Seven Lessons in Program Design

Norman Ramsey
Tufts University

Contents

Steps

1. *Forms of data.* Using the descriptions given to you, understand the forms of the data that will be input to the function. Forms might be described by proof rules, by list of cases, or even by a type definition. However they are described, the forms of data must be *distinguishable* by writing code.

2. *Example inputs.* For each form of data, write an example input which has that form. Every form of data needs an example.

3. *Function's name.* If it's not already given to you, choose a name for the function. Use a noun, verb, or property, as described in the general coding rubric, under "Naming."

4. *Function's contract.* If it's not already given to you, write the function's contract: in a simple, clear sentence, what should the function return, as determined by its argument(s)?

5. *Example results.* Look the example inputs from part 2. On each example, what should the function return? Write your answer as a `check-expect` or `check-assert` unit test.

6. *Algebraic laws.* Generalize the example unit tests to algebraic laws. Some values in the examples will turn into variables. This step may be hard.

7. *Code case analysis.* Looking at only the *left-hand sides* of the algebraic laws, start coding the body of the function. The function should begin by asking the input data, "How were you formed?," which tells the code which algebraic law to follow. Code one case per algebraic law. Distinguish cases using `if`-expressions.

8. *Code results.* Finish the function. For each case, ask the input, "From what parts, if any, were you formed?" Then, using the *right-hand side* of the corresponding algebraic law, compute the results.

9. *Revisit tests.* Revisit the unit tests. First, look at them. Do they test every form of input? If the function's result is Boolean, add new tests so that you have both a "true" and a "false" test for each form of input.

   Next, run them. If there are any test failures, look at the algebraic laws first.

Rationale

1. The shape of the data determines the shape of the code. This idea, popularized by Fred Brooks in *The Mythical Man-Month*, applies to many programming languages and paradigms. It has been known since the 1960s.

2. Examples are the easiest place to start, and they are what people learn from.

3. A meaningful name is critical to code review. By writing it early, you clarify what you are aiming for.

4. Contracts aren't just useful in code review: writing the contract first is a form of "design first, code later," which you may have practiced in Comp 40. And the contract can help alert you to a design that is too complex; if your contract isn't simple, your code may be hard to get right.

   If you've been taught to think of a contract only as documentation you write after the fact, you may be surprised at how much you get out of a "contract first" approach.

5. Writing down results on example inputs ensures that we know where we're going. If something is going to be wrong, misunderstood, or confusing, we want to identify it early—for example, before we start coding the wrong function.

   Writing examples as unit tests gives the interpreter the job of checking that everything works as expected—every time. If anything goes wrong with your code, you want the bug to manifest as a failed unit test.

6. Algebraic laws are the single most powerful tool you will learn in Comp 105. They occupy a middle ground between vague English and executable code, where they simplify both coding and review.

7. Case analysis is always the enemy. This step shows you where you *must* have it.

8. This step reduces the coding task to a bite-sized piece involving one case at a time.

9. Adding test cases for both "true" and "false" results finds many bugs, as does actually running the unit tests.

A nine-step design process for functions

Introduction: Why program design?

One reason to get a university education in computer science, as opposed to training at a boot camp or a code academy, is to prepare yourself to write code from scratch using a language you've never seen before. To succeed at such a task, it helps to deploy programming techniques that transcend language. Instead of spitting out fragments of code you saw somewhere, or "debugging a blank screen," you can tackle new programming tasks using a systematic design process.

If this booklet is your first encounter with a systematic design process, it may look overly detailed, bureaucratic, or even pedantic. Fortunately, bureaucracy has its limits. It should help to know that it plays three roles:

- A systematic process is necessary only when something is hard. If things are going great and you just write down code that works, you don't need a systematic process. The time to apply process is when you're stuck, or something goes wrong, or you need help, or simply when you feel that getting your code to work takes too much time.

  If you approach our teaching assistants for help with programming or debugging, they will ask about your design process.

- A systematic process is tiresome to learn—there are a lot of steps to keep in mind—but if you can master it to a point where you can apply it without thinking, it is surprisingly helpful. You get the benefits without any marginal cost.

  You are not expected to reach that level of mastery this semester, but I encourage you to apply key parts of the process, such as unit tests and algebraic laws, to your work outside of this class. Many students report excellent results applying systematic design on internship, for example.

- A systematic process is learned by applying it to easy problems—typically problems that could easily be solved *without* a systematic process. Such applications of systematic design might seem pointless, but there is a point: when you are ready to tackle a hard problem like a Boolean-formula solver, you have the tools.

  You will be expected to demonstrate parts of your design process, especially unit tests and algebraic laws, on some very simple problems—sometimes so simple that systematic design seems like overkill. The goal is not to solve an easy problem; the goal is to learn design.

The design process we use is founded on one key technique: understand your input data, then let the shape of your input data determine the shape of your code. This technique was suggested in the 1970s by Brooks, developed into an industrial design method in the 1980s by Jackson, and refined for beginning programmers in the 1990s by Flatt, Felleisen, Findler, and Krishnamurthi. In this series of lessons, we look at data defined by induction, generalize data examples to *algebraic laws*, and turn algebraic laws into code. Inspired by an established industry practice called *test-driven development*, we also generalize data examples into test cases.

Our full design process is shown on the facing page. The steps are not just for beginners: professional software engineers value effective use of names, contracts (and their associated algebraic laws), and unit tests.

Each lesson in the book applies the process in a different context: usually a form of data, a language, or both. You'll learn how to work with natural numbers; with lists, trees, and other symbolic expressions; with higher-order functions; with pattern matching; with types; with proof systems; with abstract datatypes; and with objects.

1. Proof systems and program design

In this lesson, we start to develop our code-from-data technique by examining five inductive definitions of the natural numbers, then looking at recursive functions that we derive using the definitions. The lesson introduces proof systems for describing forms of data, algebraic laws for specifying behavior, and complete examples of our design process for going from data to laws to code. This lesson explores only proofs, laws, and functions that compute with natural numbers.

1.1 Formal judgment and sequents

A regular person says something like "7 is a natural number." A semanticist[1] also says "7 is a natural number," but when they write it down, they write something like this:

$$\vdash 7\ \texttt{nat}.$$

Roughly speaking, what they mean is, "without having to make any assumptions, I claim that 7 is a natural number." The notation is an example of a *sequent* from mathematical logic, and the general form is like this:

$$\textit{context} \vdash \textit{statement}.$$

A *context* usually contains assumptions, and because 7 is a natural number regardless of assumptions, the sequent "$\vdash 7\ \texttt{nat}$" doesn't need any assumptions.

A sequent is just one form of *formal judgment*, which is how a semanticist states a claim precisely. Formal judgments play a major role in the second homework (and in programming languages more generally), and sequents are used to express judgments in type systems, which we study in mid-semester.

When we speak a sequent out loud, we don't usually pronounce the $\vdash$ symbol, but when we need to talk about the symbol, we call it "the turnstile."

1.2 Proofs and inference rules

A sequent is just a claim. As in real life, some claims are good, like "7 is a natural number," but some claims are bad, like "the square root of 2 is a natural number." We'd like to distinguish good claims from bad ones. Truth is always good, but truth is often impossible to establish. Instead, we focus on *provability.* To *prove* a judgment, as opposed to just stating it, we use a *proof system.* If the proof system is designed properly, all provable claims are true, and therefore, no false claims are provable. For example, I can write "$\vdash \sqrt{2}\ \texttt{nat}$," but I'd better not be able to prove it. (Not all true claims are provable; if you have heard of "Gödel's incompleteness theorem," it constructs a claim that is true but not provable.)

The proof systems we use are composed of *inference rules.* An inference rule can be identified by its long horizontal line. Below the line you will find a single judgment, the *conclusion.* Above the line you will find some number of judgments, called *premises.* The rule means "if you can prove every premise (above the line), you may apply this rule, after which you are considered to have proven the conclusion (below the line)." For example, if you can prove that m is a natural number, you can also prove that $m+1$ is a natural number. The rule is called Successor:

$$\textsc{Successor}\ \frac{\vdash_P m\ \texttt{nat}}{\vdash_P (m+1)\ \texttt{nat}}$$

The capital P dangling off the turnstile is a way to label this rule as belonging to a particular proof system—one of five in the lesson.

A good way to read the Successor rule is, "if you want to prove that $m+1$ is a natural number, you first have to prove that m is a natural number." This reading is good because it's like writing a recursive function: if you want to compute a function of $m+1$, you might first recursively call the function on m.[2] This trick is pretty good, and it covers every natural number except zero (the only one that can't be written in the form $m+1$, where m is also a natural number). But zero is also a natural number, and it needs a proof rule:

$$\textsc{Zero}\ \frac{}{\vdash_P 0\ \texttt{nat}}$$

Another good way to read this rule is this "if you want to prove that 0 is a natural number, there's nothing else you have to prove first—you're done." It's a bit like writing a recursive function: when you see an argument of 0, you don't have to make a recursive call.

[1] Someone who studies the meanings of languages.

[2] Perhaps you are more accustomed to think "if I want to compute a function of n, I might first recursively call the function on $n-1$." I like this thinking, but I wouldn't want to write the Successor rule this way. When writing a specification like this, we use m and $m+1$ because then the rule works for any natural number m—the rule is not restricted to, say, natural numbers greater than zero.

1.3 Five proof systems

When you write a recursive function on natural numbers, you have a lot of ways to structure the recursion. Ideally, the recursive structure of your function follows from the structure you use to describe the numbers. And a structure for describing natural numbers can be expressed in a proof system. This lesson presents five example proof systems. All except the last are based on numbering systems; the last is based on parity (even versus odd).

Peano numerals

The simplest and most standard way to characterize the natural numbers uses axioms posited by mathematician Giuseppe Peano: a natural number is either zero or is the successor of some other natural number. You may have studied these axioms in math class. Here is Peano's characterization presented as a formal proof system, identified with a subscript P on the turnstile. The rules are the two rules you've already seen, but under different names:

$$\textsc{PeanoZero}\ \frac{}{\vdash_P 0\ \texttt{nat}} \qquad \textsc{Successor}\ \frac{\vdash_P m\ \texttt{nat}}{\vdash_P (m+1)\ \texttt{nat}}$$

Binary ``numbers''

Computer scientists say "binary number," but a mathematician would balk—the binary system is a numeration system, and a "binary number" is just a numeral. A natural number is either zero or is twice a natural number m plus a bit b.

$$\textsc{BinaryZero}\ \frac{}{\vdash_B 0\ \texttt{nat}}$$

$$\textsc{BinaryNat}\ \frac{\vdash_B m\ \texttt{nat} \qquad \vdash b\ \texttt{bit}}{\vdash_B (2 \times m + b)\ \texttt{nat}}$$

A bit is either zero or one:

$$\textsc{BitZero}\ \frac{}{\vdash 0\ \texttt{bit}} \qquad \textsc{BitOne}\ \frac{}{\vdash 1\ \texttt{bit}}$$

Bits are bits regardless of proof system, so when I write $\vdash 0\ \texttt{bit}$ or $\vdash 1\ \texttt{bit}$, I don't decorate the turnstile.

A decimal system for arithmetic

The decimal (also called Arabic) numerals have a proof system very similar to "binary numbers." A natural number is either zero or is ten times a natural number m plus a decimal digit d.

$$\textsc{DecimalZero}\ \frac{}{\vdash_D 0\ \texttt{nat}}$$

$$\textsc{DecimalNat}\ \frac{\vdash_D m\ \texttt{nat} \qquad \vdash d\ \texttt{digit}}{\vdash_D (10 \times m + d)\ \texttt{nat}}$$

Proving that d is a digit requires ten highly repetitive rules:

$$\textsc{Digit0}\ \frac{}{\vdash 0\ \texttt{digit}} \qquad \textsc{Digit1}\ \frac{}{\vdash 1\ \texttt{digit}}$$

$$\textsc{Digit2}\ \frac{}{\vdash 2\ \texttt{digit}} \qquad \textsc{Digit3}\ \frac{}{\vdash 3\ \texttt{digit}}$$

$$\textsc{Digit4}\ \frac{}{\vdash 4\ \texttt{digit}} \qquad \textsc{Digit5}\ \frac{}{\vdash 5\ \texttt{digit}}$$

$$\textsc{Digit6}\ \frac{}{\vdash 6\ \texttt{digit}} \qquad \textsc{Digit7}\ \frac{}{\vdash 7\ \texttt{digit}}$$

$$\textsc{Digit8}\ \frac{}{\vdash 8\ \texttt{digit}} \qquad \textsc{Digit9}\ \frac{}{\vdash 9\ \texttt{digit}}$$

This proof system is great for guiding implementations of arithmetic on natural numbers, including addition, subtraction, multiplication, and division.

