DANIEL W. BAROWY

CSCI 334: READINGS

WILLIAMS COLLEGE

Portions copyright © 2024 Daniel W. Barowy

PRINTED BY WILLIAMS COLLEGE

Unless otherwise noted, this work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International license, Version 4.0 (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at https://creativecommons.org/licenses/by-nc-nd/4.0/. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

20240430th printing.

Science is what we understand well enough to explain to a computer; art is everything else.

Donald Knuth, *Things a Computer Scientist Rarely Talks About*

Contents

Preface

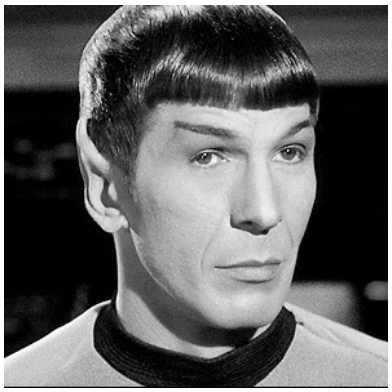


Figure 1: Not a typical programmer.

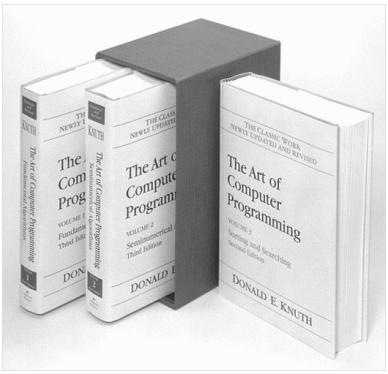


Figure 2: *The Art of Computer Programming*, one of the most important works of computer *science*, has the word *art* in the title. Is it art or science? Maybe it's both?

[1] $Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$

[2] Y is, in a sense, the meaning of *recursion*.

To ask whether a given programming language is suitable for a given task is logical, and you might think that this is what this course is about. We will certainly discuss that topic. But the practice of programming is as much about the aesthetic and emotional experience as it is about logic. The best programmers are not emotionless, Spock-like beings (Fig. 1). They are keenly aware of the emotional connection to their work. The best languages aren't just useful, they're pleasant to use.

This argument may seem strange to you. Isn't a computer the very embodiment of cold, hard rationality? I argue that it is not, and that thinking about it that way will prevent you from becoming a great programmer. I'm not the only one who thinks this way (Fig. 2).

In this course, we will talk about your thoughts *and* your feelings about programming. As we progress, I will point out the things that I find mysterious[1] and wonderful.[2] Seeing code as something that can be "beautiful" changes your relationship with coding. It's not mechanical drudgery—it's an art!

We will also talk about the key concept that underlies every good programming language: its *formal model*. A formal model describes the rules of the language, and is often mathematical in nature. A good formal model is a clarifying concept. Once you understand a language's model, you will know when it is appropriate to use one language versus another.

Before we start, take a moment and recall some of your programming experiences. Remember the feelings that you had. Now ask youself: did it feel good to program a computer? Should it?

A Brief Introduction to F#

This tutorial gets you started learning the F# programming language. F# is a modern version of the highly influential ML programming language. We focus at first on the F# programming environment and basic syntax. In subsequent reading, we will dive deeper into F#.

F# strikes many programmers new to the language as foreign. The language requires that you think about programming in new ways. However, even if you never program in F# again, the experience will likely influence your programming in the future. After I discovered F#, I wondered why conventional languages like Java and C++ had to be so complicated. The short answer is: they don't have to be!

If you have F# installed on your computer, you should be able to follow along by starting the F# interpreter[3] and then by typing expressions into your console. You are strongly encouraged to actively follow along by typing in and running the code examples in this book.

[3] Type `dotnet fsi` on the command line. To quit, type `#quit;;`. You will sometimes hear me refer to this interpreter as a "REPL," which stands for "read-eval-print loop."

Let's look at our favorite starter program written in F#.

```
printfn "Hello world!\n"
```

That is the entire program. Refreshing, isn't it?

What is F#?

F# is a *functional* programming language. A functional programming language differs in form than a conventional programming language like C[4] or Java[5]. Even if you decide that functional programming is not for you, exposure to functional programming ideas will change the way you think about coding for the better.

Functional programming encourages *expressions* over *statements*, *immutable* instead of *mutable* variables, and *pure, first-class functions* instead of *side-effecting* procedures. F# is also *strongly typed* unlike C, which is *weakly typed*, and Python, which is *dynamically typed*. These differences contrast sharply with those encountered in languages like C, Java, or

[4] C is an *imperative* programming language, meaning that programmers are expected to spell out every step of a program, including how values are stored in memory.

[5] Java is an *object-oriented* programming language. Object-oriented languages frame memory management and common software designs in terms of "objects," and are also usually imperative. Python and Java are imperative and object-oriented.

Python. The end result is that functional programs read more like mathematical statements than a sequence of steps. Let's briefly touch on each of these concepts.

Immutable variables

In a language like Python or C, a variable can be declared and written to many times. E.g.,

```
x = 2
x += 1  # the value of x is now 3
x += 1  # the value of x is now 4
x += 1  # the value of x is now 5
```

In a functional programming language, a variable can only be assigned once, when it is declared.

```
let x = 2
x += 1   // can't do this in F#; will not compile
```

You might be wondering how on earth you "update" data. It's done like this:

```
let x = 2
let y = x + 1
```

where `x` and `y` are *not* the same variable.[6]

[6] In other words, variables are *never* updated!

Variables in F# are *immutable*, meaning that once they are declared, their values will never change. If you're like me, this idea probably has left you scratching your head. Good. The value of this model of programming will become apparent to you in time.

Expressions

In a language like Python or C, a line of code can either return a value or not. For example, in Python:

```
print("hi")   # returns nothing; this is a statement
x + 1         # returns the value 3; this is an expression
```

In a functional language, all language constructs are expressions.

```
let x = 2  // returns a binding of the value 2 to the variable 'x'
x + 1      // returns the value 3
```

When a line of code returns nothing, we call it a *statement*. Since it

is pointless to have a line of code that does nothing, a statement does something by *changing the state of the computer*. Changing the state of the computer independently of a return value is called a *side effect*. Side effects are either banned in functional languages (e.g., pure Lisp, Haskell, Excel) or strongly discouraged (e.g., Standard ML, F#).

Pure, first-class functions

A *pure* function is a function that has no side effects. In F#, we usually write pure functions.

In C, one can write the following:

```
int i = 0;

void increment() {
    i++;
}
increment();  // i has the value 1
```

Observe that the `increment` function takes no arguments and returns no values and yet, it does something useful by altering[7] the variable `i`. One is not permitted to write code like this in a functional programming language because variables are immutable and functions are pure. Instead, one might write

[7] The technical term is *mutating*.

```
let increment n = n + 1
let i = 0
let i' = increment i  // i has the value 0; i' has the value 1
```

where `i` and `i'` are different variables, and where `increment` is a *function definition* for a function called `increment` that takes a single argument, `n`. Function calls look a little strange in F#, so you should expect that will take some time before you are adept at recognizing their form. It often helps to rewrite a program to use explicit parentheses and type annotations:

```
let increment(n: int) : int = n + 1
let i: int = 0
let i': int = increment(i)
```

This is also a valid F# program—in fact, it's exactly the same program—and if you find yourself struggling with syntax, I encourage you to write in this style instead.

Function definitions in F# are also *first class values*. What does that mean? Among other things, any first class value can always be assigned

to a variable. So yes, you can assign a function definition to a variable.[8] For instance,

```
let increment(n: int) : int = n + 1
let addone = increment
addone(3)  // returns 4
```

The type of the variable `addone` is a function definition (specifically, a function that takes an `int` as input and returns an `int`, or as we say for short "a function from `int` to `int`"), and since it's a function we can *call* it just as we would call `increment`.

Since values and variables can be passed into functions, one can pass variables of "function type" into functions as well:

```
let increment(n: int) : int = n + 1
let doer_thinger(f: int -> int, n: int) : int = f(n)
doer_thinger(increment, 3)  // returns 4
```

And, just for fun, let's get rid of the unnecessary syntax so you can see how simple this program can look:

```
let increment n = n + 1
let doer_thinger f n = f n
doer_thinger increment 3  // returns 4
```

Strong types

F# is a *strongly-typed* programming language. A strongly-typed language is one that enforces data types strictly and consistently. That means that the following kinds of programs are not admissible in F#. For example, the Python program,

```
x = 1
x = "hi"
```

or the C program,

```
int x = -3;
unsigned y = x;
```

Even with all the warnings enabled, a C compiler (like `clang`), won't flinch: no errors or warnings are printed for the above program. Nevertheless, it doesn't make sense to disregard the fact that an `int` is not an `unsigned int`, because assigning `-3` to `y` dramatically changes the meaning of the value. `y` is very much not `-3` anymore[9].

Both of the above programs would be considered *incorrect* in F#, since both contain *type errors*. Neither program will compile. To convert from

[8] Most students struggle with this concept, but it is very important. If you're struggling to understand this idea, this is a great topic of discussion for class or help hours.

[9] If you know some C, try running a little experiment to see what happens.

an integer to an unsigned integer, we must explicitly convert them in F#:

```
let x: int = -3
let y: uint32 = uint32 x
```

Strong types help you avoid easy-to-make but costly mistakes.

Other features

F# has many other features, such as garbage collection (like Java), lambda expressions, pattern matching, type inference, concurrency primitives, a large, mature standard library, object-orientation, inheritance, and many other features. Don't worry if you don't know what these words mean now. We will discuss their meanings throughout the remainder of the semester.

Microsoft .NET

F# is a part of an ecosystem of languages and tools developed by Microsoft called .NET (pronounced "dot net"). Programs written in .NET are almost entirely interoperable, meaning that different parts of the same program can be written in different languages. For instance, I routinely write software that makes use of modules written C#, F#, and Visual Basic combined into a single program.

.NET is also *portable*, meaning that it can run on many computer platforms. Unless you specifically seek to write platform-specific code, .NET code can be run anywhere the .NET Common Language Runtime (CLR) is available. This language architecture is similar to, and heavily inspired by, the technology behind the Java Virtual Machine (JVM). The .NET Core CLR is available on Windows, the macOS, and Linux. Additional platforms (like Android, iOS, and FreeBSD) are supported by the open source Mono project.

We will be using the .NET Core framework on Linux for this class. If you would like to install .NET Core on your own machine, you may do so by downloading the installer[10].

[10] https://www.microsoft.com/net/download

Modularity

One feature that we will address right away is F#'s strong support for modularity. *Modules* are a way of organizing code so that similarly named functions and variables in different parts of code do not conflict. In C, libraries are imported by the C preprocessor which performs the

moral equivalent of pasting code from included libraries into a single file. As a result, it is easy to accidentally give two different function definitions the same name, a so-called *name conflict*. Name conflicts are an annoying and commonplace occurence in C. In F# and other .NET languages, name conflicts are impossible, because names are *scoped*, meaning that they only have meaning within certain boundaries. When a duplicate name appears, .NET signals that something is wrong by issuing a compilation error.[11]

[11] Compilation errors are a good thing. When they occur, the compiler is telling you that you definitely made a logic error. Learn to be friends with compiler errors.

F# has a variety of constructs available to scope names: solutions, projects, namespaces, and modules. For now, we will focus on projects.

A *project* is a unit of organization defined by .NET. A project contains a collection of source code files, all in the *same* language. A project is either a *library*, meaning that it must be called by another project, or an *application*, meaning that it has an *entry point* and can run by itself.

Creating the HelloWorld project

In order to provide some structure for our hello world program, let's generate an *application project*. Having an application packaged in this way makes it self-contained and easy to manage during the development process.

We create new F# projects using the `dotnet` command on the UNIX command line. Because `dotnet` creates a project in the existing directory, you should first create a directory for your project.

```
$ mkdir helloworld
```

Now `cd` into the directory and create the project.

```
$ cd helloworld
$ dotnet new console -lang "F#"
```

By default, the above command will generate a Hello World program.

```
// Learn more about F# at http://fsharp.org

open System

[<EntryPoint>]
let main argv =
    printfn "Hello World from F#!"
    0 // return an integer exit code
```

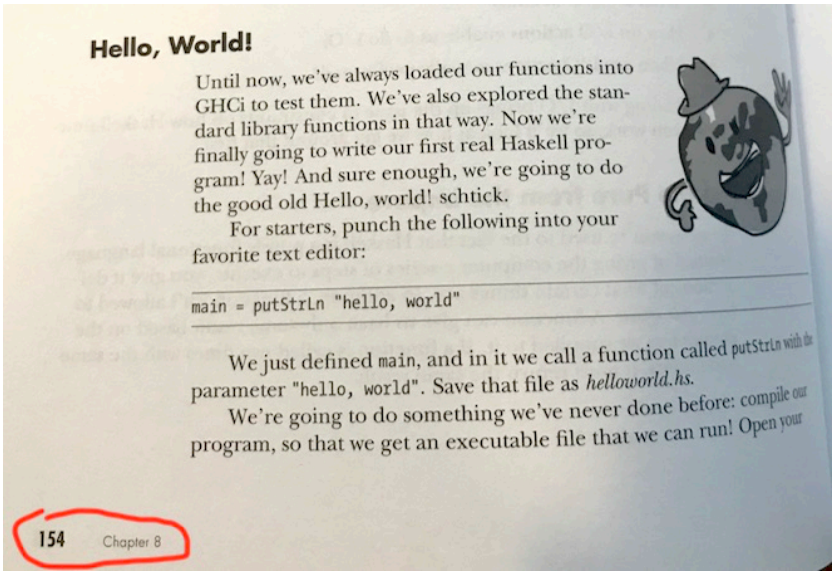
There's a little more boilerplate here than we saw when using the `dotnet fsi` REPL, and it is mostly unnecessary. But we will keep it

around because it makes working with arguments a little easier.

F# is a *whitespace sensitive* language, like Python. Whitespace sensitivity means that the scope of a function definition is determined by indentation rules instead of explicit delimeters like curly braces. Thus the last line in the above `main` function is the expression `0`. The last line of a function definition denotes the function's return value, so this function returns `0`[12].

One last thing. Since every language construct in F# is an expression, `printfn` is an expression. However, it falls into a special class of side-effecting expressions. Input and output are inherently side-effecting, so any functional language that does not allow at least some side effects is seriously constrained in terms of expressiveness. Pure functional languages like Haskell have a clever but somewhat complicated scheme for dealing with side effects. For example, the first "hello world" program in the "easy-to-read" Haskell programming book on my bookshelf appears on page 154! Pure functional languages are not easy to learn without prior exposure to some of their ideas. I chose F# for this class specifically because it is not pure; however, if you are interested, your experience in this class will make it easy to learn Haskell or other pure languages on your own.

[12] When a program's `main` function returns `0` it informs the operating system that everything went A-OK. A non-zero return value indicates a failure.



Hello, World!

Until now, we've always loaded our functions into GHCi to test them. We've also explored the standard library functions in that way. Now we're finally going to write our first real Haskell program! Yay! And sure enough, we're going to do the good old Hello, world! schtick.

For starters, punch the following into your favorite text editor:

```
main = putStrLn "hello, world"
```

We just defined `main`, and in it we call a function called `putStrLn` with the parameter `"hello, world"`. Save that file as *helloworld.hs*.

We're going to do something we've never done before: compile our program, so that we get an executable file that we can run! Open your

154 Chapter 8

Compiling and running your project

Compile your project with:

```
$ dotnet build
```

You may also just run the project, and if it needs to be built, `dotnet` will build it for you before running it.

```
$ dotnet run
```

I personally prefer to run the `build` command separately because the `run` command hides compiler output. I like to see compiler output since it will inform me when it finds problems with my program. Unlike other languages you may have used, F#'s compiler generally produces very good error messages.

Code editors

You are welcome to use whatever code editor you wish in this class. Two in particular stand out for F#, however: Visual Studio Code and `emacs`. Both are installed on our lab machines. Note, however, that we will strictly manage our projects using the `dotnet` command line tool.

Visual Studio Code

Visual Studio Code works out of the box with F#, but an extension called Ionide[13] adds additional features like syntax highlighting and tooltips to your editor. To install Ionide, follow this tutorial on installing extensions[14].

Note: Ionide comes with a variety of build tools such as FAKE, Forge, Paket, and project scaffolds. Please do not use these tools for this class as they do not interoperate well with our class environment. Instead, please use the `dotnet` command line tool to compile and run your tool as discussed earler.

[13] `http://ionide.io/`

[14] `https://code.visualstudio.com/docs/editor/extension-gallery`

emacs

If you prefer `emacs`, you can add the `fsharp-mode` which adds syntax highlighting, tooltips, and a variety of other nice features. I personally prefer this environment, but I understand that `emacs` is not everybody's cup of tea.

If using `emacs` on a lab machine, try pasting the following into `~/.local_emacs`:[15]

[15] On your personal machine, use `~/.emacs` instead.

```
(require 'package)
(add-to-list
  'package-archives
  '("melpa" . "http://melpa.org/packages/"))
(unless package-archive-contents (package-refresh-contents))
(package-initialize)

(unless (package-installed-p 'fsharp-mode)
  (package-install 'fsharp-mode))
(require 'fsharp-mode)
```

The above will install both MELPA, which is an online package repository for `emacs`, and the `fsharp-mode` package. Note that MELPA has many other modes you can install if you like what you see. One downside to MELPA is that it adds a few seconds of startup time to `emacs`, but in my opinion, the delay is well worth the wait.

The next time you start `emacs` with F# code, you will see the new mode in action.

Beating the Averages

By Paul Graham. Originally published in April, 2001. Revised in April, 2003. This article is derived from a talk given at the 2001 Franz Developer Symposium.

In the summer of 1995, my friend Robert Morris and I started a startup called Viaweb. Our plan was to write software that would let end users build online stores. What was novel about this software, at the time, was that it ran on our server, using ordinary Web pages as the interface.

A lot of people could have been having this idea at the same time, of course, but as far as I know, Viaweb was the first Web-based application. It seemed such a novel idea to us that we named the company after it: Viaweb, because our software worked via the Web, instead of running on your desktop computer.

Another unusual thing about this software was that it was written primarily in a programming language called Lisp. It was one of the first big end-user applications to be written in Lisp, which up till then had been used mostly in universities and research labs.[16]

[16] Viaweb at first had two parts: the editor, written in Lisp, which people used to build their sites, and the ordering system, written in C, which handled orders. The first version was mostly Lisp, because the ordering system was small. Later we added two more modules, an image generator written in C, and a back-office manager written mostly in Perl.

In January 2003, Yahoo released a new version of the editor written in C++ and Perl. It's hard to say whether the program is no longer written in Lisp, though, because to translate this program into C++ they literally had to write a Lisp interpreter: the source files of all the page-generating templates are still, as far as I know, Lisp code. (See Greenspun's Tenth Rule.)

The Secret Weapon

Eric Raymond has written an essay called "How to Become a Hacker," and in it, among other things, he tells would-be hackers what languages they should learn. He suggests starting with Python and Java, because they are easy to learn. The serious hacker will also want to learn C, in order to hack Unix, and Perl for system administration and cgi scripts. Finally, the truly serious hacker should consider learning Lisp:

> Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

This is the same argument you tend to hear for learning Latin. It won't get you a job, except perhaps as a classics professor, but it will

improve your mind, and make you a better writer in languages you do want to use, like English.

But wait a minute. This metaphor doesn't stretch that far. The reason Latin won't get you a job is that no one speaks it. If you write in Latin, no one can understand you. But Lisp is a computer language, and computers speak whatever language you, the programmer, tell them to.

So if Lisp makes you a better programmer, like he says, why wouldn't you want to use it? If a painter were offered a brush that would make him a better painter, it seems to me that he would want to use it in all his paintings, wouldn't he? I'm not trying to make fun of Eric Raymond here. On the whole, his advice is good. What he says about Lisp is pretty much the conventional wisdom. But there is a contradiction in the conventional wisdom: Lisp will make you a better programmer, and yet you won't use it.

Why not? Programming languages are just tools, after all. If Lisp really does yield better programs, you should use it. And if it doesn't, then who needs it?

This is not just a theoretical question. Software is a very competitive business, prone to natural monopolies. A company that gets software written faster and better will, all other things being equal, put its competitors out of business. And when you're starting a startup, you feel this very keenly. Startups tend to be an all or nothing proposition. You either get rich, or you get nothing. In a startup, if you bet on the wrong technology, your competitors will crush you.

Robert and I both knew Lisp well, and we couldn't see any reason not to trust our instincts and go with Lisp. We knew that everyone else was writing their software in C++ or Perl. But we also knew that that didn't mean anything. If you chose technology that way, you'd be running Windows. When you choose technology, you have to ignore what other people are doing, and consider only what will work the best.

This is especially true in a startup. In a big company, you can do what all the other big companies are doing. But a startup can't do what all the other startups do. I don't think a lot of people realize this, even in startups.

The average big company grows at about ten percent a year. So if you're running a big company and you do everything the way the average big company does it, you can expect to do as well as the average big company—that is, to grow about ten percent a year.

The same thing will happen if you're running a startup, of course. If you do everything the way the average startup does it, you should expect average performance. The problem here is, average performance means that you'll go out of business. The survival rate for startups is way less than fifty percent. So if you're running a startup, you had better be doing something odd. If not, you're in trouble.

Back in 1995, we knew something that I don't think our competitors understood, and few understand even now: when you're writing software that only has to run on your own servers, you can use any language you want. When you're writing desktop software, there's a strong bias toward writing applications in the same language as the operating system. Ten years ago, writing applications meant writing applications in C. But with Web-based software, especially when you have the source code of both the language and the operating system, you can use whatever language you want.

This new freedom is a double-edged sword, however. Now that you can use any language, you have to think about which one to use. Companies that try to pretend nothing has changed risk finding that their competitors do not.

If you can use any language, which do you use? We chose Lisp. For one thing, it was obvious that rapid development would be important in this market. We were all starting from scratch, so a company that could get new features done before its competitors would have a big advantage. We knew Lisp was a really good language for writing software quickly, and server-based applications magnify the effect of rapid development, because you can release software the minute it's done.

If other companies didn't want to use Lisp, so much the better. It might give us a technological edge, and we needed all the help we could get. When we started Viaweb, we had no experience in business. We didn't know anything about marketing, or hiring people, or raising money, or getting customers. Neither of us had ever even had what you would call a real job. The only thing we were good at was writing software. We hoped that would save us. Any advantage we could get in the software department, we would take.

So you could say that using Lisp was an experiment. Our hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower. If this were so, we could offer a better product for less money, and still make a profit. We would end up getting all the users, and our competitors would get none, and eventually go out of business. That was what we hoped would happen, anyway.

What were the results of this experiment? Somewhat surprisingly, it worked. We eventually had many competitors, on the order of twenty to thirty of them, but none of their software could compete with ours. We had a wysiwyg online store builder that ran on the server and yet felt like a desktop application. Our competitors had cgi scripts. And we were always far ahead of them in features. Sometimes, in desperation, competitors would try to introduce features that we didn't have. But

with Lisp our development cycle was so fast that we could sometimes duplicate a new feature within a day or two of a competitor announcing it in a press release. By the time journalists covering the press release got round to calling us, we would have the new feature too.

It must have seemed to our competitors that we had some kind of secret weapon—that we were decoding their Enigma traffic or something. In fact we did have a secret weapon, but it was simpler than they realized. No one was leaking news of their features to us. We were just able to develop software faster than anyone thought possible.

When I was about nine I happened to get hold of a copy of *The Day of the Jackal*, by Frederick Forsyth. The main character is an assassin who is hired to kill the president of France. The assassin has to get past the police to get up to an apartment that overlooks the president's route. He walks right by them, dressed up as an old man on crutches, and they never suspect him.

Our secret weapon was similar. We wrote our software in a weird AI language, with a bizarre syntax full of parentheses. For years it had annoyed me to hear Lisp described that way. But now it worked to our advantage. In business, there is nothing more valuable than a technical advantage your competitors don't understand. In business, as in war, surprise is worth as much as force.

And so, I'm a little embarrassed to say, I never said anything publicly about Lisp while we were working on Viaweb. We never mentioned it to the press, and if you searched for Lisp on our Web site, all you'd find were the titles of two books in my bio. This was no accident. A startup should give its competitors as little information as possible. If they didn't know what language our software was written in, or didn't care, I wanted to keep it that way.[17]

The people who understood our technology best were the customers. They didn't care what language Viaweb was written in either, but they noticed that it worked really well. It let them build great looking online stores literally in minutes. And so, by word of mouth mostly, we got more and more users. By the end of 1996 we had about 70 stores online. At the end of 1997 we had 500. Six months later, when Yahoo bought us, we had 1070 users. Today, as Yahoo Store, this software continues to dominate its market. It's one of the more profitable pieces of Yahoo, and the stores built with it are the foundation of Yahoo Shopping. I left Yahoo in 1999, so I don't know exactly how many users they have now, but the last I heard there were about 20,000.

[17] Robert Morris says that I didn't need to be secretive, because even if our competitors had known we were using Lisp, they wouldn't have understood why: "If they were that smart they'd already be programming in Lisp."

The Blub Paradox

What's so great about Lisp? And if Lisp is so great, why doesn't everyone use it? These sound like rhetorical questions, but actually they have straightforward answers. Lisp is so great not because of some magic quality visible only to devotees, but because it is simply the most powerful language available. And the reason everyone doesn't use it is that programming languages are not merely technologies, but habits of mind as well, and nothing changes slower. Of course, both these answers need explaining.

I'll begin with a shockingly controversial statement: programming languages vary in power.

Few would dispute, at least, that high level languages are more powerful than machine language. Most programmers today would agree that you do not, ordinarily, want to program in machine language. Instead, you should program in a high-level language, and have a compiler translate it into machine language for you. This idea is even built into the hardware now: since the 1980s, instruction sets have been designed for compilers rather than human programmers.

Everyone knows it's a mistake to write your whole program by hand in machine language. What's less often understood is that there is a more general principle here: that if you have a choice of several languages, it is, all other things being equal, a mistake to program in anything but the most powerful one.[18]

There are many exceptions to this rule. If you're writing a program that has to work very closely with a program written in a certain language, it might be a good idea to write the new program in the same language. If you're writing a program that only has to do something very simple, like number crunching or bit manipulation, you may as well use a less abstract language, especially since it may be slightly faster. And if you're writing a short, throwaway program, you may be better off just using whatever language has the best library functions for the task. But in general, for application software, you want to be using the most powerful (reasonably efficient) language you can get, and using anything else is a mistake, of exactly the same kind, though possibly in a lesser degree, as programming in machine language.

You can see that machine language is very low level. But, at least as a kind of social convention, high-level languages are often all treated as equivalent. They're not. Technically the term "high-level language" doesn't mean anything very definite. There's no dividing line with machine languages on one side and all the high-level languages on the other. Languages fall along a continuum[19] of abstractness, from the most powerful all the way down to machine languages, which themselves vary in power.

[18] All languages are equally powerful in the sense of being Turing equivalent, but that's not the sense of the word programmers care about. (No one wants to program a Turing machine.) The kind of power programmers care about may not be formally definable, but one way to explain it would be to say that it refers to features you could only get in the less powerful language by writing an interpreter for the more powerful language in it. If language A has an operator for removing spaces from strings and language B doesn't, that probably doesn't make A more powerful, because you can probably write a subroutine to do it in B. But if A supports, say, recursion, and B doesn't, that's not likely to be something you can fix by writing library functions.

[19] Note to nerds: or possibly a lattice, narrowing toward the top; it's not the shape that matters here but the idea that there is at least a partial order.

Consider Cobol. Cobol is a high-level language, in the sense that it gets compiled into machine language. Would anyone seriously argue that Cobol is equivalent in power to, say, Python? It's probably closer to machine language than Python.

Or how about Perl 4? Between Perl 4 and Perl 5, lexical closures got added to the language. Most Perl hackers would agree that Perl 5 is more powerful than Perl 4. But once you've admitted that, you've admitted that one high level language can be more powerful than another. And it follows inexorably that, except in special cases, you ought to use the most powerful you can get.

This idea is rarely followed to its conclusion, though. After a certain age, programmers rarely switch languages voluntarily. Whatever language people happen to be used to, they tend to consider just good enough.

Programmers get very attached to their favorite languages, and I don't want to hurt anyone's feelings, so to explain this point I'm going to use a hypothetical language called Blub. Blub falls right in the middle of the abstractness continuum. It is not the most powerful language, but it is more powerful than Cobol or machine language.

And in fact, our hypothetical Blub programmer wouldn't use either of them. Of course he wouldn't program in machine language. That's what compilers are for. And as for Cobol, he doesn't know how anyone can get anything done with it. It doesn't even have x (Blub feature of your choice).

As long as our hypothetical Blub programmer is looking down the power continuum, he knows he's looking down. Languages less powerful than Blub are obviously less powerful, because they're missing some feature he's used to. But when our hypothetical Blub programmer looks in the other direction, up the power continuum, he doesn't realize he's looking up. What he sees are merely weird languages. He probably considers them about equivalent in power to Blub, but with all this other hairy stuff thrown in as well. Blub is good enough for him, because he thinks in Blub.

When we switch to the point of view of a programmer using any of the languages higher up the power continuum, however, we find that he in turn looks down upon Blub. How can you get anything done in Blub? It doesn't even have y.

By induction, the only programmers in a position to see all the differences in power between the various languages are those who understand the most powerful one. (This is probably what Eric Raymond meant about Lisp making you a better programmer.) You can't trust the opinions of the others, because of the Blub paradox: they're satisfied with whatever language they happen to use, because it dictates the way they think about programs.

I know this from my own experience, as a high school kid writing programs in Basic. That language didn't even support recursion. It's hard to imagine writing programs without using recursion, but I didn't miss it at the time. I thought in Basic. And I was a whiz at it. Master of all I surveyed.

The five languages that Eric Raymond recommends to hackers fall at various points on the power continuum. Where they fall relative to one another is a sensitive topic. What I will say is that I think Lisp is at the top. And to support this claim I'll tell you about one of the things I find missing when I look at the other four languages. How can you get anything done in them, I think, without macros?[20]

[20] It is a bit misleading to treat macros as a separate feature. In practice their usefulness is greatly enhanced by other Lisp features like lexical closures and rest parameters.

Many languages have something called a macro. But Lisp macros are unique. And believe it or not, what they do is related to the parentheses. The designers of Lisp didn't put all those parentheses in the language just to be different. To the Blub programmer, Lisp code looks weird. But those parentheses are there for a reason. They are the outward evidence of a fundamental difference between Lisp and other languages.

Lisp code is made out of Lisp data objects. And not in the trivial sense that the source files contain characters, and strings are one of the data types supported by the language. Lisp code, after it's read by the parser, is made of data structures that you can traverse.

If you understand how compilers work, what's really going on is not so much that Lisp has a strange syntax as that Lisp has no syntax. You write programs in the parse trees that get generated within the compiler when other languages are parsed. But these parse trees are fully accessible to your programs. You can write programs that manipulate them. In Lisp, these programs are called macros. They are programs that write programs.

Programs that write programs? When would you ever want to do that? Not very often, if you think in Cobol. All the time, if you think in Lisp. It would be convenient here if I could give an example of a powerful macro, and say there! how about that? But if I did, it would just look like gibberish to someone who didn't know Lisp; there isn't room here to explain everything you'd need to know to understand what it meant. In *ANSI Common Lisp* I tried to move things along as fast as I could, and even so I didn't get to macros until page 160.

But I think I can give a kind of argument that might be convincing. The source code of the Viaweb editor was probably about 20-25% macros. Macros are harder to write than ordinary Lisp functions, and it's considered to be bad style to use them when they're not necessary. So every macro in that code is there because it has to be. What that means is that at least 20-25% of the code in this program is doing things that you can't easily do in any other language. However skeptical the Blub programmer might be about my claims for the mysterious pow-

ers of Lisp, this ought to make him curious. We weren't writing this code for our own amusement. We were a tiny startup, programming as hard as we could in order to put technical barriers between us and our competitors.

A suspicious person might begin to wonder if there was some correlation here. A big chunk of our code was doing things that are very hard to do in other languages. The resulting software did things our competitors' software couldn't do. Maybe there was some kind of connection. I encourage you to follow that thread. There may be more to that old man hobbling along on his crutches than meets the eye.

Aikido for Startups

But I don't expect to convince anyone (over 25) to go out and learn Lisp. The purpose of this article is not to change anyone's mind, but to reassure people already interested in using Lisp—people who know that Lisp is a powerful language, but worry because it isn't widely used. In a competitive situation, that's an advantage. Lisp's power is multiplied by the fact that your competitors don't get it.

If you think of using Lisp in a startup, you shouldn't worry that it isn't widely understood. You should hope that it stays that way. And it's likely to. It's the nature of programming languages to make most people satisfied with whatever they currently use. Computer hardware changes so much faster than personal habits that programming practice is usually ten to twenty years behind the processor. At places like MIT they were writing programs in high-level languages in the early 1960s, but many companies continued to write code in machine language well into the 1980s. I bet a lot of people continued to write machine language until the processor, like a bartender eager to close up and go home, finally kicked them out by switching to a RISC instruction set.

Ordinarily technology changes fast. But programming languages are different: programming languages are not just technology, but what programmers think in. They're half technology and half religion.[21] And so the median language, meaning whatever language the median programmer uses, moves as slow as an iceberg. Garbage collection, introduced by Lisp in about 1960, is now widely considered to be a good thing. Runtime typing, ditto, is growing in popularity. Lexical closures, introduced by Lisp in the early 1970s, are now, just barely, on the radar screen. Macros, introduced by Lisp in the mid 1960s, are still terra incognita.

Obviously, the median language has enormous momentum. I'm not

[21] As a result, comparisons of programming languages either take the form of religious wars or undergraduate textbooks so determinedly neutral that they're really works of anthropology. People who value their peace, or want tenure, avoid the topic. But the question is only half a religious one; there is something there worth studying, especially if you want to design new languages.

proposing that you can fight this powerful force. What I'm proposing is exactly the opposite: that, like a practitioner of Aikido, you can use it against your opponents.

If you work for a big company, this may not be easy. You will have a hard time convincing the pointy-haired boss to let you build things in Lisp, when he has just read in the paper that some other language is poised, like Ada was twenty years ago, to take over the world. But if you work for a startup that doesn't have pointy-haired bosses yet, you can, like we did, turn the Blub paradox to your advantage: you can use technology that your competitors, glued immovably to the median language, will never be able to match.

If you ever do find yourself working for a startup, here's a handy tip for evaluating competitors. Read their job listings. Everything else on their site may be stock photos or the prose equivalent, but the job listings have to be specific about what they want, or they'll get the wrong candidates.

During the years we worked on Viaweb I read a lot of job descriptions. A new competitor seemed to emerge out of the woodwork every month or so. The first thing I would do, after checking to see if they had a live online demo, was look at their job listings. After a couple years of this I could tell which companies to worry about and which not to. The more of an IT flavor the job descriptions had, the less dangerous the company was. The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. If they wanted Perl or Python programmers, that would be a bit frightening—that's starting to sound like a company where the technical side, at least, is run by real hackers. If I had ever seen a job posting looking for Lisp hackers, I would have been really worried.

A Slightly Longer Introduction to F#

This tutorial goes deeper into the F# language. As before, you are strongly encouraged to follow along on your computer.

Let's look at a very simple F# program in *source code* form.

```
[<EntryPoint>]
let main argv =
  printfn "Hello, %s!" argv[0]
  0
```

Type this program into an editor and save it with the name `helloworld.fs`. I recommend typing the program instead of copying-and-pasting it because retyping it will force you to notice important details about the program.

Hopefully it's not too much of a stretch to figure out what this program does! We will look at this program line-by-line to understand what its parts are, but first, let's understand how to run this program.

The F# Compiler

As with any other programming language, your computer cannot understand an F# program in source code form. It must be translated into another form. Unlike a language like C, however, we do not translate F# directly into an executable binary. Instead, the F# compiler, called `fsharpc`, converts F# source code into an *architecture independent* form. Architecture independence means that the resulting compiled program can be *run on any computer*: a personal computer, a cellphone, a supercomputer, a watch, or even an embedded computer (like the kind in "smart lightbulbs").

Architecture independence is achieved by building the language on top of a *virtual machine*[22]. A virtual machine provides an abstraction that hides many of the quirks present in real computers.

Building a language on top of a virtual machine greatly simplifies the process of converting software to run on other computer systems. Altering a program so that it will run on another computer system is

[22] Specifically, the language uses a *process virtual machine*. Unlike a *fully virtual machine*, which is intended to mimic an actual processor along with all its quirks, a process virtual machine is even simpler since it does not need to accurately emulate computer hardware.

called *porting*. In the past, programmers were responsible for altering programs to run on other computer systems themselves. With the virtual machine approach, the burden of producing portable software is shifted to the language designer. Language designers typically have a much better understanding of code portability concerns than a typical programmer.

Since you are likely more familiar with the Java programming language than with F#, it's worth knowing that Java uses the same approach as F#: `javac` produces Java byte code, which is then interpreted by the Java Virtual Machine (JVM). Since each computer platform has its own `java` interpreter, that compiled program can be written once and run anywhere. "Write once, run anywhere" was the marketing slogan used by Sun Microsystems to sell the Java programming language in the mid 1990's.[23]

[23] It should be noted that neither Microsoft nor Sun Microsystems invented the idea of portable bytecode. That honor appears to go to Martin Richards, who came up with *O-code* to make it easier to port the BCPL language in the mid-1960's.

Here is a (snippet of) `fsharpc`'s translation of our hello world program into VM *byte code*, which is the virtual machine's native language. The byte code shown below is in a dialect called the Common Intermediate Language (CIL), and runs on Microsoft's VM, the Common Language Runtime (CLR).

```
0000000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00
0000010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
0000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000030 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00
0000040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68
0000050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
0000060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20
0000070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00
0000080 50 45 00 00 4c 01 03 00 da 06 ca 5b 00 00 00 00
0000090 00 00 00 00 e0 00 0e 01 0b 01 08 00 00 08 00 00
...
```

Note that you do not need to know CIL or any other assembly language in this class. I'm just showing this to you to give you some perspective. If you do have some familiarity with x86 or ARM machine-level programming, CIL byte code might look similar. However, the format is quite different. I show the same information below converted into a human-readable form using CIL's instruction mnemonics (i.e., "assembly code"). If you know a little assembly already, you might notice that the CIL's mnemonics are much more expressive.[24]

[24] We will talk about the expressiveness of languages throughout the semester. For now, think of the distinction as like the difference between a simple story and great literature.

```
.class public abstract sealed auto ansi
  Program
    extends [mscorlib]System.Object
{
  .custom instance void [FSharp.Core]Microsoft.FSharp.Core.
      CompilationMappingAttribute::.ctor(valuetype [FSharp.Core]
```

```
      Microsoft.FSharp.Core.SourceConstructFlags)
    = (01 00 07 00 00 00 00 00 ) // ........
    // int32(7) // 0x00000007

  .method public static int32
    main(
      string[] argv
    ) cil managed
  {
    .entrypoint
    .custom instance void [FSharp.Core]Microsoft.FSharp.Core.
        EntryPointAttribute::.ctor()
      = (01 00 00 00 )
    .maxstack 4
    .locals init (
      [0] class [FSharp.Core]Microsoft.FSharp.Core.FSharpFunc`2<
          string, class [FSharp.Core]Microsoft.FSharp.Core.Unit>
           V_0,
      [1] string V_1
    )

    // [3 5 - 3 25]
    IL_0000: ldstr        "Hello, %s!"
    IL_0005: newobj       instance void class [FSharp.Core]
        Microsoft.FSharp.Core.PrintfFormat`5<class [FSharp.Core]
        Microsoft.FSharp.Core.FSharpFunc`2<string, class [FSharp
        .Core]Microsoft.FSharp.Core.Unit>, class [mscorlib]
        System.IO.TextWriter, class [FSharp.Core]Microsoft.
        FSharp.Core.Unit, class [FSharp.Core]Microsoft.FSharp.
        Core.Unit, string>::.ctor(string)
    IL_000a: call         !!0/*class [FSharp.Core]Microsoft.
        FSharp.Core.FSharpFunc`2<string, class [FSharp.Core]
        Microsoft.FSharp.Core.Unit>*/ [FSharp.Core]Microsoft.
        FSharp.Core.ExtraTopLevelOperators::PrintFormatLine<
        class [FSharp.Core]Microsoft.FSharp.Core.FSharpFunc`2<
        string, class [FSharp.Core]Microsoft.FSharp.Core.Unit>>(
        class [FSharp.Core]Microsoft.FSharp.Core.PrintfFormat
        `4<!!0/*class [FSharp.Core]Microsoft.FSharp.Core.
        FSharpFunc`2<string, class [FSharp.Core]Microsoft.FSharp
        .Core.Unit>*/, class [mscorlib]System.IO.TextWriter,
        class [FSharp.Core]Microsoft.FSharp.Core.Unit, class [
        FSharp.Core]Microsoft.FSharp.Core.Unit>)
    IL_000f: stloc.0      // V_0
    IL_0010: ldarg.0      // argv
    IL_0011: ldc.i4.0
    IL_0012: ldelem       [mscorlib]System.String
    IL_0017: stloc.1      // V_1
    IL_0018: ldloc.0      // V_0
    IL_0019: ldloc.1      // V_1
    IL_001a: callvirt     instance !1/*class [FSharp.Core]
        Microsoft.FSharp.Core.Unit*/ class [FSharp.Core]
        Microsoft.FSharp.Core.FSharpFunc`2<string, class [FSharp
        .Core]Microsoft.FSharp.Core.Unit>::Invoke(!0/*string*/)
    IL_001f: pop

    // [4 5 - 4 6]
    IL_0020: ldc.i4.0
    IL_0021: ret

  } // end of method Program::main
} // end of class Program
```

Ordinary assembly code is very primitive; it has no notion of data types, classes, methods, or other concepts from programming language. The CLR, on the other hand, *does* know about these things. For example see if you can find the text `.method public static int32 main` in the CIL program above. That section looks a lot like a Java method, doesn't it?

History

F# was developed by a research group at Microsoft Research, led by Don Syme (Figure 3). Although F# has some novel features, particularly the ways in which it interoperates with other .NET code, its syntax[25] and semantics[26] are largely inspired by the ML family of programming languages. Syntactically, F# added whitespace-sensitivity (like Python) and "lightweight" refinements of older ML syntax that make it more pleasant to use. If you like Python, you'll probably like F#.

ML was designed by researchers at the University of Edinburgh, most notably Robin Milner (Figure 4) and Luca Cardelli (Figure 5), in the early 1970's. ML stands for "meta language," because it was originally designed to be a meta language for writing "proof tactics" (you can think of these as search procedures) for the LCF automated theorem prover. Although ML was heavily inspired by mathematical logic and early functional programming languages like LISP[27], its authors made a concerted effort early on to create something "elegant." But what makes ML especially interesting is that the language design was not static. Milner was inspired by other programming language research happening concurrently at Edinburgh, notably the HOPE programming language. ML borrowed many ideas from these other languages whenever a feature made the language feel simpler or more elegant to its authors. For example, pattern matching, which is a feature widely enjoyed by ML users originally came from HOPE.

I enjoy reading ML's early design documents, because it is clear that the most important thing was to build a "simple and well-understood" language. ML was also one of the first languages to have a complete formal specification. Nonetheless, ML has a strong pragmatic streak that makes it—in my opinion—a lot more fun to program in than other programming languages.

ML quickly outgrew its origins in the LCF project and was used widely among academics starting in the late 1970's. "Standard ML" (SML) arose in the 1980's out of a desire to obtain many interoperable implementations of ML. One of the most popular versions of SML is the "Stan-

[25] *Syntax* defines the *appearance* of a programming language.

[26] *Semantics* defines the *meaning* of a programming language.

Figure 3: Don Syme.

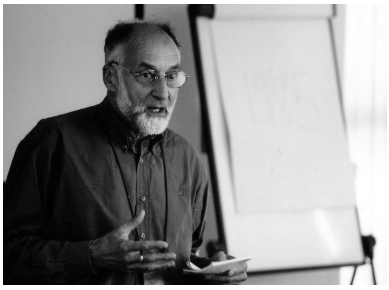


Figure 4: Robin Milner.

Figure 5: Luca Cardelli.

[27] The first version of ML was written in LISP!

dard ML of New Jersey" (SML/NJ) implementation that was jointly developed by Bell Labs and Princeton University, both of which are in New Jersey. Our lab machines have `smlnj` interpreters installed on them, so if you're curious about the differences between F# and SML, have a look.

Compiling using `dotnet`

If you've programmed using Microsoft Windows before, you may have used the Visual Studio IDE[28]. Visual Studio is the *de facto* code editor for commercial F# programmers. You are welcome to use Visual Studio if you have it, but as it is quite expensive, I do not require it for this class. Instead, we will be using the cross-platform `dotnet` tool, which runs the F# compiler for us.

[28] Confusingly, "Visual Studio" is a completely different product than "Visual Studio Code."

MSBuild

Manually managing the F# compiler can get a little annoying in the same way that managing `javac` or `gcc` or `gcc` can be annoying. Therefore, this class asks you to produce code as a part of an *MSBuild project*. MSBuild is Microsoft's (much more sophisticated) equivalent to C's Makefiles. You should have first encountered `dotnet` in A Brief Introduction to F#.

Because MSBuild projects are written in XML, we will mostly create and manage our projects using a command line tool called `dotnet`. This tool automatically generates and edits MSBuild files for you. However, we will occasionally modify MSBuild files by hand so that you understand what they mean.

New projects

We create new projects using the `dotnet new` command. Typing this command without arguments will show you a set of project templates that you can use to create a new project. For this class, we will mostly use the command `dotnet new console -lang F#` which creates a new project for a command-line program using F# as the language.

Note that `dotnet new` creates a new project in the current directory. Be careful to avoid putting your code in a location that already has code as the effect may not be what you intend.

Building

To build a project, `cd` to the directory containing your project files and type

```
$ dotnet build
```

Running

To run a project, `cd` to the directory containing your project files and type

```
$ dotnet run
```

Adding a new file to a project

By default, your project contains only a single file called `Program.fs`. Unlike Java, F# does not care where you put your code. It could all go into a single source code file.

However, as your projects grow in size, you will find it beneficial to organize your code across multiple files. I like to organize my code according to "responsibilities." For example, maybe I have a program that reads input, does some processing, builds a data structure, computes some values, and then prints out the result. In this case, I might have a file called `io.fs` for input and output processing, `utils.fs` to handle data manipulation (like converting data from arrays into hash tables), and `algorithms.fs` for the core computation. I personally like to keep very little in the `Program.fs` file, which mostly just contains the `main` function. Unless I ask you to organize your code in a specific manner, use whatever system of organization makes sense to you.

To add a new file to your project, you need to do two things. Suppose we create a new file called `io.fs` and we want to call its code from the `Program.fs` file. Look for a `.fsproj` file in your project directory. This is your project specification. Open it up with your favorite code editor. You should see something like

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="Program.fs" />
  </ItemGroup>

</Project>
```

We need to add a `Compile` tag just above the `Compile` tag for `Program.fs` so that MSBuild will compile `io.fs` first. Here's what my `.fsproj` file looks like after I make the change:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="io.fs" />
    <Compile Include="Program.fs" />
  </ItemGroup>

</Project>
```

Running `dotnet build` should now allow `Program.fs` to refer to code stored in `io.fs`. Note that if you're following along at home, you will likely see the following error when you try the above.

```
error FS0222: Files in libraries or multiple-file applications must
              begin with a namespace or module declaration
```

What's the deal? In short, in F#, you must place all library code inside either a *module* or a *namespace*. Both of these two things are a form of code organization. We'll stick to modules for this course, as namespaces are only slightly different and are not really necessary unless you are mixing F# and C# code (C# has no notion of modules, only namespaces).

Put a module declaration at the top of your module to make this er-

ror message go away. For example, in `io.fs`, put:

```
module IO
```

and in your `Program.fs`, write

```
open IO
```

"This means something. This is important." Understanding code you wrote.

Figure 6: "This means something. This is important." If you don't get the reference, your homework is to gather your friends together and watch *Close Encounters of the Third Kind*.

