The Heart of Spritely: Distributed Objects and Capability Security

Christine Lemmer-Webber, Randy Farmer

Table of Contents

NOTE: This is an early draft, still under technical review.

This paper is the second in a three-part series outlining Spritely's thinking and design. The first paper, Spritely: New Foundations for Networked Communities, explains the problems which face contemporary social network design. This paper details the core technical toolbox provided by

Spritely Goblins and how it supplies the necessary features to feasibly build out Spritely's broader vision. The third paper in the series, Spritely for Secure Applications and Communities, ties the first two papers together by showing how the architecture for user-facing software fulfills the vision of the first paper and can be built on top of ideas from this paper.

Spritely's core tooling is generally useful and this paper may be independently of interest to people with a wide variety of programming backgrounds. The architecture of this paper is also also designed designed with a purpose: to give us the firm footing to be able to achieve the ambitious journey of fulfilling the Spritely's user-facing vision. If your goal is to understand Spritely's full vision, it is our recommendation that you read each paper in order, however this is not a requirement.

1. Introduction

Building *peer-to-peer* applications on contemporary programming architecture is a complicated endeavor which requires careful planning, development, and maintenance. Building the kind of fully-decentralized design for healthy social community networks that Spritely aspires for would be too hard on systems that assume traditional client-server architecture and authority models. If each of our needs runs contrary to the grain of expected paradigms, we will have a hard time achieving our goals. Still, we must provide a development model which is comfortable in ways which match programmer intuitions. Spritely's core layers of abstractions achieve each of these seemingly contradictory requirements by drawing together decades of research from the object capability security and programming language design communities.

Spritely's core layers of abstraction make building secure *peer-to-peer* applications as natural as any other programming model. Spritely provides an integrated system for distributed asynchronous programming, *transactional* error handling, time-travel debugging, and safe serialization. All this under a security model resembling ordinary reference passing, reducing most considerations to a simple slogan: "If you don't have it, you can't use it."

2. Capability security as ordinary programming

> The Principle of Least Authority (POLA) says that code should be granted only the authority it needs to perform its task and no more. Code has a lot of power. Code can read your files, encrypt your files, delete your files, send your files (and all of the information within them) to someone else, record your keystrokes, use your laptop camera, steal your identity, hold your computer for ransom, steal your cryptocurrency, drain your bank account, and more. But most of the code that we write doesn't need to do any of those things – so why do we give it the authority to do so?
>
> POLA is ultimately about eliminating both ambient and excess authority. It's not a motto that is meant to be inspirational; POLA can actually be achieved. But how?
>
> – Kate Sills, POLA Would Have Prevented the Event-Stream Incident

The power of this model is best understood by contrast to Access Control Lists (ACL), the prevailing authority model, common to (and popularized by) Unix and nearly everything which has come before and followed since.

If Alisha is logged in to her computer and wants to play Solitaire, she can run it like so:

```
# Applications run as Alisha!
```

```
# Can do anything Alisha can do!
SHELL> solitaire
```

In an ACL permission system Solitaire, the most innocuous-seeming of programs, can wreak the maximum amount of havoc possible to Alisha's computing life. Solitaire could snoop through Alisha's love letters, upload her banking information to a shady website, and delete or cryptolock her files (possibly demanding a tidy sum on behalf of some shady group somewhere to release access).

What makes seemingly-innocent Solitaire so dangerous is the *ambient authority* of Access Control List operating systems. In such a computing environment, when Alisha types "solitaire" in a terminal window or double clicks on its icon, her computer runs Solitaire *as Alisha*. Solitaire can do everything Alisha can do, including many dangerous things Alisha would not like.

The contrast with an object capability environment is strong. Following the *principle of least authority*,[1] programs, objects, and procedures are defined in an environment with no dangerous authority. In an object capability computing environment, Solitaire would only be able to run with the authority it has been handed.

Let's think of `solitaire` as being a procedure within an object capability secure language. (To make it obvious that these ideas can extend to a variety of language environments,[2] we will use a syntax which resembles something like Javascript or Python.) Solitaire, being run, cannot do anything particularly dangerous… but it can't do anything particularly useful either.

```
# Runs in an environment with no special authority...
# not even the ability to display to the screen!
REPL> solitaire()
```

As-is, all solitaire can do is return a value… but Solitaire as a game requires interactivity: it should display to the screen, and it should be able to read input through the keyboard and mouse.

