1, characters 21-27:
1 | let prods = map (fun (x, y) -> x * y) ;;
^~~~~~
Error: This pattern matches values of type 'a * 'b
      but a pattern was expected which matches values of type int
```

but we run afoul of the typing constraints on `map`, which can only apply functions of type `int -> int`, and not `float -> float` or `int * int -> int`.

Of course, we can implement a version of `map` for lists of these types as well:

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

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

This is where we run up against the edict of irredundancy: we've written the same code three times now, once for each set of argument types.

What we'd like is a way to map functions over lists *generically*, while still obeying the constraint that whatever type the list elements are, they are appropriate to apply the function to; and whatever type the function returns, the map returns a list of elements of that type.

### 9.1 Polymorphism

The solution to this quandary is found in POLYMORPHISM. In a language with polymorphism, like OCaml, functions can apply generically to values from any type, so long as they do so *consistently and systematically*, as the various versions of `map` above do. Nonetheless, we'd still like to keep the advantages of strong static typing, so that code can be checked for this consistency and systematicity. Then what should the type of a polymorphic version of the `map` function be?

We can get a hint of the answer by taking advantage of OCaml's type inference process, first introduced in Section 4.2.1. The type inference process combines all of the type constraints implicit in the use of typed functions together with all of the constraints in explicit typings to compute the types for all of the expressions in a program. For instance, in the definition

```
# let succ x = x + 1 ;;
val succ : int -> int = <fun>
```

it follows from the fact that the `+` function is applied to `x` that `x` must have the same type as the argument type for `+`, that is, `int`. Similarly, since `succ x` is calculated as the output of the `+` function, it must have the same type as `+`'s output type, again `int`. Since `succ`'s argument is of type `int` and output is of type `int`, its type must be `int -> int`. And in fact that is the type OCaml reports for it, even though no explicit typings were provided.

Propagating type information in this way results in a fully instantiated type `int -> int` for the `succ` function. But what if there aren't enough constraints in the code to yield a fully instantiated type? The IDENTITY FUNCTION `id`, which just returns its argument unchanged, is an example:

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

Since `x` is never involved in any applications in the definition of `id`, there are no type constraints on it. All that we can conclude is that whatever type `x` is – call it  $\alpha$  – the `id` function must take values of type  $\alpha$  as argument and return values of type  $\alpha$  as output. That is, `id` must be of type  $\alpha \rightarrow \alpha$ .

The `id` function doesn't have a fully instantiated type. It is a POLYMORPHIC FUNCTION, with a POLYMORPHIC TYPE. The term *polymorphic* means *many forms*; the `id` function can take arguments of many forms and operate on them similarly.

As the type inference process has indicated in the REPL output, to express polymorphic types, we need to extend the type expression language. We use TYPE VARIABLES to specify that *any* type can be used. We write type variables as identifiers with a prefixed quote mark – `'a`, `'b`, `'c`, and so forth – and conventionally read them as their corresponding Greek letter –  $\alpha$  (alpha),  $\beta$  (beta),  $\gamma$  (gamma) – as we've done above. Notice that OCaml has reported a polymorphic type for `id`, namely, `'a -> 'a` (read, " $\alpha$  to  $\alpha$ "). This type makes the claim, "for *any* type  $\alpha$ , if `id` is applied to an argument of type  $\alpha$  it returns a value of type  $\alpha$ ."

## 9.2 Polymorphic map

Returning to the `map` function, we wanted a way to map functions over lists generically. If we just remove the typings in the definition of `map`, it would seem that we could have just such a function, a polymorphic version of `map`.

```
# let rec map f xs =
#   match xs with
#   | [] -> []
```

```
# | hd :: tl -> f hd :: (map f tl) ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

This function performs the same computation as the previous version of `map`, just without any of the explicit type constraints enforced. The function `f` is applied to elements of `xs` and returns elements that appear in the result list, so the type of the argument of `f` must be the type of the elements of `xs` and the type of the result of `f` must be the type of the elements of the returned list *simply as a consequence of the structure of the code.*

Happily, the type inference process that OCaml uses – developed by Roger Hindley (Figure 9.1) and Robin Milner (Figure 1.7) – infers these constraints automatically, concluding that `map`, like `id`, has a polymorphic type, which the OCaml type inference system has inferred and reported as `('a -> 'b) -> 'a list -> 'b list`. This type expresses the constraint that “for *any* types  $\alpha$  and  $\beta$ , if `map` is applied to a function from  $\alpha$  values to  $\beta$  values, it will return a function that when given a list of  $\alpha$  values returns a list of  $\beta$  values.”

This polymorphic version of `map` can be used to implement `double` and `prods` as above. In each case, the types for these functions are themselves properly inferred by instantiating the type variables of the polymorphic `map` type.<sup>1</sup>

```
# let double = map (fun x -> 2. *. x) ;;
val double : float list -> float list = <fun>
# let prods = map (fun (x, y) -> x * y) ;;
val prods : (int * int) list -> int list = <fun>
```

As inferred by OCaml, `double` takes a `float list` argument and returns a `float list`, and `prods` takes an `(int * int) list` argument and returns an `int list`.

### 9.3 Regaining explicit types

By taking advantage of polymorphism in OCaml, we've satisfied the edict of irredundancy by defining a polymorphic version of `map`. Unfortunately, we seem to have forgone the edict of intention, since we are no longer explicitly providing information about the intended type for `map`.

But by using the additional expressivity provided by type variables, we can express the intended typing for `map` explicitly.

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



Figure 9.1: J. Roger Hindley (1939–), codeveloper with Robin Milner (Figure 1.7) of the Hindley-Milner type inference algorithm that OCaml relies on for inferring the most general polymorphic types for expressions.

<sup>1</sup> Note the use of partial application in these examples.

The type variables make clear the intended constraints among  $f$ ,  $xs$ , and the return value  $\text{map } f \ xs$ .

#### Problem 64

For each of the following types construct an expression for which OCaml would infer that type. For example, for the type  $\text{bool} * \text{bool}$ , the expression  $\text{true}, \text{true}$  would be a possible answer. (The idea in this exercise is not that the expressions be practical or do anything useful; they need only have the requested type. But no cheating by using explicit typing annotations with the  $:$  operator!)

1.  $\text{bool} * \text{bool} \rightarrow \text{bool}$
2.  $'a \text{ list} \rightarrow \text{bool list}$
3.  $('a * 'b \rightarrow 'a) \rightarrow 'a \rightarrow 'b \rightarrow 'a$
4.  $\text{int} * 'a * 'b \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$
5.  $\text{bool} \rightarrow \text{unit}$
6.  $'a \rightarrow ('a \rightarrow 'b) \rightarrow 'b$
7.  $'a \rightarrow 'a \rightarrow 'b$

#### Exercise 65

Define polymorphic versions of `fold` and `filter`, providing explicit polymorphic typing information.

#### Problem 66

For each of the following definitions of a function  $f$ , try to work out by hand its most general type (as would be inferred by OCaml) or explain briefly why no type exists for the function.

1.  $\text{let } f \ x = x +. 42. \text{;;}$
2.  $\text{let } f \ g \ x = g (x + 1) \text{;;}$
3.  $\text{let } f \ x = \text{match } x \text{ with} | [] \rightarrow x | h :: t \rightarrow h \text{;;}$
4.  $\text{let rec } f \ x \ a = \text{match } x \text{ with} | [] \rightarrow a | h :: t \rightarrow h (f t a) \text{;;}$
5.  $\text{let } f \ x \ y = \text{match } x \text{ with} | (w, z) \rightarrow \text{if } w \text{ then } y \ z \text{ else } w \text{;;}$
6.  $\text{let } f \ x \ y = x \ y \ y \text{;;}$
7.  $\text{let } f \ x \ y = x (y \ y) \text{;;}$
8.  $\text{let rec } f \ x = \text{match } x \text{ with} | \text{None} | \text{Some } 0 \rightarrow \text{None} | \text{Some } y \rightarrow f (\text{Some } (y - 1)) \text{;;}$
9.  $\text{let } f \ x \ y = \text{if } x \text{ then } [x] \text{ else } [\text{not } x; y] \text{;;}$

## 9.4 The List library

One way, perhaps the best, for satisfying the edict of irredundancy is to avoid writing the same code twice by *not writing the code even once*, instead taking advantage of code that someone else has already written. OCaml, like many modern languages, comes with a large set of libraries (packaged as modules, which we'll cover in Chapter 12) that provide a wide range of functionality. The `List` module in particular provides exactly the higher-order list processing functions presented in this and the previous chapter as polymorphic functions. The documentation for the `List` module gives typings and descriptions for lots of useful list processing functions. For instance, the module provides the map, fold, and filter abstractions of Chapter 8, described in the documentation as

- `map : ('a -> 'b) -> 'a list -> 'b list`  
`map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.<sup>2</sup>
- `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`  
`fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.
- `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`  
`fold_right f [a1; ...; an]` is `f a1 (f a2 (... (f an b) ...))`. Not tail-recursive.
- `filter : ('a -> bool) -> 'a list -> 'a list`  
`filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved.

<sup>2</sup> We'll come back to the issue of tail recursion in Section 16.2.2.

They can be invoked as `List.map`, `List.fold_left`, and so forth. The library provides many other useful functions, including

- `append : 'a list -> 'a list -> 'a list`  
Concatenate two lists. Same as the infix operator `@....`
- `partition : ('a -> bool) -> 'a list -> 'a list * 'a list`  
`partition p l` returns a pair of lists (`l1, l2`), where `l1` is the list of all the elements of `l` that satisfy the predicate `p`, and `l2` is the list of all the elements of `l` that do not satisfy `p`. The order of the elements in the input list is preserved.

The `List` library has further functions for sorting, combining, and transforming lists in all kinds of ways.

Although these functions are built into OCaml through the `List` library, it's still useful to have seen how they are implemented and why they have the types they have. In particular, it makes clear that the power of list processing via higher-order functional programming doesn't require special language constructs; they arise from the interactions of simple language primitives like first-class functions and structured data types.

#### **Problem 67**

Provide an implementation of the `List.map` function over a list using only a call to `List.fold_right` over the same list, or provide an argument for why it's not possible to do so.

#### **Problem 68**

Provide an implementation of the `List.fold_right` function using only a call to `List.map` over the same list, or provide an argument for why it's not possible to do so.

#### **Problem 69**

In the `list` module, OCaml provides a function `partition : ('a -> bool) -> 'a list -> 'a list * 'a list`. According to the OCaml documentation, “`partition p l` returns a pair of lists (`l1, l2`), where `l1` is the list of all the elements of `l` that satisfy the predicate `p`, and `l2` is the list of all the elements of `l` that do not satisfy `p`. The order of the elements in the input list is preserved.”

For example, we can use this to divide a list into two new ones, one containing the even numbers and one containing the odd numbers:

```
# List.partition (fun n -> n mod 2 = 0)
#           [1; 2; 3; 4; 5; 6; 7] ;;
- : int list * int list = ([2; 4; 6], [1; 3; 5; 7])
```

As described above, the `List` module provides the `partition` function of type `('a -> bool) -> 'a list -> 'a list * 'a list`. Give your own definition of `partition`, implemented directly without the use of any library functions except for those in the `Stdlib` module.

#### **Exercise 70**

Define a function `permutations : 'a list -> 'a list list`, which takes a list of values and returns a list containing every permutation of the original list. For example,

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

It doesn't matter what order the permutations appear in the returned list. Note that if the input list is of length  $n$ , then the answer should be of length  $n!$  (that is, the factorial of  $n$ ). Hint: One way to do this is to write an auxiliary function, `interleave : int -> int list -> int list list`, that yields all interleavings of its first argument into its second. For example:

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

## 9.5 Problem section: Function composition

The COMPOSITION of two unary functions `f` and `g` is the function that applies `f` to the result of applying `g` to its argument.

For example, suppose you're given a list of pairs of integers, where we think of each pair as containing a number and a corresponding

weight. We'd like to compute the WEIGHTED SUM of the numbers, that is, the sum of the numbers where each has been weighted according to (that is, multiplied by) its weight. Recall the `sum` function from Exercise 44 and the `prods` function from Section 7.3.1. The weighted average of a pair-list can be computed by applying the `sum` function to the result of applying the `prods` function to the list. Thus, `weighted_sum` is just the composition of `sum` and `prods`.

#### Problem 71

Provide an OCaml definition for a higher-order function `@+` that takes two functions as arguments and returns their composition. The function should have the following behavior:

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

Notice that by naming the function `@+`, it is used as an infix, right-associative operator. See [the operator table in the OCaml documentation](#) for further information about the syntactic properties of operators. When defining the function itself, though, you'll want to use it as a prefix operator by wrapping it in parentheses, as `(@+)`.

#### Problem 72

What is the type of the `@+` function?

## 9.6 Weak type variables

The `List` module provides polymorphic `hd` and `tl` functions for extracting the head and tail of a list.

#### Exercise 73

What are the types of the `hd` and `tl` functions? See if you can determine them without looking them up.

These can be composed to allow, for instance, extracting the head of the tail of a list, that is, the list's second item.

```
# let second = List.hd @+ List.tl ;;
val second : '_weak1 list -> '_weak1 = <fun>
```

This definition works,

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

but why did the typing of `second` have those oddly named type variables?

Type variables like `'_weak1` (with the initial underscore) are WEAK TYPE VARIABLES, not true type variables. They maintain their polymorphism only temporarily, until the first time they are applied. Weak type variables arise because in certain situations OCaml's type inference can't figure out how to express the most general types and must resort to this fallback approach.

When a function with these weak type variables is applied to arguments with a specific type, the polymorphism of the function disappears. Having applied `second` to an `int list`, OCaml further instantiates the type of `second` to *only* apply to `int list` arguments, losing its polymorphism. We can see this in two ways, first by checking its type directly,

```
# second ;;
- : int list -> int = <fun>
```

and `second` by attempting to apply it to a list of another type,

```
# second [1.0; 2.1; 3.2] ;;
Line 1, characters 8-11:
1 | second [1.0; 2.1; 3.2] ;;
          ^^^
Error: This expression has type float but an expression was
expected of type
      int
```

To correct the problem, you can of course add in specific typing information

```
# let second : float list -> float =
#   List.hd @+ List.tl ;;
val second : float list -> float = <fun>
```

but this provides no polymorphism. Alternatively, you can provide a full specification of the call pattern in the definition rather than the partial application that was used above:

```
# let second x = (List.hd @+ List.tl) x ;;
val second : 'a list -> 'a = <fun>
```

which gives OCaml sufficient hints to infer types more generally. Of course, in this case, the composition operator isn't really helping. We might as well have defined `second` more directly as

```
# let second x = List.hd (List.tl x) ;;
val second : 'a list -> 'a = <fun>
```

For the curious, if you want to see what's going on in detail, you can check out the discussion in the section “[A function obtained through partial application is not polymorphic enough](#)” in the OCaml FAQ.

## 9.7 Supplementary material

- [Lab 3: Polymorphism and record types](#)



# 10

## *Handling anomalous conditions*

Despite best efforts, on occasion a condition arises – let's call it an ANOMALY – that a function can't handle. What to do? In this chapter, we present two approaches. The function can return a value that indicates the anomaly, thereby handling the anomaly explicitly. Alternatively, the function can stop normal execution altogether, throwing control to some handler of the anomaly. In OCaml, the first approach involves *option types*, the second *exceptions*.

As a concrete example, consider a function to calculate the MEDIAN number in a list of integer values, that is, the value that has an equal number of smaller and larger values. The median can be calculated by sorting all of the values in the list and taking the middle element of the sorted list. Taking advantage of a few functions from the `List` module (`sort`, `length`, and `nth`) and the `Stdlib` module (`compare`),<sup>1</sup> we can define

```
# let median (lst: 'a list) : 'a =
#   nth (sort compare lst) (length lst / 2) ;;
val median : 'a list -> 'a = <fun>
```

We can test it out on a few lists:

```
# median [1; 5; 9; 7; 3] ;;
- : int = 5
# median [1; 2; 3; 4; 3; 2; 1] ;;
- : int = 2
# median [1; 1; 1; 1; 1] ;;
- : int = 1
# median [7] ;;
- : int = 7
```

The function works fine most of the time, but there is one anomalous condition to consider, where the median isn't well defined: What should the `median` function do on the empty list?

<sup>1</sup> Since we make heavy use of the `List` module functions in this chapter, we will open the module (but preserve `compare` as the `Stdlib` version)

```
# open List ;;
# let compare = Stdlib.compare ;;
val compare : 'a -> 'a -> int = <fun>
```

so as to avoid having to prefix each use of the functions with the `List.` module qualifier. The issue will become clearer when modules are fully introduced in Chapter 12.

### 10.1 A non-solution: Error values

You might have thought to return a special ERROR VALUE in the anomalous case. Perhaps 0 or -1 or MAX\_INT come to mind as possible error values. Augmenting the code to return a globally defined error value might look like this:

```
# let cERROR = -1 ;;
val cERROR : int = -1

# let median (lst: 'a list) : 'a =
#   if lst = [] then cERROR
#   else nth (sort compare lst) (length lst / 2) ;;
val median : int list -> int = <fun>
```

There are two problems. First, the method can lead to gratuitous type instantiation; second, and more critically, it manifests in-band signaling.

Check the types inferred for the two versions of `median` above. The original is appropriately polymorphic, of type `'a list -> 'a`. But because the error value `cERROR` used in the second version is of type `int`, `median` becomes instantiated to `int list -> int`. The code no longer applies outside the type of the error value, restricting its generality and utility. And there is a deeper problem.

Consider the sad fate of poor [Christopher Null](#), a technology journalist with a rather inopportune name. Apparently, there is a fair amount of software that uses the string "null" as an error value for cases in which no last name was provided. Errors can then be checked for using code like

```
if last_name = "null" then ...
```

You see the problem. Poor Mr. Null reports that

I've been embroiled in a cordial email battle with Bank of America, literally for years, over my email address, which is simply `null@nullmedia.com`. Using null as a mailbox name simply does not work at B of A. The system will not accept it, period. ([Null, 2015](#))

These kinds of problems confront poor Mr. Null on a regular basis.

Null has fallen afoul of IN-BAND SIGNALING of errors, in which an otherwise valid value is used to indicate an error. The string "null" is, of course, a valid string that, for all the programmer knows, might be someone's name, yet it is used to indicate a failure condition in which no name was provided. (The solution is not to use a string, "dpfnzzlwrf" say,<sup>2</sup> that is less likely to be someone's last name as the error value. That merely postpones the problem.)

Similarly, 0 or -1 or MAX\_INT are all possible values for the median of an integer list. Using one of them as an in-band error value means

<sup>2</sup> In fact, "Dpfnzzlwrf" is the name of a fictitious corporation in Jonathan Caws-Elwitt's "Letter to a Customer". ([Conley, 2009](#)) Could it also be a last name? Why not? For a while, it was my username on Skype. True story.

that users of the `median` function can't tell the difference between the value being the true median or the median being undefined.

Having dismissed the in-band error signaling approach, we turn to better solutions.

## 10.2 Option types

The first approach, like the in-band error value approach, still handles the problem explicitly, right in the return value of the function. However, rather than returning an in-band value, an `int` (or whatever the type of the list elements is), the function will return an out-of-band `None` value, that has been added to the `int` type to form an *optional int*, a value of type `int option`.

Option types are another kind of structured type, beyond the lists, tuples, and records from Chapter 7. The postfix type constructor `option` creates an option type from a base type, just as the postfix type constructor `list` does. There are two value constructors for option type values: `None` (connoting an anomalous value), and the prefix value constructor `Some`. The argument to `Some` is a value of the base type.

For the `median` function, we'll use an `int option` as the return value, or, more generically, an '`a option`'. In the anomalous condition, we return `None`, and in the normal condition in which a well-defined `median v` can be computed, we return `Some v`.

```
# let median (lst: 'a list) : 'a option =
#   if lst = [] then None
#   else Some (nth (sort compare lst) (length lst / 2)) ;;
val median : 'a list -> 'a option = <fun>

# median [1; 2; 3; 4; 42] ;;
- : int option = Some 3
# median [] ;;
- : 'a option = None
```

This version of the `median` function when applied to an `int list` does not return an `int`, even when the median is well defined. It returns an `int option`, which is a distinct type altogether. Nonetheless, a caller of this function might want access to the `int` wrapped inside the `int option` value. As with all structured types, we access the component elements of an option value via pattern matching, as in this example function, which replicates the (deprecated) in-band value solution, returning the median of the list, or the error value if no median exists:

```
# let median_or_error (lst : int list) : int =
#   match median lst with
```

```
#    | None -> cERROR
#    | Some v -> v ;;
val median_or_error : int list -> int = <fun>
```

In implementing `median` above, we used the polymorphic function `nth` : `'a list -> int -> 'a` provided by the `List` module, which given a list `lst` and an integer `index` returns the element of `lst` at the given `index` (numbered starting with 0).

```
# List.nth [1; 2; 4; 8] 2 ;;
- : int = 4
# List.nth [true; false; false] 0 ;;
- : bool = true
```

#### Exercise 74

Why do you think `nth` was designed so as to take its list argument before its index argument? The designers expected that this would be a more commonly needed abstraction than a function that returns the  $n$ -th element of a list for a particular  $n$ .

If we were to reimplement this function, it might look something like this:

```
# let rec nth (lst : 'a list) (n : int) : 'a =
#   match lst with
#   | hd :: tl ->
#     if n = 0 then hd
#     else nth tl (n - 1) ;;
Lines 2-5, characters 0-19:
2 | match lst with
3 | | hd :: tl ->
4 | if n = 0 then hd
5 | else nth tl (n - 1)...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val nth : 'a list -> int -> 'a = <fun>
```

This definition works, as shown in the following examples:

```
# nth [1; 2; 3] 1 ;;
- : int = 2
# nth [0; 1; 2] (nth [1; 2; 3] 1) ;;
- : int = 2
```

However, OCaml has warned us that the pattern match in the definition of `nth` is not exhaustive – there are possible values that will match none of the provided patterns – and helpfully provides the missing case, the empty list. Of course, if we ask to take the  $n$ -th element of an empty list, there is no element to take; this represents an anomalous condition.

Leaving the handling of this case implicit violates the edict of intention; we should clearly express what happens in all cases. Once again, we can use option types to explicitly mark the condition in the return value. We do so in a function called `nth_opt`.<sup>3</sup>

<sup>3</sup> We use the suffix `_opt` to mark functions that return an optional value, as is conventional in OCaml library functions. In fact, as noted below, the `List` module provides an `nth_opt` function in addition to its `nth` function.

```
# let rec nth_opt (lst : 'a list) (n : int) : 'a option =
#   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>

# nth_opt [1; 2; 3] 1;;
- : int option = Some 2
# nth_opt [1; 2; 3] 5;;
- : int option = None
```

**Exercise 75**

Another anomalous condition for `nth` and `nth_opt` is the use of a negative index. What currently is the behavior of `nth_opt` with negative indices? Revise the definition of `nth_opt` to appropriately handle this case as well.

**Exercise 76**

Define a function `last_opt : 'a list -> 'a option` that returns the last element in a list (as an element of the option type) if there is one, and `None` otherwise.

```
# last_opt [];;
- : 'a option = None
# last_opt [1; 2; 3; 4; 5];;
- : int option = Some 5
```

**Exercise 77**

The variance of a sequence of  $n$  numbers  $x_1, \dots, x_n$  is given by the following equation:

$$\frac{\sum_{i=1}^n (x_i - m)^2}{n - 1}$$

where  $n$  is the number of elements in the sequence,  $m$  is the arithmetic mean (or average) of the elements in the sequence, and  $x_i$  is the  $i$ -th element in the sequence. The variance is only well defined for sequences with two or more elements. (Do you see why?)

Define a function `variance : float list -> float option` that returns `None` if the list has fewer than two elements. Otherwise, it should return the variance of the numbers in its list argument, wrapped appropriately for its return type.<sup>4</sup> For example:

```
# variance [1.0; 2.0; 3.0; 4.0; 5.0];;
- : float option = Some 2.5
# variance [1.0];;
- : float option = None
```

Remember to use the floating point version of the arithmetic operators when operating on floats (`+. .*`, etc). The function `float` can convert (“cast”) an `int` to a `float`.

<sup>4</sup> If you want to compare your output with an online calculator, make sure you find one that calculates the (unbiased) sample variance.

### 10.2.1 Option poisoning

There is a problem with using option types to handle anomalies, as in `nth_opt`. Whenever we want to use the value of an `nth_opt` element in a further computation, we need to carefully extract the value from the option type. We can't, for instance, merely write

```
# nth_opt [0; 1; 2] (nth_opt [1; 2; 3] 1);;
Line 1, characters 18-39:
1 | nth_opt [0; 1; 2] (nth_opt [1; 2; 3] 1);;
```

~~~~~

*Error: This expression has type int option  
but an expression was expected of type int*

Instead we must work inside out, painstakingly extracting values and passing on Nones:

```
# match (nth_opt [1; 2; 3] 1) with
# | None -> None
# | Some v -> nth_opt [0; 1; 2] v ;;
- : int option = Some 2
```

And if that result is part of a further computation, even something as simple as adding 1 to it, we have to resort to

```
# match (nth_opt [1; 2; 3] 1) with
# | None -> None
# | Some v ->
#   match nth_opt [0; 1; 2] v with
#   | None -> None
#   | Some v -> Some (v + 1) ;;
- : int option = Some 3
```

Much of the elegance of the functional programming paradigm, the ability to simply embed function applications with other functional applications, is lost. We call this phenomenon OPTION POISONING: The introduction of an option type in an embedded computation requires verbose extraction of values and reinjecting them into option types as the computation continues. (Option poisoning is a particular instance of the dreaded programming phenomenon of the PYRAMID OF DOOM.)

Functions that regularly display anomalous conditions that ought to be directly handled by the caller are well suited for use of option types. But where an anomalous condition is rare and isn't the kind of thing that the caller should handle, an alternative approach is useful, to avoid the pyramid of doom. Rather than explicitly marking the occurrence of an anomaly in the return value, it can be implicitly dealt with by changing the flow of control of the program entirely. This is the approach based on exceptions, to which we now turn.<sup>5</sup>

### 10.3 Exceptions

Instead of modifying the return type of `nth` to allow for returning a `None` marker of an anomaly, we can leave the return type unchanged, and in case of anomaly, raise an EXCEPTION.

When an exception is raised, execution of the function *stops*. Of course, if execution stops, the function can't return a value, which is appropriate given that the existence of the anomaly means that *there's no appropriate value to return*.

<sup>5</sup> Newer techniques, such as OPTIONAL CHAINING in the Swift programming language, deal with option poisoning in a more elegant way, providing a middle ground between the verbose option handling of OCaml and the use of exceptions. For the programming-language-theory-inclined, the MONAD concept from category theory, first imported into programming languages with Haskell, generalizes the concept.

The lesson here is that continuing progress is being made in the design of programming languages to deal with new and recurring programming issues.

What about the function that called the one that raised the exception? It is expecting a value of a certain type to be returned, but in this case, no such value is supplied. The calling function thus can't return either. It stops too. And so on and so forth.

We can write a version of `nth` that raises an exception when the index is too large.

```
# let rec nth (lst : 'a list) (n : int) : 'a =
#   match lst with
#   | [] -> raise Exit
#   | hd :: tl ->
#     if n = 0 then hd
#     else nth tl (n - 1) ;;
val nth : 'a list -> int -> 'a = <fun>

# nth [1; 2; 3] 1 ;;
- : int = 2
# nth [1; 2; 3] 5 ;;
Exception: Stdlib.Exit.
# (nth [0; 1; 2] (nth [1; 2; 3] 1)) + 1 ;;
- : int = 3
```

There are several things to notice here. First, the return type of `nth` remains '`a`, not '`a option`. Under normal conditions, it returns the *n*-th element itself, not an option-wrapped version thereof. This allows its use in embedded applications (as in the third example above) without leading to the dreaded option poisoning. When an error does occur, as in the second example, execution stops and a message is printed by the OCaml REPL (“Exception: `Stdlib.Exit`.”) describing the exception that was raised, namely, the `Exit` exception defined in the `Stdlib` library module. No value is returned from the computation at all, so no value is ever printed by the REPL.

The code that actually raises the `Exit` exception is in the third line of `nth: raise Exit`. The built-in `raise` function takes as argument an expression of type `exn`, the type for exceptions. As it turns out, `Exit` is a value of that type, as can be verified directly:

```
# Exit ;;
- : exn = Stdlib.Exit
```

The `Exit` exception is provided in the `Stdlib` module as a kind of catch-all exception, but other exceptions are more appropriate to raise in different circumstances.

- The value constructor `Invalid_argument : string -> exn`, is intended for use when an argument to a function is inappropriate. It would be appropriate to use when the index of `nth` is negative.
- The value constructor `Failure : string -> exn`, is intended for use when a function isn't well-defined as called. It would be

appropriate to use when the index of `nth` is too large for the given list.

Both of these constructors take a string argument, typically used to provide an explanation of what went wrong. The explanation can be used when the exception information is handled, for instance, by the REPL printing its error message.

Taking advantage of these exceptions, `nth` can be rewritten as

```
# let rec nth (lst : 'a list) (n : int) : 'a =
#   if n < 0 then
#     raise (Invalid_argument "nth: negative index")
#   else
#     match lst with
#     | [] -> raise (Failure "nth: index too large")
#     | hd :: tl ->
#       if n = 0 then hd
#       else nth tl (n - 1) ;
# val nth : 'a list -> int -> 'a = <fun>

# nth [1; 2; 4; 8] ~-3 ;;
Exception: Invalid_argument "nth: negative index".
# nth [1; 2; 4; 8] 1 ;;
- : int = 2
# nth [1; 2; 4; 8] 42 ;;
Exception: Failure "nth: index too large".
```

We've dealt with both of the anomalous conditions by raising appropriate exceptions.

Not coincidentally, the `List.nth` function (in [the List library module](#)) works exactly this way, raising `Invalid_argument` and `Failure` exceptions under just these circumstances. But a `List.nth_opt` function is also provided, for cases in which the explicit marking of anomalies with an option type is more appropriate.

Returning to the `median` example above, and repeated here for reference (but this time using our own implementation of `nth`),

```
# let median (lst : 'a list) : 'a =
#   nth (sort compare lst) (length lst / 2) ;;
val median : 'a list -> 'a = <fun>
```

this code doesn't use option types and doesn't use the `raise` function to raise any exceptions. What *does* happen when the anomalous condition occurs?

```
# median [] ;;
Exception: Failure "nth: index too large".
```

An exception was raised, not by the `median` function, but by our `nth` function that it calls, which raises a `Failure` exception when it is called to take an element of the empty list. The exception propagates from the `nth` call to the `median` call to the top level of the REPL.

### 10.3.1 Handling exceptions

Perhaps you, as the writer of some code, have an idea about how to handle particular anomalies that might otherwise raise an exception. Rather than allow the exception to propagate to the top level, you might want to handle the exception yourself. The `try () with ()` construct allows for this.

The syntax of the construction is

```
expr ::= try exprvalue with
          | exnpattern1 -> expr1
          | exnpattern2 -> expr2
          ...

```

where *expr<sub>value</sub>* is an expression that may raise an exception, and the *exnpattern<sub>i</sub>* are patterns that match against OCaml exception expressions, rather than the normal algebraic data structures.

If the *expr<sub>value</sub>* evaluates without exception, its value is returned. However, if its evaluation raises an exception, that exception is pattern-matched sequentially against the *exnpattern<sub>i</sub>* much as in a `match` expression; for the first such pattern that matches, the corresponding *expr<sub>i</sub>* is evaluated and its value returned from the `try`.

For example, we can implement `nth_opt` in terms of `nth` by embedding the call to `nth` within a `try () with ()`:<sup>6</sup>

```
# let nth_opt (lst : 'a list) (index : int) : 'a option =
#   try
#     Some (nth lst index)
#   with
#   | Failure _
#   | Invalid_argument _ -> None ;;
val nth_opt : 'a list -> int -> 'a option = <fun>

# nth_opt [1; 2; 3] 0 ;;
- : int option = Some 1
# nth_opt [1; 2; 3] (-1) ;;
- : int option = None
# nth_opt [1; 2; 3] 4 ;;
- : int option = None
```

This implementation of `nth_opt` attempts to evaluate `Some (nth lst index)`. Under normal conditions, the call to `nth` returns a value *v*, in which case `Some v` is the result of the `try` and of the function itself. But if an exception is raised in the evaluation of the `try` – presumably by an anomalous condition in the call to `nth` – the exception raised will be matched against the two patterns and the result of that pattern match will be used. If `nth` raises either a `Failure` exception or an `Invalid_argument` exception, the result of the `try...with` will be

<sup>6</sup> We've taken advantage of the ability to use the same result expression for multiple patterns, as described in Section 7.2.1.

None (as is appropriate for an implementation of `nth_opt`). If any other exception is raised, no pattern will match and the exception will continue to propagate.

### 10.3.2 Zipping lists

As another example of handling anomalous conditions, we consider a function for “zipping” lists. The result of zipping two lists together is a list of corresponding pairs of elements of the original lists. A `zip` function in OCaml ought to have the following behavior:

```
# zip ['a'; 'b'; 'c']
#      [ 1 ; 2 ; 3 ] ;;
- : (char * int) list = [('a', 1); ('b', 2); ('c', 3)]
```

Let's try to define the function, starting with its type. The `zip` function takes two lists, with types, say, `'a list` and `'b list`, and returns a list of pairs each of which has an element from the first list (of type `'a`) and an element from the second (of type `'b`). The pairs are thus of type `'a * 'b` and the return value of type `('a * 'b) list`. The type of the whole function, then, is `'a list -> 'b list -> ('a * 'b) list`. From this, the header follows directly.

```
let rec zip (xs : 'a list)
            (ys : 'b list)
            : ('a * 'b) list =
  ...
```

We'll need the first elements of each of the lists, so we match on both lists (as a pair) to extract their parts

```
let rec zip (xs : 'a list)
            (ys : 'b list)
            : ('a * 'b) list =
  match xs, ys with
  | [], [] -> ...
  | xhd :: xtl, yhd :: ytl -> ...
```

If the lists are empty, the list of pairs of their elements is empty too.

```
let rec zip (xs : 'a list)
            (ys : 'b list)
            : ('a * 'b) list =
  match xs, ys with
  | [], [] -> []
  | xhd :: xtl, yhd :: ytl -> ...
```

Otherwise, the `zip` of the non-empty lists starts with the two heads paired. The remaining elements are the `zip` of the tails.

```
let rec zip (xs : 'a list)
            (ys : 'b list)
```

```
: ('a * 'b) list =
match xs, ys with
| [], [] -> []
| xhd :: xtl, yhd :: ytl ->
  (xhd, yhd) :: (zip xtl ytl) ;;
```

You'll notice that there's an issue. And if you don't notice, the interpreter will, as soon as we enter this definition:

```
# let rec zip (xs : 'a list)
#           (ys : 'b list)
#           : ('a * 'b) list =
#   match xs, ys with
#   | [], [] -> []
#   | xhd :: xtl, yhd :: ytl ->
#     (xhd, yhd) :: (zip xtl ytl) ;;
Lines 4-7, characters 0-27:
4 | match xs, ys with
5 | | [], [] -> []
6 | | xhd :: xtl, yhd :: ytl ->
7 | (xhd, yhd) :: (zip xtl ytl)...  

Warning 8 [partial-match]: this pattern-matching is not exhaustive.  

Here is an example of a case that is not matched:  

([], _ :: _)  

val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

There are missing match cases, in particular, when one of the lists is empty and the other isn't. This can arise whenever the two lists are of different lengths. In such a case, the zip of two lists is not well defined.

As usual, we have two approaches to addressing the anomaly, with options and with exceptions. We'll pursue them in order.

We can make explicit the possibility of error values by returning an option type.

```
let rec zip_opt (xs : 'a list)
               (ys : 'b list)
               : ('a * 'b) list option = ...
```

The normal match cases can return their corresponding option type value using the `Some` constructor.

```
let rec zip_opt (xs : 'a list)
               (ys : 'b list)
               : ('a * 'b) list option =
match xs, ys with
| [], [] -> Some []
| xhd :: xtl, yhd :: ytl ->
  Some ((xhd, yhd) :: (zip_opt xtl ytl)) ;;
```

Finally, we can add a wild-card match pattern for the remaining cases.

```
# let rec zip_opt (xs : 'a list)
#               (ys : 'b list)
```

```

#           : ('a * 'b) list option =
#   match xs, ys with
#   | [], [] -> Some []
#   | xhd :: xtl, yhd :: ytl ->
#     Some ((xhd, yhd) :: (zip_opt xtl ytl))
#   | _, _ -> None ;;
Line 7, characters 20-37:
7 | Some ((xhd, yhd) :: (zip_opt xtl ytl))
               ~~~~~~
Error: This expression has type ('c * 'd) list option
      but an expression was expected of type ('a * 'b) list

```

The interpreter tells us that there's a type problem. The recursive call `zip_opt xtl ytl` is of type `('c * 'd) list option` but the `cons` requires an `('a * 'b) list`. What we have here is a bad case of option poisoning. We'll have to decompose the return value of the recursive call to extract the list within, handling the `None` case at the same time.

```

# let rec zip_opt (xs : 'a list)
#           (ys : 'b list)
#           : ('a * 'b) list option =
#   match xs, ys with
#   | [], [] -> Some []
#   | xhd :: xtl, yhd :: ytl ->
#     match zip_opt xtl ytl with
#     | None -> None
#     | Some ztl -> Some ((xhd, yhd) :: ztl)
#   | _, _ -> None ;;
Line 10, characters 2-6:
10 | | _ , _ -> None ;;
               ^^^
Error: This pattern matches values of type 'a * 'b
      but a pattern was expected which matches values of type
      ('c * 'd) list option

```

Now what? The interpreter complains of another type mismatch, this time in the final pattern, which is of type `'a * 'b`, but which, for some reason, the interpreter thinks should be of type `('c * 'd) list option`. This kind of error is one of the most confusing for beginning OCaml programmers.

#### **Exercise 78**

Try to see if you can diagnose the problem before reading on.

The indentation of this code notwithstanding, the final pattern match is associated with the *inner* match, not the *outer* one. The inner match is, indeed, for list options. The intention was that only the lines beginning `| None...` and `| Some ...` be part of that match, but the next line has been caught up in it as well.

One simple solution is to use parentheses to make explicit the intended structure of the code.

```

# let rec zip_opt (xs : 'a list)
#           (ys : 'b list)
#           : ('a * 'b) list option =
#   match xs, ys with
#   | [], [] -> Some []
#   | xhd :: xtl, yhd :: ytl ->
#     (match zip_opt xtl ytl with
#      | None -> None
#      | Some ztl -> Some ((xhd, yhd) :: ztl))
#   | _, _ -> None ;;
val zip_opt : 'a list -> 'b list -> ('a * 'b) list option = <fun>

```

Better yet is to make explicit the patterns that fall under the wildcard allowing them to move up in the ordering.

```

# 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) ;;
val zip_opt : 'a list -> 'b list -> ('a * 'b) list option = <fun>

```

### Exercise 79

Why is it necessary to make the patterns explicit before moving them up in the ordering? What goes wrong if we leave the pattern as `_`, `_`?

As an alternative, we can implement `zip` to raise an exception on lists of unequal length. Doing so simplifies the matches, since there's no issue of option poisoning.

```

# 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 ->
#     (xhd, yhd) :: (zip xtl ytl) ;;
val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>

```

### Exercise 80

Define a function `zip_safe` that returns the `zip` of two equal-length lists, returning the empty list if the arguments are of unequal length. The implementation should call `zip`.

```

# zip_safe [1; 2; 3] [3; 2; 1] ;;
- : (int * int) list = [(1, 3); (2, 2); (3, 1)]
# zip_safe [1; 2; 3] [3; 2] ;;
- : (int * int) list = []

```

What problems do you see in this function?

### 10.3.3 Declaring new exceptions

Exceptions are first-class values, of the type `exn`. Like lists and options, exceptions have multiple value constructors. We've seen some already: `Exit`, `Failure`, `Invalid_argument`. (It's for that reason that we can pattern match against them in the `try...with` construct.)

Exceptions are exceptional in that new value constructors can be added dynamically. Here we define a new exception value constructor:

```
# exception Timeout;;
exception Timeout
```

It turns out that this exception will be used in Chapter 17.

Exception constructors can take arguments. We define an `UnboundVariable` constructor that takes a string argument, used in Chapter 13, as

```
# exception UnboundVariable of string;;
exception UnboundVariable of string
```

#### Exercise 81

In Section 6.6, we noted a problem with the definition of `fact` for computing the factorial function; it fails on negative inputs. Modify the definition of `fact` to raise an exception to make that limitation explicit.

#### Exercise 82

What are the types of the following expressions (or the values they define)?

1. `Some 42`
2. `[Some 42; None]`
3. `[None]`
4. `Exit`
5. `Failure "nth"`
6. `raise (Failure "nth")`
7. `raise`
8. `fun _ -> raise Exit`
9. `let failwith s =
 raise (Failure s)`
10. `let sample x =
 failwith "not implemented"`
11. `let sample (x : int) (b : bool) : int list option =
 failwith "not implemented"`

#### Problem 83

As in Problem 64, for each of the following OCaml function types define a function `f` (with no explicit typing annotations, that is, no uses of the `:` operator) for which OCaml would infer that type. (The functions need not be practical or do anything useful; they need only have the requested type.)

1. `int -> int -> int option`
2. `(int -> int) -> int option`
3. `'a -> ('a -> 'b) -> 'b`
4. `'a option list -> 'b option list -> ('a * 'b) list`

**Problem 84**

As in Problem 66, for each of the following function definitions of a function *f*, give a typing for the function that provides its most general type (as would be inferred by OCaml) or explain briefly why no type exists for the function.

1. 

```
let rec f x =
  match x with
  | [] -> f
  | h :: t -> raise Exit ;;
```
2. 

```
let f x =
  if x then (x, true)
  else (true, not x) ;;
```

**Problem 85**

Provide a more succinct definition of the function *f* from Problem 84(2), with the same type and behavior.

### 10.4 Options or exceptions?

Which should you use when writing code to handle anomalous conditions? Options or exceptions? This is a design decision. There is no universal right answer.

Anomalous conditions when running code cover a range of cases. One class of anomalies are conditions that should *never* occur, following from true bugs in code. For instance, when a function is applied to a set of arguments for which it was explicitly not defined – for example, applying the `median` function to an empty list, where the implementer of the `median` function has specified that it is not defined in that case – this constitutes a bug. The programmer who used the `median` function in that way has made a mistake. Unfortunately, the bug appears only at run time, when it is “too late”. The best we can do in such cases is to abort the computation, returning control to some higher level for which recovery from the bug is possible (if such a higher level even exists), and providing as much information about the bug as possible. Some programming languages provide specific tools for such cases. In OCaml, exceptions are the right tool, raising an informative exception and hoping that a higher level can recover. And proper programming practice indicates doing just that.

For cases that are not simply bugs of this sort, that is, cases that are anomalous from the usual course yet expected to be handled, the choice between options and exceptions is governed by the properties of the two approaches.

Options are explicit: The type gives an indication that an anomaly might occur, and the compiler can make sure that such anomalies are handled. Exceptions are implicit: You (and the compiler) can't tell if an exception might be raised while executing a function. But exceptions are therefore more concise. The error handling doesn't impinge on the data and so doesn't poison every downstream use of the data. Code to

handle the anomaly doesn't have to exist everywhere between where the problem occurs and where it's dealt with.

Which is more important, explicitness or concision? It depends.

- If the anomaly is a standard part of the computation, a frequent occurrence, that argues for making it explicit in an option type.
- If the anomaly is a rare occurrence, that argues for hiding it implicitly in the code.
- If the anomaly is localized to a small part of the code within which it can be handled, it makes sense to use an option type in that region.
- If the anomaly is ubiquitous, with the possibility of occurring anywhere in the code, the overhead of explicitly handling it everywhere in the code with an option type is likely too cumbersome. For example, a computation may run out of memory at more or less any point. It makes no sense to have a function return an option type, with `None` reserved for the case where the computation happened to run out of memory in the function. Rather, running out of memory is a natural use for an exception (and in fact, OCaml raises exceptions when it runs out of memory).

Is the anomalous occurrence a frequent case? Use options. A rare event? Use exceptions. Is the anomalous occurrence intrinsic to the conception? Use options. Extrinsic? Use exceptions.

Design decisions like this are ubiquitous. They are the bread and butter of the programming process. The precursor to making these decisions is possessing the tools that allow the alternative designs, the understanding of what the ramifications are, and the judgement to make a reasonable choice. The importance of having the choice is why, for instance, the `List` module provides both `nth` and `nth_opt`.

## 10.5 Unit testing with exceptions

In Section 6.7, we called for unit testing of functions to verify their correctness on representative inputs. Using the methodology of that section, we might write a unit testing function for `nth`, call it `nth_test`:

```
# let nth_test () =
#   unit_test (nth [5] 0 = 5) "nth singleton";
#   unit_test (nth [1; 2; 3] 0 = 1) "nth start";
#   unit_test (nth [1; 2; 3] 1 = 2) "nth middle" ;
val nth_test : unit -> unit = <fun>
```

We run the tests by calling the function:

```
# nth_test () ;;
nth singleton passed
nth start passed
nth middle passed
- : unit = ()
```

The test function provides a report of the performance on all of the tests, showing that all tests are passed.

As mentioned in Section 6.7, we'll want to unit test `nth` as completely as is practicable, trying examples representing as wide a range of cases as possible. For instance, we might be interested in whether `nth` works in selecting the first, a middle, and the last element of a list. We've checked the first two of these conditions, but not the third. We can adjust the testing function accordingly:

```
# let nth_test () =
#   unit_test (nth [5] 0 = 5) "nth singleton";
#   unit_test (nth [1; 2; 3] 0 = 1) "nth start";
#   unit_test (nth [1; 2; 3] 1 = 2) "nth middle";
#   unit_test (nth [1; 2; 3] 2 = 3) "nth last" ;;
val nth_test : unit -> unit = <fun>
```

What about selecting at an index that is too large, as in the example `nth [1; 2; 3] 4`? We should make sure that `nth` works properly in this case as well. But what does “works properly” mean? According to [the specification in the List module](#), `nth` should raise a `Failure` exception in this case. So we'll need a boolean expression that is `true` just in case evaluating the expression `nth [1; 2; 3] 4` raises the proper exception. We can achieve this by using a `try`  $\langle \rangle$  `with`  $\langle \rangle$  to trap any exception raised and verifying that it is the correct one. We might start with

```
# try nth [1; 2; 3] 4
# with
# | Failure _ -> true
# | _ -> false ;;
Line 3, characters 15-19:
3 | | Failure _ -> true
      ^^^
Error: This expression has type bool but an expression was expected
       of type
           int
```

but this fails to type-check, since the type of the `nth` expression is `int` (since it was applied to an `int list`), whereas the `with` clauses return a `bool`. We'll need to return a `bool` in the `try` as well. In fact, we should return `false`; if `nth [1; 2; 3] 4` manages to return a value and not raise an exception, that's a sign that `nth` has a bug! We revise the test condition to be

```
# try let _ = nth [1; 2; 3] 4 in
#     false
# with
# | Failure _ -> true
# | _ -> false ;;
- : bool = true
```

Adding this unit test to the unit testing function gives us

```
# let nth_test () =
#   unit_test (nth [5] 0 = 5) "nth singleton";
#   unit_test (nth [1; 2; 3] 0 = 1) "nth start";
#   unit_test (nth [1; 2; 3] 1 = 2) "nth middle";
#   unit_test (nth [1; 2; 3] 2 = 3) "nth last";
#   unit_test (try let _ = nth [1; 2; 3] 4 in
#               false
#             with
#                 | Failure _ -> true
#                 | _ -> false) "nth index too big";
val nth_test : unit -> unit = <fun>

# nth_test () ;;
nth singleton passed
nth start passed
nth middle passed
nth last passed
nth index too big passed
- : unit = ()
```

We'll later see more elegant ways to put together unit tests (Section 17.6).

#### Exercise 86

Augment `nth_test` to verify that `nth` works properly under additional conditions: on the empty list, with negative indexes, with lists other than integer lists, and so forth.



With options and exceptions and their corresponding types, we've completed the introduction of the major compound data types that are built into the OCaml language. Table 10.1 provides a full list of these compound types, with their type constructors and value constructors. The advantages of compound types shouldn't be limited to built-ins though. In the next chapter, we'll extend the type system to allow user-defined compound types.

#### *10.6 Supplementary material*

- Lab 4: Error handling, options, and exceptions

| <i>Type</i>  | <i>Type constructor</i>                           | <i>Value constructors</i>                                                |
|--------------|---------------------------------------------------|--------------------------------------------------------------------------|
| functions    | <code>⟨⟩ -&gt; ⟨⟩</code>                          | <code>fun ⟨⟩ -&gt; ⟨⟩</code>                                             |
| tuples       | <code>⟨⟩ * ⟨⟩</code><br><code>⟨⟩ * ⟨⟩ * ⟨⟩</code> | <code>⟨⟩ , ⟨⟩</code><br><code>⟨⟩ , ⟨⟩ , ⟨⟩</code>                        |
|              |                                                   | <code>...</code>                                                         |
| lists        | <code>⟨⟩ list</code>                              | <code>[]</code><br><code>⟨⟩ :: ⟨⟩</code><br><code>[⟨⟩ ; ⟨⟩ ; ...]</code> |
| records      | <code>{⟨⟩ : ⟨⟩ ; ⟨⟩ : ⟨⟩ ; ...}</code>            | <code>{⟨⟩ = ⟨⟩ ; ⟨⟩ = ⟨⟩ ; ...}</code>                                   |
| options      | <code>⟨⟩ option</code>                            | <code>None</code><br><code>Some ⟨⟩</code>                                |
| exceptions   | <code>exn</code>                                  | <code>Exit</code><br><code>Failure ⟨⟩</code>                             |
|              |                                                   | <code>...</code>                                                         |
| user-defined | <i>See Chapter 11</i>                             |                                                                          |

Table 10.1: Built-in compound data types.



# 11

## *Algebraic data types*

Data types can be divided into the *atomic* types (with atomic type constructors like `int` and `bool`) and the *composite* types (with parameterized type constructors like `⟨⟩ * ⟨⟩`, `⟨⟩ list`, and `⟨⟩ option`).

What is common to all of the built-in composite types introduced so far<sup>1</sup> is that they allow building data structures through the combination of just two methods.

1. *Conjunction*: Multiple components can be conjoined to form a composite value containing *all* of the components.

For instance, values of pair type, `int * float` say, are formed as the conjunction of two components, the first component an `int` and the second a `float`.

2. *Alternation*: Multiple components can be disjoined, serving as alternatives to form a composite value containing *one* of the values.

For instance, values of type `int list` are formed as the alternation of two components. One alternative is `[]`; the other is the “cons” (itself a conjunction of a component of type `int` and a component of type `int list`).

Data types built by conjunction and disjunction are called ALGEBRAIC DATA TYPES.<sup>2</sup> As mentioned, we've seen several examples already, as built-in composite data types. But why should the power of algebraic data types be restricted to built-in types? Such a simple and elegant construction like algebraic types could well be a foundational construct of the language, not only to empower programmers using the language but also to provide a foundation for the built-in constructs themselves.

OCaml inherits from its antecedents (especially, the Hope programming language developed at the University of Edinburgh, the university that brought us ML as well) the ability to define new algebraic data types as user code.

<sup>1</sup> The exception is the composite type of functions. Functions are the rare case of a composite type in OCaml not structured as an algebraic data type as defined below.

<sup>2</sup> Algebra is the mathematical study of structures that obey certain laws. Typical algebras is to form such structures by operations that have exactly the duality of conjunction and alternation found here. For instance, arithmetic algebras have multiplication and addition as, respectively, the conjunction and alternation operators. Boolean algebras have logical conjunction ('and') and disjunction ('or'). Set algebras have cross-product and union. The term *algebraic data type* derives from this connection to these structured algebras.

Let's start with a simple example based on genome processing, exemplifying the use of alternation. DNA sequences are long sequences composed of only four base amino acids: guanine (G), cytosine (C), adenine (A), and thymine (T).

We can define an algebraic data type for the DNA bases via alternation. The type, called `base`, will have four value constructors corresponding to the four base letters. The alternatives are separated by vertical bars (`|`). Here is the definition of the `base` type, introduced by the keyword `type`:

```
# type base = G | C | A | T ;;
type base = G | C | A | T
```

This kind of type declaration defines a VARIANT TYPE, which lists a set of alternatives, variant ways of building elements of the type: `A or T or C or G`.<sup>3</sup> Having defined the `base` type, we can refer to values of that type.

```
# A ;;
- : base = A
# G ;;
- : base = G
```

As with all composite types, computations that depend on the particular values of the type use pattern-matching to structure the cases. For instance, each DNA base has a complementary base: A and T are complementary, as are G and C. A function to return the complement of a base uses pattern-matching to individuate the cases:

```
# let comp_base bse =
#   match bse with
#   | A -> T
#   | T -> A
#   | G -> C
#   | C -> G ;;
val comp_base : base -> base = <fun>
# comp_base G ;;
- : base = C
```

Variants correspond to the alternation approach to building composite values. The conjunction approach is enabled by allowing the alternative value constructors to take an argument of a specified type. That argument itself can conjoin components by tupling.

As an example, DNA sequences themselves can be implemented as an algebraic data type that we'll call `dna`. Taking inspiration from the `List` type for sequences, DNA sequences can be categorized into two alternatives, two variants – the empty sequence, for which we will use the value constructor `Nil`; and non-empty sequences, for which we will use the value constructor `Cons`. The `Cons` constructor will take two

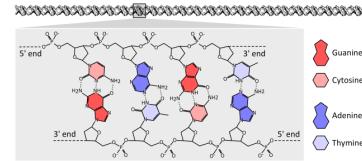


Figure 11.1: DNA carries information encoded as sequences of four amino acids.

<sup>3</sup> Using argumentless variants in this way serves the purpose of **enumerated types** in other languages – `enum` in C, C derivatives, Java, and Perl, for instance. Variants thus generalize enumerated types.

arguments (uncurried), one for the first base in the sequence and one for the rest of the dna sequence.<sup>4</sup>

```
# type dna =
#   | Nil
#   | Cons of base * dna ;;
type dna = Nil | Cons of base * dna
```

The `Cons` constructor takes two arguments (using tuple notation), the first of type `base` and the second of type `dna`. It thus serves to conjoin a `base` element and another `dna` sequence.

Having defined this new type, we can construct values of that type:

```
# let seq = Cons (A, Cons (G, Cons (T, Cons (C, Nil)))) ;;
val seq : dna = Cons (A, Cons (G, Cons (T, Cons (C, Nil))))
```

and pattern-match against them:

```
# let first_base =
#   match seq with
#   | Cons (x, _) -> x
#   | Nil -> failwith "empty sequence" ;;
val first_base : base = A
```

The `dna` type is defined recursively,<sup>5</sup> as one of its variants (`Cons`) includes another value of the same type. By using recursion, we can define data types whose values can be of arbitrary size.

To process data values of arbitrary size, recursive functions are an ideal match. A function to construct the complement of an entire DNA sequence is naturally recursive.

```
# let rec complement seq =
#   match seq with
#   | Nil -> Nil
#   | Cons (b, seq) -> Cons (comp_base b, complement seq) ;;
val complement : dna -> dna = <fun>

# complement seq ;;
- : dna = Cons (T, Cons (C, Cons (A, Cons (G, Nil))))
```

### 11.1 Built-in composite types as algebraic types

The `dna` type looks for all the world just like the `list` type built into OCaml, except for the fact that its elements are always of type `base`. Indeed, our choice of names of the value constructors (`Nil` and `Cons`) emphasizes the connection.

In fact, many of the built-in composite types can be implemented as algebraic data types in this way. Boolean values are essentially a kind of enumerated type, hence algebraic.<sup>6</sup>

```
# type bool_ = True | False ;;
type bool_ = True | False
```

<sup>4</sup> There is a subtle distinction concerning when type constructors take a single tuple argument or multiple arguments written with tuple notation. For the most part, the issue can be ignored, so long as the type definition doesn't place the argument sequence within parentheses. For the curious, see the "Note on tupled constructors" in the OCaml documentation.

<sup>5</sup> In *value* definitions (with `let`), recursion must be marked explicitly with the `rec` keyword. In *type* definitions, no such explicit marking is required, and in fact nonrecursive definitions can only be formed using distinct type names. This design decision was presumably motivated by the ubiquity of recursive type definitions as compared to recursive value definitions. It's a contentious matter as to whether this quirk of OCaml is a feature or a bug.

<sup>6</sup> We name the type `bool_` so as not to shadow the built-in type `bool`. Similarly for the underscore versions `list_` and `option_` below.

Value constructors in defined algebraic types are restricted to starting with capital letters in OCaml. The built-in type differs only in using lower case constructors `true` and `false`.

We've already seen an algebraic type implementation of base lists. Similar implementations could be generated for lists of other types.

```
# type int_list = INil | ICons of int * int_list;;
type int_list = INil | ICons of int * int_list
# type float_list = FNil | FCons of float * float_list;;
type float_list = FNil | FCons of float * float_list
```

Following the edict of irredundancy, we'd prefer not to write this same code repeatedly, differing only in the type of the list elements. Fortunately, variant type declarations can be polymorphic.

```
# type 'a list_ = Nil | Cons of 'a * 'a list_;;
type 'a list_ = Nil | Cons of 'a * 'a list_
```

In polymorphic variant data type declarations like this, a new type constructor (`list_` in this case) is defined that takes a type argument (here, the type variable '`a`'). The type constructor is always postfix, like the built-in constructors `list` and `option` that you've already seen.<sup>7</sup>

Option types can be viewed as a polymorphic variant type with two constructors for the `None` and `Some` cases.

```
# type 'a option_ = None | Some of 'a;;
type 'a option_ = None | Some of 'a
```

The point of seeing these alternative implementations of the built-in composite types (booleans, lists, options) is not that one would actually *use* these implementations. That would flout the edict of irredundancy. And the reimplementations of lists and options don't benefit from the concrete syntax niceties of the built-in versions; no infix `::` for instance, or bracketed lists. Rather than defining a `dna` type in this way, in a real application we'd just use the base `list` type. If a name for this type is desired the type name `dna` can be defined by

```
type dna = base list;;
```

The point instead is to demonstrate the power of algebraic data type definitions and show that even more of the language can be viewed as syntactic sugar for pre-provided user code. Thus, the language can again be seen as deploying a small core of basic notions to build up a highly expressive medium.

## 11.2 Example: Boolean document search

The variant type definitions in this chapter aren't the first examples of algebraic type definitions you've seen. In Section 7.4, we noted that record types were user-defined types, defined with the `type` keyword, as well.

<sup>7</sup> If we need a type constructor that takes more than one type as an argument, we use the cross-product type notation, as in the `('key, 'value)` dictionary type defined in Section 11.3.

Record types are a kind of dual to variant types. Instead of starting with alternation – this *or* that *or* the other – record types start with conjunction – this *and* that *and* the other.

As an example, consider a data type for documents. A document will be made up of a list of words (each a string), as well as some metadata about the document, perhaps its title, author, and so forth. For this example, we'll stick just to titles, so an appropriate type definition would be

```
# type document = { title : string;
#                   words : string list } ;;
type document = { title : string; words : string list; }
```

A corpus of such documents can be implemented as a document list. We build a small corpus of first lines of novels.<sup>8</sup>

```
# let first_lines : document list = (* output suppressed *)
#   [ {title = "Moby Dick";
#       words = tokenize
#             "Call me Ishmael ."};
#     {title = "Pride and Prejudice";
#       words = tokenize
#             "It is a truth universally acknowledged , \
#              that a single man in possession of a good \
#              fortune must be in want of a wife ."};
#     {title = "1984";
#       words = tokenize
#             "It was a bright cold day in April , and \
#              the clocks were striking thirteen ."};
#     {title = "Great Gatsby";
#       words = tokenize
#             "In my younger and more vulnerable years \
#              my father gave me some advice that I've \
#              been turning over in my mind ever since ."}
#   ] ;;
```

We might want to query for documents with particular patterns of words. A boolean query allows for different query types: requesting documents in which a particular word occurs; or (inductively) documents that satisfy both one query and another query; or documents that satisfy either one query or another query. We instantiate the idea in a variant type definition.

```
# type query =
#   | Word of string
#   | And of query * query
#   | Or of query * query ;;
type query = Word of string | And of query * query | Or of query *
query
```

To evaluate such queries against a document, we'll write a function eval : document -> query -> bool, which should return true just in case the document satisfies the query.

<sup>8</sup> As an aid in building a document corpus, it will be useful to have a function tokenize : string -> string list that splits up a string into its component words (here defined as any characters separated by whitespace). We use some functions from the Str library module, made available using the #load directive to the REPL, to split up the string.

```
# #load "str.cma" ;;
# let tokenize : string -> string list =
#   Str.split (Str.regexp "[ \t\n]+") ;;
val tokenize : string -> string list = <fun>
```

Did you notice the use of partial application?

We've also suppressed the output for this REPL input to save space, as indicated by the (\* output suppressed \*) comment here and elsewhere.

```
let rec eval ({title; words} : document)
  (q : query)
  : bool = ...
```

Note the use of pattern-matching right in the header line, as well as the use of field punning to simplify the pattern.

The evaluation of the query depends on its structure, so we'll want to match on that.

```
let rec eval ({title; words} : document)
  (q : query)
  : bool =
  match q with
  | Word word -> ...
  | And (q1, q2) -> ...
  | Or (q1, q2) -> ...
```

For the first variant, we merely check that the word occurs in the list of words:

```
let rec eval ({title; words} : document)
  (q : query)
  : bool =
  match q with
  | Word word -> List.mem word words
  | And (q1, q2) -> ...
  | Or (q1, q2) -> ...
```

(The function `List.mem : 'a -> 'a list -> bool` is useful here, a good reason to familiarize yourself with the rest of the `List` library module.)

What about the other variants? In these cases, we'll want to recursively evaluate the subparts of the query (`q1` and `q2`) against the same document. We've already decomposed the document into its components `title` and `words`. We could reconstruct the document as needed for the recursive evaluations:

```
let rec eval ({title; words} : document)
  (q : query)
  : bool =
  match q with
  | Word word -> List.mem word words
  | And (q1, q2) -> (eval {title; words} q1)
    && (eval {title; words} q2)
  | Or (q1, q2) -> (eval {title; words} q1)
    || (eval {title; words} q2);;
```

but this seems awfully verbose. We refer to `{title; words}` four different times. It would be helpful if we could both pattern match against the document argument and name it as a whole as well. OCaml provides a special pattern constructed as

*<pattern> as <var>*

for just such cases. Such a pattern both pattern matches against the  $\langle pattern \rangle$  as well as binding the  $\langle var \rangle$  to the expression being matched against as a whole. We use this technique both to provide a name for the document as a whole (`doc`) and to extract its components. (Once we have a variable `doc` for the document as a whole, we no longer need to refer to `title`, so we use an anonymous variable instead.)

```
let rec eval ({words; _} as doc : document)
  (q : query)
  : bool =
  match q with
  | Word word -> List.mem word words
  | And (q1, q2) -> (eval doc q1) && (eval doc q2)
  | Or (q1, q2) -> (eval doc q1) || (eval doc q2) ;;
```

That's better. But we're still calling `eval doc` four times on different subqueries. We can abstract that function and reuse it; call it `eval'`:

```
let eval ({words; _} as doc : document)
  (q : query)
  : bool =
let rec eval' (q : query) : bool =
  match q with
  | Word word -> List.mem word words
  | And (q1, q2) -> (eval' q1) && (eval' q2)
  | Or (q1, q2) -> (eval' q1) || (eval' q2) in
  ... ;;
```

There's an important idea hidden here, which follows from the scoping rules of OCaml. Because the `eval'` definition falls within the scope of the definition of `eval` and the associated variables `words` and `q`, those variables are available in the body of the `eval'` definition. And in fact, we make use of that fact by referring to `words` in the first pattern-match. (The outer `q` is actually shadowed by the inner `q`, so it isn't referred to in the body of the `eval'` definition. The occurrence of `q` in the `match q` is a reference to the `q` argument of `eval'`.)

Now that we have `eval'` defined it suffices to call it on the main query and let the recursion do the rest. At this point, however, the alternative variable name `doc` is no longer referenced, and can be eliminated.

```
# let eval ({words; _} : document)
#   (q : query)
#   : bool =
# let rec eval' (q : query) : bool =
#   match q with
#   | Word word -> List.mem word words
#   | And (q1, q2) -> (eval' q1) && (eval' q2)
#   | Or (q1, q2) -> (eval' q1) || (eval' q2) in
#   eval' q ;;
val eval : document -> query -> bool = <fun>
```

Let's try it on some sample queries. We'll use the first line of *The Great Gatsby*.

```
# let gg = nth first_lines 3 ;; (* output suppressed *)

# eval gg (Word "the") ;;
- : bool = false
# eval gg (Word "and") ;;
- : bool = true
# eval gg (And ((Word "the"), (Word "and"))) ;;
- : bool = false
# eval gg (Or ((Word "the"), (Word "and"))) ;;
- : bool = true
```

Now, we return to the original goal, to search among a whole corpus of documents for those satisfying a query. The function `eval_all : document list -> query -> string list` will return the titles of all documents in the `document list` that satisfy the query.

The `eval_all` function should be straightforward to write, as it involves filtering the document list for those satisfying the query, then extracting their titles. The `filter` and `map` list-processing functions are ideal for this.

```
# let eval_all (docs : document list)
#           (q : query)
#           : string list =
#   List.map (fun doc -> doc.title)
#           (List.filter (fun doc -> (eval doc q))
#           docs) ;;
val eval_all : document list -> query -> string list = <fun>
```

We start with the `docs`, filter them with a function that applies `eval` to select only those that satisfy the query, and then map a function over them to extract their titles.

From a readability perspective, it is unfortunate that the description of what the code is doing – start with the corpus, then filter, then map – is “inside out” with respect to how the code reads. This follows from the fact that in OCaml, functions come before their arguments in applications, whereas in this case, we like to think about a data object followed by a set of functions that are applied to it. A language with backwards application would be able to structure the code in the more readable manner.

Happily, the `Stdlib` module provides a BACKWARDS APPLICATION infix operator `|>` for just such occasions.

```
# succ 3 ;;
- : int = 4
# 3 |> succ ;;      (* start with 3; increment *)
- : int = 4
# 3 |> succ       (* start with 3; increment; ... *)
```

```
#    |> (( * ) 2) ;; (* ... and double *)
- : int = 8
```

**Exercise 87**

What do you expect the type of `|>` is?

**Exercise 88**

How could you define the backwards application operator `|>` as user code?