As a new CS student, you've probably used code that you don't understand. Doing so is a bad habit. Whenever you borrow code from somebody, you really should make the effort to understand it. Let's *understand* the `helloworld.fs` program we typed in at the beginning of this reading, line by line.

Entry points

The first line,

```
[<EntryPoint>]
```

marks the function as the entry point to the program. The entry point is the location in the program where computation begins. In Java, the `main`

function is always the entry point. F# gives you a bit more flexibility: it can be any single function, as long as that one function is labeled with the `[<EntryPoint>]` annotation.

Function definitions

The next line,

```
let main argv =
```

denotes the start of a *function definition*. Unlike C and Java, F# is whitespace-sensitive, like Python. In F# code must be indented using spaces (not tabs: F# is an *opinionated* language, meaning that code style is enforced in the language). The body of the function definition begins at the `=` character and extends until the end of the indented region below.

Note that, unlike most languages you've studied so far, F# functions are *true functions*, meaning that they must *always return a value*. While side-effecting functions are possible in F#, they are strongly discouraged, and you have do some extra things to use them (like using the `mutable` annotation). Thus you are encouraged to write *pure functions*. In this class, you should assume that we will be writing pure functions unless otherwise specified.

This function definition looks sparse, doesn't it? In fact, despite the fact that F# is a *statically typed* programming language, there are no type annotations in the above. That's because F# can usually *infer* the types of expressions without your help.

In F#, declarations of all kinds start with the keyword `let`. In general, you should assume that `let` simply means that we should bind the expression to the right of the `=` to the name on the left of the `=`.

If you've played with F# a bit you might be thinking, "wait, variables and functions are declared the same way in F#?" Indeed they are. The expression

```
let main argv = ...
```

declares a function called `main` with a single argument called `argv`, bound to the expression on the right, and

```
let x = 1
```

declares a variable called `x` bound to the value `1`. The way that F# knows the first example is a function and not just a variable is because the part of the expression to the left of the `=` sign has an argument (i.e., `argv`). In this example, we declare a function `main` with a single argument,

`argv`. Since you're new to F#, you may be thinking "how am I supposed to know what type `argv` is?" This is admittedly one of the downsides of type inference; type information is not obvious from the program text alone. That said, unlike a dynamically typed language like Python, when you get the type of an expression wrong, the F# compiler tells you.

Suppose for a moment that my `main` function was:

```
let main argv =
    argv + 1
```

Then F# would report,

```
error FS0001: The type 'int' does not match the type 'string []'
```

and I would not be able to run the program. Because the type check fails—`argv` is a `string[]`, not an `int`—compilation also fails. A compilation failure is a *feature* of a statically-typed language, because it enables you to find bugs in your program *before you run it*.

You can also add type annotations yourself, and if you are at all unsure what the types of various things are, I encourage you to write them. Let's rewrite our `main` function with types.

```
let main(argv: string[]) : int =
    printfn "Hello, %s!" argv[0]
    0
```

The syntax of a typed function in F# is the following:

```
let <function name> (<arg_1>: <type_1>) ... (<arg_n>: <type_n>) : <return type> =
        <expression>
```

It's up to you how you want to write your programs. F# doesn't care, and neither do I. I encourage you to try out the parens-less syntax, however, as once you are accustomed to it, you will find F# programs very easy to read.

Function body

The meat of our `main` function is the following:

```
    printfn "Hello, %s!" argv[0]
    0
```

Notice that this code is indented from `main`. The indentation is how we know that the code is a part of the `main` function definition. My personal convention is to use 4 spaces. Others use 2. Again, choose what you like,

but note that the F# compiler *will not* let you use tabs.

The first line *calls* the `printfn` function. Function calls in F# work *exactly* the same way as "application" in the lambda calculus, which we will discuss in detail in this class. However, the important thing to know for now is that an expression of the form

```
a b
```

means that we should *call* the function `a` with the argument `b`. The above is actually valid F#. Here's an example that might make more sense to you:

```
let b = 1
let a x = x + 1
a b
```

which returns 2. Type it into `dotnet fsi` if you don't believe me.

Function types

Let's talk a little about function types. When you type an expression into `dotnet fsi`, it will print that expression's type. The `->` type notation tells us that something is a function. So, for instance,

```
let a x = x + 1
```

has type

```
int -> int
```

because it is a *function* that *takes* an `int` and *returns* an `int`. You can tell that a type of an expression is a function whenever you see the `->` in its type.

By the way, when we put the above function `a` into `dotnet fsi`, it actually prints out the following type:

```
> let a x = x + 1;;
val a : x:int -> int
```

Try not to be confused by the extra output. F# is trying to be helpful by including names along with the types. The entire expression is called `a`. This makes sense, because we asked F# to name the entire expression `a` by using the `let` keyword. Since the entire expression has a `->` in it, we know it's a function. The stuff on the left side of `->` is the type of the function's argument. The stuff on the right side of `->` is the type of the

function's return value. Therefore, the type of the function's argument, `x`, is `int`. Finally, the return value has type `int`.

Polymorphic functions

F# has a very flexible model for polymorphism. *Polymorphic* code is code that works for different types of data. You've seen polymorphism before. Java generics are a kind of polymorphism. For example, we know that linked lists work equally well for integers and strings, so Java lets us write:

```
List<Integer> x = new List<>();
```

for an integer, or

```
List<String> y = new List<>();
```

for a string. Nevertheless, but we only need to create a single `List` implementation because `List` can be made generic.

In F#, polymorphic types are known as *tick variables* because they are prefaced by the single quote character, `'`. A function with a tick variable can take *any* kind of data. For example, let's look at the *identity function*. The identity function just returns whatever it is given. Since a procedure that "returns whatever it is given" does not need to make any obvious distinction for different types, we ought to be able to write a single polymorphic function that works for any of them.

```
let id x = x
```

If we type this into `dotnet fsi`, F# will tell us that the type is:

```
'a -> 'a
```

Let's try using `id` for values with different types.

```
> id 5;;
val it : int = 5
```

It works for numbers. We got `5` back. And you can see that it also works for strings.

```
> id "hi";;
val it : string = "hi"
```

Curried functions definitions, function types, and function application

OK, I'm about to introduce something very weird. Try not to get upset. Have a look at the following program again.

```
[<EntryPoint>]
let main argv =
        printfn "Hello, %s!" argv[0]
        0
```

We can rewrite `main` like:

```
let main(argv: string[]) : int =
    printfn("Hello, %s!")(argv[0])
    0
```

and this is exactly the same program. "BUT WAIT," you say, "WHY DOES `printfn` HAVE TWO PARENS????"

That's because, in F# function calls are *curried*.

F# is strongly based on a model of computation called the *lambda calculus*. We will discuss the lambda calculus in detail this class. For now, it's worth noting that the lambda calculus has no notation for functions that take multiple arguments. It doesn't have them because they are not necessary.

Here's a function in F# that we tend to think of as "taking two arguments."

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

However, the type for `let f x y = x y` is:

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

which may make your brain squirm a little the first time you see something like it. Type annotations are easy to read with a little practice. Let's break it into pieces.

According to the above type, `x`, our first variable, must be a function from any type `'a` to type `'b`. Since the types of `'a` and `'b` need not be the same type, F# uses two different letters (`a` and `b`).

If we squint at the type above a little, it has the form:

```
stuff -> other stuff
```

So, in principle, we should be able to give it some stuff and get some other stuff back. We know that the type of that input stuff, `x`, is `('a -> 'b)`. `x` must be a function, because that's what it's type says it is. So let's do that. Let's call `f` with a function we'll name `a`. We'll name the output

g.

```
let f x y = x y
let a x = x + 1
let g = f a
```

What is the type of `g`? It's also a function. Try it in `dotnet fsi` so you can see what type it prints out.

We can keep going. If g is a function, we can call it with an argument, right?

```
> g 3;;
val it : int = 4
```

So what we've learned is that functions of multiple arguments in F# really are functions that *return another function*. Composing a multi-argument function from single-argument functions is called *currying*, and calling them with arguments one at a time is called *partial application*.

You may find it hard to imagine why we would ever want partially applied functions. I did too, when I first learned F#! And, in fact, I went years without explicitly constructing any curried functions, which seemed to suggest that they were not necessary. Nevertheless, when I discovered their first "killer application," combinator parsers, it changed the way that I thought about them. I now write much more concise, readable code than the code I wrote before.

Let's look at the type of the `printfn` function. It is:

```
TextWriterFormat<'a> -> 'a
```

which is, perhaps, a little puzzling until you recall that F# is designed to interoperate with other .NET code. People using .NET mostly write C# code, and C# was strongly inspired by Java, especially its use of generics. This means that F# programs can have *both* polymorphic types like `'a` and generic types! For the most part, F# will handle the gory details of converting between these kinds of types for you, but you can see that the above type declaration uses both: `TextWriterFormat` is a C# generic class, but we can give it a polymorphic type `'a`.

Anyway, with the above type declaration, we can see that `printfn` takes a `TextWriterFormat<'a>` and returns an `'a`. Hang on... we called `printfn` with more than one argument, remember?

```
printfn "Hello, %s!" argv[0]
```

So should't `printfn` be a curried function? Actually, no— and the

reason is that `TextWriterFormat<'a>` is already secretly a function. For example, when used in `printfn`, the string `%s` causes F# to infer that `'a` must be a function `string -> unit`, and so the type of `printfn` becomes

```
(string -> unit) -> string -> unit
```

and you call it like

```
printfn "%s" "heya"
```

If we use the format string `"%s %d"`, then `printfn` becomes

```
(string -> int -> unit) -> string -> int -> unit
```

and you call it like

```
printfn "%s %d" "hello" 1
```

This polymorphic mechanism is how `printfn` is able to "magically" determine how many parameters to take depending on the given format string. Cool, huh? *Java cannot do this*, and in fact, Java's `String.format` method *cannot be checked by the compiler*. Instead, Java checks when the program runs and throws an exception when you mess up, which is an annoying hack in my opinion.

Return value

The last line in our `main` program is `0`. In F#, the last expression in a function definition is the return value. If you recall, returning `0` tells the operating system that "everything ran OK." Any other value signals an error.

A few more features

The ML family of languages favors pragmatism over mathematical purity. Therefore, it allows a programmer great flexibility to wiggle out of tough situations using mutable variables, side effects, imperative code, and casts. In this class, use of mutability, side effects, imperative code, and casts will be penalized, because it's hard to learn *functional* programming if you can lean on those features. After this class is over, feel free to use those other features. I myself use them in some circumstances, particularly when it is important that my code be fast. By the

end of this semester, you will have an appreciation for the tradeoffs that these little "escape hatches" entail, which are substantial.

Expressions

Everything in F# is an expression. Using the `dotnet fsi` read-eval-print-loop (REPL) program,

```
> 1;;
val it : int = 1
```

we can immediately see that anything we type into the REPL *returns a value*.

There are *no statements* in F#, although there are functions that look similar. Remember `printfn` from above? You may recall that when we called it like

```
printfn "Hello, %s!" "Dan"
```

it returned `unit`. What is `unit`? `unit` signals that a function only produces a side effect. In other words, it does not return anything. Nevertheless, everything in F# is an expression, so something must be returned. To fit into this scheme, the special value `()` is returned, which means "nothing" and has type `unit`. Let's see for ourselves in `dotnet fsi`.

```
> printfn "Hello, %s!" "Dan";;
Hello, Dan!
val it : unit = ()
```

Lambda expressions

Lambda expressions are a feature of F# that allow us to create functions definitions "anonymously." In other words, a lambda expression is a function definition with no name.

The following is a lambda expression that computes the identity function:

```
fun x -> x
```

Try it in `dotnet fsi`. Here's another example.

```
let y = 1
(fun x -> x) y
```

which evaluates to `1`.

Lambda expressions are very useful in F#, and we use them widely, particularly in `map` and `fold` functions, which we will get to in a future chapter, *Higher-Order Functions*.

Types

F# is a *statically typed* programming language, which means that every variable and datum in the language must have an associated type label, and that all operations on data must *type check*, meaning that those operations are logically consistent.

F# has a small set of *primitive data types*. These types represent fundamental categories of data representation in the underlying CLR virtual machine. The complete list may be found online[29]. Primitive types are written in lowercase in F#.

F# also allows you to create user-defined types. Note that the convention in F# is to write variables in all lowercase and user-defined types in upper camel case[30].

[29] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/basic-types

[30] https://en.wikipedia.org/wiki/Camel_case

Advanced F#

Here we cover some of the defining features of the F# programming language.

Algebraic data types and pattern matching

Algebraic data types (ADTs) and pattern matching are two features first widely used in the Hope[31]) programming language, which was developed concurrently with ML. ADTs and pattern matching are like chocolate and peanut butter: better together[32] (this commercial is totally ridiculous and worth the 30 seconds of your time.)

Algebraic data types are a way of concisely express hierarchies of types *without inheritance*. Inheritance is a feature from *object-oriented programming*, which we will discuss later in the semester. If you already happen to know what inheritance is, you can think of ADTs as its functional equivalent. As you will see, they are in many ways much more elegant and easy to reason about, although there is a big tradeoff when it comes to large-scale software engineering projects.

Let's create a data type that represents a small set of animals. We'd like, at various points in our program, to be able to work generically with animals, and then at other points, to be able to work specifically with specific animals, like ducks and geese.

```
type Animal =
| Duck
| Goose
| Cow
| Human
```

The above is an algebraic data type. In F#, we call this kind of type definition a *discriminated union*. In other ML variants, these are sometimes called *union types* or *sum types*. These terms mean the same thing.

What is the meaning of `Duck` or `Cow`? They are, in fact, constructors. So if we want a `Duck`, we type `Duck`. Let's try it in `dotnet fsi`:

[31] https://en.wikipedia.org/wiki/Hope_(programming_language

[32] https://www.youtube.com/watch?v=hHPY5yoINMA

```
> let donald = Duck;;
val donald : Animal = Duck
```

Likewise, if we want a `Cow`, we type:

```
> let ephelia = Cow;;
val ephelia : Animal = Cow
```

Notice that the types of `donald` and `ephelia` are, quite specifically, `Animals`.

Let's now make a function that takes an `Animal` and does the "right thing" depending on the kind of animal.

```
let squeeze a =
    match a with
    | Duck -> "quack!"
    | Goose -> "honk!"
    | Cow -> "meeeeph!"
    | Human -> "WHY ARE YOU SQUEEZING MEEEEEE????"
```

The `match ... with` expression tells F# that we want to perform the appropriate thing using *pattern matching*. Pattern matching is a feature of many functional programming languages that let you concisely express conditional logic.

So if we `squeeze ephelia`, the function does the right thing:

```
> squeeze ephelia;;
val it : string = "meeeeph!"
```

Pattern matching is always type-safe. Suppose we forgot to put in the case for `Human`. The F# compiler will report that we missed a case.

```
    match a with
  --------^

warning FS0025: Incomplete pattern matches on this expression. For example,
the value 'Human' may indicate a case not covered by the pattern(s).
```

Pattern matching also lets us deal concisely with cases that don't matter to us. For example, imagine that all we really care about is squeezing `Cows`. We could write:

```
let squeeze a =
  match a with
  | Cow -> "meeeeph!"
  | _ -> "complaint"
```

The _ indicates a *default case* that will only be exercised by an `Animal` that is not a `Cow`.

ADTs that store data

The ADTs we've seen so for are limited in their usefulness because they don't store any data. We can extend them to store any data that we want.

```
type 'a List =
| Node of 'a * 'a List
| Empty
```

The above is a complete definition for a linked list. Expressions of the form `'a * 'b` are *tuples*. So a `'a * 'a List` is a 2-tuple that stores data of type `'a` on the left and a reference to a `List` on the right. You can have tuples of any arity in F#.

Also, observe that, we can also write types recursively. A `Node` stores a reference to a `List`[33].

Let's write a `prettyprint` function that, when given a `List`, prints it out all pretty. Here is an imaginary example in `dotnet fsi`:

```
> let mylist = Node ("a", Node ("b", Node ("c", Empty)));;
val mylist: string List = Node ("a", Node ("b", Node ("c", Empty)))
> prettyprint mylist;;
val it: string = "[a, b, c]"
```

Below is a reasonable first attempt at making our fantasy come true:

```
let rec prettyprint ll =
  match ll with
  | Node(data, ll') -> data.ToString() + ", " + (prettyprint ll')
  | Empty -> ""
```

Each case in our `match` expression, known as a *pattern guard*, ensures that `ll` has the form specified on the left side before executing the right side. When a case matches, data is *bound* to the given variables, in this case, `data` and `ll'`. For example, the data stored in the given `Node` is bound to the variable `data`; the tail of the list is bound to the variable `ll'`.

Pattern guards are composed from *deconstructors*, which have the same syntax and are essentially the inverse of *constructors*. For example, you can construct a tuple and deconstruct it using the same syntax.

[33] If you feel comfortable with our notion of algebraic data types so far, great! See if you can modify our `List` type to represent a binary tree. After all, a binary tree really is just a linked list with an extra reference...

```
> let tup = (1, "hi");;
val tup : int * string = (1, "hi")

> let (a,b) = tup;;
val b : string = "hi"
val a : int = 1
```

Anyway, let's try out our `prettyprint` function:

```
> prettyprint mylist;;
val it: string = "a, b, c, "
```

Hmm. Not quite. But with a little massaging, we can get this to work. One approach is to define a helper function. Note that functional programming languages let use define functions inside of function definitions[34].

[34] If this sounds crazy to you ask yourself: why not? Eventually you'll see that disallowing this behavior is actually the crazy design choice. Nested function definitions are very useful.

```
let prettyprint ll =
  let rec pp ll =
    match ll with
    | Node(data, Empty) -> data.ToString()
    | Node(data, ll') -> data.ToString() + ", " + (pp ll')
    | Empty -> ""
  "[" + (pp ll) + "]"
```

Notice that we made a number of changes to our original function. First, we defined a helper function, `pp`, inside of our main `prettyprint` function. Second, we called `pp` at the very end of `prettyprint`, and we surround whatever it returns with square brackets. Finally, because we want to omit the trailing comma, we have a special case: we check to see that a node is the last node so that we can construct a special string for that case. We have to check that case first, because the second case in the above pattern match is more general. And finally, we kept the `Empty` case at the end. You might be wondering why we do that given that we check for emptiness in our first case. Well, consider the following kind of list.

```
> let mylist2 = Empty;;
val mylist2: 'a List

> prettyprint mylist2;;
val it: string = "[]"
```

In short, the above definition works for empty lists too. I personally find the use of ADTs and pattern matching a refreshing way to build

software. Many concepts in computer science are simple and elegant in theory but difficult to implement in code in practice. F# and other ML languages give us the tools to express simple concepts simply.

Limitations

There are two things to note about the above definitions. First, note that I am *not* able to write the following definition:

```
type Thing =
| A of char
| B of string
| C of A * B
```

This definition produces the following error:

```
  | C of A * B
  -------^

error FS0039: The type 'A' is not defined.
```

Although ADTs admit recursive definitions, that's not what we're looking at here. Importantly, cases in an ADT are *not data types*. Instead, `A`, `B`, and `C` are *constructors* for the one type called `Thing`. So we cannot refer to `A` or `B` or `C` in our definition of `Thing`. We could instead write the following:

```
type Thing =
| A of char
| B of string
| C of Thing * Thing
```

but that's not quite the same since that lets a `C` store a `C` and maybe that's not what you want. To get that, we'd have to write,

```
type A = char
type B = string
type C = A * B
type Thing =
  | A of A
  | B of B
  | C of C
```

and that all works although it's a tad inelegant.

A second limitation is that we had to use the `rec` keyword somewhere in our `pp` definition.

```
let rec pp ll =
```

Leaving out the `rec` produces the following error:

```
      | Node(data, ll') -> data.ToString() + ", " + (pp ll')
  ------------------------------------------------------^^

error FS0039: The value or constructor 'pp' is not defined.
```

This error occurs because F# strictly adheres to the rules of the lambda calculus[35]. We will talk more about the lambda calculus in the future.

[35] For the curious, the reason is that F# uses a "desugaring" approach to interpret `let` expressions. An expression of the form `let z = U in V` desugars to the expression `(λz.V)U`. Well, if `z` is a function name, it is not available to use in the expression `U` because `U` is outside the lambda abstraction. Adding the `rec` keyword tells F# that the name of the function should be available *inside* the function body. In effect, a recursive `let` is different than a regular `let`.

Equality and type checking

Type checking is the process of ensuring that the use of values is logically consistent. F# can sometimes check *at compile time* whether two values will ever be equal without having to inspect values.

F#'s type system is a little different than the kind you've seen before in Java and C. Java and C use a form of types called a *nominal type system*. In essence, a nominal type is simply a label. *Nominal type checking*, therefore, boils down to ensuring that these labels match. For example, the following C program fails to type check:

```
int i = 1;
char *c = i;
```

because `i` is an `int` and `c` is a `char *`, although because C is weakly typed, this is only a warning and not a fatal compilation error:

```
program.c:3:9: warning: incompatible integer to pointer conversion
initializing 'char *' with an expression of type 'int'
      [-Wint-conversion]
  char *c = i;
        ^   ~
```

Java adds extra expressiveness to C's nominal type system which allows it to express *subtypes* (i.e., inheritance), which is why we say that it has a *nominal type system with subtyping*. Subtyping essentially means that, under some circumstances, some labels can be substituted for others. Java also enforces types *strongly*, so the Java equivalent to the above C program is a fatal compilation error.

F# uses a system of *structural typing*. In this case, equality is more than just checking labels, although labels are at the "bottom" of the type system. Structural type checking means that the type checker must prove, inductively, that two expressions are equivalent because they yield the

same *structural type*.

An example helps to clarify. For example,

```
> let a = 1;;
val a : int = 1

> let b = 1;;
val b : int = 1

> a = b;;
val it : bool = true
```

In this case, not only are the types for `a` and `b` equal, so are the values (note that F# denotes equality using the = symbol, not the == symbol; "not equal" is denoted with <>).

Neither `a` nor `b` have any "structure" and so the base case for structural type checking is nominal: we simply check that the labels match (`int` equals `int`).

But how do we check the equality of something more complicated?

The following checks equality inductively:

```
> let c = (1,"hi");;
val c : int * string = (1, "hi")

> let d = (1,"hi");;
val d : int * string = (1, "hi")

> c = d;;
val it : bool = true
```

To see that `(1,"hi")` equals `(1,"hi")`, we need to know the type of each expression. Both are 2-tuples. Thus, we know for two 2-tuples to be equal, we must check that both left sides are equal and that both right sides are equal (inductive step). Both left sides are `int` and `1` equals `1` (base case). Both right sides are `string` and `"hi"` equals `"hi"` (base case). Therefore, both 2-tuples are equal. Therefore `(1,"hi")` equals `(1,"hi")`.

Structrual type systems make equality comparisons very easy, and they extend to what are normally "opaque" types in other languages, like lists and arrays.

Lists

Lists are frequently utilized in functional programming. The original functional programming language, LISP, demonstrated that lists can be

used as a fundamental unit of composition for many more complicated data structures. F# also allows you to use lists in this manner, and because they are so important and widely-used, they are even easier to manipulate than in LISP. Although we were able to define a `List` type above, lists are so important to F# that it has special, built-in syntax to support them.

For example, the following lets us define a list using *list literal* syntax in F#:

```
let xs = [1; 2; 3; 4;]
```

We can also perform a variety of operations on lists easily.

```
> let xs = [1; 2; 3; 4;];;
val xs : int list = [1; 2; 3; 4]

> List.head xs;;
val it : int = 1

> List.tail xs;;
val it : int list = [2; 3; 4]

> let xs' = 0 :: xs;;
val xs' : int list = [0; 1; 2; 3; 4]

> List.length xs';;
val it : int = 5

> List.append xs' xs';;
val it : int list = [0; 1; 2; 3; 4; 0; 1; 2; 3; 4]

> List.rev xs';;
val it : int list = [4; 3; 2; 1; 0]

> List.sum xs;;
val it : int = 10
> xs' = xs';;
val it : bool = true

> xs = xs';;
val it : bool = false
```

```
> let ys = [0; 1; 2; 3; 4];;
val ys : int list = [0; 1; 2; 3; 4]

> xs' = ys;;
val it : bool = true
```

Notice that we were able to compare two lists simply, using = instead of ==. F# does not need us to distinguish between assignment and equality testing because it can tell from the context. F#'s structural type system makes this possible. We can even pattern-match on lists, which is incredibly useful for functions that recursively do something for every element in a list:

```
let rec list_length xs =
  match xs with
  | []     -> 0
  | x::xs' -> 1 + list_length xs'
```

where `[]` or `nil` represents the empty list and `::` means "construct list," an operation we call *cons* for short.

```
> list_length xs;;
val it : int = 4
```

Complete documentation for F# lists is available online[36]. F# is a *pragmatic* language, and as a matter of pragmatism, we should probably recognize that lists have undesirable performance properties for many applications, particularly numerical computing. For many applications, richer data types are desirable. F# has these too.

[36] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/lists

Arrays

Lists have O(n) performance for random-access. In other words, in the worst case, the cost of accessing an element is the cost of traversing the entire list. Arrays have much better performance for random-access: O(1). In other words, the worst case performance for arrays is a constant time, or the amount of time it takes to execute a single operation.

The following lets us define an array using *array literal* syntax in F#:

```
let arr = [|1; 2; 3; 4;|]
```

We can also perform a variety of operations on arrays easily:

```
> let arr = [|1; 2; 3; 4;|];;
val arr : int [] = [|1; 2; 3; 4|]
```

```
> let i = arr.[3];;
val i : int = 4

> Array.length arr;;
val it : int = 4

> Array.rev arr;;
val it : int [] = [|4; 3; 2; 1|]

> Array.filter (fun x -> x > 2) arr;;
val it : int [] = [|3; 4|]

> arr.[0..1];;
val it : int [] = [|1; 2|]

> arr.[2..];;
val it : int [] = [|3; 4|]

> arr.[..2];;
val it : int [] = [|1; 2; 3|]

> Array.init 10 (fun i -> i * i);;
val it : int [] = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]

> Array.sum (Array.init 10 (fun i -> i * i));;
val it : int = 285

> Array.map (fun x -> x + 1) arr;;
val it : int [] = [|2; 3; 4; 5|]

> Array.fold (fun acc x -> acc + x) 0 arr;;  // this is fold left
val it : int = 10

> Array.foldBack (fun x acc -> acc + x) arr 0;; // this is fold right
val it : int = 10

> let arr2 =  [|1; 2; 3; 4;|];;
val arr2 : int [] = [|1; 2; 3; 4|]

> arr = arr2;;
val it : bool = true
```

Notice that we were able to compare two arrays simply, using `=`. This is possible because of F#'s structural type system.

The complete documentation on F# arrays is available online[37].

[37] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/arrays

Other types

F# has many other types, including `Tuple`, `Sequence`, `Map`, `Option`, classes, interfaces, and abstract classes, and many others. The latter three types

are object oriented features of F#. Please do not use classes, interfaces, or abstract classes unless I ask you to do so.

Conditionals

`if/else` expressions look a bit like their counterparts in Python:

```
if x > 0 then
  1
else
  2
```

Of course, since conditionals are *expressions* in F#, you can also do neat tricks like use them to conditionally assign values, much like how you might use the ternary operator (`a ? b : c`) in C:

```
let y = if x > 0 then
          1
        else
          2
```

Indentation is important for conditionals. Note that the body of the true and false clauses must be indented past the start of the `if` expression.

Loops

While F# has looping constructs, `for` and `while`, you should not use them in this class. Instead, you should use `map`, `fold`, and recursion instead.

The following table provides a handy guide for deciding which construct to use in F# when your brain tells you that you need to use a loop.

Problem	Java	F#
Do something until a condition is satisfied	`while`	Write a recursive function and make the condition a base case.
Do something for a bounded number of times	`for`	Write a recursive function and make the bound a base case.
Convert every element of a collection into something else	`for` or `foreach`	`map`
Accumulate a value	`for`, `foreach`, or `while`	`fold`
Convert a recursive data structure into another data structure	Recursive function	Recursive function or `fold`

Recursion instead of `while`

Let's start with recursion to solve problems. One nice thing about a `while` loop is that you don't need to know how many times your program needs to repeat itself until it is done computing a value. For example, when doing a membership query in an unsorted linked list in C (e.g., "is 4 in the list?"), you can use a while loop to simply continue

getting the next element until it is found.

```
bool contains(listnode *xs, int x) {
  while (xs != null) {
    if (xs->head == x) {
      return true;
    } else {
      xs = xs->tail;
    }
  }
  return false;
}
```

What characterizes problems like this is that there is no obvious bound on the loop before we find the element we're looking for. We solve this class of problems in functional programming with recursion.[38]

[38] Remember that recursive function definitions must use `let rec`.

```
let rec contains xs x =
  match xs with
  | []    -> false
  | y::ys ->
    if x = y then
      true
    else
      contains ys x
```

This particular solution pattern matches on the list to deconstruct it into a head (`y`) and a tail (`ys`), but you could also explicitly call `List.head` and `List.tail` if you wanted.

You might argue that this is a silly example because we know that there can be no more comparisons than the length of the list. That's true. But there are other problems where you actually don't know at all, and the same pattern applies. For example, what if we wanted to run a little experiment: how many times do we need to flip a coin before a heads comes up? The outcome is determined by the laws of probability. It *probably* won't take many coin flips for heads to come up—in fact, we know that we have a 50% chance on the very first try—but it *could* take a very long time. There's a nonzero probability that it could take a trillion coin flips.

```
let rec num_tries_until_match(r: System.Random)(n: int)(i: int)(e: int) =
  let rn = r.Next(n)
  if rn = e then
    i
  else
    num_tries_until_match r n (i + 1) e
```

Where `n` is the number of "sides" of our coin (or die, or whatever), `i` is the count so far, and `e` is the value we're looking for. Let's say that, when `n = 2`, then when `e = 0` that's heads and when `e = 1` that's tails. If we call this a few times, you can see that the answer can vary quite a bit:

```
> let r = System.Random();;
val r : System.Random

> num_tries_until_match r 2 1 0;;
val it : int = 3

> num_tries_until_match r 2 1 0;;
val it : int = 1

> num_tries_until_match r 2 1 0;;
val it : int = 2

> num_tries_until_match r 2 1 0;;
val it : int = 4

> num_tries_until_match r 2 1 0;;
val it : int = 5
```

Other important features

F# has essentially every feature that a modern language like Java has, and there are far too many to discuss in this class. However, there are a few more conveniences worth mentioning, so I discuss those here. One important feature, *higher-order functions*, will have to wait until you understand the lambda calculus, so we discuss them in a separate chapter.

Raising Exceptions

You can define an exception in F# like:

```
exception MyError of string
```

and you can throw it like:

```
raise (MyError("Error message"))
```

F# also has a "lightweight" syntax that I use frequently:

```
failwith "something bad happened!"
```

Runtime exceptions are more useful in F# than they are in ordinary languages. For example, I often want to compile a function that I am in the process of writing, just to see if I have made any mistakes elsewhere. Because F# is functional and strongly-typed, the type-checker will tell me that my function is incomplete, which I already know. A useful trick is to use `failwith` to make the type checker temporarily ignore my incomplete function. This is especially useful when working through cases in a pattern match.

```
let rec prettyprint e =
  match e with
  | Variable(c) -> c.ToString()
  | Abstraction(v,e') -> failwith "TODO1"
  | Application(e', e'') ->  failwith "TODO2"
```

I use this trick frequently, although you should be aware that doing so trades a compile-time error for a runtime one!

Catching Exceptions

You can catch exceptionsing using F#'s `try ... with` syntax,

```
try
  prettyprint (Abstraction('x', Variable('x')))
with
| MyError(msg) ->  "oops: " + msg
```

which returns the `string` value `"oops: TODO1"`.

Option types

Like Java and C#, you can store `null` in values.

```
let x = null
```

The use of `null` is now widely regarded as a mistake in computer science. Tony Hoare (winner of the Turing Award), and inventor of `null`, called it a "billion-dollar mistake":

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Instead, in F#, we prefer the type-safe `option` type. Let's write a function that uses `option`.

```
let onedivx x =
  if x = 0.0 then
    None
  else
    Some (1.0 / x)
```

Now when we use `onedivx`, we get back an `option` type:

```
> onedivx 0.0;;
val it : float option = None

> onedivx 1.1;;
val it : float option = Some 0.9090909091
```

`option` gives us a way to signal the failure of a computation in a type-safe manner. In fact, the type-checker *forces* us to deal with the error:

```
> (onedivx 0.0) + 2.2;;

  (onedivx 0.0) + 2.2;;
  ----------------^^^

error FS0001: The type 'float' does not match the type 'float option'
```

To use the return value, we must "unwrap" it first, using pattern

matching.

```
> match onedivx 0.0 with
- | Some res -> printfn "%f" res
- | None     -> printfn "Oh, dear!"
- ;;
Oh, dear!
val it : unit = ()
```

Further Reading

The complete F# language reference is available online[39]. I encourage you to refer to it often as it is uncharacteristically thorough for technical documentation while remaining easy to read.

[39] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/

Syntax

Figure 7: You might be intimidated by the use of the word *formal* in mathematics and computer science. You should not be. A *formal tool* is one that you can use as long as the "shape" of the problem fits the "mold" of the tool. Formalisms are intended to *simplify* one's thinking about a problem. I like to think of the baby toy shown above. The only trick to using a formal tool is to learn to recognize when the shape "fits."

What is a language? If you've studied a second language, or you are a grammar nerd, you might have a working definition, something like

> *A language is a collection of words, employed in some meaningful order.*

If you stretch your definition of "word," you might even convince yourself that a programming language bears some resemblance to the above definition. In fact, our understanding of programming languages owes a great deal to the field of linguistics from the 20th century, particularly the work of Noam Chomsky. The most important idea from that era is the idea that we can define a language *formally*, by identifying and classifying its parts.[40] Nevertheless, the formal study of programming languages diverged from linguistics over time, because the act of communicating with a computer tends to require more precision than communicating with another human.

[40] This idea is no longer in fashion in linguistics, but it is still going strong in the study of programming languages.

Let's start informally, by identifying the essential elements of a language. We will motivate this discussion with a toy language. As noted above, a language must have *words*. For example, suppose our language consists of the following set of words.

`Jack`, `candlestick`, `over`, `be`, `jump`, `nimble`, `quick`, `the`

The words above are listed in no particular order. Although you might recognize the set of words above from a nursery rhyme, in their present state they convey no clear meaning to us. A second essential element of a language is that words have an *order*. A sequence of words, given in a predefined order, is called a *sentence*. A language is simply the set of all valid sentences. For example, here are some sentences we will define to be valid:

```
"Jack be nimble"

"Jack be quick"

"Jack jump over the candlestick"
```

Here are some sentences we do not want to be valid.

`"Jack Jack Jack Jack Jack"`

`"blimey"`

`""` (i.e., the empty sentence)

As mentioned earlier, programming languages relies on formal tools, mainly the notion of sets and of functions. Let's transform our informal example language into a formal one using sets.

Let the set of all words in the languages be the set,

$$\Sigma = \{\texttt{Jack}, \texttt{candlestick}, \texttt{over}, \texttt{be}, \texttt{jump}, \texttt{nimble}, \texttt{quick}, \texttt{the}\}$$

where Σ is pronounced "sigma."

Every valid sentence in a language must consist solely of the words found in Σ. As a result, members of this set are called *terminals*. The word "terminal" is suggestive of the fact that the process of deriving a valid sentence must *terminate* with all of its words drawn from Σ. We can see that the invalid sentence `"blimey"` is not valid because $\texttt{blimey} \notin \Sigma$. We will talk more about the process of derivation shortly.

Now that we've formally defined the set of valid words, how do we define valid word orders? To do this, we define a set of *production rules*,

$$P = \{N \texttt{ ::= } (\Sigma \cup N)*\}$$

I know this definition seems like a lot, so let's unpack it. First, some symbols. N is the set of all nonterminals. Parens $()$ are used for grouping symbols and do not appear in the language itself. The symbol $\cup$ is "set union," and when we see an expression like $A \cup B$, it means that we're talking about the set formed by *both* sets A and B taken together (see Figures 8–11). The symbol $*$ is called *Kleene star*, [41] and it means "zero or more."

A production rule tells us what kinds of substitutions are allowed in our language. For example, the following could be a production rule in our language.

`<name>` `::=` Jack

A production rule always starts with a nonterminal symbol enclosed in angle brackets, like `<name>`. Think of a nonterminal as a kind of placeholder. The symbol `::=` means "is defined as." So the production rule above means "`<name>` is defined as `Jack`." In essence this production rule implies that occurrences of the nonterminal `<name>` can be substituted with the terminal `Jack`. Recall that a valid sentence in a language consists only of terminals.

Stepping back a little, the statement $P = \{N \texttt{ ::= } (\Sigma \cup N)*\}$ is saying "P is defined as the set"

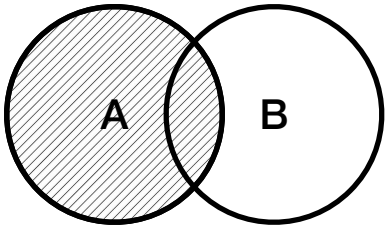


Figure 8: The set A shown in the shaded region.

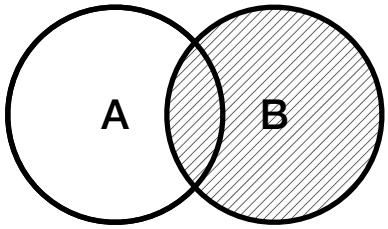


Figure 9: The set B shown in the shaded region.

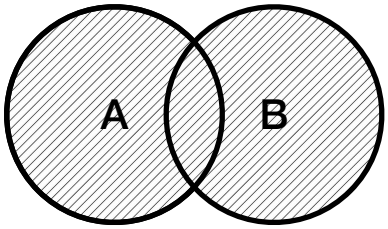


Figure 10: The union of sets A and $B = A \cup B$, shown in the shaded region.

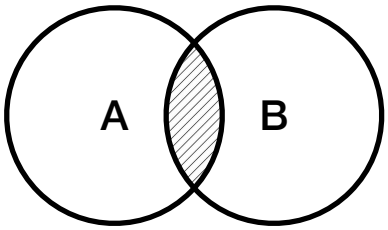


Figure 11: The intersection of sets A and $B = A \cap B$, shown in the shaded region.

[41] Kleene is pronounced *klay-nee*.

$$P = \{ \ldots \}$$

"of production rules"

$$P = \{\ldots \texttt{ ::= } \ldots\}$$

"where the left hand side of each rule is a nonterminal"

$$P = \{N \texttt{ ::= } \ldots\}$$

"and the right hand side of each rule is a sequence of zero or more nonterminals and terminals."

$$P = \{N \texttt{ ::= } (\Sigma \cup N)*\}$$

Observe that our definition of P says nothing about *which* nonterminals can be used. In fact, the same nonterminal can be used on both the left and right side of a production rule, which means that recursive rules are allowed.

This discussion may feel abstract to you right now. To make things a little more concrete, here is the complete set of production rules for our toy language.

```
      <start>  ::=  <name> <phrase>
       <name>  ::=  Jack
     <phrase>  ::=  be <adjective>
                 |  jump over the candlestick
  <adjective>  ::=  nimble | quick
```

The one symbol you have not yet seen is `|`, which means "alternatively." For example, a `<phrase>` is defined as `be <adjective>`, alternatively defined as `jump over the candlestick`. `<adjective>` is defined as either `nimble` or `quick`.

Backus-Naur Form

The *syntax* of a language is its "appearance": what words does it have, and in what order are they allowed? The grammar written as it is in the example above is in a style we call Backus-Naur Form, and is often referred to using the abbreviation BNF. Named after its inventors, John Backus and Peter Naur, BNF was first used to describe the syntax of the influential (if not widely used) ALGOL programming language.

Interestingly, the idea of a formal grammar may have been independently invented nearly 25 centuries ago by the scholar, Pāṇini, who used

something like BNF to describe Sanskrit. Regardless of the true inventor, BNF is the standard notation for defining programming language grammar, and you can find it throughtout programming literature. For example, the Python[42] programming language documentation is described using a version of BNF.

[42] https://docs.python.org/3/reference/grammar.html

Grammar

Now that we have all of the essential pieces, I can give you the formal definition of a language. A *grammar* G is a four-tuple, consisting of the set of terminals, nonterminals, and production rules for a language.

$$G = (\Sigma, N, P, S)$$

$S \subseteq N$ is the set of start symbols. A grammar encodes a procedure for *deriving* all of the valid sentences of a language. We will discuss the derivation process below. Formally, a language is the set of all possible sentences that can be derived using a grammar, G. Therefore, we often refer to a language as a function of a grammar, which we write as $\mathcal{L}(G)$.

The set of production rules for our toy language satisfies the definition of a grammar.

$$\begin{aligned} G &= (\Sigma, N, P, S) \\ \Sigma &= \{\texttt{Jack}, \texttt{candlestick}, \texttt{over}, \texttt{be}, \texttt{jump}, \texttt{nimble}, \texttt{quick}, \texttt{the}\} \\ N &= \{\texttt{<start>}, \texttt{<name>}, \texttt{<phrase>}, \texttt{<adjective>}\} \\ P &= \text{(refer to the example above)} \\ S &= \{\texttt{<start>}\} \end{aligned}$$

By convention, we only explicitly write down P, because G, Σ, and N can be derived from P, and S can often be inferred from its name. However, whenever in doubt, write it all down.

Derivation

We can use a grammar to derive all possible valid sentences in a language, $\mathcal{L}(G)$. Depending on the size of the grammar, this may take a long time. Fortunately, we don't usually need to know $\mathcal{L}(G)$. Usually, we simply want to know whether a *given* sentence s is a member of a language, more formally, whether $s \in \mathcal{L}(G)$. Checking whether $s \in \mathcal{L}(G)$

is a process we call *parsing*. If we can derive the sentence from G, then colloquially, we say that "s parses."

Let's first derive some sentences from G. Then we will try parsing given sentences.

Both derivation and parsing always start with one of the start nonterminals. Derivation proceeds as follows:

1. Write down a start symbol.
2. For each nonterminal in the current sentence, replace it with a definition from G.
3. If the current sentence contains any nonterminals, go to step 2.
4. Done.

Our grammar defines a sole start nonterminal, `<start>`. We will start there.

```
<start>
```

According to step 2, we must replace `<start>` with its definition.

```
<name> <phrase>
```

Step 3 says that if the result contains any nonterminals (it does), we must continue replacing the nonterminals. First we handle `<name>`.

```
Jack <phrase>
```

Next, we handle `<phrase>`. Here we have a choice. Which do we pick? Right now, since we're just trying to generate any valid sentence, any choice is fine. Let's pick the first alternative.

```
Jack be <adjective>
```

Again, this sentence contains nonterminals, so the derivation process must continue. `<adjective>` also has two alternatives. Let's choose the first alternative again.

```
Jack be nimble
```

Finally, because no nonterminals remain, we are done.

At this point, you might have realized that our grammar can generate individual sentences from the famous nursery rhyme, but it cannot generate the entire rhyme itself. However, we can fix this with two small modifications to the grammar.

```
      <start>  ::=  <name> <phrase>
                 |  <name> <phrase> <start>
       <name>  ::=  Jack
     <phrase>  ::=  be <adjective>
                 |  jump over the candlestick
<adjective>  ::=  nimble | quick
```

Our modification adds a recursive alternative to the `<start>` rule. How does this play out? As before, we start with the `<start>` symbol. But instead of choosing the first alternative, we might choose the second alternative, resulting in derivations like the following.

```
<start>
<name> <phrase> <start>
... omitted for brevity ...
Jack be nimble <start>
... omitted for brevity ...
Jack be nimble Jack be quick
```

In fact, we have overapproximated the grammar for this rhyme. While it can produce the famous nursery rhyme[43], it can also generate other alternatives.[44] I leave it as an exercise to you to come up with a grammar that does not have this problem. However, our slightly flawed grammar illustrates an important property of many formal languages. How many valid sentences are in $\mathcal{L}(G)$? Think about the answer to this question before continuing.

[43] Jack be nimble Jack be quick Jack jump over the candlestick

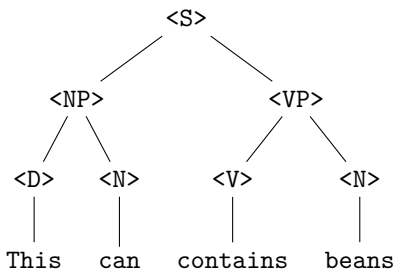
[44] Jack be quick Jack be quick Jack be quick Jack be quick

This language has an infinite number of valid sentences. This is not as crazy as it sounds. English has an infinite number of valid sentences, as do all other human languages. For example, you can always make a longer sentence in English by adding the word "and" followed by another sentence. Recursive grammar rules are used for all kinds of things, but one important use is in constructing languages of infinite size. Although it is only one aspect of a programming language, the size of a language tells you a little about its expressiveness. For instance, both Java and Python are languages of infinite size.

Parsing

As a practical matter, we usually want to use a grammar to solve a slightly different task: to know whether a *given* sentence is a valid element of $\mathcal{L}(G)$. More importantly, in programming languages, we often also want to know what the *structure* of a sentence is. You may have derived the structure of a sentence before; the task, called *sentence diagramming*, is commonly done in elementary school.

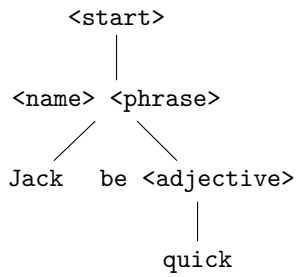
The sentence "`This can contains beans`" is diagrammed like so:



where `<S>` means *sentence*, `<NP>` means *noun phrase*, `<VP>` means *verb phrase*, `<D>` means *determiner*, `<N>` means *noun*, and `<V>` means *verb*. Observe that the symbols in angle brackets `<>` are actually nonterminals, and none of them appear in the sentence itself. However, they reveal the structure of the sentence. Derivations drawn out in tree form are a kind of *parse tree* called a *derivation tree*. Derivation trees have three important properties:

1. Every step of the derivation is shown in the tree.
2. Interior nodes in the tree always contain at least one nonterminal.
3. The leaf nodes in the tree contain only terminals.

To draw a derivation tree, we follow essentially the same process that we do when we derive a sentence, except that we are producing a derivation for a specific sentence. If we are unable to produce a derivation tree, it means that the sentence is not a member of the language. Let's look at the sentence "`Jack be quick`". Here is the complete derivation tree for that sentence.



Our grammar starts with `<start>`. We know that we must replace `<start>` with `<name> <phrase>` because nonterminals must be replaced. So the meaning of a node in our tree is that it is the state of the expression at a particular point in the derivation. Edges represent substitutions, so there is an edge between `<start>` and `<name> <phrase>` because we replaced the former with the latter. If you follow this process to its conclusion, all of the leaves should be terminals.

Ambiguity

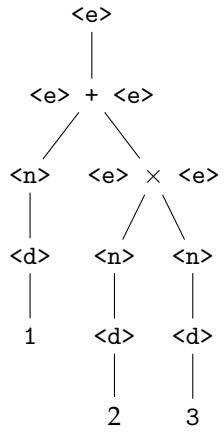
As precise as formal grammars are, believe it or not, they still suffer from some imprecision. To illustrate, let's look at another grammar.

```
<e> ::= <n> | <e> + <e> | <e> - <e> | <e> × <e> | <e> / <e>
<n> ::= <d> | <n><d>
<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

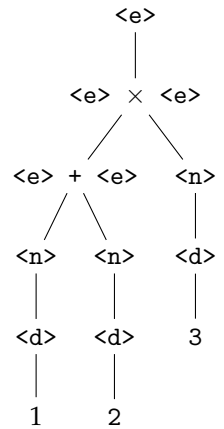
where `<e>` is the start symbol.

Suppose we want to produce a derivation tree for the expression `1 + 2 × 3`. Take a minute to produce a derivation tree using the grammar above.

Surprisingly, this grammar yields two valid trees. Which did you draw? Was it this one?



Or was it this one?



The answer matters because of what that difference implies. The first derivation tells us that 2 and 3 are bound together with a × operation. The second derivation tells us that 1 and 2 are bound together with a + operation. In arithmetic, we know that the order of operations matters, so this difference implies two different evaluation orders. The first implies that the expression should be evaluated like `1 + (2 × 3)` while the second implies that the expression should be evaluated like `(1 + 2) × 3`. Simplifying these two expressions produce two different answers (try it), so we try to avoid ambiguity in grammars whenever possible.

One way to do it is to modify the grammar so that parses are unambiguous. For example, the following grammar adds mandatory parentheses in order to avoid ambiguity.