Let's introduce a capability which has been granted more power by the underlying system, `makeWinCanvas(windowTitle)`. Let's say that `solitaire` can take a first argument which takes a window + canvas representing an object which is able to read keyboard and mouse input, but only while the window is active. We will be able to use the former to produce a value to pass to the latter, with exactly that authority and no more:

```
# Constructs a new window
REPL> solitaireWin = makeWinCanvas("Safe Solitaire")
# Pass it to solitaire
REPL> solitaire(solitaireWin)
```

If we want to allow Solitaire to be able to access a high score file, we could imagine that the `solitaire` procedure could accept a third procedure for exactly that purpose:

```
REPL> scoreFile = openFile("~/.solitaire-hs.txt", "rw")
REPL> solitaire(solitaireWin, scoreFile)
```

Consider the power of this: solitaire now has access to display to the `solitaireWin` window, it can read from the keyboard and mouse when the window is active, it can only write to the specific file we have given access to, but it cannot do anything else dangerous.[3] It cannot access the

1 **TODO:** Incorporate a paragraph from Alan Karp here explaining POLA, POLP, the links between, etc. And put in the glossary too :)

2 The requirements for a programming language to be considered object capability safe are reasonably minimal (no ambient authority, no global mutable state, lexical scoping with reference passing being the primary mechanism for capability transfer, and importing a library should not provide access to interesting authority). See A Security Kernel Based on the Lambda Calculus for more information.

3 TODO: Write this

network. It cannot read or write files from the filesystem arbitrarily (it can only access the high score file it was given). It cannot act as a keylogger (it can only read keyboard and mouse events while the window is being actively used by the user).[4]

We have built our object capability security model on completely ordinary reference passing, familiar to the kind of programming developers do every day. What can and cannot be done is clear: if you don't have it, you can't use it.

3. Spritely Goblins: Distributed, transactional object programming

At the heart of Spritely is Goblins, its *distributed object programming* environment.[5] Goblins provides an intuitive security model, automatic local *transactions* for locally synchronous operations, and easy to use and efficient asynchronous communication with *encapsulated objects* which can live anywhere on the network. Its networking model abstracts away these details so the programmer can focus on object programming rather than network protocol design. Goblins also integrates powerful distributed debugging tools, and a process persistence and upgrade model which respects its security fundamentals.[6]

Within Goblins, when we say *distributed object*, we are referring to a model where many independent objects communicate with other objects on many different machines. In other words, when we refer to *distributed object programming*, we mean "a distributed network of independent objects".[7] Objects are built out of *encapsulated behavior*: an object is *encapsulated* in the sense that its inner workings are opaque to other objects, and (contrary to the focus of many systems today) objects are *behavior-oriented* rather than *data-oriented*. Goblins enables intentional collaboration between objects even though the network is assumed hostile as a whole.

Goblins utilizes techniques common to functional programming environments which enable cheap *transactionality* (and by extension, *time travel*). The otherwise tedious plumbing associated with these kinds of techniques is abstracted away so the developer can focus on object behavior and interactions.

4 TODO: Write this

5 In recent years there has been enormous pushback against the term "object", stemming mostly from functional programming spaces and PTSD developed from navigating complicated Java-esque class hierarchies. However, the term "object" means many different things; Jonathan Rees identified nine possible properties associated with programming uses of the word "object". For Goblins, *objects* most importantly means addressable entities with encapsulated behavior. Goblins supports *distributed objects* in that it does not particularly matter where an object lives for asynchronous message passing; more on this and its relationship with *actors* later.

6 Goblins draws inspiration largely from two sources. The first is Scheme (on which its current implementations are built), and particularly the "W7" Scheme variant found in A Security Kernel Based on the Lambda Calculus, and the E programming language. (Both of these have rich histories of their own, particularly E's predecessor Joule, so of course Goblins inherits those too.) W7's primary contribution is the observation that a purely lexically scoped language, with Scheme in particular, is already an excellent candidate for an object capability security environment. E's primary contribution is the *distributed object* approach that Goblins largely adopts, including the first version of the *CapTP* protocol used by Goblins as the object communication layer abstraction of OCapN. Goblins can thus be seen as a combination of Scheme/W7 and E, with Goblins' primary innovative contribution being its *transactional* design.