Taking advantage of the backwards application operator can make the code considerably more readable. Instead of

```
List.filter (fun doc -> (eval doc q))
docs
```

we can start with `docs` and then filter it:

```
docs
|> List.filter (fun doc -> (eval doc q))
```

Then we can map the title extraction function over the result:

```
docs
|> List.filter (fun doc -> (eval doc q))
|> List.map (fun doc -> doc.title)
```

The final definition of `eval_all` is then

```
# let eval_all (docs : document list)
#           (q : query)
#           : string list =
# docs
# |> List.filter (fun doc -> (eval doc q))
# |> List.map (fun doc -> doc.title) ;
val eval_all : document list -> query -> string list = <fun>
```

Some examples:

```
# eval_all first_lines (Word "and") ;;
- : string list = ["1984"; "Great Gatsby"]
# eval_all first_lines (Word "me") ;;
- : string list = ["Moby Dick"; "Great Gatsby"]
# eval_all first_lines (And (Word "and", Word "me")) ;;
- : string list = ["Great Gatsby"]
# eval_all first_lines (Or (Word "and", Word "me")) ;;
- : string list = ["Moby Dick"; "1984"; "Great Gatsby"]
```

The change in readability from using backwards application has a moral. Concrete syntax can make a big difference in the human usability of a programming language. The addition of a backwards application adds not a jot to the expressive power of the language, but when used appropriately it can dramatically reduce the cognitive load on a human reader.<sup>9</sup>

<sup>9</sup> Not coincidentally, natural languages often allow alternative orders for phrases for just this same goal of moving “heavier” phrases to the right. For example, the normal order for verb phrases with the verb “give” places the object before the recipient, as in “Arden gave the book to Bellamy”. But when the object is very “heavy” (long and complicated), it sounds better to place the object later, as in “Arden gave to Bellamy every last book in the P. G. Wodehouse collection.” Backwards application gives us this same flexibility, to move “heavy” expressions (like complicated functions) later in the code.

### 11.3 Example: Dictionaries

A dictionary is a data structure that manifests a relationship between a set of *keys* and their associated *values*. In an English dictionary, for instance, the keys are the words of the language and the associated values are their definitions. But dictionaries can be used in a huge variety of applications.

A dictionary data type will depend on the types of the keys and the values. We'll want to define the type, then, as polymorphic – a ('key, 'value) dictionary.<sup>10</sup> One approach (an exceptionally poor one as it will turn out, but bear with us) is to store the keys and values as separate equal-length lists in two record fields.

```
# type ('key, 'value) dictionary = { keys : 'key list;
#                                     values : 'value list } ;;
type ('key, 'value) dictionary = { keys : 'key list; values :
'value list; }
```

Looking up an entry in the dictionary by key, returning the corresponding value, can be performed in a few ways. Here's one:

```
# let rec lookup ({keys; values} : ('key, 'value) dictionary)
#           (request : 'key)
#           : 'value option =
# match keys, values with
# | [], [] -> None
# | key :: keys, value :: values ->
#   if key = request then Some value
#   else lookup {keys; values} request ;;
Lines 4-8, characters 0-34:
4 | match keys, values with
5 | | [], [] -> None
6 | | key :: keys, value :: values ->
7 | if key = request then Some value
8 | else lookup {keys; values} request...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
([], _::_)
val lookup : ('key, 'value) dictionary -> 'key -> 'value option =
<fun>
```

The problem with this dictionary representation is obvious. The entire notion of a dictionary assumes that for each key there is a single value. But this approach to implementing dictionaries provides no such guarantee. An illegal dictionary – like {keys = [1; 2; 3]; values = ["first"; "second"]}, in which one of the keys has no value – is representable. In such cases, the `lookup` function will raise an exception.

```
# let bad_dict = {keys = [1; 2; 3];
#                 values = ["first"; "second"]} ;;
```

<sup>10</sup> Names of type variables are arbitrary, so we might as well use that ability to give good mnemonic names to them – 'key and 'value instead of 'a and 'b – following the edict of intention in making our intentions clear to readers of the code.

```

val bad_dict : (int, string) dictionary =
  {keys = [1; 2; 3]; values = ["first"; "second"]}
# lookup bad_dict 4;;
Exception: Match_failure ("//toplevel//", 4, 0).
# lookup bad_dict 3;;
Exception: Match_failure ("//toplevel//", 4, 0).

```

Adding additional match cases merely postpones the problem.

```

# let rec lookup ({keys; values} : ('key, 'value) dictionary)
#           (request : 'key)
#           : 'value option =
#   match keys, values with
#   | [], _
#   | _, [] -> None
#   | key :: keys, value :: values ->
#     if key = request then Some value
#     else lookup {keys; values} request ;;
val lookup : ('key, 'value) dictionary -> 'key -> 'value option =
  <fun>
# lookup bad_dict 4;;
- : string option = None
# lookup bad_dict 3;;
- : string option = None

```

The function still allows data structures that do not express legal dictionaries to be used. Indeed, we can no longer even distinguish between simple cases of lookup of a missing key and problematic cases of lookup in an ill-formed dictionary structure.

A better dictionary design would make such illegal structures impossible to even represent. This idea is important enough for its own edict.

### *Edict of prevention: Make the illegal inexpressible.*

We've seen this idea before in the small. It's the basis of type checking itself, which allows the use of certain values only with functions that are appropriate to apply to them – integers with integer functions, booleans with boolean functions – preventing all other uses. In a strongly typed language like OCaml, illegal operations, like applying an integer function to a boolean value, simply can't be expressed as valid well-typed code.

The edict of prevention<sup>11</sup> challenges us to find an alternative structure in which this kind of mismatch between the keys and values can't occur. Such a structure may already have occurred to you. Instead of thinking of a dictionary as a *pair of lists* of keys and values, we can think of it as a *list of pairs* of keys and values.<sup>12</sup>

```

# type ('key, 'value) dict_entry =
#   { key : 'key; value : 'value }

```

<sup>11</sup> This idea has a long history in functional programming with algebraic data types, but seen in its crispest form is likely due to Yaron Minsky, who phrases it as “[Make illegal states unrepresentable](#).” Ben Feldman uses “[Make impossible states impossible](#).” But the idea dates back to at least the beginnings of statically typed programming languages. By referring to inexpressibility, rather than unrepresentability, we generalize the notion to include cases we consider in Chapter 12.

<sup>12</sup> An idiosyncrasy of OCaml requires that the dictionary type be defined in stages in this way, rather than all at once as

```

# type ('key, 'value) dictionary =
#   { key : 'key; value : 'value } list ;;
Line 2, characters 31-35:
2 | { key : 'key; value : 'value } list ;;
   ^^^
Error: Syntax error

```

The use of and to combine multiple type definitions into a single simultaneous definition isn't required here, but is when the type definitions are mutually recursive.

```
# and ('key, 'value) dictionary =
#   ('key, 'value) dict_entry list ;;
type ('key, 'value) dict_entry = { key : 'key; value : 'value; }
and ('key, 'value) dictionary = ('key, 'value) dict_entry list
```

The type system will now guarantee that every dictionary is a list whose elements each have a key and a value. A dictionary with unequal numbers of keys and values is *not even expressible*. The lookup function can still recur through the pairs, looking for the match:

```
# let rec lookup (dict : ('key, 'value) dictionary)
#           (request : 'key)
#           : 'value option =
# match dict with
# | [] -> None
# | {key; value} :: tl ->
#   if key = request then Some value
#   else lookup tl request ;;
val lookup : ('key, 'value) dictionary -> 'key -> 'value option =
<fun>

# let good_dict = [{key = 1; value = "one"};
#                   {key = 2; value = "two"};
#                   {key = 3; value = "three"}] ;;
val good_dict : (int, string) dict_entry list =
[{key = 1; value = "one"}; {key = 2; value = "two"};
 {key = 3; value = "three"}]
# lookup good_dict 3 ;;
- : string option = Some "three"
# lookup good_dict 4 ;;
- : string option = None
```

In this particular case, changing the structure of dictionaries to make the illegal inexpressible also very slightly simplifies the lookup code as well. But even if pursuing the edict of prevention makes code a bit more complex, it can be well worth the trouble in preventing bugs from arising in the first place.

Not all illegal states can be prevented by making them inexpressible through the structuring of the types. For instance, this updated dictionary structure still allows dictionaries that are ill-formed in allowing the same key to occur more than once. We'll return to this issue when we further apply the edict of prevention in Chapter 12.

#### **Problem 89**

The game of mini-poker is played with just six playing cards: You use only the face cards (king, queen, jack) of the two suits spades and diamonds. There is a ranking on the cards: Any spade is better than any diamond, and within a suit, the cards from best to worst are king, queen, jack.

In this two-player game, each player picks a single card at random, and the player with the better card wins.

For the record, it's a terrible game.

Provide appropriate type definitions to represent the cards used in the game. It should contain structured information about the suit and value of the cards.

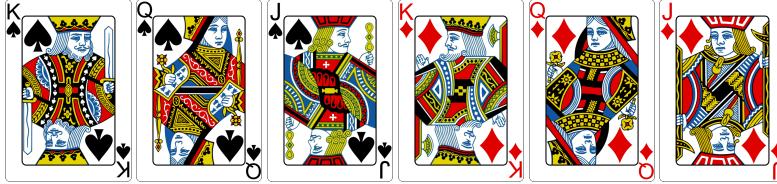


Figure 11.2: The cards of mini-poker, depicted in order from best to worst.

### Problem 90

What is an appropriate type for a function `better` that determines which of two cards is “better” in the context of mini-poker, returning `true` if and only if the first card is better than the second?

### Problem 91

Provide a definition of the function `better`.

## 11.4 Example: Arithmetic expressions

One of the elegancies admitted by the generality of algebraic data types is their use in capturing languages.

By way of example, a language of simple integer arithmetic expressions can be defined by the following grammar, written in Backus-Naur form as described in Section 3.1.

```
 $\langle \text{expr} \rangle ::= \langle \text{integer} \rangle$ 
|  $\langle \text{expr}_1 \rangle + \langle \text{expr}_2 \rangle$ 
|  $\langle \text{expr}_1 \rangle - \langle \text{expr}_2 \rangle$ 
|  $\langle \text{expr}_1 \rangle * \langle \text{expr}_2 \rangle$ 
|  $\langle \text{expr}_1 \rangle / \langle \text{expr}_2 \rangle$ 
|  $\sim\sim \langle \text{expr} \rangle$ 
```

(We'll take this to define the abstract syntax of the language. Concrete syntax notions like precedence and associativity of the operators and parentheses for disambiguating structure will be left implicit in the usual way.)

We can define a type for abstract syntax trees for these arithmetic expressions as an algebraic data type. The definition follows the grammar almost trivially, one variant for each line of the grammar.

```
# type expr =
#   | Int of int
#   | Plus of expr * expr
#   | Minus of expr * expr
#   | Times of expr * expr
#   | Div of expr * expr
#   | Neg of expr ;;
type expr =
  Int of int
| Plus of expr * expr
| Minus of expr * expr
```

```
| Times of expr * expr
| Div of expr * expr
| Neg of expr
```

The arithmetic expression given in OCaml concrete syntax as (3 + 4)

\* ~~ 5 corresponds to the following value of type expr:

```
# Times (Plus (Int 3, Int 4), Neg (Int 5)) ;;
- : expr = Times (Plus (Int 3, Int 4), Neg (Int 5))
```

A natural thing to do with expressions is to evaluate them. The recursive definition of the expr type lends itself to recursive evaluation of values of that type, as in this definition of a function eval : expr -> int.

```
# let rec eval (exp : expr) : int =
#   match exp with
#   | Int v      -> v
#   | Plus (x, y) -> (eval x) + (eval y)
#   | Minus (x, y) -> (eval x) - (eval y)
#   | Times (x, y) -> (eval x) * (eval y)
#   | Neg x       -> ~~ (eval x) ;;
Lines 2-7, characters 0-29:
2 | match exp with
3 | | Int v      -> v
4 | | Plus (x, y) -> (eval x) + (eval y)
5 | | Minus (x, y) -> (eval x) - (eval y)
6 | | Times (x, y) -> (eval x) * (eval y)
7 | | Neg x       -> ~~ (eval x)...  

Warning 8 [partial-match]: this pattern-matching is not exhaustive.  

Here is an example of a case that is not matched:  

Div (_, _)  

val eval : expr -> int = <fun>
```

Helpfully, the interpreter warns us of a missing case in the match. One of the variants in the algebraic type definition, division, is not covered by the match. A key feature of defining variant types is that the interpreter can perform these kinds of checks on your behalf. The oversight is easily corrected.

```
# let rec eval (exp : expr) : int =
#   match exp with
#   | Int v      -> v
#   | Plus (x, y) -> eval x + eval y
#   | Minus (x, y) -> eval x - eval y
#   | Times (x, y) -> eval x * eval y
#   | Div (x, y)   -> eval x / eval y
#   | Neg x       -> ~~ (eval x) ;;
val eval : expr -> int = <fun>
```

We can test the evaluator with examples like the one above.

```
# eval (Times (Plus (Int 3, Int 4), Neg (Int 5))) ;;
- : int = -35
```

```
# eval (Int 42) ;;
- : int = 42
# eval (Div (Int 5, Int 0)) ;;
Exception: Division_by_zero.
```

Of course, we already have a way of doing these arithmetic calculations in OCaml. We can just type the expressions into OCaml directly using OCaml's concrete syntax.

```
# (3 + 4) * ~- 5 ;;
- : int = -35
# 42 ;;
- : int = 42
# 5 / 0 ;;
Exception: Division_by_zero.
```

So what use is this kind of thing?

This evaluator is not trivial. By making the evaluation of this language explicit, we have the power to change the language to diverge from the language it is implemented in. For instance, OCaml's integer division truncates the result towards zero. But maybe we'd rather round to the nearest integer? We can implement the evaluator to do that instead.

### Exercise 92

Define a version of eval that implements a different semantics for the expression language, for instance, by rounding rather than truncating integer divisions.

### Exercise 93

Define a function e2s : expr → string that returns a string that represents the fully parenthesized concrete syntax for the argument expression. For instance,

```
# e2s (Times (Plus (Int 3, Int 4), Neg (Int 5))) ;;
- : string = "((3 + 4) * (~- 5))"
# e2s (Int 42) ;;
- : string = "42"
# e2s (Div (Int 5, Int 0)) ;;
- : string = "(5 / 0)"
```

The opposite process, recovering abstract syntax from concrete syntax, is called parsing. More on this in the final project (Chapter A).

## 11.5 Problem section: Binary trees

Trees are a class of data structures that store values of a certain type in a hierarchically structured manner. They constitute a fundamental data structure, second only perhaps to lists in their repurposing flexibility. Indeed, the arithmetic expressions of Section 11.4 are a kind of tree structure.

In this section, we concentrate on a certain kind of polymorphic BINARY TREE, a kind of tree whose nodes have distinct left and right subtrees, possibly empty. Some examples can be seen in Figure 11.3. A binary tree can be an empty tree (depicted with a bullet symbol (•))

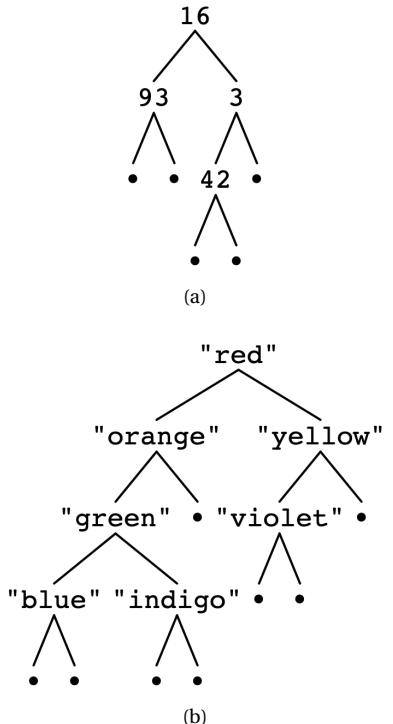


Figure 11.3: Two trees: (a) an integer tree, and (b) a string tree.

in the diagrams), or a node that stores a single value (of type '*a*, say) along with two subtrees, referred to as the left and right subtrees.

A polymorphic binary tree type can thus be defined by the following algebraic data type definition:

```
# type 'a bintree =
#   | Empty
#   | Node of 'a * 'a bintree * 'a bintree ;;
type 'a bintree = Empty | Node of 'a * 'a bintree * 'a bintree
```

For instance, the tree of Figure 11.3(a) can be encoded as an instance of an `int bintree` as

```
# let int_bintree =
#   Node (16,
#     Node (93, Empty, Empty),
#     Node (3,
#       Node (42, Empty, Empty),
#       Empty)) ;;
val int_bintree : int bintree =
  Node (16, Node (93, Empty, Empty),
        Node (3, Node (42, Empty, Empty), Empty))
```

#### Exercise 94

Construct a value `str_bintree` of type `string bintree` that encodes the tree of Figure 11.3(b).

Now let's write a function to sum up all of the elements stored in an integer tree. The natural approach to carrying out the function is to follow the recursive structure of its tree argument.

```
# let rec sum_bintree (t : int bintree) : int =
#   match t with
#   | Empty -> 0
#   | Node (n, left, right) -> n + sum_bintree left
#                                + sum_bintree right ;;
val sum_bintree : int bintree -> int = <fun>
```

#### Exercise 95

Define a function `preorder` of type `'a bintree -> 'a list` that returns a list of all of the values stored in a tree in PREORDER, that is, placing values stored at a node before the values in the left subtree, in turn before the values in the right subtree. For instance,

```
# preorder int_bintree ;;
- : int list = [16; 93; 3; 42]
```

You'll notice a certain commonality between the `sum_bintree` and `preorder` functions. Both operate by "walking" the tree, traversing it from its root down, recursively operating on the subtrees, and then combining the value stored at a node and the recursively computed values for the subtrees into the value for the tree as a whole. What differs among them is what value to return for empty trees and what function to apply to compute the overall value from the subparts. We

can abstract this tree walk functionality with a function that takes three arguments: (i) the value to use for empty trees, (ii) the function to apply at nodes to the value stored at the node and the values associated with the two subtrees, along with (iii) a tree to walk; it carries out the recursive process on that tree. Since this is a kind of “fold” operation over binary trees, we’ll name the function `foldbt`.

**Exercise 96**

What is the appropriate type for the function `foldbt` just described?

**Exercise 97**

Define the function `foldbt` just described.

**Exercise 98**

Redefine the function `sum_bintree` using `foldbt`.

**Exercise 99**

Redefine the function `preorder` using `foldbt`.

**Exercise 100**

Define a function `find : 'a bintree -> 'a -> bool` in terms of `foldbt`, such that `find t v` is true just in case the value `v` is found somewhere in the tree `t`.

```
# find int_bintree 3 ;;
- : bool = true
# find int_bintree 7 ;;
- : bool = false
```

## 11.6 Supplementary material

- Lab 5: Variants, algebraic types, and pattern matching
- Problem set A.3: Bignums and RSA encryption
- Lab 6: Recursive algebraic types
- Problem set A.4: Symbolic differentiation



## 12

# *Abstract data types and modular programming*

The algebraic data types introduced in the last chapter are an expressive tool for defining sophisticated data structures. But with great power comes great responsibility.

As an example, consider one of the most fundamental of all data structures, the QUEUE. A queue is a collection of elements that admits of operations like creating an empty queue, adding elements one by one (called ENQUEUEING), and removing them one-by-one (called DEQUEUEING), where crucially the first element enqueued is the first to be dequeued. The common terminology for this regimen is FIRST-IN-FIRST-OUT or FIFO.

We can provide a concrete implementation of the queue data type using the list data type, along with functions for enqueueing and dequeuing. An empty queue will be implemented as the empty list, with non-empty queues storing elements in order of their enqueueing, so newly enqueued elements are added at the end of the list.

```
# (* empty_queue -- An empty queue *)
# let empty_queue = [] ;;
val empty_queue : 'a list = []

# (* enqueue elt q -- Returns a queue resulting from
#    enqueueing a new elt onto q. *)
# let enqueue (elt : 'a) (q : 'a list) : 'a list =
#   q @ [elt] ;;
val enqueue : 'a -> 'a list -> 'a list = <fun>

# (* dequeue q -- Returns a pair of the next element
#    in q and the queue resulting from dequeuing
#    that element. *)
# let dequeue (q : 'a list) : 'a * 'a list =
#   match q with
#   | [] -> raise (Invalid_argument
#                   "dequeue: empty queue")
#   | hd :: tl -> hd, tl ;;
val dequeue : 'a list -> 'a * 'a list = <fun>
```

We can use these functions to enqueue and then dequeue a series of integers. Notice how the first element enqueued (the 1) is the first element dequeued.

```
# let q = empty_queue
#           |> enqueue 1      (* enqueue 1, 2, and 4 *)
#           |> enqueue 2
#           |> enqueue 4 ;;
val q : int list = [1; 2; 4]
# let next, q = dequeue q ;; (* dequeue 1 *)
val next : int = 1
val q : int list = [2; 4]
# let next, q = dequeue q ;; (* dequeue 2 *)
val next : int = 2
val q : int list = [4]
# let next, q = dequeue q ;; (* dequeue 4 *)
val next : int = 4
val q : int list = []
```

Data structures built in this way can be used as intended, as they were above. (You'll note the FIFO behavior.) But if used in unexpected ways, things can go wrong quickly. Here, for instance, we enqueue some integers, then *reverse the queue* before dequeuing the elements in a *last-in-first-out* (LIFO) order. That's not supposed to happen.

```
# let q = empty_queue
#           |> enqueue 1      (* enqueue 1, 2, and 4 *)
#           |> enqueue 2
#           |> enqueue 4
#           |> List.rev ;;     (* yikes! *)
val q : int list = [4; 2; 1]
# let next, q = dequeue q ;; (* dequeue 4 *)
val next : int = 4
val q : int list = [2; 1]
# let next, q = dequeue q ;; (* dequeue 2 *)
val next : int = 2
val q : int list = [1]
# let next, q = dequeue q ;; (* dequeue 1 *)
val next : int = 1
val q : int list = []
```

Of course, reversing the elements is *not* an operation that ought to be possible on a queue. Queues, like other data structures, are defined by what operations can be performed on them, namely, enqueue and dequeue. These operations obey an INVARIANT, that the order in which elements appear when dequeued is the same as the order in which they were enqueued. Performing inappropriate operations on data structures is the path to violating such invariants, leading to software errors. Our implementation of queues as lists allows all sorts of inappropriate operations, like reversal of the enqueued elements, or taking the  $n$ -th element, or mapping over the elements, or any

other operation appropriate for lists but not queues. What we need is the ability to enforce restraint on the operations applicable to a data structure so as to preserve the invariants.

An analogy: The lights and heating in hotel rooms are intended to be on when the room is occupied, but they should be lowered when the room is empty. We can think of this as an invariant: If the room is unoccupied, the lights and heating are off. One approach to increasing compliance with this invariant is through documentation, placing a sign at the door “Please turn off the lights when you leave.” But many hotels now use a key card switch, a receptacle near the door in which you insert the key card for the hotel room when you enter, in order to enable the lights and heating. (See Figure 12.1.) Since you have to bring your key card with you when you leave the room, thereby disabling the lights and heating, there is literally no way to violate the invariant. The state of California estimates that widespread use of hotel key card switches saves tens of millions of dollars per year ([California Utilities Statewide Codes and Standards Team, 2011](#), page 6). Preventing violation of an invariant beats documenting it.

We’ve seen this idea of avoiding illegal states before in the edict of prevention. But in the queue example, type checking doesn’t stop us from representing a bad state, and simple alternative representations for queues that prevent inappropriate operations don’t come to mind. We need a way to implement new data types and operations such that the values of those types can only be used with the intended operations. We can’t make the bad queues *unrepresentable*, but perhaps we can make them *inexpressible*, which should be sufficient for gaining the benefit of the edict of prevention.

The key idea is to provide an ABSTRACT DATA TYPE (ADT), a data type definition that provides not only a concrete IMPLEMENTATION of the data type values and operations on them, but also enforces that *only* those operations can be applied, making it impossible to express the application of other operations. This influential idea, the basis for modular programming, was pioneered by Barbara Liskov (Figure 12.2) in her CLU programming language.

The allowed operations are specified in a SIGNATURE; no other aspects of the implementation of the data type can be seen other than those specified by the signature. Users of the abstract data type can avail themselves of the functionality specified in the signature, while remaining oblivious of the particularities of the implementation. The signature specifies an *interface* to using the data structure, which serves as an ABSTRACTION BARRIER; only the aspects of the implementation specified in the signature may be made use of.

The idea of hiding aspects of the implementation from those who



Figure 12.1: Two approaches to preserving the invariant that the lights are off when the room is vacant: (a) an exhortation documenting the invariant; (b) a key card switch that disables the lights when the key is removed.



Figure 12.2: The idea of abstract data types – grouping some functionality over types and hiding the implementation of that functionality behind a strict interface – is due to computer scientist Barbara Liskov, and is first seen in her influential CLU programming language from 1974. Her work on data abstraction and object-oriented programming led to her being awarded the 2008 Turing Award, computer science’s highest honor.

shouldn't need access to those aspects is fundamental enough for an edict of its own, the edict of compartmentalization:

*Edict of compartmentalization:  
Limit information to those with a need to know.*

In the case of the queue abstract data type, all that users of the implementation have a need to know is the types for the operations involving queues, viz., the creation of queues and the enqueueing and dequeueing of elements; that's all the signature should specify. The implementation may be in terms of lists (or any of a wide variety of other methods) but the users of the abstract data type should not be able to avail themselves of the further aspects of the implementation. By preventing them from using aspects of the implementation, the invariants implicit in the signature can be maintained. A further advantage of hiding the details of the implementation of a data structure behind the abstraction barrier (in addition to making illegal operations inexpressible) is that it becomes possible to modify the implementation without affecting its use. This aspect of abstract data types is tremendously powerful.

We've seen other applications of the edict of compartmentalization before, for instance, in the use of helper functions local to (and therefore only accessible to) a function being defined. The alternative, defining the helper function globally could lead to unintended use of and reliance on that function, which had been intended only for its more focused purpose.

### 12.1 Modules

In OCaml, abstract data types are implemented using MODULES. Modules provide a way of packaging together several components – types and values involving those types, including functions manipulating values of those types – subject to constraints of a signature. A module is specified by placing the definitions of its components between the keywords `struct` and `end`:

```
struct
  <definition1>
  <definition2>
  <definition3>
  ...
end
```

Each  $\langle\text{definition}\rangle$  is a definition of a type or value (including functions, and even exceptions).

Just as values can be named using the `let` construct, modules can be named using the `module` construct:

```
module <modulename> =
  <moduledefinition>
```

## 12.2 A queue module

As a first example of the use of modules to provide for abstract data types, we return to the queue data type that we started with, which provides a type for, say, integer queues, `int_queue`, together with functions `enqueue : int -> int_queue -> int_queue` and `dequeue : int_queue -> int * int_queue`. (Even better would be to generalize queues as polymorphically allowing for elements of any base type.

We'll do so in Section 12.4.)

A module `IntQueue`<sup>1</sup> implementing the queue abstract data type is

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

### Exercise 101

Define a different implementation of integer queues as `int` lists where the elements are kept with older elements farther from the head of the list. What are the advantages and disadvantages of this implementation?

Components of a module are referenced using the already familiar notation of prefixing the module name and a dot before the component. We've seen this already in examples like `List.nth` or `Str.split`. Similarly, users of the `IntQueue` module can refer to `IntQueue.empty_queue` or `IntQueue.enqueue`. Let's use this module to perform various queue operations:

<sup>1</sup> Module names are required to begin with an uppercase letter. You've seen examples before in the `Stdlib` and `List` module names.

```
# let q = IntQueue.empty_queue
#      |> IntQueue.enqueue 1 (* enqueue 1, 2, and 4 *)
#      |> IntQueue.enqueue 2
#      |> IntQueue.enqueue 4 ;;
val q : IntQueue.int_queue = [1; 2; 4]
```

All of this module prefixing gets cumbersome quickly. We can instead just “open” the module to gain access to all of its components.<sup>2</sup>

```
# open IntQueue;;
# let q = empty_queue
#      |> enqueue 1      (* enqueue 1, 2, and 4 *)
#      |> enqueue 2
#      |> enqueue 4 ;;
val q : IntQueue.int_queue = [1; 2; 4]
# let next, q = dequeue q ;; (* dequeue 1 *)
val next : int = 1
val q : IntQueue.int_queue = [2; 4]
# let next, q = dequeue q ;; (* dequeue 2 *)
val next : int = 2
val q : IntQueue.int_queue = [4]
# let next, q = dequeue q ;; (* dequeue 4 *)
val next : int = 4
val q : IntQueue.int_queue = []
```

Unfortunately, nothing restricts us from using arbitrary aspects of the module’s implementation, for instance, reversing the elements of the queue.

```
# let q = empty_queue
#      |> enqueue 1      (* enqueue 1, 2, and 4 *)
#      |> enqueue 2
#      |> enqueue 4
#      |> List.rev (* this shouldn't be allowed *) ;;
val q : int list = [4; 2; 1]
# let next, q = dequeue q ;; (* dequeue 1 *)
val next : int = 4
val q : IntQueue.int_queue = [2; 1]
# let next, q = dequeue q ;; (* dequeue 2 *)
val next : int = 2
val q : IntQueue.int_queue = [1]
# let next, q = dequeue q ;; (* dequeue 4 *)
val next : int = 1
val q : IntQueue.int_queue = []
```

What we need is a signature that restricts the use of the components of a module, just as a type restricts use of a value. This signature/module pairing carefully separates what the caller of code sees – the module signature, which provides the abstract type structure of the components, that is, how they are *used* – from what the implementer or developer sees – the module implementation, including the concrete types and values for the components, that is, how they are *implemented*.

<sup>2</sup>A useful technique to simplify access to a module without opening it (and thereby shadowing any existing names) is to provide a short alternative name for the module.

```
# module IQ = IntQueue;;
module IQ = IntQueue
# let q = IQ.empty_queue
#      |> IQ.enqueue 1
#      |> IQ.enqueue 2
#      |> IQ.enqueue 4 ;;
val q : IQ.int_queue = [1; 2; 4]
```

Also of great utility is to open a module just within a particular local scope. OCaml provides for this with its LOCAL OPEN construct:

```
# let q =
#   let open IntQueue in
#   empty_queue
#   |> enqueue 1
#   |> enqueue 2
#   |> enqueue 4 ;;
val q : IntQueue.int_queue = [1; 2; 4]
```

The notation for specifying signatures is similar to that for modules, except for the use of `sig` instead of `struct`; and naming signatures is like naming modules with the addition of the evocative type keyword.

```
module type <modulename> =
  sig
    <definition1>
    <definition2>
    <definition3>
    ...
  end
```

We can define a signature `INT_QUEUE`<sup>3</sup> for an integer queue module:

```
# module type INT_QUEUE =
#   sig
#     type int_queue
#     val empty_queue : int_queue
#     val enqueue : int -> int_queue -> int_queue
#     val dequeue : int_queue -> int * int_queue
#   end ;;
module type INT_QUEUE =
  sig
    type int_queue
    val empty_queue : int_queue
    val enqueue : int -> int_queue -> int_queue
    val dequeue : int_queue -> int * int_queue
  end
```

The signature provides a full listing of all the aspects of a module that are visible to users of the module. In particular, the module provides a type called `int_queue`, but since the concrete implementation of that type is not provided in the signature, it is unavailable to users of modules satisfying the signature. The signature states that the module must provide a value `empty_queue` but what the concrete implementation of that value is is again hidden. And so on.

Notice that where the module implementation defines named values using the `let` construct, the signature uses the `val` construct, which provides a name and a type, but no definition of what is named.

Extending the analogy between signatures and types further, we can specify that a module satisfies and is constrained by a signature with a notation almost identical to that constraining a value to a certain type.

```
module <modulename> : <signature> =
  <moduledescription>
```

We could define `IntQueue` as satisfying the `INT_QUEUE` signature by adding this kind of “typing” as in the highlighted addition below:

```
# (* IntQueue -- An implementation of integer queues as
#   int lists, where the elements are kept with older
```

<sup>3</sup> Signature names must also begin with an uppercase letter. We follow the stylistic convention of using all uppercase for signature names.

```

#   elements closer to the head of the list. *)
# module IntQueue [ : INT_QUEUE ] =
# struct
#   type int_queue = int list
#   let empty_queue : int_queue = []
#   let enqueue (elt : int) (q : int_queue)
#     : int_queue =
#     q @ [elt]
#   let dequeue (q : int_queue) : int * int_queue =
#     match q with
#     | [] -> raise (Invalid_argument
#                     "dequeue: empty queue")
#     | hd :: tl -> hd, tl
#   end ;;
module IntQueue : INT_QUEUE

```

This module implements integer queues abstractly, allowing access only as specified by the INT\_QUEUE signature. For instance, after building a queue, we no longer have access to its concrete implementation.

```

# open IntQueue ;;
# let q = empty_queue
#   |> enqueue 1      (* enqueue 1, 2, and 4 *)
#   |> enqueue 2
#   |> enqueue 4 ;;
val q : IntQueue.int_queue = <abstr>

```

The value of `q` is reported simply as `<abstr>` connoting an abstract value hidden behind the abstraction barrier. We can't "see inside". Similarly, application of an operation not sanctioned by the signature, like list reversal, now fails.

```

# List.rev q ;;
Line 1, characters 9-10:
1 | List.rev q ;;
^
Error: This expression has type IntQueue.int_queue
      but an expression was expected of type 'a list

```

OCaml reports a type error. The function `List.rev` requires an argument of type `'a list`, but it is being applied to a queue, of type `IntQueue.int_queue`. True, the type `IntQueue.int_queue` is implemented as an `'a list`, but that fact is hidden from users of the module by the signature, hidden behind the abstraction barrier.

### 12.3 Signatures hide extra components

What happens when a module defines more components than its signature provides for? As a trivial example, we will define an ORDERED\_TYPE as a type that has an associated comparison function that provides an ordering on elements of the type. The definition of such a

module provides for these two components: a type, call it `t`, and a function that takes two elements `x` and `y` of type `t` and returns an integer indicating the ordering of the two, `-1` if `x` is smaller, `+1` if `x` is larger, and `0` if the two are equal in the ordering.<sup>4</sup>

This specification of what constitutes an ordered type can be captured in a signature `ORDERED_TYPE`:

```
# module type ORDERED_TYPE =
#   sig
#     type t
#     val compare : t -> t -> int
#   end ;;
module type ORDERED_TYPE = sig type t val compare : t -> t -> int end
```

A simple implementation of an ordered type is based on the string type. Notice that we explicitly specify the signature for the module:

```
# module StringOrderedType : ORDERED_TYPE =
#   struct
#     type t = string
#     let compare = Stdlib.compare
#   end ;;
module StringOrderedType : ORDERED_TYPE
```

We take advantage of the built in `compare` function in the `Stdlib` module,<sup>5</sup> which is a general purpose comparison function that uses the same return value convention of `-1`, `0`, `+1` for elements that are less than, equal, and greater than, respectively. A more interesting example is an ordered type for points (pairs of floats) where the ordering on points is based on which is closer to the origin. This time, however, we don't specify a signature for the module:

```
# module PointOrderedType =
#   struct
#     type t = float * float
#     let norm (x, y) =
#       x ** 2. +. y ** 2.
#     let compare p1 p2 =
#       Stdlib.compare (norm p1) (norm p2)
#   end ;;
module PointOrderedType :
```

- `sig`
- `type t = float * float`
- `val norm : float * float -> float`
- `val compare : float * float -> float * float -> int`
- `end`

We can make use of the module to see how this ordering works on some examples.

```
# let open PointOrderedType in
#   compare (1., 1.) (5., 0.),
```

<sup>4</sup> We use this arcane approach for the `compare` function to mimic the `Stdlib.compare` library function. Frankly, a better approach would be to take the result of the comparison to be a value in an enumerated type defined as `type order = Less | Equal | Greater`.

<sup>5</sup> Although the `Stdlib` prefix isn't needed – the components of the `Stdlib` module are always available – we add it here for clarity.

```
#  compare (1., 1.) (-1., -1.),
#  compare (1., 1.) (0., 1.1) ;;
- : int * int * int = (-1, 0, 1)
```

Note that the `PointOrderedType` module contains three components: the type `t`, and functions `norm` and `compare`. It goes beyond the `ORDERED_TYPE` signature in providing an extra function,

```
# PointOrderedType.norm ;;
- : float * float -> float = <fun>
# PointOrderedType.norm (1., 1.) ;;
- : float = 2.
```

since we did not explicitly restrict it to that signature. If instead we restrict `PointOrderedType` to the `ORDERED_TYPE` signature, only the components in that signature are made available.

```
# module PointOrderedType [ : ORDERED_TYPE ] =
#   struct
#     type t = float * float
#     let norm (x, y) =
#       x ** 2. +. y ** 2.
#     let compare p1 p2 =
#       Stdlib.compare (norm p1) (norm p2)
#   end ;;
module PointOrderedType : ORDERED_TYPE
```

The `norm` function is no longer defined:

```
# PointOrderedType.norm ;;
Line 1, characters 0-21:
1 | PointOrderedType.norm ;;
~~~~~
Error: Unbound value PointOrderedType.norm
```

In general, *only the aspects of a module consistent with its signature are visible outside of its implementation* to users of the module. All other aspects are hidden behind the abstraction barrier. In particular, the `norm` function is not available, and the identity of the type `t` is hidden as well. We can tell, because we no longer can compare two points.

```
# PointOrderedType.compare (1., 1.) (5., 0.) ;;
Line 1, characters 25-33:
1 | PointOrderedType.compare (1., 1.) (5., 0.) ;;
~~~~~
Error: This expression has type 'a * 'b
      but an expression was expected of type PointOrderedType.t
```

The arguments we are providing are expected to be of type `t` but we are providing arguments of type `float * float`. Although the implementation equates these types, outside of the abstraction barrier their equality isn't known. (Yes, this is a problem. We'll address it using sharing constraints later in Section 12.5.2.)

A fundamental role of modules and their signatures is to establish these abstraction barriers so that information about how data types happen to be implemented can't leak out and be taken advantage of.

### 12.4 *Modules with polymorphic components*

Returning to the queue example, there's no reason to restrict queues to integer elements. We can make the components of the module polymorphic, using type variables as usual to capture the places where arbitrary types can appear. We start with a polymorphic queue signature:

```
# module type QUEUE = sig
#   type 'a queue
#   val empty_queue : 'a queue
#   val enqueue : 'a -> 'a queue -> 'a queue
#   val dequeue : 'a queue -> 'a * 'a queue
# end ;;
module type QUEUE =
sig
  type 'a queue
  val empty_queue : 'a queue
  val enqueue : 'a -> 'a queue -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end
```

and define a queue module satisfying the signature:

```
# (* Queue -- An implementation of polymorphic queues
#   as lists, where the elements are kept with older
#   elements closer to the head of the list. *)
# module Queue : QUEUE = struct
#   type 'a queue = 'a list
#   let empty_queue : 'a queue = []
#   let enqueue (elt : 'a) (q : 'a queue) : 'a queue =
#     q @ [elt]
#   let dequeue (q : 'a queue) : 'a * 'a queue =
#     match q with
#     | [] -> raise (Invalid_argument
#                   "dequeue: empty queue")
#     | hd :: tl -> hd, tl
# end ;;
module Queue : QUEUE
```

Now we can avail ourselves of queues of different types:

```
# open Queue ;;
# let intq = empty_queue
#       |> enqueue 1
#       |> enqueue 2 ;;
val intq : int Queue.queue = <abstr>
# let boolq = empty_queue
#       |> enqueue true
```

```

#           |> enqueue false;;
val boolq : bool Queue.queue = <abstr>
# dequeue intq;;
- : int * int Queue.queue = (1, <abstr>)
# dequeue boolq;;
- : bool * bool Queue.queue = (true, <abstr>)

```

**Exercise 102**

In Section 11.3, we provided a data type for dictionaries that makes sure that the keys and values match up properly. We noted, however, that nothing prevents building a dictionary with multiple occurrences of the same key.

Define a dictionary module signature and implementation that implements dictionaries using the type from Section 11.3, and provides a function

```

add : ('key, 'value) dictionary -> 'key -> 'value ->
('key, 'value) dictionary

```

for adding a key and its value to a dictionary, and a function

```

lookup : ('key, 'value) dictionary -> 'key -> 'value
option

```

for looking keys up in the dictionary. The `add` function should raise an appropriate exception if the key being added already appears in the dictionary. The `lookup` function should return `None` if the key being looked up does not appear in the dictionary. The signature should hide the implementation of the type and the functions so that the only access to the dictionary is through these two functions.

Can you express a dictionary built using this module that has duplicate keys?

## 12.5 Abstract data types and programming for change

One of the primary advantages of using abstract data types (as opposed to concrete data structures) is that by hiding the data type implementations, the implementations can be changed without affecting users of the data types.

Recall the `query` type from Section 11.2.

```

# type query =
#   | Word of string
#   | And of query * query
#   | Or of query * query;;
type query = Word of string | And of query * query | Or of query *
query

```

In that section, a corpus of documents was structured as a list of pairs, each containing a name and a list of strings, the words in the document. Given that we'll be searching for particular words in documents, an alternative data structure useful for search is the REVERSE INDEX, a kind of dictionary with words as the keys and a set of document identifiers (the title strings, say) as the values.

If we implement this concretely, using a list of pairs for the dictionary and a string list for the set of document titles, we end up with the following type:

```
# type index = (string * (string list)) list;;
type index = (string * string list) list
```

Using a reverse index, the code for evaluating a query is quite simple:

```
# let rec eval (q : query)
#           (idx : index)
#           : string list =
# match q with
# | Word word ->
#   let (_key, targets) =
#     List.find (fun (w, _lst) -> w = word) idx
#   in targets
# | And (q1, q2) ->
#   intersection (eval q1 idx) (eval q2 idx)
# | Or (q1, q2) ->
#   (eval q1 idx) @ (eval q2 idx) ;;
Line 10, characters 0-12:
10 | intersection (eval q1 idx) (eval q2 idx)
    ~~~~~~
Error: Unbound value intersection
```

Of course, we'll need code for the intersection of two lists. Here's an approach, in which the lists are kept sorted to facilitate finding duplicates:

```
# let rec intersection set1 set2 =
# match set1, set2 with
# | [], _
# | _, [] -> []
# | h1 :: t1, h2 :: t2 ->
#   if h1 = h2 then h1 :: intersection t1 t2
#   else if h1 < h2 then intersection t1 set2
#   else intersection set1 t2 ;;
val intersection : 'a list -> 'a list -> 'a list = <fun>
```

Now, we might get lucky and notice a problematic clash of assumptions in the `eval` function. The `intersection` function assumes the lists are sorted, but the final match in `eval` just appends two lists to form the union of their elements. Nothing guarantees that the result of the union is sorted. We can fix that up by using a `sort` function from the `List` module.

```
# let rec eval (q : query)
#           (idx : index)
#           : string list =
# match q with
# | Word word ->
#   let (_, targets) =
#     List.find (fun (w, _lst) -> w = word) idx
#   in targets
# | And (q1, q2) ->
#   intersection (eval q1 idx) (eval q2 idx)
# | Or (q1, q2) ->
```

```
#      List.sort_uniq compare
#      ((eval q1 idx) @ (eval q2 idx)) ;;
val eval : query -> index -> string list = <fun>
```

But maybe then we notice that in our application, this `List.find` lookup takes too much time. It has to look through the elements of the list sequentially to find the one for the word we're looking up. That takes time proportional to the number of words being indexed. (More on this kind of issue in Chapter 14.) Maybe you recall from an earlier course that hash tables allow lookup in constant time, and you think to use them. Luckily, the `Hashtbl` library module provides hash tables. To incorporate hash tables, we have to change the `index` type:

```
# type index = (string, string list) Hashtbl.t ;;
type index = (string, string list) Hashtbl.t
```

as well as the word query lookup:

```
# let rec eval (q : query)
#           (idx : index)
#           : string list =
# match q with
# | Word word -> Hashtbl.find idx word
# | And (q1, q2) ->
#   intersection (eval q1 idx) (eval q2 idx)
# | Or (q1, q2) ->
#   List.sort_uniq compare
#   ((eval q1 idx) @ (eval q2 idx)) ;;
val eval : query -> index -> string list = <fun>
```

There's a theme here. *Every change to the underlying data representation requires multiple changes to the code*, even though nothing has changed conceptually in the underlying use of the data. We're still searching in the data, taking unions and intersections.

Let's go back to the original specification of the reverse index: "a kind of dictionary with words as the keys and a set of document identifiers (the title strings, say) as the values." This specification talks about abstract data types like dictionaries and sets, but we've been trying to directly implement them in terms of lists and pairs and hash tables. By embracing the abstractions, we can hide all of the details from our indexing code.

Suppose we had modules for string sets and for indexes. The string set module, call it `StringSet`, would presumably provide set functions like `union` and `intersection`. The index module, call it `Index` would provide a `lookup function`. The `eval` function using these modules then becomes

```
let rec eval (q : query)
            (idx : Index.dict)
            : StringSet.set =
```

```

match q with
| Word word -> (match Index.lookup idx word with
    | None -> StringSet.empty
    | Some v -> v)
| And (q1, q2) -> StringSet.intersection (eval q1 idx)
                           (eval q2 idx)
| Or (q1, q2) -> StringSet.union (eval q1 idx)
                           (eval q2 idx) ;;

```

This is much nicer. It says what the code does *at the right level of abstraction*, in terms of high-level operations like dictionary lookup, or set intersection and union. It remains silent, as it should, about exactly how those operations are implemented.

Now we'll need module definitions for `Index` and `StringSet`. We start with `StringSet` first, and in particular, its module signature, since this specifies how the module can be used.

### 12.5.1 A string set module

A string set module needs to provide some operations for creating and manipulating the sets. The requirements can be specified in a module signature. Here's a first cut:

```

# module type STRING_SET =
#   sig
#     (* Type of string sets *)
#     type set
#     (* An empty set *)
#     val empty : set
#     (* Returns true if set is empty, false otherwise *)
#     val is_empty : set -> bool
#     (* Adds string to existing set (if not already a member) *)
#     val add : string -> set -> set
#     (* Union of two sets *)
#     val union : set -> set -> set
#     (* Intersection of two sets *)
#     val intersection : set -> set -> set
#     (* Returns true iff string is in set *)
#     val member: string -> set -> bool
#   end ;;
module type STRING_SET =
sig
  type set
  val empty : set
  val is_empty : set -> bool
  val add : string -> set -> set
  val union : set -> set -> set
  val intersection : set -> set -> set
  val member : string -> set -> bool
end

```

Any implementation of this signature must provide:

- a type, called `set`;
- an element of that type called `empty`;
- a function that maps elements of the type to `bool`, called `is_empty`;
- and so forth.

From the point of view of the users (callers) of this abstract data type, this is all they need to know: The name of the type and the functions that apply to values of that type.

To drive this point home, we'll make use of an implementation (`StringSet`) of this abstract data type before even looking at the implementing code.

```
# let s = StringSet.add "c"
#           (StringSet.add "b"
#             (StringSet.add "a" StringSet.empty)) ;;
val s : StringSet.set = <abstr>
```

Note that the string set we've called `s` is of the abstract type `StringSet.set` and the particulars of the value implementing the set are hidden from us as `<abstr>`.

The types, values, and functions provided in the signature are normal OCaml objects that interact with the rest of the language as usual. We can still avail ourselves of the rest of OCaml. For instance, we can clean up the definition of `s` using reverse application and a local open:

```
# let s =
#   let open StringSet in
#   empty
#   |> add "a"
#   |> add "b"
#   |> add "c" ;;
val s : StringSet.set = <abstr>
```

Other operations work as well.

```
# StringSet.member "a" s ;;
- : bool = true
# StringSet.member "d" s ;;
- : bool = false
```

Of course, the ADT must have an actual implementation for it to work. We've just been assuming one, but we can provide a possible implementation (the one we've been using as it turns out), obeying the specific signature we just defined.<sup>6</sup>

```
module StringSet : STRING_SET =
(* Implementation of STRING_SET as list of strings.
Assumes list may be unsorted but with no duplicates. *)
struct
```

<sup>6</sup> You'll notice that we don't bother adding types to the definitions of the values in this module implementation. Since the signature already provided explicit types (satisfying the edict of intention), OCaml can verify that the implementation respects those types. Nonetheless, it can sometimes be useful to provide further typing information in a module implementation.

```

type set = string list
let empty = []
let is_empty set = (set = [])
let member = List.mem
let add elt set =
  if List.mem elt set then set
  else elt :: set
let union = List.fold_right add
let rec intersection set1 set2 =
  match set1 with
  | [] -> []
  | hd :: tl -> let tlist = intersection tl set2 in
    if member hd set2 then add hd tlist
    else tlist
end ;;

```

In this implementation, sets are implemented as string lists. A comment documents the invariant in the implementation that the lists have no duplicates, though they might not be sorted. But there's no way for a user of this module to know any of that; the signature doesn't reveal anything about the implementation type. Even though the sets are implemented as string lists, if we try to do string-list-like operations, we'll be thwarted.

```

# s @ ["b"; "e"] ;;
Line 1, characters 0-1:
1 | s @ ["b"; "e"] ;;
^
Error: This expression has type StringSet.set
      but an expression was expected of type 'a list

```

And it's a good thing too, because if we could have added the "b" to the list, suddenly, the list doesn't obey the invariant required by the implementation that there be no duplicates. But because of the abstraction barrier, there's no way for a user of the module to break the invariant, so long as the implementation maintains it.

Because the sets are implemented as unsorted lists, when taking the union of two sets `set1` and `set2`, we must traverse the entirety of the `set2` list once for each element of `set1`. For small sets, this is not likely to be problematic, and worrying about this inefficiency may well be a premature effort at optimization.<sup>7</sup> But for a set implementation likely to be used widely and on very large sets, it may be useful to address the issue.

A better alternative from an efficiency point of view is to implement sets as sorted lists. This requires a bit more work in adding elements to a set to place them in the right order, but saves effort for union and intersection. We redefine the `StringSet` module accordingly, still satisfying the same `STRING_SET` signature.

```
# (* Implementation of STRING_SET as list of strings.
```

<sup>7</sup> In the introduction to Chapter 14 you'll learn that "premature optimization is the root of all evil."

```

#      Assumes list is *sorted* with no duplicates. *)
# module StringSet : STRING_SET =
# struct
#   type set = string list
#   let empty = []
#   let is_empty s = (s = [])
#   let rec member elt s =
#     match s with
#     | [] -> false
#     | hd :: tl -> if elt = hd then true
#                      else if elt < hd then false
#                      else member elt tl
#   let rec add elt s =
#     match s with
#     | [] -> [elt]
#     | hd :: tl -> if elt < hd then elt :: s
#                      else if elt = hd then s
#                      else hd :: add elt tl
#   let union = List.fold_right add
#   let rec intersection set1 set2 =
#     match set1, set2 with
#     | [], _ -> []
#     | _, [] -> []
#     | h1::t1, h2::t2 ->
#       if h1 = h2 then h1 :: intersection t1 t2
#       else if h1 < h2 then intersection t1 set2
#       else intersection set1 t2
#   end ;;
module StringSet : STRING_SET

```

Now we can test the revised definition.

```

# let s =
#   let open StringSet in
#   empty
#   |> add "a"
#   |> add "b"
#   |> add "c" ;;
val s : StringSet.set = <abstr>
# StringSet.member "a" s ;;
- : bool = true
# StringSet.member "d" s ;;
- : bool = false

```

And here's the payoff. Even though we've completely changed the implementation of string sets, even using a data structure obeying a different invariant, the code for using string sets changes *not at all*.

### 12.5.2 A generic set signature

For document querying, we needed a string set module. For other purposes we may need sets of other element types. We could generate similar modules for, say, integer sets, with an appropriate signature:

```

# module type INT_SET =
#   sig
#     (* Type of integer sets *)
#     type set
#     (* The empty set *)
#     val empty : set
#     (* Returns true if set is empty; false otherwise *)
#     val is_empty : set -> bool
#     (* Adds integer to existing set (if not already a member) *)
#     val add : int -> set -> set
#     (* Union of two sets *)
#     val union : set -> set -> set
#     (* Intersection of two sets *)
#     val intersection : set -> set -> set
#     (* Returns true iff integer is in set *)
#     val member: int -> set -> bool
#   end ;;
module type INT_SET =
sig
  type set
  val empty : set
  val is_empty : set -> bool
  val add : int -> set -> set
  val union : set -> set -> set
  val intersection : set -> set -> set
  val member : int -> set -> bool
end

```

but we'd be violating the edict of irredundancy. Rather, we'd prefer a generic signature for set modules that provides a set type for any element type.

Here is such a signature. We've added a new type to the module, the type `element` for elements of the set, and we use it in the types of the various functions.

```

# module type SET =
#   sig
#     (* Type of sets *)
#     type set
#     (* and their elements *)
#     type element
#     (* The empty set *)
#     val empty : set
#     (* Returns true if set is empty; false otherwise *)
#     val is_empty : set -> bool
#     (* Adds element to existing set (if not already a member) *)
#     val add : element -> set -> set
#     (* Union of two sets *)
#     val union : set -> set -> set
#     (* Intersection of two sets *)
#     val intersection : set -> set -> set
#     (* Returns true iff element is in set *)
#     val member: element -> set -> bool

```

```
# end ;;
module type SET =
sig
  type set
  type element
  val empty : set
  val is_empty : set -> bool
  val add : element -> set -> set
  val union : set -> set -> set
  val intersection : set -> set -> set
  val member : element -> set -> bool
end
```

A string set implementation satisfying this signature defines the `element` type as `string`:

```
# module StringSet : SET =
#   struct
#     type element = string
#     type set = element list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
#       | [] -> false
#       | hd :: tl -> if elt = hd then true
#                      else if elt < hd then false
#                      else member elt tl
#     let rec add elt s =
#       match s with
#       | [] -> [elt]
#       | hd :: tl -> if elt < hd then elt :: s
#                      else if elt = hd then s
#                      else hd :: add elt tl
#     let union = List.fold_right add
#     let rec intersection set1 set2 =
#       match set1, set2 with
#       | [], _ -> []
#       | _, [] -> []
#       | h1::t1, h2::t2 ->
#         if h1 = h2 then h1 :: intersection t1 t2
#         else if h1 < h2 then intersection t1 set2
#         else intersection set1 t2
#   end ;;
module StringSet : SET
```

We can use this `StringSet` to, for instance, generate an empty string set:

```
# StringSet.empty ;;
- : StringSet.set = <abstr>
```

We run into a major problem, though, in the simple act of checking if a string is a member of the set.

```
# StringSet.member "a" StringSet.empty ;;
Line 1, characters 17-20:
1 | StringSet.member "a" StringSet.empty ;;
^ ^
Error: This expression has type string but an expression was
expected of type
StringSet.element
```

What's the problem? It turns out that the abstraction barrier provided by the SET signature is doing exactly what it should. The implementation promises to deliver something that satisfies and reveals SET. And that's all. The SET signature reveals types `set` and `element`, not `string` `list` and `string`. Viewed from within the implementation, the types `element` and `string` are the same. But from outside the module implementation, only `element` is available, leading to the type mismatch with `string`.

This is a case in which the abstraction barrier is too strict. (We saw this before in Section 12.3.) We do want to allow the user of the module to have access to the implementation of the `element` type, if only so that module users can provide elements of that type. Rather than using the too abstract SET signature, we can define slightly less abstract signatures using SHARING CONSTRAINTS, which augment a signature with one or more type equalities across the abstraction barrier, identifying abstract types within the signature (`element`) with implementations of those types accessible outside the implementation (`string`).<sup>8</sup>

```
# module type STRING_SET = SET with type element = string ;;
module type STRING_SET =
sig
  type set
  type element = string
  val empty : set
  val is_empty : set -> bool
  val add : element -> set -> set
  val union : set -> set -> set
  val intersection : set -> set -> set
  val member : element -> set -> bool
end
```

Now we can declare the implementation as satisfying this relaxed signature.

```
# module StringSet : STRING_SET =
#   struct
#     type element = string
#     type set = element list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
```

<sup>8</sup> Notice how in printing out the result of defining the new STRING\_SET signature, OCaml specifies that the type of elements is `string`. Compare this with the version above without the sharing constraint.

This example requires only a single sharing constraint, but multiple constraints can be useful as well. They are combined with the `and` keyword, for example, the pair of sharing constraints `with type key = D.key` and `type value = D.value` used in the definition of the `MakeOrderedDict` module in Section 12.6.

```

#      | [] -> false
#      | hd :: tl -> if elt = hd then true
#                      else if elt < hd then false
#                      else member elt tl
# let rec add elt s =
#   match s with
#   | [] -> [elt]
#   | hd :: tl -> if elt < hd then elt :: s
#                   else if elt = hd then s
#                   else hd :: add elt tl
# let union = List.fold_right add
# let rec intersection set1 set2 =
#   match set1, set2 with
#   | [], _ -> []
#   | _, [] -> []
#   | h1::t1, h2::t2 ->
#     if h1 = h2 then h1 :: intersection t1 t2
#     else if h1 < h2 then intersection t1 set2
#     else intersection set1 t2
# end ;;
module StringSet : STRING_SET

```

This implementation now allows us to perform operations involving particular strings.

```

# StringSet.empty ;;
- : StringSet.set = <abstr>
# StringSet.member "a" StringSet.empty ;;
- : bool = false
# let s =
#   let open StringSet in
#   empty
#   |> add "first"
#   |> add "second"
#   |> add "third" ;;
val s : StringSet.set = <abstr>
# StringSet.union s s ;;
- : StringSet.set = <abstr>
# StringSet.member "a" s ;;
- : bool = false

```

### 12.5.3 A generic set implementation

Sharing constraints solve the problem of duplicative signatures, because we can define different signatures by adding different sharing constraints to the generic SET signature:

```

# module type STRING_SET =
#   SET with type element = string ;;
# module type INT_SET =
#   SET with type element = int ;;
# module type INTBOOL_SET =
#   SET with type element = int * bool ;;

```

Unfortunately, they do nothing for the problem of duplicative implementations. To implement a module satisfying the INT\_SET signature, we'd need to build the whole module from scratch, like this:

```
# module IntSet : INT_SET =
#   struct
#     type element = int
#     type set = element list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
#       | [] -> false
#       | hd :: tl -> if elt = hd then true
#                      else if elt < hd then false
#                      else member elt tl
#     let rec add elt s =
#       match s with
#       | [] -> [elt]
#       | hd :: tl -> if elt < hd then elt :: s
#                      else if elt = hd then s
#                      else hd :: add elt tl
#     let union = List.fold_right add
#     let rec intersection set1 set2 =
#       match set1, set2 with
#       | [], _ -> []
#       | _, [] -> []
#       | h1::t1, h2::t2 ->
#         if h1 = h2 then h1 :: intersection t1 t2
#         else if h1 < h2 then intersection t1 set2
#         else intersection set1 t2
#   end;;
```

The redundancy is massive; the only differences from the `StringSet` implementation are those highlighted in red. To solve this violation of the edict of irredundancy requires more powerful tools.

What we need is a way of *generating* implementations that depend on some stuff. In the case at hand, the stuff is just the implementation of the `element` type, and perhaps some functionality involving that type. For instance, in the implementations of the `StringSet` and `IntSet` modules, we availed ourselves of comparing elements using the `<` operator. Any type we build a set from using this implementation approach needs some way of performing such comparisons, but the `<` operator may not always be appropriate for that purpose. More generally, the implementations may depend not only on a type but on some values of that type or functions over the type, or even multiple types.

If only we had a way of packaging up some types and related values and functions. But we do have such a way: the module system itself. In effect, what we need is something akin to a function that takes as ar-

gument a module defining the parameters of the implementation and returns the desired module. We call these “functions” from modules to modules FUNCTORS.

We can use the `StringSet` and `IntSet` implementations as the basis for a functor `MakeOrderedSet`, which takes a module as argument to provide the element type and returns a module satisfying the `SET` signature. As described above, the argument module should have a type (call it `t`) and a way of comparing elements of the type (call it `compare`). We’ll have the `compare` function take two elements of type `t` and return an integer specifying whether the first integer is less than (-1), equal to (0), or greater than (1) the second integer.

You may recognize this signature. It’s the `ORDERED_TYPE` signature from Section 12.3, repeated here for reference.

```
# module type ORDERED_TYPE =
#   sig
#     type t
#     val compare : t -> t -> int
#   end ;;
module type ORDERED_TYPE = sig type t val compare : t -> t -> int
end
```

The argument to the functor should satisfy this signature.

A functor that takes a module with this signature and delivers a `SET` implementation can be constructed just by factoring out the type and the comparison from our previous implementations of `IntSet` and `StringSet`.

```
# module MakeOrderedSet (Elements : ORDERED_TYPE) : SET =
#   struct
#     type element = Elements.t
#     type set = element list
#     let empty = []
#     let is_empty s = (s = [])
#     let rec member elt s =
#       match s with
#       | [] -> false
#       | hd :: tl ->
#         (match Elements.compare elt hd with
#          | 0 (* equal *) -> true
#          | -1 (* less *) -> false
#          | _ (* greater *) -> member elt tl)
#     let rec add elt s =
#       match s with
#       | [] -> [elt]
#       | hd :: tl ->
#         (match Elements.compare elt hd with
#          | 0 (* equal *) -> s
#          | -1 (* less *) -> elt :: s
#          | _ (* greater *) -> hd :: add elt tl)
#     let union = List.fold_right add
```

```

#     let rec intersection set1 set2 =
#       match set1, set2 with
#         | [], _ -> []
#         | _, [] -> []
#         | h1::t1, h2::t2 ->
#           (match Elements.compare h1 h2 with
#             | 0 (* equal *) -> h1 :: intersection t1 t2
#             | -1 (* less *) -> intersection t1 set2
#             | _ (* greater *) -> intersection set1 t2)
#     end ;;
module MakeOrderedSet : functor (Elements : ORDERED_TYPE) -> SET

```

But this won't do. The returned module satisfies SET, but we've already seen how this is too strong a requirement. The solution is the same as before, use sharing constraints to allow access to the element type.

```

# module MakeOrderedSet (Elements : ORDERED_TYPE)
#   : (SET with type element = Elements.t) =
# struct
#   type element = Elements.t
#   type set = element list
#   let empty = []
#   let is_empty s = (s = [])
#   let rec member elt s =
#     match s with
#     | [] -> false
#     | hd :: tl ->
#       (match Elements.compare elt hd with
#         | 0 (* equal *) -> true
#         | -1 (* less *) -> false
#         | _ (* greater *) -> member elt tl)
#   let rec add elt s =
#     match s with
#     | [] -> [elt]
#     | hd :: tl ->
#       (match Elements.compare elt hd with
#         | 0 (* equal *) -> s
#         | -1 (* less *) -> elt :: s
#         | _ (* greater *) -> hd :: add elt tl)
#   let union = List.fold_right add
#   let rec intersection set1 set2 =
#     match set1, set2 with
#     | [], _ -> []
#     | _, [] -> []
#     | h1::t1, h2::t2 ->
#       (match Elements.compare h1 h2 with
#         | 0 (* equal *) -> h1 :: intersection t1 t2
#         | -1 (* less *) -> intersection t1 set2
#         | _ (* greater *) -> intersection set1 t2)
#   end ;;
module MakeOrderedSet :
  functor (Elements : ORDERED_TYPE) ->
    sig

```

```

type set
type element = Elements.t
val empty : set
val is_empty : set -> bool
val add : element -> set -> set
val union : set -> set -> set
val intersection : set -> set -> set
val member : element -> set -> bool
end

```

Here we finally have a functor that can generate a set module for any type. Let's generate a few, starting with a string set module, which we can generate by applying the `MakeOrderedSet` functor to a module satisfying `ORDERED_TYPE` linking the `string` type to an appropriate ordering function (here, the default `Stdlib.compare` function).

```

# module StringSet = MakeOrderedSet
#           (struct
#             type t = string
#             let compare = compare
#           end) ;;
module StringSet :
sig
  type set
  type element = string
  val empty : set
  val is_empty : set -> bool
  val add : element -> set -> set
  val union : set -> set -> set
  val intersection : set -> set -> set
  val member : element -> set -> bool
end

```

It works as expected:

```

# let s =
#   let open StringSet in
#     empty
#   |> add "first"
#   |> add "second"
#   |> add "third" ;;
val s : StringSet.set = <abstr>
# StringSet.union s s ;;
- : StringSet.set = <abstr>
# StringSet.member "a" s ;;
- : bool = false

```

How about an integer set module? Again, a couple of lines of code suffice.

```

# module IntSet = MakeOrderedSet
#           (struct
#             type t = int
#             let compare = compare
#           end) ;;

```

```

#           end) ;;
module IntSet :
sig
  type set
  type element = int
  val empty : set
  val is_empty : set -> bool
  val add : element -> set -> set
  val union : set -> set -> set
  val intersection : set -> set -> set
  val member : element -> set -> bool
end
# let s =
#   let open IntSet in
#     empty
#   |> add 1
#   |> add 2
#   |> add 3 ;;
val s : IntSet.set = <abstr>
# IntSet.union s s ;;
- : IntSet.set = <abstr>
# IntSet.member 4 s ;;
- : bool = false

```

## 12.6 A dictionary module

The query evaluation application we've been working on (remember that?) required not only an implementation of a set ADT, but also a dictionary ADT. Dictionaries are data structures that associate keys to values, and allow for insertion and deletion of key-value associations, and looking up of the value associated with a given key (if one exists).