A decimal system for numeration

The Decimal proof system is useful for arithmetic, but it is not at all good for looking at properties of numerals. For example, if you want to know if a number n is represented by a numeral that is all 4's, you should avoid the Decimal system.[3] Instead, you should prefer this DecNumeral system:

$$\textsc{DecNumeralDigit}\ \frac{\vdash d\ \texttt{digit}}{\vdash_{DN} d\ \texttt{nat}}$$

$$\textsc{DecNumeralNat}\ \frac{\begin{array}{c}\vdash_{DN} m\ \texttt{nat}\\ m \neq 0 \qquad \vdash d\ \texttt{digit}\end{array}}{\vdash_{DN} (10 \times m + d)\ \texttt{nat}}$$

Parity

This strange little proof system relies on numbers being even or odd. It says that a natural number is zero, or it's one, or it's two plus another natural number:

$$\textsc{EvenParity}\ \frac{}{\vdash_E 0\ \texttt{nat}} \qquad \textsc{OddParity}\ \frac{}{\vdash_E 1\ \texttt{nat}}$$

$$\textsc{SameParity}\ \frac{\vdash_E m\ \texttt{nat}}{\vdash_E (m+2)\ \texttt{nat}}$$

This system captures the the insight that 0 is even, 1 is odd, and whenever m is even or odd, so is $m + 2$.

1.4 From proof system to algebraic specification

A proof system is a starting point for designing recursive functions. Design begins by looking at the *forms* of natural numbers as they appear in the conclusions of the

[3] Using the Decimal system would have the same effect as writing every numeral with a leading zero.

Proof system	*Left-hand side*	
Peano	$(f\ 0)$	$= \cdots$
	$(f\ (m+1))$	$= \cdots$
Binary	$(f\ 0)$	$= \cdots$
	$(f\ (2 \times m + b))$	$= \cdots$
Decimal (arithmetic)	$(f\ 0)$	$= \cdots$
	$(f\ (10 \times m + d))$	$= \cdots$
Decimal (numeration)	$(f\ d)$	$= \cdots$
	$(f\ (10 \times m + d))$	$= \cdots$
Parity	$(f\ 0)$	$= \cdots$
	$(f\ 1)$	$= \cdots$
	$(f\ (m+2))$	$= \cdots$

Table 1.1: Forms of laws for a one-argument function f

rules. For example, the Peano system has natural numbers of the forms "0" and "$(m+1)$." The binary system has natural numbers of the forms "0" and "$(2 \times m + b)$."

Once you know the forms, the next step in designing a function is to specify what the function is supposed to do for each form. The specification is written as a set of equations called *algebraic laws.* These laws are stylized: there is typically one law for each form of each input, and the left-hand side of the law applies the function to that form.

Writing left-hand sides is mechanistic: a left-hand side is determined by the name of the function being defined, the number of arguments it expects, and the form of each argument. As examples, forms of laws for all one-argument natural-number functions appear in Table 1.1.

The laws in Table 1.1 are missing their right-hand sides. Right-hand sides require thought: a right-hand side specifies what a function must do in one particular case. When writing a right-hand side, you can get a valuable hint from the underlying proof system:

1. Look at the proof rule whose *conclusion* has the form of input used on the law's left-hand side.

2. If the rule has no premises, you have a base case. The right-hand side should not make any recursive calls.

3. If the rule has premises, each premise *with the same form of judgment* represents a potential recursive call. A premise with a *different* form of judgment may represent a call to a helper function.

If there is more then one input, you may have to consider more than one proof rule. Some examples appear below.

Example: is a number even?

To design a function `even?`, which tells if a natural number is even, we can reasonably use the Peano, binary, or parity system. The Peano system has two rules, PEANOZERO and SUCCESSOR. In the zero case, there is no premise above the line, and I ought to be able to tell whether zero is even without making a recursive call. In the successor case, there *is* a judgment $\vdash_P m$ `nat` above the line, and I should consider a recursive call `(even?` m`)`. With these considerations in mind, I propose these laws:

$$\texttt{(even?}\ 0\texttt{)} = \textbf{true}$$
$$\texttt{(even?}\ (m+1)\texttt{)} = \neg\texttt{(even?}\ m\texttt{)}$$

where $\neg$ is the "logical not" operator.

Using the binary-numeral system, I'm pedantic enough to want a helper function `even-bit?`, which is based on the proof system for the judgment form "$\vdash b$ `bit`":

$$\texttt{(even?}\ 0\texttt{)} = \textbf{true}$$
$$\texttt{(even?}\ (2 \times m + b)\texttt{)} = \texttt{(even-bit?}\ b\texttt{)}$$
$$\texttt{(even-bit?}\ 0\texttt{)} = \textbf{true}$$
$$\texttt{(even-bit?}\ 1\texttt{)} = \textbf{false}$$

This is the system you would be using if you were coding `even?` "in the normal way." To see why, answer this question: if $n = 2 \times m + b$, how do you get b from n?[4]

Here are the laws for `even?` using the parity system:

$$\texttt{(even?}\ 0\texttt{)} = \textbf{true}$$
$$\texttt{(even?}\ 1\texttt{)} = \textbf{false}$$
$$\texttt{(even?}\ (m+2)\texttt{)} = \texttt{(even?}\ m\texttt{)}$$

I wouldn't want to implement `even?` using either of the decimal proof systems. These systems don't really fit a computation of `even?`, and no sane person would try to use them—there's no point in coding `even?` on a digit d when you could more easily test n directly.

Example: Multiplication

The decimal proof systems are useless for parity, but for problems like multiplication, they work well. Since multiplication is a two-argument function, here is the general form of laws for a two-argument function g, using the forms of input from the decimal-arithmetic proof system:

$$(g\ 0\ 0) = \cdots$$
$$(g\ (10 \times m + d)\ 0) = \cdots$$
$$(g\ 0\ (10 \times m' + d')) = \cdots$$
$$(g\ (10 \times m + d)\ (10 \times m' + d')) = \cdots$$

[4] $b = n \bmod 2$.

Proof system	*Form of n*	*Test for form*	*Parts n is formed from*	
Peano	0	$n = 0$		
	$(m + 1)$	$n \neq 0$	$m = n - 1$	
Binary	0	$n = 0$		
	$(2 \times m + b)$	$n \neq 0$	$m = n \operatorname{div} 2$	$b = n \bmod 2$
Decimal	0	$n = 0$		
(arithmetic)	$(10 \times m + b)$	$n \neq 0$	$m = n \operatorname{div} 10$	$b = n \bmod 10$
Decimal	d	$n < 10$	$d \ = n$	
(numerals)	$(10 \times m + b)$	$n \geq 10$	$m = n \operatorname{div} 10$	$b = n \bmod 10$
Parity	0	$n = 0$		
	1	$n = 1$		
	$(m + 2)$	$n \neq 0 \land n \neq 1$	$m = n - 2$	

Table 1.2: Identifying the form of n and extracting its parts

The laws for multiplication are

$$
\begin{aligned}
(\texttt{*}\ 0\ 0) &= 0 \\
(\texttt{*}\ (10 \times m + d)\ 0) &= 0 \\
(\texttt{*}\ 0\ (10 \times m' + d')) &= 0 \\
(\texttt{*}\ (10 \times m + d)\ (10 \times m' + d')) &= \\
100 \times m \times m' + 10 \times (m \times d' + m' \times d) + d \times d'
\end{aligned}
$$

Example: ``All fours''

Suppose I want to write a function `all-fours?`, which tells me if an argument's decimal representation uses only the digit 4. That is, it likes arguments 4, 44, 444, and so on. I don't want the DECIMAL system, since 0 is the wrong base case. Instead, I want DECIMALNAT:

$$
\begin{aligned}
(\texttt{all-fours?}\ d) &= (d = 4) \\
(\texttt{all-fours?}\ (10 \times m + d)) &= (\texttt{all-fours?}\ m) \land d = 4,
\end{aligned}
$$

where $\land$ is the "logical and" symbol.

1.5 From algebraic laws to recursive function

When the algebraic laws are complete, we write the code. Conceptually, the code emerges in response to three questions:

1. We ask of each input, *how were you formed?* (Example answer: $10 \times m + d$.)

2. Once we know the form of an input, we proceed to ask *from what parts were you formed?* (Example answer: from m and d, where $m = n \operatorname{div} 10$ and $d = n \bmod 10$.)

3. Finally, when we know how each input was formed and from what parts, we can ask *which algebraic law applies* and *what does the law say we are supposed to do with the parts?* (Example answer: make a recursive call on m and check if $d = 4$.)

The first two questions can be answered using the tests and equations shown in Table 1.2. The third question is answered by selecting the algebraic law whose left-hand side has the right form, then using the right-hand side.

Detailed example

As a first example, let's implement function `is_even`, in C, using the parity system. Here are the laws:

$$
\begin{aligned}
(\texttt{is\_even}\ 0) &= \textbf{true} \\
(\texttt{is\_even}\ 1) &= \textbf{false} \\
(\texttt{is\_even}\ (m + 2)) &= (\texttt{is\_even}\ m)
\end{aligned}
$$

Table 1.2 shows that we can distinguish the forms of `n` using tests for 0 and 1, so the first draft of our function uses `if` statements to distinguish three cases: one for each law.

```
bool is_even (unsigned n) {
  if (n == 0) {
    ...
  } else if (n == 1) {
    ...
  } else {
    ...
  }
}
```

In the first two cases, n isn't formed from any other parts, and we can knock off the cases by filling in the right-hand sides of the laws:

```
bool is_even (unsigned n) {
  if (n == 0) {
    return true;
  } else if (n == 1) {
    return false;
  } else {
    ...
  }
}
```

In the final case, the law mentions m, which is computed as $n - 2$:

```
bool is_even (unsigned n) {
  if (n == 0) {
    return true;
  } else if (n == 1) {
    return false;
  } else {
    unsigned m = n - 2;
    ...
  }
}
```

With `m` computed, we use the right-hand side of the law to write a recursive call:

```
bool is_even (unsigned n) {
  if (n == 0) {
    return true;
  } else if (n == 1) {
    return false;
  } else {
    unsigned m = n - 2;
    return is_even(m);
  }
}
```

In practice, I probably would not bother with `m` in the third case, writing instead `is_even(n-2)`.

Condensed example

As another example, suppose I want to design a function that sums the natural numbers from 0 to n. I choose the Peano proof system, and I write these laws:

$$(\texttt{sumto}\ 0) = 0$$
$$(\texttt{sumto}\ (m+1)) = (\texttt{sumto}\ m) + (m+1)$$

I distinguish case $n = 0$ from case $n = (m+1)$ by testing $n = 0$, and when $n = (m+1)$, I get the “part” m by computing $m = n - 1$:

```
int sumto(unsigned n) { // not tested
  if (n == 0} {
    return 0;
  } else {
    return sumto(n - 1) + n;
  }
}
```

In this code, I don’t bother with an explicit m.

1.6 Complete process examples

The preceding sections of this handout look at proof systems, forms of data, and algebraic laws, which are the technical core of our recommended software process. This section works two more examples, showing all 9 steps of the complete process.

Design of a factorial function

1. *Understand the forms of data.* To design a factorial function, I choose the Peano system, with forms 0 and $(m + 1)$. Choosing the right system is not always obvious, but I’ve written factorial functions before, and I know the Peano system will work.

2. *Write a sample input for each form.* My example input of form 0 is 0, and my example input of form $(m+1)$ is 4.

3. *Choose a name.* I choose the name `factorial`. This name is a noun that describes what the function returns.

4. *Write the contract.* The contract is trivial, pedantic, and boring. It says

   ```
   ;; (factorial n) returns n factorial,
   ;; sometimes written "n!", where n
   ;; is a natural number
   ```

   In practice, I would write this function without a contract—the name alone is contract enough.

5. *Write examples.* My examples:

   ```
   (check-expect (factorial 0) 1)
   (check-expect (factorial 4) 24)
   ```

6. *Generalize to algebraic laws.* The zero case is already an algebraic law:

   ```
   (factorial 0) == 1
   ```

   For the successor case, the Peano proof system has judgment $\vdash m$ `nat` above the line, so I’ll be looking to make a recursive call with m. I write

```
(factorial (+ m 1))
  == 24            ; assuming m = 3
  == (* 4 6)       ; assuming m = 3
  == (* 4 (factorial m)) ; m = 3, recurs
  == (* (+ m 1) (factorial m)) ; general
```

7. *Code the case analysis.* I've got two laws with distinct left-hand sides, so two cases. Table 1.2 on page 6 recommends testing for $n = 0$. I get

   ```
   (define factorial (n)
     (if (= n 0)
         ...    ; zero case
         ...))  ; successor case
   ```

   This code will compile but not run.

8. *Code the results.* To finish the function, I plug in the right-hand side for each case. I don't fuss with m—instead I just write n for $(m + 1)$ and $n - 1$ for m:

   ```
   (define factorial (n)
     (if (= n 0)
         1
         (* n (factorial (- n 1)))))
   ```

9. *Revisit unit tests.* My tests cover every form of input, and there are no Booleans in play. I add them to my source file and run them:

   ```
   $ impcore -q < fact.imp
   factorial
   Both tests passed.
   ```

The complete solution in file fact.imp looks like this:

```
;; (factorial n) returns n factorial,
;; sometimes written "n!", where n
;; is a natural number
(define factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))

        (check-expect (factorial 0) 1)
        (check-expect (factorial 4) 24)
```

(Unit tests are indented eight spaces.)