7 This is not to be confused with "the abstract conceptual objects themselves are distributed/replicated across different machines", which we do address as the *Unum Pattern* in the Distributed behavior and why we need it section.
Similarly we do not mean distributed *convergent machines* (such as *blockchains* or *quorums*), where a single abstract *machine*, with all of its contained objects, can be deterministically replicated by multiple independent machines on the network. While such designs can be composable with Spritely Goblins (or even easily built on top of its *transactional* architecture), they are not the essential infrastructure to achieve Spritely's goals. Further discussion of *convergent machines* is reserved for a future paper.

3.1. On language and syntax choice

The following examples will illustrate Goblins using its implementation in *Guile* (which is a *dialect* of *Scheme*, which is itself a *dialect* of *Lisp*).[8] While the ideas here could be ported across many kinds of programming languages, Scheme's minimalism and flexibility allow for cleanly expressing the core ideas of Goblins.

Prior knowledge of Scheme is not necessary, but some familiarity with programming in general is expected. See Appendix: A small-ish Scheme and Wisp primer if you'd like an introduction to Scheme.

We have chosen an unusual representation of Lisp syntax which is whitespace-based instead of parenthetical, named Wisp.[9] Experience has shown that while parenthetical representations of Lisp tend to feel alien to newcomers with prior programming experience, Wisp tends to look fairly pleasantly like pseudocode. We have aimed to have these examples be as simple as possible to understand just by reading them.

Nonetheless, a short mention of how Wisp and Lisp relate is useful. The left-hand syntax is written in Wisp, whereas the right-hand code is written in standard parenthetical Scheme:

```
define (add-drawing p f)              |   (define (add-drawing p f)
  define drawer                       |     (define drawer
    make-pict-drawer p                |       (make-pict-drawer p))
  new canvas%                         |     (new canvas%
    parent f                          |       (parent f)
    style '(border)                   |       (style '(border))
    paint-callback                    |       (paint-callback
      lambda (self dc)                |         (lambda (self dc)
     drawer dc 0 0                    |           (drawer dc 0 0)))))
```

These are just different *surface syntax* representations of the same program. The code can mostly be read by indentation, with deeper nested indentation levels representing nested sub-expressions. Sections of code wrapped in parentheses retain their parenthetical representation as-is.

There are only a couple of tricky details to know. First, lines starting with a dot continue a previous expression, and keywords (eg, `#:happy?` below) are implicitly considered to be continuing arguments in the previous expression:

```
render-to-file                        |   (draw-to-file
  . "cool-cat.png"                    |     "cool-cat.png"
  make-cat-drawing                    |     (make-cat-drawing
    #:happy? #t                       |       #:happy? #t
    #:size 100                        |       #:size 100))
```

Second, a colon can be used to nest a sub-expression on the same line:

```
define (get-and-save-username db)     |   (define (get-and-save-username db)
  define name : input "Name:"         |     (define name (input "Name:"))
  db-store db "username" name         |     (db-store db "username" name))
```

8 At present, Goblins has two implementations, one on Racket (the initial implementation), and one on Guile (which is newer). While both will be maintained and interoperable with each other in terms of distributed communication, the Guile implementation is becoming the "main" implementation on top of which the rest of Spritely is being built. Goblins' ideas are fairly general though and Goblins is implemented simply as a library on top of a host programming language, and Goblins' key ideas could be ported to any language with sensible lexical scoping (but it might not look as nice or be as pleasant to use or elegant).

9 Wisp's rules are defined in SRFI 119. Wisp's key feature is that it has all the same structural properties as a parenthetical representation and can be translated back and forth between the parenthetical form and the whitespace-based form bidirectionally with few key rules.

That's all you need to know about Wisp.