We now have all the tools to build that as well. An appropriate signature for a dictionary is

```

# module type DICT =
#   sig
#     type key
#     type value
#     type dict
#
#     (* empty -- An empty dictionary *)
#     val empty : dict
#     (* lookup dict key -- Returns as an option the value
#        associated with the provided key. If the key is
#        not in the dictionary, returns None. *)
#     val lookup : dict -> key -> value option
#     (* member dict key -- Returns true if and only if the
#        key is in the dictionary. *)
#     val member : dict -> key -> bool
#     (* insert dict key value -- Inserts a key-value pair into
#        dict. If the key is already present, updates the key to
#        have the new value. *)

```

```

#     val insert : dict -> key -> value -> dict
#     (* remove dict key -- Removes the key and its value from the
#        dictionary, if present. If the key is not present,
#        returns the original dictionary. *)
#     val remove : dict -> key -> dict
#   end ;;
module type DICT =
sig
  type key
  type value
  type dict
  val empty : dict
  val lookup : dict -> key -> value option
  val member : dict -> key -> bool
  val insert : dict -> key -> value -> dict
  val remove : dict -> key -> dict
end

```

We'll want a functor that builds dictionaries for all kinds of keys and values. In order to make sure we can compare the keys properly, including ordering them, we'll need a comparison function for keys as well. While we're at it, we might as well use a nicer convention for the comparison function, which will return a value of type

```
type order = Less | Equal | Greater ;;
```

The argument to the functor should thus satisfy the following signature:

```

# module type DICT_ARG =
#   sig
#     type key
#     type value
#     (* We need to reveal the order type so users of the
#        module can match against it to implement compare *)
#     type order = Less | Equal | Greater
#     (* Comparison function on keys compares two elements
#        and returns their order *)
#     val compare : key -> key -> order
#   end ;;
module type DICT_ARG =
sig
  type key
  type value
  type order = Less | Equal | Greater
  val compare : key -> key -> order
end

```

An implementation of such a functor is given here. It takes a module *D* satisfying *DICT\_ARG*, providing all the needed information about the key and value types and the ordering of keys. It allows access to the key and value types via sharing constraints, so users of modules generated

by the functor can provide values of those types. This particular implementation of dictionaries is a simple list of key-value pairs, sorted by unique keys.

```

# module MakeOrderedDict (D : DICT_ARG)
#           : (DICT with type key = D.key
#                           and type value = D.value) =
# struct
#   type key = D.key
#   type value = D.value
#
#   (* Invariant: sorted by key, no duplicate keys *)
#   type dict = (key * value) list
#
#   let empty = []
#
#   let rec lookup d k =
#     match d with
#     | [] -> None
#     | (k1, v1) :: d1 ->
#       let open D in
#         match compare k k1 with
#         | Equal -> Some v1
#         | Greater -> lookup d1 k
#         | Less -> None
#
#   let member d k =
#     match lookup d k with
#     | None -> false
#     | Some _ -> true
#
#   let rec insert d k v =
#     match d with
#     | [] -> [k, v]
#     | (k1, v1) :: d1 ->
#       let open D in
#         match compare k k1 with
#         | Less -> (k, v) :: d
#         | Equal -> (k, v) :: d1
#         | Greater -> (k1, v1) :: (insert d1 k v)
#
#   let rec remove d k =
#     match d with
#     | [] -> []
#     | (k1, v1) :: d1 ->
#       let open D in
#         match compare k k1 with
#         | Equal -> d1
#         | Greater -> (k1, v1) :: (remove d1 k)
#         | Less -> d
#   end ;;
module MakeOrderedDict :
  functor (D : DICT_ARG) ->
    sig
```

```

type key = D.key
type value = D.value
type dict
val empty : dict
val lookup : dict -> key -> value option
val member : dict -> key -> bool
val insert : dict -> key -> value -> dict
val remove : dict -> key -> dict
end

```

A reverse index, recall, is just a dictionary for mapping `string` keys to string set values. (The latter we've already built as the type `StringSet.set`.) Let's build one using the `MakeOrderedDict` functor.

The argument to the functor should specify the key and value types and the ordering on keys:

```

# module StringStringSetDictArg
#   : (DICT_ARG with type key = string
#                 and type value = StringSet.set) =
# struct
#   type key = string
#   type value = StringSet.set
#   type order = Less | Equal | Greater
#   let compare x y = if x < y then Less
#                     else if x = y then Equal
#                     else Greater
# end ;;
module StringStringSetDictArg :
sig
  type key = string
  type value = StringSet.set
  type order = Less | Equal | Greater
  val compare : key -> key -> order
end

```

Now to generate an index module requires only a single line.

```

# module Index = MakeOrderedDict (StringStringSetDictArg) ;;
module Index :
sig
  type key = StringStringSetDictArg.key
  type value = StringStringSetDictArg.value
  type dict = MakeOrderedDict(StringStringSetDictArg).dict
  val empty : dict
  val lookup : dict -> key -> value option
  val member : dict -> key -> bool
  val insert : dict -> key -> value -> dict
  val remove : dict -> key -> dict
end

```

By making use of these generic constructs for sets and dictionaries, we can build a reverse index type easily, and implement query evaluation in a manner that is oblivious to, hence robust to any changes in,

the implementation of the sets and dictionaries. The code for eval can be as specified before, and repeated here.

```
# let rec eval (q : query)
#           (idx : Index.dict)
#           : StringSet.set =
# match q with
# | Word word -> (match Index.lookup idx word with
#                   | None -> StringSet.empty
#                   | Some v -> v)
# | And (q1, q2) -> StringSet.intersection (eval q1 idx)
#                     (eval q2 idx)
# | Or (q1, q2) -> StringSet.union (eval q1 idx)
#                     (eval q2 idx) ;;
val eval : query -> Index.dict -> StringSet.set = <fun>
```

More generally, modules allow separating an interface from its implementation, the key premise of abstract data types and modular programming, and OCaml's functors provide for constructing modules that operate generically.

## 12.7 Alternative methods for defining signatures and modules

We've already seen two ways to define a module subject to a particular signature. First is to name the signature explicitly using `module type`, and use that name in defining the module itself.

```
module type SIG_NAME =
sig
  ...component declarations...
end ;;

module ModuleName : SIG_NAME =
struct
  ...component implementations...
end ;;
```

Second is to place an unnamed signature directly constraining the module definition

```
module ModuleName : sig
  ...component declarations...
end =
struct
  ...component implementations...
end ;;
```

useful on occasions where the signature is quite short and will only be used once, so retaining a name for it isn't needed.

There is a third method, widely used within OCaml's own implementation of library modules. All of the components defined in a .ml

file *automatically* constitute a module, whose name is generated by converting the first letter of the filename to uppercase. For example, if we have a file named `queue.ml` whose contents is

```
type 'a queue = 'a list
let empty_queue : 'a queue = []
let enqueue (elt : 'a) (q : 'a queue) : 'a queue =
  q @ [elt]
let dequeue (q : 'a queue) : 'a * 'a queue =
  match q with
  | [] -> raise (Invalid_argument
                  "dequeue: empty queue")
  | hd :: tl -> hd, tl
```

then we can refer in other files to `Queue.queue` to gain access to the type defined in that file, to `Queue.enqueue` to access the enqueueing function, and so forth. We can even place an open `Queue` at the top of another file to have unprefixed access to the components of the module.

How to define a signature for such a module though? OCaml looks for a file with the same prefix but the extension `.mli` (the `i` is for “interface”), which holds the component declarations for the signature. Thus, we should place in a file `queue.mli` these declarations:

```
type 'a queue
val empty_queue : 'a queue
val enqueue : 'a -> 'a queue -> 'a queue
val dequeue : 'a queue -> 'a * 'a queue
```

The `Queue` module will then be constrained by this signature, simply by virtue of the matching filenames.

### 12.7.1 Set and dictionary modules

The facilities for generating set modules – including the `SET` signature and `MakeOrderedSet` functor – might well be packaged up into a single module themselves. A file `set.ml` providing such a module might look like the following:

```
(* A Set Module *)

(*.....
  Set interface
*)

module type SET =
sig
  type element (* elements of the set *)
  type set      (* sets formed from the elements *)

(* The empty set *)
```

```

val empty : set
(* Returns true if set is empty; false otherwise *)
val is_empty : set -> bool
(* Adds element to existing set (if not already a member) *)
val add : element -> set -> set
(* Union of two sets *)
val union : set -> set -> set
(* Intersection of two sets *)
val intersection : set -> set -> set
(* Returns true iff element is in set *)
val member : element -> set -> bool
end ;;

(*.....*
   An implementation for elements of ordered type
*)

(* Module for types with a comparison function *)

module type COMPARABLE =
sig
  (* The type of comparable elements *)
  type t
  (* We need to reveal the order type so users of the
     module can match against it *)
  type order = Less | Equal | Greater
  (* Comparison function compares two elements of the
     type and returns their order *)
  val compare : t -> t -> order
end

(* Functor that generates sets for any comparable type *)

module MakeOrderedSet (Elements : COMPARABLE)
  : (SET with type element = Elements.t) =
(* Implementation of SET as list of elements. Assumes
   list is sorted with no duplicates. *)
struct
  type element = Elements.t
  type set = element list

  let empty = []
  let is_empty s = (s = [])
  let rec member elt s =
    match s with
    | [] -> false
    | hd :: tl ->
      let open Elements in
      (* so that Elements.compare, Elements.Less,
         etc. are in scope *)
      match compare elt hd with
      | Equal -> true
      | Less -> false
      | Greater -> member elt tl
end

```

```

let rec add elt s =
  (* add the elt in the proper place in the
     ordered list *)
  match s with
  | [] -> [elt]
  | hd :: tl ->
    let open Elements in
    match compare elt hd with
    | Less -> elt :: s
    | Equal -> s
    | Greater -> hd :: add elt tl

let union s1 s2 = List.fold_right add s1 s2

let rec intersection xs ys =
  match xs, ys with
  | [], _ -> []
  | _, [] -> []
  | xh :: xt, yh :: yt ->
    let open Elements in
    match compare xh yh with
    | Equal -> xh :: intersection xt yt
    | Less -> intersection xt ys
    | Greater -> intersection xs yt
  end ;;

```

This file defines a module called `set` that enables usage like the following, to define and use a `StringSet` module:

```

module StringSet =
  let open Set in
  MakeOrderedSet
  (struct
    type t = string
    type order = Less | Equal | Greater
    let compare s t = if s < t then Less
                     else if s = t then Equal
                     else Greater
  end) ;;

let s = StringSet.create
  |> StringSet.add "a"
  |> StringSet.add "b"
  |> StringSet.add "a" ;;

```

## 12.8 Library Modules

Data structures like sets and dictionaries are so generally useful that you might think the language ought to provide them so that each individual programmer doesn't need to implement them. In fact, OCaml *does* provide these and many other data structures – as LIBRARY MODULES.

In particular, the [Set library module](#) provides functionality much like the Set module in the previous section, and the [Map library module](#) provides functionality much like our dictionary module and its `MakeOrderedDict` functor.

In later chapters, we'll happily avail ourselves of these built-in libraries. Nonetheless, it's still important to see how such simple and general abstract data structures can be provided as modules, for several reasons: to demystify what's going on in the library-provided modules, to instantiate the idea that the language itself is sufficient for implementing these ideas, and as examples to inspire ways to implement other, more application-specific abstract data structures.

### 12.9 Problem section: Image manipulation

We define here a signature for modules that deal with images and their manipulation.

```
module type IMAGING =
  sig
    (* types for images, which are composed of pixels *)
    type image
    type pixel
    (* an image size is a pair of ints giving number of
       rows and columns *)
    type size = int * int
    (* converting between integers and pixels *)
    val to_pixel : int -> pixel
    val from_pixel : pixel -> int
    (* apply an image filter, a function over pixels,
       to every pixel in an image *)
    val filter : (pixel -> pixel) -> image -> image
    (* apply an image filter to two images, combining
       the images pixel by pixel *)
    val filter2 : (pixel -> pixel -> pixel)
                  -> image -> image -> image
    (* return a "constant" image of the specified size
       where every pixel has the same value *)
    val const : pixel -> size -> image
    (* display the image in a graphics window *)
    val depict : image -> unit
  end ;;
```

The pixels that make up an image are specified by the following signature:

```
module type PIXEL =
  sig
    type t
    val to_pixel : int -> t
    val from_pixel : t -> int
  end
```

**Problem 103**

We'd like to implement a functor named `MakeImaging` for generating implementations of the `IMAGING` signature based on modules satisfying the `PIXEL` signature. How should such a functor start? Give the header line of such beginning with the keyword `module` and ending with the `= struct....`

Here is a module implementing the `PIXEL` signature for integer pixels.

```
module IntPixel : (PIXEL with type t = int) =
  struct
    type t = int
    let to_pixel x = x
    let from_pixel x = x
  end ;;
```

**Problem 104**

Write code that uses the `IntPixel` module to define an imaging module called `IntImaging`.

**Problem 105**

Write code to use the `IntImaging` module that you defined in Problem 104 to display a 100 by 100 pixel image where all of the pixels have the constant integer value 5000.

### 12.10 Problem section: An abstract data type for intervals

A good candidate for an abstract data type is the `INTERVAL`. Abstractly speaking, an interval is a region between two points, where all that is required of points is that we be able to compare them as an ordering (so that we have a well-defined notion of “between”). That is, points ought to obey the following signature, which may look familiar, as you've seen it in other contexts:

```
module type COMPARABLE =
  sig
    type t
    type order = Less | Equal | Greater
    val compare : t -> t -> order
  end ;;
```

Intervals come up in many different contexts. As an informal example, calendars need to associate events with time intervals, such as 3-4pm or 11:30am-3:30pm; the endpoints in this case would be times. Natural operations over intervals include: the construction of an interval between two points, the extraction of the endpoints of an interval, taking the union of two intervals (the smallest interval containing both) or their intersection, and determining the relation between two intervals (whether they are disjoint, overlapping, or one contains the other). Here is a signature that provides for this functionality.

```
module type INTERVAL =
  sig
```

```

type point
type interval
type relation = Disjoint | Overlaps | Contains
(* Returns the interval between two points *)
val interval : point -> point -> interval
(* Returns the endpoints of an interval as a pair
   with the first point less than the second. *)
val endpoints : interval -> point * point
(* Returns the union of two intervals *)
val union : interval -> interval -> interval
(* Returns the relation holding between two intervals *)
val relation : interval -> interval -> relation
end ;

```

The possible relations between two intervals are depicted in Figure 12.3. (For the interval arithmetic cognoscenti, we've left out many details, such as whether intervals are open or closed; more fine-grained relations; and many other useful operations on intervals. These issues are beyond the scope of this problem.)

#### Problem 106

We'd like to have a functor named `MakeInterval` for generating implementations of the `INTERVAL` signature based on modules satisfying the `COMPARABLE` signature. How should such a functor start? Give the header line of such a functor definition beginning with the keyword `module` and ending with the `= struct....`

#### Problem 107

An appropriate module satisfying `COMPARABLE` for the purpose of generating *discrete* time intervals would be one where the type is `int`, with an appropriate comparison function. Define a module named `DiscreteTime` satisfying `COMPARABLE` where the type is `int`. Make sure the type is accessible outside the module.

#### Problem 108

Now use the functor `MakeInterval` to define a module `DiscreteTimeInterval` that provides interval functionality over discrete times as defined by the module `DiscreteTime` above.

#### Problem 109

The intersection of two intervals is only well-defined if the intervals are not disjoint. Assume that the `DiscreteTimeInterval` module has been opened, allowing you to make use of everything in its signature. Now, define a function `intersection : interval -> interval -> interval option` that takes two intervals and returns `None` if they are disjoint and otherwise returns their intersection (embedded appropriately in the option type).

#### Problem 110

Provide three different unit tests that would be useful in testing the correctness of the `DiscreteTimeInterval` module.

### 12.11 Problem section: Mobiles

The artist Alexander Calder (1898–1976) is well known for his distinctive mobiles, sculptures with different shaped objects hung from a cascade of connecting metal bars. An example is given in Figure 12.4.

His mobiles are made with varying shapes at the ends of the connectors – circles, ovals, fins. The exquisite balance of the mobiles

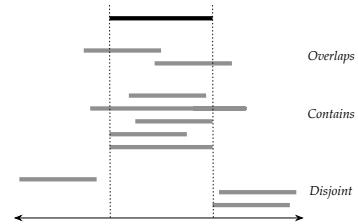


Figure 12.3: A diagrammatic depiction of the possible relations holding between two intervals. In the diagram, the gray intervals in the three groups below the black interval are in the “overlaps” (top 2), “contains” (next 5), and “disjoint” (bottom 3) relations, respectively, with the black interval at top. The vertical dotted lines depict the endpoints of the black interval.

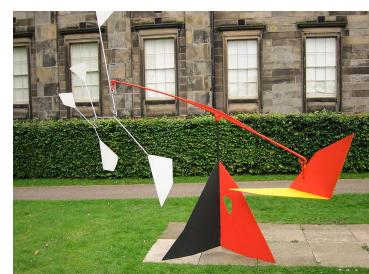


Figure 12.4: Alexander Calder's *Lempennage* (1953).

depends on the weights of the various components. In the next few exercises of this problem, you will model the structure of mobiles as binary trees such that one can determine if a Calder-like mobile design is balanced or not. Let's start with the objects at the ends of the connectors. For our purposes, the important properties of an object will be its shape and its weight (in arbitrary units; you can interpret them as pounds).

#### Problem 111

Define a weight type consisting of a single floating point weight.

#### Problem 112

Define a shape type, a variant type that allows for three different shapes: circles, ovals, and fins.

#### Problem 113

Define an object type that will be used to store information about the objects at the ends of the connectors, in particular, their weight and their shape.

A mobile can be modeled as a kind of binary tree, where the leaves of the tree, representing the objects, are elements of type obj, and the internal nodes, representing the connectors, have a weight, and each internal node (connector) connects two submobiles. Rather than directly writing code for a mobile type, though, we'll digress to build a more general binary tree module, and then model mobiles using that. An appropriate signature BINTREE for a simple binary tree module might be the following:

```
module type BINTREE =
  sig
    type leaft (* the type for the leaves of the tree *)
    type nodet (* the type for the internal nodes of the tree *)
    type tree  (* the type for the trees themselves *)

    val make_leaf : leaft -> tree
    val make_node : nodet -> tree -> tree -> tree
    val walk : (leaft -> 'a)
              -> (nodet -> 'a -> 'a -> 'a) -> tree -> 'a
  end ;;
```

This module signature specifies separate types for the leaves of trees and the internal nodes of trees, along with a type for the trees themselves; functions for constructing leaf and node trees; and a single function to "walk" the tree. (We'll come back to the `walk` function later.) In addition to the signature for binary tree modules, we would need a way of generating implementations of modules satisfying the BINTREE signature, which we'll do with a functor `MakeBintree`. The `MakeBinTree` functor takes an argument module of type BINTREE\_ARG that packages up the particular types for the leaves and nodes, that is, the types to use for `leaft` and `nodet`. The following module signature will work:

```
module type BINTREE_ARG =
  sig
    type leaft
    type nodet
  end ;;
```

**Problem 114**

Write down the header of a definition of a functor named `MakeBintree` taking a `BINTREE_ARG` argument, which generates modules satisfying the `BINTREE` signature. Keep in mind the need for users of the functor-generated modules to access appropriate aspects of the generated trees. (You don't need to fill in the actual implementation of the functor.)

Using the `MakeBintree` functor described above, you can now generate a `Mobile` module, which has `objs` at the leaves and `weights` at the interior nodes.

**Problem 115**

Define a module `Mobile` using the functor `MakeBintree`.

**Problem 116**

You've just used the `MakeBintree` functor without ever seeing its implementation. Why is this possible?

You can now build a representation of a mobile using the functions that the `Mobile` module makes available.

**Problem 117**

Define a value `mobile1` of type `Mobile.tree` that represents a mobile structured as the one depicted in Figure 12.5.

The `walk` function, of type `(leaft -> 'a) -> (nodet -> 'a -> 'a -> tree -> 'a)`, is of special interest, since it is the sole method for performing computations over these binary trees. The function is a kind of fold that works over trees instead of lists. It takes two functions – one for leaves and one for nodes – and applies these functions to a tree to generate a single value. The leaf function takes a `leaft` and returns some value of type `'a`. The node function takes a `nodet`, as well as the two `'a` values recursively returned by walking its two subtrees, and computes the value for the node itself. For example, we can use `walk` to define a function `size` that counts how many objects there are in a mobile. The function uses the fact that leaves are of size 1 and the size of a non-leaf is the sum of the sizes of its subtrees.

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

**Problem 118**

What is the type of `size`?

**Problem 119**

Use the fact that the `walk` function is curried to give a slightly more concise definition for `size`.



Figure 12.5: A simple Calder-style mobile. The depicted mobile has two connectors and three objects (an oval and two fins). The connectors each weigh 1.0, and the objects' weights are as given in the figure.

**Problem 120**

Use the `walk` function to implement a function `shape_count : shape -> Mobile.tree -> int` that takes a shape and a mobile (in that order), and returns the number of objects in the mobile that have that particular shape.

A mobile is said to be *balanced* if every connector has the property that the total weight of all components (that is, objects and connectors) of its left submobile is the same as the total weight of all components of its right submobile. (In actuality, we'd have to worry about other things like the relative lengths of the arms of the connectors, but we'll ignore all that.)

**Problem 121**

Is the mobile shown balanced? Why or why not?

**Problem 122**

Implement a function `balance : Mobile.tree -> weight option` that takes a mobile, and returns `None` if the argument mobile is not balanced, and `Some w` if the mobile is balanced, where `w` is the total weight of the mobile.

### 12.12 Supplementary material

- Lab 7: Modules and abstract data types
- Lab 8: Functors
- Problem set A.5: Ordered collections

# 13

## *Semantics: The substitution model*

We've introduced a broad swath of OCaml, describing both the syntax of different constructions and their use in constructing programs. But *why* the expressions of OCaml actually have the meanings they have has been dealt with only informally.

Semantics is about what expressions *mean*. As described so far, asking what an OCaml expression means is tantamount to asking what it evaluates to, what value it "means the same" as. Before getting into the details, however, it bears considering why a *formal, rigorous, precise* semantics of a programming language is even useful. Why not stick to the informal discussion of what the constructs of a programming language do? After all, such informal discussions, written in a natural language (like English), seem to work just fine for reference manuals and training videos.

There are three reasons that formalizing a semantics with mathematical rigor is beneficial.

*Mental hygiene* Programming is used to communicate our computational intentions to others. But what *exactly* is being communicated? Without a precise meaning to the expressions of the programming language, there is room for miscommunication from program author to reader.

*Interpreters* Computers generate computation by interpreting the expressions of the programming language. Developers of interpreters (or compilers) for a programming language implement their understanding of the meaning of the constructs of the programming language. Without a precise meaning to the expressions of the programming language, two interpreters might generate different computations for the same expression, even though both were written in good faith efforts to manifest the interpreter developers' understandings of the language.

*Metaprogramming* Programs that operate over expressions of the

programming language – such as programs to verify correctness of a program or transform it for efficiency or analyze it for errors – must use a precise notion of the meanings of those expressions.

For these reasons, we introduce in this chapter a technique for giving a semantics to some small subsets of OCaml. We continue this exercise in Chapter 19. The final project described in Chapter A – the implementation of a small subset of OCaml – relies heavily on the discussion in these two chapters.

As noted, we'll cash out the meaning of an expression by generating a simpler expression that “means the same”. In essence, this is the notion of evaluation that we've seen before. In this chapter we'll introduce a first method for providing a *rigorous* semantics of a programming language, based on the substitution of subexpressions, substituting for particular expressions expressions that “mean the same” but that are simpler.

The underlying conception of substitution as the basis for semantics dates from 1677 in Gottfried Leibniz's statement of the *identity of indiscernibles*:

That *A* is the same as *B* signifies that the one can be substituted for the other, *salva veritate*, in any proposition whatever.

*Salva veritate* – preserving the truth. Leibniz claims that substituting one expression with another that means the same thing preserves the truth of expressions.

We'll see later (Chapters 15 and 16) that a naive interpretation of Leibniz's law isn't sustainable. In particular, in the presence of state and state change, the province of imperative programming, the law seems to fail. But for the portion of OCaml we've seen so far, Leibniz's statement works quite well.

Following Leibniz's view, in this chapter we provide a semantics for a language that can be viewed as a (simple and untyped) subset of OCaml, with constructs like arithmetic and boolean operators, conditionals, functions (including recursive functions), and local naming.

We provide these semantic notions in two ways: as formal rule systems that define the evaluation relation, and as computer programs to evaluate expressions to their values.

The particular method of providing formal semantics that we introduce in this chapter is called *large-step operational semantics* and is based on the NATURAL SEMANTICS method of computer scientist Gilles Kahn (Figure 13.2).

The semantics we provide is FORMAL in the sense that the semantic rules rely only on manipulations based on the *forms* of the notations



Figure 13.1: Gottfried Wilhelm Leibniz (1646–1716), German philosopher, (co-)inventor of the differential and integral calculus, and philosopher. His law of the identity of indiscernibles underlies substitution semantics.



Figure 13.2: Gilles Kahn (1946–2006), French computer scientist, developer of the natural semantics approach to programming language semantics. Kahn was president of the French research institute INRIA, where OCaml was developed.

we introduce. The semantics we provide is an **OPERATIONAL SEMANTICS** because we provide a formal specification of what programs *evaluate to*, rather than what they *denote*.<sup>1</sup> The semantics we provide is a **LARGE-STEP** semantics because it characterizes directly the relation between expressions and what they (eventually, after perhaps many individual small steps) evaluate to, rather than characterizing the relation between expressions and what they lead to after each individual small step. (That would be a **SMALL-STEP SEMANTICS**.) Notationally, we characterize this relation between an expression  $P$  and the value  $v$  it evaluates to with an evaluation JUDGEMENT notated  $P \Downarrow v$ , which can be read as “the expression  $P$  evaluates to the value  $v$ ”.

<sup>1</sup> The primary alternative method of providing a formal semantics is **DENOTATIONAL SEMANTICS**, which addresses exactly this issue of what expressions denote.

### 13.1 Semantics of arithmetic expressions

Recall the language of arithmetic expressions from Section 11.4. We start by augmenting that language with a local naming construct, the `let ⟨var⟩ = <expr> in <expr>`. We’ll express the abstract syntax of the language using the following BNF:

```

⟨binop⟩ ::= + | - | * | /
⟨var⟩ ::= x | y | z | ...
⟨expr⟩ ::= ⟨integer⟩
         | ⟨var⟩
         | ⟨expr⟩ ⟨binop⟩ ⟨expr⟩
         | let ⟨var⟩ = ⟨expr⟩ in ⟨expr⟩
    
```

#### Exercise 123

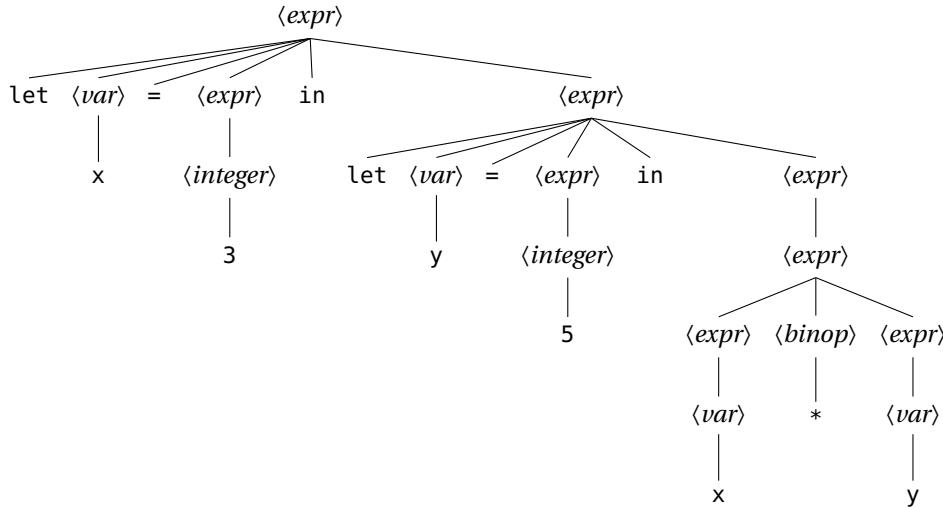
For brevity, we left off unary operators. Extend the grammar to add unary operators (negation, say).

With this grammar, we can express the abstract syntax of the concrete expression

```

let x = 3 in
let y = 5 in
x * y
    
```

as the tree



What rules shall we use for evaluating the expressions of the language? Recall that we write a judgement  $P \Downarrow v$  to mean that the expression  $P$  evaluates to the value  $v$ . The VALUES, the results of evaluation, are those expressions that evaluate to themselves. By convention, we'll use italic capitals like  $P$ ,  $Q$ , etc. to stand for arbitrary expressions, and  $v$  (possibly subscripted) to stand for expressions that are values. You should think of  $P$  and  $v$  as expressions structured as per the abstract syntax of the language – it is the abstract, structured expressions that have well-defined meanings by the rules we'll provide – though we notate them using the concrete syntax of OCaml, since we need some linear notation for specifying them.

Certain cases are especially simple. Numeric literal expressions like 3 or 5 are already as simplified as they can be. They evaluate to themselves; they are values. We could enumerate a plethora of judgements that express this self-evaluation, like

$$1 \Downarrow 1$$

$$2 \Downarrow 2$$

$$3 \Downarrow 3$$

$$4 \Downarrow 4$$

$$5 \Downarrow 5$$

...

but we'd need an awful lot of them. Instead, we'll just use a schematic rule for capturing permissible judgements:

$$\bar{n} \Downarrow \bar{n} \quad (R_{int})$$

Here, we use a schematic variable  $n$  to stand for any integer, and use the notation  $\bar{n}$  for the OCaml numeral expression that encodes the number  $n$ .

Using this schematic rule notation we can provide general rules for evaluating other arithmetic expressions. To evaluate an expression of the form  $P + Q$ , where  $P$  and  $Q$  are two subexpressions, we first need to know what values  $P$  and  $Q$  evaluate to; since they will be numeric values, we can take them to be  $\bar{m}$  and  $\bar{n}$ , respectively. Then the value that  $P + Q$  evaluates to will be  $\bar{m+n}$ . We'll write the rule as follows:

$$\begin{array}{c} P + Q \Downarrow \\ | \\ P \Downarrow \bar{m} \\ | \\ Q \Downarrow \bar{n} \\ \Downarrow \bar{m+n} \end{array} \quad (R_+)$$

In this rule notation, the first line is intended to indicate that we are evaluating  $P + Q$ , the blank space to the right of the  $\Downarrow$  indicating that some further evaluation judgements are required. Those are the two indented judgements provided to the right of the long vertical bar between the two occurrences of  $\Downarrow$ . The final line provides the value that the original expression evaluates to.

Thus, this rule can be glossed as “To evaluate an expression of the form  $P + Q$ , first evaluate  $P$  to an integer value  $\bar{m}$  and  $Q$  to an integer value  $\bar{n}$ . The value of the full expression is then the integer literal representing the sum of  $m$  and  $n$ .” The two subderivations for  $P \Downarrow \bar{m}$  and  $Q \Downarrow \bar{n}$  are derived independently, and not in any particular order.

Using these two rules, we can now show a particular evaluation, like that of the expression  $3 + 5$ :<sup>2</sup>

$$\begin{array}{c} 3 + 5 \Downarrow \\ | \\ 3 \Downarrow 3 \\ | \\ 5 \Downarrow 5 \\ \Downarrow 8 \end{array}$$

or the evaluation of  $3 + 5 + 7$ :

$$\begin{array}{c} 3 + 5 + 7 \Downarrow \\ | \\ 3 + 5 \Downarrow \\ | \\ 3 \Downarrow 3 \\ | \\ 5 \Downarrow 5 \\ \Downarrow 8 \\ | \\ 7 \Downarrow 7 \\ \Downarrow 15 \end{array}$$

<sup>2</sup> Wait, where did that 8 come from exactly? Since  $3 \equiv \bar{3}$  and  $5 \equiv \bar{5}$ , the rule  $R_{int}$  gives the result as  $\bar{3+5} \equiv \bar{8} \equiv 8$ .

**Exercise 124**

Why is the proof for the value of  $3 + 5 + 7$  not structured as

$$3 + 5 + 7 \Downarrow$$

$$\begin{array}{c} 3 \Downarrow 3 \\ 5 + 7 \Downarrow \\ | \\ 5 \Downarrow 5 \\ | \\ 7 \Downarrow 7 \\ \Downarrow 12 \\ \Downarrow 15 \quad ? \end{array}$$

We should have similar rules for other arithmetic operators. Here's a possible rule for division:

$$P / Q \Downarrow$$

$$\begin{array}{c} P \Downarrow \bar{m} \\ Q \Downarrow \bar{n} \\ \hline \Downarrow \lfloor m/n \rfloor \end{array} \quad (R/)$$

In this rule, we've used some standard mathematical notation in the final result:  $/$  for numeric division and  $\lfloor \rfloor$  for truncating a real number to an integer.

These rules for addition and division may look trivial, but they are not. The division rule specifies that the  $/$  operator in OCaml when applied to two numerals specifies the integer portion of their ratio. The language being specified might have been otherwise.<sup>3</sup> The language might have used a different operator (like  $//$ ) for integer division,

$$P // Q \Downarrow$$

$$\begin{array}{c} P \Downarrow \bar{m} \\ Q \Downarrow \bar{n} \\ \hline \Downarrow \lfloor m/n \rfloor \end{array}$$

(as happens to be used in Python 3 for instance). The example should make clear the distinction between the OBJECT LANGUAGE whose semantics is being defined and the METALANGUAGE being used to define it.

Similarly, the rule could have defined the result differently, say

$$P / Q \Downarrow$$

$$\begin{array}{c} P \Downarrow \bar{m} \\ Q \Downarrow \bar{n} \\ , \\ \hline \Downarrow \lceil m/n \rceil \end{array}$$

which specifies that the result of the division is the integer resulting from rounding up, rather than down.

Nonetheless, there is not *too* much work being done by these rules, and if that were all there were to defining a semantics, there would be

<sup>3</sup> What may be mind-boggling here is the role of the mathematical notation used in the result part of the rule. How is it that we can make use of notations like  $\lfloor m/n \rfloor$  in defining the semantics of the  $/$  operator? Doesn't appeal to that kind of mathematical notation beg the question? Or at least call for its own semantics? Yes, it does, but since we have to write down the semantics of constructs *somewhat* or other, we use commonly accepted mathematical notation applied in the context of natural language (in the case at hand, English). You may think that this merely postpones the problem of giving OCaml semantics by reducing it to the problem of giving semantics for mathematical notation and English. You would be right, and the problem is further exacerbated when the semantics makes use of mathematical notation that is not so familiar, for instance, the substitution notation to be introduced shortly. But we have to start somewhere.

little reason to go to the trouble. Things get more interesting, however, when additional constructs such as local naming are considered, which we turn to next.

#### Exercise 125

Write evaluation rules for the other binary operators and the unary operators you added in Exercise 123.

### 13.2 Semantics of local naming

The  $\langle \text{expr} \rangle$  language defined in the grammar above includes a local naming construct, whose concrete syntax is expressed with `let ⟨⟩ in ⟨⟩`. What is the semantics of such an expression? It is here that substitution starts to play a critical role. We will take the meaning of this local naming construct to work by *substituting the value of the definition for occurrences of the variable in the body*. More precisely, we use the following evaluation rule:

$$\begin{array}{c} \text{let } x = D \text{ in } B \Downarrow \\ \left| \begin{array}{l} D \Downarrow v_D \\ B[x \mapsto v_D] \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{\text{let}})$$

We've introduced a new notation –  $Q[x \mapsto P]$  – for substituting the expression  $P$  for occurrences of the variable  $x$  in the expression  $Q$ . For instance,

$$(x * x)[x \mapsto 5] = 5 * 5$$

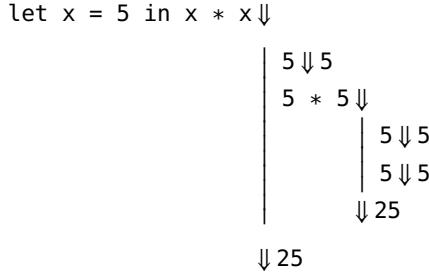
that is, substituting 5 for  $x$  in the expression  $x * x$  yields  $5 * 5$ . (It doesn't yield 25 though. That would require a further evaluation, which is what the part of the rule  $B[x \mapsto v_D] \Downarrow v_B$  does.)

The evaluation rule  $R_{\text{let}}$  can be glossed as follows: “To evaluate an expression of the form `let x = D in B`, first evaluate the expression  $D$  to a value  $v_D$  and evaluate the result of substituting  $v_D$  for occurrences of  $x$  in the expression  $B$  to a value  $v_B$ . The value of the full expression is then  $v_B$ .”

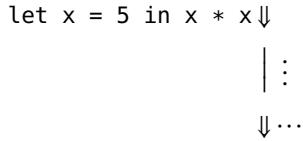
Using this rule (and the others), we can now show

$$\text{let } x = 5 \text{ in } x * x \Downarrow 25$$

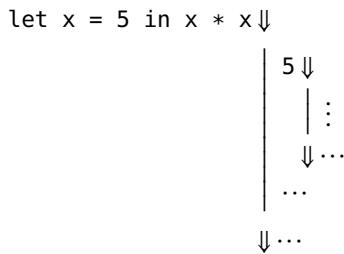
as per the following derivation:



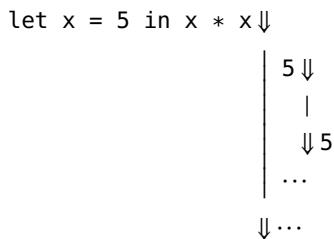
Let's put this first derivation together step by step so the steps are clear. We want a derivation that demonstrates what `let x = 5 in x * x` evaluates to. It will be of the form



This pattern matches rule  $R_{let}$ , where  $x$  plays the role of the schematic variable  $x$ ,  $5$  plays the role of the schematic expression  $D$ , and  $x * x$  plays the role of  $B$ . We will plug these into the two subderivations required. First is the subderivation evaluating  $D$  (that is,  $5$ ):



This subderivation can be completed using the  $R_{int}$  rule, which requires no subderivations itself.



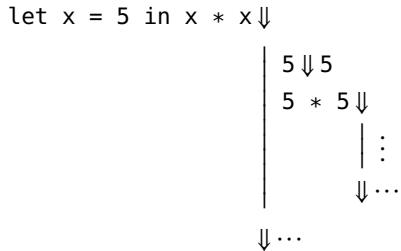
Thus, the result of this subderivation,  $v_D$  is  $5$ .

Second is the subderivation for evaluating  $B[x \mapsto v_D]$  to its value  $v_B$ .

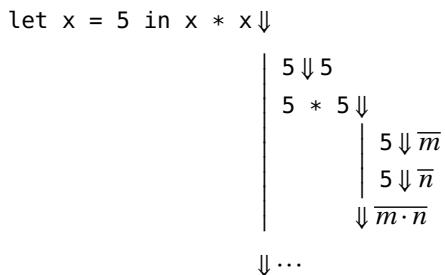
Now

$$\begin{aligned}
 B[x \mapsto v_D] &= (x * x)[x \mapsto 5] \\
 &= x[x \mapsto 5] * x[x \mapsto 5] \\
 &= 5 * 5
 \end{aligned}$$

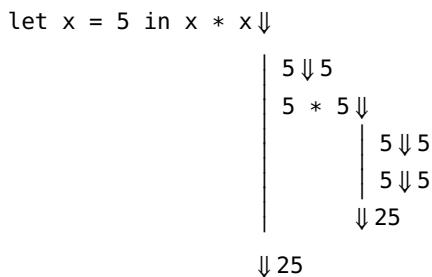
(We'll define this substitution operation carefully in Section 13.3.) So the second subderivation must evaluate the expression  $5 * 5$ :



This second subderivation matches a rule  $R_*$  analogous to  $R_+$ . (You would have written it in Exercise 125.) Here, 5 plays the role of both  $P$  and  $Q$ :



Now, the subderivations of the  $5 * 5$  subderivation both evaluate to 5. We use the  $R_{int}$  rule twice, with 5 for both  $\overline{m}$  and  $\overline{n}$ , so  $m$  and  $n$  are both 5, and  $\overline{m} \cdot \overline{n}$  is 25. The result for the original expression as a whole is therefore also 25.



#### Exercise 126

Carry out derivations for the following expressions:

1. let  $x = 3$  in let  $y = 5$  in  $x * y$

```

2. let x = 3 in let y = x in x * y
3. let x = 3 in let x = 5 in x * y
4. let x = 3 in let x = x in x * x
5. let x = 3 in let x = y in x * x

```

Are the values for these expressions according to the semantics consistent with how OCaml evaluates them?

### 13.3 Defining substitution

Because of the central place of substitution in providing the semantics of the language, this approach to semantics is referred to as a SUBSTITUTION SEMANTICS.

Some care is needed in precisely defining this substitution operation. A start (which we'll see in Section 13.3.2 isn't fully correct) is given by the following recursive **equational definition**:<sup>4</sup>

$$\begin{aligned}
\overline{m}[x \mapsto Q] &= \overline{m} \\
x[x \mapsto Q] &= Q \\
y[x \mapsto Q] &= y \quad \text{where } x \neq y \\
(P + R)[x \mapsto Q] &= P[x \mapsto Q] + R[x \mapsto Q]
\end{aligned}$$

and similarly for other binary operators

$$(\text{let } y = D \text{ in } B)[x \mapsto Q] = \text{let } y = D[x \mapsto Q] \text{ in } B[x \mapsto Q]$$

<sup>4</sup> The  $\equiv$  operator here is intended to indicate syntactic identity, that is, that its arguments are the same (syntactic) expression. Thus,  $x \neq y$  specifies that the two variables notated  $x$  and  $y$  are not two occurrences of the same variable.

#### Exercise 127

Verify using this definition for substitution the derivation above showing that  $(x * x)[x \mapsto 5] = 5 * 5$ .

#### 13.3.1 A problem with variable scope

You may have noticed in Exercise 126 that some care must be taken when substituting. Consider the following case:

```
let x = 3 in let x = 5 in x
```

Intuitively, given the scope rules of OCaml described informally in Section 5.5, this expression should evaluate to 5, since the final occurrence of  $x$  is bound by the inner `let` (defined to be 5), not the outer one.

However, if we're not careful, we'll get a derivation like this:

```
let x = 3 in let x = 5 in x
  ↓
  3 ↓ 3
  |   ↓
  |   let x = 5 in 3
  |   |   ↓
  |   |   5 ↓ 5
  |   |   |   ↓
  |   |   |   3 ↓ 3
  |   |   |   ↓
  |   |   |   3
  ↓ 3
```

The highlighted expression is supposed to be the result of replacing  $x$  with its value 3 in the body of the definition `let x = 5 in x`, that is,

$$(\text{let } x = 5 \text{ in } x)[x \mapsto 3] .$$

Using the equational definition given above, we have

$$\begin{aligned} & (\text{let } x = 5 \text{ in } x)[x \mapsto 3] \\ &= \text{let } x = 5[x \mapsto 3] \text{ in } x[x \mapsto 3] \\ &= \text{let } x = 5 \text{ in } x[x \mapsto 3] \\ &= \text{let } x = 5 \text{ in } 3 . \end{aligned}$$

### 13.3.2 Free and bound occurrences of variables

It appears we must be very careful in how we define this substitution operation  $P[x \mapsto Q]$ . In particular, we don't want to replace *every* occurrence of the token  $x$  in  $P$ , only the *free* occurrences. The variable being introduced in a `let` should definitely not be replaced, nor should any occurrences of  $x$  within the body of a `let` that also introduces  $x$ .

A binding construct (a `let` or a `fun`) is said to **BIND** the variable that it introduces. A variable occurrence is said to be **BOUND** if it falls within the scope of a construct that binds that variable. Thus, in the expressions `fun x -> [x] + y` or `let x = 3 in [x] + y`, the highlighted occurrences of  $x$  are bound occurrences, bound by the `fun` or `let`, respectively, in the expressions.

A variable occurrence is said to be **FREE** if it is not bound. Thus, in the expressions `fun x -> x + [y]` or `let x = 3 in x + [y]`, the occurrences of  $y$  are free occurrences.

#### Exercise 128

In the following expressions, draw a line connecting each bound variable to the binding construct that binds it. Then circle all of the free occurrences of variables.

1.  $x$
2.  $x + y$

```

3. let x = 3 in x
4. let f = f 3 in x + y
5. (fun x -> x + x) x
6. fun x -> let x = y in x + 3

```

We can define the set<sup>5</sup> of FREE VARIABLES in an expression  $P$ , denoted  $FV(P)$ , through the recursive definition in Figure 13.3. By way of example, the definition says that the free variables in the expression  $\text{fun } y \rightarrow f(x + y)$  are just  $f$  and  $x$ , as shown in the following derivation:

$$\begin{aligned}
FV(\text{fun } y \rightarrow f(x + y)) &= FV(f(x + y)) - \{y\} \\
&= FV(f) \cup FV(x + y) - \{y\} \\
&= \{f\} \cup FV(x) \cup FV(y) - \{y\} \\
&= \{f\} \cup \{x\} \cup \{y\} - \{y\} \\
&= \{f, x, y\} - \{y\} \\
&= \{f, x\}
\end{aligned}$$

<sup>5</sup> For a review of the set notations that we use, see Section B.5.

### Exercise 129

Use the definition of  $FV$  to derive the set of free variables in the expressions below. Circle all of the free occurrences of the variables.

1. let x = 3 in let y = x in f x y
2. let x = x in let y = x in f x y
3. let x = y in let y = x in f x y
4. let x = fun y -> x in x

### Exercise 130

The definition of  $FV$  in Figure 13.3 is incomplete, in that it doesn't specify the free variables in a `let rec` expression. Add appropriate rules for this construct of the language, being careful to note that in an expression like `let rec x = fun y -> x in x`, the variable `x` is not free. (Compare with Exercise 129(4).)

#### 13.3.3 Handling variable scope properly

Now that we have formalized the idea of free and bound variables, it may be clearer what is going wrong in the previous substitution example. The substitution rule for substituting into a `let` expression

$$(\text{let } y = D \text{ in } B)[x \mapsto Q] = \text{let } y = D[x \mapsto Q] \text{ in } B[x \mapsto Q]$$

shouldn't apply when  $x$  and  $y$  are *the same* variable. In such a case, the occurrences of  $x$  in  $D$  or  $B$  are not free occurrences, but are bound by the `let`. We modify the definition of substitution accordingly:

$$\begin{aligned}
 \overline{m}[x \mapsto Q] &= \overline{m} \\
 x[x \mapsto Q] &= Q \\
 y[x \mapsto Q] &= y && \text{where } x \neq y \\
 (P + R)[x \mapsto Q] &= P[x \mapsto Q] + R[x \mapsto Q] && \text{and similarly for other binary operators} \\
 (\text{let } y = D \text{ in } B)[x \mapsto Q] &= \text{let } y = D[x \mapsto Q] \text{ in } B[x \mapsto Q] && \text{where } x \neq y \\
 (\text{let } x = D \text{ in } B)[x \mapsto Q] &= \text{let } x = D[x \mapsto Q] \text{ in } B
 \end{aligned}$$

**Exercise 131**

Use the definition of the substitution operation above to give the expressions (in concrete syntax) specified by the following substitutions:

1.  $(x + x)[x \mapsto 3]$
2.  $(x + x)[y \mapsto 3]$
3.  $(x * x)[x \mapsto 3 + 4]$
4.  $(\text{let } x = y \text{ in } y + x)[y \mapsto z]$
5.  $(\text{let } x = y \text{ in } y + x)[x \mapsto z]$

**Exercise 132**

Use the semantic rules developed so far (see Figure 13.5) to reduce the following expressions to their values. Show the derivations.

1.  $\text{let } x = 3 * 4 \text{ in } x + x$
2.  $\text{let } y = \text{let } x = 5 \text{ in } x + 1 \text{ in } y + 2$

### 13.4 Implementing a substitution semantics

Given a grammar and appropriate semantic evaluation rules and definitions for substitution, it turns out to be quite simple to implement the corresponding semantics, as a function that evaluates expressions to their values.

The grammar defining the abstract syntax of the language (repeated here for reference)

```

<binop> ::= + | - | * | /
<var> ::= x | y | z | ...
<expr> ::= <integer>
          | <var>
          | <expr1> <binop> <expr2>
          | let <var> = <exprdef> in <exprbody>
  
```

can be implemented, as we have done before (Section 11.4), with an algebraic type definition

```

# type binop = Plus | Divide ;;
type binop = Plus | Divide

# type varspec = string ;;
type varspec = string

# type expr =
#   | Int of int
#   | Var of varspec
#   | Binop of binop * expr * expr
#   | Let of varspec * expr * expr ;;
type expr =
  Int of int
  | Var of varspec
  | Binop of binop * expr * expr
  | Let of varspec * expr * expr

```

The `varspec` type specifies strings as a means to differentiate distinct variables. The `binop` type enumerates the various binary operators. (For brevity, in this example, we've only included two binary operators, for addition and division.) The `expr` type provides the alternative methods for building expressions recursively.

Then, the abstract syntax for the concrete expression

```

let x = 3 in
let y = 5 in
x / y

```

is captured by the OCaml expression

```

# Let ("x", Int 3,
#       Let ("y", Int 5,
#             Binop (Divide, Var "x", Var "y"))) ;;
- : expr =
Let ("x", Int 3, Let ("y", Int 5, Binop (Divide, Var "x", Var
"y")))

```

### Exercise 133

Augment the type definitions to allow for other binary operations (subtraction and multiplication, say) and for unary operations (negation).

#### 13.4.1 Implementing substitution

With a representation of expressions in hand, we can proceed to implement various useful functions over the expressions. Rather than provide implementations, we leave them as exercises.

### Exercise 134

Write a function `subst : expr -> varspec -> expr -> expr` that performs substitution, that is, `subst p x q` returns the expression that is the result of substituting `q` for the variable `x` in the expression `p`. For example,

```
# subst (Binop (Plus, Var "x", Var "y")) "x" (Int 3) ;;
- : expr = Binop (Plus, Int 3, Var "y")
# subst (Binop (Plus, Var "x", Var "y")) "y" (Int 3) ;;
- : expr = Binop (Plus, Var "x", Int 3)
# subst (Binop (Plus, Var "x", Var "y")) "z" (Int 3) ;;
- : expr = Binop (Plus, Var "x", Var "y")
```

### 13.4.2 Implementing evaluation

Now the semantics of the language – the evaluation of expressions to their values – can be implemented as a recursive function eval : expr → expr, which follows the evaluation rules just introduced. The type of the function indicates that the header line should be

```
let rec eval (exp : expr) : expr = ...
```

The computation proceeds based on the structure of exp, which might be any of the structures introducing the semantic rules. Consequently, we match on these structures:

```
let rec eval (exp : expr) : expr =
  match exp with
  | Int n -> ...
  | Var x -> ...
  | Binop (Plus, e1, e2) -> ...
  | Binop (Divide, e1, e2) -> ...
  | Let (var, def, body) -> ...
```

The computation for each of the cases mimics the computations in the evaluation rules exactly. Integers, for instance, are self-evaluating.

```
let rec eval (exp : expr) : expr =
  match exp with
  | Int n -> Int n
  | Var x -> ...
  | Binop (Plus, e1, e2) -> ...
  | Binop (Divide, e1, e2) -> ...
  | Let (var, def, body) -> ...
```

The second pattern concerns what should be done for evaluating *free* variables in expressions. (Presumably, any *bound* variables were substituted away by virtue of the final pattern-match.) We have provided no evaluation rule for free variables, and for good reason. Expressions with free variables, called OPEN EXPRESSIONS, don't have a value in and of themselves. Consequently, we can simply report an error upon evaluation of a free variable. We introduce an exception for this purpose.

```
let rec eval (exp : expr) : expr =
  match exp with
  | Int n -> Int n
  | Var x -> raise (UnboundVariable x)
  | Binop (Plus, e1, e2) -> ...
  | Binop (Divide, e1, e2) -> ...
  | Let (var, def, body) -> ...
```

The binary operator rules work by recursively evaluating the operands and applying an appropriate computation to the results.

```
let rec eval (exp : expr) : expr =
  match exp with
  | Int n -> Int n
  | Var x -> raise (UnboundVariable x)
  | Binop (Plus, e1, e2) ->
    let Int m = eval e1 in
    let Int n = eval e2 in
    Int (m + n)
  | Binop (Divide, e1, e2) ->
    let Int m = eval e1 in
    let Int n = eval e2 in
    Int (m / n)
  | Let (var, def, body) -> ...
```

Finally, the naming rule  $R_{let}$  performs substitution of the value of the definition in the body, and evaluates the result. We appeal to the function `subst` from Exercise 134.

```
# exception UnboundVariable of string;;
exception UnboundVariable of string

# let rec eval (exp : expr) : expr =
#   match exp with
#   | Int n -> Int n           (* R_int *)
#   | Var x -> raise (UnboundVariable x)
#   | Binop (Plus, e1, e2) ->      (* R_+ *)
#     let Int m = eval e1 in
#     let Int n = eval e2 in
#     Int (m + n)
#   | Binop (Divide, e1, e2) ->      (* R_- *)
#     let Int m = eval e1 in
#     let Int n = eval e2 in
#     Int (m / n)
#   | Let (var, def, body) ->        (* R_let *)
#     let def' = eval def in
#     eval (subst body var def') ;;
Lines 7-8, characters 0-11:
7 | let Int n = eval e2 in
8 | Int (m + n)
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Var _|Binop (_, _, _)|Let (_, _, _))
Lines 6-8, characters 0-11:
6 | let Int m = eval e1 in
7 | let Int n = eval e2 in
8 | Int (m + n)
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Var _|Binop (_, _, _)|Let (_, _, _))
Lines 11-12, characters 0-11:
11 | let Int n = eval e2 in
```

```

12 | Int (m / n)
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Var _|Binop (_, _, _)|Let (_, _, _))
Lines 10-12, characters 0-11:
10 | let Int m = eval e1 in
11 | let Int n = eval e2 in
12 | Int (m / n)
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Var _|Binop (_, _, _)|Let (_, _, _))
val eval : expr -> expr = <fun>

```

Two problems jump out: First, violating the edict of intention, we've not provided information about what to do in cases where the arguments to an integer operator evaluate to something other than integers. These show up as “pattern-matching not exhaustive” warnings. Second, violating the edict of irredundancy, the code for binary operators is quite redundant. We'll solve both problems simultaneously by factoring out the redundancy into a function for evaluating binary operator expressions. We'll introduce another exception for reporting ill-formed expressions.

```

# exception UnboundVariable of string ;;
exception UnboundVariable of string
# exception IllFormed of string ;;
exception IllFormed of string

# let binopeval (op : binop) (v1 : expr) (v2 : expr)
#           : expr =
#   match op, v1, v2 with
#   | Plus, Int x1, Int x2 -> Int (x1 + x2)
#   | Plus, _, _ ->
#     raise (IllFormed "can't add non-integers")
#   | Divide, Int x1, Int x2 -> Int (x1 / x2)
#   | Divide, _, _ ->
#     raise (IllFormed "can't divide non-integers") ;;
val binopeval : binop -> expr -> expr -> expr = <fun>

# let rec eval (e : expr) : expr =
#   match e with
#   | Int _ -> e
#   | Var x -> raise (UnboundVariable x)
#   | Binop (op, e1, e2) ->
#     binopeval op (eval e1) (eval e2)
#   | Let (x, def, body) ->
#     eval (subst body x (eval def)) ;;
val eval : expr -> expr = <fun>

```

This function allows evaluating expressions in the language reflecting the semantics of those expressions.

```

# eval (Binop (Plus, Int 5, Int 10)) ;;
- : expr = Int 15

```

```
# eval (Let ("x", Int 3,
#           Let ("y", Int 5,
#                 Binop (Divide, Var "x", Var "y")))) ;;
- : expr = Int 0
```

### 13.5 Problem section: Semantics of booleans and conditionals

#### Exercise 135

Augment the abstract syntax of the language to introduce boolean literals `true` and `false`. Add substitution semantics rules for the new constructs. Adjust the definitions of `subst` and `eval` to handle these new literals.

#### Exercise 136

Augment the abstract syntax of the language to add conditional expressions (`if ⟨⟩ then ⟨⟩ else ⟨⟩`). Add substitution semantics rules for the new construct. Adjust the definitions of `subst` and `eval` to handle conditionals.

### 13.6 Semantics of function application

We can extend our language further, by introducing (anonymous) functions and their application. We augment the language with two rules for function expressions and function applications as follows:

```
 $\langle binop \rangle ::= + | - | * | /$ 
 $\langle var \rangle ::= x | y | z | \dots$ 
 $\langle expr \rangle ::= \langle integer \rangle$ 
|  $\langle var \rangle$ 
|  $\langle expr_1 \rangle \langle binop \rangle \langle expr_2 \rangle$ 
|  $\text{let } \langle var \rangle = \langle expr_{def} \rangle \text{ in } \langle expr_{body} \rangle$ 
|  $\text{fun } \langle var \rangle \rightarrow \langle expr_{body} \rangle$ 
|  $\langle expr_{fun} \rangle \langle expr_{arg} \rangle$ 
```

To complete the semantics for this language, we simply have to add rules for the evaluation of functions and applications.

The case of functions is especially simple. Functions are pending computations; they don't take effect until they are applied. So we can take functions to be values, that is, they self-evaluate.

$$\text{fun } x \rightarrow B \Downarrow \text{fun } x \rightarrow B \quad (R_{\text{fun}})$$

All the work happens upon application. To evaluate an application, we must evaluate the function part to get the function to be applied and evaluate the argument part to get the argument's value, and then evaluate the body of the function, after substituting in the argument

for the variable bound by the function.

$$\begin{array}{c}
 P \ Q \Downarrow \\
 \left| \begin{array}{l} P \Downarrow \text{fun } x \rightarrow B \\ Q \Downarrow v_Q \\ B[x \mapsto v_Q] \Downarrow v_B \end{array} \right. \quad (R_{app}) \\
 \Downarrow v_B
 \end{array}$$

### Exercise 137

Give glosses for these two rules  $R_{fun}$  and  $R_{app}$ , as was done for the previous rules  $R_+$  and  $R_{let}$ .

Let's try an example:

`(fun x -> x + x) (3 * 4)`

Intuitively, this should evaluate to 24. The derivation proceeds as follows:

$$\begin{array}{c}
 (\text{fun } x \rightarrow x + x) (3 * 4) \\
 \Downarrow \\
 \left| \begin{array}{l} (\text{fun } x \rightarrow x + x) \Downarrow (\text{fun } x \rightarrow x + x) \\ 3 * 4 \Downarrow \\ \quad \left| \begin{array}{l} 3 \Downarrow 3 \\ 4 \Downarrow 4 \\ \Downarrow 12 \end{array} \right. \\ 12 + 12 \Downarrow \\ \quad \left| \begin{array}{l} 12 \Downarrow 12 \\ 12 \Downarrow 12 \\ \Downarrow 24 \end{array} \right. \end{array} \right. \\
 \Downarrow 24
 \end{array}$$

The combination of local naming and anonymous functions gives us the ability to give names to functions:

```
let double = fun x -> 2 * x in
double (double 3)
```

The derivations start getting a bit complicated:

```

let double = fun x -> 2 * x in double (double 3)
↓
fun x -> 2 * x ↓ fun x -> 2 * x
(fun x -> 2 * x) ((fun x -> 2 * x) 3)
↓
fun x -> 2 * x ↓ fun x -> 2 * x
(fun x -> 2 * x) 3
↓
fun x -> 2 * x ↓ fun x -> 2 * x
3 ↓ 3
2 * 3 ↓
   ↓ 2
   3 ↓ 3
   ↓ 6
   ↓ 6
2 * 6 ↓ 12
↓ 12
↓ 12

```

### Exercise 138

Carry out similar derivations for the following expressions:

1. (fun x -> x + 2) 3
2. let f = fun x -> x in  
f (f 5)
3. let square = fun x -> x \* x in  
let y = 3 in  
square y
4. let id = fun x -> x in  
let square = fun x -> x \* x in  
let y = 3 in  
id square y

#### 13.6.1 More on capturing free variables

There is still a problem in our definition of substitution. Consider the following expression: `let f = fun x -> y in (fun y -> f 3)`

1. Intuitively speaking, this expression seems ill-formed; it defines a function `f` that makes use of an unbound variable `y` in its body. But using the definitions that we have given so far, we would have the

following derivation:

```

let f = fun x -> y in (fun y -> f 3) 1
  ↓
  fun x -> y ↓ fun x -> y
  (fun y -> (fun x -> y) 3) 1
    ↓
    (fun y -> (fun x -> y) 3) ↓ (fun y -> (fun x -> y) 3)
    1 ↓ 1
    (fun x -> 1) 3 ↓
      | fun x -> 1 ↓ fun x -> 1
      | 1 ↓ 1
      | 1 ↓ 1
      ↓ 1
    ↓ 1
  ↓ 1
  ↓ 1

```

The problem happens in the highlighted expression, where according to the  $R_{let}$  rule we should be evaluating  $((\text{fun } y \rightarrow f 3) 1)[f \mapsto \text{fun } x \rightarrow y]$ , which according to our current understanding of substitution should be the highlighted  $(\text{fun } y \rightarrow (\text{fun } x \rightarrow y) 3)$  1.

We're sneaking the  $y$  in  $\text{fun } x \rightarrow y$  inside the scope of the  $\text{fun } y$ . That's not kosher. And the OCaml interpreter seems to agree:

```

# let f = fun x -> y in (fun y -> f 3) 1 ;;
Line 1, characters 17-18:
1 | let f = fun x -> y in (fun y -> f 3) 1 ;;
^
Error: Unbound value y

```

We need to change the definition of substitution to make sure that such VARIABLE CAPTURE doesn't occur. The following rules for substituting inside a function work by replacing the bound variable  $y$  with a new freshly minted variable, say  $z$ , that doesn't occur elsewhere, renaming all occurrences of  $y$  accordingly.

$$\begin{aligned}
(\text{fun } x \rightarrow P)[x \mapsto Q] &= \text{fun } x \rightarrow P \\
(\text{fun } y \rightarrow P)[x \mapsto Q] &= \text{fun } y \rightarrow P[x \mapsto Q] \\
&\quad \text{where } x \not\equiv y \text{ and } y \notin FV(Q) \\
(\text{fun } y \rightarrow P)[x \mapsto Q] &= \text{fun } z \rightarrow P[y \mapsto z][x \mapsto Q] \\
&\quad \text{where } x \not\equiv y \text{ and } y \in FV(Q) \text{ and } z \text{ is a fresh variable}
\end{aligned}$$

**Exercise 139**

Carry out the derivation for

```
let f = fun x -> y in (fun y -> f 3) 1
```

as above but with this updated definition of substitution. What happens at the step highlighted above?

**Exercise 140**

What should the corresponding rule or rules defining substitution on `let ... in ...` expressions be? That is, how should the following rule be completed? You'll want to think about how this construct reduces to function application in determining your answer.

```
(let y = Q in R)[x ↦ P] = ...
```

Try to work out your answer before checking it with the full definition of substitution in Figure 13.4.

**Exercise 141**

Use the definition of the substitution operation above to determine the results of the following substitutions:

1.  $(\text{fun } x \rightarrow x + x)[x \rightarrow 3]$
2.  $(\text{fun } x \rightarrow y + x)[x \rightarrow 3]$
3.  $(\text{let } x = y * y \text{ in } x + x)[x \rightarrow 3]$
4.  $(\text{let } x = y * y \text{ in } x + x)[y \rightarrow 3]$

The implementation of substitution should be updated to handle this issue of avoiding the capture of free variables. The next two exercises do so.

**Exercise 142**

Write a function `free_vars : expr -> vars` that returns a set of variables corresponding to the free variables in the expression as per Figure 13.3. (Recall the discussion of the OCaml library module `Set` in Section 12.8.)

**Exercise 143**

Revise the definition of the function `subst` from Section 13.4.1 to eliminate the problem of variable capture by implementing the set of rules given in Figure 13.4.

$$FV(\bar{m}) = \emptyset \quad (\text{integers}) \quad (13.1)$$

$$FV(x) = \{x\} \quad (\text{variables}) \quad (13.2)$$

$$FV(P + Q) = FV(P) \cup FV(Q) \quad (\text{and similarly for other binary operators}) \quad (13.3)$$

$$FV(P \ Q) = FV(P) \cup FV(Q) \quad (\text{applications}) \quad (13.4)$$

$$FV(\text{fun } x \rightarrow P) = FV(P) - \{x\} \quad (\text{functions}) \quad (13.5)$$

$$FV(\text{let } x = P \text{ in } Q) = (FV(Q) - \{x\}) \cup FV(P) \quad (\text{binding}) \quad (13.6)$$

Figure 13.3: Definition of  $FV$ , the set of free variables in expressions for a functional language with naming and arithmetic.

$$\bar{m}[x \mapsto P] = \bar{m} \quad (13.7)$$

$$x[x \mapsto P] = P \quad (13.8)$$

$$y[x \mapsto P] = y \quad \text{where } x \not\equiv y \quad (13.9)$$

$$(Q + R)[x \mapsto P] = Q[x \mapsto P] + R[x \mapsto P] \quad (13.10)$$

and similarly for other binary operators

$$QR[x \mapsto P] = Q[x \mapsto P]R[x \mapsto P] \quad (13.11)$$

$$(\text{fun } x \rightarrow Q)[x \mapsto P] = \text{fun } x \rightarrow Q \quad (13.12)$$

$$(\text{fun } y \rightarrow Q)[x \mapsto P] = \text{fun } y \rightarrow Q[x \mapsto P] \quad (13.13)$$

where  $x \not\equiv y$  and  $y \notin FV(P)$

$$(\text{fun } y \rightarrow Q)[x \mapsto P] = \text{fun } z \rightarrow Q[y \mapsto z][x \mapsto P] \quad (13.14)$$

where  $x \not\equiv y$  and  $y \in FV(P)$  and  $z$  is a fresh variable

$$(\text{let } x = Q \text{ in } R)[x \mapsto P] = \text{let } x = Q[x \mapsto P] \text{ in } R \quad (13.15)$$

$$(\text{let } y = Q \text{ in } R)[x \mapsto P] = \text{let } y = Q[x \mapsto P] \text{ in } R[x \mapsto P] \quad (13.16)$$

where  $x \not\equiv y$  and  $y \notin FV(P)$

$$(\text{let } y = Q \text{ in } R)[x \mapsto P] = \text{let } z = Q[x \mapsto P] \text{ in } R[y \mapsto z][x \mapsto P] \quad (13.17)$$

where  $x \not\equiv y$  and  $y \in FV(P)$  and  $z$  is a fresh variable

Figure 13.4: Definition of substitution of expressions for variables in expressions for a functional language with naming and arithmetic.

### 13.7 Substitution semantics of recursion

You may observe that the rule for evaluating `let`  $\langle\rangle$  `in`  $\langle\rangle$  expressions doesn't allow for recursion. For instance, the Fibonacci example proceeds as follows:

```
let f = fun n -> if n = 0 then 1 else n * f (n - 1) in f 2
↓
| fun n -> if n = 0 then 1 else n * f (n - 1) ↓ fun n -> if n = 0 then 1 else n * f (n - 1)
| (fun n -> if n = 0 then 1 else n * f (n - 1)) 2
| ↓
| | fun n -> if n = 0 then 1 else n * f (n - 1) ↓ fun n -> if n = 0 then 1 else n * f (n - 1)
| | 2 ↓ 2
| | | if 2 = 0 then 1 else 2 * f (2 - 1) ↓ ???
| | |
| | | ↓ ???
| | |
| | | ↓ ???
| | |
| | | ↓ ???
```

The highlighted expression, `if 2 = 0 then 1 else 2 * f (2 - 1)`, eventually leads to an attempt to apply the unbound variable `f` to its argument 1.

Occurrences of the name **definiendum** in the body are properly replaced with the **definiens**, but occurrences in the definiens itself are not. But what should those recursive occurrences of `f` be replaced by? It doesn't suffice simply to replace them with the definiens, as that still has a free occurrence of the definiendum. Rather, we'll replace them with their own recursive `let` construction, thereby allowing later occurrences to be handled as well. In the factorial example, we'll replace the free occurrence of `f` in the definiens by `let rec f = fun n -> if n = 0 then 1 else n * f (n - 1) in f`, that is, an expression that evaluates to whatever `f` evaluates to *in the context of the recursive definition itself*.