Design of a power function

Same drill, but now I define a function of two arguments, x and n, to compute x^n. And I do something sophisticated: I know that this computation depends only on the form of n, not on the form of x. So have only one form to analyze, and I get by with just two cases instead of four or more.

1. *Understand the forms of data.* Again, I choose the Peano system, with forms 0 and $(m + 1)$.

2. *Write a sample input for each form.* Again, I choose examples 0 and 4.

3. *Choose a name.* I choose power, a noun that describes what the function returns.

4. *Write the contract.* This contract is *not* trivial: we need to know which argument is the base and which is the exponent.

   ```
   ;; (power x n) returns x raised to the
   ;; nth power, where n is a natural number
   ```

5. *Write examples.* My examples:

   ```
   (check-expect (power 3 0) 1)
   (check-expect (power 3 4) 81)
   ```

6. *Generalize to algebraic laws.* In math form,

   $$x^0 = 1$$
   $$x^{(m+1)} = x^m \times x$$

   When you're designing, math form is always legitimate and often helpful.

   In code form,

   ```
   (power x 0) == 1
   (power x (+ m 1)) == (* (power x m) x)
   ```

7. *Code the case analysis.* It's the same proof system, the same form of input, and therefore the same case analysis as for factorial:

   ```
   (define power (x n)
     (if (= n 0)
         ...    ; zero case
         ...))  ; successor case
   ```

8. *Code the results.* Again, instead of m, I write $n - 1$:

   ```
   (define power (x n)
     (if (= n 0)
         1
         (* (power x (- n 1)) x)))
   ```

9. *Revisit unit tests.* My tests cover every form of input, and there are no Booleans in play. And they pass.

Here's the complete solution in file power.imp:

```
;; (power x n) returns x raised to the
;; nth power, where n is a natural number
(define power (x n)
  (if (= n 0)
      1
      (* (power x (- n 1)) x)))

        (check-expect (power 3 0) 1)
        (check-expect (power 3 4) 81)
```

1.7 Mistakes to avoid in algebraic laws

An algebraic law sits halfway between a mathematical statement and a function definition. When you are writing a law for function f, here are some mistakes to avoid:

- A left-hand side is not an application of f.
- On a left-hand side, f is applied to something that is not a form of data. For example, m/b or `(/ m b)`.
- There is a permissible form of data that does not match the left-hand side of any law.
- A law's right-hand side mentions a variable that does not appear on the left-hand side. (This mistake is just like trying to use an undeclared variable in a C++ function.)

In addition to these basic mistakes, people who are first learning to code with algebraic laws often make three other, more subtle mistakes. The more subtle mistakes reflect confusion about what a variable in a law stands for: an actual parameter or a *part* of an actual parameter? When a variable stands for a part of a parameter, trouble sometimes follows.

In the first subtle mistake, the right-hand side of a law uses a variable that is intended to stand for a formal parameter, but the parameter doesn't actually appear on the left. For example,

```
(has-digit? (10 * m + d) d2) ==
  (has-digit? (/ n 10) d2), where d2 != d
```

The `n` on the right-hand side is not specified—it could be anything. This mistake is a special case of the fourth bullet above, and I can see what's going on: the left-hand side specifies the *parts* of the first parameter, but the right-hand side uses `n` to name the parameter itself. What's meant by `(/ n 10)` is actually `m`.

To avoid this mistake, remember this rule: *The right-hand side of an algebraic law may use any variable that appears on the left-hand side, and only those variables.*

In the second subtle mistake, a right-hand side misuses a variable from the left as if it were the argument, rather than a part of the argument. Here's an example for function `(power x n)`:

```
(power x (+ m 1)) == (* (power x (- m 1)) x)
```

The variable `m` is *already* meant to be one less than the argument: `m` = `n` − 1. The right-hand side of the law incorrectly applies to `m` the operation that is meant to be applied to `n`.

In the third subtle mistake, a name like `m` is used in the algebraic laws to stand for a part of an argument, but in the code to stand for the entire argument. Here's an example:

```
;; (has-digit? (+ (* 10 m) d) d)  == 1
;; (has-digit? (+ (* 10 m) d) d2) == ...
;; ...

(define has-digit? (m d2)
    ...)  ;; cases 1 & 2:
          ;;   m == (+ (* 10 m) d)???
```

Each part is technically correct by itself, but mixing the two is just too confusing: the argument can't be m and $10 \times m + d$. To avoid this mistake, make sure each name stands consistently either for an argument or for a part of an argument, but not both.

2. Scheme values and more algebraic laws

Now we remove the training wheels and start working with more structured data. This lesson presents the bare bones of programming with μScheme values, including lists and S-expressions, using proof systems and algebraic laws. We apply the same design techniques as before, but with new forms of data.

This lesson also introduces a wider world of algebraic laws, distinguishing *algorithmic* laws from non-algorithmic *properties.* When coding from scratch, you must learn to make your laws algorithmic.

2.1 Describing μScheme data

The first section of this lesson revisits the ideas in the natural-number lesson, but for some common forms of μScheme data.

Proof systems for μScheme data

As noted in Figure 2.1 on page 99 of *Programming Languages: Build, Prove, and Compare*, a μScheme value is either an atom, a function, or a `cons` cell. A "fully general S-expression" is any of these except a function. We could write a proof system like this:

$$\frac{\vdash v\ \texttt{symbol}}{\vdash v\ \texttt{gsx}} \qquad \frac{\vdash n\ \texttt{number}}{\vdash n\ \texttt{gsx}} \qquad \frac{}{\vdash \texttt{\#t gsx}}$$

$$\frac{}{\vdash \texttt{\#f gsx}} \qquad \frac{}{\vdash \texttt{'() gsx}} \qquad \frac{\vdash v_1\ \texttt{gsx} \qquad \vdash v_2\ \texttt{gsx}}{\vdash \texttt{(cons}\ v_1\ \ v_2\texttt{)}\ \texttt{gsx}}$$

We could define "list of A's" using the proof system from section 2.5, which starts on page 119 of the textbook:

$$\textsc{EmptyList}\ \frac{}{\texttt{'()} \in \mathit{LIST}(A)}$$

$$\textsc{ConsList}\ \frac{a \in A \qquad \mathit{as} \in \mathit{LIST}(A)}{\texttt{(cons}\ a\ \mathit{as}\texttt{)} \in \mathit{LIST}(A)}$$

On your homework I'll ask you to define "*nonempty* list of A's."

We could define "ordinary" S-expression using just ideas 1 and 2 from Figure 2.1 on page 99. The notation of that last rule gets a little dodgy:

$$\frac{\vdash n\ \texttt{number}}{\vdash n\ \texttt{osx}} \qquad \frac{}{\vdash \texttt{\#t osx}} \qquad \frac{}{\vdash \texttt{\#f osx}}$$

$$\frac{\mathit{vs} \in \mathit{LIST}(\texttt{osx})}{\vdash \mathit{vs}\ \texttt{osx}}$$

Writing $\mathit{LIST}(\texttt{osx})$ is flagrant abuse of notation. There's a better way.

An informal alternative

Proof systems are great for describing the structure of natural numbers, as well as more complex structures like computations. But for describing simpler data structures, we don't need the expressive power of proof systems, and it's often difficult to come up with good judgment forms—that's where we got into trouble above. As an alternative, we can write an inductive definition informally. We name the set we're trying to define, and we list all the ways that data in the set could be formed. Examples follows.

A *fully general S-expression* is one of the following:

- A symbol
- A number
- A Boolean
- The empty list `'()`
- `(cons` v_1 v_2`)`, where v_1 and v_2 are fully general S-expressions

A *list of A's* is one of the following:

- The empty list `'()`
- `(cons` a as`)`, where a is an A and as is a list of A's

An *ordinary S-expression* is one of the following:

- A symbol
- A number
- A Boolean
- A list of ordinary S-expressions

It is frequently useful to expand that last bullet. It is equally true that an ordinary S-expression is one of the following:

- A symbol
- A number
- A Boolean
- The empty list `'()`
- `(cons` v vs`)`, where v is an ordinary S-expression and vs is a list of ordinary S-expressions

Data	*Left-hand side*	
Fully general S-expression	(*f* *a*)	= ⋯, where *a* is an atom
	(*f* (cons *y* *z*))	= ⋯
List of *A*	(*f* '())	= ⋯
	(*f* (cons *y* *ys*))	= ⋯
Ordinary S-expression	(*f* *a*)	= ⋯, where *a* is an atom
	(*f* (cons *y* *ys*))	= ⋯
Also Ordinary S-expression	(*f* '())	= ⋯
	(*f* *a*)	= ⋯, where *a* is an atom but not '()
	(*f* (cons *y* *ys*))	= ⋯
Nonempty list of *A*	(homework) ⋯	

Table 2.1: Forms of laws for a one-argument function *f*

Data	*Form of argument x or xs*	*Test for form*	*Parts argument is formed from*	
Fully general S-expression	*a*	(atom? *x*)	*a* = *x*	
	(cons *y* *z*)	(not (atom? *x*)) or (pair? *x*)	*y* = (car *x*)	*z* = (cdr *x*)
List of *A*	'()	(null? *xs*)		
	(cons *y* *ys*)	(not (null? *xs*))	*y* = (car *xs*)	*ys* = (cdr *xs*)
Ordinary S-expression	*a*	(atom? *x*)	*a* = *x*	
	(cons *y* *ys*)	(not (atom? *x*)) or (pair? *x*)	*y* = (car *x*)	*ys* = (cdr *x*)
Also Ordinary S-expression	'()	(null? *x*)		
	a	(atom? *x*)	*a* = *x*	
	(cons *y* *ys*)	(not (atom? *x*))	*y* = (car *x*)	*ys* = (cdr *x*)
Nonempty list of *A*	⋯ (homework) ⋯			

Table 2.2: Identifying forms and extracting parts

2.2 Laws for μScheme data

As with natural numbers, the forms of data determine the left-hand sides of algebraic laws, which determine the case analysis that goes into your code. Table 2.1 shows what laws will look like for a one-argument function *f*. Table 2.2 shows how to identify forms of data and how to extract the parts from which data is formed.

2.3 More uses of algebraic laws

Our first homework assignment introduced you to algebraic laws purely as a tool for designing functions that you code from scratch. The tool works even better for lists and S-expressions than it works for natural numbers. For example, here are laws that define a function for asking how many elements there are in a list:

```
(length '())          == 0
(length (cons x xs)) == (+ 1 (length xs))
```

A set of laws like this is called *algorithmic*: the laws specify the algorithm for `length`, and they are very close to an implementation.

More generally, we can write algebraic laws for any property that we believe is true. For example, if we append two lists, the length of the result is the sum of the lengths of the arguments:

```
(length (append xs ys)) ==
        (+ (length xs) (length ys))
```

This law is *not* algorithmic—a law like this is called a *property*. Let's explore the distinction.

Understanding and using algorithmic laws

To identify a set of laws as algorithmic, learn to recognize these hallmarks:

- Each left-hand side is a function to be defined, applied to one or more arguments, where each argument is either a *variable* or a *form of data*. In the `length` example, both `'()` and `(cons x xs)` are forms of data.

- In an algorithmic set of laws, each law is *mutually exclusive* with the others. That is, given any particular input, at most one law applies. Mutual exclusion is accomplished either using mutually exclusive forms of data, like `'()` and `(cons x xs)`, or by using mutually exclusive side conditions.

  (In rare cases, algorithmic laws can *overlap*: there are inputs for which more than one law could apply. In such cases, *all applicable right-hand sides must produce the same result*. These cases are sufficiently rare that I don't present an example.)

- Collectively, an algorithmic set of laws *accounts for every input that is permitted by a function's contract*. If an input is permissible, there must be a law that applies.

- In every recursive call on every right-hand side, *some input is getting smaller*.

Algorithmic laws are used for these purposes:

- Algorithmic laws are used primarily *to design and implement functions*.

- Algorithmic laws can also be used *to test functions*.

Understanding and using properties

Technically, every law in an algorithmic set is also a property. But not every property is an algorithmic law. To identify a property law as non-algorithmic, learn to recognize these hallmarks:

- On a left-hand side, *a function is applied to the result of another function*. For example, in the `length` property, `length` is applied to the result of `append`.

- Properties might not be mutually exclusive, and they needn't account for every permissible input.

Properties have many more uses than algebraic laws, including these purposes:

- Properties are used for *testing*. Substitute a permissible value for each variable in the property, and check that equality holds. For example, here's a property we use to test arithmetic in Smalltalk:

  ```
  (* 2 n) == (+ n n)
  ```

  We can test this property with any natural number `n`.

  Here's a property about lists that is useful only for testing:

  ```
  (permutation? (cons x (cons y zs))
                (cons y (cons x zs))) == #t
  ```

  Good tooling for programming languages frequently includes *random, automated, property-based testing* based on substituting randomly generated values for variables in properties.