Lisp and Scheme programming (not only, but especially) tends to involve a cycle between experimenting at the interactive *REPL* and code you keep around in a file. We follow the same convention in this paper. Code examples that have lines preceding with `REPL>` are meant to demonstrate examples of interactive use. Lines which follow and are preceded by underscores represent continued entries for the same expression:

```
REPL> define name "Doris"
REPL> string-append "Hello " name "!"
; => "Hello Doris!"
REPL> display "Hello screen output!\n"
; prints: Hello screen output!
```

To get your *REPL* set up properly for live programming, you will need to do a few things; see Appendix: Following the code examples for more.

3.2. A taste of Goblins

The following section gives a high-level demonstration of Goblins through practical use.

If you do choose to follow along by entering the code from this section, you can define this as a full-fledged module, say perhaps `taste-of-goblins.w`, like so:

```
define-module : taste-of-goblins
  #:use-module : goblins
  #:use-module : goblins actor-lib methods
  #:export (^cell ^greeter ^cgreeter ^borked-cgreeter
          ^car-factory ^borked-car-factory)
```

Code examples that are *not* interactive can/should be entered into this file.

TODO: The above goes into Appendix: Following the code examples, not here.

Now you're ready to go. Read on!

3.2.1. A simple greeter

Here we will give an extremely brief taste of what programming in Goblins is like. The following code is adapted from the Guile version of Goblins.[9]

First, let us implement a friend who will greet us:

```
;; define with next argument wrapped in parentheses
;; defines a named function
define (^greeter _bcom our-name)   ; constructor (outer procedure)
  lambda (your-name)               ; behavior    (inner procedure)
    format #f "Hello ~a, my name is ~a!"     ; returned implicitly
         . your-name our-name
```

The outer procedure, defined by `define`, is named `^greeter`, which is its constructor.[10] The inner procedure, defined by the `lambda` (an "anonymous function"), is its behavior procedure, which implicitly returns a formatted string. Both of these are most easily understood by usage, so let's try instantiating one:[11]

10 The `^` character is conventionally prefixed on Goblins constructors and is called a *hard hat*, referring to the kind used by construction workers.

11 Any code line preceded by `REPL>` represents the prompt for interactively entered code at a developer's REPL (Read Eval Print Loop). Lines following represent expected returned values or behavior, and those prefixed with `=>` represent an expected return value.

```
;; define with next argument *not* in parentheses
;; defines an ordinary variable
REPL> define gary
_____   spawn ^greeter "Gary"
```

As we can see, `spawn`'s first argument is the constructor for the Goblins object which will be spawned. For now, we'll ignore the `_bcom`, which is not used in this first example (an underscore prefix is the conventional way to note an unused variable; we'll see some examples where this *is* used soon). The rest of the arguments to `spawn` are passed in as the rest of the arguments to the constructor. So in our case, `"Gary"` is passed as the value of `our-name`.

The constructor returns the procedure representing its current behavior. In this case, that behavior is a simple anonymous `lambda`. We can now invoke our `gary` friend using the synchronous call-return `$` operator:

```
REPL> $ gary "Alice"
;; => "Hello Alice, my name is Gary!"
```

As we can see, `"Alice"` is passed as the value for `your-name` to the inner `lambda` behavior-procedure. Since `our-name` was already bound through the outer constructor procedure, the inner behavior is able to pass both of these names to `format` to give a friendly greeting.

3.2.2. State as updating behavior

Let's introduce a simple cell which stores a value. This cell will have two methods: `'get` retrieves the current value, and `'set` replaces the current value with a new value.

```
define (^cell bcom val)
  methods             ; syntax for first-argument-symbol-based dispatch
    (get)             ; takes no arguments
      . val           ; returns current value
    (set new-val)     ; takes one argument, new-val
      bcom : ^cell bcom new-val  ; become a cell with the new value
```

Let's try it. Cells hold values, and so do treasure chests, so let's make a treasure chest flavored cell. Taking things out and putting them back in is easy.

```
REPL> define chest
_____   spawn ^cell "sword"
REPL> $ chest 'get
;; => "sword"
REPL> $ chest 'set "gold"
REPL> $ chest 'get
;; => "gold"
```

Now we can see what `bcom` is: a capability specific to this object instance which allows it to change its behavior! (For this reason, `bcom` is pronounced "become"!)