Thus the substitution semantics rule for `let rec`, subtly different from the `let` rule, will be as follows:

```
let rec x = D in B ↓
  | D ↓ v_D
  | B[x ↦ v_D[x ↦ let rec x = v_D in x]] ↓ v_B
  | ↓ v_B
    (Rletrec)
```

Continuing the factorial example above, we would substitute for `f` in the third line the expression `let rec f = fun n -> if n = 0 then 1 else n * f (n - 1) in f`, forming

```
(fun n -> if n = 0 then 1
           else n * (let rec f = fun n -> if n = 0 then 1
                           else n * f (n-1) in f) (n-1)) 2
```

Proceeding further, the final line becomes

```
if 2 = 0 then 1
else 2 * (let rec f = fun n -> if n = 0 then 1
                           else n * f (n-1) in f) (2-1))
```

which will (eventually) evaluate to 2.

#### **Exercise 144**

Thanklessly continue this derivation until it converges on the final result for the factorial of 2, viz., 2. Then thank your lucky stars that we have computers to do this kind of rote repetitive task for us.

We'll provide an alternative approach to semantics of recursion when we introduce environment semantics in Chapter 19.



We defined a set of formal rules providing the meanings of OCaml expressions via simplifying substitutions of equals for equals, resulting finally in the values that most simply encapsulate the meanings of complex expressions.

An interpreter for a programming language (the object language) written in the same programming language (as metalanguage) – a METACIRCULAR INTERPRETER – provides another way of getting at the semantics of a language. In fact, the first semantics for the programming language LISP was given as a metacircular interpreter.

In both cases, we see the advantage of having a language with a small core, sprinkled liberally with syntactic sugar, since only the core need be given the formal treatment through rules or metacircular interpretation. The syntactic sugar can be translated out. For instance, although the metacircular interpreter that we started developing here does not handle the more compact function definition notation seen in

```
let f x = x + 1
```

this expression can be taken as syntactic sugar for (that is, a variant concrete syntax for the abstract syntax of) the expression

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

which we already have defined formal rules to handle. In the case of a metacircular interpreter, we can imagine that the parser for the former expression will simply provide the abstract syntax of the latter.

Our exploration of rigorous semantics for programs will continue once the substitution approach starts to falter in the presence of state change and imperative programming.

$\bar{n} \Downarrow \bar{n}$  $(R_{int})$  $\text{fun } x \rightarrow B \Downarrow \text{fun } x \rightarrow B$  $(R_{fun})$  $P + Q \Downarrow$ 

$$\begin{array}{l} P \Downarrow \bar{m} \\ Q \Downarrow \bar{n} \end{array}$$

 $(R_+)$  $\Downarrow \overline{m+n}$  $P / Q \Downarrow$ 

$$\begin{array}{l} P \Downarrow \bar{m} \\ Q \Downarrow \bar{n} \end{array}$$

 $(R/_)$  $\Downarrow \overline{[m/n]}$  $P Q \Downarrow$ 

$$\begin{array}{l} P \Downarrow \text{fun } x \rightarrow B \\ Q \Downarrow v_Q \\ B[x \mapsto v_Q] \Downarrow v_B \end{array}$$

 $(R_{app})$  $\Downarrow v_B$  $\text{let } x = D \text{ in } B \Downarrow$ 

$$\begin{array}{l} D \Downarrow v_D \\ B[x \mapsto v_D] \Downarrow v_B \end{array}$$

 $(R_{let})$  $\Downarrow v_B$  $\text{let rec } x = D \text{ in } B \Downarrow$ 

$$\begin{array}{l} D \Downarrow v_D \\ B[x \mapsto v_D[x \mapsto \text{let rec } x = v_D \text{ in } x]] \Downarrow v_B \end{array}$$

 $(R_{letrec})$ 

Figure 13.5: Substitution semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

*13.8 Supplementary material*

- Lab 9: Substitution semantics



# 14

## *Efficiency, complexity, and recurrences*

We say that some agent is *efficient* if it makes the best use of a scarce resource to generate a desired output. Furnaces turn the scarce resource of fuel into heating, so an efficient furnace is one that generates the most heat using the least fuel. Similarly, an efficient shooter in basketball generates the most points using the fewest field goal attempts. Standard measurements of efficiency reflect these notions. Furnaces are rated for [Annual Fuel Utilization Efficiency](#), NBA players for [Effective Field Goal Percentage](#).

Computer programs use scarce resources to generate desired outputs as well. Most prominently, the resources expended are time and “space” (the amount of memory required during the computation), though power is increasingly becoming a resource of interest.

Up to this point, we haven’t worried about the efficiency of the programs we’ve written. And for good reason. Donald Knuth, Professor Emeritus of the Art of Computer Programming at Stanford University and Turing-Award-winning algorithmist, warns of PREMATURE OPTIMIZATION:

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. ([Knuth, 1974](#))

Knuth’s point is that *programmers’ time is a scarce resource too*, and often the most important one.

Nonetheless, sometimes issues of code efficiency become important – Knuth’s 3% – and in any case the special ways of thinking and tools for reasoning about efficiency of computation are important aspects of computational literacy, most centrally ideas of

- Complexity as the *scaling* of resource usage,



Figure 14.1: Donald Knuth (1938– ), Professor Emeritus of the Art of Computer Programming at Stanford University. In this photo, he holds a volume of his seminal work *The Art of Computer Programming*.

- Comparison of *asymptotic* scaling,
- *Big-O notation* for specifying asymptotic scaling, and
- *Recurrences* as the means to capture and solve for resource usage.

In this chapter, we describe these important computational notions in the context of an extended example, the comparison of two sorting functions.

### 14.1 The need for an abstract notion of efficiency

With furnaces and basketball players, we can express a notion of efficiency as a single number – Annual Fuel Utilization Efficiency or Effective Field Goal Percentage. With computer programs, things are not so simple. Consider, for example, one of the most fundamental of all computations, **SORTING** – ordering the elements of a list according to a comparison function. Given a particular function to sort lists, we can't characterize its efficiency – how long it takes to sort lists – as a single number. What number would we use? That is, how long does it take to sort a list of integers using the function? The answer, of course, is “it depends”; in particular, it depends on

- *Which input?* How many elements are in the list? What order are they in? Are there a lot of duplicate items, or very few?
- *How computed?* Which computer are you using, and which software environment? How long does it take to execute the primitive computations out of which the function is built?

All of these issues affect the running time of a particular sorting function. To make any progress on comparing the efficiency of functions in the face of such intricacy, it is clear that we will need to come up with a more abstract way of characterizing the efficiency of computations.

We address these two issues separately. To handle the question of “which input”, we might characterize the efficiency of the sorting program not as a number (a particular running time), but as a *function* from inputs to numbers. However, this doesn’t seem an appealing option; we want to be able to draw some general conclusions for comparing sorting programs, not have to reassess for each possible input.

Nonetheless, the idea of characterizing efficiency in terms of some function is a useful one. Broadly speaking, algorithms take longer on bigger problems, so we might use a function that provides the time required as a function of the *size* of the input. In the case of sorting lists, we might take the size of the input to be the number of elements in the list to be sorted. Unfortunately, for any given input size, the program

might require quite different amounts of time. What should we take to be *the* time required for problems of a given size. There are several options: We might consider the time required *on average* for instance. But we will use the time required *in the worst case*. When comparing algorithms, we might well want to plan for the worst case behavior of a program, just to play it safe. We will refer to the function from input sizes to worst-case time needed as the WORST-CASE COMPLEXITY of the algorithm.

We've made some progress. Rather than thinking of resource usage as a single number (too coarse) or a function from problem inputs to numbers (too fine), we use the programs worst-case complexity, a function from sizes of inputs to worst-case resource usage on inputs with those sizes. But even this is not really well defined, because of the "How computed?" question. One and the same program, running on different computers, say, may have wildly different running times.

To make further progress, let's take a concrete example. We'll examine two particular sorting algorithms.

## 14.2 Two sorting functions

A module signature for sorting can be given by

```
# module type SORT =
#   sig
#     (* sort lt xs -- Returns the list xs sorted in increasing
#        order by the "less than" function lt. *)
#     val sort : ('a -> 'a -> bool) -> 'a list -> 'a list
#   end ;
module type SORT =
sig val sort : ('a -> 'a -> bool) -> 'a list -> 'a list end
```

The `sort` function takes as its first argument a comparison function, which specifies when one element should be sorted before another in the desired ordering.

A simple implementation of the signature is the INSERTION SORT algorithm, which operates by inserting the elements of the unsorted list one by one into an empty list, each in its appropriate place.<sup>1</sup>

```
# module InsertSort : SORT =
#   struct
#     let rec insert lt xs x =
#       match xs with
#       | [] -> [x]
#       | hd :: tl -> if lt x hd then x :: xs
#                      else hd :: (insert lt tl x)
#
#     let rec sort (lt : 'a -> 'a -> bool)
#                 (xs : 'a list)
#                 : 'a list =
```

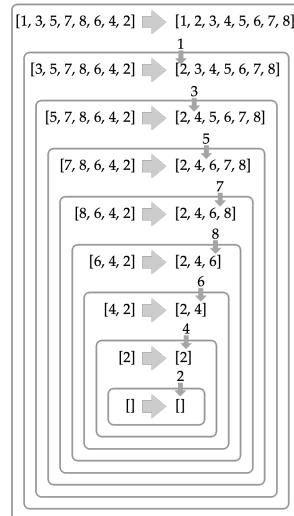


Figure 14.2: An example of the recursive insertion sort algorithm, sorting the list [1, 3, 5, 7, 8, 6, 4, 2]. Each recursive call is marked with a rounded box, in which the tail is sorted, and the head then inserted.

<sup>1</sup> Insertion sort could have been implemented more elegantly using a single `fold_left`, but we make the recursion explicit to facilitate the later complexity analysis.

```

#      match xs with
#      | [] -> []
#      | hd :: tl -> insert lt (sort lt tl) hd
#  end;;
module InsertSort : SORT

```

We can use insertion sort to sort some integers in increasing order:

```

# InsertSort.sort (<) [1; 3; 5; 7; 8; 6; 4; 2] ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]

```

or some floats in decreasing order:

```

# InsertSort.sort (>) [2.71828; 1.41421; 3.14159; 1.61803] ;;
- : float list = [3.14159; 2.71828; 1.61803; 1.41421]

```

An especially elegant implementation of sorting is the MERGE SORT algorithm, first described by John von Neumann in 1945 (according to [Knuth \(1970\)](#)). It works by dividing the list to be sorted into two lists of (roughly) equal size. Each of the halves is then sorted, and the resulting sorted halves are merged together to form the sorted full list. This recursive process of dividing the list in half can't continue indefinitely; at some point the recursion must “bottom out”, or the process will never terminate. In the implementation below, we bottom out when the list to be sorted contains at most a single element. The sort function can be defined then as

```

let rec sort lt xs =
  match xs with
  | []
  | [_] -> xs
  | _ -> let first, second = split xs in
            merge lt (sort lt first) (sort lt second) ;;

```

The mergesort definition above makes use of functions

```
split : 'a list -> 'a list * 'a list
```

and

```
merge : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list .
```

A call to `split lst` returns a pair of lists, each containing half of the elements of `lst`. (In case, `lst` has an odd number of elements, the extra element can go in either list in the returned pair.) A call to `merge lt xs ys` returns a list containing all of the elements of `xs` and `ys` sorted according to `lt`; it assumes that `xs` and `ys` are themselves already sorted.

#### Exercise 145

Provide implementations of the functions `split` and `merge`, and package them together with the `sort` function just provided in a module `MergeSort` satisfying the `SORT` module type. You should then have a module that allows for the following interactions:

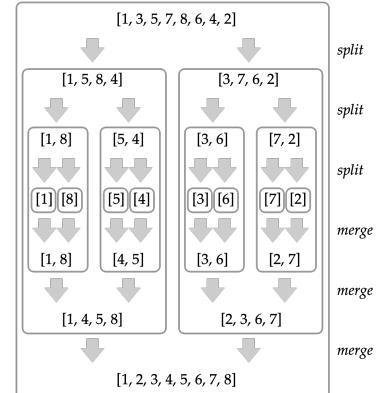


Figure 14.3: An example of the recursive mergesort algorithm, sorting the list [1, 3, 5, 7, 8, 6, 4, 2]. Each recursive call is marked with a rounded box, in which the list is split, sorted, and merged.

```
# MergeSort.sort (<) [1; 3; 5; 7; 8; 6; 4; 2] ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
# MergeSort.sort (>) [2.7183; 1.4142; 3.1416; 1.6180] ;;
- : float list = [3.1416; 2.7183; 1.618; 1.4142]
```

(Another elegant recursive sorting algorithm, quicksort, is explored further in Section 16.4.)

### 14.3 Empirical efficiency

How efficient are these algorithms? The time usage of the algorithms can be compared by timing each of them on the same input. Here, we make use of a simple timing function `call_timed` : ('a -> 'b) -> 'a -> ('b \* float). Calling `call_timed f x` evaluates the application of the function `f` to `x`, returning the result paired with the number of milliseconds required to perform the computation.

Now we can sort a list using the two sorting algorithms, reporting the timings as well.<sup>2</sup>

```
# (* Generate some lists of random integers *)
# let shortlst = List.init 5 (fun _ -> Random.int 1000) ;;
val shortlst : int list = [344; 685; 182; 641; 439]
# let longlst = List.init 500 (fun _ -> Random.int 1000) ;;
val longlst : int list =
[500; 104; 20; 921; 370; 217; 885; 949; 678; 615; ...]

# (* test_repeated count f x -- Apply `f` to `x` `count` times,
#    ignoring the results and returning the time
#    taken in milliseconds. *)
# let test_repeated sort lst label =
#   let _, time = Absbook.call_timed
#     (List.init 1000)
#     (fun _ -> (sort (<) lst)) in
#   Printf.printf "%-20s %10.4f\n" label time ;;
val test_repeated : (('a -> 'a -> bool) -> 'b -> 'c) -> 'b ->
  string -> unit =
<fun>

# (* Sort each list two ways *)
# List.iter (fun (sort, lst, label) ->
#   test_repeated sort lst label)
#   [ InsertSort.sort, shortlst, "insertion short";
#     MergeSort.sort, shortlst, "merge short";
#     InsertSort.sort, longlst, "insertion long";
#     MergeSort.sort, longlst, "merge long" ] ;;
insertion short      0.6180
merge   short       1.2631
insertion long      3023.9391
merge   long        459.2381
- : unit = ()
```

Not surprisingly, it appears that sometimes the insertion sort algorithm is faster (as on `shortlst`) and sometimes mergesort is faster (as

<sup>2</sup>We're taking advantage of several useful functions here. The `map` function from the `List` library module is familiar from Chapter 8. The `Absbook` module, available at <http://url.cs51.io/absbookml> provides some useful functions that we'll use throughout the book, for example, the `range` function and several timing functions.

on `longlst`). It doesn't seem possible to give a definitive answer as to which is faster in general.

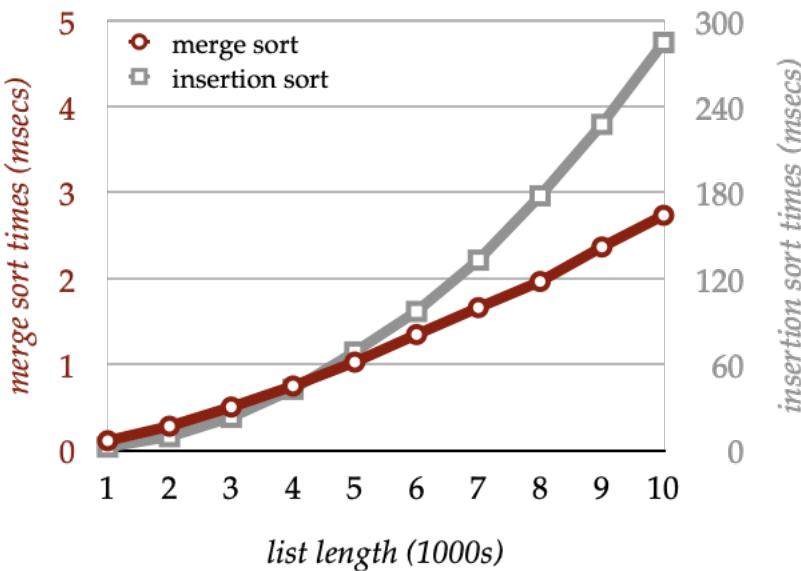


Figure 14.4: Run time in seconds for sorting random lists of lengths varying from 1,000 to 10,000 elements, generated by averaging run time over 100 trials. The two lines show performance for insertion sort and merge sort, with insertion sort times using the right scale to allow for comparison.

If we examine the performance of the algorithm for a broader range of cases, however, a pattern emerges. For short lists, insertion sort is somewhat faster, but as the lists grow in length, the time needed to sort them grows faster for insertion sort than for mergesort, so that eventually mergesort shows a consistent performance advantage. The pattern is quite clear from the graph in Figure 14.4. The key to comparing the algorithms, then, is not their comparative efficiency on any particular list, but rather the character of their efficiency *as their inputs grow in size*. As we argued in Section 14.1, thinking about the time required as a function of the size of the inputs looks like a good idea.

However, as also noted above, a problem with analyzing algorithms, as we have just done, by running them with particular implementations on particular computers on particular lists, is that the results may apply only for those particulars. Instead, we'd like a way of characterizing the algorithms' relative performance whatever the particulars. Measuring running times empirically is subject to idiosyncrasies of the measurement exercise: the relative time required for different primitive operations on the particular computer being used and with the particular software tools, what other operations were happening on the computer at the same time, imprecision in the computer's clock, whether the operating system is slowing down or speeding up the CPU

for energy-saving purposes, and on and on. The particularities also may not be predictive of the future as computers change over time, with processing and memory retrieval and disk accesses becoming faster – and faster at varying rates. The empirical approach doesn't get at the intrinsic properties of the algorithms.

The approach we will take, then, is to analyze the algorithms in terms of the intrinsic growth rate of their performance as the size of their inputs grow, their worst-case complexity. Detailed measurement and analysis can be saved for later, once the more fundamental complexity issues are considered. We thus take an abstract view of performance, rather than a concrete one. This emphasis on abstraction, as usual, comes from thinking like a computer scientist, and not a computer programmer.

The time complexity of the two sorting algorithms can be thought of as functions (!) from the size of the input to the amount of time needed to sort inputs of that size. As it turns out – and as we will show in Sections 14.5.5 and 14.5.9 – for insertion sort on a list of size  $n$ , the time required to sort the list grows as the function

$$T_{is}(n) = a \cdot n^2 + b$$

whereas for mergesort, the time required to sort the list grows as the function

$$T_{ms}(n) = c \cdot n \log n + d$$

where  $a$ ,  $b$ ,  $c$ , and  $d$  are some constants. For a given  $n$ , which is larger? That depends on these constants of course. But regardless of the constants, as  $n$  increases  $T_{is}$  grows “faster” than  $T_{ms}$  in a way that we will make precise shortly.

In order to make good on this idea of comparing algorithms by comparing their growth functions, then, we must pay on two promissory notes:

1. How to figure out the growth function for a given algorithm, and
2. How to determine which growth functions are growing faster.

In the remainder of this chapter, we will address the first of these with a technique of recurrence equations, and the second with the idea of asymptotic complexity and “big- $O$ ” notation.

#### 14.4 Big- $O$ notation

Which is better, an algorithm (like insertion sort) with a complexity that grows as  $a \cdot n^2 + b$  or an algorithm (like mergesort) with a complexity that grows as  $c \cdot n \log n + d$ ? The answer “it depends on the values

of the constants” seems unsatisfactory, since intuitively, a function that grows quadratically (as the square of the size) like the former will *eventually* outstrip a function that grows like the latter. Figure 14.5 shows this graphically. The gray lines all grow as  $c \cdot n \log n$  for increasing values of  $c$ . But regardless of  $c$ , the red line, displaying quadratic growth, eventually outpaces all of the gray lines. In a sense, then, we’d *eventually* like to use the  $n \log n$  algorithm regardless of the constants. It is this ASYMPTOTIC (that is, long term or eventual) sense that we’d like to be able to characterize.

To address the question of how fast a function grows asymptotically, independent of the annoying constants, we introduce a generic way of expressing the growth rate of a function – BIG-O NOTATION.

We’ll assume that problem sizes are non-negative integers and that times are non-negative as well. Given a function  $f$  from non-negative integers to non-negative numbers,  $O(f)$  is the set of functions that *grow no faster than*  $f$ , in the following precise sense.<sup>3</sup> We define  $O(f)$  to be the set of all functions  $g$  such that for all “large enough”  $n$  (that is,  $n$  larger than some value  $n_0$ ),  $g(n) \leq c \cdot f(n)$ .

The roles of the two constants  $n_0$  and  $c$  are exactly to move beyond the details of constants like the  $a$ ,  $b$ ,  $c$ , and  $d$  in the sorting algorithm growth functions. In deciding whether a function grows no faster than  $f$ , we don’t want to be misled by a few input values here and there where  $g(n)$  may happen to be larger than  $f(n)$ , so we allow exempting values smaller than some fixed value  $n_0$ . The point is that as the inputs grow in size, *eventually* we’ll get past the few input sizes  $n$  where  $g(n)$  is larger than  $f(n)$ . Similarly, if the value of  $g(n)$  is always, say, twice the value of  $f(n)$ , the two aren’t growing at qualitatively different rates. Perhaps that factor of 2 is based on just the kinds of idiosyncrasies that can change as computers change. We want to ignore such constant multiplicative factors. For that reason, we don’t require that  $g(n)$  be less than  $f(n)$ ; instead we require that  $g(n)$  be less than *some constant multiple*  $c$  of  $f(n)$ .

As an example of big-O notation, consider two simple polynomial functions. It will be convenient to use Church’s elegant lambda notation (see Section B.1.4) to specify these functions directly:  $\lambda n.10n^2 + 3$  and  $\lambda n.n^2$ .

Is the function  $\lambda n.10n^2 + 3$  an element of the set  $O(\lambda n.n^2)$ ? To demonstrate that it is, we need to find constants  $c$  and  $n_0$  such that for all  $n > n_0$ ,  $10n^2 + 3 \leq c \cdot n^2$ . It turns out that the values  $n_0 = 0$  and  $c = 13$  do the trick, that is, for all  $n > 0$ ,  $10n^2 + 3 \leq 13n^2$ . We can prove this as follows: Since  $n \geq 1$ , it follows that  $n^2 \geq 1$  and thus  $3 \leq 3n^2$ . Thus  $10n^2 + 3 \leq 10n^2 + 3n^2 = 13n^2$ . We conclude, then, that

$$\lambda n.10n^2 + 3 \in O(\lambda n.n^2) .$$

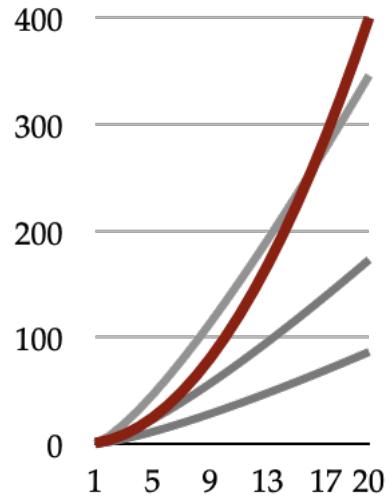


Figure 14.5: A graph of functions with different growth rates. The highlighted line grows as  $n^2$ . The three gray lines grow as  $c \cdot n \log n$ , where  $c$  is, from bottom to top, 1, 2, and 4.

<sup>3</sup> Since it takes a function as its argument and returns sets of functions as its output,  $O$  is itself a higher-order function!

Of course, the converse is also true:

$$\lambda n.n^2 \in O(\lambda n.10n^2 + 3) \quad .$$

We can just take  $n_0$  again to be 0 and  $c$  to be 1, since  $n^2 < 10n^2 + 3$  for all  $n$ .

#### 14.4.1 Informal function notation

It is conventional, when using big-*O* notation, to stealthily move between talk of functions (like  $\lambda n.n^2$ ) to the corresponding body expression (like  $n^2$ ), leaving silent the particular variable (in this case  $n$ ) that represents the input of the function. Typically, the variable is clear from context (and indeed is frequently the variable  $n$  itself). For instance, we might say

$$10n^2 + 3 \in O(n^2) \quad ,$$

rather than the more complex formulation above.

Continuing this abuse of notation, we sometimes write variables for functions, like  $f$  or  $g$ , not only to stand for a function, but also the corresponding body expression. This allows us to write things like  $k \cdot f$  (where  $k$  is a constant) to mean the function whose body expression is the product of  $k$  and the body expression of  $f$ . (This turns out to be equivalent to the rather more cumbersome  $\lambda n.k \cdot f(n)$ .) Suppose  $f$  is the function  $\lambda n.n^2$ . Then we write  $k \cdot f$  to mean, not  $k \cdot \lambda n.n^2$  (which is an incoherent formula), but rather  $k \cdot n^2$ , which, again by convention, glosses  $\lambda n.k \cdot n^2$ .

This convention of sliding between functions and their body expressions may seem complicated, but it soon becomes quite natural. And it allows us to formulate important properties of big-*O* notation very simply, as we do in the next section.

Of course, there are problems with playing fast and loose with notation in this way. First, writing  $O(n^2)$  makes it look like  $O$  is a function that takes integers as input, since  $n^2$  looks like it specifies an integer. But  $O$  is a function from *functions*, not from integers. Second, what are we to make of something like  $O(m \cdot n^2)$ ? Does this specify the set of functions that grow no faster than the function from  $m$  to  $m \cdot n^2$ , that is,  $O(\lambda m.m \cdot n^2)$ ? Or does it specify the set of functions that grow no faster than the function from  $n$  to  $m \cdot n^2$ , that is,  $O(\lambda n.m \cdot n^2)$ ? The notation doesn't make clear which variable is the one representing the input to the function – which variable the growth is relative to. In cases such as this, computer scientists rely on context to make clear what the notation is supposed to mean.

We'll stick to this informal notation since it is universally used.

But you'll want to always remember that  $O$  maps *functions* to sets of functions.

#### 14.4.2 Useful properties of $O$

In general, it's tedious to prove particular cases of big- $O$  membership like the example in Section 14.4. Instead, you'll want to acquire a general understanding of these big- $O$  sets of functions, and reason on the basis of that understanding.

The big- $O$  notation brings together whole classes of functions whose growth rates are similar. These classes of functions have certain properties that make them especially useful.<sup>4</sup> First of all, every function grows no faster than itself:

$$f \in O(f)$$

Adding a constant to a function doesn't change its big- $O$  classification:

If  $g \in O(f)$ , then<sup>5</sup>

$$g + k \in O(f) .$$

We can reason immediately, then, that  $2n^2 + 3 \in O(2n^2)$  (or, more pedantically,  $\lambda n.2n^2 + 3 \in O(\lambda n.2n^2)$ ), without going through a specific proof.

Similarly, multiplying by a constant ( $k$ ) doesn't affect the class either. If  $g \in O(f)$ , then

$$k \cdot g \in O(f) .$$

Thus,  $2n^2 \in O(n^2)$ . Together with the results above, we can conclude that  $2n^2 + 3 \in O(n^2)$ .

In fact, adding in any lower degree terms doesn't matter. If  $f \in O(n^k)$  and  $g \in O(n^c)$ , where  $k > c$ :

$$f + g \in O(n^k)$$

The upshot of all this is that in determining the big- $O$  growth rate of a polynomial function, we can always just drop lower degree terms and multiplicative constants. In thinking about the growth rate of a complicated function like  $4n^3 + 142n + 3$ , we can simply ignore all but the largest degree term ( $4n^3$ ) and even the multiplicative constant 4, and conclude that

$$4n^3 + 142n + 3 \in O(n^3)$$

<sup>4</sup> The mathematically inclined might want to take a stab at proving these properties of big- $O$ .

<sup>5</sup> Here's our first instance of the informal function notation in the wild.

#### Exercise 146

Which of these claims about the growth rates of various functions hold?

1.  $3n + 5 \in O(n)$
2.  $n \in O(3n + 5)$
3.  $n + n^2 \in O(n)$
4.  $n^3 + n^2 \in O(n^3 + 2n)$
5.  $n^2 \in O(n^3)$
6.  $n^3 \in O(n^2)$
7.  $32n^3 \in O(n^2 + n + k)$

Finally, the sum or product of functions grows no faster than the sum or product, respectively, of their respective growth rates. If  $f' \in O(f)$  and  $g' \in O(g)$ , then

$$\begin{aligned} f' + g' &\in O(f + g) \\ f' \cdot g' &\in O(f \cdot g) \end{aligned}$$

We can thus conclude that

$$(5n^3 + n^2) \cdot (3\log n + 7) \in O(n^3 \cdot \log n)$$

#### 14.4.3 Big-O as the metric of relative growth

We are interested in the big-O classification of functions in particular because we can use it to compare functions as to which asymptotically grows faster. In particular, if  $f \in O(g)$  but  $g \notin O(f)$ , then  $g$  grows faster than  $f$ , which we notate  $g \gg f$ .

For example,

$$n^2 \in O(n^3) \quad ,$$

but the converse doesn't hold:

$$n^3 \notin O(n^2) \quad .$$

We can conclude, then that

$$n^3 \gg n^2 \quad ,$$

that is,  $n^3$  grows faster than  $n^2$ .

More generally,

- Functions with bigger exponents grow faster:

$$n^k \gg n^c \quad \text{when } k > c$$

- Linear functions grow faster than logarithmic functions:

$$n \gg \log n$$

- Exponentials grow faster than polynomials:

$$2^n \gg n^k$$

- The exponential base matters; exponentials with larger base grow faster:

$$3^n \gg 2^n$$

We can think of big- $O$  as defining classes of functions that grow at similar rates, up to multiplicative constants. Thus,  $O(n)$  is the set of functions whose growth rate is (at most) linear,  $O(n^2)$  the set of functions whose growth rate is (at most) quadratic,  $O(n^3)$  the set of cubic functions,  $O(2^n)$  the set of base-two exponential functions. We can then place these classes in an ordering ( $\gg$ ) as to which classes grow faster inherently (and not just because of the values of some contingent constants).

From the properties above, we can conclude that  $n^2 \gg n \log n$ , and therefore, since  $T_{is} \in O(n^2)$  and  $T_{ms} \in O(n \log n)$ , that  $T_{is} \gg T_{ms}$ .<sup>6</sup> Mergesort has lower complexity – is asymptotically more efficient – than insertion sort. This conclusion is independent of which computers we time the algorithms on, or other particularities that affect the constants.

Of course, this reasoning relies on knowing the functions for how each algorithm's performance scales. (In the discussion above, we merely asserted the growth rate functions for the two sorting algorithms.) Only then can we use big- $O$  to determine which algorithm scales better, which is more efficient in a deeper sense than just testing a particular instance or two. We still need a way to determine for a particular algorithm the particular resource-usage function. This is the second promissory note, and the one that we now address.

#### Problem 147

Two friends who work at EuclidCo tell you that they're looking for a fast algorithm to solve a problem they're working on. So far, they've each developed an algorithm: algorithm A has time complexity  $O(n^3)$  and algorithm B is  $O(2^n)$ . They prefer algorithm A, and use three different arguments to convince you of their preference. For each argument, evaluate the truth of the bolded statement, and justify your answer.

1. “We’re all about speed at EuclidCo, and **A will always be faster than B**.”
2. “In a high stakes industry like ours, we can’t afford to have more than a finite number of inputs that run slower than polynomial time, and **we can avoid this if we go with A**.”
3. “We work with big data at EuclidCo. **For suitably large inputs, A will be faster on average than B**.”

<sup>6</sup> Strictly speaking, we'd have to further show that  $T_{is} \notin O(n \log n)$ , but we'll ignore this nicety in general here and in the following discussion.

## 14.5 Recurrence equations

Given an algorithm, how are we to determine how much time it needs as a function of the size of its input? In this section, we introduce one

method, based on the solving of recurrence equations, to address this question.

We start with a simple example, the `append` function to append two lists, defined as

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

An appropriate measure for the size of the input to the function is the sizes of the two lists it is to append. Let's use  $T_{\text{append}}(n, m)$  for the time required to run the `append` function on lists with  $n$  and  $m$  elements respectively. What do we know about this  $T_{\text{append}}$ ?

When the first argument, `xs`, is the empty list (so  $n = 0$ ), the function performs just a few simple actions, pattern-matching the input against the empty list pattern, and then returning `ys`. If we say that the time for the pattern match is some constant  $c_{\text{match}}$  and the time for the return is some constant  $c_{\text{returnys}}$ , then we have that

$$T_{\text{append}}(0, m) = c_{\text{match}} + c_{\text{returnys}} .$$

Since the sum of the two constants is itself a constant, we can simplify by treating the whole as a new constant  $c$ :

$$T_{\text{append}}(0, m) = c$$

When the first argument is nonempty, the computation performed again has a few parts: the match against the first pattern (which fails), the match against the second pattern (which succeeds), the recursive call to `append`, the cons of `hd` and the result of the recursive call. Each of these (except for the recursive call) takes some constant time, so we can characterize the amount of time as

$$T_{\text{append}}(n + 1, m) = c_{\text{matchcons}} + T_{\text{append}}(n, m) .$$

Here, we use  $n + 1$  as the length of the first list, as we know it is at least one element long. The recursive call is appending the tail of `xs`, a list of length  $n$ , to `ys`, a list of length  $m$ , and thus (by hypothesis) takes time  $T_{\text{append}}(n, m)$ .

Merging and renaming constants, we thus have the following two RECURRENCE EQUATIONS that characterize the running time of the `append` function in terms of the size of its arguments:

$$T_{\text{append}}(0, m) = c$$

$$T_{\text{append}}(n + 1, m) = k + T_{\text{append}}(n, m)$$

#### 14.5.1 Solving recurrences by unfolding

Because the recurrence equations defining  $T_{append}$  use  $T_{append}$  itself, recursively, in the definition (hence the term “recurrence”), they don’t provide a CLOSED-FORM (nonrecursive) solution to the question of characterizing the running time of the function. To get a solution in closed form, we will use a method called UNFOLDING to solve the recurrence equations.

Consider the general case of  $T_{append}(n, m)$  and assume that  $n > 0$ . By the second recurrence equation,

$$T_{append}(n, m) = k + T_{append}(n - 1, m) \quad .$$

Now  $T_{append}(n - 1, m)$  itself can be unfolded as per the second recurrence equation, so

$$T_{append}(n, m) = k + k + T_{append}(n - 2, m) \quad .$$

Continuing in this vein, we can continue to unfold until the first argument to  $T_{append}$  becomes 0:

$$\begin{aligned} T_{append}(n, m) &= k + T_{append}(n - 1, m) \\ &= k + k + T_{append}(n - 2, m) \\ &= k + k + k + T_{append}(n - 3, m) \\ &= \dots \\ &= k + k + k + \dots + k + T_{append}(0, m) \end{aligned}$$

How many unfoldings are required until the first argument reaches 0? We’ll have had to unfold  $n$  times. There will therefore be  $n$  instances of  $k$  being summed in the unfolded equation. Completing the derivation, then, using the first recurrence equation,

$$\begin{aligned} T_{append}(n, m) &= k \cdot n + T_{append}(0, m) \\ &= k \cdot n + c \end{aligned}$$

We now have the closed-form solution

$$T_{append}(n, m) = k \cdot n + c$$

Notice that the time required is independent of  $m$ , the size of the second argument. That makes sense because the code for append never looks inside the structure of the second argument; the computation therefore doesn’t depend on its size.

Now, the function  $k \cdot n + c \in O(n)$ . Thus the time complexity of append is  $O(n)$  or *linear* in the length of its first argument. This is typical of algorithms that operate by recursively marching down a list one element at a time.

In order to apply the same kinds of techniques to determine the time complexity of the two sorting algorithms, we'll work through a series of examples.

#### 14.5.2 Complexity of reversing a list

There are multiple ways of implementing list reversal. We show that they can have quite different time complexities. We start with a naive implementation, which works by reversing the tail of the list and appending the head on the end:

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

We define recurrence equations for the time  $T_{rev}(n)$  to reverse a list of length  $n$  using this implementation. If the list is empty, we have (similarly to the case of append, and introducing constants as needed):

$$T_{rev}(0) = c_{match} + c_{return} = q$$

For nonempty lists, we must perform the appropriate match, reverse the tail, cons the head onto the empty list, and perform the append:

$$\begin{aligned} T_{rev}(n+1) &= c_{match} + c_{cons} + T_{rev}(n) + T_{append}(n, 1) \\ &= r + T_{rev}(n) + T_{append}(n, 1) \\ &= r + T_{rev}(n) + k \cdot n + c \\ &= k \cdot n + s + T_{rev}(n) \end{aligned}$$

The closed form solution for append from the previous section becomes useful here. And again, notice our free introduction of new constants to simplify things. We take the sum of  $c_{match}$  and  $c_{cons}$  to be  $r$ , then for  $r + c$  we introduce  $s$ . Summarizing, the reverse implementation above yields the recurrence equations

$$T_{rev}(0) = q$$

$$T_{rev}(n+1) = k \cdot n + s + T_{rev}(n)$$

which we must now solve to find a closed form.

We again unfold  $T_{rev}(n)$ :

$$\begin{aligned}
 T_{rev}(n) &= k \cdot (n - 1) + s + T_{rev}(n - 1) \\
 &= k \cdot (n - 1) + s + k \cdot (n - 2) + s + T_{rev}(n - 2) \\
 &= k \cdot (n - 1) + s + k \cdot (n - 2) + s \\
 &\quad + k \cdot (n - 3) + s + T_{rev}(n - 3) \\
 &= \dots \\
 &= k \cdot \sum_{i=1}^{n-1} (n - i) + s \cdot n + T_{rev}(0) \\
 &= k \cdot \sum_{i=1}^{n-1} (n - i) + s \cdot n + q \\
 &= k \cdot \sum_{i=1}^{n-1} i + s \cdot n + q \\
 &= k \cdot \sum_{i=1}^n i - k \cdot n + s \cdot n + q \\
 &= k \cdot \sum_{i=1}^n i + (s - k) \cdot n + q
 \end{aligned}$$

To make further progress on achieving a simple closed form for the recurrence, it would be ideal to simplify the summation  $\sum_{i=1}^n i$  of the integers from 1 to  $n$ . Famously (if apocryphally), the seven-year-old mathematical prodigy Carl Friedrich Gauss (1777–1855) solved this problem in his head. Gauss was asked by his teacher, so the story goes, to sum all of the integers from 1 to 100. That'll keep him quiet for a bit, the teacher presumably thought. But Gauss came up with the answer – 5050 – immediately, by taking advantage of the simple identity

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

For a graphical “proof” that the identity holds, see Figure 14.6. A more traditional proof is provided in Section B.2.

Making use of this identity,

$$\begin{aligned}
 T_{rev}(n) &= k \cdot \sum_{i=1}^n i + (s - k) \cdot n + q \\
 &= k \cdot \frac{n \cdot (n + 1)}{2} + (s - k) \cdot n + q \\
 &= \frac{k}{2} n^2 + \frac{k}{2} n + (s - k) \cdot n + q \\
 &= \frac{k}{2} n^2 + (s - \frac{k}{2}) \cdot n + q \\
 &\in O(n^2)
 \end{aligned}$$

concluding that the function has quadratic ( $O(n^2)$ ) complexity. The last step really shows the power of big- $O$  notation, allowing to strip

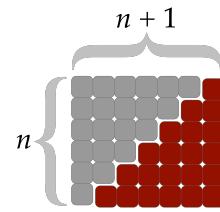


Figure 14.6: A graphical proof that

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

Two triangles, each formed by piling up squares with rows from 1 to  $n$  can be combined to form a rectangle of area  $n \cdot (n + 1)$ . Each triangle is half that area, that is,  $\frac{n \cdot (n + 1)}{2}$ . A more algebraic proof is given in Section B.2.

away all of the constants and lower order terms to get at the essence of the growth rate.

#### Problem 148

Recall that the `Stdlib.compare` function compares two values, returning an `int` based on their relative magnitude: `compare x y` returns `0` if `x` is equal to `y`, `-1` if `x` is less than `y`, and `+1` if `x` is greater than `y`.

A function `compare_lengths : 'a list -> 'b list -> int` that compares the lengths of two lists can be implemented using `compare` by taking advantage of the `length` function<sup>7</sup> from the `List` module:

```
let compare_lengths xs ys =
  compare (List.length xs) (List.length ys) ;;
```

For instance,

```
# compare_lengths [1] [2; 3; 4] ;;
- : int = -1
# compare_lengths [1; 2; 3] [4] ;;
- : int = 1
# compare_lengths [1; 2] [3; 4] ;;
- : int = 0
```

However, this implementation of `compare_lengths` does a little extra work than it needs to. Its complexity is  $O(n)$  where  $n$  is the length of the *longer* of the two lists.

Why does `compare_lengths` have this big- $O$  complexity? In particular, why does the length of the shorter list not play a part in the complexity? We're looking for a brief informal argument here, not a full derivation of its complexity.

Provide an alternative implementation of `compare_lengths` whose complexity is  $O(n)$  where  $n$  is the length of the *shorter* of the two lists, not the longer.

<sup>7</sup> For reference, this built-in `length` function is, unsurprisingly, linear in the length of its argument.

#### 14.5.3 Complexity of reversing a list with accumulator

An alternative method of reversing a list uses an accumulator. As each element in the list is processed, it is consed on the front of the accumulating list. The process begins with the empty accumulator.

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

# let rev xs = revappend xs [] ;;
val rev : 'a list -> 'a list = <fun>
```

As before, we can set up recurrence equations for this version of `rev` and its auxiliary function `revappend`.

$$T_{rev}(n) = q + T_{revapp}(n, 0)$$

$$T_{revapp}(0, m) = c$$

$$T_{revapp}(n+1, m) = k + T_{revapp}(n, m+1)$$

By an unfolding argument similar to that for append, we can solve these recurrence equations to closed form:

$$\begin{aligned} T_{revapp}(n, m) &= k \cdot n + c \\ &\in O(n) \end{aligned}$$

so that

$$\begin{aligned} T_{rev}(n) &= q + T_{revapp}(n, 0) \\ &= q + k \cdot n + c \\ &\in O(n) \end{aligned}$$

Unlike the quadratic simple reverse, the revappend approach is linear. The difference is born out empirically as well, as shown in Figure 14.7.

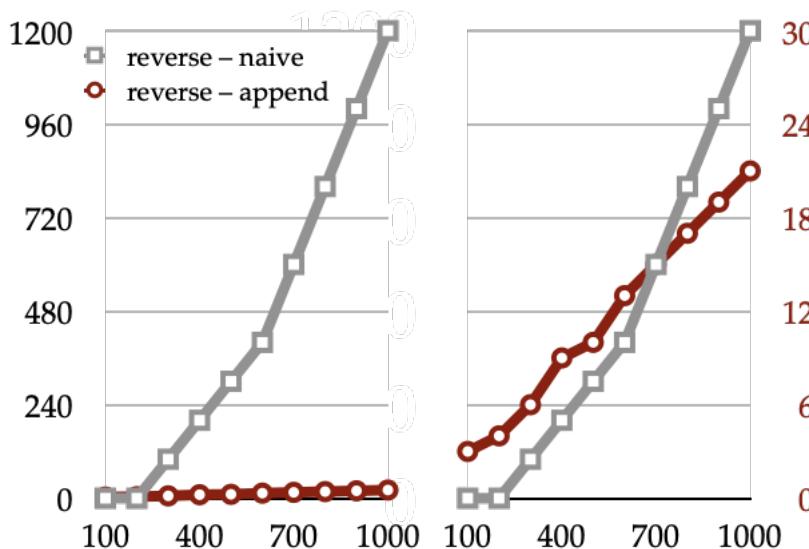


Figure 14.7: Time in microseconds to reverse lists of lengths 100 to 1000 using the naive (square) and revappend (circle, highlighted) implementations. The left graph places both lines on the same (left) vertical scale. The right graph places the revappend line on the right vertical scale (equivalent to multiplying all of the revappend times by 50) to emphasize the difference in growth rate of the functions. Despite the change in constants, the naive reverse still eventually overtakes the revappend.

#### 14.5.4 Complexity of inserting in a sorted list

The insertion sort algorithm uses a function `insert` to insert an element in its place in a sorted list:

```
# let rec insert xs x =
#   match xs with
#   | [] -> [x]
#   | hd :: tl -> if x > hd then hd :: (insert tl x)
#                   else x :: xs;;
val insert : 'a list -> 'a -> 'a list = <fun>
```

As usual, we construct appropriate recurrence equations for  $T_{\text{insert}}(n)$  where  $n$  is the length of the list being inserted into. (We ignore the element argument, as its size is irrelevant to the time required.) Inserting into the empty list takes constant time.

$$T_{\text{insert}}(0) = c$$

Inserting into a nonempty list (of size  $n + 1$ ) is more subtle. The time required depends on whether the element should come at the start of the list (the *else* clause of the conditional) or not (the *then* clause). In the former case, the `cons` operation takes constant time, say  $k_2$ ; in the latter case, it involves a recursive call to `insert` ( $T_{\text{insert}}(n)$ ) plus some further constant overhead ( $k_1$ ). Since we don't know which way the computation will branch, we have to make the worst-case assumption: whichever is bigger. Which of the two is bigger depends on the constants, but we can be sure, in any case, that the time required is certainly less than the sum of the two.

$$\begin{aligned} T_{\text{insert}}(n + 1) &= \max(k_1 + T_{\text{insert}}(n), k_2) \\ &\leq k_1 + T_{\text{insert}}(n) + k_2 \\ &= k + T_{\text{insert}}(n) \end{aligned}$$

Unfolding these proceeds as usual:

$$\begin{aligned} T_{\text{insert}}(n) &= k + T_{\text{insert}}(n - 1) \\ &= k + k + T_{\text{insert}}(n - 2) \\ &= \dots \\ &= k \cdot n + T_{\text{insert}}(0) \\ &= k \cdot n + c \\ &\in O(n) \end{aligned}$$

Insertion is thus linear in the size of the list to be inserted into.

#### 14.5.5 Complexity of insertion sort

Recall the implementation of insertion sort:

```
let rec sort (lt : 'a -> 'a -> bool)
            (xs : 'a list)
            : 'a list =
  match xs with
  | [] -> []
  | hd :: tl -> insert lt (sort lt tl) hd ;;
```

Using similar arguments as above, the recurrence equations can be determined to be

$$T_{\text{isort}}(0) = c$$

$$T_{\text{isort}}(n + 1) = k + T_{\text{isort}}(n) + T_{\text{insert}}(n)$$

Solving the recurrence equations:

$$\begin{aligned}
 T_{isort}(n) &= k + T_{isort}(n - 1) + O(n - 1) \\
 &= k + k + T_{isort}(n - 2) + O(n - 1) + O(n - 2) \\
 &= k \cdot n + T_{isort}(0) + O(n - 1) + O(n - 2) + \dots + O(0) \\
 &= k \cdot n + c + \sum_{i=1}^n O(i) \\
 &\in O(n^2)
 \end{aligned}$$

We conclude that insertion sort is quadratic in its run time.

#### 14.5.6 Complexity of merging lists

Continuing our exploration of the time complexity of sorting algorithms, we turn to the components of mergesort. The `merge` function, defined by

```

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) ;;

```

takes two list arguments; their sizes will be two of the arguments of the complexity function  $T_{merge}$ . Each recursive call of `merge` reduces the total number of items in the two lists. We will for that reason use the sum of the sizes of the two lists as the argument to  $T_{merge}$ .

If the total number of elements in the two lists is 1, then one of the two lists must be empty, and we have

$$T_{merge}(1) = c$$

In the worst case, neither element will become empty until the total number of elements in the lists is 2. Thus, for  $n \geq 2$ , we have the “normal” case, when the lists are nonempty, which involves (in addition to some constant overhead) a recursive call to `merge` with one fewer element in the lists. In the worst case, both elements will still be nonempty.

$$T_{merge}(n + 1) = k + T_{merge}(n)$$

Solving these recurrence equations:

$$\begin{aligned}
 T_{\text{merge}}(n) &= k + T_{\text{merge}}(n - 1) \\
 &= k + k + T_{\text{merge}}(n - 2) \\
 &= \dots \\
 &= k \cdot n + T_{\text{merge}}(1) \\
 &= k \cdot n + c \\
 &\in O(n)
 \end{aligned}$$

#### 14.5.7 Complexity of splitting lists

We leave as an exercise to show that the `split` function defined by

```

let rec split lst =
  match lst with
  | []
  | [_] -> lst, []
  | first :: second :: rest ->
    let first', second' = split rest in
    first :: first', second :: second' ;;
  
```

has linear time complexity.

**Exercise 149**

Show that `split` has time complexity linear in the size of its first list argument.

#### 14.5.8 Complexity of divide and conquer algorithms

Before continuing to the analysis of mergesort, we look more generally at algorithms that (like mergesort) attack problems by dividing them into equal parts, recursively solving them, and putting the subsolutions back together to solve the full problem – DIVIDE-AND-CONQUER algorithms.

The recurrences of such algorithms are typically structured with a base case requiring constant time

$$T(1) = c$$

and a recursive case that involves two recursive calls on some problems each of half the size. At first, we'll assume that the time to break apart and put together the two parts takes constant time  $k$ .

$$T(n) = k + 2 \cdot T(n/2)$$

For simplicity in solving these recurrence equations, we assume that

$n$  is a power of 2. Then unfolding a few times:

$$\begin{aligned} T(n) &= k + 2 \cdot T(n/2) \\ &= k + k + 4 \cdot T(n/4) \\ &= k + k + k + 8 \cdot T(n/8) \\ &= \dots \end{aligned}$$

How many times can we unfold? The denominator keeps doubling. We can keep doubling, then,  $m$  times until  $2^m = n$ , that is,  $m = \log n$ :

$$\begin{aligned} T(n) &= k + 2 \cdot T(n/2) \\ &= k + k + 4 \cdot T(n/4) \\ &= k + k + k + 8 \cdot T(n/8) \\ &= \dots \\ &= k \cdot \log n + n \cdot T(n/n) \\ &= k \cdot \log n + c \cdot n \\ &\in O(n) \end{aligned} \quad \left. \right\} \text{log } n \text{ times}$$

More realistically, however, the time required to divide the problem up and to merge the subsolutions together may take time linear in the size of the problem. In that case, the recurrence would be something like

$$T(n) = k \cdot n + 2 \cdot T(n/2)$$

and the closed form is derived as

$$\begin{aligned} T(n) &= k \cdot n + 2 \cdot T(n/2) \\ &= k \cdot n + k \cdot n + 4 \cdot T(n/4) \\ &= k \cdot n + k \cdot n + k \cdot n + 8 \cdot T(n/8) \\ &= \dots \\ &= k \cdot n \cdot \log n + n \cdot T(n/n) \\ &= k \cdot n \cdot \log n + c \cdot n \\ &\in O(n \log n) \end{aligned} \quad \left. \right\} \text{log } n \text{ times}$$

The  $O(n \log n)$  complexity is the hallmark of divide-and-conquer algorithms. Since  $\log n$  grows extremely slowly, such algorithms are almost linear in their complexity, thus very efficient.

#### 14.5.9 Complexity of mergesort

Having determined the time complexity for the components of mergesort, we put them together to determine the complexity of the mergesort function itself:

```
let rec msort xs =
  match xs with
  | [] -> xs
  | [_] -> xs
  | _ -> let fst, snd = split xs in
            merge (msort fst) (msort snd) ;;
```

$$T_{msort}(0) = T_{msort}(1) = c_1$$

$$T_{msort}(n) = c_2 + T_{split}(n) + 2 \cdot T_{msort}(n/2) + T_{merge}(n)$$

Since both  $T_{split}$  and  $T_{merge}$  are linear, we can write

$$T_{msort}(n) = k \cdot n + c + 2 \cdot T_{msort}(n/2)$$

These recurrence equations are just of the divide-and-conquer sort, so we know immediately that the complexity of mergesort is  $O(n \log n)$ .

And since

$$n^2 \gg n \log n$$

mergesort is shown to be asymptotically more efficient than insertion sort.

Consistent with this analysis of the sorting algorithms is their empirical performance, as shown in Figure 14.4. The figure depicts well the almost linear behavior of mergesort and the much steeper quadratic growth of insertion sort.

#### 14.5.10 Basic Recurrence patterns

Table 14.1 summarizes some of the basic types of recurrence equations and their closed-form solution in terms of big- $O$ .

|                                         |                              |
|-----------------------------------------|------------------------------|
| $T(n) = c + T(n - 1)$                   | $T(n) \in O(n)$              |
| $T(n) = c + k \cdot n + T(n - 1)$       | $T(n) \in O(n^2)$            |
| $T(n) = c + k \cdot n^d + T(n - 1)$     | $T(n) \in O(n^{d+1})$        |
| $T(n) = c + 2 \cdot T(n/2)$             | $T(n) \in O(n)$              |
| $T(n) = c + T(n/2)$                     | $T(n) \in O(\log n)$         |
| $T(n) = c + k \cdot n + 2 \cdot T(n/2)$ | $T(n) \in O(n \cdot \log n)$ |

Table 14.1: Some common recurrence patterns and their closed-form solution in terms of big- $O$ .

### 14.6 Problem section: Complexity of the Luhn check

Recall the Luhn check algorithm from Section 8.5, and its various component functions: `evens`, `odds`, `doublemod9`, `sum`.

#### Problem 150

What is an appropriate recurrence equation for defining the time complexity of the `odds` function from Problem 60 in terms of the length of its list argument?

#### Problem 151

What is the time complexity of the `odds` function from Problem 60 (in big-*O* notation)?

#### Problem 152

If the function  $f(n)$  is the time complexity of `odds` on a list of  $n$  elements, which of the following is true?

- $f \in O(1)$
- $f \in O(\log n)$
- $f \in O(\log n/c)$  for all  $c > 0$
- $f \in O(c \cdot \log n)$  for all  $c > 0$
- $f \in O(n)$
- $f \in O(n/c)$  for all  $c > 0$
- $f \in O(c \cdot n)$  for all  $c > 0$
- $f \in O(n^2)$
- $f \in O(n^2/c)$  for all  $c > 0$
- $f \in O(c \cdot n^2)$  for all  $c > 0$
- $f \in O(2^n)$
- $f \in O(2^n/c)$  for all  $c > 0$
- $f \in O(c \cdot 2^n)$  for all  $c > 0$

#### Problem 153

What is the time complexity of the `luhn` function implemented in Problem 63 in terms of the length  $n$  of its list argument? Use big-*O* notation. Explain why your implementation has that complexity.

### 14.7 Supplementary material

- Lab 10: Time complexity, big-*O*, and recurrence equations
- Problem set A.6: The search for intelligent solutions

# 15

## *Mutable state and imperative programming*

The range of programming abstractions presented so far – first-order and higher-order functions; strong, static typing; polymorphism; algebraic data types; modules and functors – all fall squarely within a view of functional programming that we might term **PURE**, in which computation is identified solely with the evaluation of expressions.

Pure programming has to do with what expressions *are*, not what they *do*. Pure programs have *values* rather than *effects*. Indeed, the slightly pejorative term **SIDE EFFECT** is used in the functional programming literature for effects that impure programs manifest while they are being evaluated beyond their values themselves.

In a pure functional programming language, there are no side effects. Computation can be thought of as simplifying expressions to their values by repeated substitution of equals for equals. Because this notion of program meaning is so straightforward, functional programs are easier to reason about. Hopefully, the preceding chapters have shown that the functional paradigm is also more powerful than you might have thought.

Strictly speaking, however, pure functional programming is pointless. We write code to have an *effect* on the world. It might be pretty to think that “side effects” aren’t the main point. But they’re the main point.

Take this simple computation of the twentieth Fibonacci number:

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

# fib 20 ;;
- : int = 10946
```

The computation of `fib 20` proceeds purely functionally – at least until that very last step where the OCaml REPL *prints out the computed value*. Printing is *the* quintessential side effect. It’s a thing that

a program does, not a value that a program has. Without that one side effect, the `fib` computation would be useless. We'd gain no information from it.

So we need at least a *little* impurity in any programming system. But there are some algorithms that actually require impurity, in the form of side effects that change state. For instance, we've seen implementation of a dictionary data type in Chapter 12. That implementation allowed for linear time insertion and linear time lookup. More efficient implementations allow for constant time insertion and linear lookup (or vice versa) or for logarithmic insertion and lookup. But by taking advantage of side effects that change state, we can implement mutable dictionaries that achieve constant time insertion and constant time lookup, for instance, with hash tables. (In fact, we do so in Section 15.6.)

In this chapter and the next, we introduce IMPERATIVE PROGRAMMING, a programming paradigm based on side effects and state change. We start with mutable data structures, moving on to imperative control structures in the next chapter.

In the pure part of OCaml, we don't change the state of the computation, as encoded in the computer's memory. In languages that have mutable state, variables name blocks of memory whose contents can change. Assigning a new value to such a variable mutates the memory, changing its state by replacing the original value with the new one. OCaml variables, by contrast, aren't mutable. They name values, and once having named a value, the value named doesn't change.

You might think that OCaml does allow changing the value of a variable. What about, for instance, a global renaming of a variable?

```
# let x = 42 ;;
val x : int = 42
# x ;;           (* x is 42 *)
- : int = 42
# let x = 21 ;;
val x : int = 21
# x ;;           (* ...but now it's 21 *)
- : int = 21
```

Hasn't the value of `x` changed from 42 to 21?

No, it hasn't. Rather, there are two separate variables that happen to both have the same name, `x`. In the second expression, we are referring to the first `x` variable. In the fourth expression, we are referring to the second `x` variable, which shadows the first one. But the first `x` is still there. We can tell by the following experiment:

```
# let x = 42 ;;  (* establishing first x... *)
val x : int = 42
# x ;;           (* ...whose value is 42 *)
- : int = 42
```

```

# let f () = x ;; (* f returns value in first x *)
val f : unit -> int = <fun>
# let x = 21 ;; (* establishing second x... *)
val x : int = 21
# x ;; (* ...with a different value *)
- : int = 21
# f () ;; (* but f still references first x *)
- : int = 42

```

The definition of the function `f` makes use of the first variable `x`, simply by returning its value when called. Even if we add a new `x` naming a different value, the application `f ()` still returns 42, the value that the first variable `x` names, thereby showing that the first `x` is still available.

The `let` naming constructs of OCaml thus don't provide for mutable state. If we want to make use of mutable state, for instance for the purpose of building mutable data structures, we'll need new constructs. OCaml provides references for this purpose.

### 15.1 References

The OCaml language provides an abstract notion of REFERENCE to a block of mutable memory with its REFERENCE TYPES. To maintain the type discipline of the language, we want to keep track of the type of thing stored in the block; although the particular value stored there may change, we don't want its type to vary. Thus, we have separate types for references to integers, references to strings, references to functions from booleans to integers, and so forth. The postfix type constructor `ref` is used to construct reference types: `int ref`, `string ref`, `(bool -> int) ref`, and the like.

To create a value of some reference type, OCaml provides the prefix value constructor `ref`.<sup>1</sup> The supplied expression must be of the type appropriate for the reference type, and the value of that expression is stored as the initial value in the block of memory that the reference references. Here, for instance, we create a reference to a block of memory storing the integer value 42:

```

# let r : int ref = ref 42 ;;
val r : int ref = {contents = 42}

```

As with all variables, `r` is an immutable name, but it is a name for a block of memory that is itself mutable. (The value is printed as `{contents = 42}` for reasons that we allude to in Section 15.2.1.)

The natural operations to perform on a reference value are two: first, DEREference, that is, retrieve the value stored in the referenced block; and second, UPDATE, modify the value stored in the referenced block (with a value of the same type, of course). Dereferencing is done with the prefix `!` operator, and updating with the infix `:=` operator.

<sup>1</sup> Yes, the same symbol, `ref`, is used at the type level for the type constructor and at the value level for the value constructor. And to make matters more confusing, the type constructor is postfix while the value constructor is prefix. Learning the concrete syntax of a new programming language sure can be frustrating.

```
# !r ;;
- : int = 42
# r := 21 ;;
- : unit = ()
# !r ;;
- : int = 21
```

Here, we've dereferenced the same variable `r` twice (in the two highlighted expressions), getting two different values – first 42, then 21. This is quite different from the example with two `x` variables. Here, there is only one variable `r`, and yet a single expression `!r` involving `r` whose value has changed!<sup>2</sup>

This example puts in sharp relief the difference between the pure language and the impure. In the pure language, an expression in a given lexical context (that is, the set of variable names that are available) always evaluates to the same value. But in this example, two instances of the expression `!r` evaluate to two different values, even though the same `r` is used in both instances of the expression. The assignment has the side effect of changing what value is stored in the block that `r` references, so that reevaluating `!r` to retrieve the stored value finds a different integer.

The expression causing the side effect here was easy to spot. But in general, these side effects could happen as the result of a series of function calls quite obscure from the code that manifests the side effect. This property of side effects can make it difficult to reason about what value an expression has.

In particular, the substitution semantics of Chapter 13 has Leibniz's law as a consequence. Substitution of equals for equals doesn't change the value of an expression. But here, we have a clear counterexample. The first evaluation implies that `!r` and 42 are equal. Yet if we substitute 42 for `!r` in the third expression, we get 42 instead of 21. Once we add mutable state to the language, we need to extend the semantics from one based purely on substitution. We do so in Chapter 19, where we introduce environment semantics.

### 15.1.1 Reference operator types

The reference system is specifically designed so as to retain OCaml's strong typing regimen. Each of the operators, for instance, can be seen as a well-typed function. The dereference operator `!`, for instance, takes an argument of type `'a ref` and returns the `'a` referenced. It is thus typed as `(!) : 'a ref -> 'a`. The reference value constructor `ref` works in the opposite direction, taking an `'a` and returning an `'a ref`, so it types as `(ref) : 'a -> 'a ref`.

Finally, the assignment operator `:=` takes two arguments, a refer-

<sup>2</sup> But like all variables, `r` has not itself changed its value. It still points to the same block of memory.

ence to update, of type '`a ref`', and the new '`a`' value to store there. But what should the assignment operator return? Assignment is performed entirely for its side effect – the update in the state of memory – rather than for its return value. Given that there is no information in the return value, it makes sense to use a type that conveys no information. This is a natural use for the `unit` type (Section 4.3). Since `unit` has only one value (namely, the value `()`), that value conveys no information. The hallmark of a function that is used only for its side effects (which we might call a PROCEDURE) is the `unit` return type. The typing for assignment is appropriately then `(:=) : 'a ref -> 'a -> unit`.

These typings can be verified in OCaml itself:

```
# (!) ;;
- : 'a ref -> 'a = <fun>
# (ref) ;;
- : 'a -> 'a ref = <fun>
# (:=) ;;
- : 'a ref -> 'a -> unit = <fun>
```

### 15.1.2 Boxes and arrows

It can be helpful to visualize references using BOX AND ARROW DIAGRAMS. When establishing a reference,

```
# let r = ref 42 ;;
val r : int ref = {contents = 42}
```

we draw a box (standing for a block of memory) named `r` with an arrow pointing to another box (block of memory) containing the integer 42 (Figure 15.1(a)). Adding another reference with

```
# let s = ref 42 ;;
val s : int ref = {contents = 42}
```

generates a new named box and its referent (Figure 15.1(b)), which happens to store the same value. But we can tell that the referents are distinct, since assigning to `r` changes `!r` but not `!s` (Figure 15.1(c)).

```
# r := 21 ;;
- : unit = ()
# !r, !s ;;
- : int * int = (21, 42)
```

To have `s` refer to the value that `r` does, we need to assign to it as well (Figure 15.1(d)).

```
# s := !r ;;
- : unit = ()
```

We can have a reference `s` that points to the *same* block of memory as `r` does (Figure 15.1(e)).

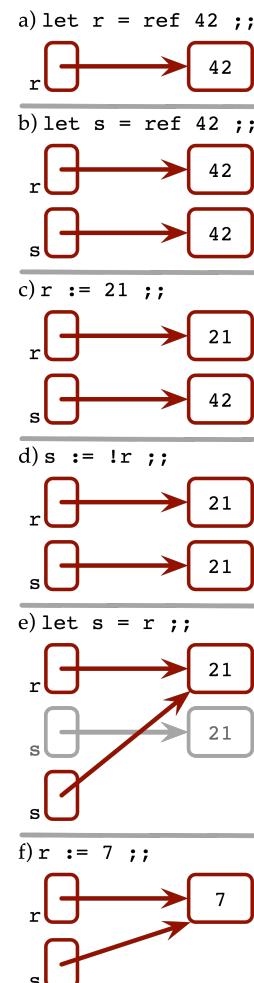


Figure 15.1: Box and arrow diagrams for the state of memory as various references are created and updated.

```
# let s = r ;;
val s : int ref = {contents = 21}
```

Now `s` and `r` have the same value (that is, refer to the same block of memory). We say that `s` is an ALIAS of `r`. (The old `s` is shadowed by the new one, as depicted by showing it in gray. Since we no longer have access to it and whatever it references, the gray blocks of memory are garbage. See the discussion in Section 15.1.3.)

Changing the value stored in a block of memory changes the value of all its aliases as well. Here, updating the block referred to by `r` (Figure 15.1(f)) changes the value for `s`:

```
# r := 7 ;;
- : unit = ()
# !r, !s ;;
- : int * int = (7, 7)
```

In a language with references and aliases, we are confronted with two different notions of equality. STRUCTURAL EQUALITY holds when two values have the same structure, regardless of where they are stored in memory, such as `r` and `s` in Figure 15.1(d). PHYSICAL EQUALITY holds when two values are the identical “physical” block of memory, as `r` and `s` in Figure 15.1(e). Values that are physically equal are of course structurally equal as well but the converse needn’t hold.

In OCaml, structural equality and inequality are tested with `(=)` : `'a -> 'a -> bool` and `(<>)` : `'a -> 'a -> bool`, respectively, whereas physical equality and inequality of mutable types are tested with `(==)` : `'a -> 'a -> bool` and `(!=)` : `'a -> 'a -> bool`.<sup>3</sup>

#### Exercise 154

Construct an example defining values `r` and `s` that are structurally but not physically equal. Construct an example defining values `r` and `s` that are both structurally and physically equal. Verify these conditions using the equality functions.

<sup>3</sup>The behavior of `==` and `!=` tests on immutable (pure) types is allowed to be implementation-dependent and shouldn’t be relied on. These operators should only be used with values of mutable types.

### 15.1.3 References and pointers

You may have seen this kind of thing before. In programming languages like c, references to blocks of memory are manipulated through POINTERS to memory, which are explicitly created (with `malloc`) and freed (with `free`), dereferenced (with `*`), and updated (with `=`). Some correspondences between OCaml and c syntax for these operations are given in Table 15.1.

Notable differences between the OCaml and c approaches are:

- In OCaml, unlike in c, references can’t be created without initializing them. Referencing uninitialized blocks of memory is a recipe for difficult to diagnose bugs. OCaml’s type regime eliminates this entire class of bugs, since a reference type like `int ref` specifies

| <i>Operation</i>         | <i>OCaml</i>                | <i>c</i>                                               |
|--------------------------|-----------------------------|--------------------------------------------------------|
| Create, initialize       | <code>ref 42</code>         |                                                        |
| Create, name             |                             | <code>int *r = malloc(sizeof int);</code>              |
| Create, initialize, name | <code>let r = ref 42</code> | <code>int *r = malloc(sizeof int);<br/>*r = 42;</code> |
| Dereference              | <code>!r</code>             | <code>*r</code>                                        |
| Update                   | <code>r := 21</code>        | <code>*r = 21</code>                                   |
| Free                     |                             | <code>free(r)</code>                                   |

that the block must at all times store an `int` and the operators maintain this invariant.