```
<e> ::= <n> | (<e> + <e>) | (<e> - <e>) | (<e> × <e>) | (<e> / <e>)
<n> ::= <d> | <n><d>
<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The LISP programming language uses this approach pervasively. For example, here is our previously-ambiguous expression rewritten using the new grammar.

```
(1 + (2 × 3))
```

However, many users consider this kind of modification to be burdensome. Parentheses must be used everywhere, even for expressions that are unambiguous. In the expression above, the expression is unambiguous once we have the inner set of parentheses, but the grammar does not let us write that. Fortunately, there are other approaches that allow us to have unambiguous grammars without extra syntax.

Precedence

You might recall the mnemonic your teacher taught you to remember the order of operations in arithmetic expressions: PEMDAS. This mnemonic tells us that we should evaluate *parentheses* first, then *exponents*, then *multiplication* and *division*, then *addition* and *subtraction*. Grammars that are ambiguous usually come with an additional table that tells you what the precedence is for expressions. For example, the grammar for C is ambigious, so its documentation comes with an operator precedence table. An abbreviated version of the table is shown in Table 1.

Precedence literally means that something takes priority. When parsing alternatives, it means that we should try to match the highest priority alternatives first. So when the C language parser encounters an expression like `1 + 2 × 3`, the parse tree resembles the first parse tree shown above, the one with the × subexpression *deeper* in the tree. We

Precedence	Operator	Description
1	`++ --`	Postincrement/decrement
1	`()`	Parentheses
2	`++ --`	Preincrement/ decrement
2	`!`	Logical not
3	`* / %`	Mult, div, and rem
4	`+ -`	Add and sub

Table 1: An excerpt of the precedence rules used by the C programming language. In C, a *lower* number means *higher* priority.

will discuss how computer programs are evaluated at length in a future reading; for now, remember a simple rule: elements that are deeper in a parse tree are usually evaluated before elements that are shallower in a parse tree. This makes sense for our example expression because we want to evaluate multiplication before we evaluate addition.

Associativity

You may have noticed something about the precedence table in Table 1: some operators, like `*` and `/`, have the same precedence. Nevertheless, we still need to be careful about the order in which we parse. Take the following expression, which only uses division operations. Because all of the operations in this expression are the same, they unavoidably have the same precedence.

```
1 / 2 / 3
```

Again, try parsing this expression. There are two possible parse trees, even with precedence rules. Which should be the correct derivation? We can try evaluating these expressions as if they were parenthesized to see what the effect different parses have.

```
((1 / 2) / 3)  =  ≈ 0.16666666666666666667
(1 / (2 / 3))  =  1.5
```

If we do this calculation out on paper using the rules of arithmetic (try it), we get $\frac{1}{6}$, which is roughly the number C gives me. Like many languages, when faced with grammar alternatives of the same precedence, C turns to a tie-breaking rule called *associativity*.

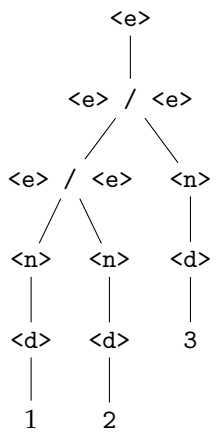
Table 2 shows C's associativity rules. Associativity rules tell us which alternative we should consider when parsing an expression from left-to-right: does the *left* expression go deeper in the tree or does the *right* expression go deeper in the tree?

Let's consider `1 / 2 / 3` again. Since our parser sees two division operators (the `/`), it must decide whether the expression has the form `(1 / 2) / 3` or `1 / (2 / 3)`. Looking in the table, we see that division is *left-associative*, so that means that we should parse the expression as

Precedence	Operator	Description	Associativity
1	`++ --`	Postincrement/decrement	left
1	`()`	Parentheses	left
2	`++ --`	Preincrement/decrement	right
2	`!`	Logical not	right
3	`* / %`	Mult, div, and rem	left
4	`+ -`	Add and sub	left

Table 2: An excerpt of the precedence rules used by the C programming language. In C, a *lower* number means *higher* priority.

if it were `(1 / 2) / 3`. In other words, the leftmost `/` is deeper in the parse tree.



Derivation trees are essential for understanding syntax. Being able to draw one is important if you are ever in the position of writing a parser, which is a common software engineering task. You will be writing a handful of parsers in this class. However, derivation trees have limited usefulness when trying to understand the *meaning* of a program, an area of programming language design we call *semantics*. We will build up to discussing semantics over the next couple of chapters.

Introduction to the Lambda Calculus, Part 1

This is part 1 of a two-part reading on the lambda calculus. In this reading, I introduce the lambda calculus, its origins and purpose, and discuss its syntax. In part 2, I will discuss its semantics.

Introduction

The lambda calculus is a model of computation. It was invented in the 1930's by the mathematician Alonzo Church, who like his contemporary, Alan Turing, was interested in understanding what computers were capable of doing *in principle*.

For me, there are three really remarkable things about the lambda calculus.

1. Although this fact has never been proven, it is widely believed that the lambda calculus is capable of expressing *any* computation.
2. It was invented before any computers actually existed in the real world.
3. It is really, really simple.

The lambda calculus is equivalent in expressive power to that other famous model for computation, the Turing machine. Unlike the Turing machine, though, the lambda calculus has an elegance to it that the Turing machine lacks. For starters, with some effort, you can understand a "program" written in the lambda calculus. It is incredibly difficult to understand any kind of "program" written for a Turing machine, because a Turing machine closely corresponds to a mechanical computer. The lambda calculus, on the other hand, is essentially the language of *functions*.

As a result, the lambda calculus serves as the theoretical foundation for many real programming languages, most notably LISP. More modern languages, like ML and Haskell, are also deeply influenced by the lambda calculus. Many ideas that came from the lambda calculus, like "anonymous functions," have found their way into bread-and-butter

languages like Javascript, Java, C#, and even C++. When we talk about "functional programming," at its core, what we're talking about is the lambda calculus.

The lambda calculus as a programming language

The lambda calculus can be thought of as a kind of minimal, universal programming language. What do I mean by "minimal"? I mean that it is small and that its features are essential in some way. By "universal" I mean that it is capable of expressing all *computable functions*. While it is probably not the *most* minimal programming language[45], it is the most useful minimal language that I know of. We will talk about what a *computable function* is later; for now, understand it to mean that an ideal computer should be capable of computing it.

[45] In fact, the Intel `mov` instruction all by itself has the same expressive power. See the paper *`mov` is Turing-complete*, by Stephen Dolan, University of Cambridge Computer Laboratory.

Formal definitions

Since I claim that the lambda calculus is like a programming language, we ought to be able to examine it formally like a programming language. Most formal specifications of a language come in two pieces: 1. syntax and 2. semantics.

Syntax is the "surface appearance" of a programming language. For example, the following snippets are from Java and F#, respectively.
Java:

```
int sum(List<Integer> lst) {
    int accumulator = 0;
    for (Integer i: lst) {
        accumulator += i;
    }
    return accumulator;
}
```

F#:

```
  let sum xs
      let mutable accumulator = 0
      for x in xs do
          accumulator <- accumulator + x
      accumulator
```

These two languages have different *syntax*, so they look different; they use different words. But both `sum` functions written here convey exactly the same meaning. Therefore, the two functions have the same *semantics*.

Syntax of the lambda calculus

So let's start with the appearance of the lambda calculus. What does it look like?

Here's one of the simplest functions you can write in the lambda calculus: the identity function.

```
λx.x
```

You might have seen this in algebra before. It looks something like:

$$f(x) = x$$

Or maybe Java?

```
T identity<T>(T t) {
    return t;
}
```

The lambda calculus version expresses exactly the same concept as the algebraic and Java versions.

Actually, the lambda version is more concise: In the algebra version and the Java version, functions are *named*. In the algebraic expression above, the function is named `f`. In the Java version is it called `identity`. In the lambda calculus, functions do not have names associated with them. They are all *anonymous*. Why? Because Church was looking for a *minimal* model of computation. As it turns out, function names are not essential.

Backus-Naur form

Syntax is the arrangement of words and phrases to create well-formed sentences in a language. When we say well-formed, what we mostly mean is that we are not just stringing together words arbitrarily, forming nonsense. Instead, there are rules that dictate how words go together. These rules ensure that sentences in our language follow patterns that allow us to extract conventional meanings from them.

The rules that define a syntax are called a *grammar*. The bootstrapping problem of how exactly one comes up with a language that describes languages dates back to Indian scholars of antiquity. In the 1950's, John Backus and Peter Naur devised a simple, formal solution. Using

their system, one can "generate" all of the valid sentences belonging to a language. That system is now known as Backus-Naur form, or BNF.

Here is the grammar for the lambda calculus:

```
<expression>  ::= <variable>
                |  <abstraction>
                |  <application>

<variable>    ::= x
<abstraction> ::= λ<variable>.<expression>
<application> ::= <expression><expression>
```

This grammar is important enough—and simple enough—that *I suggest that you memorize it*.

BNF has two kinds of grammar constructions: *nonterminals* and *terminals*. In the grammar above, nonterminals are written between angle brackets (< and >). Other characters or words are terminals. I will explain what these things mean in a moment.

The ::= means "is defined as" and the | means "alternatively." So if you read the definitions literally, they say:

> <expression> is defined as <variable>. Alternatively, <expression> is defined as <abstraction>. Alternatively, <expression> is defined as <application>.
>
> <variable> is defined as x.
>
> <abstraction> is defined as λ<variable>.<expression>.
>
> <application> is defined as <expression><expression>.

Each *line* in the grammar is known as a *production rule* (or just *rule* for short), because it *produces* valid syntax. If you start with the "top level" nonterminal <expression>, and follow the production rules, when you finally have a sentence that contains only terminals, you now have a valid sentence in the grammar. Since our grammar is for the lambda calculus, this means that you have a valid lambda expression (program).

Following a rule is called an *expansion*. Let's try a couple expansions and see what kind of sentences we can get.

Example 1:

1. Start with `<expression>`.

```
<expression>
```

2. Expand it into either `<variable>`, `<abstraction>`, or `<application>`. Let's choose `<variable>`.

```
<variable>
```

3. If we look at our definition for variable, there is not much choice. It must be expanded into one thing only:

```
x
```

Since no nonterminals remain, we now know that `x` is a valid lambda calculus program.

Example 2:

1. Start with `<expression>`.

```
<expression>
```

2. Let's expand it into `<abstraction>`.

```
<abstraction>
```

3. This also expands into the following.

```
λ<variable>.<expression>
```

4. As we know, `<variable>` only expands into one thing, `x`, but `<expression>` could be many things.

```
λx.<expression>
```

5. It should be apparent, at this point, that this process is recursive. Since we don't have all day, let's expand `<expression>` into `<variable>` and choose `x` again.

Since no nonterminals remain, we now know that `λx.x` is a valid lambda calculus program. In fact, it's the identity function we discussed before.

```
λx.x
```

Parsing

If you think about the kind of sentence generation we did above as a function, at a high level, it would look something like this (pay attention to the parameters and return types):

```
Sentence generate(Grammar g) {
  // algorithm
}
```

Parsing is, in some ways, the converse:

```
bool parse(Sentence s, Grammar g) {
  // algorithm
}
```

In other words, instead of generating a sentence in a grammar, a parser *recognizes whether a sentence belongs to a grammar*. If a sentence *can be generated* from a grammar, `parse` returns `true`. If a sentence cannot be generated, `parse` returns `false`.

In practice, we often expand the definition of `parse` a tad so that, if `parse` would return `true`, it returns a *derivation*, otherwise it returns `null`.

For example, we already know that `λx.x` parses, but what does its derivation look like? (Fig. 12)

In programming languages, a derivation has a special name: we call it a *syntax tree*, because it shows how the parts of a program are related to each other. Understanding how a lambda expression parses will help you understand how you can "compute" things using it.

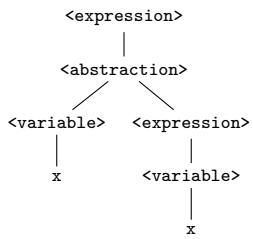


Figure 12: Derivation of the expression `λx.x`.

Precedence and Associativity

A few little details often trip up newcomers to the lambda calculus. These details rely on concepts called *precedence* and *associativity*.

The dictionary defines "precedence" as the condition of being more important than other things, and this is essentially the same idea when we talk about languages. Precedence in a programming language means that certain operations are evaluated before others. Using algebra again as an example, this should already be familiar to you. For example, you know that the algebraic expression

```
x + y * z
```

needs to be evaluated by first multiplying `y` and `z`, then finally by adding `x` to the product. Without the rule that says that multiplication

has higher precedence than addition, the above expression would be ambiguous, because it could be parsed in one of two ways (Fig. 13)

You probably recognize that the parse on the left is the correct one, because it implies that the addition depends on the multiplication, not that the multiplication depends on the addition.

Therefore, precedence is used to remove ambiguity from a language. In the lambda calculus, *application has higher precedence than abstraction*. This means that if you have an expression like

```
λx.xx
```

you should understand it to mean the derivation shown in Figure 14 and not the one shown in Figure 15.

Even with precedence, we are occasionally faced with ambiguity. For example, the lambda expression

```
xxx
```

Should we think of this expression as `(xx)x` or `x(xx)`? Both forms utilize application. We know that application has higher precedence than abstraction, but there's no abstraction here. Just two different ways to apply the application rule. Associativity solves this problem.

Associativity rules tell us, in cases where the precedence is the same, which parse we should choose.

Application is left-associative. Therefore, we group application expressions to the left (`((xx)x)` instead of to the right (`x(xx)`).

We also have the same problem with lambda abstractions. For example,

```
λx.xλx.x
```

Should we interpret this expression to mean `λx.(xλx.x)` or `(λx.x)(λx.x)`? In other words, how much of the sentence is included in the expression that comes after the first period?

Abstraction is right-associative. I like to think of this as meaning that "the period is greedy." The expression after the period extends as far to the right as makes sense logically. So the correct interpretation is actually `λx.(xλx.x)`.

These rules take a little time to internalize, but after some practice, you will eventually get the hang of them.

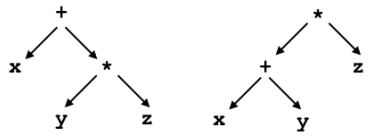


Figure 13: Without precedence, you'd have multiple derivations of the expression `x + y * z`. The left derivation is the one we want.

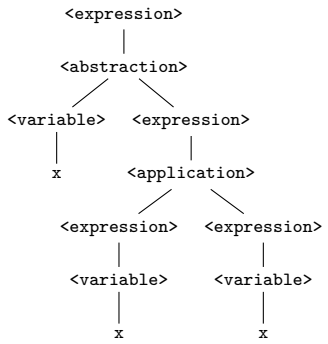


Figure 14: Correct derivation of the expression `λx.xx`.

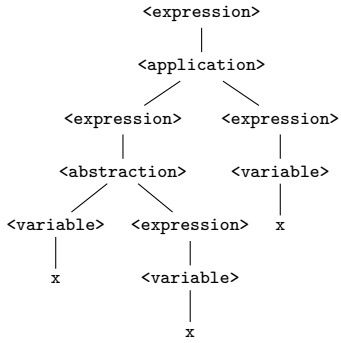


Figure 15: Incorrect derivation of the expression `λx.xx`.

Introduction to the Lambda Calculus, Part 2

In our second reading on the lambda calculus, we finish up discussing syntax and move on to the semantics of the language.

Ambiguity

In part 1, we looked at an expression `λx.xx` and asked what its *derivation tree* should look like. Should it look like the tree in Figure 16 or the tree in Figure 17?

I told you that, because *abstraction is right-associative*, the correct interpretation is the tree shown in Figure 16.

But what if you wanted to encode the second tree as a lambda program? Unless you write your program as separate pieces, such as

```
a is λx.x
```

and

```
b is x
```

such that the whole program is

```
ab
```

then there does not appear to be a way to encode the second tree using the syntax we are given. Adding parentheses to the language solves this problem.

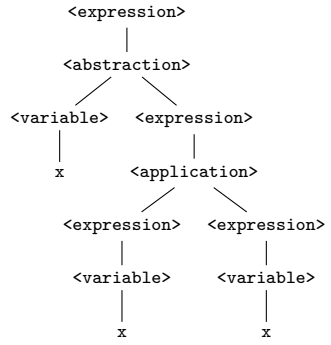


Figure 16: Correct derivation of the expression `λx.xx`.

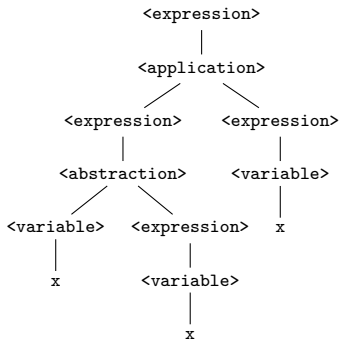


Figure 17: Incorrect derivation of the expression `λx.xx`.

Parentheses

Parentheses remove ambiguity. Let's start by introducing a simple axiom into our system:

$$[\![\texttt{(<expression>)}]\!] \equiv \texttt{<expression>}$$

This axiom says that "the meaning of" (the part inside the $[\![\ ]\!]$ symbols) any expression enclosed in parentheses "is identical to" ($\equiv$) that same expression without parentheses. With this rule, you can feel free

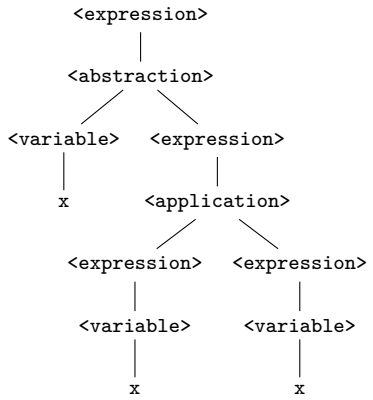
to surround an expression with parens. When an expression would not be made ambiguous by doing so, you can also remove them.

Let's augment our grammar so that we don't run into any more ambiguity traps.

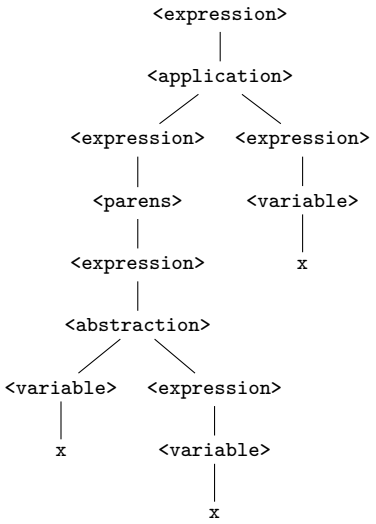
```
<expression>  ::= <variable>
                | <abstraction>
                | <application>
                | <parens>

<variable>    ::= x
<abstraction> ::= λ<variable>.<expression>
<application> ::= <expression><expression>
<parens>      ::= (<expression>)
```

Now we can avoid the problem with `λx.xx`. If we write `λx.xx`, the interpretation, because abstraction is right associative, is:



and if we write `(λx.x)x` then the interpretation is



It is conventional, among people who use the lambda calculus, to drop parenthesis whenever an interpretation is unambiguous. Googling for lambda expressions will often turn up scads of examples that omit parens. When you are aware of this fact, many of these examples will be easier to interpret.

Other extensions

Additional variables—in other words, not limiting ourselves to just the variable `x`—makes the grammar easier to work with. Here is our grammar augmented with additional variables:

```
<expression>  ::= <variable>
                | <abstraction>
                | <application>
                | <parens>

<variable>    ::= α ∈ { a ... z }
<abstraction> ::= λ<variable>.<expression>
<application> ::= <expression><expression>
<parens>      ::= (<expression>)
```

where, hopefully, it is clear that $\alpha \in$ `{ a ... z }` denotes *any letter*. E.g., any letter could be `x`, or `y`, or `b`, or `g`, etc.

Finally, to make the lambda calculus more immediately useful, let's add one more rule, `<arithmetic>`. Note that this rule is not strictly required, because although it may not be apparent to you, arithmetic can be encoded using all of the pieces we've already discussed. But learning those encodings can be difficult and they are not required to understand the lambda calculus, so we will put them off until later.

An example of arithmetic of the kind we're talking about might be an expression like `1 + 1`. The expression `1 + 1` is in *infix form*, because the operator (`+`) is in between the operands (`1` and `1`). You are accustomed to infix notation because of years of practice, but from a computational standpoint, it is a little bit of a hassle to work with. Instead, we will write all arithmetic in *prefix form*, which is easier to manipulate programmatically. In prefix form,

- The entire expression is enclosed within parentheses.
- The operator is the first term written inside the parentheses.
- All subsequent terms written after the operator, but still within the parentheses, are operands.

  Formally,

  ```
  <arithmetic> ::= (<op> <expression> ... <expression>)
  ```

For example, `1 + 1` is written as `(+ 1 1)`. Another example is `c` ÷ `z`, which is written `(÷ c z)`.

What is `<op>`? A simple formulation might include just `+`, `-`, ×, and ÷. I will stick to `+` and `-` in this class.

One nice thing about prefix form, which is why we're using it here, is that the order of operations is very clear. For example, with a little practice, you should have no difficulty computing `(+ 3 (* 5 4))`. Recall that the equivalent expression in "normal" arithmetic might be written as `3 + 5 * 4`, which is ambiguous if you don't remember your precedence and associativity rules from algebra class.

Here's an updated grammar.

```
<expression>   ::= <value>
                 |  <abstraction>
                 |  <application>
                 |  <parens>
                 |  <arithmetic>

<value>        ::= v ∈ ℕ
                 |  <variable>
<variable>     ::= α ∈ { a … z }
<abstraction>  ::= λ<variable>.<expression>
<application>  ::= <expression><expression>
<parens>       ::= (<expression>)
<arithmetic>   ::= (<op> <expression> ... <expression>)
<op>           ::= o ∈ { +, - }
```

Finally, notice that I added one more component, called `<value>`. What is `<value>`? In this language, `<value>` is either a number or an `<variable>`.

You can see why I don't introduce these complications all up front: the grammar is starting to look a little hairy. Still, if you remember that there are essentially three parts to the lambda calculus, *variables*, *abstractions*, and *applications*, you will be fine. To recap, we added:

1. Parentheses to remove ambiguity.
2. Extra variables.
3. Simple arithmetic in prefix form.

Abstract syntax

One of the tasks newcomers to the lambda calculus most struggle with is identifying parts of an expression. Before we talk about what the lambda calculus means, let's expand on our parsing skills. Given a

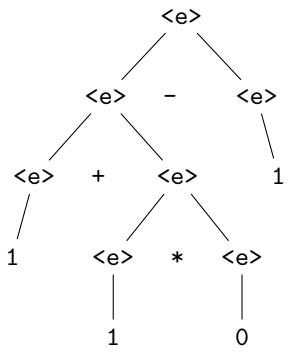
grammar and and expression, by now you can probably give me a derivation tree for that expression, or you can tell me that the expression is not a valid sentence in the language defined by the grammar. However, although derivation trees are useful, they are also cumbersome. They focus more on *how an expression is derived* and less on *what a sentence means*. Although the two are clearly related in some ways, sometimes we really just want to understand an expression's meaning.

When we want to understand the meaning of a sentence, we usually turn to an alternative kind of parse tree called an *abstract syntax tree* or AST. Abstract syntax makes the meaning of an expression quite clear. The term *abstract* means that we no longer care about all the details of a language's syntax; instead, we focus on the most important content. Typically, the content we care about is *data* and *operations*. In an AST, interior nodes are operations and leaf nodes are data.

As an example, let's look at a variation of a calculator grammar from one of our previous readings.

```
<e> ::= 0 | 1 | (<e> + <e>) | (<e> - <e>) | (<e> * <e>)
```

Suppose we have the expression `((1 + (1 * 0)) - 1)`. It has the following derivation:

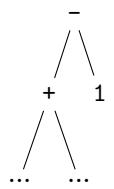


Instead of filling our tree with `<e>`, let's define our tree purely in terms of our three operations, `+`, `-`, and `*`, and our two number literals, `0` and `1`. If you need a little more precision than this, imagine we're using the following `type` definition from ML.

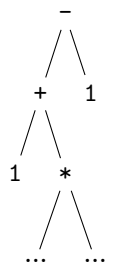
```
type Expr =
  | Zero
  | One
  | Addition of Expr * Expr
  | Subtraction of Expr * Expr
  | Multiplication of Expr * Expr
```

The top level operation in our derivation is subtraction, so our new tree will start like this:

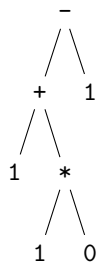
The expression on the left of the subtraction is addition, and the expression on the right of the subtraction is `1`.



The expression on the left of the addition is `1`, and the expression on the right of the addition is multiplication.



The expression on the left of the multiplication is `1`, and the expression on the right of the multiplication is `0`.



Observe that the meaning of this expression is much clearer. As noted before, all of the data is at the leaves and all of the operations are in the interior.

A big reason why we like ASTs is that they suggest a natural procedure for evaluating an expression: any operator whose operands are data can be evaluated. See if you can evaluate the tree above from the "bottom up." [46] Since you already know how to perform arithmetic, I hope you'll see that ASTs provide additional clarity about what to do with an expression. As you shall see, we will rely on ASTs to aid our understanding of the meaning of lambda calculus expressions.

[46] Hint: the only operation whose operands are data is `*`. After evaluating `*`, redraw the tree, replacing the `*` subtree with the result.

Semantics of the Lambda Calculus

Semantics is the study of *meaning*. In the context of programming languages, what we usually care about is how an expression can be converted into a sequence of mechanical steps that can be performed by a machine.

So how do we interpret the meaning of a lambda expression?

Unfortunately, the meaning of the word "interpret" is unclear. There are two meanings for the word "interpret":

1. To understand.
2. To evaluate.

In programming languages, and more generally in computer science, we tend to favor the latter definition of the word "interpret," i.e., to evaluate. Why? Because evaluation is a process that we can carry out on a machine.[47] In a well-designed programming language, there is no room for ambiguity. Precision is what sets programming languages apart from natural languages. It is why we can precisely describe programs and run them billions of times without error. But it also means, because they are so different from natural languages, that they are usually harder to learn.

[47] Maybe someday our study of artificial intelligence will allow computers to "understand" things, but we're not there yet. Part of the difficulty may be that we don't really know what "understanding" itself means.

Evaluation via term rewriting

The lambda calculus is an example of a *term rewriting system*. You've seen a term rewriting system before. The system of mathematics you learned in high school—algebra—is a term rewriting system. The big idea behind term rewriting systems is that, by following simple substitution rules, you can "solve" the system. In algebra, we mix term rewriting (substitution) and arithmetic (e.g., `1 + 2`) to solve for the value of a variable.

The amazing and surprising fact about the lambda calculus is that term rewriting is sufficient to *compute anything*. Yes, really, all you need to do is follow a few simple rewriting rules, just like in algebra, and you don't even need the arithmetical part.

That said, if you intend to compute "interesting" things—the kinds of programs we normally write—the language is inconvenient to work with. Although it is not as primitive as a Turing machine, it's pretty darned primitive. Alonzo Church's contribution was to show that the functions we most care about—the ones that are computable in principle—can all be written as lambda expressions. In this class, I prefer that you not have to work at such a primitive level, and so our version of the lambda calculus has "first class" support for arithmetic (`<arithmetic>`). Without it, writing useful programs requires a lot of work, but this one

simple extension makes the language useful without sacrificing much of its simplicity.

There are essentially two rewriting rules in the lambda calculus. To understand them, I need to introduce few things first: equivalence and bound/free variables.

Equivalence

In the lambda calculus, we say that two expressions are *equivalent* when they are *lexically identical*. In other words, when they have exactly the same string of characters. For example, the expressions `λx.x` and `λy.y` mean the same thing, but they are not equivalent because they *literally* have different letters in them. `λx.x` and `λx.x` *are* equivalent, however, because they are exactly the same.

We often talk a little informally about equivalence and you might hear someone say that `λx.x` and `λy.y` "are equivalent." What they mean is that, after rewriting, the two expressions can be made equivalent.

Bound and free variables

Depending on where a variable appears in a lambda expression, it is either *bound* or *free*.

What is a bound variable? A bound variable is a variable named in a lambda abstraction. For example, in the expression

```
λx.<expression>
```

where `<expression>` denotes some expression (that I don't care about at the moment), and where `x` is the bound variable. How do we know that `x` is bound? Well, that's precisely what a lambda abstraction *means*. You can read the expression `λx.<expression>` literally as saying "the variable `x` is bound in expression `<expression>`."

`x` is a variable, and when `x` is used inside of `<expression>`, its value takes on whatever value is given when the lambda abstraction is evaluated. Lambda abstractions are the precise mathematical meaning of what we're talking about when we *define a function* in a conventional programming language. To give you an analogy in a language with which you might be more familiar (Python), you can informally think of the above expression to mean almost exactly the same thing as the following program

```
def foo(x):
    <expression>
```

except that in the lambda calculus, functions don't have names (i.e., no `foo`).

Now it should make sense to you when I say that `x` is a bound variable. When we call `foo`, for example, `foo(3)`, we know that wherever we see `x` in the function body, we should substitute in the value `3`. Let's say we have the Python program:

```
plus_one(x):
    return x + 1
```

Then the expression

```
plus_one(3)
```

means

```
3 + 1
```

Until we actually *call* the function, `x` could pretty much be anything; it's just a parameter. Its value is tied to the calling of the function.

Back to the lambda calculus. In the following expression,

```
λx.x
```

all instances of `x` refer to the same value. How do we know? Because the lambda abstraction tells us that the value of any `x` within its scope (remember: abstraction is right-associative) is *bound* to the value of the parameter `x`.

What about the following expression?

```
λx.y
```

The lambda abstraction tells us that `x` is bound but it says nothing about `y`. In fact, we can't really make any assumptions about `y`. Therefore, we call `y` a *free* variable.

To understand a lambda expression, you must always determine whether every variable is bound or free. Be careful! Consider the following expression:

```
λx.λy.xy
```

Both `x` and `y` are bound. Why? Let's rewrite using parens. Let's start with the rightmost lambda. We know, because of right-associativity, that everything to the right of the last period must belong to the rightmost lambda. So,

```
λx.λy.(xy)
```

We also know that everything to the right of the leftmost period must belong to the first lambda.

```
λx.(λy.(xy))
```

Is the rightmost y bound or free? It is bound because it is within the scope of the λy.___ abstraction. What about the rightmost x? It is *also* bound, because it is within the scope of the λx.___ abstraction. If you don't believe me, just look at the outermost parens:

```
λx.(... x ...)
```

Here's a more complicated example with a free variable:

```
(λx.x)y
```

Can you spot which one is free? [48]

[48] It's y.

One more:

```
λx.λx.xx
```

Which variables are bound?

In this case, all the xes you see are bound. However, there are *two* x variables. Let's put in some parens to make the expression easier to understand.

```
λx.(λx.(xx))
```

So both x values are within the scope of both lambda abstractions. To which abstraction is x bound? In the lambda calculus, the *last abstraction wins*. Therefore, it is *as if* λx.(λx.(xx)) were written

```
λy.(λx.(xx))
```

and to give you an intuitive sense of this, this is more or less equivalent to the following (perfectly valid but slightly unusual) Python program,

```
def yfunc(y):
    def xfunc(x):
        return x(x)
    return xfunc
```

but, of course, without the function names.

Reductions

The rules used to rewrite expressions in the lambda calculus are called *reduction rules*. Reductions are the heart of what it means *to evaluate*, or more colloquially, "to execute," a lambda expression.

α Reduction

The first rewriting rule is called *alpha reduction*. Alpha reduction is a rule introduced to deal with ambiguity surrounding the use of variables. Specifically, alpha reduction relies on the property that, in the lambda calculus, *the given name* of a bound variable largely does not matter.

Let's look at a concrete example. Remember the identity function?

```
def identity(x):
    return x
```

It returns whatever we give it. Here's a Python interpreter session:

```
$ python
Python 2.7.15 (default, Aug 22 2018, 16:36:18)
[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def identity(x):
...     return x
...
>>> identity(3)
3
```

Hopefully no surprises there, right? What if, after we explored the above definition of `identity`, I asked you what the following function meant:

```
def identity2(z):
    return z
```

I suspect that you might be a little annoyed. Why? Because obviously `identity1` and `identity2` are the same function.

Remember, though, that in the lambda calculus, we have a strict notion of equivalence. Two expressions are equivalent if and only if they are exactly the same string. So even though `identity` in the lambda calculus is:

```
λx.x
```

and `identity2` in the lambda calculus is:

```
λy.y
```

and we can sort of squint and see that they're the same, that's not enough. We have to *prove* it.

α Equivalence

The notation `[a/b]<expression>` means "replace the variable `b` with variable `a` in `<expression>`." To perform alpha reduction, we rely on the following property of the lambda calculus, which is called "alpha equivalence":

$$[\![\lambda x_1.e]\!] =_\alpha [\![\lambda x_2.[x_2/x_1]e]\!]$$

where x_1 and x_2 are variables, and e is an expression.

This property says that the meaning of the expression of the form $\lambda x_1.e$ is the same when you replace it with an expression of the form $\lambda x_2.[x_2/x_1]e$. The two expressions are "alpha equivalent" (that's what $=_\alpha$ means). Since $[x_2/x_1]$ is not a valid lambda-calculus expression, you must continue replacement of x_1 with x_2 wherever you find it in *e*. You continue doing this until you can proceed no further with substitution.

Here's a proof that the two expressions are the same.

`λx.x`	given
`[y/x] λx.x`	alpha reduce x with y
`λy.[y/x]x`	step 1: rename outer x and continue to inner expression
`λy.y`	step 2: rename inner x

Therefore, `λx.x` $=_\alpha$ `λy.y`.

I mentioned before that sometimes you can "proceed no further with substitution". Substitution can be "blocked" by nested lambdas. Recall the expression from before:

```
λx.λx.xx
```

You might be wondering: is this expression equivalent to `λy.λx.yy`?

This is a perfect time to do alpha reduction. Let's replace `x` with `y`.

`λx.λx.xx`	given
`[y/x] λx.λx.xx`	alpha reduce x with y
`λy.[y/x] λx.xx`	step 1: rename outer x and continue to inner expression
`λy.λx.xx`	done (substitution blocked by `λx`)

We cannot rename the `x` inside the inner expression because it is a *different* `x` than the outer `x`. The inner lambda "redefines" `x`. Therefore `λx.λx.xx` is *not* equivalent to `λy.λx.yy`, at least not using alpha reduction.

When two expressions can be made equivalent using alpha reduction, we call them *alpha equivalent*.

β Reduction

There is one other kind of reduction called *beta reduction*. Beta reduction is the beating heart of the lambda calculus because it is, essentially, what it means to *call a function*. We refer to calling a lambda function *application*.

The simplest possible example uses the identity function:

```
λx.x
```

Let's "call" this function with a value. Say, `y`.

```
(λx.x)y
```

Recall that we use parentheses to make this expression unambiguous.

At a high level. This expression has two parts: *left* and *right*. The left side is `λx.x`. The right side is `y`. How do we know? The parens tell us.

What is the *one* grammar rule in the lambda calculus that lets us interpret an expression with a left part and a right part? It's this rule:

```
<application> ::= <expression><expression>
```

Colloquially, we call the left part the *function* and the right part the *argument*. Why? Because those two parts work just like the function and argument parts you might see in a conventional programming language.

```
(λx.x)y
```

works just like

```
def identity(x):
    return x

identity(y)
```

because we expect to get `y` back when we call the `identity` function with `y` as an argument[49].

[49] The exact Python equivalent to `(λx.x)y` is actually `(lambda x: x)(y)`. Python will complain that `y` is not defined if you do not define it somewhere; the lambda calculus is less strict because it doesn't really care if `y` is free. Python is an *eager* language, which essentially means that variables can never be free.

β Equivalence

Beta reduction is a substitution rule that achieves the same effect as calling a function. We again use the substitution operation, `[a/b] c` which means "substitute variable `b` with expression `a` in the expression `c`", but in the case of beta reduction, substitution *eliminates* both the function (the lambda abstraction) and its argument.

We rely on the following property, which is called "beta equivalence":

$$[\![(\lambda x_1.e)x_2]\!] =_\beta [\![[x_2/x_1]e]\!]$$

where x_1 is a variable, and x_2 and e are expressions.

This property says that an expression of the form $(\lambda x_1.e)x_2$ has the same meaning as an expression of the form $[x_2/x_1]e$. The two expressions are "beta equivalent" (that's what $=_\beta$ means). Since $[x_2/x_1]$ means "substitute x_2 for x_1 in e" and is not a valid lambda calculus expression, you must continue replacement until you can proceed no further. As with alpha reduction, redefinition of a variable inside a lambda "blocks" substitution.

Example:

`(λx.x)y`	given
`([y/x] x)`	β-reduce x with y; step 1: eliminate abstraction and argument
`(y)`	step 2: replace x with y
y	eliminate parens (because `<expression> = (<expression>)`)

Let's look at another example.

`(λx.xx)z`	given
`([z/x] xx)`	β-reduce x with z; step 1: eliminate abstraction and argument
`(zz)`	step 2: replace x with z
zz	eliminate parens

In the fourth step, we beta reduce inside the application `xx` because `[z/x]<expression><expression>` means `([z/x]<expression>)([z/x]<expression>)`.

Reduction order

In the lambda calculus, the order of reductions does not matter.[50] You are already familiar with this idea. Suppose I ask you to evaluate the polynomial $2x^2 + y/3$, where $x = 1$ and $y = 3$. Does it matter which variable you substitute first? Clearly the answer is no. We could first substitute x to obtain $2 + y/3$ and then y, yielding $2 + 1$. Or we could substitute y to obtain $2x^2 + 1$ and then x, also yielding $2 + 1$. The result is the same. A term rewriting system whose substitution order does not matter is *confluent*.

[50] This idea is called the Church-Rosser Theorem.

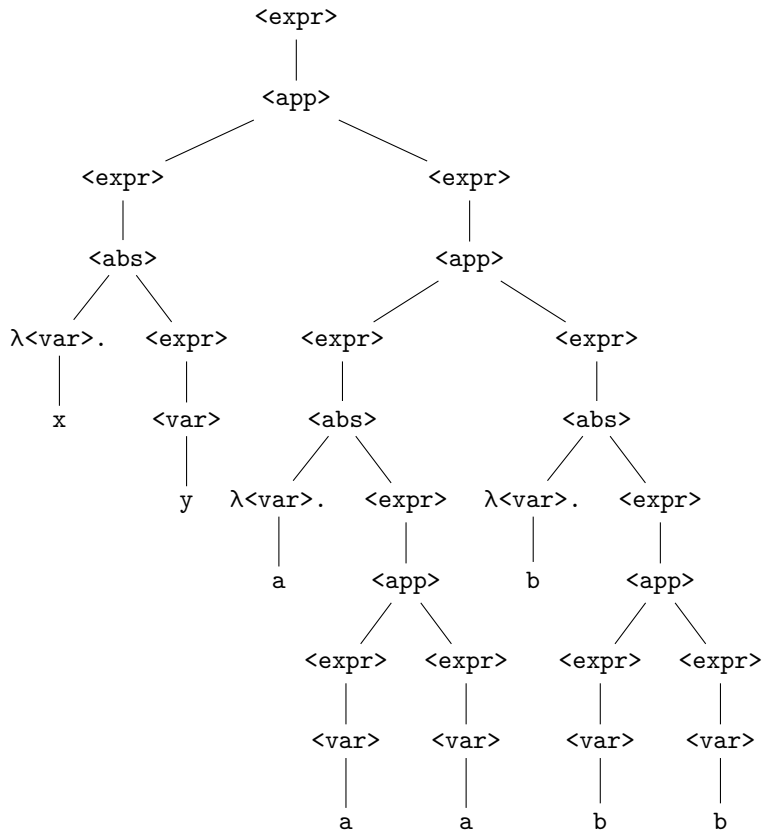
Likewise, the order of reductions does not matter in the lambda calculus. The lambda calculus is confluent. Let's look at an example of an expression where there is a choice about which reduction we can apply.

```
(λx.y)((λa.aa)(λb.bb))
```

It's probably hard for you to see where reductions can be applied in the above expression. Things are greatly clarified by drawing a lambda

expression's abstract syntax tree. Let's start by producing a derivation tree, then converting it into an abstract syntax tree like we did before with our arithmetic expression.[51]

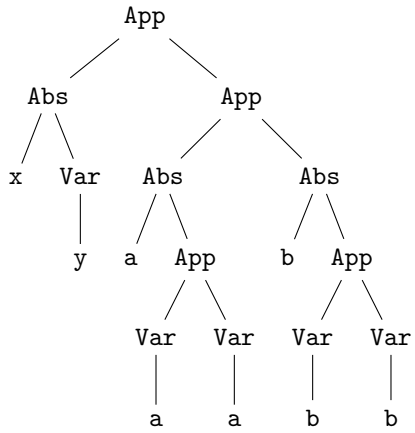
[51] You will eventually be able to produce ASTs directly from an expression without first having to draw a derivation tree.



That's a lot of drawing! Hopefully the AST is simpler and clearer. As before, we need to define our AST's operations and data. Suppose we use the following ML `type` definition for our tree, where a `char` is data and everything else is an operation of some kind.

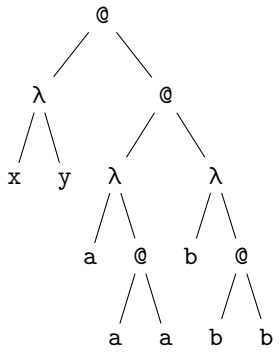
```
type Expr =
| Var of char
| Abs of char * Expr
| App of Expr * Expr
```

Applying the above definition to our expression, we obtain the following AST.



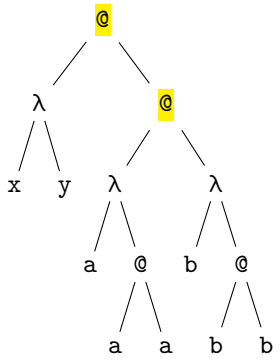
Much clearer, right? As you will see, I like to write lambda ASTs with a shorthand that makes them even easier to jot down. Everywhere we see `Var`, we simply replace it with its variable. Everywhere we see `Abs`, we write `λ`. Finally, everywhere we see `App`, we write `@`.[52]

[52] At-plication. Get it? No? OK, I think it's only amusing to me. Still, it's easier to write.



Identifying reducible expressions

Reducible expressions, or *redexes* for short, are the parts of a lambda expression where we can apply β-reductions. To find a redex, convert the expression to an AST and look for an application whose left side is an abstraction. If you think about this for a moment, this is like saying "look for a function definition that is being called." Here's the same AST with all the reducible applications highlighted.



Normal form

When do we stop doing reductions? We stop when there are no beta reductions left to do. One easy way to see that an expression can no longer be reduced is to look for redexes using our AST-drawing procedure. If there are no redexes, the expression is what we call a *normal form*.

For example, the following expressions are already in normal form:

```
x
xx
λx.y
xz
```

However, the following are not:

```
(λx.x)(λx.x)
(λx.λx.z)y
y(λx.xx)(λx.xx)
```

Try reducing the above expressions yourself. [53]

[53] The third expression has no normal form.

Normal order

In the tree above, the "outermost leftmost" reduction applies the argument `((λa.aa)(λb.bb))` to the function `(λx.y)`.[54] Always following the outermost leftmost beta reduction, at every step, is what we call the *normal order reduction.*

[54] "Outer" means *up* the tree in our diagrams.

Let's reduce this expression using the normal order.

`(λx.y)((λa.aa)(λb.bb))`	given
`([((λa.aa)(λb.bb))/x] y)`	β reduce `((λa.aa)(λb.bb))` for `x`
`y`	substitute and done

Applicative order

In the tree above, the "innermost leftmost" reduction applies the argument `(λb.bb)` to the function `(λa.aa)`.[55] Always following the innermost leftmost beta reduction, at every step, is what we call the *applicative order reduction*.

[55] "Inner" means *down* the tree in our diagrams.

Let's reduce this expression using the applicative order.

`(λx.y)((λa.aa)(λb.bb))`	given
`(λx.y)(([(λb.bb)/a] aa))`	β reduce `(λb.bb)` for `a`
`(λx.y)((λb.bb)(λb.bb))`	substitute
`(λx.y)((λa.[a/b] bb)(λb.bb))`	α reduce `a` for `b`
`(λx.y)((λa.aa)(λb.bb))`	uh-oh... we're back to where we started
...	

In this case, the applicative order reduction does not terminate. If you were following along, though, you know that we proved that the expression has a normal form because we were able to reduce it using the normal order.

Confluent, but with a catch

If an expression has a normal form, reductions can be applied in any order. Except, as you saw above, that's not the entire story. When we say "reductions can be applied in any order," we mean that you can never derive an incorrect expression by your choice of β reductions. Nevertheless, an unwise choice may reproduce an expression you had already evaluated.

Fortunately, there is an easy way to avoid the pain: choose the normal order. If a normal form exists for expression *e*, then the normal order reduction **will** find it. By contrast, if a normal form exists for expression *e*, then the applicative order reduction **may** find it.

You might be wondering why we even care about applicative order. It turns out that applicative order is equivalent to the order employed by most ordinary programming languages like Java and C! We call this kind of program evaluation *eager evaluation*. Very few languages utilize the normal order because it is difficult to implement, however there are examples. Haskell, for example, uses the normal order utilizing a form of evaluation we call *lazy evaluation*.

Nontermination

You might be thinking "I'm so glad I read this far in the course packet. Now all of my lambda calculus reductions will terminate!" I have sad news for you. As with ordinary programming languages, it is possible to write lambda expressions whose reductions will never terminate, regardless of your choice of reduction order. Consider the expression

`(λa.aa)(λb.bb)`. Since there is only one redex, the normal order and the applicative order are the same. Go ahead, give the reduction a try. I'll wait.[56]

[56] Forever.

When we say that an expression does not have a normal form, nontermination is what we mean. The expression `(λa.aa)(λb.bb)` does not have a normal form.

Remember, the lambda calculus was designed to capture all the essential parts of computation. If it is possible to write an infinitely-looping program like the following in an ordinary language,

```
while(true) {}
```

then we should not be surprised that we are also able to write nonterminating programs in the lambda calculus.

Higher-Order Functions

Now that you have some experience with the lambda calculus, let's return to F#. F# semantics are strongly influenced by the lambda calculus. Put another way, when the designers of F# asked themselves what the language should be capable of doing, the answer was "whatever the lambda calculus can do."[57]

For example, in the lambda calculus, we saw that we could apply function definitions to other function definitions.

```
(λa.a)(λb.b)
```

The above evaluates to

```
λb.b
```

Can we also pass definitions to definitions in F#? You bet![58]

```
> let id x = x;;
val id: x: 'a -> 'a

> let plusone n = n + 1;;
val plusone: n: int -> int

> id plusone;;
val it: (int -> int) = <fun:it@35>
```

The reason is that in a functional programming language, functions are *values*. Since functions take values as parameters, this means that in F#, a function can take a function as an argument.[59] Any language that treats functions as values is said to have *first-class functions*. First-class functions enable us to completely rethink how we structure programs.

As discussed earlier, we avoid looping constructs like `for` and `while` in F#, and often instead use recursion instead. Recursion is powerful, and it can express any kind of loop you want. But if you're like me, you've probably made the mistake of forgetting or messing up your base case, causing your program to run forever. For some programs, running forever is desirable.[60] More often, we *do not* want a loop to run forever.

[57] Strictly speaking, F# uses the *simple imperative polymorphism* model invented by Andrew K. Wright in 1995. This model is less expressive than the ordinary lambda calculus, but has the advantage of preventing certain logical paradoxes.