`methods` was also new to this version. It turns out that `methods` is simply syntax sugar, a macro which returns a procedure which supports symbol dispatch on its first argument. There is nothing special about `methods`: you could easily write your own version or use it outside of Goblins objects to build general symbol-based-method-dispatch.

3.2.3. Objects which contain objects

Objects can also contain and define other object references, including in their outer constructor procedure.

Here is the definition of a "counting greeter" we call `^cgreeter`:

```
define (^cgreeter _bcom our-name)
  define times-called   ; keeps track of how many times 'greet called
    spawn ^cell 0      ; starts count at 0
  methods
    (get-times-called)
      $ times-called 'get
    (greet your-name)
      define current-times-called
      $ times-called 'get
      ;; increase the number of times called
      $ times-called 'set
      + 1 current-times-called
      format #f "[~a] Hello ~a, my name is ~a!"
           $ times-called 'get
           . your-name our-name
```

As we can see near the top, `times-called` is instantiated as an `^cell` like the one we defined earlier. The current value of this cell is returned by `get-times-called` and is updated every time the `greet` method is called:

```
REPL> define julius
_____   spawn ^cgreeter "Julius"
REPL> $ julius 'get-times-called
;; => 0
REPL> $ julius 'greet "Gaius"
;; => "[1] Hello Gaius, my name is Julius!"
REPL> $ julius 'greet "Brutus"
;; => "[2] Hello Brutus, my name is Julius!"
REPL> $ julius 'get-times-called
;; => 2
```

3.2.4. Asynchronous message passing

We have shown that the behavior of objects may be invoked synchronously with `$`. However, this only works if two objects are both defined on the same machine on the network and the same event loop within that machine. Since Goblins is designed to allow for object invocation across a distributed network, what can we do?

This is where `<-` comes in. In contrast to `$`, `<-` can be used against objects which live anywhere, even on remote machines. However, unlike invocation with `$`, we do not get back an immediate result, we get a promise:

```
REPL> <- julius 'greet "Lear"
;; => #<promise>
```

This promise must be listened to. The procedure to listen to promises in Goblins is called `on`:

```
REPL> on (<- julius 'greet)
_____   lambda (got-back)
_____     format #t "Heard back: ~a\n"
_____             . got-back
; prints (eventually):
;   Heard back: [4] Hello Lear, my name is Julius!
```

Not all communication goes as planned, especially in a distributed system. `on` also supports the keyword arguments of `#:catch` and `#:finally`, which both accept a procedure defining handling errors in the former case and code which will run regardless of successful resolution or

failure in the latter case:

```
REPL> define (^broken-bob _bcom)
_____   lambda ()
_____     error "Yikes, I broke!"
REPL> define broken-bob
_____   spawn ^broken-bob
REPL> on (<- broken-bob)
_____    lambda (what-did-bob-say)
_____      format #t "Bob says: ~a\n" what-did-bob-say
_____    #:catch
_____    lambda (err)
_____      format #t "Got an error: ~a\n" err
_____    #:finally
_____    lambda ()
_____      display "Whew, it's over!\n"
; prints (eventually):
;   Got an error: <error ...>
;   Whew, it's over!
```

3.2.5. Transactions make errors survivable

Mistakes happen, and when they do, we'd like damage to be minimal. But with many moving parts, accomplishing this can be difficult.

However, Goblins makes our life easier. To see how, let's intentionally insert a couple of print debugging lines (with `pk`, which is pronounced and means "peek") and then an error:

```
define (^borked-cgreeter _bcom our-name)
  define times-called
    spawn ^cell 0
  methods
    (get-times-called)
      $ times-called 'get
    (greet your-name)
      pk 'before-incr : $ times-called 'get
      ;; increase the number of times called
      $ times-called 'set
     + 1 ($ times-called 'get)
      pk 'after-incr : $ times-called 'get
      error "Yikes"
      format #f "[~a] Hello ~a, my name is ~a!"
          $ times-called 'get
          . your-name our-name
```

Now let's spawn this friend and invoke it:

```
REPL> define horatio
_____   spawn ^borked-cgreeter "Horatio"
REPL> $ horatio 'get-times-called
;; => 0
REPL> $ horatio 'greet "Hamlet"
;; pk debug: (before-incr 0)
;; pk debug: (after-incr 1)
;; ice-9/boot-9.scm:1685:16: In procedure raise-exception:
;;   Yikes
;; Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
```