- Properties are used for *refactoring*, which means rewriting code to improve its structure, without changing its semantics. A good example is code simplification. Many of the properties found in section 2.4 of the textbook, like this `append-cons` law, can be used to simplify code:

  ```
  (append (cons x '()) xs) == (cons x xs)
  ```

- Properties are used for *code improvement*, which means rewriting code to improve its performance, without changing its semantics. (Code improvement is often called "optimization.") Some of the properties found in section 2.4 of the textbook, like this `append-append` law, can be used to improve performance:

  ```
  (append (append xs ys) zs) ==
                (append xs (append ys zs))
  ```

- Properties are used for *specification*, especially of abstract data types. Programmers may use properties to say how an abstraction behaves without saying how it is implemented. Here's a typical property from an abstraction of sets:

  ```
  (member? x (add-element x xs)) == #t
  ```

  The property says that if we add an element `x` to any set, then `x` is a member of the resulting set.

2.4 Common issues using algebraic laws with Scheme

Below are some issues you might run into when writing algebraic laws for Scheme functions.

Correct use of variables

A common mistake is to write laws thinking that variables are mutually exclusive with other forms of data. They aren't. When you write a variable, you are saying implicitly, "this could be *any* form of data, and I don't care which." In other words, when you write a variable in an argument position, you are promising not to look and see how the argument was formed. In particular, when you write a variable, you are promising never to apply `null?`, `car`, or `cdr` to that variable.

Here's an example of this common mistake:

```
(sublist? xs '()) == #f   ;; WRONG
(sublist? '() ys) == #t
... more cases below ...
```

The student who wrote these laws meant for `xs` and `ys` meant to be nonempty. But a variable could be any list, including the empty list. In this example, if both `xs` and `ys` are empty, the laws give inconsistent results. That's how we're certain that something is wrong. Here's a correct version, in which every argument is either explicitly empty or explicitly nonempty.

```
(sublist? (cons w ws) '())          == #f ;; RIGHT
(sublist? '()         (cons z zs)) == #t
... more cases below ...
```

These left-hand sides can't possibly be confused.

This version can be refined by observing that in the original set, the problem lies with the first law. The law `(sublist? '() ys) == #t` is actually good: the empty list is a sublist of any list `ys`, whether `ys` is empty or not. So we could write the laws this way:

```
(sublist? (cons w ws) '()) == #f ;; RIGHT
(sublist? '()         ys)  == #t ;; SPLENDID
... more cases below ...
```

The advantage of this final specification is that we might then have to consider fewer alternatives in the "more cases below."

Breaking S-expression inputs down by cases

Quite often it's useful to define an ordinary S-expression as one of the following:

- The empty list
- `(cons z zs)`, where `z` is an S-expression and `zs` is a list of S-expressions
- `a`, where `a` is an atom but not the empty list

A *common mistake* here is to forget the side condition on `a`. Here are some mistaken laws for counting the number of atoms in an ordinary S-expression:

```
(atom-count '())         = 0
(atom-count (cons z zs)) =
            (+ (atom-count z) (atom-count zs))
(atom-count a)           = 1     ;; WRONG
```

The last law needs a side condition:

```
(atom-count a)           = 1,
     where a is a non-null atom  ;; RIGHT
```

You can't break a function down by cases

Some of the problems on the homework, like `takewhile`, `dropwhile`, and `arg-max`, take functions as inputs. You can't break a function down by cases, because there's no way to ask a function how it was formed. All you can do with a function is apply it. How, then, should a function appear in an algebraic law? As a variable. Here's an example for `takewhile`, which takes two arguments, a predicate `p?` and a list of values. A function has one case and a list has two, and multiplied together there are two in total:

```
(takewhile p? '())         = ...
(takewhile p? (cons x xs)) = ...
```

That's not the end of the story, however: once we have both `p?` and an `x` that we could apply `p?` to, we could have extra cases depending on whether `(p? x)` is true or false. Those cases would be written as side conditions.

One final example: function `arg-max` takes a function and a *nonempty* list of values. The laws for `arg-max` will have one case for the function input (just a variable), and other cases for the nonempty list. (Finding the forms of a nonempty list is a homework problem.)

3. Higher-order functions

In addition to "constructed data" (`cons`), Scheme also has first-class, nested functions. This key feature, which originated with Scheme, is now used prominently in such scripting languages as JavaScript, Python, Lua, and Perl. Functions are not quite like other forms of data, but they still participate in the design process. Functions that appear as arguments are handled somewhat like other forms of data; functions that appear as results are different. The details are explained in this lesson.

3.1 Designing with functions as arguments

In a principled design process, here's how we work with function arguments:

1. *Forms of data.* A function is unlike any other form of data so far, in this way: you can't interrogate a function to ask how it was made or from what parts. Instead,

   > *All you can do with a function is apply it.*

   However, we can and do use information about a function's *contract* as a proxy for its *form.* Step 1 of the design process is therefore to identify what is important about the function's contract. Usually what's most important involves the number of parameters and sometimes the types of the parameters and result. Here are some ways to think about the form of a function:

   - Takes one argument
   - Takes two arguments
   - Takes one argument, returns a Boolean
   - Takes two arguments, returns a Boolean
   - Takes one argument, returns a function
   - Takes two arguments, returns a result that is like the second argument

2. *Example inputs.* Once you've identified the form of a function, you can come up with examples. The best examples are familiar functions, like those from the initial basis.

   ```
   abs      ; one arg
   +        ; two args
   symbol?  ; one arg, returns Boolean
   <        ; two args, returns Boolean
   curry    ; one arg, returns function
   cons     ; two args, result like 2nd
   ```

3, 4, 5. When functions are used as arguments, steps 3 to 5 are unchanged.

6. *Algebraic laws.* Forms of algebraic laws are as before: on the left-hand side, the function being defined is applied to arguments. What's different is *there can be no case analysis on an argument that is a function.* The only thing you can do with a function is apply it. Case analysis on other arguments proceeds as usual.

7. *Code case analysis.* Because we can't do case analysis on a function, the presence of a function as an argument doesn't change the way we code case analysis. Case analysis on non-function arguments proceeds as usual.

8, 9. When functions are used as arguments, steps 8 and 9 are unchanged.

3.2 Designing with functions as results

When a function returns another function as a result, the design process is affected more broadly. That's because we can't do much with a function result. In particular, we can't compare a function result with another function result—all we can do is apply it. If we want to test or specify a function result, we have to test or specify what happens when we apply the function to something. The central principle is this:

> *Two functions are equal if and only if when they are applied to equal arguments, they return equal results.*

Here's how this principle affects our design process.

1, 2. We're talking about results, not inputs, so the story about forms of data and inputs is unchanged.

3, 4. Naming and contracts are unchanged.

5. *Example results.* Example results may or may not be helpful here—it depends whether the expected result has a well-known name. Here's a case where example results are helpful: function `flip` takes a two-argument function as argument and returns a function that is like the argument function, except

the result function expects its arguments in the opposite order. A couple of good example results are

```
(flip <)  ==  >
(flip >=) ==  <=
```

With these examples, we got lucky. More commonly, the result doesn't have a name. For example, we don't have a name for `(flip append)`.

While you can *sometimes* find examples to be equal to a function result, you can *always* construct examples about what a function result is applied to. This design technique is reliable, and it might look like this:

```
((flip <)  3 4)  ==  #f
((flip <=) 3 4)  ==  #f
((flip append) '(a b c) '(1 2 3)) ==
   '(1 2 3 a b c)
```

This general form of example has three parts:

a) The function being designed, `flip`, is applied to arguments, producing a result.
b) The result is *itself* applied to (more) arguments.
c) The result of the result (the whole application) is equal to something.

6. *Algebraic laws.* Right-hand sides of algebraic laws don't change, but left-hand sides do. When a function returns a function, the left-hand side is now going to include at least *two* applications, just like the example results in step 5.

   a) The function being designed is applied to variables and/or forms of data.
   b) The *result* of the function being designed is then applied to *more* variables and/or forms of data.

   As examples, here are the laws for every predefined μScheme function that returns a function as a result:

   ```
   ((o f g) x)       == (f (g x))
   ((uncurry f) x y) == ((f x) y)
   ```

   If the result of a result is also a function, we keep applying until we get to a point where we have results that can be tested for equality:

   ```
   (((curry f) x) y) == (f x y)
      ;; *three* applications on left!
   ```

7. *Code left-hand side.* When a function returns a function, this step changes a bit. The key change is that we code *one function for every application on the left-hand side.* The outermost function can be coded using either `define` or `lambda`. Inner functions are coded using `lambda`.

   One possibly confusing point:

   > The *innermost* application in the law corresponds to the *outermost* `define` or `lambda` in the definition.

   That's because in the law, the innermost application is evaluated first, but in the definition, the outermost `lambda` is evaluated first. Let's see how it works.

   In the law for `((o f g) x)`, we have two nested applications:

   $$\texttt{(o f g)} = \textit{result}$$
   $$(\textit{result}\ \texttt{x}) = \texttt{(f (g x))}$$

   That *result* is going to become an anonymous `lambda`.

   First step in the code: innermost application from the left-hand side:

   ```
   (define o (f g)
      ; law: (result x) == (f (g x))
      ... result function ...)
   ```

   Next step of the left-hand side: result function expects `x` as an argument:

   ```
   (define o (f g)
      ; law: (result x) == (f (g x))
     (lambda (x) ... right-hand side ...))
   ```

   And leaping ahead to step 8:

   ```
   (define o (f g)
      ; law: (result x) == (f (g x))
     (lambda (x) (f (g x))))
   ```

   Here's the same development for `uncurry`:

   ```
   (define uncurry (f)
     ; law: (result x y) = ((f x) y)
     ... result function ...)
   ```

   And finishing the left-hand side with `lambda`:

   ```
   (define uncurry (f)
     ; law: (result x y) = ((f x) y)
     (lambda (x y) ... RHS ...))
   ```

And leaping ahead to step 8:

```
(define uncurry (f)
  ; law: (result x y) = ((f x) y)
  (lambda (x y) ((f x) y)))
```

8. *Code results.* The code for a final result on a right-hand side is written in the same way as usual, but we'll find it inside at least one additional `lambda`.

9. In order to test a function's result, we have to apply it to something. Here is one bad example accompanied by three good ones:

```
(check-expect (flip <) >) ;; USELESS
(check-expect ((flip <) 3 4) (> 3 4)) ;; GOOD
(check-expect ((flip <) 3 3) (> 3 3)) ;; GOOD
(check-expect ((flip <) 3 2) (> 3 2)) ;; GOOD
```

4. ML types and pattern matching

This lesson sketches how to apply our design process to ML, a language with types and pattern matching. Popular languages with these features include Haskell, Elm, OCaml, Reason, F♯, Erlang, and Scala. More obscure languages include Agda, Idris, and Coq/Gallina.

This lesson is followed by a page of syntax help.

4.1 Design steps

Our design method is affected by the introductions of *constructed data* and *types*.

1. *Forms of data* for numbers and functions are as in Scheme. But forms of data for lists, pairs, tuples, trees, and other constructed data are determined by primitive types and user-defined types, including algebraic datatypes. These forms are shown in Table 4.1 on the next page, as *patterns*.

   Patterns are the technical name for the phrases that appear as function arguments on the left-hand sides of algebraic laws—so you already know how to program with them. But to define them carefully, here are ML's rules for patterns:

   - Any variable, as in `x`, is a pattern.
   - The "wildcard," as in `_` (underscore), is a pattern
   - A sequence of patterns separated by commas and wrapped in round brackets, as in `(l, x, r)`, is a *tuple* pattern.
   - The *empty tuple* `()` is a pattern.
   - A sequence of patterns separated by commas and wrapped in square brackets, as in `[x1, x2, x3]`, is a *list* pattern.
   - The *empty list* `[]` is a pattern.
   - A value constructor by itself, as in `nil` or `NONE`, is a pattern.
   - A *value constructor* applied to a pattern, as in `SOME x`, is a pattern.
   - An *infix* value constructor placed between patterns, as in `x :: xs`, is a pattern.[1]
   - A sequence of pattern bindings separated by commas and wrapped in curly brackets, as in `{ ps1 = s, ps2 = s' }`, is a *record* pattern.
   - A literal number, as in `1852`, is a pattern.
   - A literal string, as in `"frogs"`, is a pattern.
   - A literal character, as in `#"f"`, is a pattern.

   A key feature of ML is that you get to define new forms of data, using the `datatype` definition. For example, a binary tree of machine integers:

   ```
   datatype inttree
     = ILEAF
     | INODE of inttree * int * inttree
   ```

2. *Example inputs* include what you would expect from Scheme: numbers written using numeric literals, and anonymous lambda functions written using `fn`, as in `(fn (x, y) => y + 1)`. ML also has string literals.

   In addition, examples of constructed data are formed using the pattern rules: if a pattern has no variables or wildcards, it specifies a value:

   ```
   (105, "hello")
   [2, 3, 5, 7, 11]
   SOME 33
   ```

3. *Function names* are limited. In ML, you may use *either* "symbolic" characters like `<`, `?`, `+`, and so on, *or* you may use alphanumeric characters[2] with an underscore, but you may not use both in the same name. Symbolic characters may be combined into arbitrarily long names, such as `<=>` or `/*/`.