- In c, nothing conspires to make sure that the size of the block allocated is appropriate for the value being stored, leading to the possibility of BUFFER OVERFLOWS – assignments that overflow one block of memory to overwrite others. Buffer overflows allow for widely exploited security holes in code. In OCaml, the strong typing again eliminates this class of bug. Similarly, BUFFER OVER-READS occur when a program reading from a block of memory continues to read past the end of the block into adjacent memory, potentially compromising the security of information in the adjacent block. An infamous example is the HEARTBLEED bug in OpenSSL, so notorious that it even acquired its own logo (Figure 15.2).
- In c, programs must free memory explicitly in order to reclaim the previously allocated memory for future use. When blocks are freed while still being used, the memory can be overwritten, leading to MEMORY CORRUPTION and once again to insidious bugs. Conversely, not freeing blocks even when they are no longer needed, called a MEMORY LEAK, leads to programs running out of memory needlessly.

OCaml has no function for explicitly freeing memory. Instead, blocks of memory that are no longer needed, as determined by the OCaml run-time system itself, are referred to as GARBAGE. The run-time system reclaims garbage automatically, in a process called GARBAGE COLLECTION. Since computers can typically analyze the status of memory blocks better than people, the use of garbage collection eliminates memory corruption and memory leaks.

However, the garbage collection approach takes the *timing* of memory reclamation out of the hands of the programmer. The run-time system may decide to perform computation-intensive garbage collection at inopportune times. For applications where careful control of such timing issues is necessary, the garbage collection approach

Table 15.1: Approximate equivalencies between OCaml references and c pointers.



Figure 15.2: The logo for HEARTBLEED, a buffer over-read bug in the widely used OpenSSL library (written in c) for securing web interactions. The bug was revealed in 2014 after two years undiscovered in the field.

may be undesirable; use of a language, like C, that allows explicit allocation and deallocation of memory may be necessary.<sup>4</sup>

#### Problem 155

For each of the following expressions, give its type and value, if any.

1. 

```
let a = ref 3 in
let b = ref 5 in
let a = ref b in
  (!a) ;;
```
2. 

```
let rec a, b = ref b, ref a in
  !a ;;
```
3. 

```
let a = ref 1 in
let b = ref a in
let a = ref 2 in
  (!b) ;;
```
4. 

```
let a = 2 in
let f = (fun b -> a * b) in
let a = 3 in
  f (f a) ;;
```

<sup>4</sup> A new class of functional programming languages is exploring the design space of languages with high-level abstraction mechanisms as in OCaml, including strongly typed safe references, while providing finer control of memory deallocation, in order to obtain the best of both the explicit approach and the garbage collection approach. The prime example is Mozilla's Rust language.

## 15.2 Other primitive mutable data types

In addition to references, OCaml provides two other primitive data types that allow for mutability: mutable record fields and arrays. We mention them briefly for completeness; full details are available in the OCaml documentation.

### 15.2.1 Mutable record fields

Records (Section 7.4) are compound data structures with named fields, each of which stores a value of a particular type. As introduced, each field of a record, and hence records themselves, are immutable. However, when a record type is defined with the `type` construct, and the individual fields are specified and typed, its individual fields can also be marked as allowing mutability by adding the keyword `mutable`.

For instance, we can define a `person` record type with immutable name fields but a mutable address field.

```
# type person = {lastname : string;
#                 firstname : string;
#                 mutable address : string} ;;
type person = {
  lastname : string;
  firstname : string;
  mutable address : string;
}
```

Once constructed, the address of a person can be updated.

```
# let sms = {lastname = "Shieber";
#             firstname = "Stuart";
#             address = "123 Main"} ;;
val sms : person =
```

```
{lastname = "Shieber"; firstname = "Stuart"; address = "123
Main"}
# sms.address <- "124 Main" ;; (* I moved next door *)
- : unit = ()
```

To update a mutable record, the operator `<-` is used, rather than `:=` as for references.

In fact, reference types and their operators can be thought of as being implemented using mutable records by the following type and operator definitions:

```
type 'a ref_ = {mutable contents : 'a} ;;

let ref_ (v : 'a) : 'a ref_ = {contents = v} ;;
let (:=) (r : 'a ref_) (v : 'a) : unit = r.contents <- v ;;
let (!) (r : 'a ref_) : 'a = r.contents ;;
```

This should explain the otherwise cryptic references to `contents` when the REPL prints values of reference type.

### 15.2.2 Arrays

Arrays are a kind of cross between lists and tuples with added mutability. Like lists, they can have an arbitrary number of elements all of the same type. Unlike lists (but like tuples), they cannot be extended in size; there is no `cons` equivalent for arrays. Finally, each element of an array can be individually indexed and updated. An example may indicate the use of arrays:

```
# let a = Array.init 5 (fun n -> n * n) ;;
val a : int array = [|0; 1; 4; 9; 16|]
# a ;;
- : int array = [|0; 1; 4; 9; 16|]
# a.(3) <- 0 ;;
- : unit = ()
# a ;;
- : int array = [|0; 1; 4; 0; 16|]
```

Here, we've created an array of five elements, each the square of its index. We update the third element to be 0, and examine the result, which now has a 0 in the appropriate location.

### 15.3 References and mutation

To provide an example of the use of mutating references, we consider the task of counting the occurrences of an event. We start by establishing a location to store the current count as an `int ref` named `gctr` (for “global counter”).

```
# let gctr = ref 0 ;;
val gctr : int ref = {contents = 0}
```

Now we define a function that “bumps” the counter (adding 1) and then returns the current value of the counter.

```
# let bump () =
#   gctr := !gctr + 1;
#   !gctr ;;
val bump : unit -> int = <fun>
```

We've used a new operator here, the binary sequencing operator ( ; ), which is a bit like the pair operator ( , ) in that it evaluates its left and right arguments, except that the sequencing operator returns the value only of the second.<sup>5</sup> But then what could possibly be the point of evaluating the first argument? Since the argument isn't used for its value, it must be of interest for its side effects. That is the case in this example; the expression `gctr := !gctr + 1` has the side effect of updating the counter to a new value, its old value (retrieved with `!gctr`) plus one.<sup>6</sup> Since the sequencing operator ignores the value returned by its first argument, it requires that argument to be of type `unit`, the type for expressions with no useful value.<sup>7</sup>

We can test it out.

```
# bump () ;;
- : int = 1
# bump () ;;
- : int = 2
# bump () ;;
- : int = 3
```

Again, you see the hallmark of impure code – the same expression in the same context evaluates to different values. The change between invocations happens because of the side effects of the earlier calls to `bump`. We can see evidence of the side effects also in the value of the counter, which is globally visible.

```
# !gctr ;;
- : int = 3
```

In the case of the `bump` function, it is the intention to provide these side effects. They are what generates the counting functionality. However, it is not necessarily the intention to make the current counter visible to users of the `bump` function. Doing so enables unintended side effects, like manipulating the value stored in the counter outside of the manipulation by the `bump` function itself, enabling misuses such as the following:

```
# gctr := -17 ;;
- : unit = ()
# bump () ;;
- : int = -16
```

<sup>5</sup> You can think of `P ; Q` as being syntactic sugar for `let () = P in Q`.

<sup>6</sup> This part of the `bump` function that does the actual incrementing of an `int ref` is a common enough activity that OCaml provides a function `incr : int ref -> unit` in the `Stdlib` library for just this purpose. It works as if implemented by

```
let incr (r : int ref) : unit =
  r := !r + 1;;
```

We could therefore have substituted `incr gctr` as the second line of the `bump` function.

<sup>7</sup> Sometimes, you may want to sequence an expression that returns a value other than `()`. The `ignore` function of type `'a -> unit` in `Stdlib` comes in handy in such cases.

To eliminate this abuse we'd like to avoid a global variable for the counter. We've seen this kind of information hiding before – in the use of local variables within functions, and in the use of signatures to hide auxiliary values and functions from users of modules, all instances of the edict of compartmentalization. But in the context of assignment, making `gctr` a local variable (we'll call it `ctr`) requires some thought. A naive approach doesn't work:

```
# let bump () =
#   let ctr = ref 0 in
#   ctr := !ctr + 1;
#   !ctr ;;
val bump : unit -> int = <fun>
```

#### Exercise 156

What goes wrong with this definition? Try using it a few times and see what happens.

The problem: This code establishes the counter variable `ctr` upon *application* of `bump`, and establishes a new such variable at each such application. Instead, we want to define `ctr` just once, upon the *definition* of `bump`, and not its applications.

In this case, the compact notation for function definition, which conflates the defining of the function and its naming, is doing us a disservice. Fortunately, we aren't obligated to use that syntactic sugar.

We can use the desugared version:

```
let bump =
  fun () ->
    ctr := !ctr + 1;
    !ctr ;;
```

Now the naming (first line) and the function definition (second line and following) are separate. We want the definition of `ctr` to outscope the function definition but fall within the local scope of its naming:

```
# let bump =
#   let ctr = ref 0 in
#   fun () ->
#     ctr := !ctr + 1;
#     !ctr ;;
val bump : unit -> int = <fun>
```

The function is defined within the scope of – and therefore can access and modify – a local variable `ctr` whose scope is *only* that function.

This definition operates as before to deliver incremented integers:

```
# bump () ;;
- : int = 1
# bump () ;;
- : int = 2
# bump () ;;
- : int = 3
```

but access to the counter variable is available only within the function, as it should be, and not outside of it:

```
# !ctr ;;
Line 1, characters 1-4:
1 | !ctr ;;
    ^
Error: Unbound value ctr
Hint: Did you mean gctr?
```

This example – the counter with local, otherwise inaccessible, persistent, mutable state – is one of the most central to understand. We'll see a dramatic application of this simple pattern in Chapter 18, where it underlies the idea of instance variables in object-oriented programming.

#### Problem 157

Suppose you typed the following OCaml expressions into the OCaml REPL sequentially.

```
1 let p = ref 11;;
2 let r = ref p;;
3 let s = ref !r;;
4 let t =
5   !s := 14;
6   !p + !(!r) + !(!s);;
7 let t =
8   s := ref 17;
9   !p + !(!r) + !(!s);;
```

Try to answer the questions below about the status of the various variables being defined before typing them into the REPL yourself.

1. After line 1, what is the type of p?
2. After line 2, what is the type of r?
3. After line 3, which of the following statements are true?
  - (a) p and s have the same type
  - (b) r and s have the same type
  - (c) p and s have the same value (in the sense that p = s would be true)
  - (d) r and s have the same value (in the sense that r = s would be true)
4. After line 6, what is the value of t?
5. After line 9, what is the value of t?

#### 15.4 Mutable lists

To demonstrate the power of imperative programming, we use OCaml's imperative aspects to provide implementations of two mutable data structures: mutable lists and mutable queues.

As noted in Section 11.1, the OCaml list type operates as if defined by

```
type 'a list =
| Nil
| Cons of 'a * 'a list;;
```

A mutable list allows values constructed in this way to be updated; we thus take values of type `'a list` to be references to such compound structures.

```

# type 'a mlist = 'a mlist_internal ref
#  and 'a mlist_internal =
#    | Nil
#    | Cons of 'a * 'a mlist;;
type 'a mlist = 'a mlist_internal ref
and 'a mlist_internal = Nil | Cons of 'a * 'a mlist

```

(In this mutually recursive pair of type definitions, the intention is to make use of values of type '`'a mlist`'. The auxiliary type '`'a mlist_internal`' is just an expedient, required because OCaml needs a name for the type of values that references refer to.)

The shortest such mutable list is simply a reference to the `Nil` value (in this case, coerced to an integer mutable list).

```

# let r : int mlist = ref Nil;;
val r : int mlist = {contents = Nil}

```

We can build longer mutable lists by consing on a couple of integers. We'll do that bit by bit to allow naming of the intermediate lists.

```

# let s : int mlist = ref (Cons (1, r));;
val s : int mlist = {contents = Cons (1, {contents = Nil})}
# let t : int mlist = ref (Cons (2, s));;
val t : int mlist =
{contents = Cons (2, {contents = Cons (1, {contents = Nil})})}

```

We can compute the length of such a list using the usual recursive definition.

```

# let rec mlength (lst : 'a mlist) : int =
#   match lst with
#   | Nil -> 0
#   | Cons (_hd, tl) -> 1 + mlength tl;;
val mlength : 'a mlist -> int = <fun>

```

Comparing this with the definition of `length : 'a list -> int` from Section 7.3.1, the only difference here is the dereferencing of the mutable list before it can be matched. Sure enough, this definition works on the example mutable lists above.

```

# mlength r;;
- : int = 0
# mlength s;;
- : int = 1
# mlength t;;
- : int = 2

```

Box and arrow diagrams (Figure 15.3) help in figuring out what's going on here.

#### Exercise 158

Write functions `mhead` and `mtail` that extract the head and the (dereferenced) tail from a mutable list. For example,

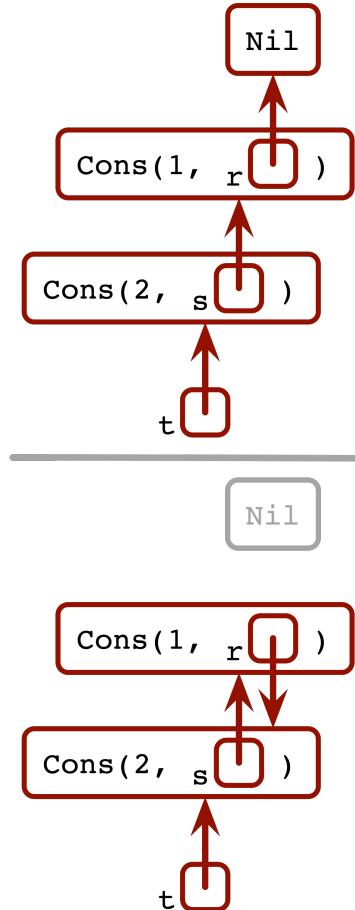


Figure 15.3: Pictorial representation of (top) the state of memory after building some mutable list structures, and (bottom) updating with `r := !t`. The `nil` has become garbage and the mutable lists `r`, `s`, and `t` now have cycles in them.

```
# mhead t ;;
- : int = 2
# mtail t ;;
- : int mlist = {contents = Cons (1, {contents = Nil})}
```

Because the lists are mutable, we can modify the tail of `s` (that is, `r`) to point to `t`.

```
# r := !t ;;
- : unit = ()
```

Since the tail of `s` points to `t` and the tail of `t` to `s`, we've constructed a CYCLIC data structure. Doing so uncovers a bug in the `mlength` function,

```
# mlength t ;;
Stack overflow during evaluation (looping recursion?).
```

demonstrating once again how adding impure features to a language introduces new and quite subtle complexities.

### Problem 159

For each of the following expressions, give its type and value, if any.

1. let `a` = ref (Cons (2, ref (Cons (3, ref Nil)))) ;;
2. let Cons (\_hd, tl) = !`a` in  
let `b` = ref (Cons (1, `a`)) in  
`tl` := !`b` ;  
`mhead` (`mtail` (`mtail` `b`)) ;;

### Problem 160

Provide an implementation of the `mlength` function that handles cyclic lists, so that

```
# mlength t ;;
- : int = 3
```

You'll notice that the requirement to handle cyclic lists dramatically increases the complexity of implementing `length`. (Hint: Keep a list of sublists you've already visited and check to see if you've already visited each sublist. What is a reasonable value to return in that case?)

### Problem 161

Define a function `mfist` : `int -> 'a mlist -> 'a list` that returns a list (immutable) of the first `n` elements of a mutable list:

### Problem 162

Write code to define a mutable integer list `alternating` such that for all integers `n`, the expression `mfist n alternating` returns a list of alternating 1s and 2s, for example,

```
# mfist 5 alternating ;;
- : int list = [1; 2; 1; 2; 1]
# mfist 8 alternating ;;
- : int list = [1; 2; 1; 2; 1; 2; 1; 2]
```

## 15.5 Imperative queues

By way of review, the pure functional queue data structure in Section 12.4 implemented the following signature:

```
# module type QUEUE = sig
#   type 'a queue
#   val empty_queue : 'a queue
#   val enqueue : 'a -> 'a queue -> 'a queue
```

```

#   val dequeue : 'a queue -> 'a * 'a queue
# end ;;
module type QUEUE =
sig
  type 'a queue
  val empty_queue : 'a queue
  val enqueue : 'a -> 'a queue -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end

```

Each call to enqueue and dequeue returns a new queue, differing from its argument queue in having an element added or removed.

In an imperative implementation of queues, the enqueueing and dequeuing operations can and do mutate the data structure, so that the operations don't need to return an updated queue. The types for the operations thus change accordingly. We'll use the following IMP\_QUEUE signature for imperative queues:

```

# module type IMP_QUEUE = sig
#   type 'a queue
#   val empty_queue : unit -> 'a queue
#   val enqueue : 'a -> 'a queue -> unit
#   val dequeue : 'a queue -> 'a option
# end ;;
module type IMP_QUEUE =
sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val enqueue : 'a -> 'a queue -> unit
  val dequeue : 'a queue -> 'a option
end

```

Here again, you see the sign of a side-effecting operation: the enqueue operation returns a unit. Dually, to convert a procedure that modifies its argument and returns a unit into a pure function, the standard technique is to have the function return instead a modified copy of its argument, leaving the original untouched. Indeed, when we generalize the substitution semantics of Chapter 13 to handle state and state change in Chapter 19, we will use just this technique of passing a representation of the computation state as an argument and returning a representation of the updated state as the return value.

Another subtlety introduced by the addition of mutability is the type of the empty\_queue value. In the functional signature, we had empty\_queue : 'a queue; the empty\_queue value was an empty queue. In the mutable signature, we have empty\_queue : unit -> 'a queue; the empty\_queue value is a function that returns a (new, physically distinct) empty queue. Without this change, the empty\_queue value would be “poisoned” as soon as something was inserted in it, so that further references to empty\_queue would see the modified

(non-empty) value. Instead, the `empty_queue` function can generate a new empty queue each time it is called.

### 15.5.1 Method 1: List references

Perhaps the simplest method to implement an imperative queue is as a (mutable) reference to an (immutable) list of the queue's elements.

```
# module SimpleImpQueue : IMP_QUEUE =
  struct
    type 'a queue = 'a list ref
    let empty_queue () = ref []
    let enqueue elt q =
      q := (!q @ [elt])
    let dequeue q =
      match !q with
      | first :: rest -> (q := rest; Some first)
      | [] -> None
  end ;;
module SimpleImpQueue : IMP_QUEUE
```

This is basically the same as the list implementation from Section 12.4, but with the imperative signature. Nonetheless, internally the operations are still functional, and enqueueing an element requires time linear in the number of elements in the queue. (Recall from Section 14.5 that the functional append function (here invoked as `Stdlib.(@)`) is linear.)

We'll examine two methods for generating constant time implementations of an imperative queue.

### 15.5.2 Method 2: Two stacks

An old trick is to use two stacks to implement a queue. The two stacks hold the front of the queue (the first elements in, and hence the first out) and the reversal of the rear of the queue. For example, a queue containing the elements 1 through 4 in order might be represented by the two stacks (implemented as int lists) [1; 2] and [4; 3], or pictorially as in Figure 15.4 (upper left).

Enqueuing works by adding an element (5 in upper right) to the *rev rear* stack. Dequeueing works by popping the top element in the *front* stack, if there is one (middle right and left and lower right). If there are no elements to dequeue in the *front* stack (middle left), the *rev rear* stack is reversed onto the *front* stack first (lower left).

The stacks can be implemented with type `'a list ref` and the two stacks packaged together in a record.

```
module TwoStackImpQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a list ref;
```

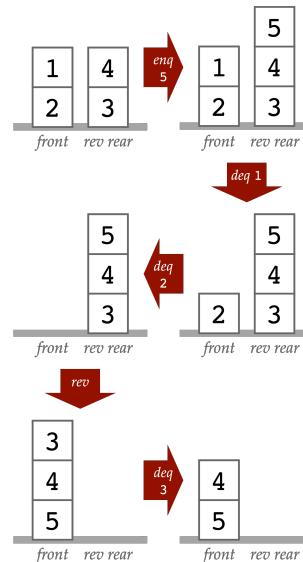


Figure 15.4: Pictorial representation of implementing a queue with two stacks.

```

    revrear : 'a list ref}
...

```

The empty queue has two empty lists.

```

module TwoStackImpQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a list ref;
                      revrear : 'a list ref}
    let empty_queue () =
      {front = ref []; revrear = ref []}
...

```

Enqueuing simply places the element on the top of the rear stack.

```

module TwoStackImpQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a list ref;
                      revrear : 'a list ref}
    let empty_queue () =
      {front = ref []; revrear = ref []}
    let enqueue elt q =
      q.revrar := elt :: !(q.revrar)
...

```

Dequeuing is the more complicated operation.

```

# module TwoStackImpQueue : IMP_QUEUE =
#   struct
#     type 'a queue = {front : 'a list ref;
#                       revrear : 'a list ref}
#     let empty_queue () =
#       {front = ref []; revrear = ref []}
#     let enqueue elt q =
#       q.revrar := elt :: !(q.revrar)
#     let rec dequeue q =
#       match !(q.front) with
#       | h :: t -> (q.front := t; Some h)
#       | [] -> if !(q.revrar) = [] then None
#                 else ((* reverse revrear onto front *)
#                         q.front := List.rev !(q.revrar));
#                         (* clear revrear *)
#                         q.revrar := [];
#                         (* try the dequeue again *)
#                         dequeue q)
#     end ;;
module TwoStackImpQueue : IMP_QUEUE

```

As in method 1, the enqueue operation takes constant time. But dequeuing usually takes constant time too, unless we have to perform the reversal of the rear stack. Since the stack reversal takes time linear in the number of enqueues, the time to enqueue and dequeue elements is, on average, constant time per element.

**Exercise 163**

An alternative is to use mutable record fields, so that the queue type would be

```
type 'a queue = {mutable front : 'a list;
                 mutable revrear : 'a list}
```

Reimplement the `TwoStackImpQueue` module using this type for the queue implementation.

*15.5.3 Method 3: Mutable lists*

To allow for manipulation of both the head of the queue (where enqueueing happens) and the tail (where dequeuing happens), a final implementation uses mutable lists. The queue type

```
module MutableListQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a mlist;
                     rear : 'a mlist}
    ...
  
```

provides a reference to the front of the queue as well as a reference to the last cons in the queue if there is one. When the queue is empty, both of these lists will be `ref Nil`.

```
module MutableListQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a mlist;
                     rear : 'a mlist}
    let empty_queue () = {front = ref Nil;
                          rear = ref Nil}
    ...
  
```

Enqueuing a new element differs depending on whether the queue is empty. If it already contains at least one element, the rear will have a head and a `ref Nil` tail (because the rear always points to the last cons).

```
module MutableListQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a mlist;
                     rear : 'a mlist}
    let empty_queue () = {front = ref Nil;
                          rear = ref Nil}
    let enqueue elt q =
      match !(q.rear) with
      | Cons (hd, tl) -> (assert (!tl = ref Nil);
                            tl := Cons (elt, ref Nil);
                            q.rear := !tl)
      | Nil -> ...
  
```

If the queue is empty, we establish a single element mutable list with front and rear pointers to its single element.

```

module MutableListQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a mlist;
                     rear : 'a mlist}
    let empty_queue () = {front = ref Nil;
                          rear = ref Nil}
    let enqueue elt q =
      match !(q.rear) with
      | Cons (hd, tl) -> (assert (!tl = Nil);
                            tl := Cons (elt, ref Nil);
                            q.rear := !tl)
      | Nil -> (assert (!(q.front) = Nil);
                  q.front := Cons (elt, ref Nil);
                  q.rear := !(q.front))
    ...
  
```

Finally, dequeuing involves moving the front pointer to the next element in the list, and updating the rear to Nil if the last element was dequeued and the queue is now empty.

```

# module MutableListQueue : IMP_QUEUE =
#   struct
#     type 'a queue = {front : 'a mlist;
#                      rear : 'a mlist}
#
#     let empty_queue () = {front = ref Nil;
#                           rear = ref Nil}
#     let enqueue elt q =
#       match !(q.rear) with
#       | Cons (_hd, tl) -> (assert (!tl = Nil);
#                             tl := Cons (elt, ref Nil);
#                             q.rear := !tl)
#       | Nil -> (assert (!(q.front) = Nil);
#                  q.front := Cons (elt, ref Nil);
#                  q.rear := !(q.front))
#     let dequeue q =
#       match !(q.front) with
#       | Cons (hd, tl) ->
#         (q.front := !tl;
#          (match !tl with
#           | Nil -> q.rear := Nil
#           | Cons (_, _) -> ());
#           Some hd)
#       | Nil -> None
#     end ;
#   module MutableListQueue : IMP_QUEUE
  
```

Figure 15.5 depicts the queue data structure as it performs the following operations:

```

# let open MutableListQueue in
# let q = empty_queue () in
# enqueue 1 q;
# enqueue 2 q;
  
```

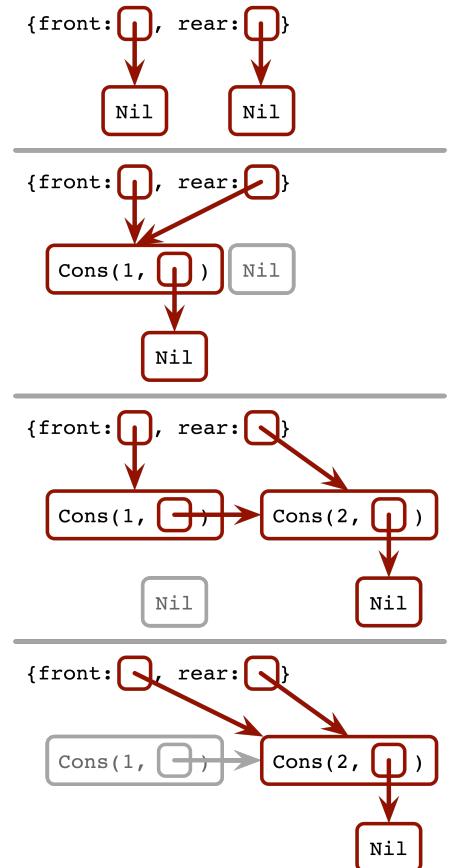


Figure 15.5: Pictorial representation of implementing a queue with a mutable list.

```
# dequeue q ;;
- : int option = Some 1
```

## 15.6 Hash tables

A hash table is a data structure implementing a mutable dictionary.

We've seen functional key-value dictionaries already in Section 12.6, which implement a signature like the following:

```
module type DICT =
  sig
    type key
    type value
    type dict

    (* An empty dictionary *)
    val empty : dict
    (* Returns as an option the value associated with the
       provided key. If the key is not in the dictionary,
       returns None. *)
    val lookup : dict -> key -> value option
    (* Returns true if and only if the key is in the
       dictionary. *)
    val member : dict -> key -> bool
    (* Inserts a key-value pair into the dictionary. If the
       key is already present, updates the key to have the
       new value. *)
    val insert : dict -> key -> value -> dict
    (* Removes the key from the dictionary. If the key is
       not present, returns the original dictionary. *)
    val remove : dict -> key -> dict
  end ;;
```

In a mutable dictionary, the data structure state is actually modified by side effect when inserting or removing key-value pairs. Consequently, those functions need not (and should not) return an updated dictionary. (As with mutable lists, because dictionaries can be modified by side effect, care must also be taken with specifying an empty dictionary. Instead of a single empty dictionary value, we provide a function from unit that returns a new empty dictionary.) An appropriate signature for a mutable dictionary, then, is

```
# module type MDICT =
#   sig
#     type key
#     type value
#     type dict
#
#     (* Returns an empty dictionary. *)
#     val empty : unit -> dict
#     (* Returns as an option the value associated with the
#        provided key. If the key is not in the dictionary,
```

```

#      returns None. *)
#  val lookup : dict -> key -> value option
#  (* Returns true if and only if the key is in the
#     dictionary. *)
#  val member : dict -> key -> bool
#  (* Inserts a key-value pair into the dictionary. If the
#     key is already present, updates the key to have the
#     new value. *)
#  val insert : dict -> key -> value -> unit
#  (* Removes the key from the dictionary. If the key is
#     not present, leaves the original dictionary unchanged. *)
#  val remove : dict -> key -> unit
# end ;;
module type MDICT =
sig
  type key
  type value
  type dict
  val empty : unit -> dict
  val lookup : dict -> key -> value option
  val member : dict -> key -> bool
  val insert : dict -> key -> value -> unit
  val remove : dict -> key -> unit
end

```

In a HASH TABLE implementation of this signature, the key-value pairs are stored in a mutable array of a given size at an index specified by a HASH FUNCTION, a function from keys to integers within the range provided. The idea is that the hash function should assign well distributed locations to keys, so that inserting or looking up a particular key-value pair involves just computing the hash function to generate the location where it can be found. Thus, insertion and lookup are constant-time operations.

An important problem to resolve is what to do in case of a HASH COLLISION, when two different keys hash to the same value. We assume that only a single key-value pair can be stored at a given location in the hash table – called CLOSED HASHING – so in case of a collision when inserting a key-value pair, we keep searching in the table at the sequentially following array indices until an empty slot in the table is found. Similarly, when looking up a key, if the key-value pair stored at the hash location does not match the key being looked up, we sequentially search for a pair that does match. This process of trying sequential locations is known as LINEAR PROBING. Frankly, linear probing is not a particularly good method for handling hash collisions (see Exercises 165 and 166), but it will do for our purposes here.

To define a new kind of hash table, we need to provide types for the keys and values, a size for the array, and an appropriate hash function. We package all of this up in a module that can serve as the argument to

a functor.

```
# module type MDICT_ARG =
#   sig
#     (* Types to be used for the dictionary keys and values *)
#     type key
#     type value
#     (* size -- Number of elements that can be stored in the
#        dictionary *)
#     val size : int
#     (* hash_fn key -- Returns the hash value for a key. *)
#     val hash_fn : key -> int
#   end ;;
module type MDICT_ARG =
  sig type key type value val size : int val hash_fn : key -> int
end
```

Here is the beginning of an implementation of such a functor:

```
module MakeHashtableDict (D : MDICT_ARG)
  : (MDICT with type key = D.key
             and type value = D.value) =
struct
  type key = D.key
  type value = D.value

  (* A hash record is a key value pair *)
  type hashrecord = { key : key;
                      value : value }

  (* An element of the hash table array is a hash record
     (or empty) *)
  type hashelement =
    | Empty
    | Element of hashrecord
  (* The hash table itself is a (mutable) array of hash
     elements *)
  type dict = hashelement array

  let empty () = Array.make D.size Empty
  ...
end ;;
```

With a full implementation of the `MakeHashtableDict` functor (Exercise 164), we can build an `IntStringHashtbl` hash table module for hash tables that map integers to strings as follows:<sup>8</sup>

```
# module IntStringHashtbl : (MDICT with type key = int
                                 and type value = string) =
#   MakeHashtableDict (struct
#     type key = int
#     type value = string
#     let size = 100
#     let hash_fn k = (k / 3) mod size
#   end) ;;
module IntStringHashtbl :
```

<sup>8</sup>The hash function we use here is an especially poor choice; we use it to make it easy to experiment with hash collisions.

```

sig
  type key = int
  type value = string
  type dict
  val empty : unit -> dict
  val lookup : dict -> key -> value option
  val member : dict -> key -> bool
  val insert : dict -> key -> value -> unit
  val remove : dict -> key -> unit
end

```

Let's experiment:

```

# open IntStringHashtbl;;
# let d = empty () ;;
val d : IntStringHashtbl.dict = <abstr>
# insert d 10 "ten" ;;
- : unit = ()
# insert d 9 "nine" ;;
- : unit = ()
# insert d 34 "34" ;;
- : unit = ()
# insert d 1000 "a thousand" ;;
- : unit = ()
# lookup d 10 ;;
- : IntStringHashtbl.value option = Some "ten"
# lookup d 9 ;;
- : IntStringHashtbl.value option = Some "nine"
# lookup d 34 ;;
- : IntStringHashtbl.value option = Some "34"
# lookup d 8 ;;
- : IntStringHashtbl.value option = None
# remove d 9 ;;
- : unit = ()
# lookup d 10 ;;
- : IntStringHashtbl.value option = Some "ten"
# lookup d 9 ;;
- : IntStringHashtbl.value option = None
# lookup d 34 ;;
- : IntStringHashtbl.value option = Some "34"
# lookup d 8 ;;
- : IntStringHashtbl.value option = None

```

#### Exercise 164

Complete the implementation by providing implementations of the remaining functions `lookup`, `member`, `insert`, and `remove`.

#### Exercise 165

Improve the collision handling in the implementation by allowing the linear probing to “wrap around” so that if it reaches the end of the array it keeps looking at the beginning of the array.

#### Exercise 166

A problem with linear probing is that as collisions happen, contiguous blocks of the array get filled up, so that further collisions tend to yield long searches to get past

these blocks for an empty location. Better is to use a method of rehashing that leaves some gaps. A simple method to do so is QUADRATIC PROBING: each probe increases quadratically, adding 1, then 2, then 4, then 8, and so forth. Modify the implementation so that it uses quadratic probing instead of linear probing.

### 15.7 Conclusion

With the introduction of references, we move from thinking about what expressions *mean* to what they *do*. The ability to mutate state means that data structures can now undergo change. By modifying existing data structures, we may be able to avoid building new copies, thereby saving some space. More importantly, performing small updates may be much faster than constructing large copies, leading to improvements in both space and time complexity.

But making good on these benefits requires much more subtle reasoning about what programs are up to. The elegant substitution model – which says that expressions are invariant under substitution of one subexpression by another with the same value – doesn't hold when side effects can change those values out from under us. Aliasing means that changes in one part of the code can have ramifications far afield. Modifying data structures means that the hierarchical structures can be modified to form cycles, with the potential to fall into infinite loops. (We explore the changes needed to the substitution semantics of Chapter 13 to allow for mutable state in Chapter 19.)

Nonetheless, the underlying structure of modern computer hardware is based on stateful memory to store program and data, so that at some point imperative programming is a necessity. Imperative programming can be a powerful way of thinking about implementing functionality.



We've now introduced essentially all of the basic language constructs that we need. In the following chapters, we deploy them in new combinations that interact to provide additional useful programming abstractions – providing looping constructs to enable the repetition of side effects (Chapter 16); the ability to perform a computation “lazily”, delaying it until its result is needed (Chapter 17); and the encapsulation of computations within data objects that they act on (Chapter 18).

### 15.8 Supplementary material

- Lab 12: Imperative programming and references

# 16

## *Loops and procedural programming*

Back in Section 7.3.1, we implemented a function to compute the length of a list, by capturing how the length is *defined*: the length of the empty list is 0; the length of a non-empty list is one more than the length of its tail. This definition can be immediately cashed out as

```
# let rec length (lst : 'a list) : int =
#   match lst with
#   | [] -> 0
#   | _hd :: tl -> 1 + length tl ;;
val length : 'a list -> int = <fun>
```

An alternative approach, in the spirit of imperative programming, is to think not about what the length *is* but about what one *does* when calculating the length: For each element of the list, add one to a counter until the end of the list is reached.

This approach – which we might term PROCEDURAL PROGRAMMING because it emphasizes the steps in the procedure to be carried out – is typical of how introductory programming is taught, with an emphasis on *commands* with *side effects* that are executed repeatedly through *loops*.

In this chapter, we'll provide examples of procedural programming, emphasizing one of the main benefits of the paradigm, SPACE EFFICIENCY. Procedural programming can be more space efficient in a couple of ways. First, it can reduce the need for storing suspended computations in so-called “stack frames”, though as we'll see, the functional language technique of tail-recursion optimization can provide this benefit as well. Second, it can reduce the need for copying data structures as they are manipulated.

Although OCaml is at its core a functional programming language, it supports procedural programming as well. There are, for instance,

`while` loops:

```
 $\langle \text{expr} \rangle ::= \text{while } \langle \text{expr}_{\text{condition}} \rangle \text{ do}$ 
     $\langle \text{expr}_{\text{body}} \rangle$ 
    done
```

which specify that the body expression be executed repeatedly so long as the condition expression is `true`.

In addition, the `for` loop, familiar from other procedural languages, is expressed as follows to count up from a start value to an end value:

```
 $\langle \text{expr} \rangle ::= \text{for } \langle \text{var} \rangle = \langle \text{expr}_{\text{start}} \rangle \text{ to } \langle \text{expr}_{\text{end}} \rangle \text{ do}$ 
     $\langle \text{expr}_{\text{body}} \rangle$ 
    done
```

or, counting down,

```
 $\langle \text{expr} \rangle ::= \text{for } \langle \text{var} \rangle = \langle \text{expr}_{\text{start}} \rangle \text{ downto } \langle \text{expr}_{\text{end}} \rangle \text{ do}$ 
     $\langle \text{expr}_{\text{body}} \rangle$ 
    done
```

### 16.1 Loops require impurity

In a pure language, an expression in a given context always has the same value. Thus, in a `while` loop of the form

```
while  $\langle \text{expr}_{\text{condition}} \rangle$  do
     $\langle \text{expr}_{\text{body}} \rangle$ 
done
```

if the condition expression  $\langle \text{expr}_{\text{condition}} \rangle$  is `true` the first time it's evaluated, it will remain so perpetually and the loop will never terminate. Conversely, if the condition expression is `false` the first time it's evaluated, it will remain so perpetually and the loop body will never be evaluated. Similarly, the body expression  $\langle \text{expr}_{\text{body}} \rangle$  will always evaluate to the same value, so what could possibly be the point of evaluating it more than once?

In summary, procedural programming only makes sense in a language with *side effects*, the kind of impure constructs (like variable assignment) that we introduced in the previous chapter. You can see this need in attempting to implement the `length` function in this procedural paradigm. Here is a sketch of a procedure for calculating the length of a list:

```

let length (lst : 'a list) : int =
  (* initialize the counter *)
  while (* the list is not empty *) do
    (* increment the counter *)
    (* drop an element from the list *)
  done;
  (* return the counter *);;

```

We'll need to establish the counter in such a way that its value can change. Similarly, we'll need to update the list each time the loop body is executed. We'll thus need both the counter and the list being manipulated to be references, so that they can change. Putting all this together, we get the following procedure for computing the length of a list:

```

# let length_iter (lst : 'a list) : int =
#   let counter = ref 0 in      (* initialize the counter *)
#   let lst_ref = ref lst in    (* initialize the list *)
#   while !lst_ref <> [] do    (* while list not empty... *)
#     incr counter;           (* increment the counter *)
#     lst_ref := List.tl !lst_ref (* drop element from list *)
#   done;
#   !counter;;                (* return the counter value *)
val length_iter : 'a list -> int = <fun>

# length_iter [1; 2; 3; 4; 5] ;;
- : int = 5

```

## 16.2 Recursion versus iteration

Is this impure, iterative, procedural method better than the pure, recursive, functional approach? It certainly seems more complex, and gaining an understanding that it provides the correct values as specified in the definition of list length is certainly more difficult.

### 16.2.1 Saving stack space

There is one way, however, in which this approach might be superior. Think of the calculation of the length of a list, say [1; 2; 3], using the functional definition. Since the list is non-empty, we need to add one to the result of evaluating `length [2; 3]`, and we'll need to suspend the addition until that evaluation completes. Likewise, to evaluate `length [2; 3]` we'll need to add one to the result of evaluating `length [3]`, again suspending the addition until *that* evaluation completes. Continuing on in this way, at run time we'll eventually have a nested stack of suspended calls. Each element of this stack, carrying information about the suspended computation, is referred to as a STACK FRAME. Only once we reach `length []` can we start unwinding this stack, performing all of the suspended additions specified in

the stack frames, to calculate the final answer. Figure 16.1 depicts this linearly growing stack of suspended calls.

The iterative approach, on the other hand, needs no stack of suspended computations. The single call to `length_iter` invokes the `while` loop to iteratively increment the counter and drop elements from the list. The computation is “flat”.

The difference can be seen forcefully when computing the length of a very long list. Here, we’ve defined `very_long_list` to be a list with one million elements.

```
# let very_long_list = List.init 1_000_000 Fun.id ;;
val very_long_list : int list = [0; 1; 2; 3; 4; 5; 6; 7; ...]
```

The iterative procedure for computing its length works well.

```
# length_iter very_long_list ;;
- : int = 1000000
```

But the functional recursive version overflows the stack dedicated to storing the suspended computations. Apparently, one million stack frames is more than the computer has space for.

```
# length very_long_list ;;
Stack overflow during evaluation (looping recursion?).
```

### 16.2.2 Tail recursion

The profligate use of space for stack frames is not inherent in all purely functional recursive computations however. Consider the following purely functional method `length_tr` for implementing the length calculation.

```
# let length_tr lst =
#   let rec length_plus lst acc =
#     match lst with
#     | [] -> acc
#     | _hd :: tl -> length_plus tl (1 + acc) in
#   length_plus lst 0 ;;
val length_tr : 'a list -> int = <fun>
```

Here, a local auxiliary function `length_plus` takes two arguments, the list and an integer accumulator of the count of elements counted so far. It returns *the length of its list argument plus the value of its accumulator*. Thus, the call to `length_plus lst 0` calculates the the length of `lst` plus 0, which is just the length desired.

This `length_tr` version of calculating list length still operates recursively; `length_plus` is the locus of the recursion as indicated by the `rec` keyword. The nesting of recursive calls proceeds as shown in Figure 16.2.

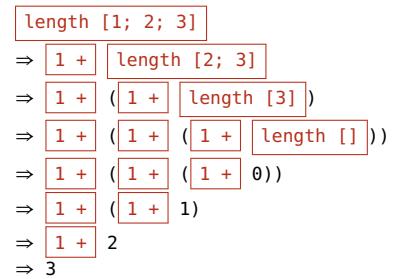


Figure 16.1: The nested stack of suspended calls in evaluating a non-tail-recursive `length` function. We indicate each stack frame with a highlighted box. Notice that the number of stack frames increases as each recursive call is generated.

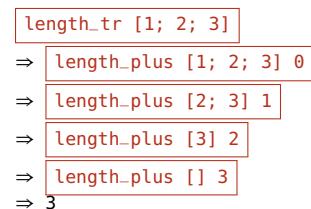


Figure 16.2: The call structure in evaluating a tail-recursive `length` function. Note the lack of nesting of suspended calls.

As with the previous recursive version, the number of such recursive computations is linear in the length of the list. One might think, then, that the same problem of stack overflow will haunt the `length_tr` implementation as well. Let's try it.

```
# length_tr very_long_list ;;
- : int = 1000000
```

This version doesn't have the same problem. It's easy to see why. For the recursive `length`, the result of each call is a computation using the result of the embedded call to `length`; that computation must therefore be suspended, and a stack frame must be allocated to store information about that pending computation. But the result of each call to the recursive `length_plus` is not just a *computation using* the result of the embedded call to `length_plus`; it *is* the result of that nested call. We don't need to store any information about a suspended computation – no need to allocate a stack frame – because the embedded call result is all that is needed.

Recursive programs written in this way, in which every recursive invocation *is* the result of the invoking call, are deemed TAIL RECURSIVE (hence the `_tr` in the function's name). Tail-recursive functions need not use a stack to keep track of suspended computations. Programming language implementations that take advantage of this possibility by not allocating a stack frame to tail-recursive applications are said to perform TAIL-RECURSION OPTIMIZATION, effectively turning the recursion into a corresponding iteration, and yielding the benefits of the procedural iterative solution. The OCaml interpreter is such a language implementation.

Thus, this putative advantage of loop-based procedures over recursive functions – the ability to perform computations space-efficiently – can often be replicated in functional style through careful tail-recursive implementation where needed.

You'll see discussion of this issue, for instance, in the description of functions in the `List` library, which calls out those functions that are not tail-recursive.<sup>1</sup> For instance, the library function `fold_left` is implemented in a tail-recursive manner, so it can fold over very long lists without running out of stack space. By contrast, the `fold_right` implementation is not tail-recursive, so may not be appropriate when processing extremely long lists.

### 16.3 Saving data structure space

Another advantage of procedural programming is the ability to avoid building of new data structures. Think of the `map` function over lists, which can be implemented as follows:

<sup>1</sup> From the [List library documentation](#):  
“Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. . . . The above considerations can usually be ignored if your lists are not longer than about 10000 elements.”

```
# let rec map (fn : 'a -> 'b) (lst : 'a list) : 'b list =
#   match lst with
#   | [] -> []
#   | hd :: tl -> fn hd :: map fn tl ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

We can use `map` to increment the values in a list:

```
# let original = [1; 2; 3] ;;
val original : int list = [1; 2; 3]
# map succ original ;;
- : int list = [2; 3; 4]
```

The result is a list with different values. Most notably, the result is a new list. The `original` is unchanged.

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

The new list is created by virtue of the repeated construction of conses with the `::` operator highlighted in the `map` definition above. Every time `map` is called to operate over a list, more conses will be needed. There's no free lunch here. Under the hood, every cons takes up space; storage must be allocated for each one. If we start with a list of length  $n$ , we'll end up allocating  $n$  more conses to compute the map.

### 16.3.1 Problem section: Metering allocations

We can determine how many allocations are going on by metering them. Imagine there were a module `Metered` satisfying the following signature:

```
# module type METERED =
#   sig
#     (* reset () -- Resets the count of allocations *)
#     val reset : unit -> unit
#     (* count () -- Returns the number of allocations
#        since the last reset *)
#     val count : unit -> int
#     (* cons hd tl -- Returns the list cons of `hd` and
#        `tl`, increasing the allocation count accordingly *)
#     val cons : 'a -> 'a list -> 'a list
#     (* pair first second -- Returns the pair of `first`
#        and `second`, increasing the allocation count
#        accordingly *)
#     val pair : 'a -> 'b -> 'a * 'b
#   end ;;
module type METERED =
sig
  val reset : unit -> unit
  val count : unit -> int
  val cons : 'a -> 'a list -> 'a list
  val pair : 'a -> 'b -> 'a * 'b
end
```

The functions `cons` and `pair` could be used to replace their built-in counterparts for consing (`::`) and pairing `(,)` to track the number of allocations required.

**Problem 167**

Implement the module `Metered`.

**Problem 168**

Reimplement the `zip` function of Section 10.3.2 using metered conses and pairs.

Having metered the `zip` function, we can observe the count of allocations.

```
# Metered.reset () ;;
- : unit = ()
# zip [1; 2; 3; 4; 5] [5; 4; 3; 2; 1] ;;
- : (int * int) list = [(1, 5); (2, 4); (3, ...); ...]
# Metered.count () ;;
- : int = 10
```

### 16.3.2 Reusing space through mutable data structures

Now consider, by contrast to the functional `map` over lists above, a function (call it `map_array`) to map a function over a mutable data structure, an array. Instead of returning a new data structure, we'll mutate the values in the original data structure. For that reason, `map_array` doesn't itself need to return an array.<sup>2</sup>

```
# let map_array (fn : 'a -> 'a) (arr : 'a array) : unit =
#   for i = 0 to Array.length arr - 1 do
#     arr.(i) <- fn arr.(i)
#   done ;;
val map_array : ('a -> 'a) -> 'a array -> unit = <fun>
```

We can perform a similar computation, mapping the successor function over the elements of an array.

```
# let original = [|1; 2; 3|] ;;
val original : int array = [|1; 2; 3|]
# map_array succ original ;;
- : unit = ()
```

We see the effect of the `map` this time not in the return value but in the modified `original` array.

```
# original ;;
- : int array = [|2; 3; 4|]
```

By using imperative techniques, we gain access to the incremented values, and without incurring the cost of allocating further storage. There is a cost, however. We no longer have access to the original unincremented values. They've been destroyed, replaced by the new values. There's a tradeoff – reduced storage versus loss of access to prior results. Under what conditions the tradeoff is beneficial is a judgement call. But as an issue of efficiency, we'd want to heed Knuth's warning against premature optimization.

<sup>2</sup> The function being applied must be of type `'a -> 'a` since the output of the function is being stored in the same location as the input, and must thus be of the same type. For that reason, `map_array` can't be as polymorphic as `map`.

### 16.4 In-place sorting

As another example of the use of procedural programming to reduce storage requirements, we consider one of the most elegant sorting algorithms, **QUICKSORT**. Quicksort works by selecting a *pivot* value – the first element of the list, say – and partitioning the list into those elements less than the pivot and those that are greater. The two sublists are recursively sorted, and then concatenated to form the final sorted list. A recursive implementation of quicksort, following the **SORT** signature of Section 14.2, is as follows:

```
# module QuickSort : SORT =
#   struct
#     (* 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
#       | [] -> [], []
#       | hd :: tl ->
#         let first, second = partition lt pivot tl in
#         if lt hd pivot then hd :: first, second
#         else first, 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
#         (sort lt first) @ [pivot] @ (sort lt second)
#     end ;;
module QuickSort : SORT
```

#### Problem 169

Implement a metered version of quicksort, and experiment with how many allocations are needed to sort lists of different lengths.

Just as we built a version of `map` that mutated an array to map over its elements, we can build a version of quicksort that mutates an array to sort its elements. This approach, **IN-PLACE** sorting, is much more space-efficient. As we'll see, though, there is a cost in transparency of the implementation.

The type for an in-place sort differs from its pure alternative, which allocates extra space. A signature for an in-place sorting module makes clear the differences.

```
# module type SORT_IN_PLACE =
#   sig
```

```

#      (* sort lt xs -- Sorts the array `xs` in place in increasing
#         order by the "less than" function `lt`. *)
#      val sort : ('a -> 'a -> bool) -> 'a array -> unit
#    end;;
module type SORT_IN_PLACE =
sig val sort : ('a -> 'a -> bool) -> 'a array -> unit end

```

First, we're sorting a mutable data structure, an array, rather than a list. Second, the `sort` function returns a `unit` as it works by side effect rather than by returning a sorted version of the unchanged argument list. The sorting function, then, begins with a header line

```
let sort (lt : 'a -> 'a -> bool) (arr : 'a array) : unit =
```

The primitive operation of in-place sorting is the swapping of two elements in the array, specified by their indices. We'll make use of a function `swap` to perform this operation.

```

let swap (i : int) (j : int) : unit =
  let temp = arr.(i) in
  arr.(i) <- arr.(j);
  arr.(j) <- temp

```

We'll need to partition a *region* of the array, by which we mean a contiguous subportion of the array between two indices. For that purpose, we'll have a function `partition` that takes two indices (`left` and `right`) demarcating the region to partition (the elements between the indices inclusive). The `partition` function returns the index of the split point in the region, the position that marks the border between the left partition and the right partition where the pivot element resides. We note that for our purposes, there should and will always be at least two elements in the region; otherwise, no recursive sorting is necessary, hence no need to partition.

To partition the region, we select the leftmost element as the pivot. We keep a “current” index that moves from left to right as we process each element in the region. At the same time, we maintain a moving “border” index, again moving from left to right. At any point, all of the elements to the left of the border will be guaranteed to be less than the pivot value. Those between the border and the current index are greater than or equal to the pivot. Those to the right of the current index are yet to be processed. Eventually, when we've processed all elements, we swap the pivot element itself into the correct position at the border. Here's the implementation of this quite subtle process:

```

let partition (left : int) (right : int) : int =
  (* region has at least two elements *)
  assert (left < right);

```

```

(* select the pivot element to be the first element in
   the region *)
let pivot_val = arr.(left) in
(* all elements to the left of `border` are guaranteed
   to be strictly less than pivot value *)
let border = ref (left + 1) in
(* current element being partitioned, starting just
   after pivot *)
let current = ref (left + 1) in

(* process each element, moving those less than the
   pivot to before the border *)
while !current <= right do
  if lt arr.(!current) pivot_val then
    begin
      (* current should be left of pivot *)
      swap !current !border; (* swap into place at border *)
      incr border (* move border to the right to make room *)
    end;
  incr current
done

(* the split point is just to left of the border *)
let split = !border - 1 in
(* move pivot into place at the split point *)
swap left split;
(* return the split index *)
split

```

With the availability of the `partition` function, we can implement a function `sort_region` to sort a region, again picked out by two indices.

```

let rec sort_region (left : int) (right : int) : unit =
  if left >= right then ()
  else
    let split = partition left right in
    (* recursively sort left and right regions *)
    sort_region left (split - 1);
    sort_region (split + 1) right

```

Finally, to sort the entire array, we can sort the region between the leftmost and rightmost indices

```
sort_region 0 ((Array.length arr) - 1)
```

Putting this all together leads to the implementation shown in Figure 16.3. (We've placed the `swap` and `partition` functions within the `sort` function so that they are within the scope of (and can thus access) the `lt` and `arr` arguments of `sort`.)

You'll note that the in-place quicksort is considerably longer than the pure version. In part that is because of the much more detailed work that must be done in partitioning a region, maintaining complex

```

module QuickSort : SORT_IN_PLACE =
  struct
    let sort (lt : 'a -> 'a -> bool) (arr : 'a array) : unit =
      (* swap i j -- Update the `arr` array by swapping the
       elements at indices `i` and `j` *)
      let swap (i : int) (j : int) : unit =
        let temp = arr.(i) in
        arr.(i) <- arr.(j);
        arr.(j) <- temp in

      (* partition left right -- Partition the region of the
       `arr` array between indices `left` and `right`
       *inclusive*, returning the split point, that is, the
       index of the pivot element. Assumes the region
       contains at least two elements. At the end,
       everything to left of the split is less than the
       pivot; everything to the right is greater. *)
      let partition (left : int) (right : int) : int =
        (* region has at least two elements *)
        assert (left < right);

        (* select the pivot element to be the first element in
         the region *)
        let pivot_val = arr.(left) in
        (* all elements to the left of `border` are guaranteed
         to be strictly less than pivot value *)
        let border = ref (left + 1) in
        (* current element being partitioned, starting just
         after pivot *)
        let current = ref (left + 1) in

```

Figure 16.3: Implementation of an in-place quicksort.

```

(* process each element, moving those less than the
   pivot to before the border *)
while !current <= right do
  if lt arr.(!current) pivot_val then
    begin
      (* current should be left of pivot *)
      swap !current !border; (* swap into place *)
      incr border (* move border right to make room *)
    end;
  incr current
done;

(* the split point is just to left of the border *)
let split = !border - 1 in
(* move pivot into place at the split point *)
swap left split;
(* return the split index *)
split in

(* sort_region left right -- quicksort the subarray of
   the `arr` array between indices `left` and `right`
   *inclusive* *)
let rec sort_region (left : int) (right : int) : unit =
  if left >= right then ()
  else
    let split = partition left right in
    (* recursively sort left and right regions *)
    sort_region left (split - 1);
    sort_region (split + 1) right
  in

  (* sort the whole `arr` array *)
  sort_region 0 ((Array.length arr) - 1)
end

```

Figure 16.3: (continued) Implementation of an in-place quicksort.

invariants concerning the left, right, current, and border indices and the elements in the various subregions. In part the length is a result of considerably more documentation in the implementation, but that is not a coincidence. The implementation requires this additional documentation to be remotely as understandable as the pure version. (Even still, an understanding of the in-place version is arguably more complex. It's hard to imagine understanding how the partition function works without manually "playing computer" on some examples to verify the procedure.)

The payoff is that the in-place version needs to allocate only a tiny amount of space beyond the storage in the various stack frames for the function applications – just the storage for the current and border elements. Is the cost in code complexity and opaqueness worth it? That depends on the application. If sorting huge amounts of data is necessary, the reduction in space may be needed.

**Problem 170**

A completely in-place version of mergesort that uses only a fixed amount of extra space turns out to be quite tricky to implement. However, a version that uses only a single extra array is possible, and still more space-efficient than the pure version described in Section 14.2. Implement a version of mergesort that uses a single extra array as "scratch space" for use while merging. To sort a region, we sort the left and right subregions recursively, then merge the two into the scratch array, and finally copy the merged region back into the main array.

### 16.5 *Supplementary material*

- Lab 12: Procedural programming and loops



# 17

## *Infinite data structures and lazy programming*

Combining functions as first-class values, algebraic data types, and references enables programming with infinite data structures, the surprising topic of this chapter. We'll build infinite lists (streams) and infinite trees. The primary technique we use, lazy evaluation, has many other applications.

### 17.1 Delaying computation

OCaml is an EAGER language. Recall the semantic rule for function application from Chapter 13:

$$\begin{array}{c} P \ Q \Downarrow \\ \left| \begin{array}{l} P \Downarrow \text{fun } x \rightarrow B \\ Q \Downarrow v_Q \\ B[x \mapsto v_Q] \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{app})$$

According to this rule, before generating the result of the application (by substituting into the body expression  $B$ ), we *first* evaluate the argument  $Q$ . Similarly, in a local let expression,

$$\begin{array}{c} \text{let } x = D \text{ in } B \Downarrow \\ \left| \begin{array}{l} D \Downarrow v_D \\ B[x \mapsto v_D] \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{let})$$

before substituting the definition  $D$  into the body expression  $B$ , we *first* evaluate  $D$  to a value.

There are disadvantages of this eager evaluation approach. For instance, if the argument value is not used in the body of the function, the computation to generate the value will still be carried out, an entirely wasted effort. An extreme case occurs when the computation of the argument value doesn't even terminate:

```
# let rec forever n = 1 + forever n;;
val forever : 'a -> int = <fun>
# (fun x -> "this value ignores x") (forever 42) ;;
Line 1, characters 5-6:
1 | (fun x -> "this value ignores x") (forever 42) ;;
^
Warning 27 [unused-var-strict]: unused variable x.
Stack overflow during evaluation (looping recursion?).
```

If we had delayed the computation of `forever 42` until after it had been substituted in as the argument of the function, we would never have had to evaluate it at all, and the evaluation of the full expression would have terminated with "`this value ignores x`".

Examples like this indicate the potential utility of LAZY EVALUATION – being able to DELAY computation until such time as it is needed, at which time the computation can be FORCED to occur.

There are, in fact, constructs of OCaml that work lazily. The conditional expression `if <exprtest> then <exprtrue> else <exprfalse>` delays evaluation of `<exprtrue>` and `<exprfalse>` until after evaluating `<exprtest>`, and in fact will refrain from evaluating the unchosen branch of the conditional entirely. Thus the following computation terminates, even though the `else` branch, if it were evaluated, would not.

```
# if true then 3 else forever 42 ;;
- : int = 3
```

Another construct that delays computation is *the function itself*. The body of a function is not evaluated until the function is applied. If application is postponed indefinitely, the body is never evaluated. Thus the following “computation” terminates.

```
# fun () -> forever 42 ;;
- : unit -> int = <fun>
```

This latter approach provides a universal method for delaying and forcing computations: wrapping the computation in a function (delay), and applying the function (forcing) if and when we need the value. What should the argument to the function be? Its only role is to postpone evaluation, so there needn’t be a real datum as argument – just a `unit`. As noted above, we refer to this wrapping a computation in a function from `unit` as *delay* of the computation. Conversely, we *force* the computation when the delayed expression is applied to `unit` so as to carry out the computation and get the value.

Though OCaml is eager in its evaluation strategy (with the few exceptions noted), some languages have embraced lazy evaluation as the default, starting with Rod Burstall’s Hope language and finding its widest use in the Haskell language named after Haskell Curry (Figure 6.2).

We'll make use of lazy evaluation in perhaps the most counter-intuitive application, the creation and manipulation of infinite data structures. We start with the *stream*, a kind of infinite list.

## 17.2 Streams

Here's a new algebraic data type definition defining the STREAM.

```
# type 'a stream = Cons of 'a * 'a stream;;
type 'a stream = Cons of 'a * 'a stream
```

It may look familiar; it shares much in common with the algebraic type definition of the polymorphic list, from Section 11.1, except that it dispenses with the `Nil` marking the end of the list.

We can define some operations on streams, like taking the head or tail of a stream.

```
# let head (Cons (hd, _tl) : 'a stream) : 'a = hd;;
val head : 'a stream -> 'a = <fun>

# let tail (Cons (_hd, tl) : 'a stream) : 'a stream = tl;;
val tail : 'a stream -> 'a stream = <fun>
```

It's all well and good to have streams and functions over them, but how are we to build one? It looks like we have a chicken and egg problem, requiring a stream in order to create one. Nonetheless, we press on, building a stream whose head is the integer 1. We start with

```
let ones = Cons (1, ...);;
```

We need to fill in the `...` with an `int stream`, but where are we to find one? How about the `int stream` named `ones` itself?

```
# let ones = Cons (1, ones) ;;
Line 1, characters 20-24:
 1 | let ones = Cons (1, ones) ;;
          ^^^^
Error: Unbound value ones
```

Of course, that doesn't work, because the name `ones` isn't itself available in the definition. That requires a `let rec`.

```
# let rec ones = Cons (1, ones) ;;
val ones : int stream = Cons (1, <cycle>)
```

It works! And the operations on this stream work as well:

```
# head ones;;
- : int = 1
# head (tail ones);;
- : int = 1
# head (tail (tail ones));;
- : int = 1
```

Its head is one, as is the head of its tail, and the head of the tail of the tail. It seems to be an infinite sequence of ones!

What is going on here? How does the implementation make this possible? Under the hood, the components of an algebraic data type have implicit pointers to their values. When we define `ones` as above, OCaml allocates space for the cons without initializing it (yet) and connects the name `ones` to it. It then initializes the contents of the cons, the head and tail, a pair of hidden pointers. The head pointer points to the value 1, and the tail points to the cons itself. This explains where the notation `<cycle>` comes from in the REPL printing out the value. In any case, the details of how this behavior is implemented isn't necessary to make good use of it.

Not all such cyclic definitions are well defined however. Consider this definition of an integer `x`:

```
# let rec x = 1 + x;;
Line 1, characters 12-17:
1 | let rec x = 1 + x ;;
           ^^^^^^
Error: This kind of expression is not allowed as right-hand side of
`let rec'
```

We can allocate space for the integer and name it `x`, but when it comes to initializing it, we need more than just a pointer to `x`; we need its value. But that isn't yet defined, so the whole process fails and we get an error message.

### 17.2.1 Operations on streams

We can look to lists for inspiration for operations on streams – operations like map, fold, and filter. Here is a definition for map on streams, which we call `smap`:

```
# let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
#   match s with
#   | Cons (hd, tl) -> Cons (f hd, smap f tl) ;;
val smap : ('a -> 'b) -> 'a stream -> 'b stream = <fun>
```

or, alternatively, using our recent definitions of `head` and `tail`,

```
# let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
#   Cons (f (head s), smap f (tail s)) ;;
val smap : ('a -> 'b) -> 'a stream -> 'b stream = <fun>
```

Now, we map the successor function over the stream of ones to form a stream of twos.

```
# let twos = smap succ ones ;;
Stack overflow during evaluation (looping recursion?).
```

Of course, that doesn't work at all. We're asking OCaml to add one to each element in an infinite sequence of ones. Luckily, `smap` isn't tail recursive, so we blow the stack, instead of just hanging in an infinite loop. This behavior makes streams as currently implemented less than useful since there's little we can do to them without getting into trouble. If only the system were less eager about doing all those infinite number of operations, doing them only if it "needed to".

The problem is that when calculating the result of the map, we need to generate (and cons together) both the head of the list (`f (head s)`) and the tail of the list (`smap f (tail s)`). But the tail already involves calling `smap`.

Why isn't this a problem in *calling* regular recursive functions, like `List.map`? In that case, there's a base case that is eventually called.

Why isn't this a problem in *defining* regular recursive functions?  
Why is there no problem in defining, say,

```
let rec fact n =
  if n = 0 then 1
  else n * fact (pred n) ;;
```

Recall that this definition is syntactic sugar for

```
let rec fact =
  fun n ->
    if n = 0 then 1
    else n * fact (pred n) ;;
```

The name `fact` can be associated with a function that uses it because *functions are values*. The parts inside are not further evaluated, at least not until the function is *called*. In essence, a function delays the latent computation in its body until it is applied to its argument.

We can take advantage of that in our definition of streams by using functions to perform computations lazily. We achieve laziness by wrapping the computation in a function delaying the computation until such time as we need the value. We can then force the value by applying the function.

To achieve the delay of computation, we'll take a stream not to be a cons as before, but a delayed cons, a function from `unit` to the cons. Other functions that need access to the components of the delayed cons can force it as needed. We need a new type definition for streams, which will make use of a simultaneously defined auxiliary type `stream_internal`.<sup>1</sup>

```
# type 'a stream_internal = Cons of 'a * 'a stream
# and 'a stream = unit -> 'a stream_internal ;;
type 'a stream_internal = Cons of 'a * 'a stream
and 'a stream = unit -> 'a stream_internal
```

An infinite stream of ones is now defined as so:

<sup>1</sup> The `and` connective allows mutually recursive type definitions. Unfortunately, OCaml doesn't allow direct definition of nested types, like

```
type 'a stream = unit -> (Cons of 'a * 'a stream)
```

```
# let rec ones : int stream =
#   fun () -> Cons (1, ones) ;;
val ones : int stream = <fun>
```

Notice that it returns a delayed cons, that is, a function which, when applied to a unit, returns the cons.

We need to redefine the functions accordingly to take and return these new lazy streams. In particular, `head` and `tail` now force their argument by applying it to unit.

```
# let head (s : 'a stream) : 'a =
#   match s () with
#   | Cons (hd, _tl) -> hd ;;
val head : 'a stream -> 'a = <fun>

# let tail (s : 'a stream) : 'a stream =
#   match s () with
#   | Cons (_hd, tl) -> tl ;;
val tail : 'a stream -> 'a stream = <fun>

# let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
#   fun () -> Cons (f (head s), smap f (tail s)) ;;
val smap : ('a -> 'b) -> 'a stream -> 'b stream = <fun>
```

The `smap` function now returns a lazy stream, a function, so that the recursive call to `smap` isn't immediately evaluated (as it was in the old definition). Only when the cons is needed (as in the `head` or `tail` functions) is the function applied and the cons constructed. That cons itself has a stream as its tail, but that stream is also delayed.

Now, finally, we can map the successor function over the infinite stream of ones to form an infinite stream of twos.

```
# let twos = smap succ ones ;;
val twos : int stream = <fun>
# head twos ;;
- : int = 2
# head (tail twos) ;;
- : int = 2
# head (tail (tail twos)) ;;
- : int = 2
```

We can convert a stream – or at least the first  $n$  of its infinity of elements – into a corresponding list,

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

allowing us to examine the first few elements of the streams we have constructed:

```
# first 10 ones ;;
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1]
```

```
# first 10 twos ;;
- : int list = [2; 2; 2; 2; 2; 2; 2; 2; 2; 2]
```

So far, we've constructed a few infinite streams, but none of much interest. But the tools are in hand to do much more. Think of the natural numbers: 0, 1, 2, 3, 4, 5, .... What is this sequence? We can think of it as the sequence formed by taking the natural numbers, incrementing them all to form the sequence 1, 2, 3, 4, 5, 6, ..., and then prepending a zero to the front, as depicted in Figure 17.1.

We'll define a stream called `nats` in just this way.