[58] Note that my F# example is slightly different than the lambda calculus expression. The equivalent F# would be `(fun a -> a)(fun b -> b)`, but F#'s type system is not capable of handling such expressions. The reason, called *value restriction*, is beyond the scope of this course. If you're enjoying F# and you want to dig deeper into its workings, see if you can understand why F#'s compiler makes this choice.

[59] As long as the expression type-checks, of course.

[60] Ideally, a web server should run forever. It waits until it receives a web request and then, after sending the webpage to the user, goes back to waiting.

Instead, we would like our algorithm to process every item in a finite collection like a list or an array. Processing every element in a finite collection is called *bounded iteration*. F# provides two features that rely on first-class functions which make it easier to write algorithms that use bounded iteration.

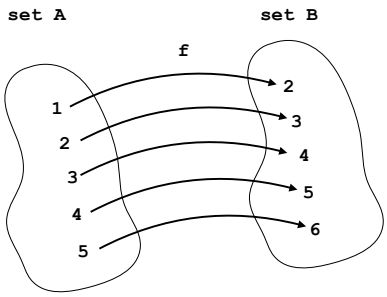


Figure 18: `f` is defined as `fun x -> x + 1`. When given the mapping function `f` and the set `A`, the `map` operation yields set `B`. Note that this idea is related to the mathematical definition of a map, but is not exactly the same.

The Map Operation

Suppose you want to convert every number in a list into a string. You can probably imagine how to do this using a `for` or a `while` loop. This is an example of bounded iteration because you know exactly how much work you need to do. The work is proportional to the length of the list.

There's another quality to this operation: it takes n elements and produces n elements. Whenever you need a bounded iteration that takes n elements and produces n elements, use `map`.

```
> List.map (fun x -> x.ToString()) [11;22;33;44];;
val it : string list = ["11"; "22"; "33"; "44"]
```

Observe that the type of the output need not be the type of the input. The expression above converts `int` values into `string` values.

Let's try to understand the type signature for `List.map`:

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

We will step through this definition, bit by bit. The parens tell us that the first argument's type is a `'a -> 'b`. Right away, because the type contains an `->`, we know that it is a function. What kind of function? A function from *some unknown type* `'a` to *some other unknown type* `'b`. While `'a` and `'b` *may* be the same type (e.g., both could be `int`), the type definition does not require that they *must* be the same type. We call this first parameter to `map` the *mapping function*. It "converts" or "maps" a given value to another value.

The next parameter has the type `'a list`. This parameter is the set of inputs. Because we used `List.map` in our example, the collection must be a `list`.

Finally, the last type is `'b list`, which is the type of the function's output. Hopefully this last type strikes you as intuitive: if you call a mapping function on a list, you will get a get a list back. Those two lists have different types because the mapping function can return any type. In our example, we converted a list of `int`s into a list of `string`s.

What makes `map` such a powerful idea is that it completely decouples the traversal of the list from the procedure that operates on the list's elements.

Why not try to see if you understand this idea? Try writing an expression that converts a list of integer strings into a list of integers; in other

words, the converse of the problem above. Your expression should look a little like:

```
> List.map ( ... your function here ... ) ["11";"22";"33";"44"];;
val it : int list = [11;22;33;44]
```

The Fold Operation

Another common form of iteration is to compute a single value from every element of a collection. This is different than `map` because although our operation still takes n values, it returns only a single value. You've almost certainly done something like this before. Here, we sum all of the elements in a list.

```
> let sum = List.fold (fun acc x -> acc + x) 0 [1;2;3;4;5];;
val sum: int = 15
```

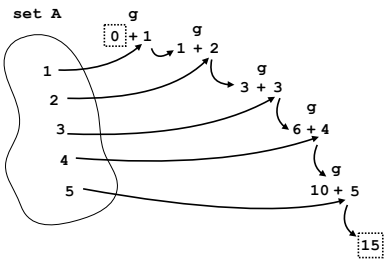


Figure 19: `g` is defined as `fun acc x -> acc + x`. When given the folding function `g`, the set `A`, and the initial value 0, the `fold` operation yields `15`.

Let's look at `fold`'s type signature:

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

The first parameter, called the *folding function*, takes two values, `'a` and `'b`, and returns `'a`. In our example `fold` above, the function takes two values and returns their sum. The first parameter to a folding function is called the *accumulator*: it's where we store the result of the operation from one iteration of `fold` to the next. The second parameter to the folding function is a single element, taken from a list.

The second parameter to `fold` is the *initial value of the accumulator*. Since we want to sum, in this case, we set our initial value to `0`.

The last parameter to `fold` is the list. This is where the argument for the folding function's `'b` parameter comes from. Finally, the return value is a `'a`, because that's the type of our accumulator. When there are no more elements in the list, the accumulator is returned.

Here, we convert strings into numbers and sum them:

```
> List.fold (fun acc x -> acc + int x) 0 ["1";"2";"3";"4";"5"];;
val it : int = 15
```

`fold` is a remarkably flexible function. It is not limited to lists. One can fold arrays, trees, graphs, etc. In fact, with a little cleverness, one can even implement `map` using fold.[61]

The `fold` we've discussed here is technically a form called "fold left." "Left" refers to the fact that `fold` works through the elements of a list from the beginning to the end. We usually visualize the beginning of a list as being on the left and the end on the right. Folding "right" does the converse, taking elements from the end first.

[61] If you want a little challenge, see if you can figure it out!

In LISP, the original functional programming language, `fold` is called `reduce`. When you pair mapping and folding together, you get a form of computation called `map-reduce`. Google's MapReduce framework, which is a platform for fault-tolerant, massively parallel computation is so called because all computation must be in `map-reduce` form.

Forward pipe

Forward pipe, `|>`, is my single favorite feature of F#. It allows one to build sophisticated data-processing pipelines that are easy to write and easy to read. `|>` passes the result of the left side to the function on the right side. For example, instead of writing:

```
List.map (fun x -> x + 1) [1;2;3;4]
```

one can write:

```
[1;2;3;4] |> List.map (fun x -> x + 1)
```

I find the latter easier to read, but the benefit really becomes apparent when one chains multiple operations.

```
> ["1";"2";"3";"4"]
- |> List.map (fun x -> int x)
- |> List.map (fun x -> x + 1)
- |> List.fold (fun acc x -> acc * x) 1
- ;;
val it: int = 120
```

The above converts each string containing an integer into an integer, adds one to each, and then multiplies them all together.

Proof by Reduction

> An engineer and a mathematician were hiking when they were suddenly attacked by a bear. The engineer grabbed a stick and, yelling and stabbing wildly with the stick, managed to fight off the bear. The next day, the two went out for another hike, and again, they were attacked by the same bear. This time, the mathematician, realizing that he was the closest to a stick, picked it up and handed it to the engineer, thereby reducing the bear problem to a previously solved problem.

Reduction proofs[62] are a little counterintuitive. When we construct them for the purposes of computability proofs, we will always have two facts in mind:

1. We want to disprove a fact about some problem of interest, A (e.g., "A is computable.")
2. We already know a fact something about some other, possibly related problem, B (e.g., "B is not computable").

Like all formal tools, reduction proofs are a template (a "form"). The trick is to recognize when the problem fits the mold. When the tool is used correctly, out pops the answer.

Here's the template we're going to follow. Let P be a logical proposition (a statement that is either true or false), and let Q be another logical proposition implied by P. In other words,

$$P \Rightarrow Q$$

For example, P could be the proposition "it is sunny outside." Q could be "it is not snowing." If we think that one implies the other, then we would read $P \Rightarrow Q$ as "if it is sunny outside, then it is not snowing." This statement is clearly true. P really does imply Q. However, since this is Williamstown, if you all look outside, it might actually be snowing. If that's the case, it cannot be sunny outside.

The above example should suggest to you that one way you might try to disprove a statement P is to show that an implied statement Q is false. In other words, if we claim $P \Rightarrow Q$, and P really implies Q, and then we show $\neg Q$, then it must also be the case that $\neg P$. (If you're having trouble

[62] Reduction proofs should not be confused with lambda reductions. Although they both share the word "reduction," they are completely unrelated topics.

seeing why I am allowed to use this trick, see the derivation from first principles at the bottom of this section.)

Finding a reduction

Let's apply this template. What is the problem of interest? Consider the following question: Is it possible to write a function, $\texttt{halt}_0$? When given a program `p` and an input `i`, $\texttt{halt}_0$ returns `true` if and only if `p(i)` does not halt.

Given about what we know about computability (i.e., that `halt` is not computable), we should have a nagging suspicion that $\texttt{halt}_0$ is also not computable. But can we set up a logical implication of the above form to *prove* that $\texttt{halt}_0$ is not computable? Indeed we can. Remember—the key is to imply something that we know *cannot* be true.

Let's start with a fact that we know cannot be true. Q: "`halt` is computable."

Now, can we show that Q follows logically from the fact that we want to disprove? P: "$\texttt{halt}_0$ is computable."

We're going to use the same $P \Rightarrow Q$ logic trick as in our snowing example above, and show that if P is true, P logically implies Q. This is where reductions fit in. A reduction is an *algorithm* that turns one problem into another problem. Why do we want an algorithm? Well, last time I checked, if computers did one thing well, it was logic. So if one can write an algorithm for transforming problems, it's logical, and a computer really could do it.

Remember when you learned about proving things using mathematical induction? When proving the inductive step, which is an implication of the form $P \Rightarrow Q$, recall that you were allowed to assume P. Since we are also attempting to prove an implication, we also get to assume P is true. Remember that P is "$\texttt{halt}_0$ is computable."

Here's an algorithm (in Python) that turns instances of `halt` into $\texttt{halt}_0$.

```
def halt(p,i):
  return not halt0(p,i)
```

If we assume that P is computable, we really could have a $\texttt{halt}_0$ function. Some really smart person could have coded it up and stuck it in a library for us. So the above function, `halt`, should be possible, right? I didn't do anything fancy. I just followed the rules of Python. `halt` just calls $\texttt{halt}_0$ and negates the result.

Looking back at our statements,
Q: "`halt` is computable"
and
P: "$\texttt{halt}_0$ is computable"

what the reduction just showed is that $P \Rightarrow Q$. We can't avoid $P \Rightarrow Q$, because look, I just made Q happen using P. Therefore, it is true that P implies Q.

But we also know, because I showed you in class (or if you're reading this early, I will), is that Q cannot be true. `halt` is not computable. $\neg Q$.

Therefore $\texttt{halt}_0$ is not computable. $\neg P$.

Why does $\neg Q \Rightarrow \neg P$ when $P \Rightarrow Q$?

We can derive what happens to the antecedent (P) of an implication ($P \Rightarrow Q$) when we know that the consequent (Q) is not true. I claim that $P \Rightarrow Q$ is logically equivalent to the statement $\neg P \vee Q$. We can prove this equivalence rigorously by working out a truth table.[63] $\neg P \vee Q$ is easier to work with, because it gets rid of the pesky implication symbol (whatever *that* means).

[63] When I am confused about what conditionals do in code, I sometimes work out truth tables. A fellow student once mocked me for using this trick. "Dan has to write out the truth tables to understand it! Ha ha." This is a silly thing to be elitist about. Programming is hard. I can still solve the problem. More importantly, I am not confused.

P	Q	$\neg P \vee Q$	$P \Rightarrow Q$
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

We know that $\neg P \vee Q$ is true, just as we do with our Python program above. Let's start our proof with that fact.

$\neg P \vee Q = \texttt{true}$	given
$\neg P \vee \texttt{false} = \texttt{true}$	because Q is false
$\neg P = \texttt{true}$	because "anything" $\vee$ `false` is just "anything"
$P = \texttt{false}$	by negation

Therefore, if $P \Rightarrow Q$ itself is true and Q is false, then P must be false.

How to Fix a Motorcycle

Zen and the Art of Motorcycle Maintenance, by Robert Pirsig, is an unusual work of fiction inspired by real life events. The story follows a father and son on a motorcycle trip across the United States. During their travels, we hear the narrator's internal dialog as he considers a question of great importance to him: *what is quality*? Like Plato's *Dialogues*, the following is a work of philosophy, but what I love about it is that the author presents his thesis in the form of a discourse on motorcycle maintenance.

You are reading this excerpt because it is relevant—and in my opinion, of primary importance—to the technical work of programming a computer. Note that I have omitted some passages to make the reading shorter, and because the author intersperses the text with comments about what he is seeing and doing *as he rides his motorcycle*.

Note that Pirsig uses the term "Chautauqua" throughout the text. A *Chautauqua* is a form of storytelling education that was popular in the early 20th century in the US. You can just mentally substitute the word "dialogue" instead. [—ed.]

I like the word "gumption" because it's so homely and so forlorn and so out of style it looks as if it needs a friend and isn't likely to reject anyone who comes along. It's an old Scottish word, once used a lot by pioneers, but which, like "kin," seems to have all but dropped out of use. I like it also because it describes exactly what happens to someone who connects with quality. He gets filled with gumption.

The Greeks called it *enthousiasmos*, the root of "enthusiasm." which means literally "filled with theos," or God, or quality. See how that fits?

A person filled with gumption doesn't sit around dissipating and stewing about things. He's at the front of the train of his own awareness, watching to see what's up the track and meeting it when it comes. That's gumption.

***

The gumption-filling process occurs when one is quiet long enough to see and hear and feel the real universe, not just one's own stale opinions about it. But it's nothing exotic. That's why I like the word.

You see it often in people who return from long, quiet fishing trips. Often they're a little defensive about having put so much time to "no account" because there's no intellectual justification for what they've been doing. But the returned fisherman usually has a peculiar abundance of gumption, usually for the very same things he was sick to death of a few weeks before. He hasn't been wasting time. It's only our limited cultural viewpoint that makes it seem so.

If you're going to repair a motorcycle, an adequate supply of gumption is the first and most important tool.

If you haven't got that you might as well gather up all the other tools and put them away, because they won't do you any good.

Gumption is the psychic gasoline that keeps the whole thing going. If you haven't got it there's no way the motorcycle can possibly be fixed. But if you have got it and know how to keep it there's absolutely no way in this whole world that motorcycle can keep from getting fixed. It's bound to happen. Therefore the thing that must be monitored at all times and preserved before anything else is the gumption.

This paramount importance of gumption solves a problem of format of this Chautauqua. The problem has been how to get off the generalities. If the Chautauqua gets into the actual details of fixing one individual machine the chances are overwhelming that it won't be your make and model and the information will be not only useless but dangerous, since information that fixes one model can sometimes wreck another. For detailed information of an objective sort, a separate shop manual for the specific make and model of machine must be used. In addition, a general shop manual such as *Audel's Automotive Guide* fills in the gaps.

But there's another kind of detail that no shop manual goes into but that is common to all machines and can be given here. This is the detail of the quality relationship, the gumption relationship, between the machine and the mechanic, which is just as intricate as the machine itself. Throughout the process of fixing the machine things always come up, low-quality things, from a dusted knuckle to an accidentally ruined "irreplaceable" assembly. These drain off gumption, destroy enthusiasm and leave you so discouraged you want to forget the whole business. I call these things "gumption traps."

There are hundreds of different kinds of gumption traps, maybe thousands, maybe millions. I have no way of knowing how many I don't know. I know it seems as though I've stumbled into every kind of gumption trap imaginable. What keeps me from thinking I've hit them all is that with every job I discover more. Motorcycle maintenance gets frustrating. Angering. Infuriating. That's what makes it interesting.

***

What I have in mind now is a catalog of "Gumption Traps I Have Known." I want to start a whole new academic field, gumptionology, in which these traps are sorted, classified, structured into hierarchies and interrelated for the edification of future generations and the benefit of all mankind.

`Gumptionology 101 ... An examination of affective, cognitive and psychomotor blocks in the perception of quality relationships ... 3 cr, VII, MWF.` I'd like to see that in a college catalog somewhere.

In traditional maintenance gumption is considered something you're born with or have acquired as a result of good upbringing. It's a fixed commodity. From the lack of information about how one acquires this gumption one might assume that a person without any gumption is a hopeless case.

In nondualistic maintenance gumption isn't a fixed commodity. It's variable, a reservoir of good spirits that can be added to or subtracted from. Since it's a result of the perception of quality, a gumption trap, consequently, can be defined as anything that causes one to lose sight of quality, and thus lose one's enthusiasm for what one is doing. As one might guess from a definition as broad as this, the field is enormous and only a beginning sketch can be attempted here.

As far as I can see there are two main types of gumption traps. The first type is those in which you're thrown off the quality track by conditions that arise from external circumstances, and I call these "setbacks." The second type is traps in which you're thrown off the quality track by conditions that are primarily within yourself. These I don't have any generic name for . . . "hang-ups" I suppose. I'll take up the externally caused

setbacks first.

The first time you do any major job it seems as though the out-of-sequence-reassembly setback is your biggest worry. This occurs usually at a time when you think you're almost done. After days of work you finally have it all together except for: What's this? A connecting-rod bearing liner?! How could you have left that out? Oh Jesus, everything's got to come apart again! You can almost hear the gumption escaping. Psssssssssssss.

There's nothing you can do but go back and take it all apart again—after a rest period of up to a month that allows you to get used to the idea.

There are two techniques I use to prevent the out-of-sequence-reassembly setback. I use them mainly when I'm getting into a complex assembly I don't know anything about.

It should be inserted here parenthetically that there's a school of mechanical thought which says I shouldn't be getting into a complex assembly I don't know anything about. I should have training or leave the job to a specialist. That's a self-serving school of mechanical eliteness I'd like to see wiped out. That was a "specialist" who broke [. . .] this machine. I've edited manuals written to train specialists for IBM, and what they know when they're done isn't that great. You're at a disadvantage the first time around and it may cost you a little more because of parts you accidentally damage, and it will almost undoubtedly take a lot more time, but the next time around you're way ahead of the specialist. You, with gumption, have learned the assembly the hard way and you've a whole set of good feelings about it that he's unlikely to have.

Anyway, the first technique for preventing the out-of-sequence-reassembly gumption trap is a notebook in which I write down the order of disassembly and note anything unusual that might give trouble in reassembly later on. This notebook gets plenty grease-smeared and ugly. But a number of times one or two words in it that didn't seem important when written down have prevented damage and saved hours of work. The notes should pay special attention to left-hand and right-hand and up-and-down orientations of parts, and color coding and positions of wires. If incidental parts look worn or damaged or loose this is the time to note it so that you can make all your parts purchases at the same time.

The second technique for preventing the out-of-sequence-reassembly gumption trap is newspapers opened out on the floor of the garage on which all the parts are laid left-to-right and top-to-bottom in the order in which you read a page. That way when you put it back together in reverse order the little screws and washers and pins that can be easily overlooked are brought to your attention as you need them.

Even with all these precautions, however, out-of-sequence-reassemblies sometimes occur and when they do you've got to watch the gumption. Watch out for gumption desperation, in which you hurry up wildly in an effort to restore gumption by making up for lost time. That just creates more mistakes. When you first see that you have to go back and take it apart all over again it's definitely time for that long break.

It's important to distinguish from these the reassemblies that were out of sequence because you lacked certain information. Frequently the whole reassembly process becomes a cut-and-try technique in which you have to take it apart to make a change and then put it together again to see if the change works. If it doesn't work, that isn't a setback because the information gained is a real progress.

But if you've made just a plain old dumb mistake in reassembly, some gumption can still be salvaged by the knowledge that the second disassembly and reassembly is likely to go much faster than the first one. You've unconsciously memorized all sorts of things you won't have to relearn.

***

The intermittent failure setback is next. In this the thing that is wrong becomes right all of a sudden just as you start to fix it. Electrical short circuits are often in this class. The short occurs only when the machine's bouncing around. As soon as you stop everything's okay. It's almost impossible to fix it then. All you can do is try to get it to go wrong again and if it won't, forget it. Intermittents become gumption traps when they fool you into thinking you've really got the machine fixed. It's always a good idea on any job to wait a few hundred miles before coming to that conclusion. They're discouraging when they crop up again and again, but when they do you're no worse off than someone who goes to a commercial mechanic. In fact you're better off. They're much more of a gumption trap for the owner who has to drive his machine to the shop again and again and never get satisfaction. On your own machine you can study them over a long period of time, something a commercial mechanic can't do, and you can just carry around the tools you think you'll need until the intermittent happens again, and then, when it happens, stop and work on it.

When intermittents recur, try to correlate them with other things the cycle is doing. Do the misfires, for example, occur only on bumps, only on turns, only on acceleration? Only on hot days? These correlations are clues for cause-and-effect hypotheses. In some intermittents you have to resign yourself to a long fishing expedition, but no matter how tedious that gets it's never as tedious as taking the machine to a commercial mechanic five times. I'm tempted to go into long detail about "Intermittents I Have Known" with a blow-by-blow description of how these were solved. But this gets like those fishing stories, of interest mainly to the fisherman, who doesn't quite catch on to why everybody yawns. He enjoyed it.

Next to misassemblies and intermittents I think the most common external gumption trap is the parts setback. Here a person who does his own work can get depressed in a number of ways. Parts are something you never plan on buying when you originally get the machine. Dealers like to keep their inventories small. Wholesalers are slow and always understaffed in the spring when everybody buys motorcycle parts.

The pricing on parts is the second part of this gumption trap. It's a well-known industrial policy to price the original equipment competitively, because the customer can always go somewhere else, but on parts to overprice and clean up. The price of the part is not only jacked up way beyond its new price; you get a special price because you're not a commercial mechanic. This is a sly arrangement that allows the commercial mechanic to get rich by putting in parts that aren't needed.

One more hurdle yet. The part may not fit. Parts lists always contain mistakes. Make and model changes are confusing. Out-of-tolerance parts runs sometimes get through quality control because there's no operating checkout at the factory. Some of the parts you buy are made by specialty houses who don't have access to the engineering data needed to make them right. Sometimes they get confused about make and model changes. Sometimes the parts man you're dealing with jots down the wrong number. Sometimes you don't give him the right identification. But it's always a major gumption trap to get all the way home and discover that a new part won't work.

The parts traps may be overcome by a combination of a number of techniques. First, if there's more than one supplier in town by all means choose the one with the most cooperative parts man. Get to know him on a first-name basis. Often he will have been a mechanic once himself and can provide a lot of information you need.

Keep an eye out for price cutters and give them a try. Some of them have good deals. Auto stores and mail-order houses frequently stock the commoner cycle parts at prices way below those of the cycle dealers. You can buy roller chain from chain manufacturers, for example, at way below the inflated cycle-shop prices.

Always take the old part with you to prevent getting a wrong part. Take along some machinist's calipers for comparing dimensions.

Finally, if you're as exasperated as I am by the parts problem and have some money to invest, you can

take up the really fascinating hobby of machining your own parts. I have a little 6-by-18-inch lathe with a milling attachment and a full complement of welding equipment: arc, heli-arc, gas and mini-gas for this kind of work. With the welding equipment you can build up worn surfaces with better than original metal and then machine it back to tolerance with carbide tools. You can't really believe how versatile that lathe-plus-milling-plus-welding arrangement is until you've used it. If you can't do the job directly you can always make something that will do it. The work of machining a part is very slow, and some parts, such as ball bearings, you're never going to machine, but you'd be amazed at how you can modify parts designs so that you can make them with your equipment, and the work isn't nearly as slow or frustrating as a wait for some smirking parts man to send away to the factory. And the work is gumption building, not gumption destroying. To run a cycle with parts in it you've made yourself gives you a special feeling you can't possibly get from strictly store-bought parts.

***

Well, those were the commonest setbacks I can think of: out-of-sequence reassembly, intermittent failure and parts problems. But although setbacks are the commonest gumption traps they're only the external cause of gumption loss. Time now to consider some of the internal gumption traps that operate at the same time.

As the course description of gumptionology indicated, this internal part of the field can be broken down into three main types of internal gumption traps: those that block affective understanding, called "value traps"; those that block cognitive understanding, called "truth traps"; and those that block psychomotor behavior, called "muscle traps." The value traps are by far the largest and the most dangerous group.

Of the value traps, the most widespread and pernicious is value rigidity. This is an inability to revalue what one sees because of commitment to previous values. In motorcycle maintenance, you must rediscover what you do as you go. Rigid values make this impossible.

The typical situation is that the motorcycle doesn't work. The facts are there but you don't see them. You're looking right at them, but they don't yet have enough value. [. . .] Quality, value, creates the subjects and objects of the world. The facts do not exist until value has created them. If your values are rigid you can't really learn new facts.

This often shows up in premature diagnosis, when you're sure you know what the trouble is, and then when it isn't, you're stuck. Then you've got to find some new clues, but before you can find them you've got to clear your head of old opinions. If you're plagued with value rigidity you can fail to see the real answer even when it's staring you right in the face because you can't see the new answer's importance.

The birth of a new fact is always a wonderful thing to experience. It's dualistically called a "discovery" because of the presumption that it has an existence independent of anyone's awareness of it. When it comes along, it always has, at first, a low value. Then, depending on the value-looseness of the observer and the potential quality of the fact, its value increases, either slowly or rapidly, or the value wanes and the fact disappears.

The overwhelming majority of facts, the sights and sounds that are around us every second and the relationships among them and everything in our memory. . . these have no quality, in fact have a negative quality. If they were all present at once our consciousness would be so jammed with meaningless data we couldn't think or act. So we preselect on the basis of quality [. . .].

What you have to do, if you get caught in this gumption trap of value rigidity, is slow down. . . you're going to have to slow down anyway whether you want to or not. . . but slow down deliberately and go over ground

that you've been over before to see if the things you thought were important were really important and to—well—just stare at the machine. There's nothing wrong with that. Just live with it for a while. Watch it the way you watch a line when fishing and before long, as sure as you live, you'll get a little nibble, a little fact asking in a timid, humble way if you're interested in it. That's the way the world keeps on happening. Be interested in it.

At first try to understand this new fact not so much in terms of your big problem as for its own sake. That problem may not be as big as you think it is. And that fact may not be as small as you think it is. It may not be the fact you want but at least you should be very sure of that before you send the fact away. Often before you send it away you will discover it has friends who are right next to it and are watching to see what your response is. Among the friends may be the exact fact you are looking for.

After a while you may find that the nibbles you get are more interesting than your original purpose of fixing the machine. When that happens you've reached a kind of point of arrival. Then you're no longer strictly a motorcycle mechanic, you're also a motorcycle scientist, and you've completely conquered the gumption trap of value rigidity.

***

I keep wanting to go back to that analogy of fishing for facts. I can just see somebody asking with great frustration, "Yes, but which facts do you fish for? There's got to be more to it than that."

But the answer is that if you know which facts you're fishing for you're no longer fishing. You've caught them. I'm trying to think of a specific example.

All kinds of examples from cycle maintenance could be given, but the most striking example of value rigidity I can think of is the old South Indian Monkey Trap, which depends on value rigidity for its effectiveness. The trap consists of a hollowed-out coconut chained to a stake. The coconut has some rice inside which can be grabbed through a small hole. The hole is big enough so that the monkey's hand can go in, but too small for his fist with rice in it to come out. The monkey reaches in and is suddenly trapped. . . by nothing more than his own value rigidity. He can't revalue the rice. He cannot see that freedom without rice is more valuable than capture with it. The villagers are coming to get him and take him away. They're coming closer—closer!—now! What general advice. . . not specific advice. . . but what general advice would you give the poor monkey in circumstances like this?

Well, I think you might say exactly what I've been saying about value rigidity, with perhaps a little extra urgency. There is a fact this monkey should know: if he opens his hand he's free. But how is he going to discover this fact? By removing the value rigidity that rates rice above freedom. How is he going to do that? Well, he should somehow try to slow down deliberately and go over ground that he has been over before and see if things he thought were important really were important and, well, stop yanking and just stare at the coconut for a while. Before long he should get a nibble from a little fact wondering if he is interested in it. He should try to understand this fact not so much in terms of his big problem as for its own sake. That problem may not be as big as he thinks it is. That fact may not be as small as he thinks it is either. That's about all the general information you can give him.

***

The next one is important. It's the internal gumption trap of ego. Ego isn't entirely separate from value rigidity but one of the many causes of it.

If you have a high evaluation of yourself then your ability to recognize new facts is weakened. Your ego isolates you from the quality reality. When the facts show that you've just goofed, you're not as likely to admit it. When false information makes you look good, you're likely to believe it. On any mechanical repair job ego comes in for rough treatment. You're always being fooled, you're always making mistakes, and a mechanic who has a big ego to defend is at a terrific disadvantage. If you know enough mechanics to think of them as a group, and your observations coincide with mine, I think you'll agree that mechanics tend to be rather modest and quiet. There are exceptions, but generally if they're not quiet and modest at first, the work seems to make them that way. And skeptical. Attentive, but skeptical, But not egoistic. There's no way to bullshit your way into looking good on a mechanical repair job, except with someone who doesn't know what you're doing.

I was going to say that the machine doesn't respond to your personality, but it does respond to your personality. It's just that the personality that it responds to is your real personality, the one that genuinely feels and reasons and acts, rather than any false, blown-up personality images your ego may conjure up. These false images are deflated so rapidly and completely you're bound to be very discouraged very soon if you've derived your gumption from ego rather than quality.

If modesty doesn't come easily or naturally to you, one way out of this trap is to fake the attitude of modesty anyway. If you just deliberately assume you're not much good, then your gumption gets a boost when the facts prove this assumption is correct. This way you can keep going until the time comes when the facts prove this assumption is incorrect.

Anxiety, the next gumption trap, is sort of the opposite of ego. You're so sure you'll do everything wrong you're afraid to do anything at all. Often this, rather than "laziness," is the real reason you find it hard to get started. This gumption trap of anxiety, which results from overmotivation, can lead to all kinds of errors of excessive fussiness. You fix things that don't need fixing, and chase after imaginary ailments. You jump to wild conclusions and build all kinds of errors into the machine because of your own nervousness. These errors, when made, tend to confirm your original underestimation of yourself. This leads to more errors, which lead to more underestimation, in a self-stoking cycle.

The best way to break this cycle, I think, is to work out your anxieties on paper. Read every book and magazine you can on the subject. Your anxiety makes this easy and the more you read the more you calm down. You should remember that it's peace of mind you're after and not just a fixed machine.

When beginning a repair job you can list everything you're going to do on little slips of paper which you then organize into proper sequence. You discover that you organize and then reorganize the sequence again and again as more and more ideas come to you. The time spent this way usually more than pays for itself in time saved on the machine and prevents you from doing fidgety things that create problems later on.

You can reduce your anxiety somewhat by facing the fact that there isn't a mechanic alive who doesn't louse up a job once in a while. The main difference between you and the commercial mechanics is that when they do it you don't hear about it. . . just pay for it, in additional costs prorated through all your bills. When you make the mistakes yourself, you at]east get the benefit of some education.

Boredom is the next gumption trap that comes to mind. This is the opposite of anxiety and commonly goes with ego problems. Boredom means you're off the quality track, you're not seeing things freshly, you've lost your "beginner's mind" and your motorcycle is in great danger. Boredom means your gumption supply is low and must be replenished before anything else is done.

When you're bored, stop! Go to a show. Turn on the TV. Call it a day. Do anything but work on that machine. If you don't stop, the next thing that happens is the Big Mistake, and then all the boredom plus the Big Mistake

combine together in one Sunday punch to knock all the gumption out of you and you are really stopped.

My favorite cure for boredom is sleep. It's very easy to get to sleep when bored and very hard to get bored after a long rest. My next favorite is coffee. I usually keep a pot plugged in while working on the machine. If these don't work it may mean deeper quality problems are bothering you and distracting you from what's before you. The boredom is a signal that you should turn your attention to these problems... that's what you're doing anyway... and control them before continuing on the motorcycle.

For me the most boring task is cleaning the machine. It seems like such a waste of time. It just gets dirty again the first time you ride it. [My friend] John always kept his BMW spic and span. It really did look nice, while mine's always a little ratty, it seems.

***

One solution to boredom on certain kinds of jobs such as greasing and oil changing and tuning is to turn them into a kind of ritual. There's an esthetic to doing things that are unfamiliar and another esthetic to doing things that are familiar. I have heard that there are two kinds of welders: production welders, who don't like tricky setups and enjoy doing the same thing over and over again; and maintenance welders, who hate it when they have to do the same job twice. The advice was that if you hire a welder make sure which kind he is, because they're not interchangeable. I'm in that latter class, and that's probably why I enjoy troubleshooting more than most and dislike cleaning more than most. But I can do both when I have to and so can anyone else. When cleaning I do it the way people go to church... not so much to discover anything new, although I'm alert for new things, but mainly to reacquaint myself with the familiar. It's nice sometimes to go over familiar paths.

Zen has something to say about boredom. Its main practice of "just sitting" has got to be the world's most boring activity [...]. You don't do anything much; not move, not think, not care. What could be more boring? Yet in the center of all this boredom is the very thing Zen Buddhism seeks to teach. What is it? What is it at the very center of boredom that you're not seeing?

Impatience is close to boredom but always results from one cause: an underestimation of the amount of time the job will take. You never really know what will come up and very few jobs get done as quickly as planned. Impatience is the first reaction against a setback and can soon turn to anger if you're not careful.

Impatience is best handled by allowing an indefinite time for the job, particularly new jobs that require unfamiliar techniques; by doubling the allotted time when circumstances force time planning; and by scaling down the scope of what you want to do. Overall goals must be scaled down in importance and immediate goals must be scaled up. This requires value flexibility, and the value shift is usually accompanied by some loss of gumption, but it's a sacrifice that must be made. It's nothing like the loss of gumption that will occur if a Big Mistake caused by impatience occurs.

My favorite scaling-down exercise is cleaning up nuts and bolts and studs and tapped holes. I've got a phobia about crossed or jimmied or rust-jammed or dirt-jammed threads that cause nuts to turn slow or hard; and when I find one, I take its dimensions with a thread gauge and calipers, get out the taps and dies, recut the threads on it, then examine it and oil it and I have a whole new perspective on patience. Another one is cleaning up tools that have been used and not put away and are cluttering up the place. This is a good one because one of the first warning signs of impatience is frustration at not being able to lay your hand on the tool you need right away. If you just stop and put tools away neatly you will both find the tool and also scale down your impatience without wasting time or endangering the work.

***

Well, that about does it for value traps. There's a whole lot more of them, of course. I've really only just touched on the subject to show what's there. Almost any mechanic could fill you in for hours on value traps he's discovered that I don't know anything about. You're bound to discover plenty of them for yourself on almost every job. Perhaps the best single thing to learn is to recognize a value trap when you're in it and work on that before you continue on the machine.

***

I want to talk now about truth traps and muscle traps and then stop this Chautauqua for today.

Truth traps are concerned with data that are apprehended and are within [your mind]. For the most part these data are properly handled by conventional dualistic logic and the scientific method [. . .]. But there's one trap that isn't. . . the truth trap of yes-no logic.

Yes and no—this or that—one or zero. On the basis of this elementary two-term discrimination, all human knowledge is built up. The demonstration of this is the computer memory which stores all its knowledge in the form of binary information. It contains ones and zeros, that's all.

Because we're unaccustomed to it, we don't usually see that there's a third possible logical term equal to yes and no which is capable of expanding our understanding in an unrecognized direction. We don't even have a term for it, so I'll have to use the Japanese mu [□].

Mu means "no thing." [. . .] Mu simply says, "No class; not one, not zero, not yes, not no." It states that the context of the question is such that a yes or no answer is in error and should not be given. "Unask the question" is what it says.

Mu becomes appropriate when the context of the question becomes too small for the truth of the answer. When the Zen monk Joshu was asked whether a dog had a Buddha nature he said "Mu," meaning that if he answered either way he was answering incorrectly. The Buddha nature cannot be captured by yes or no questions.

That mu exists in the natural world investigated by science is evident. It's just that, as usual, we're trained not to see it by our heritage. For example, it's stated over and over again that computer circuits exhibit only two states, a voltage for "one" and a voltage for "zero." That's silly!

Any computer-electronics technician knows otherwise. Try to find a voltage representing one or zero when the power is off! The circuits are in a mu state. They aren't at one, they aren't at zero, they're in an indeterminate state that has no meaning in terms of ones or zeros. Readings of the voltmeter will show, in many cases, "floating ground" characteristics, in which the technician isn't reading characteristics of the computer circuits at all but characteristics of the voltmeter itself. What's happened is that the power-off condition is part of a context larger than the context in which the one zero states are considered universal. The question of one or zero has been "unasked." And there are plenty of other computer conditions besides a power-off condition in which mu answers are found because of larger contexts than the one-zero universality.

The dualistic mind tends to think of mu occurrences in nature as a kind of contextual cheating, or irrelevance, but mu is found throughout all scientific investigation, and nature doesn't cheat, and nature's answers are never irrelevant. It's a great mistake, a kind of dishonesty, to sweep nature's mu answers under the carpet. Recognition and valuation of these answers would do a lot to bring logical theory closer to experimental practice. Every laboratory scientist knows that very often his experimental results provide mu answers to the

yes-no questions the experiments were designed for. In these cases he considers the experiment poorly designed, chides himself for stupidity and at best considers the "wasted" experiment which has provided the mu answer to be a kind of wheel-spinning which might help prevent mistakes in the design of future yes-no experiments.

This low evaluation of the experiment which provided the mu answer isn't justified. The mu answer is an important one. It's told the scientist that the context of his question is too small for nature's answer and that he must enlarge the context of the question. That is a very important answer! His understanding of nature is tremendously improved by it, which was the purpose of the experiment in the first place. A very strong case can be made for the statement that science grows by its mu answers more than by its yes or no answer. Yes or no confirms or denies a hypothesis. Mu says the answer is beyond the hypothesis. Mu is the "phenomenon" that inspires scientific enquiry in the first place! There's nothing mysterious or esoteric about it. It's just that our culture has warped us to make a low value judgment of it.

In motorcycle maintenance the mu answer given by the machine to many of the diagnostic questions put to it is a major cause of gumption loss. It shouldn't be! When your answer to a test is indeterminate it means one of two things: that your test procedures aren't doing what you think they are or that your understanding of the context of the question needs to be enlarged. Check your tests and restudy the question. Don't throw away those mu answers! They're every bit as vital as the yes or no answers. They're more vital. They're the ones you grow on!

***

The mu expansion is the only thing I want to say about truth traps at this time. Time to switch to the psychomotor traps. This is the domain of understanding which is most directly related to what happens to the machine.

Here by far the most frustrating gumption trap is inadequate tools. Nothing's quite so demoralizing as a tool hang-up. Buy good tools as you can afford them and you'll never regret it. If you want to save money don't overlook the newspaper want ads. Good tools, as a rule, don't wear out, and good secondhand tools are much better than inferior new ones. Study the tool catalogs. You can learn a lot from them.

Apart from bad tools, bad surroundings are a major gumption trap. Pay attention to adequate lighting. It's amazing the number of mistakes a little light can prevent.

Some physical discomfort is unpreventable, but a lot of it, such as that which occurs in surroundings that are too hot or too cold, can throw your evaluations way off if you aren't careful. If you're too cold, for example, you'll hurry and probably make mistakes. If you're too hot your anger threshold gets much lower. Avoid out-of-position work when possible. A small stool on either side of the cycle will increase your patience greatly and you'll be much less likely to damage the assemblies you're working on.

There's one psychomotor gumption trap, muscular insensitivity, which accounts for some real damage. It results in part from lack of kinesthesia, a failure to realize that although the externals of a cycle are rugged, inside the engine are delicate precision parts which can be easily damaged by muscular insensitivity. There's what's called "mechanic's feel," which is very obvious to those who know what it is, but hard to describe to those who don't; and when you see someone working on a machine who doesn't have it, you tend to suffer with the machine.

The mechanic's feel comes from a deep inner kinesthetic feeling for the elasticity of materials. Some materials, like ceramics, have very little, so that when you thread a porcelain fitting you're very careful not to apply

great pressures. Other materials, like steel, have tremendous elasticity, more than rubber, but in a range in which, unless you're working with large mechanical forces, the elasticity isn't apparent.

With nuts and bolts you're in the range of large mechanical forces and you should understand that within these ranges metals are elastic. When you take up a nut there's a point called "finger-tight" where there's contact but no takeup of elasticity. Then there's "snug," in which the easy surface elasticity is taken up. Then there's a range called "tight," in which all the elasticity is taken up. The force required to reach these three points is different for each size of nut and bolt, and different for lubricated bolts and for locknuts. The forces are different for steel and cast iron and brass and aluminum and plastics and ceramics. But a person with mechanic's feel knows when something's tight and stops. A person without it goes right on past and strips the threads or breaks the assembly.

A "mechanic's feel" implies not only an understanding for the elasticity of metal but for its softness. The insides of a motorcycle contain surfaces that are precise in some cases to as little as one ten-thousandth of an inch. If you drop them or get dirt on them or scratch them or bang them with a hammer they'll lose that precision. It's important to understand that the metal behind the surfaces can normally take great shock and stress but that the surfaces themselves cannot. When handling precision parts that are stuck or difficult to manipulate, a person with mechanic's feel will avoid damaging the surfaces and work with his tools on the nonprecision surfaces of the same part whenever possible. If he must work on the surfaces themselves, he'll always use softer surfaces to work them with. Brass hammers, plastic hammers, wood hammers, rubber hammers and lead hammers are all available for this work. Use them. Vise jaws can be fitted with plastic and copper and lead faces. Use these too. Handle precision parts gently. You'll never be sorry. If you have a tendency to bang things around, take more time and try to develop a little more respect for the accomplishment that a precision part represents.

***

Maybe it's just the usual late afternoon letdown, but after all I've said about all these things today I just have a feeling that I've somehow talked around the point. Some could ask, "Well, if I get around all those gumption traps, then will I have the thing licked?"

The answer, of course, is no, you still haven't got anything licked. You've got to live right too. It's the way you live that predisposes you to avoid the traps and see the right facts. You want to know how to paint a perfect painting? It's easy. Make yourself perfect and then just paint naturally. That's the way all the experts do it. The making of a painting or the fixing of a motorcycle isn't separate from the rest of your existence. If you're a sloppy thinker the six days of the week you aren't working on your machine, what trap avoidances, what gimmicks, can make you all of a sudden sharp on the seventh? It all goes together.

But if you're a sloppy thinker six days a week and you really try to be sharp on the seventh, then maybe the next six days aren't going to be quite as sloppy as the preceding six. What I'm trying to come up with on these gumption traps I guess, is shortcuts to living right.

The real cycle you're working on is a cycle called yourself. The machine that appears to be "out there" and the person that appears to be "in here" are not two separate things. They grow toward quality or fall away from quality together.

Parsing

We've discussed parsing lightly until this point. We will now dig down into the algorithmic details of parsing.

Before we start, you should know that there is a wealth of literature on parsing. For practical reasons, it was one of the earliest problems attacked by computer scientists. As a result, exploring this topic on your own can be a little daunting, as a typical description of parsing goes deep into the weeds about grammar classes, computational complexity, and so on. Compounding this, many computer scientists like to say offhandedly that parsing "is a solved problem," which is only true in the shallowest sense. Even with nice formal models from theoretical computer science, building a real-world parser remains something of an art.

Instead, we will look at parsing from a functional standpoint. A *parser* is a program that reads in a string as input and, if the input is a valid sentence in a grammar, (1) it emits a result, otherwise it (2) fails. This very simple definition allows us to construct a parser in a simple, recursive manner, using little building blocks. We call these building blocks *parser combinators*.

Why do we need parsers?

If you've never built a parser before, its role may not be obvious, so I will state it here clearly. In computer science, we use parsers to transform serial data (e.g., a string) into structured data (e.g., a tree). When building a programming language, the first thing we need to do is to convert a string (a program) into our preferred representation of a computer program, which is a kind of tree. We call that tree an *abstract syntax tree*, or AST.

An AST is a tree where the interior nodes are operations and the leaf nodes store data. Why do we want this representation? Because, in this form, evaluating a program boils down to a traversal of the tree. For example, in the form of program evaluation we call *interpretation*, we determine the "output" of a program by essentially performing a depth-first, post-order traversal of the tree, combining data from the leaves with op-

erations in the nodes. The output is the final value computed when we are done traversing the root node. In the form of program evaluation we call *compilation*, we also traverse the AST, but instead, we emit machine instructions as we go, converting each step of the interpreter into a sequence of instructions for a machine.

Parsers are used in many more places, from data storage to network protocols. You will probably encounter a few in your professional life. It suffices to say that we need them in the design of programming languages because they form the basis for building user interfaces for *humans*.

Parser Combinators

Before we dig into the technical details of how parser combinators work, let me try to develop an intuition as to what they do. There are two essential ideas.

The first essential idea regarding parser combinators is to build "big" parsing functions out of "little" parsing functions. And when I say *function*, I mean the simplest kind of function you can have: a *combinator*. A *combinator* is a function of *only bound variables*. In other words, this is a combinator:

```
let add a b = a + b
```

but this is *not* a combinator:

```
let add a = a + b
```

because `b` is a free variable. Where does `b` come from? It comes from the environment somewhere, and its precise meaning depends on the *scope rules* for your language. Therefore, a combinator is a function that can be understood without needing any context. It's a simple function without any tricks up its sleeve. A *parser combintor* is therefore a simple function that does parsing.

But how do we make "big" parsers out of "little" parsers? The second essential idea is that some parser combinators function as "glue." We call such "glue" functions *combining forms*.

Armed with "little" parsers and "glue," we can make "big" parsers of great sophistication that don't seem complex. Parser combinators are exactly the kind of big, ugly problem that becomes easy (or at least manageable) when you employ a functional programming approach.

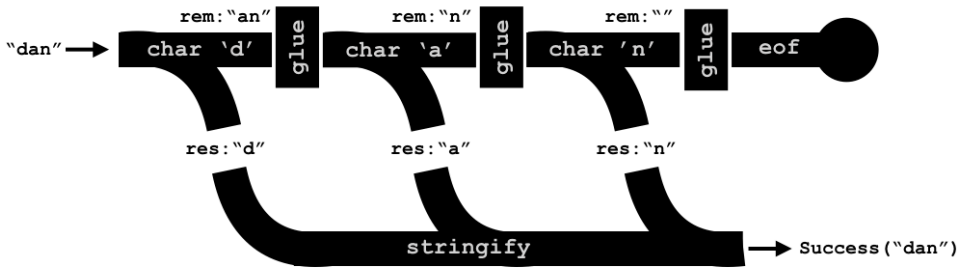
Metaphor

When I work with parser combinators, I like to keep a simple metaphor in my head: plumbing. The function of a pipe is to carry liquid from one point to another. As you're installing the plumbing in your house, you're only tangentially thinking about water (or sewage); you're thinking about how the shapes of the pipes fit together. Sometimes you join multiple pipes. Sometimes you split them. When you put the right stuff in the pipes, they do their jobs, moving liquid from one place to another. When you put stuff in pipes that you shouldn't, they get clogged and back up (Figure 20).

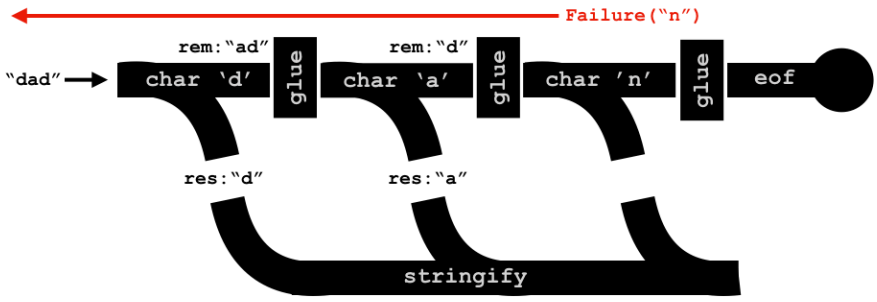
Figure 20: As a toddler, I once flushed a Kewpie doll down the toilet. When my father asked me where my doll was, I pointed at the toilet and said "In there." I have no memory of this incident. My father, who had to disassemble the house's plumbing to find the doll, remembers it well and loves to remind me of the fact.

A combinator is like a pipe. It takes an input `string`, the `string` we are parsing. Depending on what the pipe does, it usually outputs a `string` of some form; that `string` represents the remainder of the input. But combinators may also connect to other combinators; the output of one combinator is fed into the input of the other.