   ML offers a design choice not available in Scheme: function *names* can be "infix." Predefined functions like `mod`, `o`, and `+` all have infix names, as does the value constructor `::`. The fixity of names can be changed by an `infix` or `nonfix` definition form. It's especially common for symbolic names to be made infix.

   Infix names like `::` and `+` can't be used as first-class values; when you write an infix name, ML thinks you mean to apply it. But there is a workaround: putting the reserved word `op` in front of any infix

[1] Confusingly, "fixity" is a local property of a name, not a property of the value constructor itself.

[2] The ASCII quote mark, here pronounced "prime," counts as alphanumeric, as in `x'`, pronounced "x-prime."

Type of e	*Patterns*	*Test in definition*	*Test in expression*	*Types of parts*
'a list	[] x :: xs	fun f [] = ··· | f (x :: xs) = ···	case *e* of [] => ··· | x :: xs => ···	x : 'a, xs : 'a list
'a option	NONE SOME x	fun f NONE = ··· | f (SOME x) = ···	case *e* of NONE => ··· | SOME x => ···	x : 'a
order	LESS EQUAL GREATER	fun f LESS = ··· | f EQUAL = ··· | f GREATER = ···	case *e* of LESS => ··· of EQUAL => ··· of GREATER => ···	
int	0 n	fun f 0 = ··· | f n = ···	case *e* of 0 => ··· | n => ···	n : int
'a * 'b	(x, y)	fun f (x, y) = ···	let val (x, y) = *e* in ··· end	x : 'a, y : 'b
'a * 'b * 'c	(x, y, z)	fun f (x, y, z) = ···	let val (x, y, z) = *e* in ··· end	x : 'a, y : 'b, z : 'c
{ f1 : 'a , f2 : 'b , f3 : 'c ··· } (record)	{ f1 = x , f2 = y , f3 = z , ... } ("f1" is short for "field 1", and so on)	fun f { f1 = x , f2 = y , f3 = z , ... } = ···	let val { f1 = x , f2 = y , f3 = z , ... } = *e* in ··· end	x : 'a y : 'b z : 'c
'a tree (homework)	LEAF NODE(l,x,r)	fun f (LEAF) = ··· | f (NODE(l,x,r)) = ···	case *e* of LEAF => ··· | NODE(l,x,r) => ···	l : 'a tree, x : 'a r : 'a tree
μScheme exp (page ??)	LITERAL v VAR x SET (x, e) ⋮	fun f (LITERAL v) = ··· | f (VAR x) = ··· | f (SET (x, e)) = ··· ⋮	case *e* of LITERAL v => ··· | VAR x => ··· | SET (x, e) => ··· ⋮	v : value x : name x : name, e : exp ⋮

Table 4.1: Identifying forms and extracting parts (ML builtins and 105 types)

name turns it into a nonfix name, which you can use as a value. Here are two classic examples:

```
fun sum  ns = foldl op + 0 ns
fun prod ns = foldl op * 1 ns
```

4. *Function contracts* are now enhanced: every function has a *most general type*. We consider the type to be part of the function's contract. The type is enforced by the compiler. In many cases, like `List.all` and `List.exists`, the name and the type form a sufficient contract all by themselves.

5. *Example results* and test cases work using the same ideas as in μScheme ("check-expect," "check-assert," and "check-error"), but the mechanisms are different. Compared with μScheme, ML's unit testing is verbose and awkward. But there is one small compensation: unlike μScheme, ML indicates checked run-time errors using *exceptions*. This feature makes it possible to check for the presence of a *particular* exception, like `Subscript`, `Empty`, or `Overflow`. In μScheme, that's not possible.

6. *Algebraic laws* are as helpful as ever. They must respect types. We will also develop a new design method that helps with writing right-hand sides of algebraic laws. The new method is based on types, and when we are ready to study types in detail, it will be presented in class.

7. *Coding case analysis* is *much* simpler than in Scheme: for case analysis over constructed data, we just use pattern matching. This feature makes the code look an awful lot like the algebraic laws. For case analysis of natural numbers or machine integers, we can often use partial pattern matching (one or more cases of interest, followed by a catchall case).

8. *Coding results* uses the same design methods as in Scheme. But in ML, both the concrete syntax and the abstract syntax are different from Scheme. Here are the key differences in the abstract syntax and our use of it:

   - In ML, the `let` form uses definitions and has a similar semantics to Scheme's `let*`. Direct recursion is accomplished by using `fun`, and mutual recursion by using `and`.
   - ML has a `case` form for pattern matching in an expression. (But usually we will pattern match in the `fun` definition form.)
   - Deconstruction of input data is *always* done by pattern matching. ML has functions like `car`, `cdr`, and `null?`, but they are used rarely, and only by experts.

9. *Revisiting tests* has the same intellectual content, but it's much more fussy to code. To run your tests, you'll need to study the `Unit` interface that is described in the guide to learning ML.

Syntax help for Standard ML

Standard ML is a full language, not simplified for teaching, and an exhaustive syntax summary would be overwhelming. This section presents the key syntactic tools that you will use most frequently. It is not exhaustive!

ML has four major syntactic categories. From the top down:

d	Definitions
p	Patterns
e	Expressions
τ	Types

These categories are related like this:

- *Definitions* sit at the top, and they contain both patterns and expressions. A typical definition form has a pattern on the left and an expression on the right. The definition of a Curried function may have multiple patterns on the left.

  The `val` form you already know is present, but instead of just a name on the left, it can take any pattern.

  The `define` form you already know is a special case of `fun`, but `fun` is more typically used with patterns, to express algebraic laws directly.

  There are two kinds of *type definitions*: type abbreviations (`type`) and fresh, algebraic data types (`datatype`). Both are called "types," and both definition forms contain types.

- *Patterns* are new. They are one of the two main interesting features of ML, and they are described in detail in Lesson 4 above. Patterns may contain types, but they usually don't—we put types in patterns only when we're debugging.

- *Expressions* resemble those that you already know, except for the `let` form. ML's `let` form contains *definitions*, not Scheme's name-value bindings. And unlike Scheme's `let`, which binds all names simultaneously, ML's `let` evaluates definitions in sequence, like Scheme's `let*`.

  Expressions may contain definitions and types. Expressions commonly contain definitions (any `let` form), but they rarely contain types—we put types in expressions only when we're debugging.

- *Types* are more general than the types you know from C and C++. We will study types at length.

To summarize the common forms of the categories listed above, I use these symbols for nonterminals:

x, f	Name (of a variable or function)
k	Literal (like 7 or `#"a"`)
K	Name of a value constructor
t	Name of a type

Using these symbols, here are some examples of the most commonly used forms of ML syntax:

$$
\begin{array}{rl}
p \Rightarrow & x \mid k \mid \texttt{(}p_1\texttt{, }p_2\texttt{)} \mid \texttt{(}p_1\texttt{, }p_2\texttt{, }p_3\texttt{)}\\
\mid & \texttt{[]} \mid p_1\texttt{ :: }p_2 \mid \texttt{[}p_1\texttt{,}p_2\texttt{, ..., }p_n\texttt{]}\\
\mid & \texttt{NONE} \mid \texttt{SOME }p \mid \texttt{LESS} \mid \texttt{EQUAL} \mid \texttt{GREATER}\\
\mid & K \mid K\ p
\end{array}
$$

$$
\begin{array}{rl}
e \Rightarrow & x \mid k \mid \texttt{(}e_1\texttt{, }e_2\texttt{)} \mid \texttt{(}e_1\texttt{, }e_2\texttt{, }e_3\texttt{)}\\
\mid & \texttt{[]} \mid e_1\texttt{ :: }e_2 \mid \texttt{[}e_1\texttt{,}e_2\texttt{, ..., }e_n\texttt{]}\\
\mid & \texttt{NONE} \mid \texttt{SOME }e \mid \texttt{LESS} \mid \texttt{EQUAL} \mid \texttt{GREATER}\\
\mid & K \mid K\ e\\
\mid & e\ e\ \cdots\\
\mid & \texttt{if }e_1\texttt{ then }e_2\texttt{ else }e_3\\
\mid & \texttt{let }d\ \cdots\texttt{ in }e\texttt{ end}\\
\mid & \texttt{(case }e\texttt{ of }p_1\texttt{ => }e_1\texttt{ | }p_2\texttt{ => }e_2\texttt{ | }\cdots\texttt{)}\\
\mid & \texttt{raise }e \mid \texttt{(}e_1\texttt{ handle }p\texttt{ => }e\texttt{)}\\
\mid & e_1\texttt{ andalso }e_2 \mid e_1\texttt{ orelse }e_2
\end{array}
$$

$$
\begin{array}{rl}
d \Rightarrow & \texttt{val }p\texttt{ = }e\\
\mid & \texttt{val (}x_1\texttt{, }x_2\texttt{) = }e\text{ (overlooked special case)}\\
\mid & \texttt{fun }f\ p_1\texttt{ = }e_1\texttt{ | }f\ p_2\texttt{ = }e_2\texttt{ | }\cdots\\
\mid & \texttt{fun }f\ p_1\ \cdots\texttt{ = }e_1\texttt{ | }f\ p_2\ \cdots\texttt{ = }e_2\texttt{ | }\cdots\\
\mid & \texttt{exception }K\\
\mid & \texttt{exception }K\texttt{ of }\tau\\
\mid & \texttt{type }t\texttt{ = }\tau\\
\mid & \texttt{type 'a }t\texttt{ = }\tau\\
\mid & \texttt{datatype }t\texttt{ = }K_1\texttt{ of }\tau_1\texttt{ | }K_2\texttt{ of }\tau_2\texttt{ | }\cdots\\
\mid & \texttt{datatype }t\texttt{ = }K_1\texttt{ | }\cdots\\
\mid & \texttt{datatype 'a }t\texttt{ = }K_1\texttt{ of }\tau_1\texttt{ | }K_2\texttt{ of }\tau_2\texttt{ | }\cdots\\
\mid & \texttt{datatype 'a }t\texttt{ = }K_1\texttt{ | }\cdots
\end{array}
$$

$$
\begin{array}{rl}
\tau \Rightarrow & \texttt{int} \mid \texttt{string} \mid \texttt{bool} \mid \texttt{char}\\
\mid & \tau\texttt{ list} \mid \tau\texttt{ option}\\
\mid & \tau_1\texttt{ * }\tau_2 \mid \tau_1\texttt{ * }\tau_2\texttt{ * }\tau_3\\
\mid & \tau_1\texttt{ -> }\tau_2\\
\mid & \texttt{'a} \mid \texttt{'b} \mid \texttt{'c}
\end{array}
$$

5. Program design with typing rules

One reason to use formal proofs (for operational semantics and type systems) is that proof rules often tell us how to write code. We can approach the coding task the same way we approach other coding tasks: we can start by viewing a proof as data and each proof *rule* as a form of data. But if the goal is to translate a type system (or any other proof system) into code, we are better off specializing the design process to write the translation directly. This lesson presents a suitably specialized process, which you will use for two assignments: type checking and type inference.

The special process for turning rules into code, even more than other processes, is ultimately meant to be internalized and abbreviated. It is possible to complete all the homework successfully without mastering the process, but if you do master it, you will find yourself writing clean code, fluently. Your coding will be driven primarily by questions like "what do I know?", "what can I compute from what I know?", and "what do I want to compute next." If you reach this level of fluency, you will not need to leave evidence of your design process, and you will not need to follow the coding suggestions to the letter.

5.1 Overall program design

When we code up a type system or other proof system, every judgment form is implemented by a function. A judgment form is a logical statement, amounting to "a claim is provable." A function implementing a judgment form follows one of two models:

- Every metavariable in the judgment form is an input to the function, and the function returns a Boolean that answers the question, "is this claim provable?"

  Example: judgment $x \in \operatorname{dom}\Gamma$. Both x and Γ are inputs, so the function takes a name and an environment and returns a Boolean. This is the function `isBound` from the ML homework.

  Example: judgment $\tau_1 \equiv \tau_2$. Both τ_1 and τ_2 are inputs, so the function takes two types and returns a Boolean. This is the function `eqType` from the type-systems chapter.

- Some metavariables are inputs and some are outputs. The function tries to compute values for the output metavariables such that the whole judgment is provable. If it succeeds, it returns those values. If it fails, it raises an exception.

  Example: judgment $\Delta, \Gamma \vdash e : \tau$. The inputs are Δ, Γ, and e, and the output is τ. This is the function `typeof` from the type-systems homework.

  To give an example of `typeof`, I assume I'm using the environments from the initial basis. If, in addition, I pass the input expression `(+ 2 2)`, `typeof` will succeed and will return type `int`. If I pass `(+ 2 #t)`, `typeof` will fail by raising the exception `TypeError`.

  To identify functions like this, I frequently write a judgment form with boxes around the outputs, as in "$\Delta, \Gamma \vdash e : \boxed{\tau}$."