```
# let rec nats =
#   fun () -> Cons (0, smap succ nats) ;;
val nats : int stream = <fun>
# first 10 nats ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Let's just pause for a moment to let that sink in.

A function to map over two streams simultaneously, like the `List.map2` function, allows even more powerful ways of building streams.

```
# let rec smap2 f s1 s2 =
#   fun () -> Cons (f (head s1) (head s2),
#                     smap2 f (tail s1) (tail s2)) ;;
val smap2 : ('a -> 'b -> 'c) -> 'a stream -> 'b stream -> 'c stream
= <fun>
```

We can, for instance, generate the Fibonacci sequence (see Exercise 33) in this way. Figure 17.2 gives the recipe.

```
# let rec fibs =
#   fun () -> Cons (0,
#     fun () -> Cons (1,
#       (smap2 (+) fibs (tail fibs)))) ;;
val fibs : int stream = <fun>
```

Here, we've timed generating the first 10 elements of the sequence. It's slow, but it works.

```
# Absbook.call_reporting_time (first 10) fibs ;;
time (msecs): 1.309872
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34]
```

### 17.3 Lazy recomputation and thunks

Recall the definition of streams:

```
type 'a stream_internal = Cons of 'a * 'a stream
and 'a stream = unit -> 'a stream_internal ;;
```

*Start with the natural numbers*  
 $0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ \dots$   
*Increment them*  
 $1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ \dots$   
*Prepend a zero*  
 $0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ \dots$

Figure 17.1: Creating an infinite stream of natural numbers by taking the natural numbers, incrementing them, and prepending a zero.

*Start with the Fibonacci sequence*  
 $0 \ 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ \dots$   
*Take its tail*  
 $1 \ 1 \ 2 \ 3 \ 5 \ 8 \ 13 \ \dots$   
*Sum them*  
 $1 \ 2 \ 3 \ 5 \ 8 \ 13 \ 21 \ \dots$   
*Prepend a zero and one*  
 $0 \ 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ 13 \ 21 \ \dots$

Figure 17.2: Creating an infinite stream of the Fibonacci numbers.

Every time we want access to the head or tail of the stream, we need to rerun the function. In a computation like the Fibonacci definition above, that means that every time we ask for the  $n$ -th Fibonacci number, we recalculate all the previous ones – more than once. But if the value being forced is pure, without side effects, there's no reason to recompute it. We should be able to avoid the recomputation by *remembering* its value the first time it's computed, and using the remembered value from then on. The term of art for this technique is MEMOIZATION.<sup>2</sup>

We'll encapsulate this idea in a new abstraction called a THUNK, essentially a delayed computation that stores its value upon being forced. We implement a thunk as a mutable value (a reference) that can be in one of two states: not yet evaluated or previously evaluated.

The type definition is thus structured with two alternatives.

```
# type 'a thunk = 'a thunk_internal ref
# and 'a thunk_internal =
#   | Unevaluated of (unit -> 'a)
#   | Evaluated of 'a ;;
type 'a thunk = 'a thunk_internal ref
and 'a thunk_internal = Unevaluated of (unit -> 'a) | Evaluated of
  'a
```

Notice that in the unevaluated state, the thunk stores a delayed value of type ' $a$ '. Once evaluated, it stores an immediate value of type ' $a$ '.

When we need to access the actual value encapsulated in a thunk, we'll use the force function. If the thunk has been forced before and thus evaluated, we simply retrieve the value. Otherwise, we compute the value, remember it by changing the state of the thunk to be evaluated, and return the value.

```
# let rec force (t : 'a thunk) : 'a =
#   match !t with
#   | Evaluated v -> v
#   | Unevaluated f ->
#     t := Evaluated (f ());
#     force t ;;
val force : 'a thunk -> 'a = <fun>
```

Here's a thunk for a computation of, say, factorial of 15. To make the timing clearer, we'll give it a side effect of printing a short message.

```
# let fact15 =
#   ref (Unevaluated (fun () ->
#                     print_endline "evaluating 15!";
#                     fact 15)) ;;
val fact15 : int thunk_internal ref = {contents = Unevaluated
  <fun>}
```

which can be forced to carry out the calculation:

<sup>2</sup> Not “memorization”. For unknown reasons, computer scientists have settled on this bastardized form of the word.

```
# Absbook.call_reporting_time force fact15 ;;
evaluating 15!
time (msecs): 0.013828
- : int = 1307674368000
```

Now that the value has been forced, it is remembered in the thunk and can be returned without recomputation. You can tell that no recomputation occurs because the printing side effect doesn't happen, and the computation takes orders of magnitude less time.

```
# fact15 ;;
- : int thunk_internal ref = {contents = Evaluated 1307674368000}
# Absbook.call_reporting_time force fact15 ;;
time (msecs): 0.001192
- : int = 1307674368000
```

### 17.3.1 The Lazy Module

Thunks give us the ability to delay computation, force a delayed computation, and memoize the result. But the notation is horribly cumbersome. Fortunately, OCaml provides a module and some appropriate syntactic sugar for working with lazy computation implemented through thunks – the `Lazy` module.

In the built-in `Lazy` module, the type of a delayed computation of an '`a` value is given not by '`a thunk` but by '`a Lazy.t`. A delayed computation is specified not by wrapping the expression in `ref (Unevaluated (fun () -> ...))` but by preceding it with the new keyword `lazy`. Finally, forcing a delayed value uses the function `Lazy.force`.

Availing ourselves of the `Lazy` module, we can perform the same experiment more simply:

```
# let fact15 =
#   lazy (print_endline "evaluating 15!";
#         fact 15) ;;
val fact15 : int lazy_t = <lazy>
# Absbook.call_reporting_time Lazy.force fact15 ;;
evaluating 15!
time (msecs): 0.010967
- : int = 1307674368000
# Absbook.call_reporting_time Lazy.force fact15 ;;
time (msecs): 0.000954
- : int = 1307674368000
```

Now we can reconstruct infinite streams using the `Lazy` module. First, the stream type:

```
# type 'a stream_internal = Cons of 'a * 'a stream
# and 'a stream = 'a stream_internal Lazy.t ;;
type 'a stream_internal = Cons of 'a * 'a stream
and 'a stream = 'a stream_internal Lazy.t
```

Functions on streams will need to force the stream values. Here, for instance, is the head function:

```
let head (s : 'a stream) : 'a =
  match Lazy.force s with
  | Cons (hd, _tl) -> hd ;;
```

#### Exercise 17.1

Rewrite tail, smap, smap2, and first to use the Lazy module.

The Fibonacci sequence can now be reconstructed. It runs hundreds of times faster than the non-memoized version in Section 17.2.1:

```
# let rec fibs =
#   lazy (Cons (0,
#     lazy (Cons (1,
#       smap2 (+) fibs (tail fibs)))))) ;;
val fibs : int stream = <lazy>
# Absbook.call_reporting_time (first 10) fibs ;;
time (msecs): 0.005960
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34]
```

### 17.4 Application: Approximating $\pi$

A nice application of infinite streams is in the numerical approximation of the value of  $\pi$ . In 1715, the English mathematician Brook Taylor showed how to approximate functions as an infinite sum of terms, a technique we now call TAYLOR SERIES. For instance, the trigonometric arctangent function can be approximated by the following infinite sum:

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots$$

As a special case, the arctangent of 1 is  $\frac{\pi}{4}$  (Figure 17.4). So

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

and

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \dots$$

We can thus approximate  $\pi$  by adding up the terms in this infinite stream of numbers.

We start with a function to convert a stream of integers to a stream of floats.

```
# let to_float = smap float_of_int ;;
val to_float : int stream -> float stream = <fun>
```

Next, we build a stream of odd integers to serve as the denominators in all the terms in the Taylor series:



Figure 17.3: English mathematician Brook Taylor (1685–1731), inventor of the Taylor series approximation of functions.

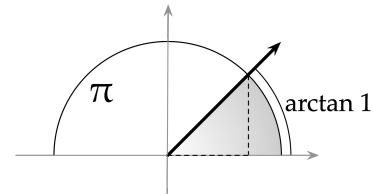


Figure 17.4: The arctangent of 1, that is, the angle whose ratio of opposite to adjacent side is 1, is a 45 degree angle, or  $\frac{\pi}{4}$  in radians.

```
# let odds = smap (fun x -> x * 2 + 1) nats;;
val odds : int stream = <lazy>
```

and a stream of alternating positive and negative ones to represent the alternate adding and subtracting:

```
# let alt_signs =
#   smap (fun x -> if x mod 2 = 0 then 1 else -1) nats;;
val alt_signs : int stream = <lazy>
```

Finally, the stream of terms in the  $\pi$  sequence is

```
# let pi_stream = smap2 ( /.
#   (to_float (smap (( * ) 4) alt_signs))
#   (to_float odds));
val pi_stream : float stream = <lazy>
```

A check of the first few elements in these streams verifies them:

```
# first 5 odds ;;
- : int list = [1; 3; 5; 7; 9]
# first 5 alt_signs ;;
- : int list = [1; -1; 1; -1; 1]
# first 5 pi_stream ;;
- : float list =
[4.; -1.333333333333326; 0.8; -0.571428571428571397;
 0.444444444444442]
```

Now that we have an infinite stream of terms, we can approximate  $\pi$  by taking the sum of the first few elements of the stream, a PARTIAL SUM. The function `pi_approx` extracts the first  $n$  elements of the stream and sums them up using a fold.

```
# let pi_approx n =
#   List.fold_left ( +. ) 0.0 (first n pi_stream) ;;
val pi_approx : int -> float = <fun>
# pi_approx 10 ;;
- : float = 3.04183961892940324
# pi_approx 100 ;;
- : float = 3.13159290355855369
# pi_approx 1000 ;;
- : float = 3.14059265383979413
# pi_approx 10000 ;;
- : float = 3.14149265359003449
# pi_approx 100000 ;;
- : float = 3.14158265358971978
```

After 100,000 terms, we have a pretty good approximation of  $\pi$ , good to about four decimal places.

Of course, this technique of partial sums isn't in the spirit of infinite streams. Better would be to generate an infinite stream of all of the partial sums. Figure 17.5 gives a recipe for generating a stream of partial sums from a given stream. Starting with the stream, we take its

The given sequence  
 1 2 3 4 5 6 7  
 ...  
 ... and its partial sums  
 1 3 6 10 15 21 28  
 ...  
 Prepend a zero to the partial sums  
 0 1 3 6 10 15 21 28  
 ...  
 ... plus the original sequence  
 1 2 3 4 5 6 7 8  
 ...  
 ... yields the partial sums  
 1 3 6 10 15 21 28 36  
 ...

Figure 17.5: Creating an infinite stream of partial sums of a given stream, in this case, the stream of positive integers. We prepend a zero to the sequence's partial sums and add in the original sequence to generate the sequence of partial sums. Only by virtue of lazy computation can this possibly work.

partial sums (!) and prepend a zero. Adding the original stream and the prepended partial sums stream yields...the partial sums stream. This technique, implemented as a function over streams, is:

```
# let rec sums s =
#   smap2 ( +. ) s (lazy (Cons (0.0, sums s))) ;;
val sums : float stream -> float stream = <fun>
```

Now the first few approximations of  $\pi$  are easily accessed:

```
# let pi_approximations = sums pi_stream ;;
val pi_approximations : float stream = <lazy>
# first 5 pi_approximations ;;
- : float list =
[4.; 2.666666666666666696; 3.46666666666666679; 2.89523809523809561;
 3.33968253968254025]
```

If we want to find an approximation within a certain tolerance, say  $\epsilon$ , we can search for two terms in the stream of approximations whose difference is less than  $\epsilon$ .

```
# let rec within epsilon s =
#   let hd, tl = head s, tail s in
#   if abs_float (hd -. (head tl)) < epsilon then hd
#   else within epsilon tl ;;
val within : float -> float stream -> float = <fun>
```

We can now search for a value accurate to within any number of digits we desire:

```
# within 0.01 pi_approximations ;;
- : float = 3.13659268483881615
# within 0.001 pi_approximations ;;
- : float = 3.14109265362104129
```

Continuing on in this vein, we might explore methods for SERIES ACCELERATION – techniques to cause series to converge more quickly – or apply infinite streams to other applications. But for now, this should be sufficient to give a sense of the power of computing with infinite streams.

#### **Exercise 172**

As mentioned in Exercise 33, the ratios of successive numbers in the Fibonacci sequence approach the golden ratio (1.61803...). Show this by generating a stream of ratios of successive Fibonacci numbers and use it to calculate the golden ratio within 0.000001.

### 17.5 Problem section: Circuits and boolean streams

A boolean circuit is a device with one or more inputs and a single output that receives over time a sequence of boolean values on its inputs and converts them to a corresponding sequence of boolean values on its output. The building blocks of circuits are called *gates*.

For instance, the *and* gate is a boolean device with two inputs; its output is `true` when its two inputs are both `true`, and `false` if either output is `false`. The *not* gate is a boolean device with a single input; its output is `true` when its input is `false` and vice versa.

In this problem, you'll generate code for modeling boolean circuits. The inputs and outputs will be modeled as *lazy boolean streams*.

Let's start with an infinite stream of `false` values.

#### Exercise 173

Define a value `falses` to be an infinite stream of the boolean value `false`.

#### Exercise 174

What is the type of `falses`?

#### Exercise 175

A useful function is the `trueat` function. The expression `trueat n` generates a stream of values that are all `false` except for a single `true` at index `n`:

```
# first 5 (trueat 1) ;;
- : bool list = [false; true; false; false; false]
```

Define the function `trueat`.

#### Exercise 176

Define a function `circnot` : `bool stream -> bool stream` to represent the *not* gate.

It should have the following behavior:

```
# first 5 (circnot (trueat 1)) ;;
- : bool list = [true; false; true; true; true]
```

#### Exercise 177

Define a function `circand` to represent the *and* gate. It should have the following behavior:

```
# first 5 (circand (circnot (trueat 1)) (circnot (trueat 3))) ;;
- : bool list = [true; false; true; false; true]
```

A *nand* gate is a gate that computes the negation of an *and* gate. That is, it negates the *and* of its two inputs, so that its output is `false` only if *both* of its inputs are `true`.

#### Exercise 178

Succinctly define a function `circnand` using the functions above to represent the *nand* gate. It should have the following behavior:

```
# first 5 (circnand falses (trueat 3)) ;;
- : bool list = [true; true; true; true; true]
# first 5 (circnand (trueat 3) (trueat 3)) ;;
- : bool list = [true; true; true; false; true]
```

## 17.6 A unit testing framework

With the additional tools of algebraic data types and lazy evaluation, we can put together a more elegant framework for unit testing. Lazy evaluation in particular is useful here, since a unit test is nothing other than an expression to be evaluated for its truth at some later time when the tests are run. Algebraic data types are useful in a couple of

ways, first to package together the components of a test and second to express the alternative ways that a test can come out.

Of course, tests can pass or fail, which we represent by an expression that returns either `true` or `false` respectively. But tests can have other outcomes as well; there are other forms of failing than returning `false`. In particular, a test might raise an exception, or it might not terminate at all. In order to deal with tests that might not terminate, we'll need a way of safely running these tests in a context in which we cut off computation after a specified amount of time. The computation will be said to have **TIMED OUT**. To record the outcome of a test, we'll define a variant type:

```
# type status =
# | Passed
# | Failed
# | Raised_exn of string (* string describing exn *)
# | Timed_out of int      (* timeout in seconds *);;
type status = Passed | Failed | Raised_exn of string | Timed_out of
int
```

A unit test type will package together a mnemonic label for the test, the test condition itself (as a delayed expression), and a timeout period in seconds.

```
# type test =
# { label : string;
#   condition : bool Lazy.t;
#   time : int };;
type test = { label : string; condition : bool Lazy.t; time : int;
}
```

Notice that the condition of the test is a lazy boolean, so that the condition will not be evaluated until the test is run.

To construct a test, we provide a function that packages together the components.<sup>3</sup>

```
# (* test ?time label condition -- Returns a test with the
#   given label and condition, with optional timeout time
#   in seconds (defaulting to 5 seconds). *)
# let test ?(time=5) label condition =
#   {label; condition; time};;
val test : ?time:int -> string -> bool Lazy.t -> test = <fun>
```

The crux of the matter is the running of a test. Doing so generates a value of type `status`. The `run_test` function will be provided a function `continue` to be applied to the label of the test and its status. For instance, an appropriate such function might print out a line in a report describing the outcome, like this:

```
# (* present_labels status -- Prints a line describing the
#   outcome of a test. Appropriate for use as the continue
```

<sup>3</sup> We make use of an optional argument for the timeout time, which defaults to five seconds if not provided. For the interested, details of optional arguments are discussed [here](#).

```

#     function in run_test. *)
# let present (label : string) (status : status) : unit =
#   let open Printf in
#   match status with
#   | Passed ->
#     printf "%s: passed\n" label
#   | Failed ->
#     printf "%s: failed\n" label
#   | Timed_out secs ->
#     printf "%s: timed out after %d seconds\n" label secs
#   | Raised_exn msg ->
#     printf "%s: raised %s\n" label msg ;;
val present : string -> status -> unit = <fun>
```

The `run_test` function needs to evaluate the test by forcing evaluation of the delayed condition. As a first cut, we'll look only to the normal case, where a test returns `true` or `false`.

```

# (* run_test test continue -- Runs the test, applying the
#    continue function to the test label and status. *)
# let run_test ({label; condition; _} : test)
#   (continue : string -> status -> unit)
#   : unit =
#   let result = Lazy.force condition in
#   if result then continue label Passed
#   else continue label Failed ;;
val run_test : test -> (string -> status -> unit) -> unit = <fun>
```

But what if the test raises an exception? We'll evaluate the test condition in a `try`  $\langle \rangle$  `with`  $\langle \rangle$  to deal with this case.

```

# (* run_test test continue -- Runs the test, applying the
#    continue function to the test label and status. *)
# let run_test ({label; condition; _} : test)
#   (continue : string -> status -> unit)
#   : unit =
#   try
#     let result = Lazy.force condition in
#     if result then continue label Passed
#     else continue label Failed
#   with
#   | exn -> continue label
#     (Raised_exn (Printexc.to_string exn)) ;;
val run_test : test -> (string -> status -> unit) -> unit = <fun>
```

Finally, we need to deal with timeouts. We appeal to a function `timeout` that forces a lazy computation, but raises a special `Timeout` exception if the computation goes on too long. The workings of this function are well beyond the scope of this text, but we provide the code in Figure 17.6.

Using the `timeout` function to force the condition and checking for the `Timeout` exception handles the final possible status of a unit test.

---

```

# (* timeout time f -- Forces delayed computation f, returning
#   what f returns, except that after time seconds it raises
#   a Timeout exception. *)

#
# exception Timeout;;
exception Timeout

# let sigalrm_handler =
#   Sys.Signal_handle (fun _ -> raise Timeout) ;;
val sigalrm_handler : Sys.signal_behavior = Sys.Signal_handle <fun>

# let timeout (time : int) (f : 'a Lazy.t) : 'a =
#   let old_behavior =
#     Sys.signal Sys.sigalrm sigalrm_handler in
#   let reset_sigalrm () =
#     ignore (Unix.alarm 0);
#   Sys.set_signal Sys.sigalrm old_behavior in
#   ignore (Unix.alarm time) ;
#   let res = Lazy.force f in
#   reset_sigalrm () ; res ;;
val timeout : int -> 'a Lazy.t -> 'a = <fun>

```

---

Figure 17.6: The function `timeout` used in the evaluation of unit tests, based on the `timeout` function of Chailloux et al. (2000)

```

# (* run_test test continue -- Runs the test, applying the
#   continue function to the test label and status. *)
# let run_test ({label; time; condition} : test)
#   (continue : string -> status -> unit)
#   : unit =
#   try
#     if timeout time condition
#     then continue label Passed
#     else continue label Failed
#   with
#   | Timeout -> continue label (Timed_out time)
#   | exn      -> continue label
#           (Raised_exn (Printexc.to_string exn)) ;;
val run_test : test -> (string -> status -> unit) -> unit = <fun>

```

By iterating over a list of unit tests, we can generate a nice report of all the tests.

```

# (* report tests -- Generates a report based on the
#   provided tests. *)
# let report (tests : test list) : unit =
#   List.iter (fun test -> run_test test present) tests ;;
val report : test list -> unit = <fun>

```

With this infrastructure in place, we can define a test suite that demonstrates all of the functionality of the unit testing framework.

```

# let tests =
#   [ test "should fail" (lazy (3 > 4));
#   test "should pass" (lazy (4 > 3));
#   test "should time out" (lazy (let rec f x = f x in f 1));
#   test "should raise exception" (lazy ((List.nth [0; 1] 3) = 3))
#   ] ;;
val tests : test list =
[{label = "should fail"; condition = <lazy>; time = 5};
 {label = "should pass"; condition = <lazy>; time = 5};
 {label = "should time out"; condition = <lazy>; time = 5};
 {label = "should raise exception"; condition = <lazy>; time =
5}]

# report tests ;;
should fail: failed
should pass: passed
should time out: timed out after 5 seconds
should raise exception: raised Failure("nth")
- : unit = ()

```

## 17.7 A brief history of laziness

The idea of lazy computation probably starts with Peter Landin (Figure 17.7). He observed “a relationship between lists and functions”:

In this relationship a nonnull list  $L$  is mirrored by a none-adic function  $S$  that produces a 2-list consisting of (1) the head of  $L$ , and (2) the function

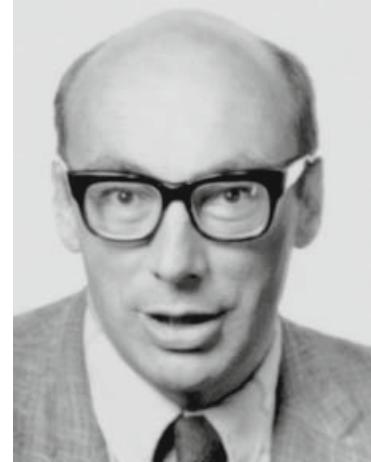


Figure 17.7: Peter Landin (1930–2009), developer of many innovative ideas in programming languages, including the roots of lazy programming. His influence transcended his role as a computer scientist, especially in his active support of gay rights.

mirroring the tail of  $L$ . . . This correspondence serves two related purposes. It enables us to perform operations on lists (such as generating them, mapping them, concatenating them) without using an “extensive,” item-by-item representation of the intermediately resulting lists; and it enables us to postpone the evaluation of the expressions specifying the items of a list until they are actually needed. The second of these is what interests us here. (Landin, 1965)

The idea of a “function mirroring the tail of” a list is exactly the delaying of the tail computation that we’ve seen in the `stream` data type.

Landin is notable for many other ideas of great currency. For instance, he invented the term “syntactic sugar” for the addition of extra concrete syntax to abbreviate some useful but otherwise complicated abstract syntax. His 1966 paper “The next 700 programming languages” (Landin, 1966) introduced several innovative ideas including the “offside rule” of concrete syntax, allowing the indentation pattern of a program to indicate its structure. Python is typically noted for making use of this Landin innovation. Indeed, the ISWIM language that Landin described in this paper is arguably the most influential programming language that no one ever programmed in.

Following Landin’s observation, Wadsworth proposed the lazy lambda calculus in 1971, and Friedman and Wise published an article proposing that “Cons should not evaluate its arguments” in 1976. The first programming language to specify lazy evaluation as the evaluation regime was Burstall’s Hope language (which also introduced the idea, found in nascent form in ISWIM, of algebraic data types). A series of lazy languages followed, most notably Miranda, but the lazy programming community came together to converge on the now canonical lazy language Haskell, named after Haskell Curry.

### 17.8 Supplementary material

- Lab 14: Lazy programming and infinite data structures: Implementing laziness as user code
- Lab 15: Lazy programming and infinite data structures: Using OCaml’s native lazy module
- Problem set A.7: Refs, streams, and music

# 18

## *Extension and object-oriented programming*

Think of your favorite graphical user interface (GUI). It probably has various WIDGETS – buttons, checkboxes, textboxes, radio buttons, menus, icons, and so forth. These widgets might undergo various operations – we might want to draw them in a window, click on them, change their location, remove them, highlight them, select from them. Each of these operations seems like a function. We'd organize functions like this:

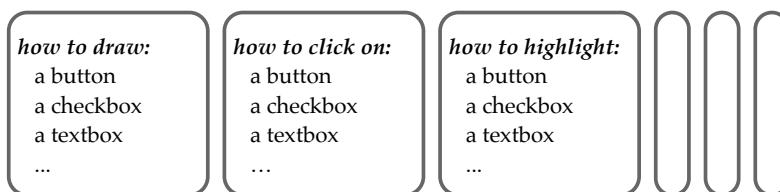


Figure 18.1: Function-oriented organization of widget software

But new widgets are being invented all the time. Every time a new widget type is added, we'd have to change every one of these functions. Instead, we might want to organize the code a different way:

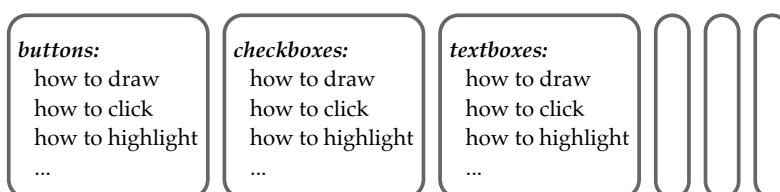


Figure 18.2: Object-oriented organization of widget software

This way, adding a new widget doesn't affect any of the existing ones. The changes are localized, and therefore likely to be much more reliably added. We are carving the software at its joints, following the edict of decomposition.

This latter approach to code organization, organizing by “object”

rather than by function, is referred to as OBJECT-ORIENTED. It's probably no surprise that the rise in popularity of object-oriented programming tracks the development of graphical user interfaces; as seen above, it's a natural fit. In particular, the idea of object-oriented programming was popularized by the Smalltalk programming language and system, which pioneered many of the fundamental ideas of graphical user interfaces that we are now accustomed to – windows, icons, menus, buttons. Smalltalk with its graphical user interface was developed in the early 1970's at Xerox PARC by Alan Kay, Adele Goldberg, Dan Ingalls, and others (Figure 18.3). Steve Jobs, seeing the Smalltalk environment in a 1979 visit to Xerox PARC, immediately imported the ideas into Apple's Lisa and Macintosh computers, thereby disseminating and indeed universalizing the ideas.

In this chapter, we introduce object-oriented programming, a programming paradigm based on organizing functionalities (in the form of methods) together with the data that they operate on, as opposed to the functional paradigm, which organizes functionalities (in the form of functions) separate from the corresponding data.

### 18.1 Drawing graphical elements

To motivate such a reorganization, consider a program to draw graphical elements on a window. We'll start by organizing the code in a function-oriented, not object-oriented, style.

Positions in the window can be captured with a point data type:

```
# type point = {x : int; y : int} ;;
type point = { x : int; y : int; }
```

We might want data types for the individual kinds of graphical elements – rectangles, circles, squares – each with its own parameters specifying pertinent positions, sizes, and the like:

```
# type rect = {rect_pos : point;
#               rect_width : int;
#               rect_height : int} ;;
type rect = { rect_pos : point; rect_width : int; rect_height :
int; }

# type circle = {circle_pos : point; circle_radius : int} ;;
type circle = { circle_pos : point; circle_radius : int; }

# type square = {square_pos : point; square_width : int} ;;
type square = { square_pos : point; square_width : int; }
```

We can think of a *scene* as being composed of a set of these *display elements*:



Figure 18.3: Alan Kay, Adele Goldberg, and Dan Ingalls, developers of the influential Smalltalk language, a pioneering object-oriented language, with an innovative user interface based on graphical widgets and direct manipulation.

```

# type display_elt =
#   | Rect of rect
#   | Circle of circle
#   | Square of square ;;
type display_elt = Rect of rect | Circle of circle | Square of
square

# type scene = display_elt list ;;
type scene = display_elt list

```

In order to make use of these elements to actually draw on a screen, we'll make use of [the OCaml Graphics module](#), which you may want to familiarize yourself with before proceeding. (We rename the module G for brevity.)

```

# module G = Graphics ;;
module G = Graphics

```

We can write a function to draw a display element of whatever variety by dispatching (matching) based on the variant of the `display_elt` type:<sup>1</sup>

```

# let draw (d : display_elt) : unit =
#   match d with
#   | Rect r ->
#     G.set_color G.black;
#     G.fill_rect (r.rect_pos.x - r.rect_width / 2)
#                   (r.rect_pos.y - r.rect_height / 2)
#                   r.rect_width r.rect_height
#   | Circle c ->
#     G.set_color G.black;
#     G.fill_circle c.circle_pos.x c.circle_pos.y
#                   c.circle_radius
#   | Square s ->
#     G.set_color G.black;
#     G.fill_rect (s.square_pos.x - s.square_width / 2)
#                   (s.square_pos.y - s.square_width / 2)
#                   s.square_width s.square_width ;;
val draw : display_elt -> unit = <fun>

```

and use it to draw an entire scene on a fresh canvas:

```

# let draw_scene (s : scene) : unit =
#   try
#     G.open_graph "";          (* open the canvas *)
#     G.resize_window 200 300; (* erase and resize *)
#     List.iter draw s;        (* draw the elements *)
#     ignore (G.read_key ())   (* wait for a keystroke *)
#   with
#     exn -> (G.close_graph () ; raise exn) ;;
val draw_scene : scene -> unit = <fun>

```

<sup>1</sup> All of the subtractions of half the widths and heights is because the Graphics module often draws graphics based on the lower left hand corner position, instead of the center of the graphic that we're using.

Let's test it on a simple scene of a few rectangles and circles:

```

# let test_scene =
#   [ Rect {rect_pos = {x = 0; y = 20};
#           rect_width = 15; rect_height = 80};
#     Circle {circle_pos = {x = 40; y = 100};
#              circle_radius = 40};
#     Circle {circle_pos = {x = 40; y = 140};
#              circle_radius = 20};
#     Square {square_pos = {x = 65; y = 160};
#             square_width = 50} ] ;;
val test_scene : display_elt list =
  [Rect {rect_pos = {x = 0; y = 20}; rect_width = 15; rect_height = 80};
  Circle {circle_pos = {x = 40; y = 100}; circle_radius = 40};
  Circle {circle_pos = {x = 40; y = 140}; circle_radius = 20};
  Square {square_pos = {x = 65; y = 160}; square_width = 50}]

# draw_scene test_scene ;;
- : unit = ()

```

A window pops up with the scene (Figure 18.4(a)).

Sadly, the scene is not centered very well in the canvas. Fortunately, it's easy to add functionality in the functional programming paradigm: just add functions. We can easily add functions to translate a display element or a scene by a given amount in the *x* and *y* directions.

```

# let translate (p : point) (d : display_elt) : display_elt =
#   let vec_sum {x = x1; y = y1} {x = x2; y = y2} =
#     {x = x1 + x2; y = y1 + y2} in
#   match d with
#   | Rect r ->
#     Rect {r with rect_pos = vec_sum p r.rect_pos}
#   | Circle c ->
#     Circle {c with circle_pos = vec_sum p c.circle_pos}
#   | Square s ->
#     Square {s with square_pos = vec_sum p s.square_pos} ;;
val translate : point -> display_elt -> display_elt = <fun>

# let translate_scene (p : point) : scene -> scene =
#   List.map (translate p) ;;
val translate_scene : point -> scene -> scene = <fun>

```

Using these, we can translate the scene to center it before drawing:

```

# draw_scene (translate_scene {x = 42; y = 50} test_scene) ;;
- : unit = ()

```

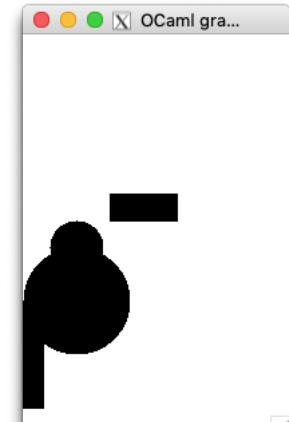
to get the depiction in Figure 18.4(b).

So adding functionality is easy. What about adding new types of data, new display elements? Suppose we want to add a textual display element to place some text in the scene.

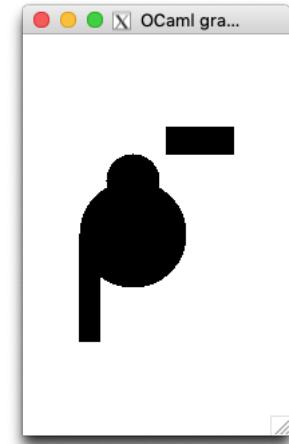
```

# type text = {text_pos : point;
#               text_title : string} ;;
type text = { text_pos : point; text_title : string; }

```



(a)



(b)

Figure 18.4: (a) A test scene. (b) The test scene translated.

We'll have to modify the `display_elt` data type to incorporate text elements:

```
# type display_elt =
#   | Rect of rect
#   | Circle of circle
#   | Square of square
#   | Text of text;;
type display_elt =
  Rect of rect
| Circle of circle
| Square of square
| Text of text
```

Now the `draw` function complains (unsurprisingly) of an inexhaustive match:

```
# let draw (d : display_elt) : unit =
#   match d with
#   | Rect r ->
#     G.set_color G.black;
#     G.fill_rect (r.rect_pos.x - r.rect_width / 2)
#                 (r.rect_pos.y - r.rect_height / 2)
#                 r.rect_width r.rect_height
#   | Circle c ->
#     G.set_color G.black;
#     G.fill_circle c.circle_pos.x c.circle_pos.y
#                   c.circle_radius
#   | Square s ->
#     G.set_color G.black;
#     G.fill_rect (s.square_pos.x - s.square_width / 2)
#                 (s.square_pos.y - s.square_width / 2)
#                 s.square_width s.square_width;;
Lines 2-16, characters 0-29:
2 | match d with
3 | | | Rect r ->
4 | | G.set_color G.black;
5 | | G.fill_rect (r.rect_pos.x - r.rect_width / 2)
6 | | (r.rect_pos.y - r.rect_height / 2)
...
13 | G.set_color G.black;
14 | G.fill_rect (s.square_pos.x - s.square_width / 2)
15 | (s.square_pos.y - s.square_width / 2)
16 | s.square_width s.square_width...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Text _
```

`val draw : display_elt -> unit = <fun>`

We'll have to augment it to handle drawing text. Ditto for the `translate` function. In fact, every function that manipulates display elements will have to be changed. If we're going to be adding new types of elements to display, translate, and the like, this will get unwieldy quickly. But there's a better way – objects.

## 18.2 Objects introduced

What do we care about about display elements? That they can be drawn. That's it. We want to abstract away from all else.

We'll define a data type, an abstraction, `display_elt`, that is a record with a single field called `draw` that stores a drawing function.

```
# type display_elt = {draw : unit -> unit} ;;
type display_elt = { draw : unit -> unit; }
```

Then rectangles, circles, squares, and texts are just ways of building display elements with that drawing functionality.

Take rectangles for example. A rectangle is a `display_elt` whose `draw` function displays a rectangle. We can establish a `rect` function that builds such a display element given its initial parameters – position, width, and height:

```
# let rect (p : point) (w : int) (h : int) : display_elt =
#   { draw = fun () ->
#     G.set_color G.black ;
#     G.fill_rect (p.x - w/2) (p.y - h/2) w h } ;;
val rect : point -> int -> int -> display_elt = <fun>
```

Similarly with circles and squares:

```
# let circle (p : point) (r : int) : display_elt =
#   { draw = fun () ->
#     G.set_color G.black;
#     G.fill_circle p.x p.y r } ;;
val circle : point -> int -> display_elt = <fun>

# let square (p : point) (w : int) : display_elt =
#   { draw = fun () ->
#     G.set_color G.black ;
#     G.fill_rect (p.x - w/2) (p.y - w/2) w w } ;;
val square : point -> int -> display_elt = <fun>
```

Now to draw a display element, we just extract the `draw` function and call it. The display element data object knows how to draw itself.

```
# let draw (d : display_elt) = d.draw () ;;
val draw : display_elt -> unit = <fun>
```

If we want to add a new display element, a text, say, we just have to provide a way to draw such a thing. No other code (`draw`, `draw_scene`) needs to change.

```
# let text (p : point) (s : string) : display_elt =
#   { draw = (fun () ->
#             let (w, h) = G.text_size s in
#             G.set_color G.black;
#             G.moveto (p.x - w/2) (p.y - h/2);
#             G.draw_string s) } ;;
val text : point -> string -> display_elt = <fun>
```

Of course, we'd probably want display elements to have more functionality than just drawing themselves – for instance, moving them to a new position, querying and changing their color, and much more. Let's start with these.

```
# type display_elt =
#   { draw : unit -> unit;
#     set_pos : point -> unit;
#     get_pos : unit -> point;
#     set_color : G.color -> unit;
#     get_color : unit -> G.color };;
type display_elt = {
  draw : unit -> unit;
  set_pos : point -> unit;
  get_pos : unit -> point;
  set_color : G.color -> unit;
  get_color : unit -> G.color;
}
```

Notice that display elements now (apparently) must have mutable state. Their position and color can be modified over time. We'll implement this state by creating appropriate references, called `pos` and `color`, respectively, that are generated upon creation of an object and are specific to it. Here, for instance, is the `circle` function to create a circular display element object:

```
# let circle (p : point) (r : int) : display_elt =
#   let pos = ref p in
#   let color = ref G.black in
#   { draw = (fun () -> G.set_color (!color);
#             G.fill_circle (!pos).x (!pos).y r);
#     set_pos = (fun p -> pos := p);
#     get_pos = (fun () -> !pos);
#     set_color = (fun c -> color := c);
#     get_color = (fun () -> !color) };;
val circle : point -> int -> display_elt = <fun>
```

The scoping is crucial. The definitions of `pos` and `color` are within the scope of the `circle` function. Thus, new references are generated each time `circle` is invoked and are accessible only to the record structure (the object) created by that invocation.<sup>2</sup> Similarly, we'll want a function to create rectangles and text boxes, each with its own state and functionality as specified by the `display_elt` type.

```
# let rect (p : point) (w : int) (h : int) : display_elt =
#   let pos = ref p in
#   let color = ref G.black in
#   { draw = (fun () ->
#             G.set_color (!color);
#             G.fill_rect ((!pos).x - w/2) ((!pos).y - h/2)
#                         w h);
#     set_pos = (fun p -> pos := p);
```

<sup>2</sup> Recall the similar idea of local, otherwise inaccessible, persistent, mutable state first introduced in the `bump` function from Section 15.3, and reproduced here:

```
# let bump =
#   let ctr = ref 0 in
#   fun () ->
#     ctr := !ctr + 1;
#     !ctr ;;
val bump : unit -> int = <fun>
```

```

#     get_pos = (fun () -> !pos);
#     set_color = (fun c -> color := c);
#     get_color = (fun () -> !color) };;
val rect : point -> int -> int -> display_elt = <fun>

# let text (p : point) (s : string) : display_elt =
#   let pos = ref p in
#   let color = ref G.black in
#   { draw = (fun () ->
#             let (w, h) = G.text_size s in
#             G.set_color (!color);
#             G.moveto ((!pos).x - w/2) ((!pos).y - h/2);
#             G.draw_string s);
#   set_pos = (fun p -> pos := p);
#   get_pos = (fun () -> !pos);
#   set_color = (fun c -> color := c);
#   get_color = (fun () -> !color) } ;;
val text : point -> string -> display_elt = <fun>

```

What we've done is to generate a wholesale reorganization of the display element code, organizing it not by functionality (with a `draw` function, a `set_pos` function, and so forth), but instead by variety of "object" bearing that functionality. We've organized the code in an **OBJECT-ORIENTED** manner.

Think of a table (as in Table 18.1) that describes for each functionality (draw, move, getting and setting color) and each class of object (rectangle, circle, text) the code necessary to carry out that functionality for that class of object. We can organize the code by functionality, packaging the rows into functions; this is the function-oriented paradigm. Alternatively, we can organize the code by class of object, packaging the columns into objects; this is the object-oriented paradigm.

|                  | <i>rectangle</i>                                              | <i>circle</i>                                                | <i>text</i>                                                                |
|------------------|---------------------------------------------------------------|--------------------------------------------------------------|----------------------------------------------------------------------------|
| <i>draw</i>      | G.set_color (!color);<br>G.fill_rect (!pos).x<br>(!pos).y w h | G.set_color (!color)<br>G.fill_circle (!pos).x<br>(!pos).y r | G.set_color (!color);<br>G.moveto (!pos).x<br>(!pos).y;<br>G.draw_string s |
| <i>move</i>      | pos := p                                                      | pos := p                                                     | pos := p                                                                   |
| <i>set color</i> | color := c                                                    | color := c                                                   | color := c                                                                 |
| <i>get color</i> | !color                                                        | !color                                                       | !color                                                                     |

Which is the better approach? The edict of decomposition appeals to cutting up software at its joints. Which of row or column constitutes the natural joints will vary from case to case. It is thus a fundamental

Table 18.1: The matrix of functionality (rows) and object classes (columns) for the display elements example.  
The code can be organized by row – function-oriented – or by column – object-oriented.

design decision as to whether to use a function- or object-oriented structuring of code. If you expect a need to add additional columns with regularity, whereas adding rows will be rare, the object-oriented approach will fare better. Conversely, if new rows, new functionality, will be needed over a relatively static set of classes of data, the function-oriented approach is preferable.

### 18.3 Object-oriented terminology and syntax

The object-oriented programming paradigm that we've reconstructed here comes with its own set of terminology. First, the data structure that encapsulates the various bits of functionality – here implemented as a simple record structure – is an **OBJECT**. The various components providing the functionality are its **METHODS**, and the state variables (like `color` and `pos`) its **INSTANCE VARIABLES**. The specification of what methods are provided by an object (like `display_elt`) is its **CLASS INTERFACE**, and the creation of an object is specified by its **CLASS** (like `circle` or `text`).

We create an object by **INSTANTIATING** the class, in this example, the `circle` class,

```
# let circle1 = circle {x = 100; y = 100} 50;;
val circle1 : display_elt =
{draw = <fun>; set_pos = <fun>; get_pos = <fun>; set_color =
<fun>;
get_color = <fun>}
```

which satisfies the `display_elt` class interface.

When we make use of a method, for instance, the `set_pos` method,

```
# circle1.set_pos {x = 125; y = 125} ;;
- : unit = ()
```

we are said to **INVOKE** the method

It should be clear that the object-oriented programming paradigm can be carried out in any programming language with the abstractions that we've relied on here, basically, first-class functions, lexical scoping, and mutable state. But, as with other programming paradigms we've looked at, providing some syntactic sugar in support of the paradigm can be quite useful. OCaml does just that. Indeed, the “O” in “OCaml” indicates that the language was developed as an extension to the Caml language by adding syntactic support for object-oriented programming.

The object-oriented syntax extensions in OCaml are summarized in Table 18.2.

The `display_elt` example can thus be stated in colloquial OCaml as follows. We start with the `display_elt` class interface:

| Concept                  | Syntax                                                      |
|--------------------------|-------------------------------------------------------------|
| Class interfaces         | <code>class type &lt;interfacename&gt; = ...</code>         |
| Class definition         | <code>class &lt;classname&gt; &lt;args&gt; = ...</code>     |
| Object definition        | <code>object ... end</code>                                 |
| Instance variables       | <code>val (mutable) &lt;varname&gt; = ...</code>            |
| Methods                  | <code>method &lt;methodname&gt; &lt;args&gt; = ...</code>   |
| Instance variable update | <code>... &lt;- ...</code>                                  |
| Instantiating classes    | <code>new &lt;classname&gt; &lt;args&gt;</code>             |
| Invoking methods         | <code>&lt;object&gt;#&lt;methodname&gt; &lt;args&gt;</code> |

Table 18.2: Syntactic extensions in OCaml supporting object-oriented programming.

```
# class type display_elt =
#   object
#     method draw : unit
#     method set_pos : point -> unit
#     method get_pos : point
#     method set_color : G.color -> unit
#     method get_color : G.color
#   end ;;
class type display_elt =
object
  method draw : unit
  method get_color : G.color
  method get_pos : point
  method set_color : G.color -> unit
  method set_pos : point -> unit
end
```

and define some classes that satisfy the interface:

```
# class circle (p : point) (r : int) : display_elt =
#   object
#     val mutable pos = p
#     val mutable color = G.black
#     method draw = G.set_color color;
#       G.fill_circle pos.x pos.y r
#     method set_pos p = pos <- p
#     method get_pos = pos
#     method set_color c = color <- c
#     method get_color = color
#   end ;;
class circle : point -> int -> display_elt

# class rect (p : point) (w : int) (h : int) : display_elt =
#   object
#     val mutable pos = p
#     val mutable color = G.black
#     method draw = G.set_color color;
#       G.fill_rect (pos.x - w/2) (pos.y - h/2)
#       w h
#     method set_pos p = pos <- p
```

```

#     method get_pos = pos
#     method set_color c = color <- c
#     method get_color = color
#   end ;;
class rect : point -> int -> int -> display_elt

```

Now we can use these to create and draw some display elements. We create a new circle,

```

# let _ = G.open_graph "";
#           G.clear_graph ;;
- : unit -> unit = <fun>
# let b = new circle {x = 100; y = 100} 40 ;;
val b : circle = <obj>

```

but nothing appears yet until we draw the element.

```

# let _ = b#draw ;;
- : unit = ()

```

(Notice that invoking the method doesn't require the application to a **unit**. In the object-oriented syntax, method invocation with no arguments can be implicit in this way.) The circle now appears, as in Figure 18.5(a).

We can erase the object by setting its color to white and redrawing it (Figure 18.5(b)).

```

# let _ = b#set_color G.white;
#           b#draw ;;
- : unit = ()

```

We move it to a new position and change its color (Figure 18.5(c)).

```

# let _ = b#set_pos {x = 150; y = 150};
#           b#set_color G.red;
#           b#draw ;;
- : unit = ()

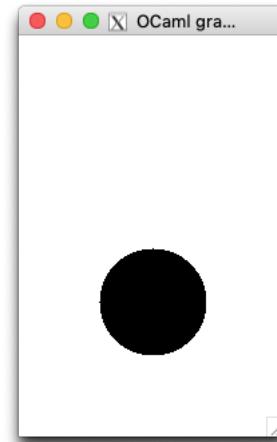
```

#### 18.4 Inheritance

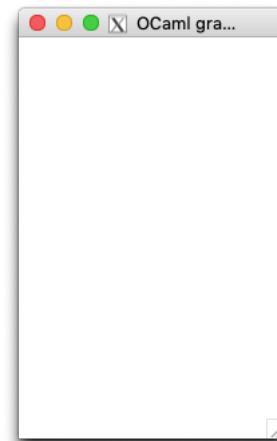
The code we've developed so far violates the edict of irredundancy. The implementations of the `circle` and `rect` classes, for instance, are almost identical, differing only in the arguments of the class and the details of the `draw` method.

To capture the commonality, the object-oriented paradigm allows for definition of a class expressing the common aspects, from which both of the classes can **INHERIT** their behaviors. We refer to the class (or class type) that is being inherited from as the **SUPERCLASS** and the inheriting class as the **SUBCLASS**.

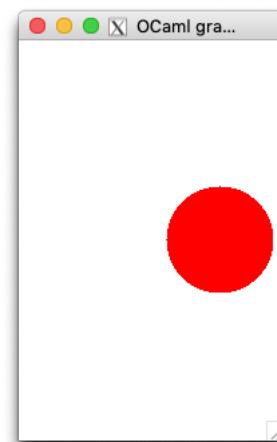
We'll define a shape superclass that can handle the position and color aspects of the more specific classes. Its class type is given by



(a)



(b)



(c)

Figure 18.5: A circle appears (a) and disappears (b). It moves and reappears with a changed color (c).

```

# class type shape_elt =
#   object
#     method set_pos : point -> unit
#     method get_pos : point
#     method set_color : G.color -> unit
#     method get_color : G.color
#   end ;;
class type shape_elt =
object
  method get_color : G.color
  method get_pos : point
  method set_color : G.color -> unit
  method set_pos : point -> unit
end

```

and a simple implementation of the class is

```

# class shape (p : point) : shape_elt =
#   object
#     val mutable pos = p
#     val mutable color = G.black
#     method set_pos p = pos <- p
#     method get_pos = pos
#     method set_color c = color <- c
#     method get_color = color
#   end ;;
class shape : point -> shape_elt

```

Notice that the new `shape_elt` signature provides access to the four methods, but not directly to the instance variables used to implement those methods. The only access to those instance variables will be through the methods, an instance of the edict of compartmentalization that seems appropriate.

The `display_elt` class type can inherit the methods from `shape_elt`, adding just the additional `draw` method.

```

# class type display_elt =
#   object
#     inherit shape_elt
#     method draw : unit
#   end ;;
class type display_elt =
object
  method draw : unit
  method get_color : G.color
  method get_pos : point
  method set_color : G.color -> unit
  method set_pos : point -> unit
end

```

The `inherit` specification works as if the contents of the inherited superclass type were simply copied into the subclass type at that location in the code.

The `rect` and `circle` subclasses can inherit much of their behavior from the `shape` superclass, just adding their own `draw` methods. However, without the ability to refer directly to the instance variables, the `draw` method will need to call its own methods for getting and setting the position and color. We can add a variable to name the object itself, by adding a parenthesized name after the `object` keyword. Although any name can be used, by convention, we use `this` or `self`. We can then invoke the methods from the `shape` superclass with, for instance, `this#get_color`.

```
# class rect (p : point) (w : int) (h : int) : display_elt =
#  object (this)
#    inherit shape p
#    method draw =
#      G.set_color this#get_color;
#      G.fill_rect (this#get_pos.x - w/2)
#                  (this#get_pos.y - h/2)
#                  w h
#    end ;;
class rect : point -> int -> int -> display_elt

# class circle (p : point) (r : int) : display_elt =
#  object (this)
#    inherit shape p
#    method draw =
#      G.set_color this#get_color;
#      G.fill_circle this#get_pos.x this#get_pos.y r
#    end ;;
class circle : point -> int -> display_elt
```

Notice how the inherited `shape` class is provided the position argument `p` so its instance variables and methods can be set up properly.

Using inheritance, a `square` class can be implemented with a single inheritance from the `rect` class, merely by specifying that the width and height of the inherited rectangle are the same:

```
# class square (p : point) (w : int) : display_elt =
#  object
#    inherit rect p w w
#  end ;;
class square : point -> int -> display_elt
```

#### **Exercise 179**

Define a class `text` : `point -> string -> display_elt` for placing a string of text at a given point position on the canvas. (You'll need the `Graphics.draw_string` function for this.)

#### *18.4.1 Overriding*

Inheritance in OCaml allows for subclasses to override the methods in superclasses. For instance, we can implement a class of bordered

rectangles (rather than the filled rectangles of the `rect` class) simply by overriding the `draw` method:

```
# class border_rect (p : point)
#           (w : int) (h : int)
#           : display_elt =
# object (this)
#   inherit rect p w h as super
#   method! draw = G.set_color this#get_color;
#           G.fill_rect (this#get_pos.x - w/2 - 2)
#                         (this#get_pos.y - h/2 - 2)
#                         (w+4) (h+4) ;
#   let c = this#get_color in
#   this#set_color G.white ;
#   super#draw ;
#   this#set_color c
# end ;;
class border_rect : point -> int -> int -> display_elt
```

Here, we've introduced the overriding `draw` method with `method!`, where the exclamation mark diacritic explicitly marks the method as overriding the superclass's `draw` method. Without that, OCaml will provide a helpful warning to the programmer in case the overriding was unintentional.

When a subclass overrides the method of a superclass, the subclass may still want access to the superclass's version of the method. That's the case here, where the subclass's `draw` method needs to call the superclass's. In the presence of overriding, then, it becomes important to have a name for the superclass object so as to be able to call its methods. The inherited superclass can be given a name for this purpose by the `as` construct used above in the `inherit` specification. The variable following the `as` – conventionally `super` though any variable can be used – then names the superclass providing access to its version of any overridden methods.

## 18.5 Subtyping

Back in Section 18.1, we defined a scene as a set of drawable elements, so as to be able to iterate over a scene to draw each element. We can obtain that ability by defining a new function that draws a list of `display_elt` objects:

```
# let draw_list (d : display_elt list) : unit =
#   List.iter (fun x -> x#draw) d ;;
val draw_list : display_elt list -> unit = <fun>
```

We've put together a small scene (Figure 18.6), evocatively called `scene`, to test the process.<sup>3</sup>

<sup>3</sup> The type of the scene is displayed not, as one might expect, as `display_elt` list but as `border_rect` list. OCaml uses class names, not class type names, to serve the purpose of reporting typing information for objects. The elements of `scene` are instances of various classes (all consistent with class type `display_elt`). OCaml selects the first element of the list, which happens to be a `border_rect` instance, to serve as the printable name of the type. This quirk of OCaml reveals that the grafting of the “O” part of the language isn't always seamless.

Figure 18.6: A test scene.

---

```

let scene =
  (* generate some graphical objects *)
  let box = new border_rect {x = 100; y = 110} 180 210 in
  let l1 = new rect {x = 70; y = 60} 20 80 in
  let l2 = new rect {x = 135; y = 100} 20 160 in
  let b = new circle {x = 100; y = 100} 40 in
  let bu = new circle {x = 100; y = 140} 20 in
  let h = new rect {x = 150; y = 170} 50 20 in
  let t = new text {x = 100; y = 200} "The CS51 camel" in
  (* bundle them together *)
  let scene = [box; l1; l2; b; bu; h; t] in
  (* change their color and translate them *)
  List.iter (fun x -> x#set_color 0x994c00) scene;
  List.iter (fun o -> let {x; y} = o#get_pos in
    o#set_pos {x = x + 50; y = y + 40})
  scene;
  (* update the surround color *)
  box#set_color G.blue;
  scene ;;

```

---

```

# scene ;;
- : border_rect list = [; <obj>; <obj>; <obj>; <obj>; <obj>;
<obj>]

```

We can draw this scene in a fresh window using `draw_list`.

```

# let test scene =
#   try
#     G.open_graph "";
#     G.resize_window 300 300;
#     G.clear_graph ();
#     draw_list scene;
#     ignore (G.read_key ())
#   with
#     exn -> (G.close_graph (); raise exn) ;;
val test : display_elt list -> unit = <fun>

# test scene ;;
- : unit = ()

```

We defined `draw_list` to operate on `display_elt` lists. But there's no reason to be so specific. It ought to be the case that any object with a `draw` method should be able to participate in a scene. We can define a new class type of drawable elements

```

# class type drawable =
#   object
#     method draw : unit

```

```
# end ;;
class type drawable = object method draw : unit end
```

and redefine `draw_list` accordingly:

```
# let draw_list (d : drawable list) : unit =
#   List.iter (fun x -> x#draw) d ;;
val draw_list : drawable list -> unit = <fun>
```

We've defined `drawable` as a SUPERTYPE of `display_elt`. It's a supertype because anything that can be done with a `drawable` can be done with a `display_elt`, but also potentially with other classes as well (namely, any that have a `draw` method). The idea is that an object with a "wider" interface (a subtype, like `display_elt`) can be used where an object with a "narrower" interface (a supertype, like `drawable`) is needed.

There is a family resemblance in this idea to polymorphism. Any function with a more polymorphic type (like `'a -> 'a list`, say) can be used where an object with a less polymorphic type (like `int -> int list`) is needed.

#### Exercise 180

Test out this polymorphism subtyping behavior in OCaml by defining two functions `mono : int -> int list` and `poly : 'a -> 'a list`, along with a function `need : (int -> int list) -> int list`. Then apply `need` to both `mono` and `poly`, thereby showing that `need` works with an argument of its required type (`int -> int list`) and also a subtype thereof (`'a -> 'a list`).

Anything that's a `display_elt` or inherits from `display_elt` or satisfies the `display_elt` interface will have at least the functionality of a `drawable`. So they are subtypes of `drawable`.<sup>4</sup>

The advantage of subtyping – allowing functions with a wider interface to be used where one with a narrower interface is called for – is just the advantage of polymorphism. It allows reuse of functionality, which redounds to the benefit of the edict of irredundancy. Rather than reimplement functions for the different interface "widths", we reuse them instead. We'll see that OCaml allows this kind of reuse, though with a little less automaticity than the reuse from polymorphism.

It ought to be the case, for instance, that, as `display_elt` is a subtype of `drawable`, our revision of `draw_scene` to apply to `drawable` objects ought to allow scenes composed of `display_elt` objects. Let's try it.

```
# let test scene =
#   try
#     G.open_graph "";
#     G.resize_window 300 300;
#     G.clear_graph ();
#     draw_list scene;
```

<sup>4</sup> There would seem to be a correlation between subclasses and subtypes. Of course, not all subtypes are subclasses; they may not be related by inheritance. But in a subclass, you have all the functionality of the superclass, plus you can add some more. So are subclasses always subtypes?

No. For instance, in the subclass, you could redefine a method to have a more restrictive signature. In that case, the subclass would not be a subtype; it would have a narrower interface (at least for that method), not a wider one.

```
#      ignore (G.read_key ())
#  with
#    exn -> (G.close_graph (); raise exn) ;;
val test : drawable list -> unit = <fun>
```

The type of `test` shows that it now takes a `drawable list` argument.

We apply it to our scene.

```
# test scene ;;
Line 1, characters 5-10:
1 | test scene ;;
^~~~~~
Error: This expression has type border_rect list
       but an expression was expected of type drawable list
       Type border_rect = display_elt is not compatible with type
                     drawable = < draw : unit >
The second object type has no method get_color
```

But the `draw_list` call no longer works. We've tripped over a limitation in OCaml's type inference. A subtype ought to be allowed where a supertype is needed, as it is in the case of polymorphic subtypes of less polymorphic supertypes. But in the case of class subtyping, OCaml is not able to perform the necessary type inference to view the subtype as the supertype and use it accordingly. We have to give the type inference system a hint.

We want the call to `draw_list` to view `scene` not as its `display_elt` list subtype but rather as the `drawable list` supertype. We use the `:>` operator to specify that view. The expression `scene :> drawable list` specifies `scene` viewed as a `drawable list`.

```
# let test scene =
#   try
#     G.open_graph "";
#     G.resize_window 300 300;
#     G.clear_graph ();
#     draw_list (scene :> drawable list);
#     ignore (G.read_key ())
#   with
#     exn -> (G.close_graph (); raise exn) ;;
val test : #drawable list -> unit = <fun>

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

Voila! The scene (Figure 18.7) appears. A little advice to the type inference mechanism has resolved the problem.

### 18.6 Problem section: Object-oriented counters

Here is a class type and class definition for “counter” objects. Each object maintains an integer state that can be “bumped” by adding

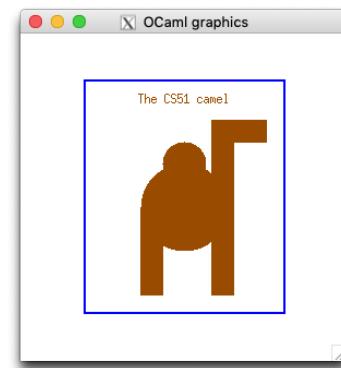


Figure 18.7: The rendered test scene.

an integer. The interface guarantees that only the two methods are revealed.

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

class counter : counter_interface =
object
  val mutable state = 0
  method bump n = state <- state + n
  method get_state = state
end ;;
```

#### **Problem 181**

Write a class definition for a class `loud_counter` obeying the same interface that works identically, except that it also prints the resulting state of the counter each time the counter is bumped.

#### **Problem 182**

Write a class type definition for an interface `reset_counter_interface`, which is just like `counter_interface` except that it has an additional method of no arguments intended to reset the state back to zero.

#### **Problem 183**

Write a class definition for a class `loud_reset_counter` satisfying the `reset_counter_interface` that implements a counter that both allows for resetting and is “loud” (printing the state whenever a bump or reset occurs).

### *18.7 Supplementary material*

- [Lab 16: Object-oriented programming](#)
- [Lab 17: Objects and classes](#)
- [Problem set A.8: Force-directed graph drawing](#)
- [Problem set A.9: Simulating an infectious process](#)

# 19

## *Semantics: The environment model*

The addition of mutability – which enables impure programming paradigms like imperative and procedural programming, with its potential for efficiencies in both time and space, and enables lazy and object-oriented programming as well – comes at a cost. Leibniz’s law no longer applies. One and the same expression in the same context can evaluate to different values, making reasoning about programs more difficult.

That complexity ramifies in providing explicit semantics for the language as well. The simple substitution semantics of Chapter 13 is no longer sufficient. For that reason, and looking forward to the implementation of an interpreter for a larger fragment of OCaml (Chapter A), we revisit the formal substitution semantics from Chapter 13, modifying and augmenting it to provide a rigorous semantics for references and assignment, showing where the additional complexity arises and clarifying issues such as scope, side effects, and order of evaluation.

### 19.1 *Review of substitution semantics*

Recall from Section 13.6 the abstract syntax of a simple functional language with arithmetic:

```
 $\langle binop \rangle ::= + | - | * | /$ 
 $\langle var \rangle ::= x | y | z | \dots$ 
 $\langle expr \rangle ::= \langle integer \rangle$ 
  |  $\langle var \rangle$ 
  |  $\langle expr_1 \rangle \langle binop \rangle \langle expr_2 \rangle$ 
  |  $\text{let } \langle var \rangle = \langle expr_{def} \rangle \text{ in } \langle expr_{body} \rangle$ 
  |  $\text{fun } \langle var \rangle \rightarrow \langle expr_{body} \rangle$ 
  |  $\langle expr_{fun} \rangle \langle expr_{arg} \rangle$ 
```

The semantics for this language was provided through the apparatus of evaluation rules, which defined derivations for judgements of the form

$$P \Downarrow v$$

where  $P$  is an expression and  $v$  is its value (a simplified expression that means the same and that cannot be further evaluated).

The substitution semantics is sufficient for this simple language because it is a pure functional programming language. But binding constructs like `let`, `let rec`, and `fun` are awkward to implement, and extending the language to handle references, mutability, and imperative programming is quite challenging if not impossible. For that reason, we start by modifying the substitution semantics to make use of an ENVIRONMENT that stores a mapping from variables to their values. In the next two sections, we develop the environment semantics for the language of Chapter 13 in two variants: a dynamic environment semantics and a lexical environment semantics. We then augment the environment semantics with a model of a mutable store to allow for reference values and their assignment.

## 19.2 Environment semantics

In an environment semantics, instead of substituting for variables the value that they specify, we directly model a mapping between variables and their values, which we call an ENVIRONMENT. We use the following notation for mappings in the semantics. A mapping from elements, say,  $x, y, z$ , to elements  $a, b, c$ , respectively, will be notated as  $\{x \mapsto a; y \mapsto b; z \mapsto c\}$ . The notation purposefully evokes the OCaml record notation, since a record also provides a kind of mapping from a finite set of elements (labels) to associated values. It also evokes, through the use of the  $\mapsto$  symbol, the idea of substitution, as these mappings will replace substitutions in the environment semantics.

Indeed, the environments that give their name to environment semantics are just such mappings – from variables to their values. We'll conventionally use the symbol  $E$  and its primed versions ( $E', E'', \dots$ ) as metavariables standing for environments. The empty environment will be notated  $\{\}$ , and the environment  $E$  augmented so as to add the mapping of the variable  $x$  to the value  $v$  will be notated  $E\{x \mapsto v\}$ . To look up what value an environment  $E$  maps a variable  $x$  to, we use **Euler's function application notation**:  $E(x)$ .

Having introduced the necessary notation, we turn to modifying the substitution semantics to use environments instead.

### 19.2.1 Dynamic environment semantics

Recall that the substitution semantics is given through a series of rules defining judgements of how expressions evaluate to values. (Reviewing Figure 13.5 may refresh your memory.)

In an environment semantics, expressions aren't evaluated in isolation. Rather, they are evaluated in the context of an environment that specifies which variables have which values. Instead of defining rules for  $P$  evaluating to  $v$  (written as the judgement  $P \Downarrow v$ ), we define rules for  $P$  evaluating to  $v$  in an environment  $E$  (written as the judgement  $E \vdash P \Downarrow v$ ). The rule for evaluating numbers, for instance, becomes

$$E \vdash \bar{n} \Downarrow \bar{n} \quad (R_{int})$$

stating that “in environment  $E$  a numeral  $\bar{n}$  evaluates to itself (independent of the environment)”, and the rule for addition provides the environment as context for evaluating the subexpressions:

$$\begin{array}{c} E \vdash P + Q \Downarrow \\ \left| \begin{array}{l} E \vdash P \Downarrow \bar{m} \\ E \vdash Q \Downarrow \bar{n} \end{array} \right. \\ \Downarrow \bar{m+n} \end{array} \quad (R_+)$$

Glossing again, the rule says “to evaluate an expression of the form  $P + Q$  in an environment  $E$ , first evaluate  $P$  in the environment  $E$  to an integer value  $\bar{m}$  and  $Q$  in the environment  $E$  to an integer value  $\bar{n}$ . The value of the full expression is then the integer literal representing the sum of  $m$  and  $n$ .”

To construct a derivation for a whole expression using these rules, we start in the empty environment  $\{\}$ . For instance, a derivation for the expression  $3 + 5$  would be

$$\begin{array}{c} \{\} \vdash 3 + 5 \Downarrow \\ \left| \begin{array}{l} \{\} \vdash 3 \Downarrow 3 \\ \{\} \vdash 5 \Downarrow 5 \end{array} \right. \\ \Downarrow 8 \end{array}$$

So far, not much is different from the substitution semantics. The differences show up in the handling of binding constructs like `let`. Recall the  $R_{let}$  rule for `let` binding in the substitution semantics.

$$\begin{array}{c} \text{let } x = D \text{ in } B \Downarrow \\ \left| \begin{array}{l} D \Downarrow v_D \\ B[x \mapsto v_D] \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{let})$$

This rule specifies that an expression of the form `let x = D in B` evaluates to the value  $v_B$ , whenever the definition expression  $D$  evaluates to  $v_D$  and the body expression  $B$  after substituting  $v_B$  for the variable  $x$  evaluates to  $v_B$ .

The corresponding environment semantics rule doesn't substitute into  $B$ . It evaluates  $B$  directly, but it does so in an environment augmented with a new binding of  $x$  to its value  $v_D$ :

$$\begin{array}{c} E \vdash \text{let } x = D \text{ in } B \Downarrow \\ \left| \begin{array}{l} E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{let})$$

According to this rule, “to evaluate an expression of the form `let x = D in B` in an environment  $E$ , first evaluate  $D$  in  $E$  resulting in a value  $v_D$  and then evaluate the body  $B$  in an environment that is like  $E$  except that the variable  $x$  is mapped to the value  $v_D$ . The result of this latter evaluation,  $v_B$ , is the value of the `let` expression as a whole.”

In the substitution semantics, we will have substituted away all of the bound variables in a closed expression, so no rule is needed for evaluating variables themselves. But in the environment semantics, since no substitution occurs, we'll need to be able to evaluate expressions that are just variables. Presumably, those variables will have values in the prevailing environment; we'll just look them up.

$$E \vdash x \Downarrow E(x) \quad (R_{var})$$

A gloss for this rule is “evaluating a variable  $x$  in an environment  $E$  yields the value of  $x$  in  $E$ .”