When all goes well, and you flush the appropriate stuff down your parser pipes, you are able to parse input successfully. The parser shown here is designed to parse `"dan"`, and when given the input `"dan"`, it works just fine.



But when you flush the wrong stuff down the parser pipes, it backs up, and you get a failure.



Formal Definitions

Let's get a little more formal in our definition of parser combinators.

I said that "a parser is a program that reads in a string as input and, if the input is a valid sentence in a grammar, (1) it emits a result, otherwise it (2) fails." Let's start with the easy part, shall we? What is "input"?

Input

Here's a simple working definition.

```
type Input = string
```

We will revisit this definition later, but it's something to build on.

Success and Failure

What does it mean for a parser to "succeed" or "fail"?

You might be tempted to say that this means that a parser simply returns a `bool`, and if you were a theoretician studying grammars, that might be sufficient. As a practical matter, we usually expect parsers to return structured data, so we need something a little more nuanced. How about the following ML data structure?

```
type Outcome<'a> =
| Success of result: 'a
| Failure
```

We use `'a` because we might want to return any kind of data.

That's pretty close to what we want, but it's not perfect. The reason is that we want to be able to *combine* little parsers into big parsers. So one way to attack the problem of parsing is to think up a small set of primitive parsers that we can glue together that make more complicated parsers. Each parser then, takes a little nibble at the input and hands the *rest of the input*, the `remainder`, off to the next parser. So let's expand our definition:

```
type Outcome<'a> =
| Success of result: 'a * remainder: Input
| Failure
```

As a practical matter, we also add a small amount of extra debugging information to `Failure`: the position in the string that the failure occurred, and which parser failed.

```
type Outcome<'a> =
| Success of result: 'a * remainder: Input
| Failure of fail_pos: int * rule: string
```

That's good enough for now.

Parser

Now we can construct an elegant definition of a parser.

```
type Parser<'a> = Input -> Outcome<'a>
```

This definition says is that a parser is a function from input to an outcome, either success or failure. On success, we communicate back a result and the remaining portion of the input.

Primitive parsers

You may be surprised to hear that this is enough to start building primitive parsers. The two most primitive are parsers that either succeed no matter what or fail no matter what. We call them `presult` and `pzero`, respectively.

```
let presult(a: 'a)(i: Input) : Outcome<'a> = Success(a,i)

let pzero(i: Input) : Outcome<'a> = Failure(0, "pzero")
```

`presult` takes a return value (`'a`) and an input and returns success. `pzero` just returns failure.

Now, because both of these functions are written using *curried* arguments, they have an interesting and very useful property. If you call them without their last argument, the input, they return a `Parser<'a>`. I clearly remember the first time I learned this fact because I was very confused, and yet, the person teaching me was very excited about this idea. Maybe you're smarter than I am and you're already excited.

In case you're not excited, here's the reason this is a useful property. It's useful because this property enables us to *glue* parsers together. As long as we supply all required arguments to a parser constructor function, except the last argument, it will construct a parser function. When we give that constructed parser object the last argument, some `Input`, it will parse the input. We need to separate these two tasks because first we want to *build* a big parser function, and then later, we want to *call* it.

Let's pop these definitions in `dotnet fsi` and play with them a bit just to be sure that we're on the same page.

```
> let presult(a: 'a)(i: Input) : Outcome<'a> = Success(a,i);;
val presult : a:'a -> i:Input -> Outcome<'a>
```

```
> let pzero(i: Input) : Outcome<'a> = Failure(0, "pzero");;
val pzero : i:Input -> Outcome<'a>

> presult;;
val it : ('a -> Input -> Outcome<'a>)

> presult "hi";;
val it : (Input -> Outcome<string>) = <fun:it@10-1>

> let p : Parser<string> = presult "hi";;
val p : Parser<string>

> pzero;;
val it : (Input -> Outcome<'a>)

> let p : Parser<'a> = pzero;;
val p : Parser<'a>
```

There's no magic here. Partially applying `presult` to `"hi"` returns a parser. `pzero` already is a parser.

OK, one more primitive parser. This is where the magic begins.

```
let pitem(i: Input) : Outcome<char> =
    if i = "" then
        Failure(0, "pitem")
    else
        Success (i.[0], i.[1..])
```

The `pitem` parser attempts to read in *one character*. Notice that the type of the `Outcome` is `char`. If it can read one character, then it returns that one character as the `result` (`i.[0]`) part of the `Success` value, putting the rest of the string (`i.[1..]`) in the `remainder` part. Otherwise, it fails.

Combining forms

Ok, we have three primitive parsers now. How do we "glue" them together? All combining forms are based on one idea, called "bind".

```
let pbind(p: Parser<'a>)(f: 'a -> Parser<'b>)(i: Input) : Outcome<'b> =
    match p i with
    | Success(a,i') -> f a i'
    | Failure(pos,rule) -> Failure(pos,rule)
```

Notice that we are prefixing all of our parser functions with the letter `p`. This is just to make it clear which functions belong to the primitive parsing library. You can name your functions whatever you want.

`pbind` takes a `Parser<'a>`, `p`, and a function `f` from `'a` to a new `Parser<'b>`,

and returns a new `Parser<'b>`. Why do I say that it "returns a new parser" when that's not precisely what the definition says? Well, look carefully in the REPL; it *does* say that. You're just not accustomed to seeing it yet. Here's an example of me partially-applying `pbind`.

```
> let p : Parser<char> = pbind pitem (fun c -> pitem);;
val p : Parser<char>
```

The key bit is that I left off the `Input`, `i`. Remember how I said that all parser combinators take `Input` as their last argument, and, if you leave it off, they're parsers? This is what I meant. That returned parser does the following:

1. Attempt to parse `Input i` with `p`.
2. On success, run `f` on the result of the successful parse, yielding a new parser, `p2`.
3. Run `p2` on the remainder of the first parse.
4. If `p2` is successful, return the outcome of the second parse, otherwise fail.

If you're like me, you might be thinking "OK, I can see *that* you can glue parsers together, but how is this useful?" Great question. I think the most obvious answer is: don't we want to be able to parse more than one character? So let's see how we can achieve that using what we know.

Parsing in sequence

Let's construct a new combining form called `pseq`. We'll use this to parse *two* characters.

```
let pseq(p1: Parser<'a>)(p2: Parser<'b>)(f: 'a*'b -> 'c) : Parser<'c> =
    pbind p1 (fun a ->
        pbind p2 (fun b ->
            presult (f (a,b))
        )
    )
```

The `pseq` parser is a "combining" function. It takes two parsers, `p1` and `p2`, and runs them, one after the other, returning the result as a pair of elements. Note that we use `pbind` in this definition, and `presult` finally makes an appearance. We first bind `p1` to a function that takes `p1`'s result as input and then binds `p2` to a function that takes `p2`'s result, which is then handed to the `presult` parser, which takes both results and runs function `f` on them. This may seem sort of abstract to you, but if you work it out on paper, it's not so bad. It captures everything we need to say in order to parse two things in sequence.

Now, notice, that this definition does not take an `Input`. Or does it? Actually, it does! But we were able to leave it off. Why? Because `p1`, `p2`, `pbind`, and `presult` also all take an `i`, and since we only ever partially apply those functions, we are always "passing the buck" to the next function in the chain. Since the entire body of the `pseq` function punts on handling input, *even though its components must handle input*, it means that `pseq` itself must handle input. In fact, that's what the return type says: `Parser<'c>`.

Part of the reason why this sort of melted my brain the first time I saw it is that I kept thinking: but why don't you just handle the input? Wouldn't the definition be clearer? The PL theorist who taught me combinators thought the answer to this question was obvious ("NO!") so he didn't spend much time on it. As a result, it took me a little time to appreciate how much simpler partial application can make a program. The gain in simplicity is especially profound when it is the case that all of your functions pass around the same parameter (in our case, `Input`).

This fact sheds light on the popularity of a another model for programming: object oriented programming. In that model, you pass around all kinds of parameters implicitly—you just stick that data inside an object and pass the object around instead. So it solves the same problem, but using a different mechanism. But unlike functional programming, where you are *forced* to think about all of the data you need all of the time, we sometimes forget about the data we stick in objects. In particular, we forget that we need to update it, leading to bugs. Functional code forces us think about that data. The tradeoff is that we never have stale values floating around in objects. Personally, functional programming forced me to stop being lazy with objects, and the benefits—fewer bugs—became immediately clear to me.

OK, enough chit-chat. Let's use `pseq` to actually parse two characters.

```
let ptwo : Parser<string> =
  pseq pitem pitem (fun (c1,c2) -> c1.ToString() + c2.ToString())
```

So we just constructed a parser that parses two characters, and then takes the pair of characters, converts them to strings (remember a `char` is not a `string` and in F# we have to explicitly convert them), and concatenates them, returning a string. Let's try it.

```
> ptwo "hello world";;
val it : Outcome<string> = Success ("he","llo world")
```

Cool, huh? Watch what happens when the input does not have two characters left.

```
> ptwo "h";;
val it: Outcome<string> = Failure (0, "pitem")
```

Because we built up our parsers simply and from first principles, the combined parser does the right thing.

End of file

There's one more essential parser that we need to specify, and it requires that we change our definition of input a little. It is often necessary, for example, in your "top level" parser, to be able to state "only succeed if you've parsed all of the input." In other words, we need to check that we've reached the end of the input string.

Unfortunately, nowhere in our definition of `Input` do we maintain a notion of "the end". We probably should. Let's modify our definition of `Input` just a little. [64]

[64] Note that the combinator library that comes with your starter code, `Combinator.fs`, has an even fancier definition for `Input` for efficiency. The basic idea is the same, but note that the definition is slightly different. Have a look if you are curious. I try to make our libraries easy to read.

```
type Input = string * bool
```

Now `Input` is a pair of `string` and `bool`. The `bool` represents whether we've found the end. This is useful because often a parser definition, which is composed of many little parsers, slices and dices the input into many pieces, and those pieces themselves are sliced and diced. Without tracking the "real end" of the string, we might be tempted to think that "the end" was merely the end of the input string. But if we've cut the string in half somewhere, that most definitely will not be the case. There's only *one end*!

This affects the definition of `pitem` above, but not by much. In fact, it doesn't change at all how we use them, just whether we can actually test for EOF. Here's a definition of an EOF parser:

```
let peof(i: Input) : Outcome<bool> =
    match pitem i with
    | Failure(pos, rule) ->
        if snd i = true then
            Success(true, i)
        else
            Failure(pos, rule)
    | Success(,) -> Failure((position i), "peof")
```

First, `peof` tries to get a character, and if that fails *and we're at the real end of the string*, succeed. Otherwise, fail.

Here's a little function that we can call on our input `string` to turn it into an `Input` so that the user does not have to think about how to set up an `Input` the first time.

```
let prepare(input: string) : Input = input, true
```

A zillion more little parsers

Hopefully now you have the basic idea. You can make and combine parsers from other parsers. That combined parser can be called using string input, and it returns what you ask. At each step, you provide a function `f` that says exactly how to build the data structure that you return in the end (e.g., "concatenate two characters into a string"). It can be whatever you want.

In this section, I am going to tell you about a collection of other useful parsers. I am not going to belabor their definitions, since you can just look through the code and understand them if you need to. In many cases, you will not need to. This, of course, is also not an exhaustive list of parsers. Like I said, they build pretty much anything you want. The set below is just a subset convenient to use for this course.

Parser	Type	Description	Example
`presult`	`'a -> Parser<'a>`	Takes a result value `'a` and an input and returns `Success`.	
`pzero`	`Parser<'a>`	Takes an input and returns `Failure`.	
`pitem`	`Parser<char>`	Reads a single character.	
`peof`	`Parser<bool>`	Takes an input and returns true if and only if there is no more input left to consume.	
`pbind`	`p:Parser<'a> -> f:('a -> Parser<'b>) -> Input -> Outcome<'b>`	Form for combining a parser `p` in an arbitrary way with another parser using a function `f`.	
`pseq`	`p1:Parser<'a> -> p2:Parser<'b> -> f:('a * 'b -> 'c) -> Parser<'c>`	Combine two parsers `p1` and `p2` in sequence, and combine their results using a function `f`.	`pseq pitem pitem (fun (a,b) -> (a,b)` parses two characters and returns them as a 2-tuple.
`psat`	`f:(char -> bool) -> Parser<char>`	Read a character, and if it satisfies the predicate `f`, successfully return the character.	`psat (fun c -> c = 'z')` parses the character `z`
`pchar`	`c:char -> Parser<char>`	Read a character, and if it is the same as the given character `c`, successfully return it.	`pchar 'z'` parses the character `z`
`pletter`	`Parser<char>`	Reads a character and returns successfully if the character is alphabetic.	`pletter` parses any alphabetic letter.
`pupper`	`Parser<char>`	Reads a character and returns successfully if the character is uppercase alphabetic.	`pupper` parses any uppercase alphabetic letter.
`pdigit`	`Parser<char>`	Reads a character and returns successfully if the character is numeric.	`pdigit` parses any numeral.
`<|>`	`p1:Parser<'a> -> p2:Parser<'a> -> Parser<'a>`	Ordered choice. Note that this is an *infix* combinator, for readability. First tries the parser `p1`, and if that fails, it backtracks the input and tries parser `p2`. The first parser to succeed returns the result. If both parsers fail, choice fails.	`(pchar 'a') <|> (pchar 'b')` parses either the character `a` or the character `b`.
`|>>`	`p:Parser<'a> -> f:('a -> 'b) -> Parser<'b>`	Function application. Applies the function `f` to the result of `p` if `p` is successful.	`pdigit |>> (fun c -> int (string c))` converts a numeric character into an integer.

Parser	Type	Description	Example
`pfresult`	`p:Parser<'a> -> x:'b -> Parser<'b>`	Run parser `p` and if successful, return result `x`. This basically ignores the output of `p`.	`pfresult (pchar 'a') 'b'` returns the character `b` when it finds `a`.
`pmany0`	`p:Parser<'a> -> Parser<'a list>`	Parse zero or more occurrences of `p` in sequence, stopping when `p` fails. Note that this parser never fails!	`pmany0 pletter` parses a sequence of letters, stopping when no more letters can be found.
`pmany1`	`p:Parser<'a> -> Parser<'a list>`	Parse one or more occurrences of `p` in sequence, stopping when `p` fails. To succeed, `p` must succeed *at least once*.	`pmany1 pletter` parses a sequence of letters of length 1 or more.
`pws0`	`Parser<char list>`	Parses a sequence of zero or more whitespace characters.	`pws0`
`pws1`	`Parser<char list>`	Parses a sequence of one or more whitespace characters.	`pws1`
`pnl`	`Parser<string>`	Parses a newline. Returns a `string` instead of a `char` because newlines are actually two characters on Windows machines.	`pnl`
`pstr`	`s:string -> Parser<string>`	Parses the string literal `s`.	`pstr "helloworld"` parses `helloworld` and only `helloworld`.
`pleft`	`p1:Parser<'a> -> p2:Parser<'b> -> Parser<'a>`	Parses `p1` and `p2` in sequence, returning *only* the result of `p1`. Discards the result of `p2`.	`pleft (pchar 'a') (pchar 'b')` parses `ab` but only returns `a`.
`pright`	`p1:Parser<'a> -> p2:Parser<'b> -> Parser<'b>`	Parses `p1` and `p2` in sequence, returning *only* the result of `p2`. Discards the result of `p1`.	`pright (pchar 'a') (pchar 'b')` parses `ab` but only returns `b`.
`pbetween`	`popen:Parser<'a> -> p:Parser<'c> -> pclose:Parser<'b> -> Parser<'c>`	Parses `p` *in between* parsers `popen` and `pclose`. Discards the resuls of `popen` and `pclose`.	`pbetween (pchar '(') (pmany0 pletter) (pchar ')')` returns `abc` when given `(abc)`.
`<!>`	`p:Parser<'a> -> label:string -> Parser<'a>`	Debug parser. This parser applies `p` and, as a side effect, prints some diagnostic information given a `label`. Very useful for figuring out why a parser succeeds or fails on a given input.	`pletter <!> "letter"`
`stringify`	`cs:char list -> string`	Convert the `char list` called `cs` into a `string`.	`stringify ['h';'e';'l';'l';'o']` returns `hello`

An example: parsing English sentences

Lets's build a small parser for parsing well-formed English sentences. As you will see, it's not hard to find the limitations of this parser. But after we build this together, you should have a good idea about how you could extend it to parse more complex sentences.

First, what is an "English sentence"? Here's some BNF:

```
<sentence>    ::= <upperword> (<ws> <word>)* <period>
<upperword>   ::= <upperletter> (<letter>)*
<word>        ::= (<letter>)+
<upperletter> ::= 'A' | 'B' | ... | 'Y' | 'Z'
<lowerletter> ::= 'a' | 'b' | ... | 'y' | 'z'
<letter>      ::= <upperletter> | <lowerletter>
<ws>          ::= ' ' | 'n' | 't' | "rn"
<period>      ::= '.'
```

Let's further stipulate that the "structure" that I want to return from my parser is a list of the words in the sentence. A `string list` should work nicely. I would also like to know whether the parse succeeded or failed, so let's wrap our `string list` in an `option` type[65]. So our end result will be a function like:

[65] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/options

```
let parse(input: string) : string list option =
    ... whatever ...
```

When building a parser using combinators, you can either start at the top of your grammar and work your way down or you can start at the bottom and work your way up. I'm sort of a bottom-up thinker, so we'll start with the simplest parts; the ones that parse terminals.

Period has a simple grammar rule with only one terminal. Should be easy.

```
let period = pchar '.'
```

This is hopefully self-explanatory.

Whitespace, as it turns out, is built-in. Let's say, for now, that `pws1` is what we want.

OK, arbitrary letters. Again, we already have a parser for this called `pletter` that parses both uppercase and lowercase letters. We also have parsers for uppercase only (`pupper`) and lowercase onle (`plower`).

How about words? Our word production says:

```
<word> ::= (<letter>)+
```

What does that mean? We're using an "extended" form of BNF called EBNF here that lets us write repetition more concisely. + means "at least one". So what this says is "at least one `<letter>`". We can unroll this if we want into regular BNF.

```
<word> ::= <letter> <word>
         |  <letter>
```

So a `<word>` really is a recursive definition that requires at least one `<letter>`. Fortunately for us, we don't have to think too hard about repetition, because there's a parser combinator that means "one or more" called `pmany1`. So a word is:

```
let word = pmany1 pletter
```

Which is great and all, but *almost* what we want. Remember how I said that we wanted a "list of words" back and I said that this translated into a `string list`? Well, what does `pmany1 pletter` actually return?

```
> let word = pmany1 pletter
val word : Parser<char list>
```

It actually returns a list of characters. Although I think we all can agree that a list of characters is pretty much a string, we have to actually convert one into the other to keep F# happy. We do that using the `|>>` combinator.

```
> let word = pmany1 pletter |>> (fun cs -> System.String.Join("", cs))
val word : Parser<string>
```

That's better.

Because we often translate lists of characters into strings, I've provided the `stringify` function that does this for you. So we can rewrite `word` a little more simply.

```
let word = pmany1 pletter |>> stringify
```

Let's test our work up until this point. Remember that we need to call `prepare` on our input `string` before we can give it to our parsers.

```
> word (prepare "foobar");;
val it : Outcome<string> = Success ("foobar",("", true))
```

That looks promising. How does it handle a space in the middle?

```
> word (prepare "foo bar");;
val it : Outcome<string> = Success ("foo",(" bar", true))
```

Notice that it succeeds, but only returns `"foo"`. `"bar"` is left in the remainder. This makes sense because `word` doesn't know anything about spaces.

```
> word (prepare " foo bar");;
val it : Outcome<string> = Failure
```

This also looks good. There are words in the string, but the string starts with a space. Again, `word` doesn't know anything about spaces so it fails.

How about words that start with an uppercase letter?

```
<upperword> ::= <upperletter> (<letter>)*
```

Again, this is EBNF. The * operator means "zero or more". So an uppercase word must be at least one uppercase letter followed by zero or more letters of any case. Unrolled into regular BNF:

```
<upperword> ::= <upperletter> <word>
              |  <upperletter>
```

As before, thinking about this recursively is a useful exercise, but we have a parser that makes our lives easier. `pmany0` parses zero or more occurrences of a parser. How do we parse first an uppercase letter and then zero or more letters of any case? Anytime your parsing logic is of the form "first parse this then parse that" you're talking about parsing *sequences*. The `pseq` parser is for parsing sequences of elements.

```
let upperword = pseq pupper (pmany0 pletter)
```

As before, we want to get back a `string`, but what this gives us back is kind of a mess. `pmany0 pletter` returns a `char list`, `pseq` returns a tuple, and `pupper` returns a char, so what we get is a `char * char list`. Fortunately, `pseq` also expects a function that lets us sort it all out. We want a `string`.

```
> let upperword = pseq pupper (pmany0 pletter) (fun (x,xs) -> stringify (x::xs));;
val upperword : Parser<string>
```

Now it does what we want!

OK, where are we? We can parse uppercase words and other words.

What is the form of a sentence? From our BNF above, it really has two pieces. Let's switch from working bottom-up to working top-down. Hopefully we can meet in the middle somewhere.

```
<sentence> ::= <upperword> (<ws> <word>)* <period>
```

There's a *prefix*, which is the first uppercase word and a middle part, which is spaces and words. Then there's a *suffix*, which is the period. When you have a complicated production rule, thinking in terms of prefixes and suffixes helps a lot. Note that we could have divided this in many different ways. Let's make a top-level `sentence` parser with these parts and then flesh each piece out.

```
let sentence = pleft prefix period
```

`pleft` applies two parsers but only returns the result of the one on the left. So `sentence` just returns the result of `prefix`, which makes sense because we don't actually care about putting a period in our list of words.

`prefix` also has two parts: an uppercase word and then zero or more whitespace-separated words.

```
let prefix = pseq upperword words0 (fun (w,ws) -> w::ws)
```

We are calling "zero or more whitespace-separated words," `words0`. OK, so clearly `upperword` returns a string. And hopefully, we can build `words0` so that it returns a `string list`. If that's what we're getting back, then combining them should be simple: just *cons* the uppercase word to the list of words from `words0`. Fortunately, `pseq` wants a function that asks us how to combine its two pieces, so we tell it to combine using *cons*.

Let's define `words0` now. So we want zero or more words prefixed by whitespace.

```
let words0 = pmany0 (pright pws1 word)
```

`pright` is like `pleft` except that it returns the result from the parser on the right, in this case, a `word`. Without any more work, this already does what we want, see?

```
> let words0 = pmany0 (pright pws1 word);;
val words0 : (Input -> Outcome<string list>)
```

which, of course, is a `Parser<string list>`.

We're almost done! We just have to define a top-level parser now. Out of habit, I always call this top-level parser `grammar`. `grammar` really only does one thing: it calls our parser and makes sure that we've parsed *all* of the input. Remember that many parsers (like `word`) will happily nibble off only a part of the input and leave the rest behind. To ensure that all the input is consumed, we make sure that the only thing left is EOF by concluding with the `peof` parser.

```
let grammar = pleft sentence peof
```

Since we don't really care what `peof` returns—just that it's successful—we use `pleft` with our `sentence` parser.

Here is the complete program along with a little `main` function so that you can try it out using `dotnet run`.

```
open Combinator

let period = (pchar '.')
let word = pfun (pmany1 pletter) (fun cs -> stringify cs)
let upperword = pseq pupper (pmany0 pletter) (fun (x,xs) -> stringify (x::xs))
let words0 = pmany0 (pright pws1 word)
let prefix = pseq upperword words0 (fun (w,ws) -> w::ws)
let sentence = pleft prefix period
let grammar = pleft sentence peof

let parse input : string list option =
    match grammar (prepare input) with
    | Success(ws,_) -> Some ws
    | Failure(_,_) -> None

[<EntryPoint>]
let main argv =
    if argv.Length <> 1 then
        printfn "Usage: dotnet run <sentence>"
        exit 1
    match parse argv.[0] with
    | Some ws -> printfn "Sentence: %A" ws
    | None    -> printfn "Invalid sentence."
    0
```

The parsers I describe above are available in a module called `Combinator.fs` which is available on the course website.

Let's run our program.

```
$ dotnet run "This is a sentence."
Sentence: ["This"; "is"; "a"; "sentence"]
```

Debugging parsers

While you can go around sticking `printfn` statements into your combinator code when things don't go as planned, there's a much better way to debug: the debug parser, `<!>`. Here's a version of the same program but this time decorated with debug parsers.

```
open Combinator

let period = (pchar '.') <!> "period"
let word = pfun (pmany1 pletter) (fun cs -> stringify cs) <!> "word"
let upperword = pseq pupper (pmany0 pletter) (fun (x,xs) -> stringify (x::xs)) <!> "upperword"
let words0 = pmany0 (pright pws1 word) <!> "words0"
let prefix = pseq upperword words0 (fun (w,ws) -> w::ws) <!> "sprefix"
let sentence = pleft prefix period <!> "sentence"
let grammar = pleft sentence peof <!> "grammar"

let parse input : string list option =
    match grammar (debug input) with
    | Success(ws,_) -> Some ws
    | Failure(_,_) -> None

[<EntryPoint>]
let main argv =
    if argv.Length <> 1 then
        printfn "Usage: dotnet run <sentence>"
        exit 1
    match parse argv.[0] with
    | Some ws -> printfn "Sentence: %A" ws
    | None    -> printfn "Invalid sentence."
    0
```

Observe that we changed `prepare input` in our `main` method to `debug input`. The difference is that `debug` sets an internal debugging flag to `true` whereas `prepare` sets it to `false`. The two functions are otherwise the same. When a debug parser, always written like `<!> "rule"`, encounters a debugging flag set to `true`, it prints diagnostic information, including `rule`, to the terminal.

Let's run this program.

```
$ dotnet run "This is a sentence."
[attempting: grammar on "This is a sentence.", next char: 0x54]
[attempting: sentence on "This is a sentence.", next char: 0x54]
[attempting: sprefix on "This is a sentence.", next char: 0x54]
[attempting: upperword on "This is a sentence.", next char: 0x54]
[success: upperword, consumed: "This", remaining: " is a sentence.", next char: 0x20]
[attempting: words0 on "This is a sentence.", next char: 0x54]
[attempting: word on "This is a sentence.", next char: 0x54]
[success: word, consumed: "is", remaining: " a sentence.", next char: 0x20]
[attempting: word on "This is a sentence.", next char: 0x54]
[success: word, consumed: "a", remaining: " sentence.", next char: 0x20]
[attempting: word on "This is a sentence.", next char: 0x54]
[success: word, consumed: "sentence", remaining: ".", next char: 0x2e]
[success: words0, consumed: " is a sentence", remaining: ".", next char: 0x2e]
[success: sprefix, consumed: "This is a sentence", remaining: ".", next char: 0x2e]
[attempting: period on "This is a sentence.", next char: 0x54]
[success: period, consumed: ".", remaining: "", next char: EOF]
[success: sentence, consumed: "This is a sentence.", remaining: "", next char: EOF]
[success: grammar, consumed: "This is a sentence.", remaining: "", next char: EOF]
Sentence: ["This"; "is"; "a"; "sentence"]
```

With this latter version, when things go wrong, we can see why.

```
$ dotnet run "This is a sentence"
[attempting: grammar on "This is a sentence", next char: 0x54]
[attempting: sentence on "This is a sentence", next char: 0x54]
[attempting: sprefix on "This is a sentence", next char: 0x54]
[attempting: upperword on "This is a sentence", next char: 0x54]
[success: upperword, consumed: "This", remaining: " is a sentence", next char: 0x20]
[attempting: words0 on "This is a sentence", next char: 0x54]
[attempting: word on "This is a sentence", next char: 0x54]
[success: word, consumed: "is", remaining: " a sentence", next char: 0x20]
[attempting: word on "This is a sentence", next char: 0x54]
[success: word, consumed: "a", remaining: " sentence", next char: 0x20]
[attempting: word on "This is a sentence", next char: 0x54]
[success: word, consumed: "sentence", remaining: "", next char: EOF]
[success: words0, consumed: " is a sentence", remaining: "", next char: EOF]
[success: sprefix, consumed: "This is a sentence", remaining: "", next char: EOF]
[attempting: period on "This is a sentence", next char: EOF]
[failure at pos 18 in rule [pchar '.']: period, remaining input: "", next char: EOF]
[failure at pos 18 in rule [pchar '.']: sentence, remaining input: "", next char: EOF]
[failure at pos 18 in rule [pchar '.']: grammar, remaining input: "", next char: EOF]
Invalid sentence.
```

Oops! It looks like we forgot the period.

Performance

One thing you may notice while playing with combinators is that the performance is not always stellar. There are a few reasons.

First, this library is not optimized in any way. It's designed as a teaching tool. If you want a commercial-grade combinator parsing library, you should look elsewhere. FParsec[66] is a good, open-source library that I use a lot.

[66] http://www.quanttec.com/fparsec/

Second, backtracking parsers are expensive, because when they fail, other alternatives are explored. For example, if you peek at the imlementation for the choice combinator, `<|>`, you will see that it does just that. When producing a commercial-grade parser, you will want to invest some time in optimizing your code. These optimizations are largely outside the scope of this class.

Finally, there is a cost to using higher-order functions, although the F# compiler does do a respectable job about optimizing away obvious inefficiencies. Still, *hand-written parsers*, not using parser libraries like combinators, will always be faster, especially when written in unsafe languages like C. Consequently, some of the fastest parsers are written in C. When performance is critical, as is often the case for code that parses network messages, hand-written parsers coded in C are the norm.[67]

[67] Producing high-performance network code in programming languages that guarantee safety properties is an area of active research. A good example is Project Everest, which aims to produce bug-free, high-performance cryptographic libraries for networking. https://project-everest.github.io/

Parsing theory is good, but is hard to apply in practice

Before we conclude, let's revisit my comments about parsing theory. As I stated before, there is a great deal of research on parsing. In fact, some parsers can be generated automatically from grammar specifications. Generated parsers are sometimes blazingly fast because they are designed to emit C code that is the moral equivalent of one big `switch` statement, something called a *table-driven parser*. The chief difficulty with parser generators, however, is that you must know the formal grammar class of your language ahead of time. Is your language "deterministic context-free?" Use an LR parser. Is it "regular"? Use regular expressions.

In practice, it is difficult to know for sure to what class your language belongs until you try to write a parser. And even though we can gener-

ate parsers, this does not mean that it is "no work" to generate the specification for the parser generator. In my experience, I am often deep into a parser implementation when I make the discovery that a given parser generator was a poor choice. One is then faced with the difficult decision of having to change the syntax or throw out the entire implementation and start over again with a different parser generator. People who do this work more often than me may have a better intuition for which parser generators are good for which jobs. Nevertheless, when I discovered combinators, I stopped using parser generators entirely and never looked back.

The fact that the "best" parsing algorithm can be chosen based on the grammar class of your language is why theoretical computer scientists call parsing a "solved problem." Like many things in computer science, theory is a good guide, but ultimately, the proof is in the pudding.[68]

[68] The original saying is "the proof of the pudding is in the eating." It means that you need to try something yourself to know whether it is good. Have you ever tried coding something up for fun? You should. The experience is worth having, even if you don't accomplish what you set out to do.

Evaluation

We now come to the heart of what a programming language *does*. In the previous chapter, we discussed how to convert a string into a form that the computer can use, which we call the AST. Here, we discuss how to use the AST to compute something.

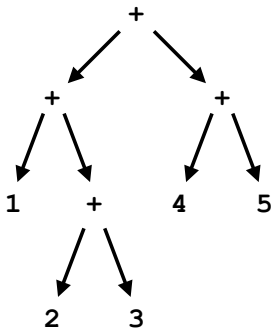
High-level discussion of evaluation can become abstract and complex without an example, so let's define a small language to help ground things. An easy place to start is a calculator program. Let's keep things simple and define a calculator that can only add. Let's also stipulate that `+` takes exactly two operands. Finally, because infix operators[69] complicates parsing, let's specify that operators should appear in prefix form. In other words, the following is a valid program in our language.

```
+ 2 3
```

The above means the same thing as `2 + 3` but is easier to parse. However, we should also be able to compute compound expressions like

```
+ + 1 + 2 3 + 4 5
```

For the above expression, our parser should produce the following AST:



Our formal grammar is relatively uncomplicated, and might look something like the following.

[69] An infix operator is an operator that appears, syntactically, between two arguments in an expression. `2 + 3` is an infix expression. The problem arises when you have expressions like `2 / 3 / 4`, which are ambiguous without extra parsing rules. Which `/` operator is evaluated first? If you parse this expression incorrectly, your evaluator will produce the wrong result. Unlike infix operators, languages with operators in prefix form do not have ambiguous parses.

```
<expr>  ::= + <expr> <expr>
          | <num>
<num>   ::= <digit>+
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

Be aware that the + and $^+$ symbols have different meanings. The + symbol stands for a literal plus sign character in our grammar. By contrast, $^+$ means *one or more*, and it allows us to define syntax for multi-digit numbers.

PlusLang AST

To start, we need to define the AST that we want. Since expressions in this language have two cases, either numbers or plus-expressions, we should define an F# type that captures that fact. Here's one way you might do it.

```
type Expr =
| Num of int
| Plus of Expr * Expr
```

PlusLang Parser

Next, we need a parser that turns valid PlusLang strings into instances of the above `Expr`. Since you already know how parser combinators work, I provide a complete parser here without walking you through every line.

```
let expr,exprImpl = recparser()
let pad p = pbetween pws0 p pws0
let num = pmany1 pdigit |>> (fun ds -> stringify ds |> int |> Num)
let numws0 = pad num
let plusws0 = pad (pchar '+')
let plusExpr = pseq (pright plusws0 expr) expr Plus
exprImpl := plusExpr <|> numws0
let grammar = pleft expr peof
```

Two features of this program require more explanation. The simplest of the two, `pad`, is a combinator I define above to allow for optional whitespace around something. For example, `pad (pchar '+')` means that I could accept the strings `"+"` or `" +"` or `" + "` or any other with arbitrary amounts of space characters. The second, more complicated thing, is `recparser`.

We need to use `recparser` because our grammar is recursive. You can see that in our formal grammar right here:

```
<expr>  ::= + <expr> <expr>
```

Due to limitations in the way F# defines functions, a recursive parser definition using combinators is a little tricky. A naïve definition produces an F# compilation error. For example, we cannot define `expr` this way, even though such a definition does not seem obviously wrong:

```
let expr = plusExpr <|> numws0
```

The reason is that `plusExpr` contains `expr` within its own definition.

```
let plusExpr = pseq (pright pluswsO expr) expr Plus
```

Most functional languages like F# are strict about the order in which things are defined. If we want to use `plusExpr` in the definition for `expr` then `plusExpr` must be defined before `expr`. But if `plusExpr` contains `expr` in its own definition, then `expr` must be defined before `plusExpr`. This contradiction means that we cannot define recursive definitions quite as simply as we might hope.

Instead, we use `recparser`. The `recparser` combinator lets us separate the *declaration* of a parser from its *definition*. You've seen the separation of declaration and definition before in another programming language, namely Java.[70] For example, suppose we have the following Java interface.

[70] Separating interface from implementation is a fundamental technique for abstraction in computer science, and it's the reason why features like pointers or references exist in programming languages.

```
interface Honkable {
  public String honk();
}
```

This `interface` *declares* that anything `Honkable` must have a `honk` method, but it does not provide code to make anything `honk`. With the `Honkable` interface in hand, we can *define* something `Honkable` later.

```
class Car implements Honkable {
  public String honk() {
    return "beep!";
  }
}
```

Our use of `recparser` is similar. While F#'s type inference (thankfully) hides some detail from us, the following expression declares a parser called `expr`.

```
let expr,exprImpl = recparser()
```

From this point forward, we can *use the name* `expr` whereever we want. However, `expr`'s implementation is not yet defined. To define it, we use the `:=` operator.[71]

```
exprImpl := plusExpr <|> numws0
```

[71] The `:=` operator stands for *mutable assignment*. I won't go into detail here, but it suffices to say that `recparser` uses mutable variables to get around `expr`'s recursive definition problem. There is a "purely functional" alternative, but it makes combinators harder to use, so in the interest of enhancing the quality of your life I chose this slightly icky approach instead.

PlusLang Evaluator

Now that we have an AST defined and a parser that can construct ASTs from valid strings, we need to say what an expression in PlusLang means. An *evaluator* is a function that takes an AST and *does something*. What does depends on what you want the parts of the language to mean.

Conventionally, we name the evaluator for a programming language `eval`.[72] Our language should add numbers together.

[72] Some programming languages actually expose their `eval` function in the language itself. Javascript is one such language. Being able to call such a function is "meta" in the true sense of the word, which makes it very cool but also dangerous.

Here is where F#'s `types` are incredibly useful. Our `Expr` type has two cases. An expression could either be a number or the addition of two expressions. Therefore, our `eval` function must also have two cases. The `eval` function must also be recursive, because `Expr` itself is recursive. We will use pattern matching to distinguish between the two cases. Our solution will look something like this:

```
let rec eval ast : Expr =
    match ast with
    | Num n -> ...
    | Plus (left, right) -> ...
```

Observe that `eval` takes an `Expr` and returns an `Expr`. You might be wondering how that can be useful. In general, an evaluator tries to simplify an expression. The simplest kind of expression in our language right now is a number. We can think of a number as a primitive in our language. This explains why we take an `Expr` and return an `Expr`. We might start with a complex expression, but at the end, we will have a simple expression, ideally just a number. Since a number is an `Expr`, that's what we return. Therefore, for each case, we must ask "how can I get a number out of this?"

For the first case, when we see a number `n`, there's not really much to do. It's already a number. So we just return it.

```
| Num n -> Num n
```

What about the second case, when we see a plus-expression? Could we just do the following?

```
| Plus (left, right) -> left + right
```

Sadly, the answer is no. If you try it, you will discover that `left` and `right` are `Exprs` and that F# will produce a compilation error. Since we ourselves defined `Expr`, and because we did not define the `+` operation for `Expr`, F# does not know what the above expression means. Furthermore, since `left` and `right` are themselves expressions, they might not be simple numbers. After all, the `left` operand might be the AST subtree representing `+ 2 3`, which is also not something a computer knows how to add. We have to evaluate expressions recursively.

```
| Plus (left, right) ->
    let n1 = eval left
    let n2 = eval right
    ...
```

Now what? Well, if those recursive calls to `eval` did their jobs correctly, then we know that we got numbers back. Still, we can't just add them, because F# itself will not be convinced that they are `Nums`. After all, `eval` returns an `Expr`. To work around this, we simply pattern match. And if, for some reason we made a mistake in our implementation and we *don't* get numbers back, we can catch the error.

```
| Num n -> Num n
| Plus (left, right) ->
    let r1 = eval left
    let r2 = eval right
    match r1, r2 with
    | Num n1, Num n2 -> Num(n1 + n2)
    | _ -> failwith "Can only add numbers."
```

I claim that this definition is good, but why? Let's think back about what makes recursive programs work. First, they must have a base case. Second, they must have a recursive case that *reduces the problem toward the base case*. Do we have those two things?

If `ast` is just a `Num`, then that's our base case, and we just return the number itself. Check.

If `ast` is more complicated, then we break it into two pieces, `left` and `right`. Both `left` and `right` are smaller than `ast` since they are the left and right parts of the addition expression, respecively. Consequently, when we make our recursive calls, we reduce the problem toward the base case. Check.

Therefore, the above solution should work fine.

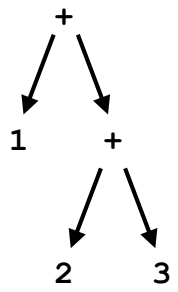


Figure 21: The AST for the subexpression `+ 1 + 2 3`.

What's Going On?

You might feel like the above explanation about evaluation is too brief. What's really going on when we evaluate code?

In short, an evaluator performs a traversal of an AST. When an AST is in its conventional form, where interior nodes are operators and where leaves are values, then evaluation can usually be performed by doing a pre-order traversal.[73] Let's examine how we might evaluate the subexpression `+ 1 + 2 3`. It has the tree shown in Figure 21.

[73] Don't remember what a pre-order traversal is? The name of the traversal tells you what order it visits data relative to a subtree's root. A *pre-order* traversal visits children before the root. A *post-order* traversal visits children after the root.

Like all traversals, we start at the root. What do we know at the root of this subtree? What we know is captured in the matching pattern `| Plus (left, right)`. We know that the node must add two things, and we know that those two things are called `left` and `right`. The type of `left` and of `right` is `Expr`. We know that because that's what the `Plus` case of the `Expr` type tells us in `| Plus of Expr * Expr`. Unfortunately, that's all we know. Importantly, we do not know the numeric values of `left` and `right`. If `left` and `right` are themselves complex expressions (i.e., they are plus-expressions), then we need to do some work to determine their values.

This uncertainty is why we tend to evaluate ASTs recursively. We continue to recurse until we can't anymore: at that point, we have reached a leaf. Recall that a leaf is a value. When we reach a leaf, we *return* its value.

After visiting a leaf, the recursion unwinds back up the tree, carrying returned values with it. Once a plus-expression's `left` and `right` children have been evaluated, the two are added and returned. The entire process is shown in Figure 22.

Complete PlusLang Code

A complete implementation for PlusLang is shown in Listing 1. I do not include the `fsproj` definition or the `Combinator.fs` library. As the usage string suggests, you can run this program on the command line like so

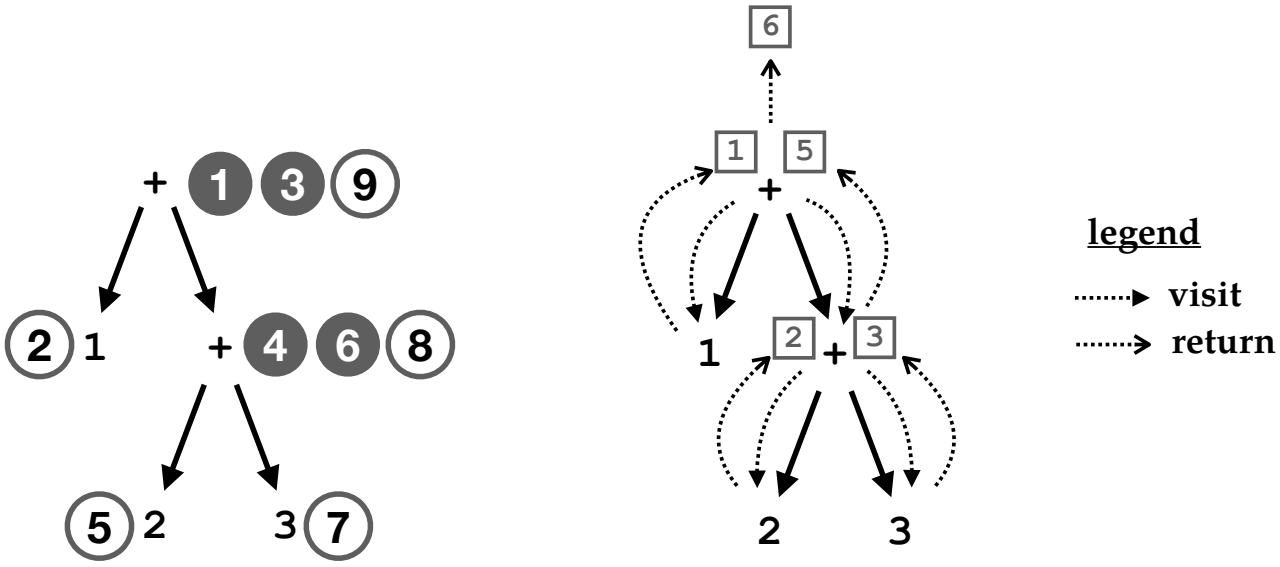


Figure 22: The order that the `eval` function visits nodes in the expression's AST. On the left, outlined circles denote *when* data is returned. Observe that data is returned in a pre-order. On the right, boxed numbers denote *what* data is returned.

```
$ dotnet run "+ + 1 + 2 3 + 4 5"
```

which prints the following

```
Expression: Plus (Plus (Num 1, Plus (Num 2, Num 3)), Plus (Num 4, Num 5))
Result: Num 15
```

Listing 1: A complete implementation for PlusLang aside from the parsers defined in the `Combinator` library.

```
open Combinator

type Expr =
| Num of int
| Plus of Expr * Expr

let expr,exprImpl = recparser()
let pad p = pbetween pws0 p pws0
let num = pmany1 pdigit |>> (fun ds -> stringify ds |> int |>
    Num)
let numws0 = pad num
let plusws0 = pad (pchar '+')
let plusExpr = pseq (pright plusws0 expr) expr Plus

exprImpl := plusExpr <|> numws0
let grammar = pleft expr peof
let parse input : Expr option =
    match grammar (prepare input) with
    | Success(ast,_) -> Some ast
    | Failure(_,_) -> None

let rec eval ast : Expr =
    match ast with
    | Num n -> Num n
    | Plus (left, right) ->
        let r1 = eval left
        let r2 = eval right
        match r1, r2 with
        | Num n1, Num n2 -> Num(n1 + n2)
        | _ -> failwith "Can only add numbers."

[<EntryPoint>]
let main argv =
    if argv.Length <> 1 then
        printfn "Usage: dotnet run \"<expression>\""
        exit 1

    match parse argv.[0] with
    | Some ast ->
        printfn "Expression: %A" ast
        let result = eval ast
        printfn "Result: %A" result
    | None -> printfn "Invalid expression."
    0
```

Package Management

Code designed specifically for reuse by others is called a *software library*. We often use software libraries to simplify the development of complex applications. Library functions for the most common tasks are often supplied with the programming language installation itself. Such libraries of functions are often called the *standard library*, and all the most popular programming languages (e.g., Python, Java, C) ship with standard libraries.

Nevertheless, the set of of things we might want to do with a computer is much larger than any provided standard library. For example, I might want to do some image manipulation tasks, like resizing images or changing their colors. These functions are not common enough to be included in a standard library, but they are still common enough that somebody else—a so-called *third party*—may have written helper code. Code distributed by a third party is called a *package* and a *package manager* is a software tool that aids in downloading and installing packages.

A Brief History of Package Management

The first package manager appears to have been CTAN, the *Comprehensive TEX Archive Network*. Founded in 1992, CTAN made it easy for TEX[74] programmers to share code. Importantly one of CTAN's innovations was a rigid, standardized package format so that package management functions could be built directly into the LATEX ecosystem. It is no coincidence that CTAN was founded around the time that the Internet first became widely available at colleges and universities.[75] Consequently, it is usually very easy to incorporate someone else's code using a package manager. For example, this book was written in TEX (specifically LATEX) and, as of this writing, it uses 26 packages written by other people as well as a book template inspired by Edward Tufte (`tufte-latex`).

Package managers are now a standard part of new programming languages. Languages that eschew package management make it hard for programmers to be productive, and with the exception of C, I bet you'd be hard pressed to name a popular programming language that does not have one. Perl added CPAN in 1995, Java followed with `mvn` in 2001,

[74] TEX is pronounced "tek" and it is the underlying programming language for the LATEX document preparation system used widely by scientists and engineers. LATEX is pronounced "lah-tek."

[75] Williams College was first connected to the Internet in 1987, thanks to the efforts of the earliest members of the CS department, Profs. Tom Murtagh and Kim Bruce.

Ruby added the `rubygems` manager in 2004, and Python added the `pip` manager in 2011. Newer languages like Rust and Go came with package managers from the beginning.

The .NET language framework, of which F# is a component, was a little late to the party. .NET was initially released in 2002, but did not acquire a package manager until 2010. Fortunately, anyone now writing .NET code has access to a very large collection of software packages through the NuGet package manager.[76]. Unusually, .NET allows library code written in C#, F#, and Visual Basic to seamlessly interoperate. In fact, most of the packages available via NuGet are written in C#, and with a little practice, you'll find that using these packages in F# is generally easy.

[76] https://www.nuget.org/

Using NuGet

To demonstrate using NuGet, in this section we will develop a small image manipulation program using the ImageSharp library.[77] Although we could certainly download and use the code directly from GitHub where the project is hosted, it is easier to download precompiled packages via NuGet.

[77] https://github.com/SixLabors/ImageSharp

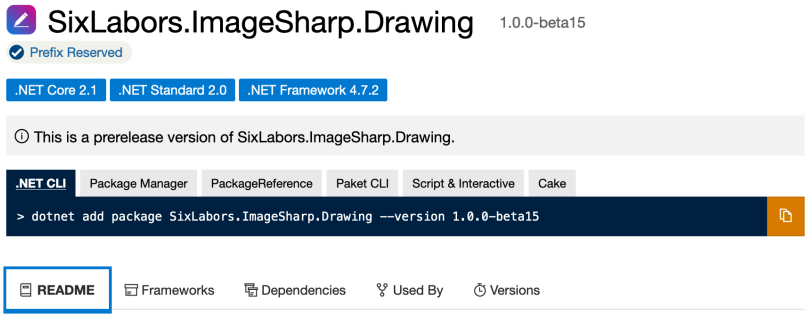
Let's start by creating a new console project.