To implement a type checker, a type inferencer, another static analyzer, or even an interpreter, you implement all the judgment forms. Here are some key questions:

- What function will each judgment form be implemented by? What are the inputs and the outputs?
- What are the proof rules for the judgment forms I implement? What judgments do those proof rules use?
- Which judgment forms are already implemented for me? Perhaps in the book?
- Which judgment forms do I have to implement?

The answers are different for each different type system, but many of the answers are found in the book:

Typed Impcore	Table 6.1 on page 376
Typed μScheme	Table 6.4 on page 407
nano-ML	Table 7.3 on page 480

5.2 Design steps for one function

To design a function that implements a judgment form, we follow the usual design steps:

1. *Forms of data.* Key data types found in type systems are

Γ	Type environment
Δ	Kind environment
C	Equality constraint (inference only)
e	Abstract syntax
τ	Type
σ	Type scheme (Hindley-Milner only)
α	Type variable

Like the judgment forms, these forms of data and their ML representations are shown in the tables on pages 376, 407, and 480.

Not all of these data are broken down by cases:

- Environments are never broken down by cases.
- Types are not usually broken down by cases, *except* when implementing a constraint solver for type inference.
- Type schemes have only one form, but when instantiating polymorphic values, type schemes are deconstructed. (Instantiation is implemented in the book.)
- Equality constraints and abstract syntax, when consumed, are broken down by cases.

2. *Example inputs.* To write examples of abstract syntax, we use concrete syntax. (Writing abstract syntax is what concrete syntax is for!) When possible, we do the same with types, which also have a concrete syntax.

3. *Function name.* To form names, we usually use nouns like "type" or verbs like "elaborate," "evaluate," "substitute," "conjoin," or "solve." Or some other name that is connected with the proof system. Function names are often given by one of the tables on pages 376, 407, and 480.

4. *Function contract.* This is a key step. At an abstract level, all the contracts are the same: "implement a judgment." But it helps to be concrete. Example: judgment form $C, \Gamma \vdash e : \tau$ from nano-ML. Here's the function and its contract:

   > `val typeof : exp * type_env -> ty * con`
   > Calling `typeof` (e, Γ) returns (τ, C) such that $C, \Gamma \vdash e : \tau$. Constraint C is not guaranteed to be solvable.[1]

   To help us remember a function's contract, we can write the corresponding judgment form with boxes around the outputs: $\boxed{C}, \Gamma \vdash e : \boxed{\tau}$.

5. *Example results.* While it is possible to write example inputs and results directly as ML values, this level of work can usually be avoided. For most type-checking tasks, try this method:

   - Whenever possible, use the environments in initial basis.
   - When necessary, extend the initial environment with just one or two definitions.
   - Write example inputs (expressions and definitions) using the *concrete* syntax of Typed Impcore, Typed μScheme, or nano-ML. Write example outputs likewise.
   - Turn the examples into unit tests using the `check-type`, `check-principal-type`, and `check-type-error` forms.

   When implementing the proof system for the constraint solver, there is no concrete syntax for constraints. Fortunately, the ML syntax for writing constraints is not too painful. So to design the constraint solver, write example inputs and results as you learned to do when coding the first ML assignment.

6, 7, 8. *Algebraic laws and code.* Experienced type-system hackers can code a type system from inference rules alone. But while you are learning, you are better off following the step-by-step translation procedure outlined below. This procedure is the one you want to internalize.

9. *Revisit unit tests.* To test a type checker, use concrete syntax with unit-test forms in source code, as described in step 5 above. To test the nano-ML constraint solver, use `Unit` functions to embed unit tests inside your interpreter.

5.3 Translating rules to code

To implement a type system or other proof system, you organize the rules by the form of the judgment that appears in the conclusion. All the rules that conclude the same form are implemented by the same function. So for example, if you consult the table 6.4 on page 407, you'll see that the Typed μScheme type system needs you to write two functions: `typeof` and `elabdef`. The corresponding forms of concluding judgment are $\Delta, \Gamma \vdash e : \boxed{\tau}$ and $\langle d, \Delta, \Gamma \rangle \to \boxed{\Gamma'}$. You'll find the corresponding collections of rules summarized in Figures 6.10 and 6.11 on pages 437 and 438. (Not everything is in the summaries; in particular, the rules for literal values are found only at the beginning of section 6.6.5, which starts on page 401.)

Now you know the function you are implementing, its inputs and outputs, and the collection of rules that specifies the implementation. Your next step is to break the rules down by forms of data, which is to say the forms

[1] The inputs and outputs to type inference are a frequent source of confusion. Examples of other type systems, as well as operational semantics, suggest the heuristic, "inputs on the left, outputs on the right." But that's not how logic works—the heuristic is wrong. The logical structure of a sequent is

$$\textit{context} \vdash \textit{claim}.$$

In type inference, one of the outputs of the algorithm is the constraint C. This constraint, which is part of the context, expresses the assumptions that have to be made in order for term e to be typable.

of the *abstract syntax* in the conclusions of the rules. For `typeof`, this is expression syntax; for `elabdef`, this is definition syntax. For each form of syntax, expect one or more rules. In our case, we're expecting *exactly* one rule per form of syntax—systems with more than rule are not hard to implement, but they are beyond the scope of this lesson.

To finish the job, you translate each rule into code. Code produced by beginners looks quite different from code produced by experts, but the expert's code reflects the same thought process as the beginner's—the code is just streamlined. This lesson teaches you to write beginner's code. But because you'll see expert code in lecture and in the model solutions, the lesson also adds notes about what experts do.

In a beginner's code, *every* right-hand side (whether a law or direct to ML code) takes the form of a `let` expression, with bindings and a body. The `let` expression is developed using the custom design process embodied in these steps:

(A) In your chosen rule, identify all the judgments above the line. Put them on a list of *unproved judgments*. (The expert knows exactly which judgments are proved and unproved at every point in the code, but the judgments might not need to be written down.)

(B) In each unproved judgment, identify (1) what function implements the judgment and (2) which parts of the judgment are inputs and which parts are outputs. Draw a box around each output. (The expert knows at a glance what function implements each judgment and what the inputs and outputs are.)

(C) Look at all the *variables* that appear in *output* positions in *unproved* judgments. If any variable appears in more than one such position, all those outputs must be equivalent. In the original rule and the list of unproved judgments, rename outputs until all output variables have unique names, and introduce equality or equivalence constraints.

If you are writing type inference, return the new constraints from the function. If you are writing a type checker, add equivalence judgments to your list of unproved judgments. (The expert may leap directly to new constraints or new judgments, without doing the intervening renaming step.)

(D) Look at all the *literals* or *expressions* that appear in output positions in unproved judgments. In a type checker, these are likely to be types like `bool` or $\tau_1 \rightarrow \tau_2$. These also must be renamed, and equality or equivalence constraints must be introduced. (Again, the expert may leap directly to suitable constraints, without the renaming step.)

(E) Once every judgment above the line has only variables in output positions, and no variable appears in more than one output position, you are ready to write code to discharge the unproved judgments. You will need to keep track of the *available metavariables*. These include any inputs to the judgment below the line—and their parts—plus any metavariables introduced in the `let` bindings—of which you don't have any yet.

As long as there is an unproved judgment, repeat the following step: find an unproved judgment whose *inputs* are all available. Now as a beginner, implement that judgment in one of two ways:

- If the judgment has any outputs, add a `let` binding that calls the function:

```
val output-pattern = f inputs
```

- If the judgment has *no* outputs, add a trivial binding that confirms the judgment is provable, and if not, raises an exception.[2]

```
val () =
  if f inputs then
    ()
  else
   raise TypeError message
```

(The expert finds a judgment in exactly the same way, but the expert may know at a glance which metavariables are available and can therefore easily choose a judgment whose inputs are available. But the expert's code may be much more heavily condensed than a beginner's code: an expert may not always bind a call's result in a `val`, and an expert is quite likely to combine the Boolean test and the exception raising into a conditional that appears in the body of the outer `let`.)

Once the judgment is implemented:

- Cross it off the list of unproved judgments.
- Note that its outputs, if any, are now available.

Beginners and experts alike continue repeating this step until there are no more unproved judgments.

(F) Once all the unproved judgments have been dealt with, return to the judgment below the line. Look at the outputs. Every output should either be an available variable or should be formed from available variables. Use these variables to make the output, and place it in the body of the `let`. (An expert carries out the same process, but if the rule is

[2] If your proof system has more than one rule per syntactic form, as in an interpreter for an operational semantics, for example, you would not raise an exception. Instead, you would try the next rule. Once you've tried all possible rules, *then* you raise the exception.

simple enough, the expert may choose simply to return a result without the administrative overhead of a `let`.)

Complete example

Let's demonstrate the process with the If rule:

$$\frac{\Delta,\Gamma \vdash e_1 : \texttt{bool} \qquad \Delta,\Gamma \vdash e_2 : \tau \qquad \Delta,\Gamma \vdash e_3 : \tau}{\Delta,\Gamma \vdash \textsc{If}(e_1, e_2, e_3) : \tau}$$

(A) We have these unproved judgments:

$$\begin{gathered}\Delta,\Gamma \vdash e_1 : \boxed{\texttt{bool}}\\ \Delta,\Gamma \vdash e_2 : \boxed{\tau}\\ \Delta,\Gamma \vdash e_3 : \boxed{\tau}\end{gathered}$$

(B) Each of the unproved judgments is implemented by `typeof`. The inputs are Δ, Γ, e_1, e_2, and e_3, and the outputs are `bool` and τ.

(C) The variable τ appears in two output positions in unproved judgments. I rename the second position τ_3, and I introduce the type-equivalence judgement $\tau \equiv \tau_3$.

My original rule now looks like this:

$$\frac{\Delta,\Gamma \vdash e_1 : \texttt{bool} \qquad \Delta,\Gamma \vdash e_2 : \tau \qquad \Delta,\Gamma \vdash e_3 : \tau_3 \qquad \tau \equiv \tau_3}{\Delta,\Gamma \vdash \textsc{If}(e_1, e_2, e_3) : \tau}$$

and the unproved judgements look like this:

$$\begin{gathered}\Delta,\Gamma \vdash e_1 : \boxed{\texttt{bool}}\\ \Delta,\Gamma \vdash e_2 : \boxed{\tau}\\ \Delta,\Gamma \vdash e_3 : \boxed{\tau_3}\\ \tau \equiv \tau_3\end{gathered}$$

(D) Next, I spot `bool` in an output position. I rewrite it to τ_1. My original rule now looks like this:

$$\frac{\Delta,\Gamma \vdash e_1 : \tau_1 \qquad \Delta,\Gamma \vdash e_2 : \tau \qquad \Delta,\Gamma \vdash e_3 : \tau_3 \qquad \tau \equiv \tau_3 \qquad \tau_1 \equiv \texttt{bool}}{\Delta,\Gamma \vdash \textsc{If}(e_1, e_2, e_3) : \tau}$$

and the unproved judgements look like this:

$$\begin{gathered}\Delta,\Gamma \vdash e_1 : \boxed{\tau_1}\\ \Delta,\Gamma \vdash e_2 : \boxed{\tau}\\ \Delta,\Gamma \vdash e_3 : \boxed{\tau_3}\\ \tau \equiv \tau_3\\ \tau_1 \equiv \texttt{bool}\end{gathered}$$

(E) All of my unproved judgments have only variables in output positions (τ_1, τ, and τ_3), and no variable appears in more than one output position. I'm ready to start discharging them.

The type system has only one rule for If, so I'm going to code it directly in one clause of a clausal definition for `typeof`. That clause begins something like this:

```
fun typeof (IFX (e1, e2, e3), Delta, Gamma) = ...
```

And my available variables are Δ, Γ, e_1, e_2, and e_3. Time to start proving judgments. Variables are available for any of the first three unproved judgments. I start with the first.

- Judgment $\Delta,\Gamma \vdash e_1 : \boxed{\tau_1}$ has output τ_1, so I add this definition form to my `let` bindings:

  ```
  val tau_1 = typeof (e1, Delta, Gamma)
  ```

  This binding adds τ_1 to my list of available variables.

- Now that τ_1 is available, I can discharge the unproved judgment $\tau_1 \equiv \texttt{bool}$. This judgment doesn't have any outputs, so I have to test it to see if it is provable. From Table 6.4, the corresponding function is `eqType`. I add this definition form to my `let` bindings:

  ```
  val () =
    if eqType (tau_1, booltype) then
      ()
    else
      raise TypeError "...message..."
  ```

- I now discharge the second unproved judgment $\Delta,\Gamma \vdash e_2 : \boxed{\tau}$ with this binding:

  ```
  val tau = typeof (e2, Delta, Gamma)
  ```

- And I discharge the third unproved judgment $\Delta,\Gamma \vdash e_3 : \boxed{\tau_3}$ with this binding:

  ```
  val tau_3 = typeof (e3, Delta, Gamma)
  ```

- With variables τ and τ_3 finally available, I can discharge the last unproved judgment, $\tau \equiv \tau_3$:

  ```
  val () =
    if eqType (tau, tau_3) then
      ()
    else
      raise TypeError "...message..."
  ```

- Finally, I return to the judgment below the line, $\Delta,\Gamma \vdash \textsc{If}(e_1, e_2, e_3) : \boxed{\tau}$. The output is τ, so `tau` is what I return in the body of the `let`.