Putting all these rules together, we can derive a value for the expression `let x = 3 in x + x`:

$$\begin{array}{c} \emptyset \vdash \text{let } x = 3 \text{ in } x + x \Downarrow \\ \left| \begin{array}{l} \emptyset \vdash 3 \Downarrow 3 \\ \{x \mapsto 3\} \vdash x + x \Downarrow \\ \left| \begin{array}{l} \{x \mapsto 3\} \vdash x \Downarrow 3 \\ \{x \mapsto 3\} \vdash x \Downarrow 3 \end{array} \right. \\ \Downarrow 6 \end{array} \right. \\ \Downarrow 6 \end{array}$$

The derivation makes clear how the environment semantics differs from the substitution semantics. Rather than replacing a bound variable with its value, we add the bound variable with its value to the environment; when an occurrence of the variable is reached, we simply look up its value in the environment.

**Exercise 184**

Construct the derivation for the expression

```
let x = 3 in
let y = 5 in
x + y ;;
```

**Exercise 185**

Construct the derivation for the expression

```
let x = 3 in
let x = 5 in
x + x ;;
```

Continuing the translation of the substitution semantics directly into an environment semantics, we turn to functions and their application. Maintaining functions as values is reflected in this simple rule:

$$E \vdash \text{fun } x \rightarrow P \Downarrow \text{fun } x \rightarrow P \quad (R_{\text{fun}})$$

and the application of a function to its argument again adds the argument's value to the environment used in evaluating the body of the function:

$$\begin{array}{c} E \vdash P \ Q \Downarrow \\ \left| \begin{array}{c} E \vdash P \Downarrow \text{fun } x \rightarrow B \\ E \vdash Q \Downarrow v_Q \\ E\{x \mapsto v_Q\} \vdash B \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{\text{app}})$$

**Exercise 186**

Provide glosses for these two rules.

We can try the example from Section 13.6:

$(\text{fun } x \rightarrow x + x) (3 * 4)$

whose evaluation to 24 is captured by the following derivation:

$$\begin{array}{c} \{\} \vdash (\text{fun } x \rightarrow x + x) (3 * 4) \\ \Downarrow \\ \left| \begin{array}{c} \{\} \vdash (\text{fun } x \rightarrow x + x) \Downarrow (\text{fun } x \rightarrow x + x) \\ \{\} \vdash 3 * 4 \Downarrow \\ \left| \begin{array}{c} \{\} \vdash 3 \Downarrow 3 \\ \{\} \vdash 4 \Downarrow 4 \\ \Downarrow 12 \end{array} \right. \\ \{\} \vdash 12 \Downarrow \\ \{\} \vdash (\text{fun } x \rightarrow x + x) \Downarrow \\ \left| \begin{array}{c} \{\} \vdash (\text{fun } x \rightarrow x + x) \Downarrow \\ \{\} \vdash 12 \Downarrow \\ \{\} \vdash 12 \Downarrow \\ \Downarrow 24 \end{array} \right. \end{array} \right. \end{array}$$

The full set of dynamic environment semantics rules so far is presented in Figure 19.1.

$$E \vdash \bar{n} \Downarrow \bar{n} \quad (R_{int})$$

$$E \vdash x \Downarrow E(x) \quad (R_{var})$$

$$E \vdash \text{fun } x \rightarrow P \Downarrow \text{fun } x \rightarrow P \quad (R_{fun})$$

$$\begin{array}{c} E \vdash P + Q \Downarrow \\ \left| \begin{array}{l} E \vdash P \Downarrow \bar{m} \\ E \vdash Q \Downarrow \bar{n} \end{array} \right. \\ \Downarrow \bar{m+n} \end{array} \quad (R_+)$$

(and similarly for other binary operators)

$$\begin{array}{c} E \vdash \text{let } x = D \text{ in } B \Downarrow \\ \left| \begin{array}{l} E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{let})$$

$$\begin{array}{c} E \vdash P \ Q \Downarrow \\ \left| \begin{array}{l} E \vdash P \Downarrow \text{fun } x \rightarrow B \\ E \vdash Q \Downarrow v_Q \\ E\{x \mapsto v_Q\} \vdash B \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{app})$$

*Problems with the dynamic semantics* The environment semantics captured in these rules (Figure 19.1) seems like it should generate the same evaluations as the substitution semantics (Figure 13.5). After all, the only difference would seem to be that instead of the binding constructs (`let` and `fun`) substituting a value for the variables they bind, they place the value in the environment, to be retrieved when the variables they bind need them. But there are subtle differences, hidden in the decision as to which variable occurrences see which values.

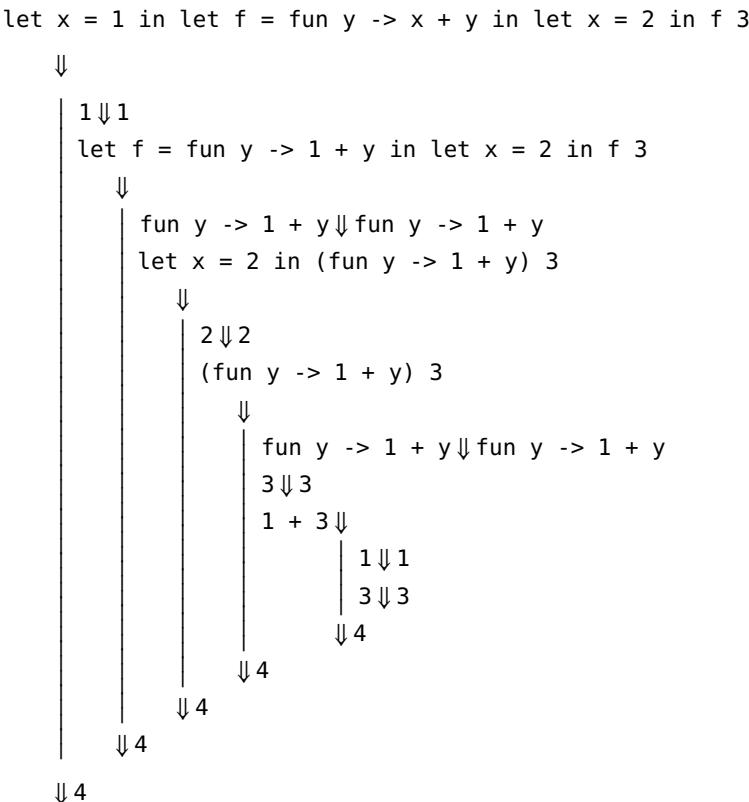
Recall (Section 5.5) that in OCaml the connection between occurrences of variables and the binding constructs they are bound by is

Figure 19.1: Dynamic environment semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

determined by the *lexical structure* of the code. For instance, in the expression

```
# let x = 1 in
# let f = fun y -> x + y in
# let x = 2 in
# f 3 ;;
Line 3, characters 4-5:
3 | let x = 2 in
      ^
Warning 26 [unused-var]: unused variable x.
- : int = 4
```

the highlighted occurrence of the variable  $x$  is bound by the outer let  $x$ , not the inner. For that reason, the result of evaluating the expression is 4, and not 5. The substitution semantics reflects this fact, as seen in the derivation



But the environment semantics evaluates this expression to 5.

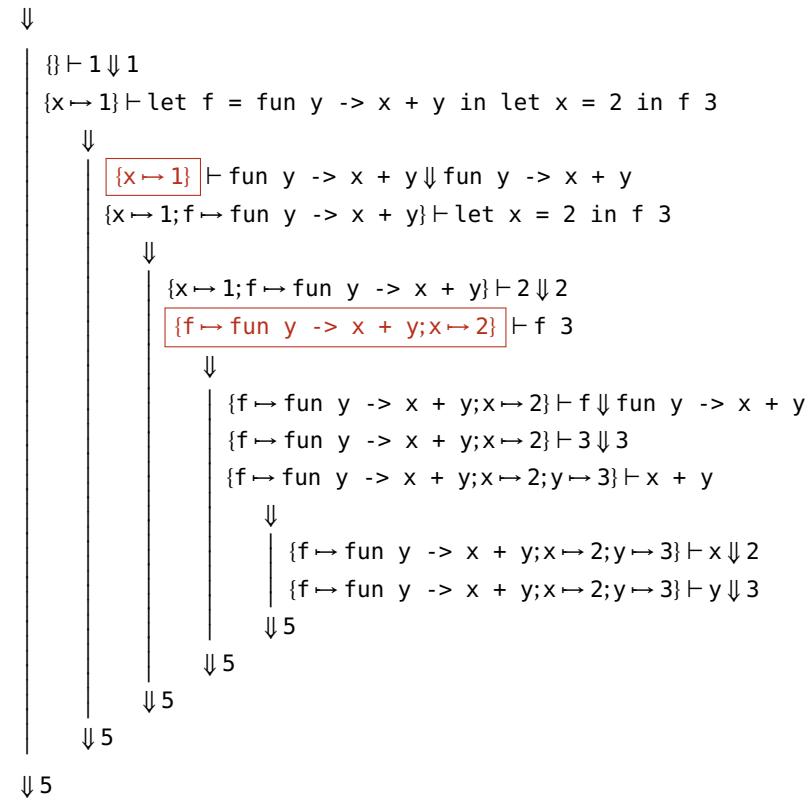
### **Exercise 187**

Before proceeding, see if you can construct the derivation for this expression according to the environment semantics rules. Do you see where the difference lies?

According to the environment semantics developed so far, a deriva-

tion for this expression proceeds as

$\{\} \vdash \text{let } x = 1 \text{ in let } f = \text{fun } y \rightarrow x + y \text{ in let } x = 2 \text{ in } f$



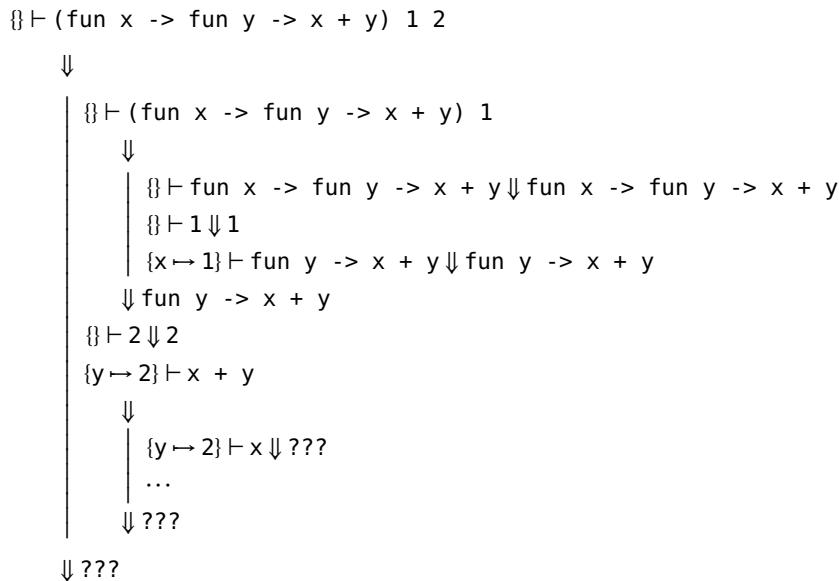
The crucial difference comes when augmenting the environment during application of the function  $\text{fun } y \rightarrow x + y$  to its argument. Examine closely the two highlighted environments in the derivation above. The first is the environment in force when the function is *defined*, the LEXICAL ENVIRONMENT of the function. The second is the environment in force when the function is *applied*, its DYNAMIC ENVIRONMENT. The environment semantics presented so far augments the dynamic environment with the new binding induced by the application. It manifests a DYNAMIC ENVIRONMENT SEMANTICS. But for consistency with the substitution semantics (which substitutes occurrences of a bound variable when the binding construct is defined, not applied), we should use the lexical environment, thereby manifesting a LEXICAL ENVIRONMENT SEMANTICS.

In Section 19.2.2, We'll develop a lexical environment semantics that cleaves more faithfully to the lexical scope of the substitution semantics, but first, we note some other divergences between dynamic and lexical semantics.

Consider this simple application of a curried function:

```
(fun x -> fun y -> x + y) 1 2
```

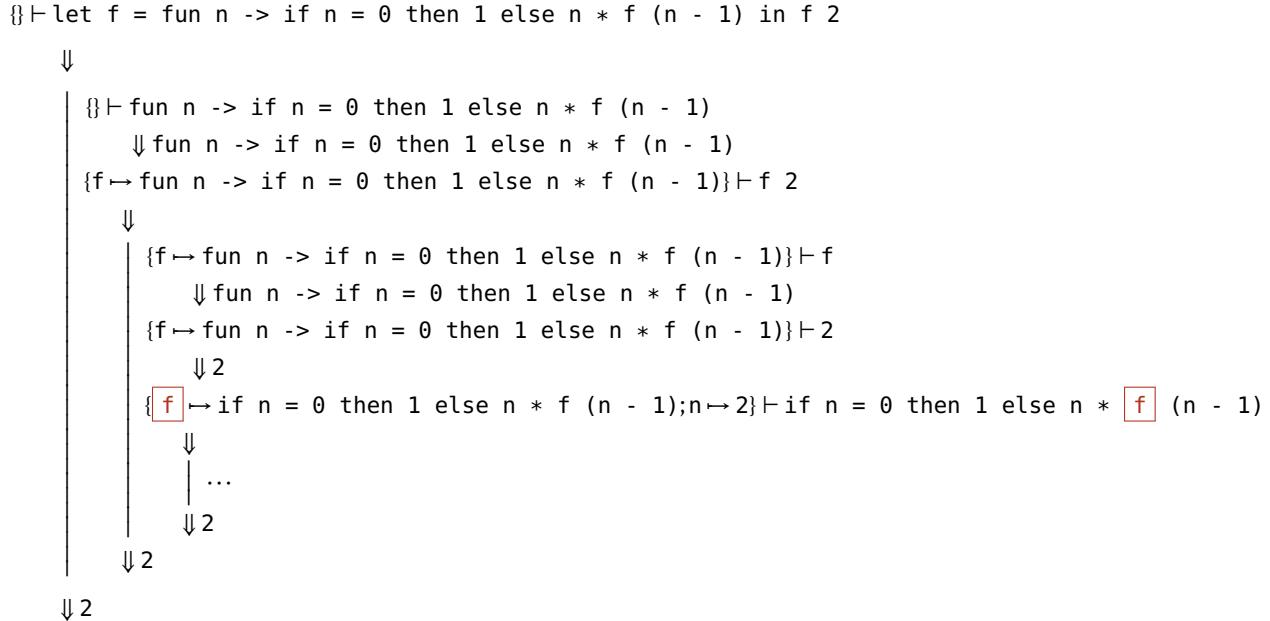
The substitution semantics rules specify that this expression evaluates to 3. But the dynamic semantics misbehaves:



We can start the derivation, but the dynamic environment available when we come to evaluate the  $x$  in the function body contains no binding for  $x$ ! (If only we had been evaluating the body in its lexical environment.) In a dynamic semantics, currying – so central to many functional idioms – becomes impossible.

On the other hand, under a dynamic semantics, recursion needs no special treatment. By using the dynamic environment in evaluating the definiendum of a let, the definition of the bound variable is already available. We revisit the derivation for factorial from Section 13.7, but

this time using the dynamic environment semantics:



Notice how the body of the function, with its free occurrence of the variable  $f$ , is evaluated in an environment in which  $f$  is bound to the function itself. By using the dynamic environment semantics rules, we get recursion “for free”. Consequently, the dynamic semantics rule for the `let rec` construction can simply mimic the `let` construction:

$$\begin{array}{c}
 E \vdash \text{let rec } x = D \text{ in } B \Downarrow \\
 \left| \begin{array}{l} E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B \end{array} \right. \quad (R_{letrec}) \\
 \Downarrow v_B
 \end{array}$$

To truly reflect the intended semantics of expressions in an environment semantics, we need to find a way of using the lexical environment for functions instead of the dynamic environment; we need a *lexical environment semantics*.

### 19.2.2 Lexical environment semantics

To modify the rules to provide a lexical rather than dynamic environment semantics, we must provide some way of capturing the lexical environment when functions are defined. The technique is to have functions evaluate not to themselves (awaiting the dynamic environment for interpretation of the variables within them), but rather to have them evaluate to a “package” containing the function and its lexical (defining) environment. This package is called a **CLOSURE**.<sup>1</sup>

<sup>1</sup> The term comes from the terminology of open versus closed expressions. Open expressions have free variables in them; closed expressions have none. By capturing the defining environment, we essentially use it to close the free variables in the function. The closure thus turns what would otherwise be an open expression into a closed expression.

We'll note the closure that packages together a function  $P$  and its environment  $E$  as  $[E \vdash P]$ . In evaluating a function, then, we merely construct such a closure, capturing the function's defining environment.

$$E \vdash \text{fun } x \rightarrow P \Downarrow [E \vdash \text{fun } x \rightarrow P] \quad (R_{\text{fun}})$$

We make use of closures constructed in this way when the function is applied:

$$\begin{aligned} E_d \vdash P \quad Q \Downarrow \\ \left| \begin{array}{l} E_d \vdash P \Downarrow [E_l \vdash \text{fun } x \rightarrow B] \\ E_d \vdash Q \Downarrow v_Q \\ E_l \{x \mapsto v_Q\} \vdash B \Downarrow v_B \end{array} \right. \quad (R_{\text{app}}) \\ \Downarrow v_B \end{aligned}$$

Rather than augmenting the dynamic environment  $E_d$  in evaluating the body, we augment the lexical environment  $E_l$  extracted from the closure.

The lexical environment semantics properly reflects the intended semantics for several of the problematic examples in Section 19.2.1, as demonstrated in the following exercises. However, the handling of recursion still requires some further work, which we'll return to in Section 19.4.

#### Exercise 188

Carry out the derivation using the lexical environment semantics for the expression

```
let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 3 ;;
```

What value does it evaluate to under the lexical environment semantics?

#### Exercise 189

Carry out the derivation using the lexical environment semantics for the expression

```
(fun x -> fun y -> x + y) 1 2 ;;
```

#### Problem 190

In problem 155, you evaluated several expressions as OCaml would, with lexical scoping. Which of those expressions would evaluate to a different value using dynamic scoping?

### 19.3 Conditionals and booleans

In Section 13.5, exercises asked you to develop abstract syntax and substitution semantics rules for booleans and conditionals. In this section, we call for similar rules for environment semantics (applicable to dynamic or lexical variants).

**Exercise 191**

Adjust the substitution semantics rules for booleans from Exercise 135 to construct environment semantics rules for the constructs.

**Exercise 192**

Adjust the substitution semantics rules for conditional expressions (`if () then () else ()`) from Exercise 136 to construct environment semantics rules for the construct.

### 19.4 Recursion

The dynamic environment semantics already allows for recursion – in fact, too much recursion – because of its dynamic nature. Think about an ill-formed “almost-recursive” function, like

```
let f = fun x -> if x = 0 then 1 else f (x - 1) in f 1 ;;
```

It's ill-formed because the lack of a `rec` keyword means that the `f` in the definition part ought to be unbound. But it works just fine in the dynamic environment semantics. When `f 1` is evaluated in the dynamic environment in which `f` is bound to `fun x -> if x = 0 then 1 else f (x - 1)`, all of the subexpressions of the definiens, including the occurrence of `f` itself, will be evaluated in an augmentation of that environment, so the “recursive” occurrence of `f` will obtain a value. (It is perhaps for this reason that the earliest implementations of functional programming languages, the original versions of LISP, used a dynamic semantics.)

The lexical semantics, of course, does not benefit from this fortuitous accident of definition. The lexical environment in force when `f` is defined is empty, and thus, when the body of `f` is evaluated, it is the empty environment that is augmented with the argument `x` bound to 1. There is no binding for the recursively invoked `f`, and the derivation cannot be completed – consistent, by the way, with how OCaml behaves:

```
# let f = fun x -> if x = 0 then 1 else f (x - 1) in f 1 ;;
Line 1, characters 38-39:
1 | let f = fun x -> if x = 0 then 1 else f (x - 1) in f 1 ;;
^
Error: Unbound value f
Hint: If this is a recursive definition,
you should add the 'rec' keyword on line 1
```

To allow for recursion in the lexical environment semantics, we should add a special rule for `let rec` then. A `let rec` expression is built from three parts: a variable name (`(x)`), a definition expression (`(D)`), and a body (`(B)`). To evaluate it, we ought to first evaluate the definition part `D`, but using what environment? Any functions inside the definition part will see this environment as their lexical environment, to

be captured in a closure. We'll thus want to make a value for  $x$  available in that environment. But what will we use for the value of  $x$  in the environment? We can't merely map  $x$  to the definition  $D$ , with its free occurrence of  $x$ ; that just postpones the problem.

In one sense, it doesn't matter what value we use for  $x$  in evaluating the definition  $D$ , because in evaluating  $D$ , we won't (or at least better not) make use of  $x$  directly, as for instance in

```
# let rec x = x + 1 in x ;;
Line 1, characters 12-17:
1 | let rec x = x + 1 in x ;;
          ^^^^
Error: This kind of expression is not allowed as right-hand side of
`let rec'
```

That wouldn't be a well-founded recursion. Instead, the occurrences of  $x$  in  $D$  will have to be in contexts where they are not evaluated. Canonically, that would be within an unapplied function, like the factorial example

```
# let rec f = fun n -> if n = 0 then 1 else n * f (n - 1) in f 2 ;;
- : int = 2
```

Because of this requirement for well-founding of the recursion, whatever value we use for  $x$ , we'll be able to evaluate the definition to some value, call it  $v_D$ . That value, however, may involve closures that capture the binding for  $x$ , and we'll need to look up the value for  $x$  later in evaluating the body. Thus, the environment used in evaluating the body best have a binding for  $x$  to  $v_D$ .

These considerations call for the following approach to handling the semantics of `let rec` in an environment  $E$ . We start by forming an environment  $E'$  that extends  $E$  with a binding for  $x$ , but a binding that is *mutable*, so it can be changed later. Initially,  $x$  can be bound to some recognizable and otherwise ungenerable value, say, `Unassigned`. We evaluate the definition  $D$  in environment  $E'$  to a value  $v_D$ , which may capture  $E'$  (or extensions of it) in closures. We then *change* the value stored for  $x$  in  $E'$  to  $v_D$ , and evaluate the body  $B$  in  $E'$  (thus modified). By mutating the value bound to  $x$ , any closures that have captured  $E'$  will see this new value for  $x$  as well, so that (recursive) lookups of  $x$  in the body will see the evaluated  $v_D$  as well.

Because this approach relies on mutation, our notation for environment semantics isn't up to the task of formalizing this idea, and doing so is beyond the scope of this discussion, so we'll leave it at that for now. But once mutability is incorporated into the semantics – that was the whole point in moving to an environment semantics, remember – we'll revisit the issue and give appropriate rules for `let rec`.

Even without formal rules for `let rec`, you'll see in Chapter A how

this approach can be implemented in an interpreter for a language with a `let rec` construction.

$$E \vdash \bar{n} \Downarrow \bar{n} \quad (R_{int})$$

$$E \vdash x \Downarrow E(x) \quad (R_{var})$$

$$E \vdash \text{fun } x \rightarrow P \Downarrow [E \vdash \text{fun } x \rightarrow P] \quad (R_{fun})$$

$$\begin{array}{c} E \vdash P + Q \Downarrow \\ \left| \begin{array}{l} E \vdash P \Downarrow \bar{m} \\ E \vdash Q \Downarrow \bar{n} \end{array} \right. \\ \Downarrow \bar{m+n} \end{array} \quad (R_+)$$

(and similarly for other binary operators)

$$\begin{array}{c} E \vdash \text{let } x = D \text{ in } B \Downarrow \\ \left| \begin{array}{l} E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{let})$$

$$\begin{array}{c} E_d \vdash P \ Q \Downarrow \\ \left| \begin{array}{l} E_d \vdash P \Downarrow [E_l \vdash \text{fun } x \rightarrow B] \\ E_d \vdash Q \Downarrow v_Q \\ E_l\{x \mapsto v_Q\} \vdash B \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{app})$$

## 19.5 Implementing environment semantics

In Section 13.4.2, we presented an implementation of the substitution semantics in the form of a function `eval : expr -> expr`. Modifying it to follow the environment semantics requires just a few simple changes. First, evaluation is relative to an environment, so the `eval` function should take an additional argument, of type, say `env`. Second, under the lexical environment semantics, expressions evaluate to values that include more than just the pertinent subset of expressions. In particular, expressions may evaluate to closures, so that an extended

Figure 19.2: Lexical environment semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

notion of value, codified in a type `value` is needed. In summary, the type of `eval` should be `expr -> env -> value`.

The new `env` type, a simple mapping from variables to the values they are bound to, can be implemented as an association list

```
type env = (varid * value ref) list
```

and the `value` type can include expression values and closures in a simple variant type

```
type value =
| Val of expr
| Closure of (expr * env)
```

(The `env` data structure maps variables to mutable `value` refs to allow for the mutation required in implementing `let rec` as described in Section 19.4.) The carrying out of this exercise is the subject of Chapter A.

## 19.6 Semantics of mutable storage

In this section, we further extend the lexical environment semantics to allow for imperative programming with references and assignment. (As a byproduct, we'll have the infrastructure to provide a formal semantics rule for `let rec`.) To do so, we'll start by augmenting the syntax of the language, and then adjust the environment semantic rules so that the context of evaluation includes not only an environment, but also a model for the mutable storage that references require.

We'll start with adding to the syntax a unit value () and operators (`ref`, !, and `:=`) to manipulate reference values:

```
 $\langle binop \rangle ::= + | - | * | /$ 
 $\langle var \rangle ::= x | y | z | \dots$ 
 $\langle expr \rangle ::= \langle integer \rangle$ 
|  $\langle var \rangle$ 
|  $\langle expr_1 \rangle \langle binop \rangle \langle expr_2 \rangle$ 
|  $\text{let } \langle var \rangle = \langle expr_{def} \rangle \text{ in } \langle expr_{body} \rangle$ 
|  $\text{fun } \langle var \rangle \rightarrow \langle expr_{body} \rangle$ 
|  $\langle expr_{fun} \rangle \langle expr_{arg} \rangle$ 
| ()
|  $\text{ref } \langle expr \rangle$ 
| !
|  $\langle var \rangle := \langle expr \rangle$ 
```

The plan for handling references is to add a new kind of value, a LOCATION, which is an index or pointer into an abstract model of memory that we will call the STORE. A store  $S$  will be a finite mapping

(like the environment) from locations to values. So a reference to a value  $v$  will be a location  $l$  such that the store  $S$  maps  $l$  to  $v$ . Evaluation will need to be relative to a store in addition to an environment, so evaluation judgements will look like  $E, S \vdash P \Downarrow \dots$

Because the store can change as a side effect of evaluation (that's the whole point of mutability), the result of evaluation can't simply be a value. We'll need access to the modified store as well. So the right-hand side of the evaluation arrow  $\Downarrow$  will provide both a value and a store. Our final form for evaluation judgements is thus

$$E, S \vdash P \Downarrow v_P, S' .$$

(See Figure 19.3 for a breakdown of such a judgement.)

A semantic rule for references reflects these ideas:

$$\begin{array}{c} E, S \vdash \text{ref } P \Downarrow \\ | \quad E, S \vdash P \Downarrow v_P, S' \\ \Downarrow l, S' \{l \mapsto v_P\} \quad (\text{where } l \text{ is a new location}) \end{array} \quad (R_{\text{ref}})$$

According to this rule, “to evaluate an expression of the form `ref P` in an environment  $E$  and store  $S$ , we evaluate  $P$  in that environment and store, yielding a value  $v_P$  for  $P$  and a new store  $S'$  (as there may have been side effects to  $S$  in the evaluation). The value for the reference is a new location  $l$ , and as side effect, a new store that is  $S'$  augmented so that  $l$  maps to the value  $v_P$ .”

To dereference such a reference, as in an expression of the form `! P`,  $P$  will need to be evaluated to a location, and the value at that location retrieved.

### Exercise 19.3

Write a semantic rule for dereferencing references.

Finally, and most centrally to the idea of mutable storage, is *assignment* to a reference. Evaluating an assignment of the form `P := Q` involves evaluating  $P$  to a location  $l$  and evaluating  $Q$  to a value  $v_Q$ , and updating the store so that  $l$  maps to  $v_Q$ . Along the way, the various subevaluations may themselves have side effects leading to updated stores, which must be dealt with. For instance, starting with an environment  $E$  and store  $S$ , evaluating  $P$  may result in an updated store  $S'$ . That updated store would then be the store with respect to which  $Q$  would be evaluated, leading to a possibly updated store  $S''$ . It is this final store that would be augmented with the new assignment. A rule

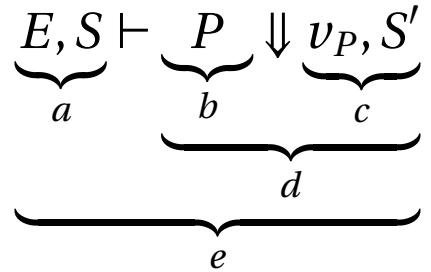


Figure 19.3: Anatomy of an evaluation judgement. (a) The context of evaluation, including an environment  $E$  and a store  $S$ . (b) The expression to be evaluated. (c) The result of the evaluation, a value and a store mutated by side effect. (d) The evaluation of  $P$  to its result. (e) The judgement as a whole. “In the environment  $E$  and store  $S$ , expression  $P$  evaluates to value  $v_P$  with modified store  $S'$ .”

specifying this semantics is

$$\begin{array}{c}
 E, S \vdash P := Q \Downarrow \\
 \left| \begin{array}{l} E, S \vdash P \Downarrow l, S' \\ E, S' \vdash Q \Downarrow v_Q, S'' \end{array} \right. \quad (R_{assign}) \\
 \Downarrow (), S''\{l \mapsto v_Q\}
 \end{array}$$

The important point of the rule is the update to the store. But like all evaluation rules, a value must be returned for the expression as a whole. Here, we've simply returned the unit value () .

#### Exercise 19.4

In the presence of side effects, sequencing (with ; ) becomes important. Write an evaluation rule for sequencing.

To complete the semantics of mutable state, the remaining rules must be modified to use and update stores appropriately. Figure 19.4 provides a full set of rules.

As an example of the deployment of these semantic rules, we consider the expression

```
let x = ref 3 in
  x := 42;
  !x
```

Here is the derivation in full.

$$\begin{array}{c}
 \emptyset, \emptyset \vdash \text{let } x = \text{ref } 3 \text{ in } x := 42; !x \\
 \Downarrow \\
 \left| \begin{array}{c} \emptyset, \emptyset \vdash \text{ref } 3 \Downarrow \\ \left| \begin{array}{c} \emptyset, \emptyset \vdash 3 \Downarrow 3, \emptyset \\ \Downarrow l_1, \{l_1 \mapsto 3\} \end{array} \right. \end{array} \right. \\
 \{x \mapsto l_1, \{l_1 \mapsto 3\} \vdash x := 42; !x \\
 \Downarrow \\
 \left| \begin{array}{c} \{x \mapsto l_1, \{l_1 \mapsto 3\} \vdash x := 42 \\ \Downarrow \\ \left| \begin{array}{c} \{x \mapsto l_1, \{l_1 \mapsto 3\} \vdash x \Downarrow l_1, \{l_1 \mapsto 3\} \\ \left| \begin{array}{c} \{x \mapsto l_1, \{l_1 \mapsto 3\} \vdash 42 \Downarrow 42, \{l_1 \mapsto 3\} \\ \Downarrow () , \{l_1 \mapsto 42\} \end{array} \right. \end{array} \right. \\
 \{x \mapsto l_1, \{l_1 \mapsto 42\} \vdash !x \\
 \Downarrow \\
 \left| \begin{array}{c} \{x \mapsto l_1, \{l_1 \mapsto 42\} \vdash x \Downarrow l_1, \{l_1 \mapsto 42\} \\ \Downarrow \\ \Downarrow 42, \{l_1 \mapsto 42\} \end{array} \right. \end{array} \right. \\
 \Downarrow 42, \{l_1 \mapsto 42\}
 \end{array}$$

|                                                                                                                                                                                                                                                                                        |             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| $E, S \vdash \bar{n} \Downarrow \bar{n}, S$                                                                                                                                                                                                                                            | $(R_{int})$ |
| $E, S \vdash x \Downarrow E(x), S$                                                                                                                                                                                                                                                     | $(R_{var})$ |
| $E, S \vdash \text{fun } x \rightarrow P \Downarrow [E \vdash \text{fun } x \rightarrow P], S$                                                                                                                                                                                         | $(R_{fun})$ |
| $E, S \vdash P + Q \Downarrow$<br>$\left  \begin{array}{l} E, S \vdash P \Downarrow \bar{m}, S' \\ E, S' \vdash Q \Downarrow \bar{n}, S'' \end{array} \right.$<br>$\Downarrow \bar{m+n}, S''$                                                                                          | $(R_+)$     |
| (and similarly for other binary operators)                                                                                                                                                                                                                                             |             |
| $E, S \vdash \text{let } x = D \text{ in } B \Downarrow$<br>$\left  \begin{array}{l} E, S \vdash D \Downarrow v_D, S' \\ E\{x \mapsto v_D\}, S' \vdash B \Downarrow v_B, S'' \end{array} \right.$<br>$\Downarrow v_B, S''$                                                             | $(R_{let})$ |
| $E_d, S \vdash P \ Q \Downarrow$<br>$\left  \begin{array}{l} E_d, S \vdash P \Downarrow [E_l \vdash \text{fun } x \rightarrow B], S' \\ E_d, S' \vdash Q \Downarrow v_Q, S'' \\ E_l\{x \mapsto v_Q\}, S'' \vdash B \Downarrow v_B, S''' \end{array} \right.$<br>$\Downarrow v_B, S'''$ | $(R_{app})$ |

Figure 19.4: Lexical environment semantics rules for evaluating expressions, for a functional language with naming, arithmetic, and mutable storage.

|                                        |                                                                                                                                                       |                               |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------|
| $E, S \vdash \text{ref } P \Downarrow$ | $\begin{array}{l}   \\ E, S \vdash P \Downarrow v_P, S' \\ \Downarrow l, S' \{l \mapsto v_P\} \end{array}$                                            | (where $l$ is a new location) |
| $E, S \vdash ! P \Downarrow$           | $\begin{array}{l}   \\ E, S \vdash P \Downarrow l, S' \\ \Downarrow S'(l), S' \end{array}$                                                            |                               |
| $E, S \vdash P := Q \Downarrow$        | $\begin{array}{l}   \\ E, S \vdash P \Downarrow l, S' \\   \\ E, S' \vdash Q \Downarrow v_Q, S'' \\ \Downarrow (), S'' \{l \mapsto v_Q\} \end{array}$ |                               |
| $E, S \vdash P ; Q \Downarrow$         | $\begin{array}{l}   \\ E, S \vdash P \Downarrow (), S' \\   \\ E, S' \vdash Q \Downarrow v_Q, S'' \\ \Downarrow v_Q, S'' \end{array}$                 |                               |

Figure 19.4: (continued) Lexical environment semantics rules for evaluating expressions, for a functional language with naming, arithmetic, and mutable storage.

### 19.6.1 Lexical environment semantics of recursion

The extended language with references and assignment is sufficient to provide a semantics for the recursive `let rec` construct. A simple way to observe this is to reconstruct a `let rec` expression of the form

```
let rec x = D in B
```

as syntactic sugar for an expression that caches the recursion out using just the trick described in Section 19.4: first assigning to  $x$  a mutable reference to a special unassigned value, then evaluating the definition  $D$ , replacing the value of  $x$  with the evaluated  $D$ , and finally, evaluating  $B$  in that environment. We can carry out that recipe with the following expression, which we can think of as the desugared `let rec`:

```
let x = ref unassigned in
  x := D[x ↦ !x];
  B[x ↦ !x]
```

(Since we've changed  $x$  to a reference type, we need to replace occurrences of  $x$  in  $D$  and  $B$  with  $!x$  to retrieve the referenced value.)

One way to verify that this approach works is to test it out in OCaml itself. Take this application of the factorial function, for instance:

```
# let rec f = fun n -> if n = 0 then 1
#                      else n * f (n - 1) in
# f 4 ;;
- : int = 24
```

Desugaring it as above, we get

```
# let unassigned = fun _ -> failwith "unassigned" ;;
val unassigned : 'a -> 'b = <fun>

# let f = ref unassigned in
# (f := fun n -> if n = 0 then 1
#                      else n * !f (n - 1));
# !f 4 ;;
- : int = 24
```

(To serve as the “unassigned” value, we define `unassigned` to simply raise an exception.)

This expression, note, makes use of only the language constructs provided in the semantics in the previous section. That semantics, with its lexical environment and mutable store, thus has enough expressivity for capturing the approach to recursion described informally in Section 19.4. In fact, we could even provide a semantic rule for `let rec` by carrying through the semantics for the desugared expression. This leads to the following `let rec` semantic rule for the `let rec` construction:

$$\begin{array}{c} E, S \vdash \text{let } \text{rec } x = D \text{ in } B \Downarrow \\ | \quad E\{x \mapsto l\}, S\{l \mapsto \text{unassigned}\} \vdash D[x \mapsto !x] \Downarrow v_D, S' \\ | \quad E\{x \mapsto l\}, S'\{l \mapsto v_D\} \vdash B[x \mapsto !x] \Downarrow v_B, S'' \\ \Downarrow v_B, S'' \\ (R_{\text{letrec}}) \end{array}$$

#### Problem 195

For the formally inclined, prove that the semantic rule for `let rec` above is equivalent to the syntactic sugar approach.

### 19.7 Supplementary material

- Lab 18: Environment semantics
- Lab 19: Synthesis: Cellular automata
- Lab 20: Synthesis: Digital halftoning

# 20

## *Concurrency*

In 1965, Gordon Moore, one of the founders of the pioneering electronics company Fairchild Semiconductor, noted the exponential growth in the number of components that were being placed on integrated circuit chips, the building blocks of all kinds of electronics but especially of computers. Extrapolating from just four points on a chart (Figure 20.1), Moore saw that the number of integrated circuit components had been growing at “a rate of roughly a factor of two per year”, and he expected that rate to continue for the foreseeable future. Ten years later, he revised his estimate to a doubling per two years. This prediction became “Moore’s law”, and has been generalized to many other aspects of computer technology and performance.

The generalized form of Moore’s law would have it that computer performance, measured, say, in total number of instructions executed per second, should grow exponentially as well, as indeed it has. Empirical data on the number of standardized operations performed per second, charted as squares in Figure 20.2, shows that Moore’s law as applied to the performance of microprocessor chips has held up remarkably well for many decades. Data on clock speed, the rate at which individual instructions can be executed, charted as circles, shows a different story. The clock speed of the processors showed the same exponential growth through the mid to late 2000’s, but flattened after that. What could account for this differential? If the processors weren’t executing instructions faster, how could they be executing more instructions in the same amount of time. The answer is given by the third series, shown as triangles in Figure 20.2, which graphs the number of processors per chip. Over the last decade or so, we’ve seen a regular rise in the number of processors per chip, making up the difference in Moore’s law by having *multiple instructions executed in parallel*.

These days, specialized architectures like graphics processing units (GPUs) and AI accelerators take advantage of even larger scale paral-

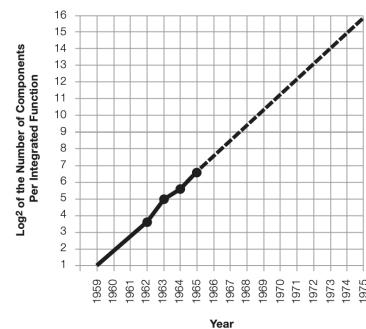


Figure 20.1: Gordon Moore’s chart on the basis of which his 1965 eponymous “law” was extrapolated.

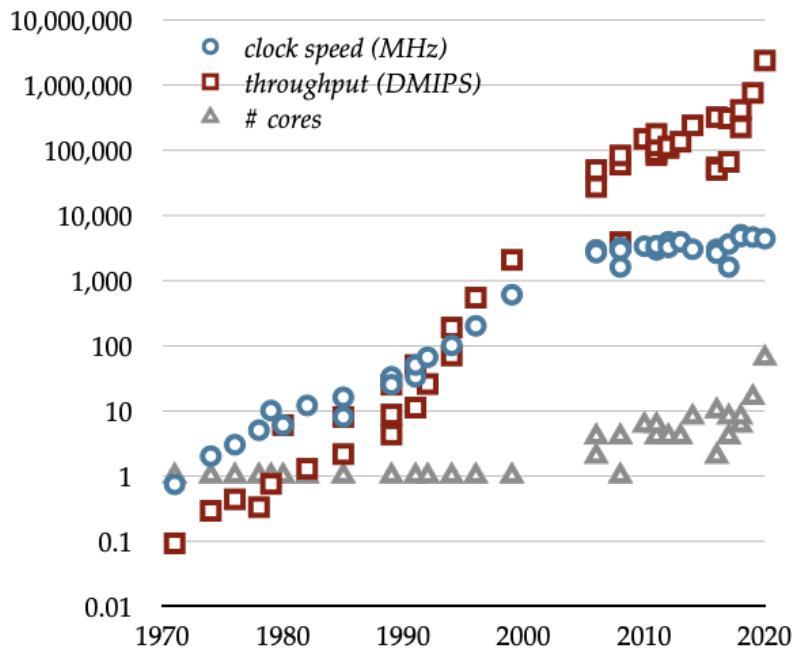


Figure 20.2: Chart showing growth in clock speed (in megaHertz (MHz), as circles), throughput (in Dhrystone millions of instructions per second (DMIPS), as squares), and number of cores per chip for Intel and recent AMD microcomputer chips. Note the logarithmic vertical scale.

lelism to speed up complex highly structured computations for graphics or machine learning. In fact, parallel computing is responsible for the recent breakthroughs in machine learning performance, and is in large part the future of maintaining the tremendous performance improvements that Moore's law has captured.

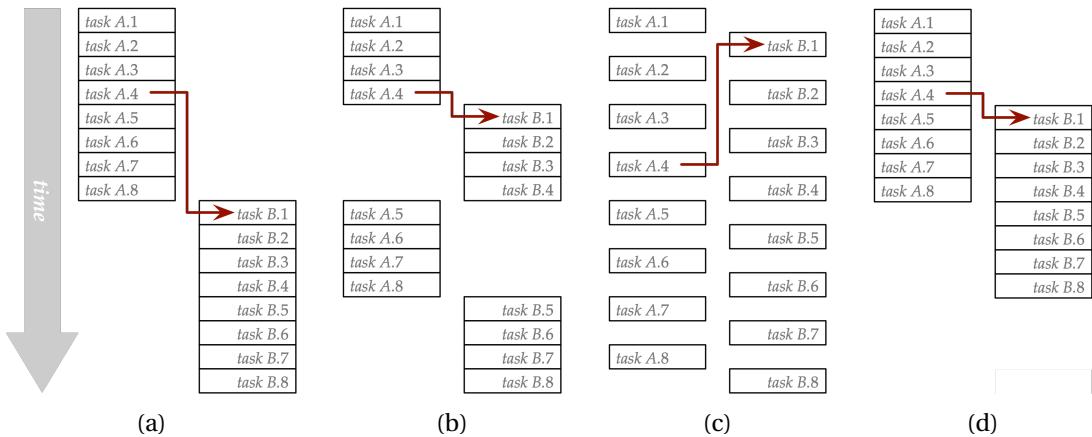
There's no free lunch. Programming computations that happen concurrently introduces new challenges, requiring new programming abstractions to manage them. In this chapter, we'll explore some of the promise, difficulty, and tools of concurrent programming. As usual, in the effort to simplify the management of the daunting problems of concurrency, new abstractions will be crucial.

### 20.1 Sequential, concurrent, and parallel computation

It will be helpful, in thinking about these issues, to imagine computation as proceeding sequentially in a series of small atomic steps. Indeed, computation *does* proceed that way, down at the level of abstraction at which the hardware processors operate. The role of a compiler is to translate programs written using higher-level abstractions down to a sequence of atomic instructions directly runnable on (possibly virtual) hardware.<sup>1</sup>

Suppose we have two tasks (A and B) to complete, each requiring

<sup>1</sup> Exactly what constitutes an atomic step depends on the particularities of the hardware; we needn't concern ourselves with the details here. We'll just assume that operations like reading a value from memory (as, for instance, accessing a variable's value), modifying a value in memory (instantiating a variable or updating a mutable variable, for instance), performing a simple operation on retrieved values (arithmetic operations, for instance), and the like are atomic. In the examples we'll use, we'll write the atomic steps on separate lines, so that any line of code will be assumed to execute atomically.



execution of a sequence of atomic steps. One way of completing the tasks is to execute the two tasks SEQUENTIALLY, all of the steps of Task A before any of the steps of Task B, as depicted in Figure 20.3(a). Alternatively, we might execute some of the steps in Task A, then some from Task B, then the remainder of Task A and the remainder of Task B (Figure 20.3(b)). The tasks are said to be running CONCURRENTLY. Even more fine-grained concurrency is possible of course (Figure 20.3(c)).

Why might such concurrency be useful? Through concurrent execution, Task B might be able to generate some useful behavior earlier than having to wait for Task A to complete. Perhaps Task A part way through its execution computes some value that is needed by Task B, or vice versa. Waiting for Task A to complete may postpone Task B for a long time. Indeed, some computations are intended never to complete. Think of the process that runs a bank ATM, which is always running a single program to handle requests from users as they walk up to and interact with it. Although the ATM process never completes, other processes may want to interact with it and intersperse their computations on the same processor, perhaps to report changes to a central database. In sum, concurrency allows multiple separate processes to interact and communicate without requiring one of them to complete before the other begins.

Where such concurrency is possible, a further benefit can accrue – carrying out the steps of the tasks IN PARALLEL (Figure 20.3(d)) by making use of separate hardware for processing the sequences of instructions. Parallelism allows both tasks to complete in fewer time steps, effectively trading time for “space” (hardware).

## 20.2 Dependencies

In taking advantage of concurrency or parallelism, delicate issues quickly arise when there are dependencies between the two sequences

Figure 20.3: Two tasks running in various forms of sequential and concurrent computation. Each task is depicted as atomic steps (the individual boxes) executing through time (running from top to bottom). (a) Task A runs sequentially to completion before task B. (b) Coarsely concurrent execution of the two tasks, with some steps of task B first running after four steps of task A. (c) Finer concurrent execution, interleaving at each atomic step. (d) Parallel computation of the tasks, with task B beginning execution after the fourth step of task A and running simultaneously. The arrows denote a dependency requiring task B.1 to run after task A.4. Note that that dependency is violated in (c).

of instructions. For instance, Task B might read a value at one of its steps (its first step, say) that Task A computes at its, say, fourth step. We've indicated such a dependency with the arrows in Figure 20.3.

If Task A and B run sequentially in that order, then of course the value generated by Task A will be available to Task B at the proper time. But concurrent computation is also possible, say if Task A completes its first four steps before Task B begins. But other interleavings can be problematic, for instance, if Tasks A and B interleave after each step. Task B will then attempt to make use of the value that Task A will calculate before it has actually been calculated. This kind of dependency, where one task must read a value only after another task writes it, is sometimes referred to as a READ-AFTER-WRITE DEPENDENCY. What happens when concurrent execution violates the read-after-write dependency may not be well defined, but it certainly is not a good situation.

In addition to read-after-write dependencies, other kinds of dependencies (WRITE-AFTER-READ, WRITE-AFTER-WRITE) are also important. The details are beyond the scope of this discussion. At this point, we're merely concerned with how to allow concurrency while avoiding violations of ordering dependencies whatever they might be.

In summary, if we just allow tasks to interleave however they happen to, with no control over which parts of which task run when, dependencies introduce a kind of race between the tasks. Will Task A's write step run faster and execute, as it should, before Task B's read? Or will Task B win the race, performing its read before task A has a chance to write? This kind of RACE CONDITION leads to the possibility of a new kind of error. Gaining the benefits of concurrency and parallelism, while avoiding race conditions and other new classes of errors, is the challenge of concurrent and parallel programming.

### 20.3 Threads

In order to demonstrate these issues and experiment with abstractions that can help avoid these new classes of errors, we need a way to implement concurrency. OCaml provides a programming abstraction that allows us to experiment with concurrency, the THREAD. A thread can be thought of as providing a separate virtual processor.<sup>2</sup>

Suppose we need to do two tasks – call them Task A and B as before – implemented as OCaml functions named accordingly. Perhaps we want to sum the results returned by these two tasks. We can easily execute them *sequentially*, task A before B:

```
let resultA = taskA () in
let resultB = taskB () in
```

<sup>2</sup> OCaml threads provide concurrency, not true parallelism, but the issues they raise apply equally well to parallel processing, so they're all we'll need to demonstrate the problems. Other OCaml modules, and aspects of many other programming languages, provide concurrency and parallelism constructs that introduce just the same issues.

---

```

(* log thread_name msg -- Prints a message recording that a
   thread with the given `thread_name` has generated a log
   message `msg`. Also prints an indication of time in seconds
   since an initialization time. Used for tracking concurrent
   executions. *)
let log =
  (* store a fixed marker time for comparison *)
  let init_time = Unix.gettimeofday () in
  fun thread_name msg ->
    Printf.printf "[%3.4f %s: %s]\n%"!
      ((Unix.gettimeofday ()) -. init_time)
      thread_name
      msg ;;

(* task_delayed name delay value -- Prints a message
   recording that a thread with the given `thread_name` has
   generated a log message `msg`. Also prints an indication of
   time in seconds since an initialization time. Used for
   tracking concurrent executions. *)
let task_delayed (name : string)
  (delay : float)
  (value : 'a)
  : 'a =
  log name "starts";
  Thread.delay delay;
  log name "ends";
  value ;;

(* Two sample tasks taking differing lengths of time and
   returning different values. *)
let task_short () = task_delayed "task_short" 0.1 1;;
let task_long () = task_delayed "task_long" 0.2 2;;

```

---

Figure 20.4: For reference, some logistical code used in the concurrency demonstrations.

```
resultA + resultB ;;
```

We can think of the two tasks (along with the computation of their sum) as being executed in a single thread of computation. The semantics of the `let` construct ensures that `taskA ()` will be evaluated, generating `resultA`, before `taskB ()` begins its evaluation.

In order to demonstrate the idea, and prepare for the significantly more subtle examples to come, we define a test function that takes two functions as its argument, which play the roles of tasks A and B.

```
# let test_sequential taskA taskB =
#   let resultA = taskA () in
#   let resultB = taskB () in
#   resultA + resultB ;;
val test_sequential : (unit -> int) -> (unit -> int) -> int = <fun>
```

We can test this sequential computation using some simulated tasks. The unit function `task_short` simulates a task that engages in a shorter computation (0.1 seconds) returning the value 1. The corresponding function `task_long` takes longer (0.2 seconds) and returns the value 2. (The details of how they're implemented aren't important, but for completeness, they're provided in Figure 20.4.) Let's test it out.

```
# test_sequential task_short task_long ;;
[1.1147 task_short: starts]
[1.2149 task_short: ends]
[1.2149 task_long : starts]
[1.4150 task_long : ends]
- : int = 3
```

The test returns the summed results 3. Along the way, various key events are logged. We see the start of the short task and its ending, followed by the start and end of the long task, indicating their sequentiality. The logged start and end times indicate that the short and long tasks required about 0.1 and 0.2 seconds, respectively, together requiring 0.3 seconds, as expected.

If we'd like the two tasks to execute *concurrently*, we can establish a separate thread (that is, a separate virtual processor) corresponding to `taskA`. We refer to this process as FORKING a new thread. OCaml provides for creating and manipulating threads in its `Thread` module.<sup>3</sup> To fork a new thread, we use the `Thread.create` function, which takes a function and its argument and returns a *separate new thread of computation* (a value of type `Thread.t`) in which the function is applied to its argument. Its type is thus `('a -> 'b) -> 'a -> Thread.t`. So we can evaluate tasks A and B in separate threads, concurrently, as follows:

```
let threadA = Thread.create taskA () in
let resultB = taskB () in
...;
```

<sup>3</sup> The `Thread` module is part of OCaml's threads library, which allows for creating multiple concurrent threads. To make use of the library in the REPL, you'll need to make it available with

```
#use "topfind" ;;
#thread;;
```

The evaluation of the `Thread.create` expression returns immediately, without waiting for the result of the application of `taskA` to () to finish in the new thread. Thus when `taskB ()` is evaluated, it doesn't wait until `taskA` completes.<sup>4</sup>

#### 20.4 Interthread communication

We've enabled two tasks to operate concurrently using threads. But we have no way as of yet for threads to communicate with each other. For instance, in the example above, how can `taskA`, isolated in its own thread, inform the thread running `taskB` about its return value? Similarly, how can `taskB` communicate information to `taskA` if it needs to?

A simple mechanism for this interthread communication is for the threads to share mutable values, which serve as channels of communication between the threads. Let's start with how the created thread executing `taskA` can communicate its return value to the main thread that needs to calculate the sum.

We'll define another test function called `test_communication` to test the communication between two tasks executed in separate threads as above.

```
let test_communication taskA taskB =
  ...
```

We'll use a shared mutable value called `shared_result` of type `int option`, initially `None` since no result is yet available.

```
...
let shared_result = ref None in
...
```

Now we can create a new thread for executing task A, saving its return value in the shared result.

```
...
let _thread =
  Thread.create
    (fun () -> shared_result := Some (taskA ())) () in
...
```

In the original thread, we perform task B, saving its result.

```
...
let resultB = taskB () in
...
```

Finally, we can extract the result from task A from the shared value, and compute with the two results, by adding them as before.

<sup>4</sup> The `Thread` library allows for concurrent execution of the various threads forked in the process, not parallel execution. An exception is that the `Thread.delay` function, which we use to simulate computations that take a long time, allows other threads to continue to run during the delay period.

```

...
match !shared_result with
| Some resultA ->
  (* compute with the two results *)
  resultA + resultB
| None ->
  (* Oops, taskA hasn't completed! *)
  failwith "shouldn't happen!" ;;

```

Putting it all together, we have

```

# let test_communication taskA taskB =
#   let shared_result = ref None in
#   let _thread =
#     Thread.create
#       (fun () -> shared_result := Some (taskA ())) () in
#   let resultB = taskB () in
#   match !shared_result with
#   | Some resultA ->
#     (* compute with the two results *)
#     resultA + resultB
#   | None ->
#     (* Oops, taskA hasn't completed! *)
#     failwith "shouldn't happen!" ;;
val test_communication : (unit -> int) -> (unit -> int) -> int =
<fun>

```

Again, we can test using the simulated tasks. To start, we fork the new thread running the shorter task, with the longer task in the main thread.

```

# test_communication task_short task_long ;;
[2.1245 task_long : starts]
[2.1245 task_short: starts]
[2.2247 task_short: ends]
[2.3246 task_long : ends]
- : int = 3

```

the communication works as expected. The short task returns 1 – passed through and retrievable from the shared variable – and the long task returns 2. The test as a whole computes their sum, 3 as expected.

The logged events show the starting of the long task in the main thread, followed immediately by the short task starting in the newly created thread. The latter short thread completes quickly (it's shorter, after all), ending before the long task does. The main thread can extract the completed value for the short task and add it to the result from the long task.<sup>5</sup>

But what if the task in the forked thread takes longer than that in the main thread? We can simulate this by swapping the long and short tasks in the test function.

```

# test_communication task_long task_short ;;
[2.3762 task_short: starts]

```

<sup>5</sup> As before, the logged times indicate that the short and long tasks required about 0.1 and 0.2 seconds. This time, however, the overall computation requires only 0.2 seconds, since the `delay` function allows for some parallelism between the two threads (as noted in footnote 4). The simulation thus gives a hint of the potential for parallel processing to speed computation.

```
[2.3762 task_long : starts]
[2.4763 task_short: ends]
Exception: Failure "shouldn't happen!".
```

In this version of the test, the short task in the main thread completes before the forked thread has time to complete the long task and update the shared variable, leading to a run-time exception. The code has a race condition with respect to a read-after-write dependency. These two executions of the test demonstrate that, depending on which task “wins the race”, the value to be read may or may not be written in time as it needs to be.

In general, one doesn’t have the kind of detailed information about run times of various tasks as we have for `task_short` and `task_long`. This kind of concurrent computation, without careful controls, thus leads to indeterminacy at runtime. And debugging these intermittent bugs that can come and go, perhaps appearing only rarely, can be especially confounding. More tools are needed.

The lesson here is that the main thread shouldn’t attempt to use the shared variable until the forked thread has completed. We thus need a way of guaranteeing that a thread has completed. One solution you may have thought of is to have the main thread test if the shared value has not been properly set, and if not, to just “try again later”. We can implement this with a simple loop,

```
while !shared_result == None do
  Thread.delay 0.01
done;
```

which continually waits for a fraction of a second so long as the shared result has not been properly set, a technique called **BUSY WAITING**.

```
# let test_communication taskA taskB =
#   let shared_result = ref None in
#   let _thread =
#     Thread.create
#       (fun () -> shared_result := Some (taskA ())) () in
#   let resultB = taskB () in
#   while !shared_result == None do
#     Thread.delay 0.01
#   done;
#   match !shared_result with
#   | Some resultA ->
#     (* compute with the two results *)
#     resultA + resultB
#   | None ->
#     (* Oops, taskA hasn't completed! *)
#     failwith "shouldn't happen!" ;;
val test_communication : (unit -> int) -> (unit -> int) -> int =
<fun>
```

The errant race condition is now handled properly.

```
# test_communication task_long task_short;;
[3.5867 task_short: starts]
[3.5867 task_long : starts]
[3.6868 task_short: ends]
[3.7868 task_long : ends]
- : int = 3
```

But this kind of brute force trick of repeatedly polling the shared variable until it is ready is profligate and inelegant. It can waste computation that would be better allocated to other threads, and can waste time if the delay is longer than needed.

Instead, we'd like to be able to directly specify to wait *until the forked thread completes*. The companion to the `fork` function `Thread.create` is the `join` function `Thread.join`. `Thread.join` takes a thread as its argument and returns *only once that thread has completed*. By requiring the join before accessing the shared variable, we are guaranteed that the variable will have been updated at the time that we need it.

```
# let test_communication taskA taskB =
#   let shared_result = ref None in
#   let thread =
#     Thread.create
#       (fun () -> shared_result := Some (taskA ())) () in
#   let resultB = taskB () in
#   Thread.join thread;
#   match !shared_result with
#   | Some resultA ->
#     (* compute with the two results *)
#     resultA + resultB
#   | None ->
#     (* Oops, taskA hasn't completed! *)
#     failwith "shouldn't happen!" ;;
val test_communication : (unit -> int) -> (unit -> int) -> int =
<fun>
```

Using this version of the test, the race condition is avoided, and the calculation completes properly.

```
# test_communication task_long task_short;;
[4.5455 task_short: starts]
[4.5455 task_long : starts]
[4.6456 task_short: ends]
[4.7457 task_long : ends]
- : int = 3
```

## 20.5 Futures

The structure of this small example, in which a thread is forked to allow it to compute a return value that is needed in the future, is so common that it deserves its own abstraction, a kind of value dubbed

a FUTURE. This abstraction is implemented via two functions: The `future` function takes a task to be carried out for its return value, and returns a “future value”. We can later use the `force` function to force the future value to be extracted when available. A module signature can help clarify the needed functionality:

```
# module type FUTURE =
#   sig
#     type 'result future
#
#     (* future fn x -- Forks a new thread within which `fn`
#        is applied to `x`. Immediately returns a `future`
#        which can be used to synchronize with the thread
#        and extract the result. *)
#     val future : ('arg -> 'result) -> 'arg -> 'result future
#
#     (* force fut -- Causes the calling thread to wait until the
#        thread computing the future value `fut` is done and then
#        returns its value. *)
#     val force : 'result future -> 'result
#   end ;;
module type FUTURE =
sig
  type 'result future
  val future : ('arg -> 'result) -> 'arg -> 'result future
  val force : 'result future -> 'result
end
```

There are multiple ways to implement this functionality, but we'll use the shared value method from the previous example. In this implementation, a future value (an element of the `future` type) is a record that contains the thread identifier in which the future task is being carried out and the mutable variable for communicating the result back to the calling thread.

```
# module Future : FUTURE =
#   struct
#     type 'result future = {tid : Thread.t;
#                           value : 'result option ref}
#
#     let future (f : 'arg -> 'result) (x : 'arg) : 'result future =
#       let r : 'result option ref = ref None in
#       let t = Thread.create (fun () -> r := Some (f x)) () in
#       {tid = t; value = r}
#
#     let force (f : 'result future) : 'result =
#       Thread.join f.tid;
#       match !(f.value) with
#         | Some v -> v
#         | None -> failwith "impossible!"
#   end ;;
module Future : FUTURE
```

With the future abstraction in hand, the `test_communication` example above can be greatly simplified.

```
# let test_future taskA taskB =
#   let futureA = Future.future taskA () in
#   let resultB = taskB () in
#   Future.force futureA + resultB ;;
val test_future : (unit -> int) -> (unit -> int) -> int = <fun>

# test_future task_long task_short ;;
[6.3619 task_short: starts]
[6.3619 task_long : starts]
[6.4620 task_short: ends]
[6.5620 task_long : ends]
- : int = 3
```

This is hardly more complicated than the sequential version (`test_sequential`) that we started with above, requiring only the simple addition of the highlighted `future` call.

#### **Exercise 196**

Exercise 96 concerned implementing a fold operation over binary trees defined by

```
# type 'a bintree =
# | Empty
# | Node of 'a * 'a bintree * 'a bintree ;;
type 'a bintree = Empty | Node of 'a * 'a bintree * 'a bintree
```

Define a version of the fold operation, `foldbt_conc`, that performs the recursive folds of the left and right subtrees concurrently, making use of futures to ensure that results are available when needed.

## *20.6 Futures are not enough*

The sharing of mutable data across two concurrent threads is a valuable ability. It implements a kind of communication channel between the threads. But managing this communication is complex. We've already seen this in the context of a thread's "return value". The calling thread mustn't read the shared variable that will be storing the called thread's return value until the latter has completed its computation and updated the return value. Managing this ordering is the whole point of the `future/force` abstraction.

Sharing mutable data across threads is a useful technique well beyond just allowing for return values to be communicated.

1. Threads may have need for coordinating data manipulation beyond the mere passing of a return value. For instance, think of multiple threads manipulating a shared database.
2. In the case of threads that are not intended to terminate, the whole notion of a return value is inapplicable. Importantly, not all concurrent computations are intended to terminate. Indeed, one of the

benefits of concurrency as a programming concept is that it allows multiple threads of nonterminating computation to interact. We still need to manage the interaction so that the concurrent computations satisfy the various dependencies among them without dangerous race conditions.

A standard example of this kind of concurrent nonterminating computation is the ATM. ATMs are computers that run a program that interacts with bank patrons to allow them to manipulate their bank accounts in various ways. The bank accounts constitute a shared database of mutable data. And because banks have multiple geographically distributed ATMs, multiple instances of the program are running concurrently, and potentially transforming the same shared data, the balances of the various accounts.

To demonstrate the problem, let's think of a bank as having multiple accounts each of which is an instance of an `account` class defined as follows:

```
# class account (initial_balance : int) =
#   object
#     val mutable balance = initial_balance
#
#     method balance = balance
#
#     method deposit (amt : int) : unit =
#       balance <- balance + amt
#
#     method withdraw (amt : int) : int =
#       if balance >= amt then begin
#         balance <- balance - amt;
#         amt
#       end else 0
#   end ;
class account :
  int ->
object
  val mutable balance : int
  method balance : int
  method deposit : int -> unit
  method withdraw : int -> int
end
```

The `deposit` and `withdraw` methods both potentially affect the value of the mutable `balance` variable. The `withdraw` function, in particular, verifies that the balance is sufficient to cover the withdrawal amount, updates the balance accordingly, and returns the amount to be dispensed (0 if the balance is insufficient).

Now what happens when we try multiple concurrent withdrawals from the same account? To simulate such an occurrence, the following `test_wds` function carries out withdrawals of \$75 and \$50 in separate

threads (call them “thread A” and “thread B” for ease of reference) from a single account with initial balance of \$100, using a `future` for the larger withdrawal. To track what goes on, the test function returns the amount dispensed in thread A and thread B, along with the final balance in the account.

```
# let test_wds () =
#   let acct = new account 100 in
#   let threadA_ftr = Future.future acct#withdraw 75 in
#   let threadB = acct#withdraw 50 in
#   let threadA = (Future.force threadA_ftr) in
#   threadA, threadB, acct#balance ;;
val test_wds : unit -> int * int * int = <fun>
```

What behavior would we like to see in this case? One or the other of the two withdrawals, whichever comes first, should see a sufficient balance, dispense the requested amount, and update the balance accordingly. The other attempted withdrawal should see a reduced and insufficient balance and dispense no funds. In particular, if task A completes first, the two accounts should see withdrawals of \$75 and \$0 respectively, leaving a balance of \$25, that is, the simulation function should return the triple  $(75, 0, 25)$ . If task B completes first, the two accounts should see withdrawals of \$0 and \$50 respectively, leaving a balance of \$50, that is, the simulation function should return the triple  $(0, 50, 50)$ . Let’s try it.

```
# test_wds () ;;
- : int * int * int = (0, 50, 50)
```

In order to experiment with the possibility of interleavings of the various components of the withdrawals, we make two changes to the withdrawal simulation. First, we divide the balance update

```
balance <- balance - amt
```

into two parts: the computation of the updated balance and the update of the `balance` variable itself:

```
let updated = balance - amt in
balance <- updated
```

Doing so separates the *reading* of the shared balance from its *writing*, allowing interposition of other threads in between.

Second, we introduce some random delays at various points in the computation: before the withdrawal first executes, immediately after the balance check, and after computing the updated balance just before carrying out the update. For this purpose, we use a function `random_delay`, which pauses a thread for a randomly selected time interval.

```
# let random_delay (max_delay : float) : unit =
#   Thread.delay (Random.float max_delay) ;;
val random_delay : float -> unit = <fun>
```

Updating the withdrawal function to insert these delays, we have

```
method withdraw (amt : int) : int =
  random_delay 0.004;
  if balance >= amt then begin
    random_delay 0.001;
    let updated = balance - amt in
    random_delay 0.001;
    balance <- updated;
    amt
  end else 0;;
```

Here is a typical outcome from this simulation.

```
# test_wds () ;;
- : int * int * int = (75, 50, -25)
```

If we run the simulation many times, we see (Figure 20.5) that the result is quite variable. Certainly, there are many occurrences (about half) showing the desired behavior, with either \$75 or \$50 dispensed and a final balance of \$25 or \$50, respectively. But we also see plenty of instances where both withdrawals go through, dispensing both \$75 and \$50, leaving a final balance of \$-25. Or \$25. Or \$50. The use of `future` ensures that the return value dependency is properly obeyed, but the various dependencies having to do with the updates to and uses of the account's balance are uncontrolled. Different interleavings of these operations can yield different results. Let's examine a few of the many possible interleavings.