```
$ mkdir nuget-helloworld
$ cd nuget-helloworld
$ dotnet new console -lang f#
```

To add a package to your project, you will need to use the `dotnet add package` command. We specifically want to use the `SixLabors.ImageSharp.Drawing` package.[78] Although this package itself depends on another set of packages, NuGet is smart enough to know that you must also download these *dependencies*, and it does so automatically.

[78] https://www.nuget.org/packages/SixLabors.ImageSharp.Drawing

If we visit the webpage for this package, we should see a small information panel like the one shown below.



Be sure to select the tab labeled `.NET CLI` and then copy and paste the command shown into your terminal.

```
$ dotnet add package SixLabors.ImageSharp.Drawing --version 1.0.0-beta15
```

The `dotnet` command will print some progress information, but assuming things go OK, the package should now be installed inside your project. You can verify that it was added correctly by looking at your `nuget-helloworld.fsproj` file. In mine, the following section was added.

```
<ItemGroup>
  <PackageReference Include="SixLabors.ImageSharp.Drawing" Version="1.0.0-beta15" />
</ItemGroup>
```

Using the `ImageSharp.Drawing` library

Every library will have its own application programming interface, or *API*, but I show you the steps I went through here because many of the small issues outlined below come up whenever you use C# code in F#. Documentation and sample code are available via the README page in the ImageSharp GitHub page. I have included one of the examples, written in C#, below.

```
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Processing;

namespace ResizeImage
{
    static class Program
    {
        static void Main(string[] args)
        {
            using Image image = Image.Load(args[0]);
            image.Mutate(x => x
                 .Resize(image.Width / 2, image.Height / 2)
                 .Grayscale());

            image.Save("mutated-" + args[0]);
        }
    }
}
```

Let's adapt this example to work with F#. C# is very similar to Java, so your experience programming Java should help you with this task. We

can see that this program imports two libraries[79] and then has a `main` method. We know how to do both of those things in F#, so let's start there.

[79] C# uses the `using` keyword to import libraries instead of the `import` keyword that Java uses, but they function the same way.

```
open SixLabors.ImageSharp
open SixLabors.ImageSharp.Processing

[<EntryPoint>]
let main args =
  0
```

Observe that I ignored the C# `namespace` keyword. C# namespaces are very much like F# `modules`, and we certainly could have added one, but for an example this small it is easy to stay organized without one. Also, observe that the C# program declares a `static class` called `Program`. Again, we do not need to do this because F# is not object-oriented like C#. To just make sure that F# can find the libraries we just imported, let's compile our program.

```
$ dotnet build
```

You should see a messages saying that the build succeeded. If you do not, go back and check whether you missed any steps. Next, let's examine the first line of C# code that actually does something.

```
using Image image = Image.Load(args[0]);
```

This line of code reads a filename from the first command line argument (`args[0]`) and stores the image loaded from that path in a variable called `image`. Note that the `using` keyword in this line does not have the same meaning as when it is used to import libraries. Here, it is used to automatically clean up objects that need to be "closed" before you are done with them. Such objects implement the `IDisposable` interface and usually come with a `Dispose()` method. The `using` keyword was created to address the fact that we often forget to close objects. Any `IDisposable` object created with the C# `using` keyword will be closed automatically at the end of its scope. F# has a similar keyword called `use`, which is a special form of `let`, otherwise the code is nearly the same. Actually, it's even simpler because we can rely on F#'s type inference to determine that the type of `image` is `Image`.

```
use image = Image.Load(args[0])
```

The next line of C# code resizes the image by half and converts it to

a greyscale image.

```
image.Mutate(x => x
     .Resize(image.Width / 2, image.Height / 2)
     .Grayscale());
```

Observe that the `Mutate` method takes a lambda expression. F# lambda expressions can used anywhere C# lambda is used, with only small changes to syntax.

```
image.Mutate (fun x ->
    x.Resize(image.Width / 2, image.Height / 2)
     .Grayscale())
```

Notice that the `x` parameter had to move down a line, as F# does not like having it on the previous line. There's one other issue owing to the fact that F# is functional, and expects all returned values to be used. As it turns out, `Resize` and `Grayscale`, return a kind of `Image` object, and F# is unhappy that we do nothing with it. This is what I see when I try to build the project.

```
nuget-helloworld/Program.fs(9,9):
  error FS0193: Type constraint mismatch.
  The type 'IImageProcessingContext' is not compatible with type 'unit'
  [nuget-helloworld/nuget-helloworld.fsproj]
```

Whenever we're in this situation, we can either bind that returned value to a variable using `let` or tell F# to just ignore it. ImageSharp functions are "side-effecting"[80] and this is a common pattern in object-oriented code, so ignoring returned objects is also common. The F# `ignore` keyword tells the compiler that it is OK to ignore this object.

[80] In other words, they mutate state, which a pure functional language will never do.

```
image.Mutate(x => x
     .Resize(image.Width / 2, image.Height / 2)
     .Grayscale() |> ignore);
```

After running `dotnet build` again, the build succeeds.
Finally, we can keep the last line more or less as-is.

```
image.Save("mutated-" + args[0])
```

To test our completed program, we need an image to mutate. If your computer has the `curl` command, you can use it to download an image like so.[81]

[81] You can also just use any `png` you already happen to have on your computer.

```
$ curl https://tinyurl.com/5ftvsn46 -o punk-rock-cow.png
```

Let's run our program.

```
$ dotnet run punk-rock-cow.png
```

The outputted file, `mutated-punk-rock-cow.png` is shown in Figure 23. The complete F# program is shown in Figure 24.

Figure 23: A half-size, greyscale punk rock cow.

```
open SixLabors.ImageSharp
open SixLabors.ImageSharp.Processing

[<EntryPoint>]
let main args =
    use image = Image.Load(args.[0])

    image.Mutate (fun x ->
        x.Resize(image.Width / 2, image.Height / 2)
         .Grayscale() |> ignore)

    image.Save("mutated-" + args[0])

    0
```

Figure 24: The sample code ported to F#. We had to make a couple changes, but it is very similar to the original.

Unit Testing in F#

Unit testing is a code testing method designed to demonstrate the correctness of code at the *unit* level. A *unit* is in terms of whatever the smallest functional unit is within a given language or project. For example, in functional code, a unit is often thought of as a module, function, or primitive operation. In object-oriented code, a unit is usually a class and its methods. More generally, unit testing ensures that an abstract data type (which is an abstract data structure and associated operations) produces expected inputs and outputs.

It should be noted that unit testing is only one form of testing. Furthermore, test procedures—unless they exhaustively test all possible inputs—are not *sufficient* to ensure the correctness of a unit. Nevertheless, tests are one of the easiest ways to check that a program behaves as expected and tests are one of the most important steps toward correctness. Tests are especially useful in helping to ensure that the addition of new features to a codebase does not change the expected behavior. Consequently, test methods like unit testing are widely practiced in the software industry.

Running example

We will be revisiting the code we built together as a part of the parser combinator tutorial: the code that parses sentences into a list of words. If you don't remember what we did, please revisit the reading on Parser Combinators.

In this tutorial, I encourage you to follow along on your own machine.

MsTest

Microsoft .NET comes equipped with a unit test framework called MsTest. Since MsTest has F# language bindings, we can write MsTest unit tests natively in F#. The `dotnet new` command is capable of generating an F# test project, however, in order to make such a test project useful, it needs to be combined with an existing F# console or library project. In .NET, the facility for combining two projects together is an organizational feature called a *solution*.

.NET solutions

Let's start by generating a solution that will tie an F# library and unit test project together. Solutions are the standard way of combining projects in .NET, and as long as all projects can be compiled on the .NET platform, they can be combined. For example, a solution can be composed of F#, C#, and Visual Basic projects, along with test projects, and so on.

First, create a new directory to house your solution and `cd` into it.

```
$ mkdir test_tutorial
$ cd test_tutorial
```

Now type:

```
$ dotnet new sln
```

If the solution is created successfully, you will see:

```
The template "Solution File" was created successfully.
```

Next, let's create a very simple parser library. We will reuse the sentence parser code developed in the chapter on Parsing.

```
$ mkdir SentenceParser
$ cd SentenceParser
$ dotnet new classlib -lang F#
The template "Class library" was created successfully.
```

First, download the Combinator.fs library[82]. Next, download the SentenceParser.fs library[83]. Note that these downloads are slightly different than example we worked through in Parsing. I've added some extra information to both the `Success` and `Failure` types to help with

[82] https://williams-cs.github.io/cs334-f22-www/assets/code/Combinator.fs.txt
[83] https://williams-cs.github.io/cs334-f22-www/assets/starter/SentenceParser.fs.txt

debugging. After downloading, you should have at least the following files in your `SentenceParser` folder:

```
$ ls
Combinator.fs.    Library.fs    SentenceParser.fs    SentenceParser.fsproj
```

Delete the auto-generated `Library.fs` file.

```
$ rm Library.fs
```

Open the `SentenceParser.fsproj` file and add `Parsers.fs` and `SentenceParser.fs` as compile targets. Remove the `Library.fs` target. Your `SentenceParser.fsproj` should look like this:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="Combinator.fs" />
    <Compile Include="SentenceParser.fs" />
  </ItemGroup>

</Project>
```

Now, `cd` back into the parent directory and add the `SentenceParser` project to the solution.

```
$ cd ..
$ dotnet sln add SentenceParser/SentenceParser.fsproj
```

If the project is added successfully, you will see:

```
Project `SentenceParser/SentenceParser.fsproj` added to the solution.
```

We should now be able to build our solution.

```
$ dotnet build
```

Note that, in a solution, all `.fs` library files must be inside a `module` or a `namespace`. In the files supplied, the code in `Combinator.fs` is under the `Combinator` module, and the code in `SentenceParser.fs` is under the `SentenceParser` module. Go ahead, have a look. If you forget to

do this for your own project, you will see a compile-time message like: `Files in libraries or multiple-file applications must begin with a namespace or module declaration`.

Creating the MsTest project

Now we can create the MsTest project and test our code. In the solution directory, create a new directory called `SentenceParserTests`, `cd` into it, and then use the `dotnet` tool to create an MsTest project.

```
$ mkdir SentenceParserTests
$ cd SentenceParserTests
$ dotnet new mstest -lang F#
```

If you did everything correctly, you should see:

```
The template "Unit Test Project" was created successfully.
```

We now need to make the `SentenceParser` a compile-time dependency of the `SentenceParserTests` project so that the test framework can call our library from test code.

```
$ dotnet add reference ../SentenceParser/SentenceParser.fsproj
Reference `..\SentenceParser\SentenceParser.fsproj` added to the project.
```

Finally, we need to `cd` back into our parent directory and add the `SentenceParserTests` project to the solution.

```
$ cd ..
$ dotnet sln add SentenceParserTests/SentenceParserTests.fsproj
Project `SentenceParserTests/SentenceParserTests.fsproj` added to the solution.
```

Again, running `dotnet build` should successfully build the entire project.

Understanding the test format

Let's open up the `SentenceParserTests/Tests.fs` file and have a look.

```
$ cat SentenceParserTests/Tests.fs
namespace SentenceParserTests

open System
open Microsoft.VisualStudio.TestTools.UnitTesting
```

```
[<TestClass>]
type TestClass () =

    [<TestMethod>]
    member this.TestMethodPassing () =
        Assert.IsTrue(true);
```

This file contains one test, called `TestMethodPassing`. Since `MsTest` was originally designed to test C#, tests utilize classes for organization.

Test suites. A collection of tests is called a *test suite*. Generally, a test suite is a set of tests designed to test *one unit*. For example, an entire suite might test different aspects of the same single algorithm. You might, for instance, write a test that checks for the common case for a sorting routine, another test that tests the corner case where the input is already sorted, and another test that tests another corner case where the input is empty (e.g., an empty list). All of these tests are packaged together in a *test class*, which houses the test suite. Test classes that house test suites must have the `[<TestClass>]` annotation as above.

Test methods. Each test is called a *test method*. In MsTest, each test method must literally be a method inside a test class. The test suite shown above has a single test called `TestMethodPassing`. There are two important facts to note about test methods. First, the method is prefixed with the `[<TestMethod>]` annotation. Second, test methods must be no-parens functions; or more precisely, they need to F# functions that take `unit`. The test above does nothing; it simply asserts `true`, which forces a test to pass.

Note that it is *up to you* how you want to organize your tests into test suites. Choose the organization that you find most useful. Out of laziness, I usually just put all the tests for an entire `module` inside a single test suite, and only break it into separate test suites once the test suite has grown to an unmanageable size. Remember, programming is an art, not a science!

Running the tests

If you are in the `SentenceParserTests` folder, you can run `dotnet test` and you should see output that looks a bit like this.

```
$ dotnet test
  ... some output omitted ...
Microsoft (R) Test Execution Command Line Tool Version 17.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed: 0, Passed: 1, Skipped: 0, Total: 1, Duration: 15 ms
```

Above, you can see that the entire test suite consisting of a single test passed (`Passed!`) and took 15 milliseconds to run.

You can also run the `dotnet test` command from the parent directory which contains the solution. In that case, you will see test output from every project that actually contain tests.

Adding a new test

Finally, let's add a new test that actually tests our parser. In fact, let's get rid of the silly parser that always succeeds.

At the highest level, a hand-wavy description of our parser is that it takes a string representing a sentence and turns it into a list of words. The purpose of a test is to ensure that such hand-wavy descriptions are backed up with real code that does what you say and is checked every time you run the test suite. A nice side-effect of such tests is that they serve to document use cases for your code.

First, add an `open` statement to the top of your `SentenceParserTests/Tests.fs` file so that it can access your `SentenceParser` library.

```
open SentenceParser
```

Next, replace the test `TestClass` with a new one. Here is the complete code for the `Tests.fs` file:

```
namespace SentenceParserTests

open System
open Microsoft.VisualStudio.TestTools.UnitTesting
open SentenceParser

[<TestClass>]
type TestClass () =

    [<TestMethod>]
    member this.ValidSentenceReturnsAWordList() =
        let input = "The quick brown fox jumped over the lazy dog."
        let expected = [ "The"; "quick"; "brown"; "fox"; "jumped"; "over"; "the"; "lazy"; "dog" ]
        let result = parse input
        match result with
        | Some ws ->
            Assert.AreEqual(expected, ws)
        | None ->
            Assert.IsTrue false
```

The logic is as follows. We supply an input called `input`, which is a sentence. We also supply an `expected` value, which is the output we *expect* `parse` to produce when given the `input`. Next, we call `parse` with `input` and store it in `result`. Since `parse` returns an `option` type (`Some` if the parser succeeds, `None` if it does not), we pattern-match on `result`. Finally,

1. if we get back `Some` word list `ws`, we check that `ws` is exactly the same as the word list `expected`. Note the position of the `expected` parameter. While `Assert.AreEqual` will fail anytime its two arguments differ, when it fails, it returns a helpful message based on the contents of the `expected` parameter. Otherwise,
2. if we get back `None`, then the parse failed when it should have succeeded. In this case, we force the test to fail by supplying `Assert.IsTrue false`.

Running `dotnet test` reports:

```
$ dotnet test
  ... some output omitted ...
Microsoft (R) Test Execution Command Line Tool Version 17.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed: 0, Passed: 1, Skipped: 0, Total: 1, Duration: 86 ms
```

So far so good.

Test-driven development

One of the many buzzwords you may hear out in industry is something called "test-driven development" or TDD. The idea behind TDD is to write tests *before* you write your implementation code. While there are many fads in software development, I believe that this is genuinely a good idea. For starters, providing an example of input and output often focuses your implementation efforts. Second, input and output examples fit nicely with functional programming, since, if you're doing it correctly, every function should be *pure* and every input should unambiguously produce the same output every time. [84]

[84] For deterministic functions.

Let's add a test for a feature we do not yet have: the ability to parse questions.

```
[<TestMethod>]
member this.ValidQuestionReturnsAWordList() =
    let input = "Does the quick brown fox jump over the lazy dog?"
    let expected = [ "Does"; "the"; "quick"; "brown"; "fox"; "jump"; "over"; "the"; "lazy"; "dog" ]
    let result = parse input
    match result with
    | Some warr ->
        Assert.AreEqual(expected, warr)
    | None ->
        Assert.IsTrue false
```

Running `dotnet test` produces our first failing test, because we do not yet support this feature.

```
$ dotnet test
  ... some output omitted ...
Microsoft (R) Test Execution Command Line Tool Version 17.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
```

```
A total of 1 test files matched the specified pattern.
  Failed ValidQuestionReturnsAWordList [35 ms]
  Error Message:
   Assert.IsTrue failed.
  Stack Trace:
     at SentenceParserTests.TestClass.ValidQuestionReturnsAWordList() in [...] Tests.fs:line 30

  Standard Error Messages:
 [attempting: grammar on "Does the quick brown fox jump over the lazy dog?", next char: 0x44]
 [attempting: sentence on "Does the quick brown fox jump over the lazy dog?", next char: 0x44]
 [attempting: sprefix on "Does the quick brown fox jump over the lazy dog?", next char: 0x44]
 [attempting: upperword on "Does the quick brown fox jump over the lazy dog?", next char: 0x44]
 [success: upperword, consumed: "Does", remaining: " the quick brown fox jump over the lazy dog?", next char: 0x20]
 [attempting: words0 on " the quick brown fox jump over the lazy dog?", next char: 0x44]
 [attempting: word on "the quick brown fox jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "the", remaining: " quick brown fox jump over the lazy dog?", next char: 0x20]
 [attempting: word on "quick brown fox jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "quick", remaining: " brown fox jump over the lazy dog?", next char: 0x20]
 [attempting: word on "brown fox jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "brown", remaining: " fox jump over the lazy dog?", next char: 0x20]
 [attempting: word on "fox jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "fox", remaining: " jump over the lazy dog?", next char: 0x20]
 [attempting: word on "jump over the lazy dog?", next char: 0x44]
 [success: word, consumed: "jump", remaining: " over the lazy dog?", next char: 0x20]
 [attempting: word on "over the lazy dog?", next char: 0x44]
 [success: word, consumed: "over", remaining: " the lazy dog?", next char: 0x20]
 [attempting: word on "the lazy dog?", next char: 0x44]
 [success: word, consumed: "the", remaining: " lazy dog?", next char: 0x20]
 [attempting: word on "lazy dog?", next char: 0x44]
 [success: word, consumed: "lazy", remaining: " dog?", next char: 0x20]
 [attempting: word on "dog?", next char: 0x44]
 [success: word, consumed: "dog", remaining: "?", next char: 0x3f]
 [success: words0, consumed: " the quick brown fox jump over the lazy dog", remaining: "?", next char: 0x3f]
 [success: sprefix, consumed: "Does the quick brown fox jump over the lazy dog", remaining: "?", next char: 0x3f]
 [attempting: period on "?", next char: 0x44]
 [failure at pos 48 in rule [pchar '.']: period, remaining input: "", next char: EOF]
 [failure at pos 48 in rule [pchar '.']: sentence, remaining input: "", next char: EOF]
 [failure at pos 48 in rule [pchar '.']: grammar, remaining input: "", next char: EOF]


Failed! - Failed: 1, Passed: 1, Skipped: 0, Total: 2, Duration: 116 ms
```

This output says that the `ValidQuestionReturnsAWordList` test failed. It failed, of course, because we have not yet implemented this feature. Observe that, since we used the `debug` feature, the failing test printed out what the program echoed. Tests *only print output when they fail*.

Let's implement the feature.

A Question Parser

I am not going to belabor parsers again here, so let's fast-forward to the most intuitive feature addition. First, add a `qmark` parser.

```
let qmark = (pchar '?') <!> "question mark"
```

Next, modify the `sentence` parser so that it accepts either a period or a question mark.

```
let sentence = pleft prefix (period <|> qmark) <!> "sentence"
```

The complete, modified code is as follows:

```
module SentenceParser

open Combinator

let qmark = (pchar '?') <!> "question mark"
let period = (pchar '.') <!> "period"
let word = pfun (pmany1 pletter) (fun cs -> stringify cs) <!> "word"
let upperword = pseq pupper (pmany0 pletter) (fun (x,xs) -> stringify (x::xs)) <!> "upperword"
let words0 = pmany0 (pright pws1 word) <!> "words0"
let prefix = pseq upperword words0 (fun (w,ws) -> w::ws) <!> "sprefix"
let sentence = pleft prefix (period <|> qmark) <!> "sentence"
let grammar = pleft sentence peof <!> "grammar"

let parse input : string list option =
    match grammar (prepare input) with
    | Success(ws,_) -> Some ws
    | Failure(_,_) -> None
```

Let's test it again.

```
$ dotnet test
  ... some output omitted ...
Microsoft (R) Test Execution Command Line Tool Version 17.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed: 0, Passed: 2, Skipped: 0, Total: 2, Duration: 79 ms
```

Looks good!

Conclusion

In this tutorial, we learned:

- How to create a solution.
- How to add a project to a solution.
- How to add a test project to a solution.
- How to add a test.
- How to run tests.
- How to do test-driven development, where tests are written before implementation code.

I encourage you to add tests to your own projects. This means that you will probably need to "wrap" your existing projects in a solution, but the above tutorial should be enough of a guide to get you started.

There are many additional `Assert` methods beside the `AreEqual` method. For additional information, see the documentation on the MsTest `Assert` class[85]. After clicking on the link, look for the "Methods" dropdown in the left column.

Finally, if you want another tutorial, have a look at Microsoft's official F# unit test tutorial[86] which goes into more detail than this tutorial.

[85] https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=mstest-net-1.2.0

[86] https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-fsharp-with-mstest

Implementing Variables

At this point in your computer science education, you've probably written many programs that used variables. Without them, we would not be able to abstract over data. Different inputs would require different programs, even when those inputs were essentially the same. In fact, it's pretty hard to imagine a program does not use any variables at all.

But what is a variable, exactly? Furthermore, how are variables *supposed* to behave in a programming language? Variables are of foundational importance to most programming languages. Understanding how they are implemented will give you a deeper appreciation for the meaning of programs. Moreover, their implementations will help you understand why we rely so heavily on language models like the lambda calculus when designing languages.

Pluslang+

In this section, we will revisit our PlusLang interpreter from the chapter on *Evaluation*. PlusLang+ adds a little syntax to support variables and augments the PlusLang interpreter with variable support. A complete PlusLang implementation is available at the end of that chapter, so if you want to follow allow, feel free to copy it.

In PlusLang, we had two kinds of expressions. We had numeric expressions like `3`. We also had plus-expressions like `+ <expr> <expr>`. Let's add two more kinds of expressions to support variables.

First, we need to allow users to *assign* a value to a variable. A let-expression has the form `let <var> <expr>`. It defines a new variable if it does not already exist and assigns the value of `<expr>` to it. Let's restrict `<var>` to be any single letter for simplicity. Second, we need to be able to support *using a variable*. Whenever a `<var>` appears outside of a let-expression, we will call that a var-expression.

At the risk of overcomplicating our example, I think you will find it convenient to have one more kind of expression. This last form is conventionally called *sequential composition*. You have most definitely used sequential composition countless times in a programming language and have never thought about it. It means "do this first and then do that"

and it is fundamental to the operation of a computer program. Because humans also to think in these terms, though, you probably have never given it a second thought. In many languages, it is represented by a `;` character. Here's an example from a programming language you already know.

```
int x = 2;
x += 2;
```

If we erase the bits of the program that don't have to do with sequential composition, you get

```
expr_1;
expr_2;
expr_3
```

which literally says "do *expr_1* and then do *expr_2* and then do *expr_3*." The last expression, *expr_3*, happens to be empty, which we might interpret as doing nothing. We'll talk more about how sequential composition is evaluated in this and coming chapters.

Our BNF grammar for PlusLang+ is the following.

```
<expr>  ::= + <expr> <expr>
          | let <var> <expr>
          | tt <expr> <expr>
          | <var>
          | <num>
<num>   ::= <digit>+
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<var>   ::= α ∈ a...z
```

For example, we might have the following program.[87]

```
tt let x 1 + x x
```

which evaluates to 2.

As in the chapter on *Evaluation*, this chapter uses prefix form. Remember that the role of the AST is to abstract syntax from semantics. In one sense, the appearance of the expression `tt let x 1 + x x` does not matter; all the computer needs to know is how to convert that string into an AST. In another sense, it does matter. Humans tend to prefer intuitive syntax. I'm favoring a syntax that is easy to parse in order to keep these readings short, but if it helps, try converting the syntax into something more familiar, like `x = 1; x + x`.[88]

It might be hard for you to see why the above expression evaluates to

[87] Remember that my aim with this language is to help you understand difficult concepts, so PlusLang+ programs are admittedly not very intuitive. Intuitive or pretty languages tend to complicated. I suspect that you prefer simplicity while you learn, so what PlusLang+ lacks in pizzazz it makes up for in simplicity.

[88] If you're looking for a challenge, extend my parser to support the more complex syntax. If you do it correctly, you won't even need to change our `type` definition or `eval`.

2 just yet, so let's work through this language one piece at a time.

PlusLang+ AST

To support let-, var-, and sequential composition expressions, our AST will need to be extended.

```
type Expr =
| Num of int
| Var of char
| Plus of left: Expr * right: Expr
| Let of var: Expr * e: Expr
| ThisThat of this: Expr * that: Expr
```

We added `Var` for variable use, `Let` for let-expressions (i.e., variable assignment), and `ThisThat` for sequential composition.

Observe that I define `Let` as `Expr * Expr` instead of `Var * Expr`. The reason is that `Var` is not a type. Rather, it is one alternative value of `Expr`. Therefore, we cannot use `Var` where F# expects a type. This is a limitation of the F# type system, but it is only mildly inconvenient. Alternatively, we could have defined `Let` as `char * Expr`. I chose this version because it turns out to be the simpler of the two approaches, but the choice is somewhat arbitrary.

PlusLang+ Parser

We are going adapt our existing PlusLang parser, so with the exception of the `expr` parser, all of our existing parsing rules will remain untouched. Let's start by adding support for parsing variable names, which are single letters. As with numbers, we are happy to let the user supply whitespace wherever they want.

```
let var = pad pletter |>> Var
```

Parsing let-expressions and sequential composition seems like a lot of work until you realize that it follows the same pattern as a plus-expression. The general pattern is *op* `expr1 expr2`. Our parsing rule for plus-expressions came in two parts:

```
let plusws0 = pad (pchar '+')
let plusExpr = pseq (pright plusws0 expr) expr Plus
```

The first part, `plusws0`, parses a `+` sign surrounded by optional whitespace. The second part looks for the first part followed by two `expr`s, and when successful, gives those two expressions to the `Plus` constructor.

Let-expressions and sequential composition work the same way.

```
let letws0 = pad (pstr "let")
let letExpr = pseq (pright letws0 var) expr Let

let ttws0 = pad (pstr "tt")
let ttExpr = pseq (pright ttws0 expr) expr ThisThat
```

Finally, we update `exprImpl` to support the new expressions.

```
exprImpl := plusExpr <|> letExpr <|> ttExpr <|> numws0 <|> var
```

PlusLang+ Evaluator

If you were following along in the chapter on *Evaluation*, you'll remember that I said that the cases of our `Expr` strongly suggest the structure of the `eval` function. That is also true here. We added three more cases to `Expr`, so that means that the `match` expression in `eval` will also need three additional cases. We will need to fill in the missing bits of this template:

```
let rec eval ast: int =
    match ast with
    | Num n -> Num n
    | Plus (left, right) ->
        let r1 = eval left
        let r2 = eval right
        match r1, r2 with
        | Num n1, Num n2 -> Num(n1 + n2)
        | _ -> failwith "Can only add numbers."
    | Var c -> ...
    | Let (var, e) -> ...
    | ThisThat (this, that) -> ...
```

But there's also something more important missing here. Mechanically speaking, what does a variable do? On a computer, it *temporarily stores* a value. Whenever we talk about storage on a computer, we're really talking about putting a value in a predetermined memory location. Since you've all taken a data structures course, you know that we can put rather sophisticated things in memory if we need to. All of the abstract data types (ADTs) you learned about are available to you when you implement a language. For variables, what we need is some kind of *mapping* from variable names to variable values. Can you think of an appropriate ADT for this task? We will need some storage soon so we

will return to this question shortly.

Here's the shortest possible program in PLUSLANG+ that has both a let-expression and a var-expression:

```
tt let x 1 x
```

The above expression has the AST shown in Figure 25. The program first assigns `1` to the variable `x`, and then it accesses the value of `x`. We will evaluate this expression's AST just as we did in the chapter on *Evaluation*, starting at the top, which is the `ThisThat` node.

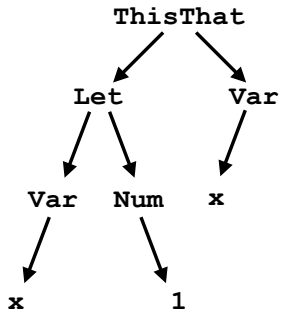


Figure 25: The AST for the subexpression `tt let x 1 x`.

Before we continue, there's just a quick note to make about the order of evaluation of a node's children. Colloquiually, in a pre-order traversal, we evaluate children from left to right. Nevertheless, "left" and "right" are fictions in a computer—it has no notion of direction. So when we evaluate an AST node, we really need to think about the order to evaluate its children in the context of the operation we are implementing. Keep that in mind, as you will see that let-expressions cannot be evaluated from left-to-right.[89]

[89] Unless I rearrange the drawing in a way that I find to be confusing.

PLUSLANG+ Evaluator: Sequential Composition

As we defined it, `ThisThat` has two children, `this` and `that`. Naturally, we should do *this* before we do *that*. Let's update our code.

```
| ThisThat (this, that) ->
  let r1 = eval this
  let r2 = eval that
  r2
```

Hopefully the first two lines strike you as intuitive. If we want to *do* `this`, then we need to evaluate it. Since we want to evaluate `this` before `that`, then we write them in the order shown. But you may be puzzled by the last line, `r2`.

The last line *returns* the result of evaluating `that`. In most languages,

sequential composition returns nothing. However, I am a fan of expression-oriented languages. In those languages, everything is an expression, and expressions return something. For example, F#, Scala, Ruby, and Rust are all expression-oriented. This is also a good opportunity to point out that some design choices are a matter of opinion, and this is one of them. Therefore, we define our expression in PLUSLANG+ to simply return the *last* thing evaluated, namely `that`.

PLUSLANG+ Evaluator: `Let`

If we first evaluate `this`, which is shown as the left subtree in Figure 25, the recusive call to our `eval` function next considers a let-expression. Again, we must be mindful about the fiction of "left" and "right" in a computer. `let` stores the result of `<expr>` in `<var>`. The drawing shows `<var>` on the left and `<expr>` on the right. We have to know the value of `<expr>` before we do anything with `<var>` so we will evaluate the right child first.

```
| Let (var, e) ->
  let r = eval e
  ...
```

What comes next though? We have two things to do. First, we must find out what name `var` refers to. Next, we must store a mapping from that name to the value of `<expr>`. Here's where we return to the question of an appropriate data structure. Put another way, what's a convenient data structure for associating a value (like an evaluated expression) with a key (like a variable name)? If you thought "a dictionary," great![90] That's what we're going to use.

To find out the name of `var`, we cannot just evaluate it. Remember, we decided that evaluating a variable returns the value stored in that variable. We need to do something simpler; we just need to know the variable's name. Let's write a helper function. We need the helper function because I defined the `var` field of our `Let` AST node as an `Expr`.

```
let charFromVar e =
    match e with
    | Var c -> c
    | _ -> failwith "Expression is not a variable."
```

Since we always expect `var` to be a `Var` (see the `var` parser above—it can't be anything else), we just need the `_` case to prove to the F# type checker that we've considered that possibility. Now we can finish our our implementation of `eval` for `Let` nodes.

[90] Likewise, if you thought "a map" or "an associative array," rest assured that those names all refer to the same ADT. If you thought "a hash table," also good, but remember that a hash table is merely an implementation of the dictionary ADT. You could also implement a dictionary using a binary search tree or even an ordered vector.

```
| Let (var, e) ->
  let r = eval e
  let c = charFromVar var
  d[c] <- r
  r
```

The expression `d[c] <- r` says "mutably store `r` in the dictionary `d` for key `c`."[91] So now we just need to define `d`, our dictionary, somewhere. I am going to define `d` at the top level of my program, before anything else.

[91] The `Dictionary` implementation I am using comes from C# and is mutable.

```
let d = new System.Collections.Generic.Dictionary<char,Expr>()
```

Finally, because I am opinionated and like expression-oriented languages, `Let` returns a value.[92]

[92] `let x let y 1` is a valid program. It looks strange, but is there anything really wrong with returning a value in an assignment? I argue no, and in fact, many statement-oriented languages (like C) make this choice as well.

PlusLang+ Evaluator: *Var*

After the left subtree containing `Let` is evaluated, our `ThisThat` operation will evaluate the right subtree. The right subtree contains a `Var` node, so let's define evaluation for `Var`. As stated before, evaluating `Var` should return the value of the variable, if the variable exists. `Var` is more or less the converse of `Let`.

```
| Var c ->
  if (d.ContainsKey c) then
    d[c]
  else
    failwith ("Unknown variable '" + c.ToString() + "'")
```

The complete implementation of PlusLang+ with an updated `main` method is shown in Listing 2. We can run PlusLang+ just like we ran PlusLang, and the language is a superset of PlusLang. Let's try the expression I used to motivate this chapter.

```
$ dotnet run "tt let x 1 + x x"
Expression: Plus (Plus (Num 1, Plus (Num 2, Num 3)), Plus (Num 4, Num 5))
Result: 15
```

"These are not the variables you are looking for."

While this mini-language implements working variables that may be sufficient for many purposes, if you use them, you will quickly discover that they behave much differently than variables in your favorite programming language. In PlusLang+, all of our variables have *global scope*. The term *scope* refers to the region of a program wherein a defini-

tion is valid. The term *global* refers to the entire program. So globally-scoped variables are defined throughout the entire program.

To see why this is not a great idea, let's look at a toy program in Java.

```
class NotGloballyScoped {
  public static void countDown() {
    for (int i = 5; i >= 0; i--) {
      System.out.println(i);
    }
  }

  public static void main(String[] args) {
    int i = 7;
    countDown();
    System.out.print(i);
  }
}
```

Running this program prints out `5 4 3 2 1 0 7` as you might expect. But we can change the scope of the variable `i` to be global, like so.

```
class GloballyScoped {
  public static int i;

  public static void countDown() {
    for (i = 5; i >= 0; i--) {
      System.out.print(i + " ");
    }
  }

  public static void main(String[] args) {
    i = 7;
    countDown();
    System.out.println(i);
  }
}
```

By making the variable `i` `static` in Java, we declare that we want it to be stored in global program storage for the lifetime of the program. Now when we run the program, it prints `5 4 3 2 1 0 -1`. If this is not what you wanted to happen, you might find it upsetting. In short, the function `countDown` uses the same variable `i` as the one used by `main`. As a result, `i` is modified during the call to `countDown`. Since we usually write methods or functions in isolation, using the same variable name twice is easy to do.[93] For a small program like this, a little discipline

[93] Accidentally declaring variables with global scope is a common mistake and source of frustration for students in CSCI 136. Perhaps you made this mistake yourself once or twice yourself.

about the use of variable names will avoid this problem. But as programs grow larger, it is harder not to reuse a variable name, especially for variables commonly used to represent things like loop indices. If that variable name happens to be globally scoped, surprising and upsetting things can happen.

To address this issue, we will add scope to our variables. We will discuss this modification in the next chapter, *Implementing Functions*.

Listing 2: A complete implementation for PLUSLANG+.

```
open Combinator

type Expr =
| Num of int
| Var of char
| Plus of left: Expr * right: Expr
| Let of name: Expr * e: Expr
| ThisThat of this: Expr * that: Expr

let d = new System.Collections.Generic.Dictionary<char,Expr>()
    // mutable dictionary

let expr,exprImpl = recparser()
let pad p = pbetween pws0 p pws0

let num = pmany1 pdigit |>> (fun ds -> stringify ds |> int |>
    Num)
let numws0 = pad num
let plusws0 = pad (pchar '+')
let plusExpr = pseq (pright plusws0 expr) expr Plus
let var = pad pletter |>> Var
let letws0 = pad (pstr "let")
let letExpr = pseq (pright letws0 var) expr Let
let ttws0 = pad (pstr "thisthat")
let ttExpr = pseq (pright ttws0 expr) expr ThisThat

exprImpl := plusExpr <|> letExpr <|> ttExpr <|> numws0 <|> var
let grammar = pleft expr peof
let parse input : Expr option =
    match grammar (prepare input) with
    | Success(ast,_) -> Some ast
    | Failure(_,_) -> None

let charFromVar e =
    match e with
    | Var c -> c
    | _ -> failwith "Expression is not a variable."

let rec eval ast : Expr =
    match ast with
    | Num n -> Num n
    | Plus (left, right) ->
        let r1 = eval left
        let r2 = eval right
        match r1, r2 with
        | Num n1, Num n2 -> Num(n1 + n2)
        | _ -> failwith "Can only add numbers."
    | Var c ->
        if (d.ContainsKey c) then
            d[c]
```

```
        else
            failwith ("Unknown variable '" + c.ToString() + "'")
    | Let (var, e) ->
        let r = eval e
        let c = charFromVar var
        d[c] <- r
        r
    | ThisThat (this, that) ->
        let _ = eval this
        let r = eval that
        r

[<EntryPoint>]
let main argv =
    if argv.Length <> 1 then
        printfn "Usage: dotnet run \"<expression>\""
        exit 1

    match parse argv.[0] with
    | Some ast ->
        printfn "Expression: %A" ast
        let result = eval ast
        printfn "Result: %A" result
    | None -> printfn "Invalid expression."
    0
```

Implementing Scope

In the last chapter on *Implementing Variables*, we learned how to implement global variables. The key insight when implementing variables is that a programming language evaluator must maintain a mapping between variable names and their values. We saw that one way you might implement variables is with a simple, global mapping. Unfortunately, this global mechanism proves insufficient when we want to use variables with functions.

In this chapter, we focus on a concept called *scope*. Although we consider scope here in a form that you will rarely encounter in "real life," as a standalone feature, doing so will help you understand what it is and why it is important. When we implement functions in the next chapter, you will see that they are little more than a thoughtful composition of variables and scope.

What is Scope?

As stated in the previous chapter, *scope* refers to the region of a program wherein a definition is valid. But scope is more than that. Scope is a procedure—an algorithm—for determining the value of a variable at a given point in a program's execution. Let's start simply, by considering the global scope model we used in the last chapter.

In a language with global scope, each variable in the program is unique. For example, if you refer to a variable `x` at the beginning of your program, and you refer to a variable `x` later in your program, that's the same `x` variable in both places. Let's consider an example from a C-like language.[94]

[94] In C, `printf` is very much like F#'s `printfn`.

```
int x = 3;
int y = 21;
x = 4;
int z = 7;
printf("%dn", x);
```

It is probably not surprising to you that the above program prints `4`. Because each variable name maps uniquely to a value, global scope can

be implemented with a simple dictionary-like mechanism as we did in the last chapter. Here's the state of that mapping at the last line of the program.

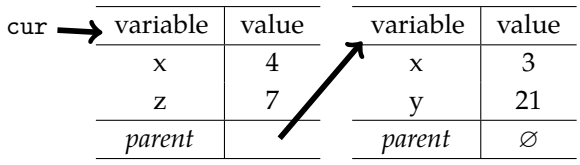
variable	value
x	4
y	21
z	7

In many languages, including C, we can ask the language to give us a *new* scope using `{` and `}` syntax. In C, regions of code enclosed inside curly braces are called *blocks*.

```
int x = 3;
int y = 21;
{
  int x = 4;
  int z = 7;
}
printf("%dn", x);
```

You may be surprised to learn that the above program prints `3`. Why? The answer is because the `printf` statement refers to the first `x`, not the second `x`. The two `x` variables in this program are different variables.

Clearly, the simple dictionary-like mechanism we used above does not capture whatever this simple C program is doing with its variables. In fact, when the program executes inside the block, variable values are found in a *different* dictionary. `cur` points at the dictionary that is current when the program is executing *inside* the block.



This model is more sophisticated than the global scope model. Observe that it contains two different instances of the variable `x`. The current dictionary is linked to a second dictionary. The definition of `x` in the current dictionary hides[95] the other definition of `x` in the linked dictionary. The way lookup works in this model is as follows. Suppose we are looking for a variable with name α.

[95] The word that language designers use is *shadows*.

1. Look for α in the current dictionary.
   (a) If found, return the value of α.
   (b) If not found and there is a parent dictionary, follow the parent pointer and search the parent dictionary for α.

2. Otherwise, fail with an error.

If your recursion chops are good, you might recognize lookup as a recursive procedure. In fact, chains of linked dictionaries can be arbitrarily long. If you're thinking "this looks like a linked list of dictionaries," you're right! This structure is sometimes called a *scope chain*. To clarify, let's define a scope chain using a simple F# type.

```
type Scope =
| Base
| Env of m: Map<char,Expr> * parent: Scope
```

In keeping with our simple model for variables from our last chapter, our dictionary is restricted to one-letter variable names (`chars`). However, observe that I am now using `Map` instead of `Dictionary`. The main difference is that `Map` is an immutable data structure, whereas `Dictionary` is mutable. I am making this change now because I would like us to move toward a more mathematical understanding of scope. Futhermore, we do not need a mutable dictionary, and keeping it around will leave us prone to state-handling errors.

Nevertheless, hopefully you can see that the above definition for `Scope` is the same structure as shown in the linked tables above. There are two cases. When there is a current mapping, what we call an *environment* (`Env` above), we will do our lookup there. If there is not a current mapping (`Base`), lookup fails, because lookup has reached the end of the scope chain.

PlusLang++

Let's extend PlusLang+ with scopes. Like the C example above, we will allow the user to be able to create and destroy new scopes. Our implementation will use the `Scope` type we defined above, but to keep things simple, we will restrict the language so that the current scope always points at the head of the `Scope` linked list. In this way, our scope chain will function as a stack.

PlusLang++ AST

Since scope will function like a stack, that means we need to add support for two operations. The first, `push` will create a new scope and push it to the head of the scope chain. The second, `pop` will discard the scope at the top of the scope chain. Our first order of business is to modify our `Expr` type to support these two new operations in our ADT.

```
type Expr =
| Num of int
| Var of char
| Plus of left: Expr * right: Expr
| Let of name: Expr * e: Expr
| ThisThat of this: Expr * that: Expr
| ScopePush of e: Expr
| ScopePop of e: Expr
```

For consistency, and because we always want to do *something* after we modify our scope chain, both `push` and `pop` operations take an `Expr`. The `push <expr>` operation adds a scope to the head of the scope chain and then evaluates `<expr>`. The `pop <expr>` operation *first evaluates* `<expr>` and then removes the scope from the head of the scope chain.[96]

[96] We need `pop` to work in the reverse order as `push` for the next chapter.

PlusLang++ Parser

Next, we must extend the language's parser to support these two operations. As with our other operations, we will recognize the operands (`push` and `pop`) with a parsing rule that allows whitespace, and then we will define a new parser that recognizes the entire expression.

```
let spushws0 = pad (pstr "push")
let spopws0 = pad (pstr "pop")
let scopeExpr =
  (pright spushws0 expr |>> ScopePush) <|>
  (pright spopws0 expr |>> ScopePop)
```

Now we add our `scopeExpr` parser to our `expr` parser.

```
exprImpl :=
  plusExpr <|>
  letExpr <|>
  ttExpr <|>
  scopeExpr <|>
  numws0 <|>
  var
```

PlusLang++ evaluator

Let's now consider the changes to our evaluator. Clearly, we need to add support for `push` and `pop`. However, there is one other alteration we must make, because we changed the data structure we use to store variables. The reason we're doing this is so that we have precise control over which portion of a scope chain is visible at any given time.

To make this change, we need to add an argument to `eval` and we also need to change its return type. First, instead of declaring a global dictionary `d` for variables, we will modify `eval` to take a second argument `s`, the scope chain. The appropriate `Scope` can then be passed in whenever `eval` is called. Second, `eval` should return not just the result of evaluation, but a 2-tuple (`Expr * Scope`) that includes both the result and the current scope chain. Returning the scope chain makes it possible for a language operation to recursively alter the `Scope` data structure and then to be able to see the result of that change when recursive evaluation returns. If this last part is not clear to you, you'll see what I mean as we work through our implementation.

To summarize, `eval`'s declaration should change from

```
let rec eval (ast: Expr): Expr  =
  ...
```

to

```
let rec eval (ast: Expr)(s: Scope): Expr * Scope =
  ...
```

Because our new `eval` function takes a `Scope`, we now need to create one when we call `eval` for the first time. We call `eval` for the first time in our `main` function.

```
// initialize root scope
let env = Env(Map.empty, Base)
```

Sadly, this change does mean that we need to modify all of the implementations for our existing AST nodes. Fortunately, our language is still small, so we will walk through modifying the logic for each AST node in turn. For each AST node, we must ask the following questions.

- Does any part of this operation modify a scope?
- What scope should be returned with the result?

Let's start at the top with `Num`, which we had previously defined like so.

```
| Num n -> Num n
```

Does any part of this operation modify a scope? No. It just returns a number. *What scope should be returned with the result?* Since we're not changing anything, we should probably just return the same scope we started with. The following modification works fine.

```
| Num n -> Num n, s
```

Next we consider `Plus`. *Does any part of this operation modify a scope?* You might think *no, it's just addition*, but remember, addition takes two `Expr` operands. Those `Exprs` must eventually evaluate to `Nums`, but they can be anything unevaluated, including a variable assignment or a scope operation. For example, the expression `+ let x 1 2` will modify a `Scope` by creating a mapping for `x` to `1`. So the answer here is *yes*, addition modifies a scope. Let's consider just the first two lines of our old definition.

```
| Plus (left, right) ->
    let r1 = eval left
    ...
```

`eval` now requires an additional argument, and it returns a second value. Let's supply our scope `s` to `eval` and capture the second returned value, the updated scope.

```
| Plus (left, right) ->
    let r1, s1 = eval left s
    ...
```

When we evaluate `right`, we need to do the same thing.

```
| Plus (left, right) ->
    let r1, s1 = eval left s
    let r2, s2 = eval right s1
    ...
```

Observe that we use the *updated scope* `s1` when evaluating `right`. Finally, we can pattern-match to ensure that `r1` and `r2` are `Nums` and then do the addition. We return both the new number and the latest scope.

```
| Plus (left, right) ->
    let r1, s1 = eval left s
    let r2, s2 = eval right s1
    match r1, r2 with
    | Num n1, Num n2 -> Num(n1 + n2), s2
    | _ -> failwith "Can only add numbers."
```

All of our expressions are going to need to deal with scopes like this. In the interest of not boring you, I provide a complete implementation at the end of this chapter (see Listing 3). For the remainder of this chapter, let's focus on how we maintain variables in our new `Scope` data structure, and then we will wrap up with support for `push` and `pop`.