With all steps complete, here's my full implementation of the If rule, in beginner's style:

```
fun typeof (IFX (e1, e2, e3), Delta, Gamma) =
  let val tau_1 = typeof (e1, Delta, Gamma)
      val () =
        if eqType (tau_1, booltype) then
          ()
        else
          raise TypeError "...message..."
      val tau   = typeof (e2, Delta, Gamma)
      val tau_3 = typeof (e3, Delta, Gamma)
      val () =
        if eqType (tau, tau_3) then
          ()
        else
          raise TypeError "...message..."
  in  tau
  end
```

And here's the code I might write as an expert, starting with an internal function `ty` that reduces the bureaucracy of the repeated calls to `typeof`. I also combine bindings; I change name `tau` to `tau_2`; and to make it easier to write good error messages, I invert conditions on the `if`s:

```
fun typeof (IFX (e1, e2, e3), Delta, Gamma) =
  let fun ty e = typeof(e, Delta, Gamma)
      val (tau_1, tau_2, tau_3) =
          (ty e1, ty e2, ty e3)
  in  if not (eqType (tau_1, booltype)) then
        raise TypeError "...message..."
      else if not (eqType (tau_2, tau_3)) then
        raise TypeError "...message..."
      else
        tau_2
  end
```

Summary of the steps

You'll repeat these steps for every rule, so it's worth having a short summary:

(A) List unproved judgments.

(B) Know function names; box the outputs.

(C) Rename duplicated output variables.

(D) Introduce variables for output literals.

(E) Discharge each unproved judgment:

- Find a judgment whose inputs are available.
- If it has outputs, `val` bind them.
- It if has no outputs, confirm it returns `true` or raise an exception.

(F) Return the outputs from the rule's conclusion (the judgment below the line).

6. Program design with abstract data types

Our 9-step design process is intended for functions. Abstract data types introduce new problems, which are dealt with by means of *abstraction functions* and *invariants*. While you will have seen these ideas in COMP 15 and possibly in COMP 40, you may not have gotten the vocabulary. So this chapter sketches the main ideas, with the vocabulary. It also explains how to extend and apply our design process to *client* code (outside of a type's module) and *implementation* code (inside of a type's module).

6.1 Creator, producer, observer, mutator

Following a taxonomy of Barbara Liskov, operations on an abstract data type are classified as *creators*, *producers*, *observers*, or *mutators*. This classification is explained in *Programming Languages: Build, Prove, and Compare*, in section 2.4.2 on page 113. It informs both the design of interfaces and the design of the *client* code that uses those interfaces.

6.2 Representation, abstraction, invariant

The new idea here is *data abstraction*: we have a *representation* that lives in the world of code, and an *abstraction* that lives in the world of ideas. This idea pervades COMP 40, and you may also have encountered it in COMP 15. "We can't put a student in the computer, but we can put a *representation* of a student in the computer."

For the classic data structures we study in COMP 15, the world of ideas is usually the world of mathematical ideas, like sets, sequences, and finite maps. For more practical system-building, the world of ideas is usually the outside world of problems we're trying to solve: students, images, games, or what have you. In all cases, there are some steps you follow before you can start designing functions:

(a) Identify the abstraction.

(b) Choose the data you will use to represent the abstraction. This choice is often informed by desires about the cost model. For example, if you want constant-time lookup, you might choose a hash table.

(c) Explain the mapping from the representation to the abstraction. This mapping is called the *abstraction function* and is written $\mathcal{A}$.

(d) Design *invariants* that restrict the representation. Invariants sometimes also support the cost model.

In data-structures class, you study implementations of well-known abstractions like sets, sequences, and finite maps (also known as "tables" or "dictionaries"). In real systems, designing new, useful abstractions is often hard. A new abstraction typically goes through multiple iterations of refinement or even redesign. In COMP 105, we avoid this part of the design process—the abstractions have been designed for you, and your role is to come up with good representations.

Choosing a representation may or may not be hard. In data-structures class, you learn about proven, effective representations. These representations are found in books, and the same representations are used in real systems. But when you're designing a representation for a new, system-specific abstraction—like a two-player game, for example—you will have to fall back on your own ideas.

Once you've chosen a representation, you write an abstraction function and a representation invariant. These elements play different roles:

- An abstraction function tells us what each value stands for, so we can be confident we are implementing a module's operations according to their contracts. An abstraction function need be defined *only* on representations that satisfy the invariant!

  As an example, the abstraction function for a binary search tree usually just accumulates all the values held at all of the nodes.

  The role of the abstraction function is to make sure your representation works, and to help you understand how to implement each function. The function's contract is written in terms of the abstraction ("in the world of ideas"), but its code operates on the representation ("in the world of code"). To argue that a function's contract is fulfilled by its implementation, you use the abstraction function to map the representations of arguments and results up to their corresponding abstractions.

- A *representation invariant* tells us what is true about the representations we encounter at run time. It could be something as simple as "the list contains no duplicate elements" (for an implementation of sets as lists) or something so complicated as to demand to be written mathematically. Interesting data structures usually satisfy multiple representation invariants. These may be referred to individually, and they may also be collectively called "the invariant."

A good example is a binary search tree. A binary search tree always has an *order invariant*—usually "smaller to the left, larger to the right." The order invariant guarantees that a search function will find an object, if present, without having to look at every node of the tree. A serious, sophisticated binary search tree also has a *balance invariant.* There are many different forms of balance invariant, but they all guarantee that search takes a number of steps that is at most logarithmic in the number of nodes in the tree.

The role of the representation invariant is to help you write the code, and often to meet expectations about costs. Every function is permitted to *rely* on the representation invariant, which means it may *assume* that every input satisfies the invariant. And every function is obligated to *guarantee* the representation invariant, which means it must ensure that every output satisfies the invariant. This combination is called *rely/guarantee reasoning.*

These ideas will be clearer with some more examples.

6.3 Two examples

Sets

My first example abstraction is a set. To write my abstraction function, I use set notations like $\{\cdots\}$ and $\cup$. Potential representations include a list, a sorted list, and a binary search tree, as shown in Table 6.1. Any of these representations will work, but they have different cost models:

- A plain list is easy to implement, and as long as sets are small, it's cheap. By avoiding repeated elements, we limit worst-case costs to the cardinality of the set. The abstraction function simply converts the list to a set.

- A sorted list stands for the same abstraction as an unsorted list, and so it shares the abstraction function with the unsorted list. But the sorted list satisfies an additional invariant: it is sorted. This invariant changes the cost model: adding a new element now takes half as much expected time as with the unsorted list, as does searching for an element that's not present.

- A binary search tree is the most difficult to implement, but if it includes a balance invariant in addition to just the order invariant, it is guaranteed to do insertion, lookup and deletion in logarithmic time, even in the worst case.

  The order invariant is a great example of rely/guarantee reasoning: the `lookup` function relies on the invariant, and the `insert` function both relies on it and guarantees it.

Priority queues

My next example abstraction is a priority queue. This abstraction is actually just a sorted list of values, with operations that provide access only to the front of the list. As shown in table 6.2, a priority queue can be represented as a sorted list or as a binary tree, but if you studied data structures at Tufts, you probably learned to represent it as an array, under the name "heap." My favorite representation is the binary tree: it is easy to implement, and with the "leftist heap" invariant, it is super efficient. Regardless of the representation, my abstraction function maps the representation onto a sequence of elements.

6.4 Suggestions

For homework, you'll write abstraction functions and representation invariants for other abstractions. Representation invariants are usually pretty easy to write down:[1] because they operate on actual representations, they can be coded, typechecked, and tested. Abstraction functions are more challenging, because by definition, they map to the world of ideas, which might not be represented in code. Here are some suggestions for the abstraction functions for homework problems:

- For natural numbers, the world of ideas is the mathematician's world of natural numbers, sometimes written $\mathbb{N}$. Operators and notations that are well-defined in this world include 0, S (successor), and $+$.

- For integers, the world of ideas is the mathematician's world of integers, sometimes written $\mathbb{Z}$. Operators and notations that are well-defined in this world include 0, $+$, $-$, and $\cdot$ (multiplication). At need, you can also resort to div and mod.

- For coins, the simplest abstraction is probably a list of denominations, such as "quarter, quarter, dime, quarter, nickel." Operators and notations that are well-defined in this world include all the usual operations on lists.

[1] The order invariant for a binary search tree is an exception; it's hard to write down correctly and completely, especially in an abstract setting where the type of a value is not known.

Abstraction	*Operations*	
Set	At minimum, `empty/new`, `insert`, `delete`, `member?`; possibly also `empty?`, `size`, `union`, `inter`, `diff`	
Representation	*Invariant*	*Abstraction Function*
List	No element is repeated.	$\mathcal{A}(\texttt{[]}) = \{\,\}$ $\mathcal{A}(x\,{:}{:}\,xs) = \{x\} \cup \mathcal{A}(xs)$
Sorted list	No element is repeated; elements are sorted.	(Same as list.)
Binary search tree	No element is repeated; smaller elements are in left subtrees; larger elements are in right subtrees; perhaps some sort of balance invariant.	$\mathcal{A}(\texttt{EMPTY}) = \{\,\}$ $\mathcal{A}(\texttt{NODE}(l,x,r)) = \mathcal{A}(l) \cup \{x\} \cup \mathcal{A}(r)$

Table 6.1: Representations of sets

Abstraction	*Operations*	
Priority queue	At minimum, `empty/new`, `insert`, `empty?`, and `delete-min`; possibly also `size`, `find-min`, `merge`	
Representation	*Invariant*	*Abstraction Function*
List	List is sorted with the smallest element at the front (inefficient unless small).	$\mathcal{A}(xs) = xs$
Array	Element at index i is not larger than the elements at indices $2i$ and $2i+1$, if any.	$\mathcal{A}(a) = \texttt{sort}(a)$
Binary tree	Element at node is not larger than elements at left and right child, if any.	$\mathcal{A}(\texttt{EMPTY}) = [\,]$ $\mathcal{A}(\texttt{NODE}(l,x,r)) = x :: \texttt{merge}(\mathcal{A}(l), \mathcal{A}(r))$
Leftist heap	Binary tree, with the additional invariant that every left subtree is at least as high as the corresponding right subtree.	(Same as binary tree.)

Table 6.2: Representations of priority queues

- For players, the abstractions are X and O.

6.5 How design steps are affected

Abstract data types affect the design process because of two changes:

- Contracts for functions are written for the abstraction ("in the world of ideas"), but the functions themselves are written for the representation ("in the world of code").
- Outside its defining module, an abstract type has no forms of data.

The implications are explored below.

Design steps for client code

Outside an abstract type's module, code is called a *client* of the module. Here's how each step of the design process works in client code:

1. *Forms of data.* Manifest types have exposed representations, so client code works with them just as usual.

   Abstract types don't give access to the forms of data. If you're a client and you're consuming abstract data, you'll typically be calling observers—sometimes mutators. And to make new abstract values, you'll use creators and producers.

A value of abstract type is a little bit analogous to a function: all you can do with it is what's in the interface.

2. *Example inputs.* Example inputs can be made only by calling creators, producers, and sometimes mutators. (Mutation makes testing harder.)

3. *Functions' names.* Nothing changes.

4. *Functions' contracts.* The client's own contracts may mention the abstraction or may even be written in terms of the abstraction. The client doesn't know about the representation—which is mostly the point.

5. *Example results.* Any example results of abstract type have to be expressed indirectly, again using creators, producers, and sometimes mutators. (Mutation makes testing harder.)

6. *Algebraic laws.* When client code consumes a value of abstract type, it can't have one algebraic law per form of data—an abstract type has only one form of data. Instead, to decide on a law, you can use side conditions from an observer, or you can even break an observed value down by forms of *its* data. For example, in the Abstract Game Solver, it's useful to write laws for the `advice` function by breaking down the list of legal moves.

7. *Case analysis.* You can't do case analysis on a value of abstract type *directly*. But you may be able to do case analysis on the results of calling an observer in the interface.

8. *Coding results.* Right-hand sides are coded as usual.

9. *Revisit unit tests.* Unit tests can be written as usual. To construct values for use in unit tests, you can call creators and producers (and possibly mutators) for the abstract type.

   Unit testing has a pitfall: before writing unit tests, we have to know how to compare test results for equality. A well-designed abstraction will include an equality-testing function, but unit tests often try to use a built-in equality, which may or may not work. In particular, the built-in equality defined by Standard ML works on *representations*, not abstractions, and it may say two values are different when they should actually be considered the same.

Design steps for implementations

Inside an abstract type's module, code has complete access to the representation. Here's how each step of the design process works in an implementation:

1. *Forms of data.* The implementation has complete access to the representation, so forms of data are available as usual.

2. *Example inputs.* Example inputs can be written as usual—but care must be taken to be sure that every example input satisfies the representation invariant.