First, thread A (the \$75 withdrawal) may execute fully before thread B (the \$50 withdrawal) begins. That is, they may execute sequentially. This interleaving is depicted in Figure 20.6. In this representation of the two threads executing, the executed lines of thread A are on the left, thread B on the right. We assume that each line of code executes atomically, with the order of the numbered lines indicating the order in which they are executed in the concurrent computation. The ellipses (...) indicate code lines that were not executed since they fell in the non-chosen branch of a conditional. In line 1, the balance test in

| valid? | first | second | balance | count |
|--------|-------|--------|---------|-------|
|        | 75    | 50     | -25     | 29    |
|        | 75    | 50     | 50      | 27    |
|        | 75    | 50     | 25      | 26    |
| ✓      | 75    | 0      | 25      | 11    |
| ✓      | 0     | 50     | 50      | 7     |

Figure 20.5: Table of outcomes from multiple runs of simultaneous withdrawals. Each row represents a possible outcome, with columns showing the amount dispensed for the first withdrawal, the amount dispensed for the second withdrawal, the final balance, and the number of times this outcome occurred in 100 trials. Only the checkmarked trials are valid in respecting dependencies.

| thread A (\$75 withdrawal)          | thread B (\$50 withdrawal)   |
|-------------------------------------|------------------------------|
| 1. if balance >= amt then begin     |                              |
| 2.   let updated = balance - amt in |                              |
| 3.   balance <- updated;            |                              |
| 4.   amt                            |                              |
| 5. ...                              | if balance >= amt then begin |
|                                     | ...                          |
| 6.                                  | end else 0                   |

Figure 20.6: An unproblematic (essentially sequential) interleaving of the threads.

| <i>thread A (\$75 withdrawal)</i> | <i>thread B (\$50 withdrawal)</i> |
|-----------------------------------|-----------------------------------|
| 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. amt                            |                                   |
| 7. ...                            | balance <- updated;               |
| 8.                                | amt                               |
|                                   | ...                               |

thread A is evaluated. Since the balance is initially 100, and the withdrawal amount is 75, the condition holds and lines 2-4 in the then branch are executed. Line 3 in particular updates the shared balance to 25, so that in line 5 when thread B tests the balance, the test fails and the second withdrawal does not complete (line 6). In summary, the \$75 withdrawal attempt succeeds, dispensing the \$75, and the \$50 withdrawal attempt fails, leaving a balance of \$25.

Of course, if thread B had executed fully before thread A, the corresponding result would have occurred, dispensing only the \$50 and leaving a balance of \$50.

But other results are also possible. For instance, consider the interleaving in Figure 20.7. Each thread verifies the balance as being adequate and computes its updated value before the other performs the balance update. Both threads go on to update the balance (lines 5 and 7); since thread B updates the balance later, its balance value, \$50, overwrites thread A's \$25 balance, so the final balance is \$50. In summary, both attempted withdrawals succeed, dispensing both \$75 and \$50, leaving a surprising \$50 balance. Sure enough, Figure 20.5 indicates that such outcomes were actually attested in the simulations.

#### Exercise 197

Construct an interleaving in which both withdrawals succeed, leaving a balance of \$25.

#### Exercise 198

Construct an interleaving in which both withdrawals succeed, leaving a balance of \$ – 25.

As Figure 20.5 shows, and these possible interleavings explain, there are important dependencies that are not being respected in the concurrent implementation of the account operations. A solution to this problem of controlling data dependencies requires further tools.

## 20.7 Locks

To gain better control over the interleavings, we introduce a new abstraction, the LOCK. The underlying idea is that while a thread is executing the withdrawal method, it ought to be the only thread with access to the balance it is manipulating. Just as you might lock your door to prevent others from using your car, you might want to lock

Figure 20.7: A problematic interleaving of the threads.

some data to prevent others from manipulating it. OCaml provides a simple interface to a locking mechanism called `MUTEX LOCKS` in its `Mutex` library. The name comes from the idea of *mutual exclusion*; other threads should be excluded from certain regions of code when a lock is in force.

To create a mutex lock for a given datum – the mutable balance, say, in the ATM example – we use `Mutex.create`.

```
# let balance_lock = Mutex.create () ;;
val balance_lock : Mutex.t = <abstr>
```

As shown, this creates a lock of type `Mutex.t`. We can then lock and unlock the lock as needed with the functions `Mutex.lock` and `Mutex.unlock`.

The mutex locks work as follows. When `Mutex.lock` is called on a lock, the lock is first verified to be in its unlocked state. If so, the lock switches to the locked state and computation proceeds.<sup>6</sup> But if not, the thread in which the call was made is suspended until such time as the lock becomes unlocked, presumably by a call to `Mutex.unlock` in another thread.

Inserting the locks in the ATM example, we would have a withdraw method like this:

```
method withdraw (amt : int) : int =
  Mutex.lock balance_lock;
  if balance >= amt then begin
    balance <- balance - amt;
    amt
  end else 0;
  Mutex.unlock balance_lock ;;
```

The code between the locking and unlocking is the CRITICAL REGION, a computation that must be carried out atomically from the point of view of the resource that is being locked. In this case, the entire body of the `withdraw` method is a critical region.

Now consider the previous problematic case of Figure 20.7 – in which thread B's withdrawal code begins executing partway through thread A's withdrawal code – except now with the locking implementation above. As seen in Figure 20.8, thread A now begins by establishing the balance lock in step 1. When the first step of thread B executes at the intermediate point within thread A's execution (after step 3) and attempts to itself acquire the balance lock, the lock causes thread B to suspend until such time as the lock becomes available, which is not until thread A releases the lock at step 6. The delay in thread B changes the interleaving to a safe one, like that of Figure 20.6.

<sup>6</sup> Crucially, the testing for unlocked status and the subsequent locking occur atomically, so that other threads can't interleave between them. How this is accomplished, the subject of fundamental research in concurrent computation, is well beyond the scope of this text.

| <i>thread A (\$75 withdrawal)</i>      | <i>thread B (\$50 withdrawal)</i> |
|----------------------------------------|-----------------------------------|
| 1.    Mutex.lock balance_lock;         |                                   |
| 2.    if balance >= amt then begin     |                                   |
| 3.      let updated = balance - amt in |                                   |
| 4.      balance <- updated;            | Mutex.lock balance_lock;          |
| 5.      amt                            |                                   |
| 6.      ...                            |                                   |
| 7.      Mutex.unlock balance_lock;     |                                   |
| 8.                                     | <i>thread B suspended</i>         |
| 9.                                     |                                   |
| 10.                                    |                                   |
| 11.                                    |                                   |
| 12.                                    | Mutex.unlock balance_lock;        |

Figure 20.8: The problematic interleaving, corrected by the use of locks.

### 20.7.1 Abstracting lock usage

This idiom – wrapping a critical region with a lock at the beginning and an unlock at the end – captures the stereotypical use of locks.

In this idiom, the lock is explicitly unlocked after the need for the lock is over. The unlocking is crucial; without it, other threads would be *permanently* prevented from carrying out their own computations requiring the lock. We can codify the importance of matching the locks and unlocks by way of an abstracted function that wraps a computation with the lock and its corresponding unlock. We call the function `with_lock`:

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

If we stick with using `with_lock`, we never need to worry that we will perform a lock without the matching unlock, in keeping with the edict of prevention.

Or will we? What would happen if the computation of `f ()` raised an exception of some sort? The body of the `let` will never be performed, and the lock will not be unlocked! (Of course, that possibility also held for the `withdraw` method just above.) We'll want to fix that by adjusting `with_lock` to make sure to handle exceptions properly, further manifesting the edict of prevention. We leave that for an exercise.

#### Exercise 199

Define a version of `with_lock` that handles exceptions by making sure to unlock the lock.

Using `with_lock`, the `withdraw` method becomes

```
method withdraw (amt : int) : int =
```

```

with_lock balance_lock (fun () ->
  if balance >= amt then begin
    balance <- balance - amt;
    amt
  end else 0) ;;

```

With this modified implementation of accounts, the simulation of many trials of simultaneous deposits performs much better, with only valid results, as depicted in Figure 20.9.

### 20.8 Deadlock

| <i>valid?</i> | <i>first</i> | <i>second</i> | <i>balance</i> | <i>count</i> |
|---------------|--------------|---------------|----------------|--------------|
| ✓             | 0            | 50            | 50             | 51           |
| ✓             | 75           | 0             | 25             | 49           |

Figure 20.9: Rerunning the test of simultaneous withdrawals, with locking in place, all trials now respect the dependencies, though the results can still vary depending on which of the two withdrawals in each trial happens to occur first.



# A

## *Final project: Implementing MiniML*

The culminating final project for CS51 is the implementation of a small subset of an OCaml-like language. Unlike the problem sets, the final project is more open-ended, and we expect you to work more independently, using the skills of design, abstraction, testing, and debugging that you've learned during the course.

### A.1 *Overview*

Unlike OCaml and the ML programming language it was derived from, the language you will be implementing includes only a subset of constructs, has only limited support for types (including no user-defined types), and does no type inference (enforcing type constraints only at run-time). On the other hand, the language is “Turing-complete”, as expressive as any other programming language in the sense specified by the Church-Turing thesis. Because the language is so small, we refer to it as MiniML (pronounced “minimal”).

The implementation of this OCaml subset MiniML is in the form of an interpreter for expressions of the language written in OCaml itself, a METACIRCULAR INTERPRETER. Actually, you will implement a series of interpreters that vary in the semantics they manifest. The first is based on the substitution model (Chapter 13); the second a dynamically scoped environment model (Chapter 19); and the third, a version of the second implementing one or more extensions of your choosing, with lexical scoping being a simple and highly recommended option.

This chapter builds on the idea of specifying the semantics of a programming language and implementing that specification begun in Chapters 13 and 19. The exercises herein are to test your understanding. We recommend that you do the exercises, but you won't be turning them in and we won't be supplying answers. The STAGES provide a sequence of nine stages to implement the MiniML interpreter. It's the result of working on these stages that you will be turning in and on

which the project grade will be based.

This project specification is divided into three sections (corresponding to the section numbers marked below):

*Substitution model (Section A.2)* Implementation of a MiniML interpreter using the substitution semantics for the language.

*Dynamic scoped environment model (Section A.3)* Implementation of a MiniML interpreter using the environment model and manifesting dynamic scoping.

*Extensions (Section A.4)* Implementation of one or more extensions to the basic MiniML language of your choosing. Special attention is paid below to an extension to the environment model manifesting lexical scoping (Section A.4.2).

#### A.1.1 *Grading and collaboration*

As with all the individual problem sets in the course, your project is to be done individually, under the course's standard rules of collaboration. (The sole exception is described in Section A.6.) You should not share code with others, nor should you post public questions about your code on Piazza. If you have clarificatory questions about the project assignment, you can post those on Piazza and if appropriate we will answer them publicly so the full class can benefit from the clarification.

The final project will be graded based on correctness of the implementation of the first two stages; design and style of the submitted code; and scope of the project as a whole (including the extensions) as demonstrated by a short paper describing your extensions, which is assessed for both content and presentation.

It may be that you are unable to complete all the code stages of the final project. You should make sure to keep versions at appropriate milestones so that you can always roll back to a working partial project to submit. Using git will be especially important for this version tracking if used properly.

Some students or groups might prefer to do a different final project on a topic of their own devising. For students who have been doing exceptionally well in the course to date, this may be possible. See Section A.6 for further information.

#### A.1.2 *A digression: How is this project different from a problem set?*

We frequently get questions about the final project of the following sort: Do I need to implement X? Am I supposed to handle Y? Is it a

sufficient extension to do Z? Should I provide tests for W? Is U the right way to handle V? Do I have to discuss P in the writeup?

The final project description doesn't specify answers to many questions of this sort. This is not an oversight; it is a pedagogical choice. In the world of software design and development, there are an infinite number of choices to make, and there are often no right answers, merely tradeoffs. Part of the point of the course is that there are many ways to implement software for a particular purpose, and they are not all equally good. (See Section 1.2.) The final project is the place in the course where you are most clearly on your own to deploy the ideas from the course to make these choices and demonstrate your best understanding of the tradeoffs involved. By implementing X, you may not have time to test Y. By implementing only Z, you may be able to do so with a more elegant or generalizable approach. By adding tests for W, you may not have time to fully discuss P in the writeup. So it goes.

Perhaps the most important of the major tradeoffs is that between spending time to make improvements to the CS51 final project software and writeup and spending time on other non-CS51 efforts. Because choices made in negotiating this tradeoff don't fall solely within the environment of CS51, it is inherently impossible for course staff to give you advice on what to do. You'll have to decide whether your time is better spent, say, systematizing your unit tests for the project, or working on the final paper in your Gen Ed course; further augmenting your implementation of int arithmetic to handle bignums, or studying for the math midterm that the instructor fatuously scheduled during reading period; generating further demonstrations of the mutable array extension you added by implementing a suite of in-place sorting algorithms, or wrangling members of the student organization you find yourself running because the president is awol.

With the final project, you are on your own. Not for issues of clarification of this project description, where the course staff stand ready to help on Piazza and in office hours. But on deontic issues, issues of what's better or worse, what you "should" do or mustn't, what is required or forbidden. This is a kind of freedom, and like all freedoms, it is not without consequences, but they are consequences you must inevitably reconcile on your own.

## A.2 *Implementing a substitution semantics for MiniML*

You'll start your implementation with a substitution semantics for MiniML. The abstract syntax of the language is given by the following type definition:

```

type unop =
| Negate ;;

type binop =
| Plus
| Minus
| Times
| Equals
| LessThan ;;

type varid = string ;;

type expr =
| Var of varid          (* variables *)
| Num of int            (* integers *)
| Bool of bool           (* booleans *)
| Unop of unop * expr   (* unary operators *)
| Binop of binop * expr * expr (* binary operators *)
| Conditional of expr * expr * expr (* if then else *)
| Fun of varid * expr    (* function def'ns *)
| Let of varid * expr * expr (* local naming *)
| Letrec of varid * expr * expr (* rec. local naming *)
| Raise                  (* exceptions *)
| Unassigned             (* (temp) unassigned *)
| App of expr * expr ;;  (* function app'ns *)

```

These type definitions can be found in the partially implemented `Expr` module in the files `expr.ml` and `expr.mli`. You'll notice that the module signature requires additional functionality that hasn't been implemented, including functions to find the free variables in an expression, to generate a fresh variable name, and to substitute expressions for free variables, as well as to generate various string representations of expressions.

#### Exercise 200

Write a function `exp_to_concrete_string : expr -> string` that converts an abstract syntax tree of type `expr` to a concrete syntax string. The particularities of what concrete syntax you use is not crucial so long as you do something sensible along the lines we've exemplified. (This function will actually be quite helpful in later stages.)

To get things started, we also provide a parser for the MiniML language, which takes a string in a concrete syntax and returns a value of this type `expr`; you may want to extend the parser in a later part of the project (Section A.4.3).<sup>1</sup> The compiled parser and a read-eval-print loop for the language are available in the following files:

*evaluation.ml* The future home of anything needed to evaluate expressions to values. Currently, it provides a trivial “evaluator” `eval_t` that merely returns the expression unchanged.

*miniml.ml* Runs a read-eval-print loop for MiniML, using the `Evaluation` module that you will complete.

<sup>1</sup> The parser that we provide makes use of the OCaml package `menhir`, which is a parser generator for OCaml. You should have installed it as per the setup instructions provided at the start of the course by running the following `opam` command:

```
% opam install -y menhir
```

The `menhir` parser generator will be discussed further in Section A.4.3.

*miniml\_lex.mll* A lexical analyzer for MiniML. (You should never need to look at this unless you want to extend the parser.)

*miniml\_parse.mly* A parser for MiniML. (Ditto.)

What's left to implement is the `Evaluation` module in `evaluation.ml`.

Start by familiarizing yourself with the code. You should be able to compile `miniml.ml` and get the following behavior.<sup>2</sup>

```
# ocamlbuild -use-ocamlfind miniml.byte
Finished, 13 targets (12 cached) in 00:00:00.
# ./miniml.byte
Entering miniml.byte...
<== 3 ;;
Fatal error: exception Failure("exp_to_abstract_string
not implemented")
```

### Stage 201

Implement the function `exp_to_abstract_string : expr -> string` to convert abstract syntax trees to strings representing their structure and test it thoroughly. If you did Exercise 200, the experience may be helpful here, and you'll want to also implement `exp_to_concrete_string : expr -> string` for use in later stages as well. The particularities of what concrete syntax you use to depict the abstract syntax is not crucial – we won't be checking it – so long as you do something sensible along the lines we've exemplified.

After this (and each) stage, it would be a good idea to commit the changes and push to your remote repository as a checkpoint and backup.

Once you write the function `exp_to_abstract_string`, you should have a functioning read-eval-print loop, except that the evaluation part doesn't do anything. (The REPL calls the trivial evaluator `eval_t`, which essentially just returns the expression unchanged.) Consequently, it just prints out the abstract syntax tree of the input concrete syntax:

```
# ./miniml.byte
Entering miniml.byte...
<== 3 ;;
==> Num(3)
<== 3 4 ;;
==> App(Num(3), Num(4))
<== (((3)) ;;
xx> parse error
```

<sup>2</sup> In building the project, you may find that you get a warning of the form:

```
+ menhir -ocamlc 'ocamlfind ocamlc
-thread -strict-sequence
-package graphics
-package CS51Utils -w
A-4-33-40-41-42-43-34-44'
-infer miniml_parse.mly
Warning: 15 states have
shift/reduce conflicts.
Warning: one state has
reduce/reduce conflicts.
Warning: 198 shift/reduce
conflicts were arbitrarily
resolved.
Warning: 18 reduce/reduce
conflicts were arbitrarily
resolved.
```

You can safely ignore this message from the parser generator, which is reporting on some ambiguities in the MiniML grammar that it has resolved automatically.

```

<== let f = fun x -> x in f f 3 ;;
=> Let(f, Fun(x, Var(x)), App(App(Var(f), Var(f)), Num(3)))
<== let rec f = fun x -> if x = 0 then 1 else x * f (x - 1) in f 4 ;;
=> Letrec(f, Fun(x, Conditional(Binop(Equals, Var(x), Num(0)), Num(1),
Binop(Times, Var(x), App(Var(f), Binop(Minus, Var(x), Num(1))))),
App(Var(f), Num(4))))
<== Goodbye.

```

**Exercise 202**

Familiarize yourself with how this “almost” REPL works. How does `eval_t` get called? What does `eval_t` do and why? What’s the point of the `Env.Val` in the definition? Why does `eval_t` take an argument `_env : Env.env`, which it just ignores? (These last two questions are answered a few paragraphs below. Feel free to read ahead.)

To actually get evaluation going, you’ll need to implement a substitution semantics, which requires completing the functions in the `Expr` module.

**Stage 203**

Start by writing the function `free_vars` in `expr.ml`, which takes an expression (`expr`) and returns a representation of the free variables in the expression, according to the definition in Figure 13.3. Test this function completely.

**Stage 204**

Next, write the function `subst` that implements substitution as defined in Figure 13.4. In some cases, you’ll need the ability to define new fresh variables in the process of performing substitutions. You’ll see we call for a function `new_varname` to play that role. Looking at the `gensym` function that you wrote in lab might be useful for that. Once you’ve written `subst` make sure to test it completely.

You’re actually quite close to having your first working interpreter for MiniML. All that is left is writing a function `eval_s` (the ‘`s`’ is for *substitution semantics*) that evaluates an expression using the substitution semantics rules. (Those rules are, conveniently, described in detail in Chapter 13, and summarized in Figure 13.5.) The `eval_s` function walks an abstract syntax tree of type `expr`, evaluating subparts recursively where necessary and performing substitutions when appropriate. The recursive traversal bottoms out when it gets to primitive values like numbers or booleans or in applying primitive functions like the unary or binary operators to values. It is at this point that the evaluator can see if the operators are being applied to values of the right type, integers for the arithmetic operators, for instance, or integers or booleans for the comparison operators.

For consistency with the environment semantics that you will implement later as the function `eval_d`, both `eval_t` and `eval_s` take

a second argument, an environment, even though neither evaluator needs an environment. Thus your implementation of `eval_s` can just ignore the environment.

We'd also like the various evaluation functions `eval_t`, `eval_s`, `eval_d`, and (if implemented) `eval_l` to all have the same return type as well. Looking ahead, the lexically-scoped environment semantics implemented in `eval_l` must allow for the result of evaluation to go beyond the simple expression values we've used so far. In particular, for the lexical environment semantics, we'll want to add closures as a new sort of value, as described in Section A.4.2. We've provided a variant type `Env.value` that allows for both the simple expression values of the sort that `eval_s` and `eval_d` generate and for closures, which only the environment-based lexical-scoped evaluator needs to generate. For consistency, then, you should make sure that `eval_s`, as well as the later evaluation functions, are of type `Expr.expr -> Env.env -> Env.value`. This will ensure that your code is consistent with our unit tests as well. You'll note that the `eval_t` evaluator that we provide already does this. In order to be type-consistent, it takes an extra `env` argument that it doesn't need or use, and it converts its `expr` argument to the `value` type by adding the `Env.Val` value constructor for that type. (This may help with Exercise 202.)

### Stage 205

Implement the `eval_s : Expr.expr -> Env.env -> Env.value` function in `evaluation.ml`. (You can hold off on completing the implementation of the `Env` module for the time being. That comes into play in later sections.) We recommend that you implement it in stages, from the simplest bits of the language to the most complex. *You'll want to test each stage thoroughly using unit tests as you complete it.* Keep these unit tests around so that you can easily unit test the later versions of the evaluator that you'll develop in future sections.

Using the substitution semantics, you should be able to handle evaluation of all of the MiniML language. If you want to postpone handling of some parts while implementing the evaluator, you can always just raise the `EvalError` exception, which is intended just for this kind of thing, when a MiniML runtime error occurs. Another place `EvalError` will be useful is when a runtime type error occurs, for instance, for the expressions `3 + true` or `3 4` or `let x = true in y`.

Now that you have implemented a function to evaluate expressions, you can make the REPL loop worthy of its name. Notice at the bottom of `evaluation.ml` the definition of `evaluate`, which is the function that the REPL loop in `miniml.ml` calls. Replace the definition with the one calling `eval_s` and the REPL loop will evaluate the read expres-

sion before printing the result. It's more pleasant to read the output expression in concrete rather than abstract syntax, so you can replace the `exp_to_abstract_string` call with a call to `exp_to_concrete_string`. You should end up with behavior like this:

```
# miniml_soln.byte
Entering miniml_soln.byte...
<== 3 ;;
==> 3
<== 3 + 4 ;;
==> 7
<== 3 4 ;;
xx> evaluation error: (3 4) bad redex
<== (((3) ;;
xx> parse error
<== let f = fun x -> x in f f 3 ;;
==> 3
<== let rec f = fun x -> if x = 0 then 1 else x * f (x - 1) in f 4 ;;
xx> evaluation error: not yet implemented: let rec
<== Goodbye.
```

Some things to note about this example:

- The parser that we provide will raise an exception `Parsing.Parse_error` if the input doesn't parse as well-formed MiniML. The REPL handles the exception by printing an appropriate error message.
- The evaluator can raise an exception `Evaluation.EvalError` at runtime if a (well-formed) MiniML expression runs into problems when being evaluated.
- You might also raise `Evaluation.EvalError` for parts of the evaluator that you haven't (yet) implemented, like the tricky `let rec` construction in the example above.

## Stage 206

After you've changed `evaluate` to call `eval_s`, you'll have a complete working implementation of MiniML. As usual, you should save a snapshot of this using a `git commit` and push so that if you have trouble down the line you can always roll back to this version to submit it.

### A.3 Implementing an environment semantics for MiniML

The substitution semantics is sufficient for all of MiniML because it is a pure functional programming language. But binding constructs like

`let` and `let rec` are awkward to implement, and extending the language to handle references, mutability, and imperative programming is impossible. For that, you'll extend the language semantics to make use of an environment that stores a mapping from variables to their values, as described in Chapter 19. We've provided a type signature for environments. It stipulates types for environments and values, and functions to create an empty environment (which we've already implemented for you), to extend an environment with a new BINDING, that is, a mapping of a variable to its (mutable) value, and to look up the value associated with a variable.

The implementation of environments for the purpose of this project follows that described in Section 19.5. We make use of an environment that allows the values to be mutable:

```
type env = (varid * value ref) list
```

This will be helpful in the implementation of recursion.

### Stage 207

Implement the various functions involved in the Env module and test them thoroughly.

How will these environments be used? Atomic literals – like numerals and truth values – evaluate to themselves as usual, independently of the environment. But to evaluate a variable in an environment, we look up the value that the environment assigns to it and return that value.

A slightly more complex case involves function application, as in this example:

```
(fun x -> x + x) 5
```

The abstract syntax for this expression is an application of one expression to another. Recall the environment semantics rule for applications from Figure 19.1:

$$\begin{array}{c}
 E \vdash P \ Q \Downarrow \\
 \left| \begin{array}{c} E \vdash P \Downarrow \text{fun } x \rightarrow B \\ E \vdash Q \Downarrow v_Q \\ E\{x \mapsto v_Q\} \vdash B \Downarrow v_B \end{array} \right. \quad (R_{app}) \\
 \Downarrow v_B
 \end{array}$$

According to this rule, to evaluate an application  $P \ Q$  in an environment  $E$ ,

1. Evaluate  $P$  in  $E$  to a value  $v_P$ , which should be a function of the form `fun x -> B`. If  $v_P$  is not a function, raise an evaluation error.

2. Evaluate  $Q$  in the environment  $E$  to a value  $v_Q$ .
3. Evaluate  $B$  in the environment obtained by extending  $E$  with a binding of  $x$  to  $v_Q$ .

The formal semantics rule translates to what is essentially pseudocode for the interpreter.

In the example: (1) `fun x -> x + x` is already a function, so evaluates to itself. (2) The argument `5` also evaluates to itself. (3) The body `x + x` is thus evaluated in an environment that maps `x` to `5`.

For `let` expressions, a similar evaluation process is used. Recall the semantics rule:

$$\begin{array}{c} E \vdash \text{let } x = D \text{ in } B \Downarrow \\ \left| \begin{array}{l} E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B \end{array} \right. \\ \Downarrow v_B \end{array} \quad (R_{let})$$

We'll apply this rule in evaluating an expression like

```
let x = 3 * 4 in x + 1 ;;
```

To evaluate this expression in, say, the empty environment, we first evaluate (recursively) the definition part in the same empty environment, presumably getting the value `12` back. We then extend the environment to associate that value with the variable `x` to form a new environment, and then evaluate the body `x + 1` in the new environment. In turn, evaluating `x + 1` involves recursively evaluating `x` and `1` in the new environment. The latter is straightforward. The former involves just looking up the variable in the environment, retrieving the previously stored value `12`. The sum can then be computed and returned as the value of the entire `let` expression.

Don't be surprised that this dynamically scoped evaluator exhibits all of the divergences from the substitution-based evaluator that were discussed in Section 19.2.1. For instance, the evaluator will return different values for certain expressions; it will allow `let`-bound variables to be used recursively; and it will fail on simple curried functions. That's fine. Indeed, it's a sign you've implemented the dynamic scope regime correctly. But it does motivate implementation of a lexical-scoped version of the evaluator described below.

### Stage 208

Implement another evaluation function `eval_d : Expr.expr -> Env.env -> Env.value` (the '`d`' is for *dynamically scoped environment semantics*), which works along the lines just discussed. Make sure to test it on a range of tests exercising all the parts of the language.

## A.4 Extending the language

In this final part of the project, you will extend MiniML in one or more ways of your choosing.

### A.4.1 Extension ideas

Here are a few ideas for extending the language, very roughly in order from least to most ambitious. Especially difficult extensions are marked with ♦ symbols.

1. Add additional atomic types (floats, strings, unit, etc.) and corresponding literals and operators.
2. Modify the environment semantics to manifest lexical scope instead of dynamic scope (Section A.4.2).
3. Augment the syntax by allowing for one or more bits of syntactic sugar, such as the curried function definition notation seen in `let f x y z = x + y * z in f 2 3 4.`
4. Add lists to the language.
5. Add records to the language.
6. Add references to the language, by adding operators `ref`, `!`, and `:=`. Since the environment is already mutable, you can even implement this extension without implementing stores and modifying the type of the `eval` function, though you may want to anyway.
7. Add laziness to the language (by adding refs and syntactic sugar for the `lazy` keyword). If you've also added lists, you'll be able to build infinite streams.
8. Add better support for exceptions, for instance, multiple different exception types, exceptions with arguments, exception handling with `try...with....`
9. ♦ Add simple compile-time type checking to the language. For this extension, the language would be extended so that *every* introduction of a bound variable (in a `let`, `let rec`, or `fun` construct) is accompanied by its (monomorphic) type. The abstract syntax would need to be extended to store those types, and you would write a function to walk the tree to verify that every expression in the program is well typed. This is a quite ambitious project.
10. ♦♦ Add type inference to the language, so that (as in OCaml) types are inferred even when not given explicitly. This is *extremely ambitious*, not for the faint of heart. Do not attempt to do this.

Most of the extensions (in fact, all except for (2)) require extensions to the concrete syntax of the language. We provide information about extending the concrete syntax in Section A.4.3. Many other extensions are possible. Don't feel beholden to this list. Be creative!

In the process of extending the language, you may find the need to expand the definition of what an expression is, as codified in the file `expr.mli`. Other modifications may be necessary as well. That is, of course, expected, but you should make sure that you do so in a manner compatible with the existing codebase so that unit tests based on the provided definitions continue to function. The ability to submit your code for testing should help with this process. In particular, if you have to make changes to `mli` files, you'll want to do so in a way that extends the signature, rather than restricting it.

**Most importantly:** It is better to do a great job (clean, elegant design; beautiful style; well thought-out implementation; evocative demonstrations of the extended language; literate writeup) on a smaller extension, than a mediocre job on an ambitious extension. That is, the scope aspect of the project will be weighted substantially less than the design and style aspects. Caveat scriptor.

#### A.4.2 *A lexically scoped environment semantics*

One possible extension is to implement a lexically scoped environment semantics, perhaps with some further extensions. Consider the following OCaml expression, reproduced from Section 19.2.2:

```
let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 3 ;;
```

##### Exercise 209

What should this expression evaluate to? Test it in the OCaml interpreter. Try this expression using your `eval_s` and `eval_d` evaluators. Which ones accord with OCaml's evaluation?

The `eval_d` evaluator that you've implemented so far is *dynamically scoped*. The values of variables are governed by the dynamic ordering in which they are evaluated. But OCaml is *lexically scoped*. The values of variables are governed by the lexical structure of the program. (See Section 19.2.2 for further discussion.) In the case above, when the function `f` is applied to 3, the most recent assignment to `x` is of the value 2, but the assignment to the `x` that lexically outscopes `f` is of the value 1. Thus a dynamically scoped language calculates the body of `f`,  $x + y$ , as  $2 + 3$  (that is, 5) but a lexically scoped language calculates the value as  $1 + 3$  (that is, 4).

The substitution semantics manifests lexical scope, as it should, but the dynamic semantics does not. To fix the dynamic semantics, we need to handle function values differently. When a function value is computed (say the value of `f, fun y -> x + y`), we need to keep track of the lexical environment in which the function occurred so that when the function is eventually applied to an argument, we can evaluate the application in that lexical environment – the environment when the function was *defined* – rather than the dynamic environment – the environment when the function was *called*.

The technique to enable this is to package up the function being defined with a snapshot of the environment at the time of its definition into a closure. There is already provision for closures in the `env` module. You'll notice that the `value` type has two constructors, one for normal values (like numbers, booleans, and the like) and one for closures. The `Closure` constructor just packages together a function with its lexical environment.

### Stage 210

(*if you decide to do a lexically scoped evaluator in service of your extension*) Make a copy of your `eval_d` evaluation function and call it `eval_l` (the ‘l’ for *lexically scoped environment semantics*). Modify the code so that the evaluation of a function returns a closure containing the function itself and the current environment. Modify the function application part so that it evaluates the body of the function in the lexical environment from the corresponding closure (appropriately updated). As usual, test it thoroughly. If you've carefully accumulated good unit tests for the previous evaluators, you should be able to fully test this new one with just a single function call.

Do not just modify `eval_d` to exhibit lexical scope, as this will cause our unit tests for `eval_d` (which assume that it is dynamically scoped) to fail. That's why we ask you to define the lexically scoped evaluator as `eval_l`. The copy-paste recommendation for building `eval_l` from `eval_d` makes for simplicity in the process, but will undoubtedly leave you with redundant code. Once you've got this all working, you may want to think about merging the two implementations so that they share as much code as possible. Various of the abstraction techniques you've learned in the course could be useful here.

*Implementing recursion in the lexically-scoped evaluator* By far the trickiest bit of implementing lexical scope is the treatment of recursion, so we address it separately. Consider this expression, which makes use of an (uninteresting) recursive function:

```
let rec f = fun x -> if x = 0 then x else f (x - 1) in f 2 ;;
```

The `let rec` expression has three parts: a variable name, a definition expression, and a body. To evaluate it, we ought to first evaluate the definition part, but using what environment? If we use the incoming (empty) environment, then what will we use for a value of `f` when we reach it? Ideally, we should use the value of the definition, but we don't have it yet.

Following the approach described in Section 19.6.1, in the interim, we'll extend the environment with a special value, `Unassigned`, as the value of the variable being recursively defined. You may have noticed this special value in the `expr` type; uniquely, it is never generated by the parser. We evaluate the definition in this extended environment, hopefully generating a value. (The definition part better not ever evaluate the variable name though, as it is unassigned; doing so should raise an `EvalError`. An example of this run-time error might be `let rec x = x in x`.) The value returned for the definition can then *replace* the value for the variable name (thus the need for environments to map variables to *mutable* values) and the environment can then be used in evaluating the body.

In the example above, we augment the empty environment with a binding for `f` to `Unassigned` and evaluate `fun x -> if x = 0 then x else f (x - 1)` in that environment. Since this is a function, it is already a value, so evaluates to itself. (Notice how we never had to evaluate `f` in generating this value.)

Now the environment can be updated to have `f` have this function as a value – not extended (using the `extend` function) but \*actually modified\* by replacing the value stored in the `value_ref` associated with `f` in the environment. Finally, the body `f 2` is evaluated in this environment. The body, an application, evaluates `f` by looking it up in this environment yielding `fun x -> if x = 0 then x else f (x - 1)` and evaluates 2 to itself, then evaluates the body of the function in the prevailing environment (in which `f` has its value) augmented with a binding of `x` to 2.

In summary, a `let rec` expression like `let rec x = D in B` is evaluated via the following five-step process:

1. Extend the incoming environment with a binding of `x` to `Unassigned`; call this extended environment `env_x`.
2. Evaluate the definition subexpression `D` in that environment to get a value `v_D`.
3. Mutate `env_x` so that `x` now maps to `v_D`.
4. Evaluate the body subexpression `B` to get a value `v_B`.
5. Return `v_B`.

### A.4.3 *The MiniML parser*

We provided you with a MiniML parser that converts the concrete syntax of MiniML to an abstract syntax representation using the `expr` type. But to extend the implemented language, you'll typically need to extend the parser. Feel free to do so, but make sure that you extend the language by adding new constructs to the `expr` type, without changing the ones that are already given. For instance, if you want to add support for multiple exceptions, you'll want to leave the `Raise` construct as is (so we can test it with our unit tests) and add your own new construct, say `RaiseExn` for the extension.

The parser we provided was implemented using `ocamllex` and `menhir`, programs designed to build lexical analyzers and parser for programming languages. Documentation for them can be found at <http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html>, <http://cambium.inria.fr/~fpottier/menhir/manual.html>, and tutorial material is available at <https://ohama.github.io/ocaml/ocamllex-tutorial/> and <https://dev.realworldocaml.org/parsing-with-ocamllex-and-menhir.html>.

In summary, `ocamllex` takes a specification of the tokens of a programming language in a file, in our case `miniml_lex.mll`. The `ocamlbuild` system knows how to use `ocamllex` to turn such files into OCaml code for a lexical analyzer in the file `miniml_lex.ml`. Similarly, a `menhir` specification of a parser in a file `miniml_parse.mly` will be transformed by `menhir` (automatically with `ocamlbuild`) to a parser in `miniml_parse.ml`. By modifying `miniml_lex.mll` and `miniml_parse.mly`, you can modify the concrete syntax of the MiniML language, which may be useful for many of the extensions you might be interested in.

## A.5 *Submitting the project*

### **Stage 21**

Write up your extensions in a short but formal paper describing and demonstrating any extensions and how you implemented them. Use Markdown or L<sup>A</sup>T<sub>E</sub>X format, and name the file `writeup.md` or `writeup.tex`. You'll submit both the source file and a rendered PDF file.

In addition to submitting the code implementing MiniML to the course grading server through the normal process, you should submit the `writeup.md` or `writeup.tex` file and the rendered PDF file `writeup.pdf` as well.

Make sure to use `git add` to track any new files you create for

the final project (such as your writeup or any code files for testing) before submitting. You can run `git status` to see if there are any untracked files in your repository. Finally, remember that you can look on Gradescope to check that your submissions contains the files you expect. Unfortunately, we can't accept any files that are not submitted on time.

#### A.6 *Alternative final projects*

Students who have been doing exceptionally well in the course to date can petition to do alternative final projects of their own devising, under the following stipulations:

1. Alternative final projects can be undertaken individually or in groups of up to four.
2. The implementation language for the project must be OCaml.
3. You will want to talk to course staff about your ideas early to get initial feedback.
4. You will need to submit a proposal for the project by April 16, 2021. The proposal should describe what the project goals are, how you will go about implementing the project, and how the work will be distributed among the members of the group (if applicable).
5. You will receive notification around April 19, 2021 as to whether your request has been approved. Approval will be based on performance in the course to date and the appropriateness of the project.
6. You will submit a progress report by April 26, 2021, including a statement of progress, any code developed to date, and any changes to the expected scope of the project.
7. You will submit the project results, including all code, a demonstration of the project system in action, and a paper describing the project and any results, by May 5, 2021.
8. You will be scheduled to perform a presentation and demonstration of your project for course staff during reading period.
9. The group as a whole may drop out of the process at any time. Individual members of the group would then submit instead the standard final project described here.

# 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  |
|          |           | (3, 3)    | (-2, 5) |
| 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.<sup>1</sup> 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.

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.

<sup>1</sup> 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!

## 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!

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



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.

`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.

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



Figure A.2: A rare daguerreotype 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](#))

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.

### 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-

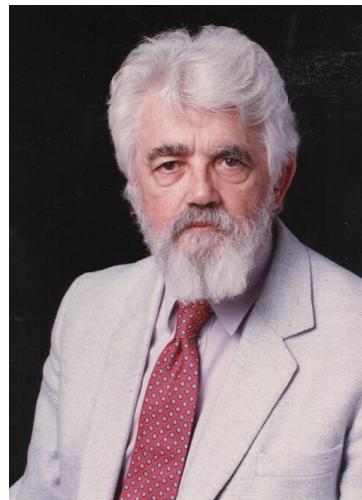


Figure A.3: John McCarthy (1927–2011), one of the founders of (and coiner of the term) artificial intelligence. His LISP programming language was widely influential in the history of programming languages. He was awarded the Turing Award in 1971.



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.

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](#).

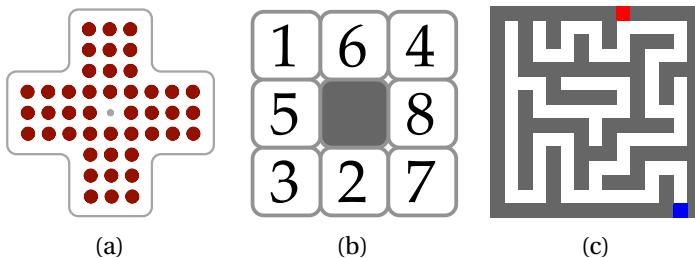


Figure A.5: Some puzzles based on search for a goal state. (a) the peg solitaire puzzle; (b) the sliding-tile 8 puzzle; (c) a maze puzzle.

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?

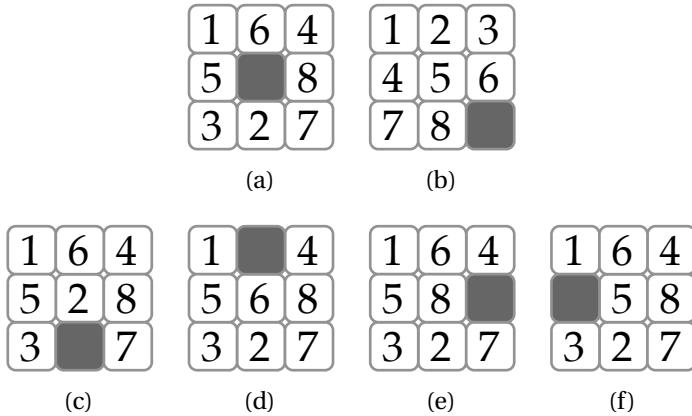


Figure A.6: The 8 puzzle: (a) an initial state, (b) the goal state, (c-f) the states resulting from moving up, down, left, and right from the initial state, respectively.

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.

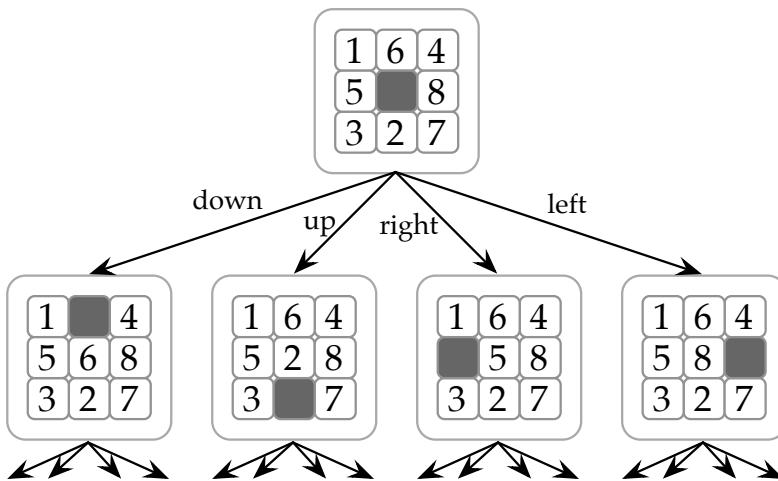


Figure A.7: A snippet from the search tree for the 8 puzzle.

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

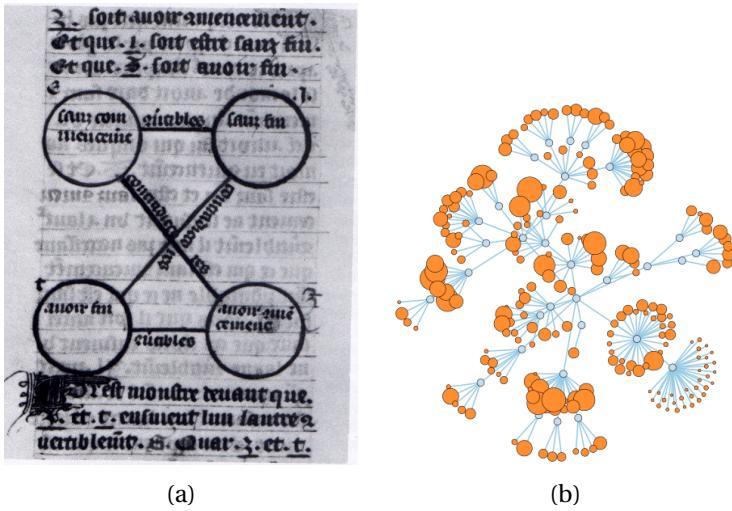


Figure A.8: Two sample graph drawings several hundred years apart. (a) A graph drawing from the 14th century with nodes depicting logical propositions in an argument and edges depicting relations among them. From Kruja et al. (2001). (b) Snapshot of a dynamic interactive force-directed graph drawing 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-

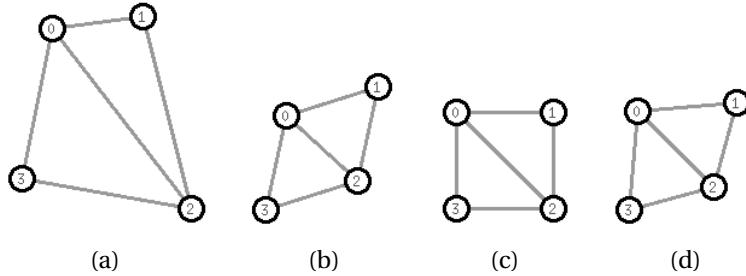
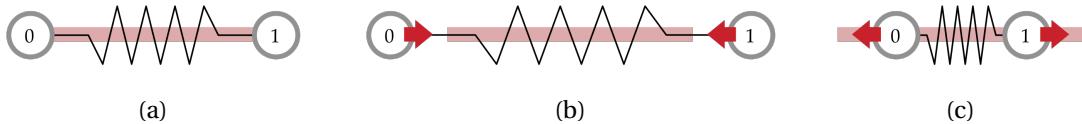


Figure A.9: Four different drawings of the same graph. (a) Nodes randomly placed. (b) With fixed length spring constraints between nodes connected by edges. (c) With fixed length spring constraints between nodes connected by outside edges, plus a horizontal alignment constraint on nodes 0 and 1 and a 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.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.

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.



# 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:

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

Sometimes the cases are depicted overtly using a large brace:

$$\text{fact}(n) = \left\{ \begin{array}{ll} 1 & \text{for } n = 0 \\ n \cdot \text{fact}(n - 1) & \text{for } n > 0 \end{array} \right.$$

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:  $\text{fact}(n)$ .

If we call this profit function  $P$ , we can use **arrow notation** and write the rule  $P: n \rightarrow 5n - 500$ , 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  $5n - 500$ .“

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.

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.

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



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

§. 7. Fit autem  $d\alpha - \frac{x d\alpha}{a}$  integrabile si multiplicatur per  $\frac{1}{a}$ , integrale enim erit  $\frac{x}{a} + c$ , designante  $c$  quantitatem constantem quamcunque ab  $a$  non dependet. 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  $f(\frac{x}{a} + c)$ . 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  $\cdot$  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

---


$$(f(x) + g(x))' = f'(x) + g'(x)$$

$$(f(x) - g(x))' = f'(x) - g'(x)$$

$$(f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$$

$$\left(\frac{f(x)}{g(x)}\right)' = \frac{(f'(x) \cdot g(x) - f(x) \cdot g'(x))}{g(x)^2}$$

$$(\sin f(x))' = f'(x) \cdot \cos f(x)$$

$$(\cos f(x))' = f'(x) \cdot -\sin f(x)$$

$$(\ln f(x))' = \frac{f'(x)}{f(x)}$$

$$(f(x)^h)' = h \cdot f'(x) \cdot f(x)^{h-1}$$

where  $h$  contains no variables

$$(f(x)^{g(x)})' = f(x)^{g(x)} \cdot \left( g'(x) \cdot \ln f(x) + \frac{f'(x) \cdot g(x)}{f(x)} \right)$$

$$(n)' = 0 \quad \text{where } n \text{ is any constant}$$

$$(x)' = 1$$


---

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  $x^3$ .<sup>1</sup>
- 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

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

<sup>1</sup> 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,  $\text{subst}(x, P, Q)$ ) 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

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

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 ( $\lambda$ ), 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  $\lambda$  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:<br><code>fun n -&gt; sqrt(n *. n)</code>     |
| $\lambda n. (m \cdot n^2)$            | the function from $n$ to $m \cdot n^2$ , so that $m$ is implicitly being viewed as a constant:<br><code>fun n -&gt; m *. (n *. n)</code> |
| $\lambda m. (m \cdot n^2)$            | the function from $m$ to $m \cdot n^2$ , so that $n$ is implicitly being viewed as a constant:<br><code>fun m -&gt; m *. (n *. n)</code> |
| $\lambda m. \lambda n. (m \cdot n^2)$ | the function from $m$ to a function from $n$ to $m \cdot n^2$ :<br><code>fun m -&gt; fun n -&gt; m *. (n *. n)</code>                    |

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 \(2019\)](#).

### 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 ( $90^\circ$ ) 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.

| $p$   | $q$   | $p$ 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.

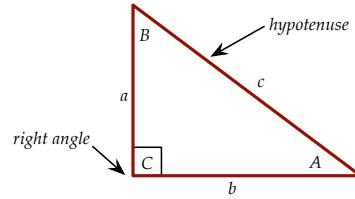


Figure B.6: A right triangle. Angle C is a right angle. The opposite side, of length  $c$ , is the hypotenuse. By Pythagorus's theorem,  $a^2 + b^2 = c^2$ .

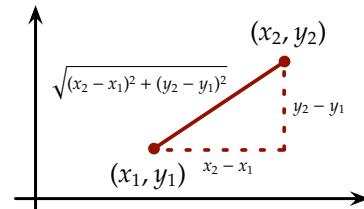


Figure B.7: Two points, given by a pair of their  $x$  (horizontal) and  $y$  (vertical) coordinates. The slope of the line between them is  $\frac{y_2 - y_1}{x_2 - x_1}$ . The distance between them, as per the Pythagorean theorem, is  $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ .

The ratio of the circumference of a circle and its diameter is (non-trivially, and perhaps surprisingly) a constant, conventionally called  $\pi$  (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  $A = \pi r^2$ .

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  $\{1, 2, 3, \dots\}$ ) in cases where the rule for filling in the remaining elements is sufficiently obvious to the reader.

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  $\{x \mid x \text{ mod } 2 = 0\}$ , 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  $\emptyset$  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

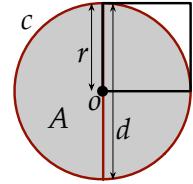


Figure B.8: Geometry of the circle at origin  $o$  of radius  $r$ , diameter  $d = 2r$ , circumference  $c$ , and area  $A$ .

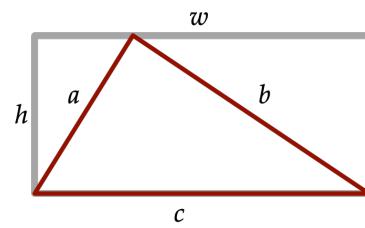


Figure B.9: A triangle and a circumscribing rectangle, with labeled edge lengths.

*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  $\equiv$  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:

$$x[x \mapsto P] = P$$

$$y[x \mapsto P] = y \quad \text{where } x \neq y$$

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  $x \neq y$  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.<sup>1</sup>

*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](#).

<sup>1</sup> 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.))

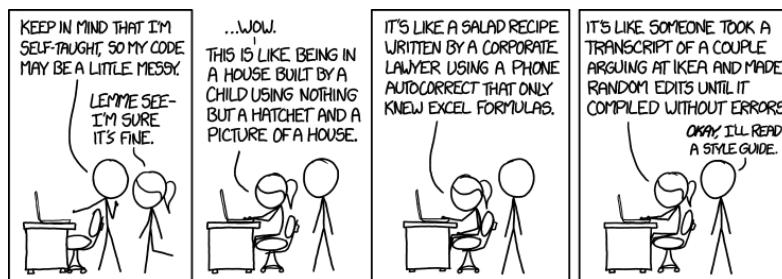


Figure C.1: Yes, coding style is important.

## 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 sparingly*

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`  $\langle \rangle$  `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.

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

```
X let f arg1 arg2 =
  let x = fst arg1 in
  let y = snd arg1 in
  let z = fst arg2 in
  ...
✓ let f (x, y) (z, _) =
  ...
X 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:

```
X 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.

```
X 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 Verbose

### 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:

| ✗                                      | ✓                               |
|----------------------------------------|---------------------------------|
| <code>if e then true else false</code> | <code>e</code>                  |
| <code>if e then false else true</code> | <code>not e</code>              |
| <code>if e then e else false</code>    | <code>e</code>                  |
| <code>if x then true else y</code>     | <code>x    y</code>             |
| <code>if x then y else false</code>    | <code>x &amp;&amp; y</code>     |
| <code>if x then false else y</code>    | <code>not x &amp;&amp; y</code> |

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:

---

| ✗                                             | ✓                                            |
|-----------------------------------------------|----------------------------------------------|
| <code>x :: []</code>                          | <code>[x]</code>                             |
| <code>length + 0</code>                       | <code>length</code>                          |
| <code>length * 1</code>                       | <code>length</code>                          |
| <code>big_expression * big_expression</code>  | <code>let x = big_expression in x * x</code> |
| <code>if x then f a b c1 else f a b c2</code> | <code>f a b (if x then c1 else c2)</code>    |
| <code>String.compare x y = 0</code>           | <code>x = y</code>                           |
| <code>String.compare x y &lt; 0</code>        | <code>x &lt; y</code>                        |
| <code>String.compare y x &lt; 0</code>        | <code>x &gt; y</code>                        |

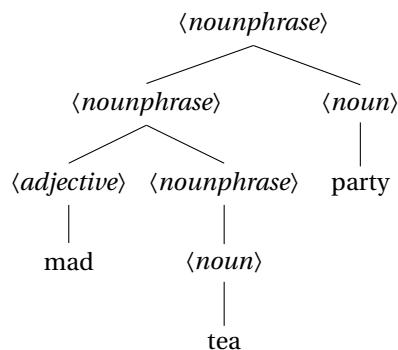
---



*D*

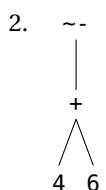
## *Solutions to selected exercises*

### **Solution to Exercise 3**

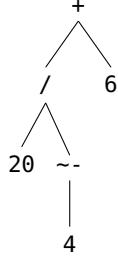


**Solution to Exercise 4** There are three structures given the rules provided, corresponding to eaters of flying purple people, flying eaters of purple people, and flying purple eaters of people.

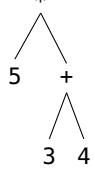
### **Solution to Exercise 6**



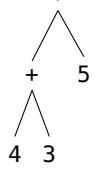
3.



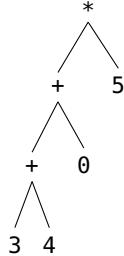
4.



5.



6.



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

1.  $\sim (1 + 42)$
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 = 3.1416000000000039
```

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

```

- 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 ...  $\text{list} \rightarrow 'a \rightarrow 'a$ . 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 ' $a \rightarrow 'a$ '. The final typing is  $f : ('a \rightarrow 'a) \text{list} \rightarrow 'a \rightarrow 'a$ .

```

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

```

- 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  $f : \text{bool} * 'a \rightarrow ('a \rightarrow \text{bool}) \rightarrow \text{bool}$ .

```

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

```

- 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 ' $a \rightarrow 'a \rightarrow 'b$ '. 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  $f : ('a \rightarrow 'a \rightarrow 'b) \rightarrow 'a \rightarrow 'b$ .

```

# let f x y =
#   x y y ;;
val f : ('a \rightarrow 'a \rightarrow 'b) \rightarrow 'a \rightarrow '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` `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 performs 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 `ints` 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 `ints` 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 cannot 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 following 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 `(leafft -> '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_count`s 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 need 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 `node : '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 `Some (node +. wt1 +. wt2)`. 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  $\text{fun } x \rightarrow B$  evaluates to itself.”

$R_{app}$ : “To evaluate an application of the form  $P \ Q$ , first evaluate  $P$  to a function value of the form  $\text{fun } x \rightarrow B$  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  $((\text{fun } y \rightarrow f \ 3) \ 1)[f \mapsto \text{fun } x \rightarrow y]$ . 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  $(\text{fun } z \rightarrow (\text{fun } x \rightarrow y) \ 3) \ 1$ , 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 ↦ P] = let x = Q[x ↦ P] in R
(let y = Q in R)[x ↦ P] = let y = Q[x ↦ P] in R[x ↦ P]
where x ≠ y and y ∉ FV(P)

(let y = Q in R)[x ↦ P] = let z = Q[x ↦ P] in R[y ↦ z][x ↦ P]
where x ≠ y and y ∈ 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 yst)
#              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 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 mlist =
#   match !mlist 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
#           cons hd first :: second

```

```

#           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**

$\emptyset \vdash \text{let } x = 3 \text{ in let } y = 5 \text{ in } x + y$

$$\begin{array}{c} \downarrow \\ \{\} \vdash 3 \Downarrow 3 \\ \{x \mapsto 3\} \vdash \text{let } y = 5 \text{ in } x + y \\ \downarrow \\ \{x \mapsto 3\} \vdash 5 \Downarrow 5 \\ \{x \mapsto 3; y \mapsto 5\} \vdash x + y \\ \downarrow \\ \{x \mapsto 3; y \mapsto 5\} \vdash x \Downarrow 3 \\ \{x \mapsto 3; y \mapsto 5\} \vdash y \Downarrow 5 \\ \downarrow 8 \\ \Downarrow 8 \end{array}$$
**Solution to Exercise 185**

$\emptyset \vdash \text{let } x = 3 \text{ in let } x = 5 \text{ in } x + y$

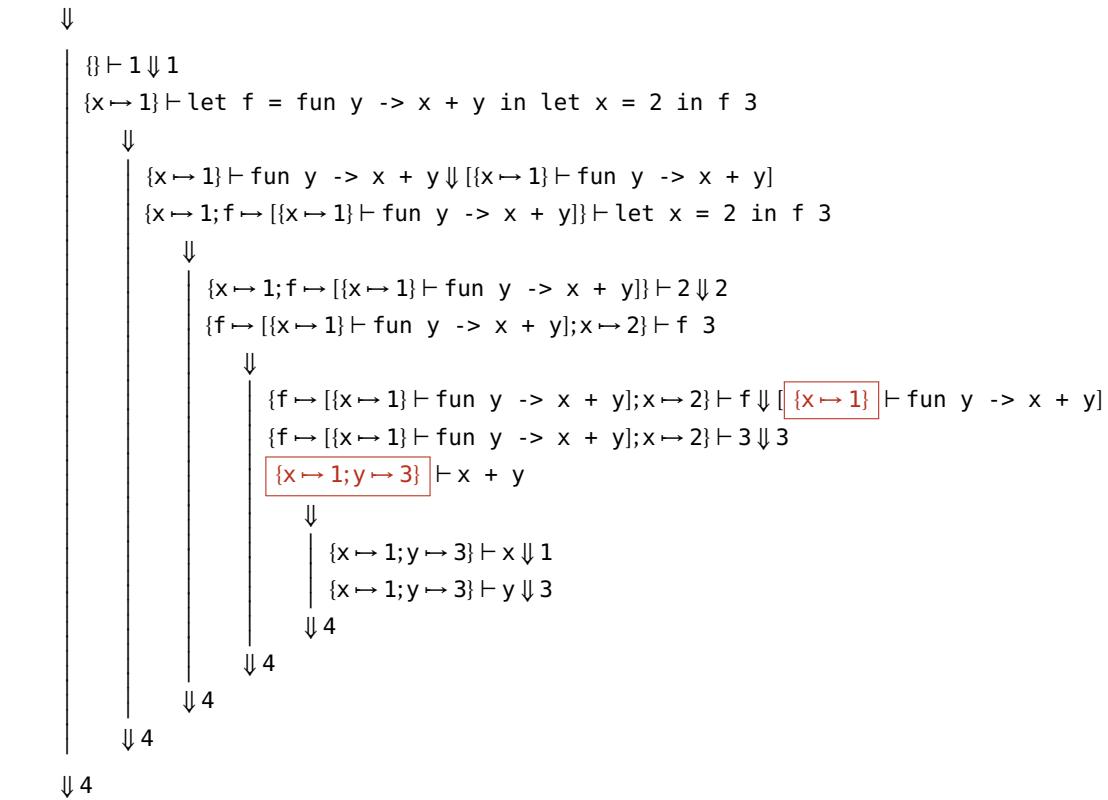
$$\begin{array}{c} \downarrow \\ \{\} \vdash 3 \Downarrow 3 \\ \{x \mapsto 3\} \vdash \text{let } x = 5 \text{ in } x + x \\ \downarrow \\ \{x \mapsto 3\} \vdash 5 \Downarrow 5 \\ \{x \mapsto 5\} \vdash x + x \\ \downarrow \\ \{x \mapsto 5\} \vdash x \Downarrow 5 \\ \{x \mapsto 5\} \vdash x \Downarrow 5 \\ \downarrow 10 \\ \Downarrow 10 \end{array}$$
**Solution to Exercise 186**

- $R_{fun}$ : “A function expression of the form  $\text{fun } x \rightarrow B$  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  $\text{fun } x$

->  $B$  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:

```
{} ⊢ let x = 1 in let f = fun y -> x + y in let x = 2 in f 3
```



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

$$E \vdash \text{true} \Downarrow \text{true} \quad (R_{\text{true}})$$

$$E \vdash \text{false} \Downarrow \text{false} \quad (R_{\text{false}})$$

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

$$\begin{array}{c} E \vdash \text{if } C \text{ then } T \text{ else } F \Downarrow \\ \left| \begin{array}{c} E \vdash C \Downarrow \text{true} \\ E \vdash T \Downarrow v_T \end{array} \right. \quad (R_{\text{ifthen}}) \\ \Downarrow v_T \end{array}$$

$$\begin{array}{c} E \vdash \text{if } C \text{ then } T \text{ else } F \Downarrow \\ \left| \begin{array}{c} E \vdash C \Downarrow \text{false} \\ E \vdash F \Downarrow v_F \end{array} \right. \quad (R_{\text{ifelse}}) \\ \Downarrow v_F \end{array}$$

**Solution to Exercise 193**

$$\begin{array}{c} E, S \vdash ! P \Downarrow \\ \left| \begin{array}{c} E, S \vdash P \Downarrow l, S' \\ \Downarrow S'(l), S' \end{array} \right. \quad (R_{\text{deref}}) \end{array}$$

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  $Q$ .

$$\begin{array}{c} E, S \vdash P ; Q \Downarrow \\ \left| \begin{array}{c} E, S \vdash P \Downarrow (), S' \\ E, S' \vdash Q \Downarrow v_Q, S'' \end{array} \right. \quad (R_{\text{seq}}) \\ \Downarrow v_Q, S'' \end{array}$$

**Solution to Exercise 195** We start by taking

`let rec x = D in B`

to be equivalent to

`let x = ref unassigned in (x := D'); B'`

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

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

$E, S \vdash \text{let } x = \text{ref } U \text{ in } (x := D'); B'$

$\Downarrow$

$E, S \vdash \text{ref } U$

$\Downarrow$

$E, S \vdash U \Downarrow U, S$

$\Downarrow l, S\{l \mapsto U\}$

$E\{x \mapsto l\}, S\{l \mapsto U\} \vdash (x := D'); B'$

$\Downarrow$

$E\{x \mapsto l\}, S\{l \mapsto U\} \vdash x := D'$

$\Downarrow$

$E\{x \mapsto l\}, S\{l \mapsto U\} \vdash x \Downarrow l, S\{l \mapsto U\}$

$E\{x \mapsto l\}, S\{l \mapsto U\} \vdash \boxed{D'}$

$\Downarrow$

$\dots$

$\Downarrow v_D, S'$

$\Downarrow (\ ), S'\{l \mapsto v_D\}$

$E\{x \mapsto l\}, S'\{l \mapsto v_D\} \vdash \boxed{B'}$

$\Downarrow$

$\dots$

$\Downarrow v_B, S''$

$\Downarrow v_B, S''$

$\left. \begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array} \right\}$

$\left. \begin{array}{c} \\ \\ \\ \\ \\ \end{array} \right\}$

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 `let rec x = D in B` (now with abbreviations expanded) that incorporates these two subderivations as premises:

$E, S \vdash \text{let } \text{rec } x = D \text{ in } B \Downarrow$

$\left| \begin{array}{c} E\{x \mapsto l\}, S\{l \mapsto \text{unassigned}\} \vdash D[x \mapsto !x] \Downarrow v_D, S' \\ E\{x \mapsto l\}, S'\{l \mapsto v_D\} \vdash B[x \mapsto !x] \Downarrow v_B, S'' \end{array} \right.$

$\Downarrow v_B, S''$

$(R_{letrec})$ 

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. let updated = balance - amt in	if balance >= amt then begin
3. balance <- updated;	
4. amt	
5. ...	
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=H4ICAAIAAJ&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](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 <http://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/ucl.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](https://doi.acm.org/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  $\lambda$ -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**
- Dijkstra, 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  
 $\pi$ , 51  
 $\pi$ , 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. <a href="#">Portrait of Donald Knuth</a> , copyright 2010 Photographic Unit, University of Glasgow. . . . .	223
Figure 15.2. <a href="#">Heartbleed logo</a> by Leena Snidate/Codenomicon, in the public domain by CC0 dedication, courtesy of Wikipedia. . . . .	253
Figure 17.3. <a href="#">Small copy of a portrait of Brook Taylor, the English mathematician</a> . 1720. National Portrait Gallery (London). Public domain, courtesy of Wikipedia. . . . .	294
Figure 17.7. <a href="#">Photo of Peter Landin</a> , courtesy of Wikipedia. . . . .	301
Figure 18.3. <a href="#">Photo of Alan Kay</a> , by Alan Kay, used under a CC-by 2.0 Generic license. Courtesy of Wikimedia Commons. <a href="#">Photo of Adele Goldberg</a> , by Terry Hancock, used under a CC-by-sa 2.5 Generic license. Courtesy of Wikimedia Commons. <a href="#">Photo of Dan Ingalls</a> , used under a CC-by-sa 3.0 Unported license. Courtesy of Wikimedia Commons. . . . .	304
Figure 20.1. <a href="#">Figure from Moore (1965)</a> . . . . .	341
Figure 20.1. <a href="#">Computer performance data</a> from Wikipedia. Used by permission (CC-BY-SA). . . . .	342
Figure A.1. <a href="#">Portrait of Whitfield Diffie</a> by Duncan Hall, courtesy of Wikimedia, licensed under CC BY-SA 4.0. <a href="#">Portrait of Martin Hellman</a> , courtesy of Wikimedia, licensed under CC BY-SA 3.0. . . . .	379
Figure A.2. <a href="#">Daguerrotype of Augusta Ada King, Countess of Lovelace</a> by Antoine Claudet about 1843. Work in the public domain. . . . .	380
Figure A.3. <a href="#">Photograph of John McCarthy</a> . Used by implicit permission of the author. . . . .	381
Figure A.4. <a href="#">Scanned photograph of Jean Sammet</a> . 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. <i>Algebra: Structure and Method: Book 1</i> : California Edition. Evanston Illinois: McDougal Littell, 2000. Page 379. . . . .	390
Figure B.2. <a href="#">Portrait of Leonhard Euler (1707-1783)</a> . Jakob Emanuel Handmann. Public domain; courtesy of Wikipedia. . . . .	390
Figure B.2. Selection from Leonhard Euler [Leonh. Eulero]. 1734-5. <i>Additamentum ad Dissertationem de Infinitis Curvis Eiusdem Generis</i> [An Addition to the Dissertation Concerning an Infinite Number of Curves of the Same Kind]. In <i>Commentarii Academiae Scientiarum Imperialis Petropolitanae [Memoirs of the Imperial Academy of Sciences in St. Petersburg]</i> , Volume VII (1734-35). Petropoli, Typis Academiae, 1740. Photo of the author. . . . .	391
Figure C.0. <a href="#">XKCD comic 1513</a> , “Code Quality”, by Randall Munro. CC-BY-NC 2.5, courtesy of Randall Munro. . . . .	399