Reading and Writing to Variables

Here is our updated code for `let` expressions and variable lookup expressions.

```
| Var c ->
    lookup c s, s
| Let (var, e) ->
    let r, s1 = eval e s
    let c = charFromVar var
    let s2 = store c r s1
    r, s2
```

The main differences from our old definition, aside from careful management of `Scope`, is that we now have `lookup` and `store` helper functions. How do they work?

```
let rec lookup c s: Expr =
    match s with
    | Base ->
        failwith ("Unknown variable '" + c.ToString() + "'")
    | Env (m, parent) ->
        if (Map.containsKey c m) then
            Map.find c m
        else
            lookup c parent
```

The `lookup` function above looks up a variable with name `c` in `Scope s`. If the data structure is in the base case, then lookup fails. Lookup can only fail when the variable does not exist in the map. For example, the PLUSLANG++ program `z` will print the error message `Unknown variable 'z'`. Otherwise, `lookup` will search the `Scope` for the first definition of the variable it finds. It will keep searching down the chain until it reaches the base case.

`store` inserts a new mapping from name `c` to value `v` into `Scope s`. Since `Scope` is immutable, it returns a new `Scope` data structure. If the mapping for the variable already exists in the current map, we overwrite it.[97]

[97] Whether to overwrite it or to fail with an error is a choice you can make as a language designer.

```
let store c v s: Scope =
    match s with
    | Base -> failwith "Cannot store to base scope."
    | Env (m, parent) ->
        let m' = Map.add c v m
        Env (m', parent)
```

push and pop

Finally, we come to the `push` and `pop` operations. For all the talk,[98] these are actually simple to implement.

[98] I know this chapter is long, and I am sorry.

The `push` operation adds a new `Env` to the head of the scope chain and then evaluates its `Expr` argument.

```
| ScopePush e ->
    let s1 = Env (Map.empty, s)
    eval e s1
```

The `pop` operation evaluates its `Expr` argument in the context of the *parent* of the scope at the head of the scope chain. This has the same effect as removing the head. To make this definition simple, we first define a `parentOf` helper function. `parentOf` fails if it is called in the `Scope` base case.

```
let parentOf env: Scope =
    match env with
    | Base ->
        failwith "Cannot get parent of base scope."
    | Env (_, parent) -> parent
```

Here's the definition for `pop`.

```
| ScopePop e ->
      let res,s1 = eval e s
      let parent = parentOf s1
      res,parent
```

Putting it All Together

A complete implementation for PlusLang++ can be found in Listing 3. To make it easier to enter in complicated multi-line programs, this implementation has been modified to read a program from a file instead of from command line arguments. Let's try a small program that manipulates scopes and see whether it handles them correctly. I like to use the convention of indenting arguments to operations since it makes them more readable. We can do that because our `pad` helper combinator lets us use any kind of whitespace to separate language terms.

```
tt
    let x 1
    push
        tt
            let x 2
            +
                x
                pop
                    x
```

If we save the above program as `scope-test.plus`, then we should be able to run it as follows.

```
$ dotnet run scope-test.plus
Expression: ThisThat
  (Let (Var 'x', Num 1),
   ScopePush
     (ThisThat (Let (Var 'x', Num 2), Plus (Var 'x', ScopePop (Var 'x')))))
Result: Num 3
```

If you work this program out on paper, you'll see that this is what we expect. The program creates two variables with the same name `x`, but with two different values. By managing scopes, it is able to add them together, yielding the number `3`.

In the next chapter, you'll see why we did all this hard work.

Listing 3: A complete implementation for PLUSLANG++.

```
open Combinator

type Expr =
| Num of int
| Var of char
| Plus of left: Expr * right: Expr
| Let of name: Expr * e: Expr
| ThisThat of this: Expr * that: Expr
| ScopePush of e: Expr
| ScopePop of e: Expr

type Scope =
| Base
| Env of m: Map<char,Expr> * parent: Scope

// lookup variable named c in given scope
let rec lookup c s: Expr =
    match s with
    | Base ->
        failwith ("Unknown variable '" + c.ToString() + "'")
    | Env (m, parent) ->
        if (Map.containsKey c m) then
            Map.find c m
        else
            lookup c parent
```

```
// store variable named c with value v in Scope
// s, overwriting an existing value if necessary;
// returns updated Scope
let store c v s: Scope =
    match s with
    | Base -> failwith "Cannot store to base scope."
    | Env (m, parent) ->
        let m' = Map.add c v m
        Env (m', parent)

// get the parent scope of the current scope
let parentOf env: Scope =
    match env with
    | Base ->
        failwith "Cannot get parent of base scope."
    | Env (_, parent) -> parent

(* forward references for recursive parsers *)
let expr,exprImpl = recparser()
(* helpers *)
let pad p = pbetween pws0 p pws0 <!> "pad"

(* numbers *)
let num = pmany1 pdigit |>> (fun ds -> stringify ds |> int |>
    Num) <!> "num"
let numws0 = pad num <!> "numws0"

(* addition *)
let pluswS0 = pad (pchar '+') <!> "pluswS0"
let plusExpr = pseq (pright pluswS0 expr) expr Plus <!> "
    plusexpr"

(* variables *)
let var = pad pletter |>> Var <!> "var"
let letws0 = pad (pstr "let") <!> "letws0"
let letExpr = pseq (pright letws0 var) expr Let <!> "letexpr"

(* sequential composition *)
let ttws0 = pad (pstr "tt") <!> "ttws0"
let ttExpr = pseq (pright ttws0 expr) expr ThisThat <!> "ttexpr"

(* scope operations *)
let spushws0 = pad (pstr "push")
let spopws0 = pad (pstr "pop")
let scopeExpr =
    (pright spushws0 expr |>> ScopePush) <|>
    (pright spopws0 expr |>> ScopePop)

(* top-level expressions *)
exprImpl :=
    plusExpr <|>
    letExpr <|>
    ttExpr <|>
    scopeExpr <|>
    numws0 <|>
    var
let grammar = pleft expr peof <!> "grammar"

(* parsing API; call this to parse *)
let parse input : Expr option =
```

```
    match grammar (prepare input) with
    | Success(ast,_) -> Some ast
    | Failure(_,_) -> None

let charFromVar e =
    match e with
    | Var c -> c
    | _ -> failwith "Expression is not a variable."

let rec eval (ast: Expr)(s: Scope): Expr * Scope =
    match ast with
    | Num n -> Num n, s
    | Plus (left, right) ->
        let r1, s1 = eval left s
        let r2, s2 = eval right s1
        match r1, r2 with
        | Num n1, Num n2 -> Num(n1 + n2), s2
        | _ -> failwith "Can only add numbers."
    | Var c ->
        lookup c s, s
    | Let (var, e) ->
        let r, s1 = eval e s
        let c = charFromVar var
        let s2 = store c r s1
        r, s2
    | ThisThat (this, that) ->
        let _, s1 = eval this s
        let r, s2 = eval that s1
        r, s2
    | ScopePush e ->
        let s1 = Env (Map.empty, s)
        eval e s1
    | ScopePop e ->
        let res,s1 = eval e s
        let parent = parentOf s1
        res,parent

[<EntryPoint>]
let main argv =
    if argv.Length <> 1 then
        printfn "Usage: dotnet run \"<file>\""
        exit 1

    // does the file exist?
    if not (System.IO.File.Exists argv[0]) then
        printfn "%s" ("File '" + argv[0] + "' does not exist.")
        exit 1

    // read file
    let input = System.IO.File.ReadAllText argv[0]

    // initialize root scope
    let env = Env(Map.empty, Base)

    match parse input with
    | Some ast ->
        printfn "Expression: %A" ast
        let result, _ = eval ast env
        printfn "Result: %A" result
    | None -> printfn "Invalid expression."
    0
```

Implementing Functions

Figure 26: From the *Café Royal Cocktail Book* by W.J. Tartling, illustrated by Frederick Carter, 1937.

THIS is a bit of a long chapter. But it's the last lesson of the semester, and it will leave you with a superpower: the knowlegde of how functions work in a programming language. Do your best to understand it. As usual, code is available at the end of the chapter for you to try out. [—ed.]

In the last chapter, *Implementing Scope*, we spent a lot of time developing a "fancy" model for scoping variables. After reading that chapter, you may have felt like you spent a lot of time thinking hard about a feature that did not seem very intuitive or natural. I am here to tell you that all that hard work was worth it, and that it was all building up to be able to talk about the subject of this chapter, functions.

Functions are not just neat or convenient in a programming language. I argue that functions are an essential element of programming because they are a *natural*, *intuitive*, and *commonplace* way for humans to think about the world. To demonstrate, I'll start with my favorite cocktail recipe.

The Last Word

$^3/_4$ oz gin
$^3/_4$ oz green chartreuse
$^3/_4$ oz maraschino liquor
$^3/_4$ oz freshly-squeezed lime juice
one cocktail cherry (like Luxardo or Amarena)

Add gin, chartreuse, maraschino liquor, and lime juice to cocktail shaker with ice and shake until chilled. Strain into a chilled coupe glass. Garnish with cherry.

What is a Function?

The recipe above has parameters, in the form of an ingredient list. It has a procedure that says what to do with those ingredients. It even has a name so that we can refer to it without having to recite the whole thing out when communicating about it. It is precise enough that anyone with even a small amount of experience with cocktails will know precisely what to do with it, and in the right hands, it will produce the same outcome predictably and repeatably. It even has convenient mathematical properties; want two cocktails? Perform the recipe twice. In essence, it has all the makings of a function.

Of course, it's not quite as rigorously mathematical as computer functions. It does not say, for instance, what a "cocktail shaker" is, how hard to shake one, how to measure out $^3/_4$ of an ounce, or even what an "ounce" is. Nevertheless, we tend to think of recipes like the above as "natural," and although many people profess to have no kitchen skills, few would say that following a recipe is well beyond their abilities. Nevertheless, recipes and functions are essentially the same idea. Whether mathematically-minded or not, humans have a natural compulsion to describe their world in terms of functions.[99]

[99] If you have not seen Prof. Tom Garrity make the same argument, give yourself a minute to appreciate his enthusiasm.

So if you've reached this point dreading the idea that learning how to implement functions on a computer is not for you, let me just say, I disagree. You are already familiar with the key components. As always, making computers do things can be exasperating. But whenever you get stuck or feel like you're spinning your wheels, rest assured, understanding how a function works on a computer is absolutely worth your time. It won't serve you well as a future software engineer, it will clarify your thinking and serve your main job, which is simply being a human.

Parts of a Function

The coarsest distinction we might make about the lives of functions on computers is the fact that their lives are divided into two parts: first

they are *defined* and then they are *evaluated*. For example, here I define a function in F# called `foo`.

```
let foo(): unit =
  printfn "Why are programmers obsessed with 'foo'?"
```

Observe that in the program above, I define a function, but I do not "use" it. Because the function is never called, it is never evaluated. We can amend the program so that a function call occurs.

```
let foo(): unit =
  printfn "Why are programmers obsessed with 'foo'?"

[<EntryPoint>]
let main args =
  foo()
  0
```

Now our function lives a full and productive life. It is both defined and used. When implementing functions, try to retain this distinction in your mind, because those two parts are implemented differently.

Another aspect of functions is that they have *formal parameters* and take *arguments*. You have likely used these words before, and you may have even said one when you meant the other. In programming language theory, formal parameters and arguments have quite distinct meanings and using them interchangeably can cause a lot of confusion.

A *formal parameter* is a *bound variable* declared at the time of a function's definition. For example, the variable `x` in the program above is a formal parameter. A formal parameter declares that a placeholder variable is valid to use throughout a function's definition.

When a function is called with a value, we call that value an *argument*.[100] In the evaluation of a function, we replace uses of that variable with uses of the variable's stored value instead. This mechanism allows us to abstract over *procedures*, which are simple sequences of code steps. Functions make the lives of programmers easier because code that does almost-the-same-thing, except for specific values, can be defined in a general way only once. Imagine how hard it would be to write an `int add(int x, int y)` function in Java if you had to write a different definition for every possible value of `x` and `y`!

When a function is called, it is almost as if we are rewriting the function's procedure, replacing formal parameters with the passed-in arguments, before evaluating the new procedure. I say "almost as if" because there are some caveats about how this should work. To figure out how it should work, we return briefly to the lambda calculus.

[100] Sometimes arguments are referred to as *actual arguments* or *actual parameters* because even good programmers get confused sometimes. Embarrassingly, "actual parameter" even once made an appearance in drafts of the C++ language definition, and had to be removed. Break the cycle and try to stick to the correct, non-confusing words.

How Should a Function Behave?

You may remember way back in the chapter on *Introduction to the Lambda Calculus, Part 1* that I claimed that the lambda calculus serves as the theoretical model for real programming languages. What does that statement mean? The idea is that whenever we are in doubt, we design our programming languages to do the same thing that the theory does. We do this because the model is usually simpler, clearer, and easier to reason about than an actual programming language. Actual programming languages are constructed things, and like all constructed things, engineering tradeoffs need to be made to accommodate certain realities.[101] The simplicity of a model allows us to predict what a well-behaved language should do.

[101] For example, real computers have finite memory and take measurable time to do things.

A good analogy is the set of "laws" that model motion in physics. You may have used these formulas before. For example, suppose we throw a ball and want to know how long that ball will take to return to its initial height. A handy dandy formula can be derived to answer this question:

$$t = 2\frac{V_0}{g}\sin\theta$$

where V_0 is a vector representing the initial velocity of the object, θ is the angle that the ball is thrown, g is gravitational acceleration on the surface of the Earth, and t is time in seconds. This formula literally predicts the future.[102] You can determine where a ball will go without ever throwing an actual ball.

[102] I have always felt that this fact is underappreciated.

Still, does this formula model the entire complexity of real life? It does not. For example, in the real world, we have frictional forces from wind. This matters at baseball games on windy days. To a lesser extent, gravitational force is also not a constant on Earth.[103] Nevertheless, the above formula is a good model because it applies in the vast majority of circumstances that normal humans care about.

[103] One of the first observations of systematic error due to local gravity was when the surveyors Charles Mason and Jeremiah Dixon were hired in 1763 to survey the borders of Pennsylvania and Maryland. That border, now known as the Mason-Dixon line, exhibited systematic error that was hypothesized to be caused by the gravitational influence of the Allegheny Mountains. Nevil Maskelyne, whose instruments Mason and Dixon used, later confirmed the existence of local gravity in 1774 in what is now called the "Schiehallion experiment."

In programming languages, we use the lambda calculus to model programming language behavior. Like physics, there are many refinements to this theory to account for the real world of programming on a computer, but the lambda calculus is, to a first approximation, the right model for functions.[104] So how might we model function definition and function calls in the lambda calculus?

[104] For example, languages with types are much closer to a model called the *simply typed lambda calculus*, and languages whose types can be generic are closer to a model called *System F*. Models remain an active area of research in programming languages.

Function definition is easy to model, because it is built right in to the lambda calculus. We define a function like so.

```
λx.[whatever]
```

That's right, lambda abstractions are just function definitions. You might remember that lambda abstractions are missing some things we consider essential in a real programming language, like function names.

We will return to this point shortly.

Function calls are also easy to model. Suppose we have a function, `a`, and an argument `b`.

```
ab
```

We can determine the effect of applying `b` to the function `a` by beta reduction. For example, supposing `a` is the lambda expression `λx.xx`, then `ab` is evaluated as follows.

`(λx.xx)b`	given
`([b/x] xx)`	beta reduction step 1: outer replacement
`(bb)`	beta reduction step 2: inner replacement
`bb`	eliminate parens; done

However, you might remember that there is a catch to this procedure. If an inner lambda redefines an argument, we must be careful. Suppose we have the following expression instead.

`(λx.λx.xx)b`	given
`([b/x] λx.xx)`	beta reduction step 1: outer replacement
`(λx.xx)`	beta reduction step 2 blocked by inner lambda abstraction with same name `x`
`λx.xx`	eliminate parens; done

We stop our evaluation before the inner lambda expression because it redefines `x`.

Here's another special case, concerning free variables. Consider the following expression.

```
(λx.λy.xy)y
```

Do you see any free variables in that expression? Recall that a *free variable* is any variable that is not bound in a lambda abstraction. Indeed, there is one: the last `y`. It it outside of any abstraction over `y` and so it is free. We need to take special steps *not to do the following* when we reduce this expression.

`(λx.λy.xy)y`	given
`([y/x] λy.xy)`	beta reduction step 1: outer replacement
`(λy.yy)`	beta reduction step 2: inner replacement
`λy.yy`	eliminate parens and... wrong!

The reason this is incorrect is that the `y` in the abstraction `λy.xy` and the rightmost `y` are totally different variables. The first is a bound variable whose value is determined by the abstraction. The second is a free variable. When we make the above mistake, it appears that we can conclude that the free `y` could be replaced were we to beta reduce the above expression with another argument. In lambda calculus parlance, the free `y` was mistakenly "captured" by the abstraction over `y`.

We should have performed the following reduction instead.

`(λx.λy.xy)y`	given
`(λx.λz.xz)y`	alpha reduce inner `y` with `z`
`([y/x] λz.xz)`	beta reduction step 1: outer replacement
`(λz.yz)`	beta reduction step 2: inner replacement
`λz.yz`	eliminate parens; done

This special handling of variable redefinition is what we call *capture-avoiding substitution*, and it forms the theoretical basis for our notion of scope. Despite appearances, this is not merely an academic problem. It occurs in any programming language that allows variables to be redefined. Recall our problem with global scope in the chapter on *Implementing Variables*. Here's a simpler version of the same program written in F#.

```
let countDown(): unit =
  for i = 5 downto 0 do
    printf "%d " i

[<EntryPoint>]
let main args =
  let i = 7
  countDown()
  printfn "%d" i
  0
```

Observe that the `main` function calls another function `countDown` and that this "inner" function redefines `i`. These are different variables that just happen to have the same name, so we must be careful to treat them differently.

Naming Things

Let's turn our attention now to naming things. Throughout the semester, I have postponed discussion of naming things. I made you learn the lambda calculus, and I claimed that "names don't really matter" when reasoning about that nature of computation. This is true, but I admit, it's also frustrating. Don't we use names all the time? Aren't they convenient and natural too? Don't we want them?

In fact, I agree with all of the above. The question is not *whether* we should name things, but *how*. I postponed this discussion until we had the right vocabulary, namely, some notion of scope. Although there are many approaches to naming, the predominant scheme for naming things in a programming language is called *lexical scope*.

In languages with lexical scope, the meaning of a variable is deter-

mined by the closest preceding definition in the program text.[105] For example, in the `countDown` function above, we know that the `i` used in the `printf` statement is the one defined in the `for` expression, not the one defined in the `main` function, because the `i` in the `for` expression is the closet definition.

F# uses clever, pretty syntax so that you don't have to explicitly think about scope like in PLUSLANG++. It uses lexical scope. Nevertheless, scope and variables are distinct ideas, and that's easier to see if we look at the language that inspired F#, Standard ML. Consider the following F# program, which prints the variable `i`.

```
let i = 7
printfn "%d" i
```

For the expression `printfn "%d" i` to be logically valid, we know that it must fall within the scope of the variable `i`, otherwise `i` would be undefined. In Standard ML, this code is written in a slightly different way.[106]

```
let
  val i = 7
in
  print ((Int.toString 5) ^ "\n");
end;
```

In Standard ML, the `let val <var> = <expr_1> in <expr_2> end` syntax means that anything defined between the `let` and `in` can be used in the expressions between the `in` and `end`. Variables are defined using `val`. Inside of the `in` and the `end`, those definitions are valid; outside, they are not. In short, `let` defines the scope of a `val`.

Whether you write the F# version or the Standard ML version, either syntactic form is converted to exactly the same lambda calculus expression.[107] So what is the magical lambda expression that captures this behavior? It's surprisingly simple. For simplicity, suppose you have an expression of the form `let val <var> = <expr_1> in <expr_2> end`.[108] This expression can then be converted into a lambda expression using the following formula.[109]

$$\texttt{let val } z \texttt{ = } U \texttt{ in } V \texttt{ end} \equiv (\lambda z.V)U$$

Strikingly, we use lambda abstractions—i.e., functions—to model variables and their scopes. In an important sense, that is all a lambda abstraction does. The lambda calculus provides rules that ensure that variable use is logical, consistent, and most importantly, not surprising.

If we let z be `i`, U be `7` and V be `printfn "%d" i`, then we can see how the code should be evaluated.

[105] You should be aware that a number of popular languages failed to carefully consider the rules for naming things when they were designed. In these languages, what a name means in a given context can be confusing or unclear. Some examples are Javascript and R. If you've tried to use these languages and felt confused, it wasn't you.

[106] Take my word that the `print` expression does the same thing as the F# one.

[107] You might be wondering how F# knows where the `end` is when it is not explicitly written. The short answer is that it uses whitespace to delimit scope, like Python.

[108] Pleasant syntax like F#'s version of `let` is often referred to as "syntactic sugar," because having it is sweet but not logically necessary. Programming languages sometimes handle pleasant syntax by first applying a preprocessing step that rewrites user programs into a form that uses alternative, less pleasant, but easier-to-parse syntax, a process called *desugaring*.

[109] The $\equiv$ sign means "is equivalent to" and it means that syntax written on the left side has exactly the same meaning as syntax written on the right side.

`(λi.(printfn "%d" i)) 7`	given
`((printfn "%d" 7))`	beta reduce 7 for `i`
`(printfn "%d" 7)`	eliminate parens
...	etc

I leave the last steps blank because we would do the same procedure recursively for `printfn`, which is also defined by a lambda abstraction somewhere.[110] That abstraction would be substituted in and evaluation would continue.

[110] Actually, two lambda abstractions, because the `printfn` call above takes two arguments. Recall that we use curried abstractions like `λx.λy...` to model multiple arguments.

Implementation

Let's extend PLUSLANG++ one more time to support function definition and function calls. Whenever we are in doubt, we will turn to the lambda calculus as a guide.

Let's add two more syntactic rules, for function definitions and function calls. `fun <parlist> <expr>` defines the curried lambda abstraction, `λ<var>`$_1$`.λ<var>`$_2$`.λ<var>`$_{...}$`.<expr>`. For example, the expression `fun [x y] + x y` is a function definition. `call <expr> <arglist>` defines curried application, `<expr> <expr>`$_1$ `<expr>`$_2$ `<expr>`$_{...}$ where the first `<expr>` must be a function definition. For example, the expression `call fun [x y] + x y [1 2]` calls a function definition.

Although it may seem right now like adding lambda abstraction and application are not the obvious way to add function definition and calls to a language, as you will see, it really is the simplest way. For example, since named functions are just ordinary variables that happen to map to function definitions, our new language can reuse the existing variable implementation without modification. That allows us to write programs like the following, which might even be starting to seem like a pleasant way of writing code.

```
tt
  let f
    fun [x y] + x y
  call f [1 2]
```

Here's our updated grammar.

```
<expr>    ::= + <expr> <expr>
            | let <var> <expr>
            | tt <expr> <expr>
            | fun <parlist> <expr>
            | call <arglist> <expr>
            | <var>
            | <num>
<num>     ::= <digit>+
<digit>   ::= 0|1|2|3|4|5|6|7|8|9
<var>     ::= α ∈ a...z
<parlist> ::= [ <var>* ]
<arglist> ::= [ <expr>* ]
```

Function Definition in PLUSLANG$^{\lambda}$

To support function definition, we will need a new `Expr` case. A function definition is essentially a parameter list and a body expression.

```
| FunDef of pars: Expr list * body: Expr
```

To parse function definitions, we will use a small variation on the pattern we've used throughout PLUSLANG to support prefix operators. The main difference is that we want to be able to handle parameter lists like `[x y z]`. Essentially, we need to be able to parse any number of variables, separated by spaces, in between square brackets. Let's break this into pieces.

We can use `pmany0` to parse any number of something. But we need more than that; we want it to be the case that whenever there is more than one of something, those somethings are separated by a separator. Let's define a new combinator to help us.

```
let pmany0sep p sep = pmany0 (pleft p sep <|> p)
```

This parser will look for `p` followed by `sep`; if it doesn't find that, it looks for just `p`. It tries to repeat this as many times as possible. Suppose we have the string `x␣y␣z`, where ␣ is a space character. Then `pmany0sep var pws1` will parse by first consuming `x␣`, then `y␣`, and then `z`.

Now, all we need to do is to parse that space-separated list of variables only when it is between square brackets. I add a little padding as well in case the user wants to write something like `[ a b c ]`.

```
let parlist =
  pbetween
    (pchar '[')
    (pad (pmany0sep var (pchar ' ')))
    (pchar ']')
```

Now, we can apply our prefix operator pattern for the rest of a function definition,

```
let funws0 = pad (pstr "fun")
let funDefExpr = pseq (pright funws0 parlist) expr FunDef
```

and then we add `funDefExpr` to `expr`.

You might be surprised how easy it is to implement evaluation for a function definition. Suppose I have the following program. What should the program do when I run it?

```
fun [x y] + x y
```

The answer is: nothing. As in an ordinary language, like Java, if we define a function but never use it, nothing should happen. It's pretty easy to make nothing happen; let's add a case to `eval` that simply returns the definition itself.[111]

[111] You can use _ in a pattern match to ignore matches.

```
let rec eval (ast: Expr)(s: Scope): Expr * Scope =
    match ast with
    ...
    | FunDef (_, _) -> ast, s
    ...
```

Amazingly, that's it for function definition.

Function Calls in PlusLang$^{\lambda}$

Function calls are a little more complicated. However, this is the last feature we will be adding to PlusLang$^{\lambda}$, and it's a fancy feature, so a little complexity is warranted. Don't worry, as before, we will walk through it step by step.

Let's start by adding a case to `Expr`. A function call applies an argument list to a function definition.

```
| FunCall of fundef: Expr * args: Expr list
```

Parsing a function call is much like parsing a function definition, since we are going to use the `[]` list parser we came up with before. It's almost exactly the same pattern.

```
let arglist = pbetween (pchar '[') (pad (pmany0sep expr (pchar ' '))) (pchar ']')
let callws0 = pad (pstr "call")
let funCallExpr = pseq (pright callws0 expr) arglist FunCall
```

And, of course, add `funCallExpr` to `expr`.

Evaluating function calls is where things get a little tricky, because we want to be able to support "calling a lambda" like,

```
call fun [x y] + x y [1 2]
```

but also "calling a function by name" like,

```
tt
  let f
    fun [x y] + x y
  call f [1 2]
```

Let's consider calling a definition directly, since it is the simplest part. What does a lambda expression like `(λx.x)y` "do"? First, it defines a new variable called `x`. Then, it assigns the argument, `y` to the variable `x`. Then it evaluates the body, in this example, `x`. In fact, PLUSLANG++ could already do all of those operations, couldn't it? Let p be a parameter, v be an argument, and e be the body in the lambda expression $(\lambda p.e)v$. Doesn't the following program essentially do the same thing?

```
push
  tt
    let p v
    pop e
```

The above program pushes a new scope, binds v to p, executes e in the new scope, then restores the old scope. If you think about it, all we really need to do is to rewrite any function call we get into something like the above. But how might we handle functions with multiple arguments? Again, the lambda calculus provides some guidance. Isn't a function of multiple arguments just a curried lambda expression? In other words, isn't `fun [x y] + x y` basically just `λx.λy.+ x y`? So we just need to "nest" the pattern.

```
push
  tt
    let p_1 v_1
    pop
      push
        tt
          let p_2 v_2
          pop e
```

Let's make a `lambda` helper method to make it easy to generate our PlusLang$^{\lambda}$ function call pattern. Think of this as a kind of template that lets us plug in the parts of a function. Because the evaluator has direct access to the `Expr` type, we can forego parsing and just create an AST fragment directly.

```
let lambda p v e =
    ScopePush(
        ThisThat(
            Let(p, v),
            ScopePop(e)))
```

Now we have the pieces we need to be able to evaluate a function call. Remember, there are two cases for function definitions: a bare lambda expression and a variable. To distinguish between the two cases, we will pattern-match on the function definition parameter. We can also return an error when somebody tries to perform a function call with something that is not a function.

```
| FunCall (f, args) ->
    match f with
    | Var _ ->
        ...
    | FunDef (pars, bod) ->
        ...
    | _ -> failwith "Can only call functions."
```

The first thing we should probably do is to make sure that the user called the function with the correct number of parameters.

```
| FunDef (pars, bod) ->
    if List.length pars <> List.length args then
        failwith "Number of arguments must match number of parameters."
    ...
```

If we look at our `lambda` helper definition above, it would be conve-

nient to be able to pair a parameter with its argument so that we can call our lambda template helper correctly.[112] What we want to do is to recursively call the `lambda` helper for every pair of parameter and argument. If you do this exercise out on paper, one thing you'll discover is that we need to generate our code from the inside out. The reason is that we want the *first* argument given in a function call to be applied to the outermost variable in a lambda abstraction. In other words, we want the parts of a function definition to be assembled as follows.

[112] I have a lot of practice writing little code generators like this so you might be surprised to learn that I still find code generation confusing. Fortunately, we can work through many problems like this by hand. I came up with this section by working through it on paper. If you're confused, try working through this on paper.

`call fun [`$p_1 \ldots p_n$`] <expr> [`$a_1 \ldots a_n$`]`$\equiv$`(`$\lambda p_1 \ldots . \lambda p_n$`.<expr>)`$a_1 \ldots a_n$

Calling the template helper `lambda` p_3 a_3 `<expr>` generates code that evaluates the expression

`(`λp_3`.<expr>)`a_3

Nesting two calls, like `lambda` p_2 a_2 `(lambda` p_3 a_3 `<expr>)` generates

`(`$\lambda p_2 . \lambda p_3$`.<expr>)`$a_2\ a_3$

Nesting three calls, like `lambda` p_1 a_1 `(lambda` p_2 a_2 `(lambda` p_3 a_3 `<expr>))` generates

`(`$\lambda p_1 . \lambda p_2 . \lambda p_3$`.<expr>)`$a_1\ a_2\ a_3$

There are many ways to approach this process, but whenever we need to call a function recursively for every argument in a list, `fold` is a good approach. The code below `zips` parameters and arguments together, reverses them so that we process them inside-out, then folds them together into a big lambda expression.

```
| FunDef (pars, bod) ->
    if List.length pars <> List.length args then
        failwith "Number of arguments must match number of parameters."
    let pa = List.zip pars args |> List.rev
    let f =
        pa |>
        List.fold (fun acc (par,arg) -> lambda par arg acc) bod
    ...
```

The last thing to do is to evaluate the code we just generated.

```
| FunDef (pars, bod) ->
    if List.length pars <> List.length args then
        failwith "Number of arguments must match number of parameters."
    let pa = List.zip pars args |> List.rev
    let f =
        pa |>
        List.fold (fun acc (par,arg) -> lambda par arg acc) bod
    eval f s
```

Now, how do we handle calling a function by name? We've already done most of the work. We simply need to lookup the value of the variable (which is a function definition), make a new function call, then call it.

```
| FunCall (f, args) ->
    match f with
    | Var _ ->
        let fundef, _ = eval f s // lookup function definition
        let fcall = FunCall(fundef, args) // replace call to var with call to def
        eval fcall s // evaluate function call
    | FunDef (pars, bod) ->
        if List.length pars <> List.length args then
            failwith "Number of arguments must match number of parameters."
        let pa = List.zip pars args |> List.rev
        let f =
            pa |>
            List.fold (fun acc (par,arg) -> lambda par arg acc) bod
        eval f s
    | _ -> failwith "Can only call functions."
```

At long last, that's it!

Does it Work?

To test our implementation, we can define some lambda expressions and see if evaluating them does the right thing.

For example, I'm pretty sure that (`(`$lx.ly$`.+ x y) 1 2` will evaluate to `3`. Let's write the corresponding PLUSLANG$^{\lambda}$ program and see whether it does. Here's our program.

```
tt
  let
    f
    fun
      [x y]
      +
        x
        y
  call
    f
    [1 2]
```

And here's us running it.

```
$ dotnet run adder-function.plus
Expression: ThisThat
  (Let (Var 'f', FunDef ([Var 'x'; Var 'y'], Plus (Var 'x', Var 'y'))),
   FunCall (Var 'f', [Num 1; Num 2]))
Result: Num 3
```

This is very satisfying. It's a real programming language now!

How Far Can You Go?

My favorite litmus test for a working language is a simple one: `(λx.x)(λx.x)`.
Does that work?

```
call
  fun [x] x
  [fun [x] x]
```

Running it,

```
$ dotnet run litmus.plus
Expression: FunCall (FunDef ([Var 'x'], Var 'x'), [FunDef ([Var 'x'], Var 'x')])
Result: FunDef ([Var 'x'], Var 'x')
```

Wow! The complete code for PLUSLANG$^{\lambda}$ can be found in Listing 4. Congratulations on making it through all of this.

If you systematically test this implementation, you will find that not all PLUSLANG$^{\lambda}$ programs do what the lambda calculus predicts. Such cases are common in real programming languages. It is difficult to meet all of the engineering challenges while remaining faithful to programming language theory. For example, programming language designers frequently sacrifice theoretical purity for speed. If you're looking for a challenge, see if you can discover one or more cases in PLUSLANG$^{\lambda}$.

Listing 4: A complete implementation for PlusLang$^{\lambda}$.

```
open Combinator

type Expr =
| Num of int
| Var of char
| Plus of left: Expr * right: Expr
| Let of name: Expr * e: Expr
| ThisThat of this: Expr * that: Expr
| ScopePush of e: Expr
| ScopePop of e: Expr
| FunDef of pars: Expr list * body: Expr
| FunCall of fundef: Expr * args: Expr list

type Scope =
| Base
| Env of m: Map<char,Expr> * parent: Scope

// lookup variable named c in given scope
let rec lookup c s: Expr =
    match s with
    | Base ->
        failwith ("Unknown variable '" + c.ToString() + "'")
    | Env (m, parent) ->
        if (Map.containsKey c m) then
            Map.find c m
        else
            lookup c parent

// store variable named c with value v in Scope
// s, overwriting an existing value if necessary;
// returns updated Scope
let store c v s: Scope =
    match s with
    | Base -> failwith "Cannot store to base scope."
    | Env (m, parent) ->
        let m' = Map.add c v m
        Env (m', parent)

// get the parent scope of the current scope
let parentOf env: Scope =
    match env with
    | Base ->
        failwith "Cannot get parent of base scope."
    | Env (_, parent) -> parent

(* forward references for recursive parsers *)
let expr,exprImpl = recparser()
(* helpers *)
let pad p = pbetween pws0 p pws0
let pmany0sep p sep = pmany0 (pleft p sep <|> p)

(* numbers *)
let num = pmany1 pdigit |>> (fun ds -> stringify ds |> int |>
    Num)
let numws0 = pad num

(* addition *)
let plusws0 = pad (pchar '+')
let plusExpr = pseq (pright plusws0 expr) expr Plus
```

```
(* variables *)
let var = pad pletter |>> Var
let letws0 = pad (pstr "let")
let letExpr = pseq (pright letws0 var) expr Let

(* sequential composition *)
let ttws0 = pad (pstr "tt")
let ttExpr = pseq (pright ttws0 expr) expr ThisThat

(* scope operations *)
let spushws0 = pad (pstr "push")
let spopws0 = pad (pstr "pop")
let scopeExpr =
    (pright spushws0 expr |>> ScopePush) <|>
    (pright spopws0 expr |>> ScopePop)

(* function definition *)
let funws0 = pad (pstr "fun")
let parlist = pbetween (pchar '[') (pad (pmany0sep var (pchar '
    '))) (pchar ']')
let funDefExpr = pseq (pright funws0 parlist) expr FunDef

(* function call *)
let callws0 = pad (pstr "call")
let arglist = pbetween (pchar '[') (pad (pmany0sep expr (pchar '
     '))) (pchar ']')
let funCallExpr = pseq (pright callws0 expr) arglist FunCall

(* top-level expressions *)
exprImpl :=
    plusExpr <|>
    letExpr <|>
    ttExpr <|>
    scopeExpr <|>
    funDefExpr <|>
    funCallExpr <|>
    numws0 <|>
    var
let grammar = pleft expr peof

let parse input : Expr option =
    match grammar (prepare input) with
    | Success(ast,_) -> Some ast
    | Failure(_,_) -> None

let charFromVar e =
    match e with
    | Var c -> c
    | _ -> failwith "Expression is not a variable."

(* generates a "function gadget" *)
let lambda p v e =
    ScopePush(
        ThisThat(
            Let(p, v),
            ScopePop(e)))

let rec eval (ast: Expr)(s: Scope): Expr * Scope =
    match ast with
    | Num n -> Num n, s
    | Plus (left, right) ->
```

```
        let r1, s1 = eval left s
        let r2, s2 = eval right s1
        match r1, r2 with
        | Num n1, Num n2 -> Num(n1 + n2), s2
        | _ -> failwith "Can only add numbers."
    | Var c ->
        lookup c s, s
    | Let (var, e) ->
        let r, s1 = eval e s
        let c = charFromVar var
        let s2 = store c r s1
        r, s2
    | ThisThat (this, that) ->
        let _, s1 = eval this s
        let r, s2 = eval that s1
        r, s2
    | ScopePush e ->
        let s1 = Env (Map.empty, s)
        eval e s1
    | ScopePop e ->
        let res,s1 = eval e s
        let parent = parentOf s1
        res,parent
    | FunDef (_, _) -> ast, s // do nothing
    | FunCall (f, args) ->
        match f with
        | Var _ ->
            let fundef, _ = eval f s // lookup function
                definition
            let fcall = FunCall(fundef, args) // replace call to
                 var with call to def
            eval fcall s // evaluate function call
        | FunDef (pars, bod) ->
            if List.length pars <> List.length args then
                failwith "Number of arguments must match number
                    of parameters."
            let pa = List.zip pars args |> List.rev
            let f =
                pa |>
                List.fold (fun acc (par,arg) -> lambda par arg
                    acc) bod
            eval f s
        | _ -> failwith "Can only call functions."

[<EntryPoint>]
let main argv =
    if argv.Length <> 1 then
        printfn "Usage: dotnet run \"<file>\""
        exit 1

    // does the file exist?
    if not (System.IO.File.Exists argv[0]) then
        printfn "%s" ("File '" + argv[0] + "' does not exist.")
        exit 1

    // read file
    let input = System.IO.File.ReadAllText argv[0]

    // initialize root scope
    let env = Env(Map.empty, Base)
```

```
match parse input with
| Some ast ->
    printfn "Expression: %A" ast
    let result, _ = eval ast env
    printfn "Result: %A" result
| None -> printfn "Invalid expression."
0
```

The Rise of Worse is Better

By Richard P. Gabriel, Lucid, Inc. An excerpt from "Lisp: Good News, Bad News, How to Win Big." (1989)

> It is worth mentioning before you read this article that UNIX and UNIX-like derivative operating systems (like Linux) are, in 2024, the most popular on the planet. Virtually every smartphone, smart TV, or smart appliance runs either Android (a form of Linux) or iOS (a direct descendent of the original AT&T UNIX). The MACOS is a hybrid of AT&T UNIX and FreeBSD UNIX. Even Windows users can now download and install an official Microsoft-maintained UNIX environment called *Windows Subsystem for Linux*. History shows us that the *New Jersey approach* has convincingly beaten the *MIT approach*. [—ed.]

I and just about every designer of Common Lisp and CLOS has had extreme exposure to the MIT/Stanford style of design. The essence of this style can be captured by the phrase *the right thing*. To such a designer it is important to get all of the following characteristics right:

- Simplicity—the design must be simple, both in implementation and interface. It is more important for the interface to be simple than the implementation.
- Correctness—the design must be correct in all observable aspects. Incorrectness is simply not allowed.
- Consistency—the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.
- Completeness—the design must cover as many important situations as is practical. All reasonably expected cases must be covered. Simplicity is not allowed to overly reduce completeness.

I believe most people would agree that these are good characteristics. I will call the use of this philosophy of design the *MIT approach*. Common Lisp (with CLOS) and Scheme represent the MIT approach to design and implementation.

The worse-is-better philosophy is only slightly different:

- Simplicity—the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.
- Correctness—the design must be correct in all observable aspects. It is slightly better to be simple than correct.
- Consistency—the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
- Completeness—the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must be sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.

Early Unix and C are examples of the use of this school of design, and I will call the use of this design strategy the *New Jersey approach*. I have intentionally caricatured the worse-is-better philosophy to convince you that it is obviously a bad philosophy and that the New Jersey approach is a bad approach.

However, I believe that worse-is-better, even in its strawman form, has better survival characteristics than the-right-thing, and that the New Jersey approach when used for software is a better approach than the MIT approach.

Let me start out by retelling a story that shows that the MIT/New-Jersey distinction is valid and that proponents of each philosophy actually believe their philosophy is better.

Two famous people, one from MIT and another from Berkeley (but working on Unix) once met to discuss operating system issues. The person from MIT was knowledgeable about ITS (the MIT AI Lab operating system) and had been reading the Unix sources. He was interested in how Unix solved the *PC loser-ing problem*. The PC loser-ing problem occurs when a user program invokes a system routine to perform a lengthy operation that might have significant state, such as IO buffers. If an interrupt occurs during the operation, the state of the user program must be saved. Because the invocation of the system routine is usually a single instruction, the PC[113] of the user program does not adequately capture the state of the process. The system routine must either back out or press forward. The right thing is to back out and restore the user program PC to the instruction that invoked the system routine so that resumption of the user program after the interrupt, for example, re-enters the system routine. It is called *PC loser-ing* because the PC is being coerced into *loser mode*, where *loser* is the affectionate name for *user* at MIT.

The MIT guy did not see any code that handled this case and asked the New Jersey guy how the problem was handled. The New Jersey guy said that the Unix folks were aware of the problem, but the solution was for the system routine to always finish, but sometimes an error code would be returned that signaled that the system routine had failed to complete its action. A correct user program, then, had to check the error code to determine whether to simply try the system routine again. The MIT guy did not like this solution because it was not the right thing.

The New Jersey guy said that the Unix solution was right because the design philosophy of Unix was simplicity and that the right thing was too complex. Besides, programmers could easily insert this extra test and loop. The MIT guy pointed out that the implementation was simple but the interface to the functionality was complex. The New Jersey guy said that the right tradeoff has been selected in Unix—namely, implementation simplicity was more important than interface simplicity.

The MIT guy then muttered that sometimes it takes a tough man to make a tender chicken, but the New Jersey guy didn't understand (I'm not sure I do either).

Now I want to argue that worse-is-better is better. C is a programming language designed for writing Unix, and it was designed using the New Jersey approach. C is therefore a language for which it is easy to write a decent compiler, and it requires the programmer to write text that is easy for the compiler to interpret. Some have called C a fancy assembly language. Both early Unix and C compilers had simple structures, are easy to port, require few machine resources to run, and provide about 50%-80% of what you want from an operating system and programming language.

Half the computers that exist at any point are worse than the median (smaller or slower). Unix and C work fine on them. The worse-is-better philosophy means that implementation simplicity has highest priority, which means Unix and C are easy to port on such machines. Therefore, one expects that if the 50% functionality Unix

[113] The *program counter*, a variable that keeps track of which instruction a program is on. On Intel machines, this value is stored in either the `eip` or `rip` registers, depending on whether the machine is 32-bit or 64-bit. [—ed.]

and C support is satisfactory, they will start to appear everywhere. And they have, haven't they?

Unix and C are the ultimate computer viruses.

A further benefit of the worse-is-better philosophy is that the programmer is conditioned to sacrifice some safety, convenience, and hassle to get good performance and modest resource use. Programs written using the New Jersey approach will work well both in small machines and large ones, and the code will be portable because it is written on top of a virus.

It is important to remember that the initial virus has to be basically good. If so, the viral spread is assured as long as it is portable. Once the virus has spread, there will be pressure to improve it, possibly by increasing its functionality closer to 90%, but users have already been conditioned to accept worse than the right thing. Therefore, the worse-is-better software first will gain acceptance, second will condition its users to expect less, and third will be improved to a point that is almost the right thing. In concrete terms, even though Lisp compilers in 1987 were about as good as C compilers, there are many more compiler experts who want to make C compilers better than want to make Lisp compilers better.

The good news is that in 1995 we will have a good operating system and programming language; the bad news is that they will be Unix and C++.

There is a final benefit to worse-is-better. Because a New Jersey language and system are not really powerful enough to build complex monolithic software, large systems must be designed to reuse components. Therefore, a tradition of integration springs up.

How does the right thing stack up? There are two basic scenarios: the *big complex system* scenario and the *diamond-like jewel scenario*.

The *big complex system* scenario goes like this:

First, the right thing needs to be designed. Then its implementation needs to be designed. Finally it is implemented. Because it is the right thing, it has nearly 100% of desired functionality, and implementation simplicity was never a concern so it takes a long time to implement. It is large and complex. It requires complex tools to use properly. The last 20% takes 80% of the effort, and so the right thing takes a long time to get out, and it only runs satisfactorily on the most sophisticated hardware.

The *diamond-like jewel* scenario goes like this:

The right thing takes forever to design, but it is quite small at every point along the way. To implement it to run fast is either impossible or beyond the capabilities of most implementors.

The two scenarios correspond to Common Lisp and Scheme.

The first scenario is also the scenario for classic artificial intelligence software.

The right thing is frequently a monolithic piece of software, but for no reason other than that the right thing is often designed monolithically. That is, this characteristic is a happenstance.

The lesson to be learned from this is that it is often undesirable to go for the right thing first. It is better to get half of the right thing available so that it spreads like a virus. Once people are hooked on it, take the time to improve it to 90% of the right thing.

A wrong lesson is to take the parable literally and to conclude that C is the right vehicle for AI software. The 50% solution has to be basically right, and in this case it isn't.

But, one can conclude only that the Lisp community needs to seriously rethink its position on Lisp design.

Appendix A: Original SML Specification

A PROPOSAL FOR STANDARD ML (TENTATIVE)

Robin Milner, April '83.

1. Introduction

The language proposed here — called here "Standard ML" but a better name may be found — is not supposed to be novel. Its aims are

(i) to remove some redundancies and bad choices in the original design of ML;

(ii) to "round out" ML in one particular respect — namely the use of patterns in parameter passing not only for the standard data constructions (pairing, lists) but also for constructions which are user defined;

(iii) to make sure that just enough input/output is standardised to allow serious work to be done (the user should be able to define enough higher-level i/o functions within the language that he feels no pressing need to extend it);

(iv) thereby to determine as clearly as possible — for the benefit of the user community — what is guaranteed in ML.

This standardisation is conservative in many respects: little attention paid to wide programming environment issues (editing etc); no lazy evaluation; no attempt to generalise escape trapping to allow other than tokens as escape values; no use of a more general typechecking scheme to allow polymorphic assignments (or indeed to allow other-than-token escape values); no use of

record and variant types à la Cardelli ; no attempt to introduce the idea of working in 'persistent' (and possibly updateable) environments.

None of these things is considered wrong : far from it. But a well-rounded language seems to emerge without them ; this means that future extensions can be examined against a firm background. It should then be easier to see what limitations are, or are not, inherent in the ML kind of functional language.

The main inspirations towards this standardisation are from Luca Cardelli — many of whose ideas are adopted — and from HOPE (Burstall et al) which, in my view, provides the most natural rounding-out of ML w.r.t. data types and data parameters.

A word about environment operations : as Standard ML develops, it emerges that Luca's important environment operator "enc" (such that in "dec1 enc dec2" dec1 exports into dec2 and both dec1 and dec2 export from the whole) gets entirely conflated with the semicolon — which has always had the meaning of enc when it occurs between top-level declarations. With one other minor shift, it then turns out that every top level command sequence is just a single declaration ! This has a pleasant unifying effect — for example, external ML files can be imported either as global or as local declarations ; the latter has important uses.

2. Bare ML

We first give a bare version of Standard ML which omits (i) Convenient alternative syntactic forms, (ii) Infixes, (iii) References and assignment, and — quite importantly — (iv) all standard Types.

A strong point in favour of the data declarations (inherited from HOPE) is that all our standard types — unit, bool, int and token — can in principle be defined. Of course, they will probably be implemented in a special way. But, by omitting them to begin with, we show how the language is largely independent of them. For example, it is only later (via typechecking constraints) that escape values are required to be tokens; a different extension of Bare ML could make a different choice.