3. *Function's names.* Nothing changes.

4. *Function's contracts.* The contract of each function is given in the interface, and it is written in terms of the abstraction. But the function itself is written in terms of the representation. To be confident that each function fulfills its contract, you must define the function with the type's abstraction function in mind.

   Also, each *exported* function[2] has these amendments added to its contract:

   - Every input of abstract type satisfies the representation invariant for that type.
   - Every output of abstract type must satisfy the representation invariant for that type.

   Private functions may, if they wish, deal with arguments and results that don't satisfy the representation invariant. Indeed, one common use of private functions is to re-establish an invariant before returning a result. Each private function's relationship to and action on representation invariants must be documented in its contract.

5. *Example results.* Example results can be written as usual, as can unit tests.

6. *Algebraic laws.* Algebraic laws can be written as usual.

7. *Case analysis.* Case analysis can be based on algebraic laws as usual.

8. *Coding.* Right-hand sides are coded as usual.

9. *Revisit unit tests.* Unit tests can be written as usual.

[2] A function is exported if it is visible in the interface that is being implemented. C++ calls these functions "public."

7. Program design with objects

Like abstract data types, objects demand that we adapt our 9-step design process, which is intended for functions. This lesson explains the adaptations, focusing on the effects of dynamic dispatch and the limitations on access to representation. It also discusses each step in the design process, and it shows how to convert algebraic laws to double dispatch.

In order to use this lesson effectively, you should refresh your memory on the basic 9-step process described in the introduction, as well as the preceding lesson (design with abstract data types), on which this one builds.

7.1 Designing with abstraction

Many ideas from abstract data types also work with objects:

- Barbara Liskov's taxonomy of *creators*, *producers*, *observers*, and *mutators* (Ramsey, section 2.4.2 on page 113) is equally useful for both abstract data types and objects.

- Whether an abstraction is implemented using abstract data types or objects, *abstraction functions* and *invariants* continue to play a key role.

- Contracts for methods are written for the abstraction ("in the world of ideas"), but the methods themselves are written for the representation ("in the world of code").

- Like an abstract data type outside its defining module, an object has only one form of data, which is opaque.

- Whether data abstraction is implemented using abstract data types or objects, program design is essentially interface design. This aspect of program design is beyond the scope of Comp 105—in 105, we work with designs that are given to us.

These similarities help, but our design and coding are affected by these key differences:

- With either form of abstraction, our 9-step design process is useful for designing each piece of the abstraction. But the pieces are different: with abstract data types, each piece is a function that sits inside a module. With objects, each piece is a method that sits inside a class definition.

- Abstract data types give access to the representation of *every* value of abstract type, including parameters to all functions. This access comes with an obligation to understand and respect the invariant. Objects give access only to the representation of the receiver. Each argument must be treated as an abstraction and dealt with by sending messages in its public protocol. (If the argument is supposed to be "like" the receiver, it can also be sent messages from the private protocol that they have in common.) This difference is discussed further in the textbook in section 10.2, which starts on page 741, and in section 10.5.8, which starts on page 788.

- With objects, interfaces are designed not just be used by clients, but also to be inherited from by subclasses.

- An abstract data type can have any kind of representation: an algebraic type, a tuple, or even a function, for example. An object can have only one kind of representation: a collection of named instance variables.

- A function that expects a value of abstract type can be used *only* with values of that type. But a method that expects an object of a given class can be used with objects of *any* class, provided those objects mimic the protocol of the class that the method is expecting. This mimicry, which is a key feature of object-oriented systems, is called *behavioral subtyping* or sometimes "duck typing."

7.2 How design steps are affected

Our design process is most heavily influenced by two aspects of object-orientation:

- *Dynamic dispatch* changes the way we identify forms of data and the way we code case analysis. Case analysis in particular is so changed that you may not recognize it as "coding." In particular, we never ask an object, "how were you made?"

- *Information hiding* denies access to the representations of other objects, even if we know everything about their class. In other words, only the receiver of a message knows from what parts it was made. We never ask an argument object, "from what parts were you made?"

The effects of dynamic dispatch

A beginning object-oriented programmer must learn new design techniques for data that take multiple forms: Booleans, lists, trees, and so on. We can draw on our experiences with with algebraic data types and abstract data types. For Smalltalk and other class-based languages, I recommend these design techniques:

- Every abstraction should be associated with a *class*. The class defines the abstraction's *protocol*, which is the Smalltalk name for its interface. If the abstraction can take multiple forms, which is common, some messages in the protocol will be designated as *subclass responsibilities*. These messages will be implemented differently for different forms. An example is the "do something to every element" (`do:`) method defined on every form of `Collection`.

  A class whose methods include subclass responsibilities is an *abstract class*. It is not itself instantiated; only subclasses are instantiated.

  An effective abstract class may also define *common methods* which are shared among multiple forms of the abstraction. An example is the `size` method on collections, whose implementation is shared by lists, sets, dictionaries, and so on.

- When the abstraction has multiple forms, *each form should be implemented by its own subclass*. For example, class `Boolean` has subclasses `True` and `False`. Class `Collection` has subclasses `Set` and `List`.

- Case analysis of forms of data is implemented by dynamic dispatch. The subclass for a form knows what form it is, and its methods contain code only for that form.

We do case analysis by dynamic dispatch because in an object-oriented system, we don't get to look at the forms of argument objects. This limitation is part of what makes object-orientation different from abstract data types, and it deserves a deeper look.

Forms of data, access to representation

When I'm using abstract data types, and I'm a function defined in a particular module, I have access to the representation of every value whose type is defined inside that module. When I'm using objects, and I'm a method defined on a particular class, my level of access depends on what object I'm looking at:

A. If I'm looking at the receiver, which is an instance of my class, I have full access to its representation. This representation, which is the representation of `self`, always takes the same form: a collection of instance variables whose names I know. They are the variables from my class definition, plus the ones inherited from the definitions of my superclasses. I can refer to them by name, and I can mutate them using `set`.

B. If I'm looking at another object, I have no access to its representation: all I can do is send messages to it. To know *what* messages, I first consult my own class's protocol, which gives me my contract. My contract tells me what class of object I'm looking at, and I can send any message—but only those messages—that are in the class's public protocol.

C. If my contract tells me that I'm looking at an object of the same class that I am, I still can't access its representation—this is the big limitation in an object-oriented system—but in addition to the messages in our shared, public protocol, I can also send it *private* messages. Private messages are included in all the arithmetic protocols, for example.

Behavioral subtyping—mimicry of a protocol by an object of a different class—can be extended to include private messages.

With the restrictions on access to representation in mind, we can now examine the design steps.

Design steps

Here's how each step of the design process works with objects:

1. *Forms of data.* Our *thinking* about forms of data is unchanged: we still want to know the different ways data can be formed, and this question is still the starting point for every design. Our *coding* is different: every form of data corresponds to a class, and each part from which that form is made corresponds to one instance variable of the class.

2. *Example inputs.* As with abstract data types, example inputs can be made only by calling creators, producers, and mutators. (In Smalltalk, mutation and mutable abstractions are common.)

3. *Method's name.* We're dealing with methods, not functions, but the criteria for naming are unchanged.

4. *Method's contract.* As with abstract data types, each contract is written on the level of the abstraction, not the representation.

5. *Example results.* Like example inputs, example results must be expressed indirectly, using creators, producers, and mutators.

6. *Algebraic laws.* Perhaps surprisingly, algebraic laws involving forms of data are just as useful as in any other setting. The key is to know *what* values you need to do case analysis on. If it's just one value, make that value the receiver, and you're good to go. If you need to do case analysis on multiple values, you need to rewrite the laws to support multiple dispatch. To see how, read **??** of this lesson.

7. *Case analysis.* In an object-oriented language, case analysis is not what you find in a functional language or an imperative language:

   - It's not in a syntactic form like `if` or `case`.
   - We can do case analysis only on the receiver of a message.
   - Instead of all cases appearing in the same function, different cases appear in different method definitions.
   - By the time a method's code is running, the case analysis is already done.

   You predetermine the case analysis when you write the code, by putting the right method definition on the right class. (The compiler does the actual case analysis at run time, by dispatching to that method.) But you can still translate algebraic laws. Here's a simple example for `append`, with laws written in ML notation:

   ```
   append []      ys = ys
   append (x::xs) ys = x :: append xs ys
   ```

   Each form of data is its own subclass, and each subclass gets the case with its form. Here's a subclass for a cons cell:

   ```
   (class MLCons Object
     [first rest]
     (class-method first:rest: (z zs) ...)
        ; allocate & initialize a cons cell

     (method append: (ys)
        (first:rest: MLCons
                     first
                     (append: rest ys)))
   )
   ```

   And here's one for an empty list:

   ```
   (class MLNil Object
     []
     (method append: (ys)
        ys)
   )
   ```

8. *Coding results.* Once you've done your case analysis, the right-hand sides of your algebraic laws are coded more or less as usual—instead of passing values to functions, you are sending messages to objects, but the design is the same.

   In Smalltalk, other things are changing that affect coding: control flow is implemented in continuation-passing style, mutable abstractions are common, and you might use loops or mutation. These changes affect our coding technique, but they don't influence the design process.[1]

9. *Revisit unit tests.* Unit tests can be written as usual. To get objects for use in unit tests, you may write the occasional numeric literal or array literal, but mostly you send messages to objects.

7.3 Laws for double dispatch

What if you want to do case analysis on two or more forms of data? You code your analysis using *multiple dispatch.* This section shows a short example coding case analysis on two forms, using *double* dispatch. The example is multiplication of signed integers.

A signed integer is one of the following:

- `:+` n, where n is a natural number
- `:-` n, where n is a natural number

Signed integers are multiplied according to these algebraic laws:

```
(:+ n) * (:+ m) = :+ (n * m)
(:+ n) * (:- m) = :- (n * m)
(:- n) * (:+ m) = :- (n * m)
(:- n) * (:- m) = :+ (n * m)
```

On the left, `*` stands for multiplication of signed integers; on the right, it stands for multiplication of natural numbers.

Coding these laws requires nested case analysis. To do it in Scheme, you write nested `if` expressions. To do it in ML, you write tuple patterns (which the compiler translates into nested `ifs`). To do it in Smalltalk, you need nested dispatch—in this case, double dispatch.

To implement a method using double dispatch, we begin by extending an object's protocol with new, private messages. Each message encodes two pieces of information: the *name* of the original method, and the `form` of the original receiver. In our example, the original method is `*`, and the original receiver can have one of two forms: `:+ n` or `:- n`. So I'll use two new messages:

- Message `timesPos:` means `*` was sent to a receiver of the form `:+ n`
- Message `timesNeg:` means `*` was sent to a receiver of the form `:- n`

[1] Mutable state actually does influence a design process, but designing with mutable state is beyond the scope of COMP 105.

Using these messages, I write "laws of dispatch:"

```
(:+ n) * A = (timesPos: A (:+ n))
(:- n) * A = (timesNeg: A (:- n))
```

In these laws, there's no case analysis on the argument `A`. The only case analysis is on `:+ n` versus `:- n`, which should be coded by dynamic dispatch of `*` on the class of the receiver. We implement the `*` method by sending `timesPos:` or `timesNeg:` to the argument `A`.

To design `timesPos:` and `timesNeg:`, we need laws. To get them, we combine the dispatch laws with the original laws for `*`. We start by expanding the dispatch laws, noting that argument `A` can take two forms. I make two copies of each dispatch law—one where `A` is `:+ m`, and one when `A` is `:- m`:

```
(:+ n) * (:+ m) = (timesPos: (:+ m) (:+ n))
(:+ n) * (:- m) = (timesPos: (:- m) (:+ n))

(:- n) * (:+ m) = (timesNeg: (:+ m) (:- n))
(:- n) * (:- m) = (timesNeg: (:- m) (:- n))
```

By design, the expanded dispatch laws have the same left-hand sides as the original laws for `*`. So the corresponding right-hand sides must be equal:

```
(timesPos: (:+ m) (:+ n)) = :+ (n * m)
(timesPos: (:- m) (:+ n)) = :- (n * m)

(timesNeg: (:+ m) (:- n)) = :- (n * m)
(timesNeg: (:- m) (:- n)) = :+ (n * m)
```

Now I can write one method definition for each law. Here are two of the four:

```
(method timesPos: (plus-n) ; self is :+ m
  (withMagnitude:
        LargePositiveInteger
        (* magnitude (magnitude plus-n))))

(method timesPos: (plus-n) ; self is :- m
  (withMagnitude:
        LargeNegativeInteger
        (* magnitude (magnitude plus-n))))
```

Which method goes on which class? You figure it out.

Acknowledgements

The design process presented in this monograph is a direct descendant of the six-step “design recipe” developed by Felleisen, Findler, Flatt, and Krishnamurthi (second edition, 2018). The process and some individual lessons also draw on the work of Tony Hoare on abstract data types, the work of John Guttag and Jeannette Wing on algebraic specification, and the work of Will Cook on data abstraction.

The first draft of Lesson 1 was written by Nate Bragg.