Possibly the only innovation in Bare ML is the function abstraction form

fun v1.exp1 | ... | vn.expn

which generalises the familiar lambda abstraction "λv.exp". When applied, this "function" matches its argument to each construct vi in left-to-right order until a match succeeds (which binds any variables in vi to components of the argument value) then evaluates the corresponding expi. If no match is found, the application escapes with a determined value (which, in the standard language, will be the token `failmatch`). This construct — which is something like a lunchtime proposal by Alan Mycroft (though he may disagree!) — cannot nicely be defined in terms of elementary lambda-abstraction by iterated escape trapping,

Since it is vital to distinguish – say – a failure to match v1 from an escape generated by evaluation of exp1. (This latter will escape from the entire application.) Note that n=1 gives ordinary function abstraction, though still with varstruct matching.

The elementary syntax classes are given in Table 1. For identifiers, we follow Luca Cardelli more or less. But it seems robust to exclude reserved words (including certain symbol sequences) from being identifiers. Since data constructors can appear in varstructs, they cannot be re-used as bound variables within the scope of the data declaration which introduced them. We seize the opportunity of discarding "*", "**", etc as type variables, and propose 'a, 'b, ... pronounced as Greek letters.

The syntax of Bare ML is in Table 2. Note that no construct is mentioned which (like conditionals presuppose bool) presupposes any standard type. We discuss the constructions class by class.

(1) Expressions: In application "exp1 exp2" no order of evaluation is assumed. In original ML, exp1 was evaluated (to a function) before exp2. In Cardelli's VAX ML, apparently exp2 is evaluated first – and this seems to work better in his abstract machine, making function application very efficient (if I understand him right). Of course the order of evaluation chosen in an implementation can be detected by escape-trapping – and this could be exploited in multiple applications like "f exp1 ... expn"; but it still seems worth giving the implementor freedom. Likewise in "exp1,exp2" no order of evaluation is assumed.

In "escape exp" exp will (in the standard language) be token-valued. "escape" seems better than "fail". In "exp trap v1.exp1|..|vn.expn" the vi will (in Standard ML) be of type token. We could have chosen "exp1 trap exp2",

where exp2 is of type "token → ..." ; but then it is irksome to have to specify how escapes in evaluating exp2 are treated. As it is, no order of evaluation question arises — and one can always get the effect (mostly) of "exp1 trap exp2" by writing "exp1 trap t.(exp2 t)"

In "let dec in exp" dec is evaluated first. Note that this is now the only use of "let"; top-level declarations will (usually) start with "val", "data" or "abstype". This differs both from DEC10 ML and from VAX ML, but I think it is justified; see the later discussion on declarations.

We have already discussed "fun match". The only extra point to be added is that — although we cannot readily define the new form in terms of simple "λv.exp" — we can probably implement it well enough in terms of the closures of Luca's abstract machine FAM, using traps.

(2) Varstructs: The wild card "any" — matching anything — is not strictly necessary, since "()" can play this rôle in varstructs without losing any power. But this double use can be mildly confusing, while any is more or less self-explanatory.

Note that the use of user-defined constructors in varstructs gives crucial extra power to the language (Luca added varstructs for his records and variants, and here the constructors play the same rôle).

A constant c is really a constructor of unit (see Table 4 defining the data type unit, which Luca called void); but we don't want the pedantry of writing c() instead of c — in varstructs or in expressions.

A constructor is formally of one argument, but for a 'binary' constructor cons (say) the forms "cons v", "cons(v, v')", "cons any" and "cons (any, any)" are all admissible constructs — but not "cons" by itself.

(3) Types: The omission of disjoint sum is discussed in Table 4. As discussed under infixes later, we propose that all type constructors (except # and →) will be postfixed. This avoids the need for separate infix statuses for identifiers in types and in expressions, which seems of slight value.

(4) Declarations: I have aimed to give all the power of Luca's various environment operators, but with considerably more restriction on the possible forms — which I believe will lead to easier understanding. This is probably the most debatable part of Bare ML; I think the choices I have made will lead to convenient and clearly understood programming style, but then so would other choices! Here are several points

(i) "local dec in dec" plays the rôle of Luca's "inside". The prefixed keyword "local" should help in reading. It corresponds precisely to "let" in expressions (see above), but the use of a different word tells a reader that this is a declaration, not an expression.

(ii) Semicolon ";" used for sequencing plays the rôle of Luca's "enc". I chose it because (a) it underlines the fact that "enc" is associative (unlike "inside"!); (b) it has the same effect as ";" separating top-level declaration commands, and its weak binding power ensures that at top level "dec ; dec" is treated as two separate commands; (c) it seems the nicest way of writing a sequence of local declarations each dependent on the last — for example

we have the three forms

DEC10 ML	VAX ML	STANDARD ML
let z = x+y in let w = z*z in	let z = x+y and w = z*z in	let var z ← x+y ; var w ← z*z in ------

(iii) The restriction of "rec" to qualify only simultaneous bindings seems quite adequate — and avoids the need for Luca's restriction "no inside within a rec". The restriction of "rec" to qualify only whole simultaneous bindings avoids the (probably useless) possibility of "rec" within "rec", and this stratified approach probably makes an implementation easier. Sufficient use of "local ... in ..." will get round any need for "rec" to qualify part of a simultaneous binding.

(iv) I can see no real need for simultaneous bindings of different kinds, as in "data .... and var ....", so have precluded them.

(v) The keyword "var" seems appropriate to match "data" and "abstype"

(vi) I believe that the only use of an abstract binding (isomorphism) is to provide operations defined via the isomorphism, so have forced abstract type declarations to have a with part. Note that, for each type constructor "tycon" introduced in such a binding, the constructor "abstycon" is available in the with part both in expressions and in var structs; this means that "reptycon" is no longer necessary.

Finally, the locality of data and abstype declarations should be enforced (as in previous ML) by the typechecker ensuring that no value is exported from their scope whose type involves the declared type constructors. This export could be the result of an expression, or by assignment to a reference (but it needs checking whether this latter export is prevented; I believe it is).

(5) Variable Bindings : I think it's time to get rid of "=" in variable bindings, so "←" is used instead. † Perhaps we can introduce "be", "is", "are" as synonyms ?

(6) Data Bindings : It seems worth having the grammatical form e.g. "cons of 'a # 'a list" rather than "cons ('a # 'a list)" as in HOPE, or "cons : 'a # 'a list". Both the latter forms are mild puns, and the last is misleading. The choice of "|" to separate alternative forms is supposed to reverberate with BNF, and with the syntactic form match used in expressions. To choose ";" instead would be overloading the comma with too many meanings.

There is no restriction on the types occurring after of (except that all type variables used must occur on the left of "←"). Thus data types don't have to be "data" all the way down — only at the top level of construction.

(7) Abstract Bindings : These are as in DEC10 ML (except for "<=>" in place of "=", following Luca).

---

† We could have use "<=" as in HOPE, but prefer it to mean "less than or equal to" to ease the transition for PASCAL programmers!

3. Adding Infixes to BareML

First, the commands

```
Com ::= ....
      | infix id1 .... idn {k}{ass}
      | nonfix id1 ... idn
```

where
k ::= 1|2|... (precedence)
ass ::= left|right (association)

are added. These commands establish the infix status of identifiers in expressions only, not within types. Default values for k, ass are 1, left. The pairing operator "," has precedence 1; thus it binds more loosely than every predefined infix (except ":="). Infixes bind more loosely than application or type constraint, more tightly than all other expression constructs. Examples:

f a + b means (f a) + b

a + b trap m means (a + b) trap m

Second, every non-infixed occurrence of an identifier with infix status must be preceded by "infix". Infixed occurrences are only allowed in expressions and varstructs, not in Data binding constructions.

Third, all infixes must stand for functions of pairs, so

a + b means infix + (a, b)

not infix + a b

See Table 6 for predefined infixes

4. References and Equality in Standard ML

Although Luis Damas produced a subtle form of typechecking to allow polymorphic assignment to references, I propose that the Standard Language sticks to monomorphic assignment. The main reason is that the original typechecking discipline has enough pedigree (Curry, Hindley) to deserve the name "Standard" better than any other; it seems wise commit the language just this far and no further. Certainly some implementations will want to be more permissive in typechecking (not only for references: another example is in recursively defined functions), and they can declare this explicitly in their documentation. But the example below shows that the effect of polymorphic assignment can be got, in the Standard Language, at the price of passing polymorphic "assignment functions" as parameters. Also, this is in harmony with the proposed treatment of equality as (mainly) monomorphic; see the end of this section.

Thus the additions to Bare ML for references are:

(1) The type construction α ref.

(2) The function ref : $\mu \to \mu$ ref at all monotypes μ, for creating new references. In constructs, however, ref : $\alpha \to \alpha$ ref may be used polymorphically.

(3) The function infix := : μ ref # $\mu \to$ unit at all monotypes, for assignment.

Besides this, the standard contents function @ : α ref $\to \alpha$, defined by "bar @(ref x) ← x", is provided.

Example. Here is how to define α row, the type of polymorphic one dimensional arrays. Its operations are parameterised on polymorphic functions

new : $\alpha \longrightarrow \alpha$ ref

assign : (α ref * α) $\longrightarrow$ unit

They are:

newrow new : $\alpha \longrightarrow$ int $\longrightarrow \alpha$ row

("newrow new v k" returns a row of length k with value v in each cell)

assignrow assign : $\alpha \longrightarrow \alpha$ row $\longrightarrow$ int $\longrightarrow$ unit

("assignrow assign v R k" places value v in k^{th} cell of row R)

conrow : α row $\longrightarrow$ int $\longrightarrow \alpha$

("conrow R k" returns value of k^{th} cell in R)

```
abstype α row <==> α ref list
with newrow ref v k <- absrow(newlist k) where rec newlist <-
        fun 0. nil | k. ref v :: newlist(k-1)
and assignrow (infix :=) v (absrow L) k <- assignlist L k where rec assignlist <-
        fun nil. escape `outofrange`
          | c::L . fun(1. c := v  | k. assignlist L (k-1))
and conrow (absrow L) k <- conlist L k where rec conlist <-
        fun nil . escape `outofrange`
          | c::L. fun(1. !c | k. conlist L (k-1))
;
```

Note that only the underlined parameters are extra to what one could write if polymorphic assignment were allowed (just as an abstype may have to be supplied with a polymorphic equality predicate as parameter).

From this, monomorphic arrays are easily set up:

```
abstype introw <=> int row
with newintrow (n:int) <- absintrow o (newrow ref n)
and assignintrow (n:int)(absintrow R) <- assignrow (infix :=) n R
and contintrow (absintrow R) <- controw R
;
```

Note that here the standard (overloaded) monomorphic "ref" and ":=" are supplied.

An example of a freely polymorphic function which uses "@" but not "ref" or ":=" is

```
var freeze :('a ref list -> 'a list) <- map @
```

Equality: Following Cardelli's approach, we take the view that
(i) We know what equality on data types (monomorphic) must mean,
(ii) we know that equality of references must mean identity,
(iii) otherwise, we should not determine its meaning.

Hence we allow the infixed predicates $=, <> : \Gamma \# \Gamma \to bool$ where Γ is any type built from polymorphic reference types $ty\ ref$ by data type constructors.

Thus for example, "=" is allowed at type 'a ref list but not at type 'a list.

Procedural programming: it seems best to throw out all the wild forms of looping in DEC10 ML, and just extend expressions by

```
exp ::= .... | exp1 ; exp2          (sequencing)
           | while exp1 do exp2     (iteration).
```

Too bad!

5. External Files in Standard ML

By virtue of the abbreviation (See Table 5)

exp ⟼ var any ← exp

every command sequence is — except for infix, nonfix commands — a sequence of declarations. In fact, since ";" is a declaration combinator, it is just a single declaration. It is therefore appropriate to add the new declaration form

dec ::= .... | use Token

where the token is a file-name, to extend the environment by declarations on an external file. There seems no need to restrict "use" to top level; non-top-level occurrences may be valuable, e.g.

local use `TREES.ML`
in abstype ... % the abstract type of balanced trees, say % ...

(Another use is to rename identifiers declared in the file, so as to avoid conflict with the current environment).

Any use file will be parsed with standard infix status; any fix commands which it contain will only affect the file itself (i.e. infix status is saved during use and restored afterwards). A use file can contain further use declarations.

All this seems to admit a later extension such as

dec ::= .... | engage `filename`

for loading and using precompiled environments (and even updating them); but I think Standard ML should stop short of this.

6. Input and output in Standard ML

The essence of these proposals is

(i) I/O functions take filenames (tokens) as parameters

(ii) As in PASCAL, files are opened either for reading or for writing — which cannot be mixed.

(iii) All *data* types Δ, ie. monotypes built with *data* type constructors, may be input or output — using current infix status for constructors.

Then we can do with just six functions

(1) openread : token → unit

"openread `f`" rewinds f to allow reading from the start; escapes with `NO FILE` if f does not exist.

(2) openwrite : token → unit

"openwrite `f`" creates a new empty file f to allow writing from the start.

(3) read : token → Δ

How is the type Δ conveyed to the operation? in what form?

"read `f`" reads the shortest value (construction) of type Δ.

Escapes with { `FAILED READ` if syntax is wrong; `EOF` on end-of-file; `NO OPENREAD` if not in read mode.

Note: read `` refers to keyboard,

"read `-`" refers to current file (within a *use* declaration)

(4) write : token → △ → unit

"write `f` exp" writes the value of exp on f.
Escapes with `NO OPENWRITE` if not in write mode.

Note: "write `""` refers to screen.

(5) readchar : token → token

"readchar `f`" reads a single character from f.
Escapes and conventions as for "read".
This allows layout characters to be read; things like PASCAL "readln" can thus be defined.

(6) writechar : token → token → unit

"writechar `f` t" writes the first character of t on f.
Escapes and conventions as for "write".

Remarks. For read/write dialogue at the Terminal, the implementation could usefully

(a) Indent the dialogue by say 3 spaces
(b) Prompt for input by the name of the type - e.g.
int list:

Note that programs can self-document the types of input values by explicit type expressions, e.g. "read``next : int list"

7. Miscellaneous

(1) Sections (DEC10 ML): I propose that sections be dropped. The feature by which they exported values (i.e. the value of "it" on exit from the section is that of the last expression in the section) seems adhoc in the new context. It seems entirely adequate to replace "begin .... exp end" by "local ..... in var result ← exp". The only thing we will lack is the naming of sections — but I don't think this is particularly useful.

(2) Conversion to and from Tokens: To match the input/output functions read, write (Section 6), the two functions

parse : Token → Δ
unparse : Δ → Token

(Δ any mono-data type)

are proposed. Since Δ can be "int", these subsume the old Int of tok, tok of int functions (and they are the identity when Δ = token).

(3) Literals in tokens and token lists: I propose we follow Cardelli's use of "/" to quote funny characters in Tokens and token lists, and his choice of representations for them.

(4) Comments: % ... %

(5) Boolean operations: the infixes "/\", "\/" (and, or) evaluate both arguments — i.e. they are true functions (following Luca).

TABLE 1 : ELEMENTARY SYNTAX CLASSES OF BARE ML

1. Identifiers

Id ::= Any sequence of letters and digits, starting with a letter, followed possibly by one or more primes (')

Any sequence of one or more of the symbols

! # $ & + - / : < = > ? @ \ ↑ _ ~ | * . ,

Exceptions : (i) Any reserved word (underlined in the syntax definitions) of Standard ML

(ii) The symbol sequences (these are also reserved words)

. | . ← : ⇐⇒

2. Constructors, Variables

Con ::= any identifier declared by a Data binding as a constructor or constant, within the scope of that binding; also the identifier absid, within the with part of an Abstract type binding which introduces id as a type constructor.

Var = Id \ Con (thus, constructors cannot be re-declared as variables within their scope).

3. Type variables

Tyvar ::= 'a | 'b | ... | 'z (pronounced alpha, beta, ... ?)

4. Type constructors

Tycon ::= any type constructor or constant declared by a Data or Abstract type binding, within the scope of that binding.

Notes (i) Infixed identifiers are introduced later – but not for type constructors.

(ii) An identifier used as a type constructor is considered distinct from the same used as a constructor or variable.

TABLE 2 : SYNTAX OF BARE ML

Conventions :
(i) {...} means optional
(ii) For any syntax class S,
S_seq ::= S | (S1, ..., Sn)
(iii) Alternatives are in order of decreasing binding power
(iv) Parentheses may enclose any named syntax class
(v) L, R mean left-, right-associative

EXPRESSIONS

exp ::=	var	(variable)
	con	(constant, constructor)
	exp1 exp2	L (application)
	exp : ty	(constraint)
	exp1, exp2	R (pairing)
	escape exp	(escape)
	exp trap match	(escape trapping)
	let dec in exp	(local declaration)
	fun match	(function abstraction)
match ::=	v1.exp1 | ... | vn.expn	(matching)

VARSTRUCTS

v ::=	any	(wild card)
	var	(variable)
	con {v_seq}	(construction)
	v : ty	(constraint)
	v1, v2	R (pairing)

TYPES

ty ::=	tyvar	(type variable)
	{ty_seq} tycon	L (type construction)
	ty1 # ty2	R (product type)
	ty1 → ty2	R (function type)

DECLARATIONS

dec ::=	{rec} var vb	(variables)
	{rec} data db	(data)
	{rec} abstype ab with dec	(abstract types)
	local dec1 in dec2	(local)
	dec1 ; dec2	(sequence)

VARIABLE BINDINGS

vb ::=	v ← exp	(simple)
	vb1 and vb2	(simultaneous)

DATA BINDINGS

db ::=	{tyvar_seq} id ← constrs	(simple)
	db1 and db2	(simultaneous)
constrs ::=	id1 {of ty1} | ... | idn {of tyn}	

ABSTRACT BINDINGS

ab ::=	{tyvar_seq} id ⇐⇒ ty	(simple)
	ab1 and ab2	(simultaneous)

COMMANDS

com ::=	dec	(declaration)
	exp	(expression)

Commands are separated by ";"

TABLE 3 : EXAMPLE OF A COMPOSITE DECLARATION

Setting up the free monoid over 'a

```
local
    rec data 'a seq ← nil | cons of 'a # 'a seq
in
abstype 'a monoid <=> 'a seq
with
    local rec var ap ← fun nil, m . m
                        | cons(x,l),m . cons(x, ap(l, m))
    in
var empty ← absmonoid nil
and singleton(x) ← absmonoid (cons(x, nil))
and concat (absmonoid l, absmonoid m) ← absmonoid (ap(l, m))
;
```

Notes (i) The local data declaration qualifies both the abstract binding ('a monoid <=> ...) and the with part.

(ii) Typechecking will ensure that no object with type involving "seq" will be exported by the whole declaration

(iii) The constructor "absmonoid" has been conveniently used in a varstruct; the need for "repmonoid" is naturally avoided.

TABLE 4 : PREDEFINED DATA DECLARATIONS FOR STANDARD ML

The type constructors unit, bool, token, int and list can be considered as predefined by the following declarations. Standard functions, also definable from these declarations, are not given here.

1. data unit ← unity ;

> CONVENTION : unity is represented instead by "()"

2. data bool ← true | false ;

3. rec data Token ← empty | C_1 of token | ... | C_n of token ;

(where $\{C_1, \cdots, C_n\}$ is the character set)

> CONVENTION : $C_{i_1}(\cdots(C_{i_k}\ \text{empty})\cdots)$ is represented instead by $`C_{i_1}\cdots C_{i_k}$

4. local rec data posint ← one | succ of posint
in data int ← zero | pos of posint | neg of posint ;

> CONVENTION : zero, pos(succ^{k-1}one), neg(succ^{k-1}one) are represented instead by the numerals $0, k, \sim k$ $\quad (k>0)$

5. infix :: 30 right ;
rec data 'a list ← nil | :: of 'a # 'a list ;

(Note : the qualifier "infix" is not required in data bindings)

Remark : Disjoint sum is not included as a standard type constructor, though it could easily be given by "data ('a,'b)sum ← inl of 'a | inr of 'b". It is likely that users will (or should) prefer their own definitions, with meaningful identifiers as constructors. On the other hand, for technical reasons it seems natural (perhaps necessary) to include Product type (pairing) in the Raw language instead of attempting to define it by data ('a,'b)prod ← pair of ('a,'b)". Note how the product '#' is needed in defining list, for example.

TABLE 5 : STANDARD ABBREVIATIONS AND ALTERNATIVE FORMS

Note : the abbreviations are not to suggest implementation, but merely to show that there is a semantically equivalent "Raw ML" form.

1. Expressions

- quit	⟼ escape `quit`	
- exp1 **or** exp2	⟼ exp1 **trap** any.exp2	
- exp1 **orif** "t1 .. tn" exp2	⟼ exp1 **trap** (`t1`.exp2 | ... | `tn`.exp2 | x. **escape** x)	(n≥0)
exp **where** dec	⟼ **let** dec **in** exp	
- **case** exp **of** match	⟼ (**fun** match) exp	
- **if** exp **then** exp1 **else** exp2	⟼ **case** exp **of** (true.exp1 | false.exp2)	
fun v1 ... vn , exp	⟼ **fun** v1.(... (**fun** vn.exp) ...)	(n≥1)
- [exp1 ; ... ; expn]	⟼ exp1:: ... :: expn ::nil	(n≥0)
- "t1 .. tn"	⟼ [`t1` ; ... ; `tn`]	(n≥0)
- exp1 ; exp2	⟼ **let** **var** any ← exp1 **in** exp2	
- **while** exp1 **do** exp2	⟼ f() **where** **rec** f() ← **if** exp1 **then** (exp2 ; f()) **else** ()	

2. Varstructs

- [v1; ... ; vn]	⟼ v1:: ... ::vn:: nil	(n≥0)
"t1 .. tn"	⟼ [`t1`; ... ; `tn`]	(n≥0)

3. Variable Bindings

id v1 .. vn {:ty} ← exp	⟼ id ← **fun** v1 ... vn {:ty}. exp	(n≥1)
? v1 id v2 v3 .. vn {:ty} ← exp (v1 id v2) v3 ... vn {:ty} ← exp }	⟼ **infix** id v1 ... vn {:ty} ← exp (when id has infix status)	(n≥2)

4. Declarations

exp ⟼ **var** any ← exp

Note: By this means expressions become a subclass of declarations. This means that any command sequence is (apart from fix commands) just a declaration! This fact is exploited in treating external ML files as declarations; see Section 5.

TABLE 6: PREDEFINED IDENTIFIERS IN STANDARD ML

NONFIXES

~	int → int	minus
@	α ref → α	contents
fst	α # β → α	unpairing
snd	α # β → β	unpairing
hd	α list → α	lists
tl	α list → α list	lists
map	(α → β) → α list → β list	lists
rev	α list → α list	lists
explode	token → token list	tokens
implode	token list → token	tokens
not	bool → bool	negation
unparse	Δ → token	unparsing
parse	token → Δ	parsing
ref	μ → μ ref	new reference (in expressions)
ref	α → α ref	reference (in variants)
it		value of last expression
openread	token → unit	input/output
openwrite	token → unit	input/output
read	token → Δ	input/output
write	token → Δ → unit	input/output
readchar	token → token	input/output
writechar	token → token → unit	input/output

INFIXES

	Type	Associativity	Precedence	Meaning
*	int # int → int	L	50	Arithmetic
div	int # int → int	L	50	Arithmetic
mod	int # int → int	L	50	Arithmetic
/\	bool # bool → bool	L		Conjunction
+	int # int → int	L	40	Arithmetic
−	int # int → int	L	40	Arithmetic
\/	bool # bool → bool	L		Disjunction
::	α # α list → α list	R	30	List cons
++	α list # α list → α list	R	30	List append
=	Γ # Γ → bool	L	20	Comparison
<>	Γ # Γ → bool	L	20	Comparison
<	int # int → bool	L	20	Integer order
>	int # int → bool	L	20	Integer order
<=	int # int → bool	L	20	Integer order
>=	int # int → bool	L	20	Integer order
o	(β → γ) # (α → β) → α → γ	L	10	Composition
&	(α → β) # (β → γ) → α → γ	L	10	Rev. composition
#	(α → β) # (α → γ) → α → (β # γ)	R	10	Pairing of functions
,	α # β → α # β	R	1	Pairing
:=	μ ref # μ → unit	L	0	Assignment

Note: μ is any monotype
Δ is any mono-data-type
Γ is any type built from reference types by data type constructors.

TABLE 7 : EXPRESSIONS IN STANDARD ML

exp ::=

var	(variable)
con	(constant, constructor)
exp1 exp2	(application)
exp : ty	(type constraint)
exp1 infix exp2	(infixed application)
escape exp	(escape with token)
quit	(escape with `quit`)
if exp1 then exp2 else exp3	(conditional)
case exp of match	(case analysis)
while exp1 do exp2	(iteration)
exp trap match	(escape trapping)
exp1 or exp2	(universal escape trapping)
exp1 orif "t1 ..tn" exp2	(selective escape trapping)
exp1 ; exp2	(sequence)
[exp1 ; .. ; expn]	(list; $n \geq 0$)
"t1 ... tn"	(token list ; $n \geq 0$)
exp where dec	(local declaration)
let dec in exp	(local declaration)
fun match	(function abstraction)
fun v1 ... vn. exp	(Curried abstraction; $n \geq 1$)

match ::= v1.exp1 | ... | vn.expn (Matching ; $n \geq 1$)

TABLE 8 : DECLARATIONS AND BINDINGS IN STANDARD ML

DECLARATIONS :

dec ::=	exp	(vacuous declaration)
	use `filename`	(external ML file)
	{rec} var vb	(variable declaration)
	{rec} data db	(data declaration)
	{rec} abstype ab with dec	(abstract type declaration)
	local dec1 in dec2	(local declaration)
	dec1 ; dec2	(declaration sequence)

VARIABLE BINDINGS :

vb ::=	v ← exp	(simple binding)
	id v1 ... vn {: ty} ← exp	(function binding : n ≥ 1)
?	{v1 infix v2} v3 ... vn {: ty} ← exp	(infix function binding : n ≥ 2)
	vb1 and vb2	(simultaneous binding)

DATA BINDINGS :

db ::=

{tyvar_seq} id ← constrs	(simple)
db1 and db2	(simultaneous)

constrs ::=

id1 {of ty1} | ... | idn {of tyn}	(n ≥ 1)

ABSTRACT TYPE BINDINGS :

ab ::=

{tyvar_seq} id ⇔ ty	(simple)
ab1 and ab2	(simultaneous)

TABLE 9 : VARSTRUCTS, TYPES and COMMANDS IN STANDARD ML

VARSTRUCTS :

v ::=	any	(wild card)
	var	(variable)
	Con {v_seq}	(construction)
	ref v	(reference)
	abstycon v	(abstraction)
	v : ty	(type constraint)
	v1 infix v2	(infixed construction)

TYPES :

ty ::=	tyvar	(type variable)
	{ty_seq} tycon	(type construction)
	ty1 # ty2	(product type)
	ty1 → ty2	(function type)

COMMANDS :

com ::=	dec	(declaration)
	exp	(expression)
	infix id1 ... idn {prec} {ass}	(infix status)
	nonfix id1 ... idn	(nonfix status)
prec ::=	1 | 2 | ...	
ass ::=	left | right	

Appendix B: Branching in `git`

The `git` version control tool has a feature called a *branch*. A *branch* is essentially a *copy* of the source code from a given point in time. Branches are particularly useful in the following scenarios:

- Multiple people are working on the source code simultaneously.
- You are working to implement an experimental feature that you may later decide to undo.
- You decide that the `master` branch is only for well-tested code destined for release.
- Each production release of a program is given its own branch.

And so on. There are many uses for branches. If you're ever found yourself copying code to another folder so that you could undo it later, you really should be using branches instead. `git` is better at tracking your files than you are.

tl;dr Version

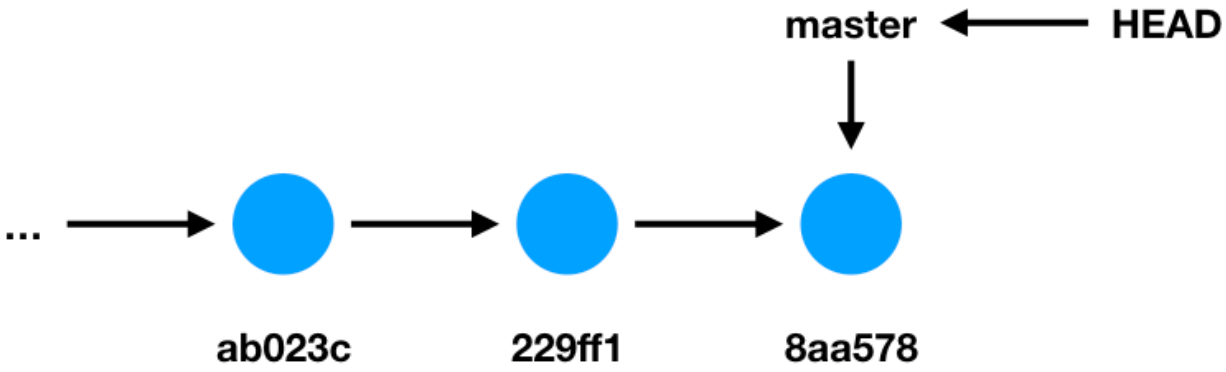
If you already know what branching is and just need a refresher, here are the commands described in this tutorial:

1. Create a new branch and switch to it: `git checkout -b <branchname>`
2. Switch branches: `git checkout <branchname>`
3. Merge changes into the current branch: `git merge <name of other branch>`

Otherwise, read on.

Tutorial

Suppose you have a source code repository with the following commits.



This diagram shows the last three commits in our repository. Recall that a commit is a *snapshot* of a repository during a point in *time*, and that each commit is identified by a random hexadecimal number such as the ones shown in the diagram.

In this diagram, all commits are currently in the default branch, which is called `master`. In this case, the `HEAD` pointer points to the branch pointer `master`. `HEAD` tells you which branch we are currently editing. In this case, the current branch is `master`.

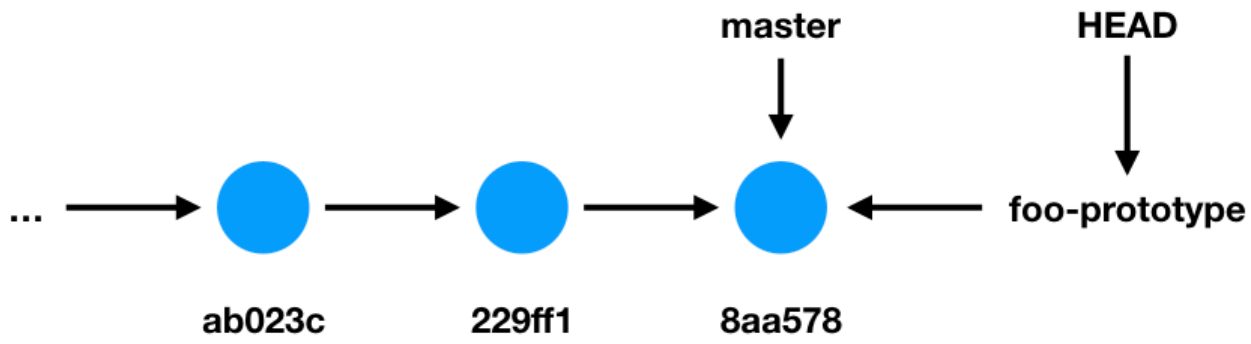
Creating a new branch

Imagine that we now want to implement a new feature: a function called `foo`. But because this feature may take some time to implement, and because other people are actively working on the codebase, we decide to work in a new branch. So create a new branch called "`foo-prototype`":

```
$ git checkout -b foo-prototype
```

The above command creates a new branch called `foo-prototype` and switches development to that new branch. Since we have not yet committed anything to our new branch, `foo-prototype` is essentially just a copy of the `master` pointer. But because `HEAD` *also* points to `foo-prototype`, that means that when we commit new code, that code will be added to the `foo-prototype` branch.

The following diagram shows the state of our repository after running the above commannd.



Switching branches

You can always switch to another branch. For example, if we wanted to switch back to `master`, we would run

```
$ git checkout master
```

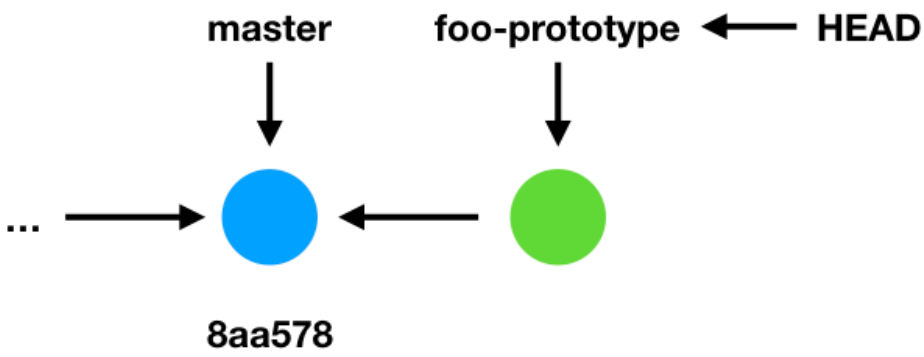
and if we wanted to switch back to `foo-prototype` we would run

```
$ git checkout foo-prototype
```

In each case, all that happens is that `git` moves the `HEAD` pointer.

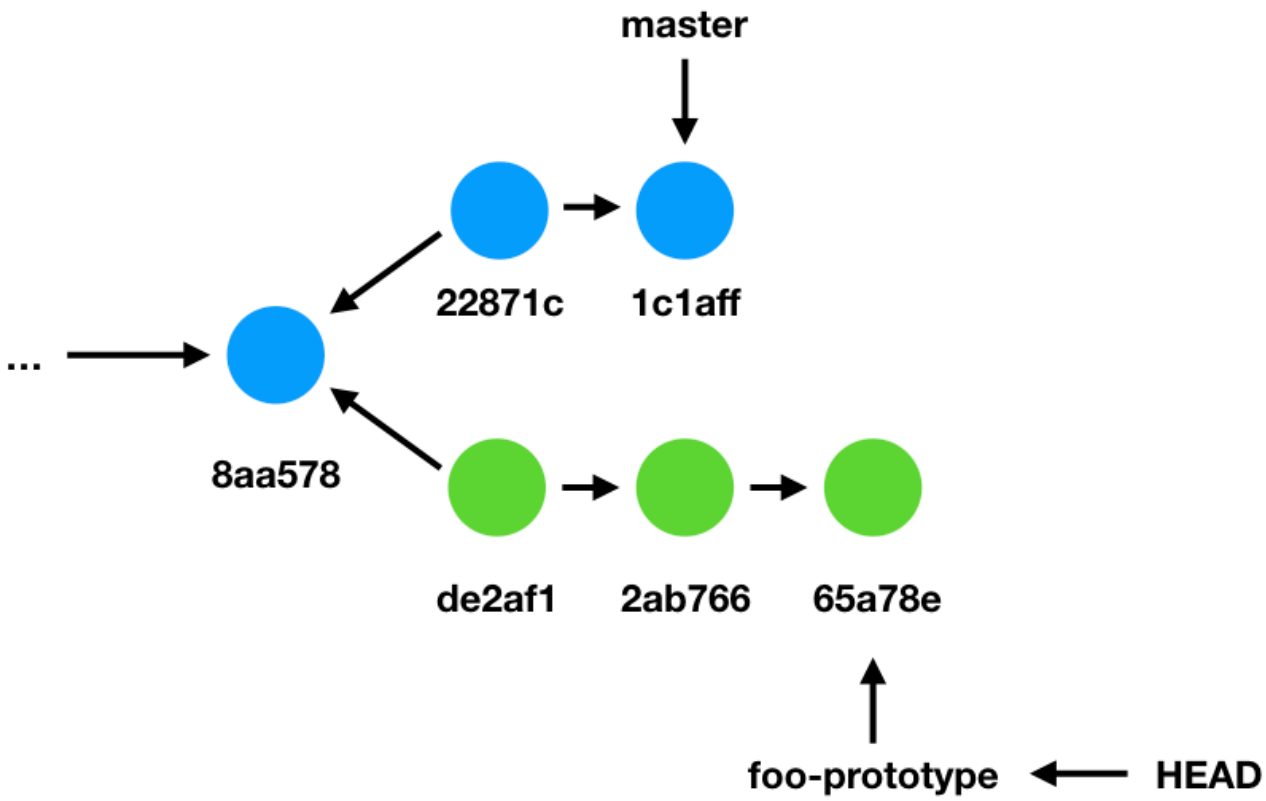
Merging

Now suppose we modify our source code—perhaps we've implemented a draft of our `foo` function—and we've committed it to our repository. This diagram shows the updated state of our repository. Notice that `foo-prototype` has now diverged from `master`.



Let's suppose that we continue on in this manner for awhile longer, committing things to our `foo-prototype` branch. Additionally, let's suppose that our collaborators have also been busy, adding things to our `master` branch in the meantime.

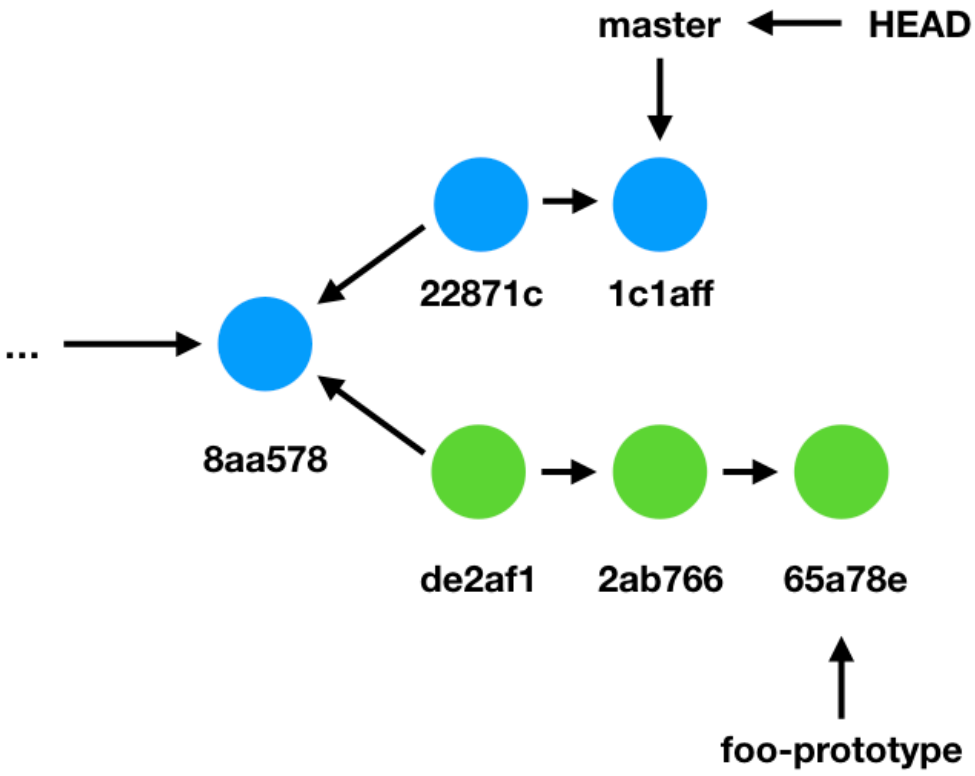
Finally, the big day arrives, and our `foo-prototype` is done. We want to bring our changes back into the `master` branch.



To merge `foo-prototype` back into `master`, first we switch back to the `master` branch.

```
$ git checkout master
```

Note that `HEAD` now points at `master`.



Next, we ask `git` to `merge` the changes from `foo-prototype` into the branch pointed to by `HEAD`.

```
$ git merge foo-prototype
```

If neither `master` nor `foo-prototype` have any changes to the *same* file, then the `merge` will happen automatically and you will be left with a new set of changes that you can commit.

Dealing with merge conflicts

It is much more likely, though, that there is a file that was edited in both branches. In this case, `git` needs your help to `merge` the two sets of files. This is called a *merge conflict*. Merge conflicts are not something to fear—rather, think of them as a helpful feature. `git` is telling you that the `merge` went fine except for a handful of files.

For example, suppose the file `filezzz` was modified in both `master` and in `foo-prototype`. After asking `git` to `merge`, we will see a message like the following.

```
$ git merge foo-prototype
Auto-merging filezzz
CONFLICT (content): Merge conflict in filezzz
Automatic merge failed; fix conflicts and then commit the result.
```

Running `git status` will show the conflicting files.

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   filezzz

no changes added to commit (use "git add" and/or "git commit -a")
```

Upon opening `filezzz` in our editor, we see that `git` helpfully marked the region of the file that is in conflict.

```
Some stuff
<<<<<<< HEAD
Some other more stuff
=======
Some more stuffs
Stuff
>>>>>>> foo-prototype
Some other stuff
```

What this says is that the entire file *is the same* except for a tiny region. In `HEAD`, that region contains

```
Some other more stuff
```

and in `foo-prototype` that region contains

```
Some more stuffs
Stuff
```

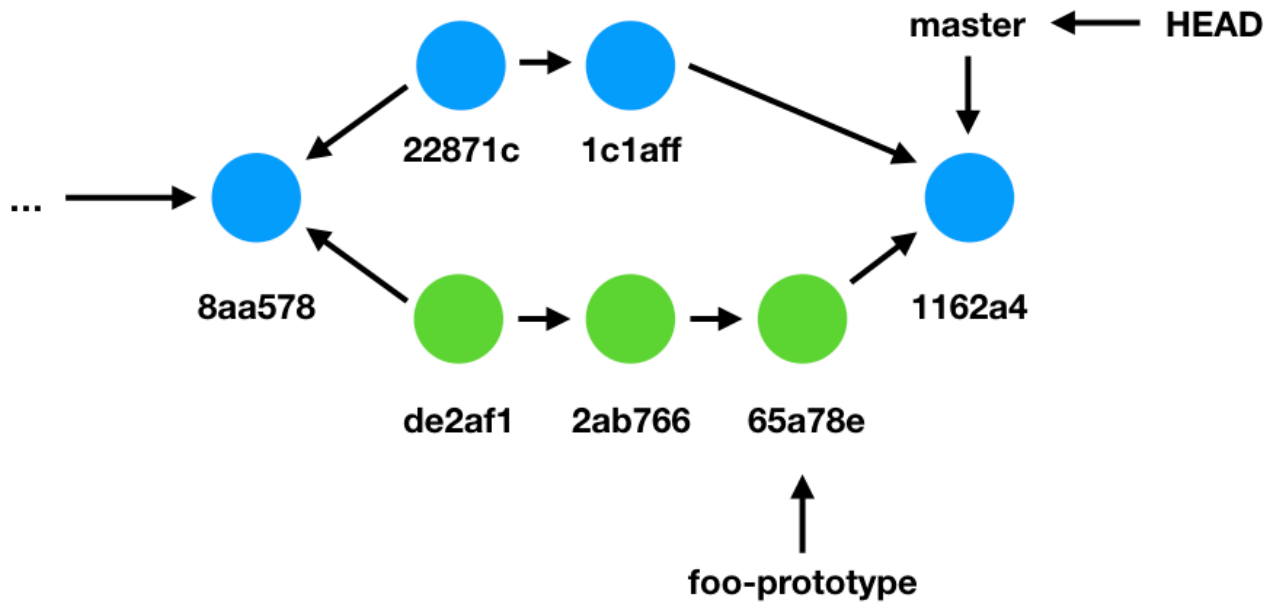
To merge, we decide what we want (e.g., "I think `Stuff` is fine for now"), and so we *replace the entire region* between `<<<<<<< HEAD` and `>>>>>>> foo-prototype`, inclusive. The file now contains:

```
Some stuff
Stuff
Some other stuff
```

`git` looks to see that you've removed the <<<<<<< and >>>>>>> markers to know when you've *resolved* a merge conflict. Now we can commit those changes.

```
$ git commit -am "merge"
[master 116d2a4] merge
```

Finally, our repository is in the following state and we are done.



Appendix C: A Short LaTeX Tutorial

This tutorial walks you through:

- how to pronounce LaTeX;
- compiling a LaTeX document;
- dealing with compiler errors;
- mathematical formulas; and
- formatting code.

This tutorial assumes that you already have LaTeX installed on your machine. If you don't, you need to install it[114]. Our UNIX lab machines already have LaTeX installed. Several students also report to me that Overleaf[115] is a pleasant alternative to installing LaTeX locally, but I have not tried it myself.

[114] https://www.latex-project.org/get/

[115] https://www.overleaf.com/

Pronouncing LaTeX

LaTeX is pronounced *LAH-tekh*. The "TeX" part consists of the Greek characters, tau, epsilon, and chi, and is a reference to the Greek word, *technē*, meaning "art" or "craft," which is the root of the word *technical*. TeX was written by the computer scientist Donald Knuth. The "La" part comes from Leslie Lamport, who was the person who packaged up TeX with a handy set of macros to make writing easier.[116]

[116] Both Knuth and Lamport have contributed extensively to the field of computer science, and have been subsequently awarded Turing Awards for their work. To be clear, though, those Turing Awards were *not* for TeX/LaTeX.

Compiling a LaTeX document

Suppose you have a LaTeX file called `file.tex`. You can compile `file.tex` to a PDF like so:

```
$ pdflatex file.tex
```

`pdflatex` will print out a lot of stuff. If your `file.tex` has no errors, you should see output that looks something like the following,

```
Output written on file.pdf (1 page, 27256 bytes).
```

and you will have a PDF version of your LaTeX document.

Dealing with compiler errors

If things go wrong, you will need to debug your LaTeX, just as you debug any other computer program. Look for missing brackets (e.g., `[` and `]`) or curly braces (e.g., `{` or `}`), look for typos in commands, and above all, read the output that the compiler is printing. LaTeX error messages are not the most understandable things, but they usually *do* tell you where to find the error.

One important thing to be aware of is that the LaTeX compiler usually drops you into an interactive shell instead of just stopping (like `javac` does). For example,

```
$ pdflatex file.tex
 ... lots of output ...
! Undefined control sequence.
<recently read> \tod

l.19   \item \tod
                {ANSWER}
?
```

Annoyingly, your cursor sits on this line and doesn't give you even a *clue* as to what to do.[117] The correct thing to do is easy: type a capital `X` and press the `ENTER` key.

More importantly, this compiler message actually tells me where to find the problem. On *line 19* (`l.19`), something about the LaTeX commands `\item \tod` causes an `Undefined control sequence`. One might argue that this is not a very good error message, but hey, all the information you need is there.

When I open up my `file.tex` I see the following on line 19:

```
\item \tod{ANSWER}
```

Oops! `\tod` should be `\todo`. After fixing this and running `pdflatex`

[117] By the way, I used LaTeX for years without actually knowing the correct way to escape from this situation. I am not ashamed to admit that I actually used to call the `kill` command to forcibly shutdown LaTeX.

again, `file.tex` again compiles correctly.

Mathematical formulas

One of LaTeX's great strengths is that it can produce beautiful-looking mathematical formulae. This is mostly easy to do, too: just put a mathematical expression in between a pair of dollar signs. For example:

```
$x + 5$
```

will produce a pretty-looking $x + 5$ expression.

I have not yet come across a mathematical figure that I could not reproduce using LaTeX. That's not to say that it is always easy, just that it is *possible*. See the LaTeX/Mathematics[118] reference for more information.

[118] https://en.wikibooks.org/wiki/LaTeX/Mathematics

Formatting code

Code is sometimes frustrating to format in LaTeX. Fortunately, there are two approaches, one easy, and one... less easy. Let's look at the easy one first.

LaTeX has many special "environments" that you can use for specially-formatted blocks of text. A very useful one for code is called `verbatim`. You use it like this:

```
\begin{verbatim}
  code goes here
\end{verbatim}
```

For example,

```
\begin{verbatim}
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
\end{verbatim}
```

`verbatim` will reproduce the text using a monospaced "typewriter" font that looks a lot like how we normally format code.

But LaTeX actually lets you do some very fancy things with code. Instead of using the `verbatim` environment, you can alternatively use the `lstlisting` environment. You need to do a little extra work:

1. You must add `\usepackage{listings}` to the top of your document.
2. You must supply a `\lstset` command, e.g., `\lstset{language=Pascal}` for the Pascal language.
3. You can then put your code inside a `lstlisting` environment.

The `listings` package can produce genuinely beautiful looking formatted code, with syntax highlighting and everything. If you're interested, see the LaTeX/Source Code Listings[119] page.

[119] https://en.wikibooks.org/wiki/LaTeX/Source_Code_Listings