Whoops! Looks like something went wrong! We can see from the `pk` debugging that the `times-called` cell should be incremented to 1. And yet…

```
REPL> $ horatio 'get-times-called
;; => 0
```

We will cover this in greater detail later, but the core idea here is that synchronous operations run with `$` are all done together as one *transaction*. If an unhandled error occurs, any state changes resulting from synchronous operations within that *transaction* will simply not be committed. This is useful, because it means most otherwise difficult cleanup steps are handled automatically.

This also sits at the foundation of Spritely Goblins' time travel debugging features. All of this will be discussed in greater detail in sections later in this document: The vat model of computation, Turns are cheap transactions, and Time-travel distributed debugging.

3.2.6. Promise pipelining

> "Machines grow faster and memories grow larger. But the speed of light is constant and New York is not getting any closer to Tokyo."
>
> — Mark S. Miller, Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control

Promise pipelining[12] provides two different features at once:

- A convenient developer interface for describing a series of asynchronous actions, allowing for invoking the objects which promises will point to before they are even resolved (sometimes before the objects even exist!)
- A network abstraction that eliminates many round trips[13]

Consider the following car factory, which makes cars carrying the company name of the factory:

```
;; Create a "car factory", which makes cars branded with
;; company-name.
define (^car-factory bcom company-name)
  ;; The constructor for cars we will create.
  define (^car bcom model color)
```

12 Like so many examples in this document, the designs of promise pipelining and the explanation of its value come from the E programming language, the many contributors to its design, and Mark S. Miller's extraordinary work documenting that work and its history. If you find this section interesting, both the Promise Pipelining page from erights.org and sections 2.5 and 16.2 of Mark Miller's dissertation.

Note that if you are familiar with promises in Javascript, those are also inspired by E (and its predecessor Joule)'s promises. However, the full version of promises, including promise pipelining (or its most powerful use combined with network programming) were never included in Javascript proper. E's full vision of promises are present in Spritely Goblins, as we outline here.

13 Promises without promise pipelining are already an improvement over raw callbacks but are still insufficiently ergonomic for convenient programming. "Callback hell" and the annoyance of "`.then()` chaining" have lead many developers to prefer coroutines via `async` and `await` type operators. Goblins does have support for coroutines, but their use is somewhat cautioned against, and we are not prioritizing them. Coroutines give the illusion of straightahead call-return style programming by flattening callback structures. Unfortunately, while call-return programming is synchronous, coroutines are really "splitchronous"… each invocation of `await` splits time. `await` makes it very easy to accidentally mistake splitchronous code as being synchronous code, but the difference is severe: the world can change around the user during the time between a coroutine's suspension and resumption, opening up a class of vulnerabilities known as "re-entrancy attacks". This risk was observed during E's development and lead E to not include coroutines at all. A couple of decades later, re-entrancy attacks became the number one way money has been stolen in Etherium due to bugs in smart contracts.

But there is another reason to prefer promise pipelining over coroutines: the reduction of round-trips! A coroutine requires waiting for a response to come back before deciding upon the next action, which is not a requirement for a promise pipelining based system.

```
    methods                         ; methods for the ^car
      (drive)                       ; drive the car
      format #f "*Vroom vroom!*  You drive your ~a ~a ~a!"
             . color company-name model
  ;; methods for the ^car-factory instance
  methods                           ; methods for the ^car-factory
    (make-car model color)          ; create a car
      spawn ^car model color
```

Here is an instance of this car factory, which we will call `fork-motors`:

```
;; Interaction on machine A
REPL> define fork-motors
_____   spawn ^car-factory "Fork"
```

Since asynchronous message passing with `<-` works across machines, it does not matter whether interactions with `fork-motors` are local or via objects communicating over the network. We will treat `fork-motors` as living on a remote machine A, and so the following interactions will happen with invocations originating from our local machine B.

Let's send a message to `fork-motors` invoking the `'make-car` method, receiving back a promise for the car which will be made, which we shall name `car-vow` (`-vow` being the conventional suffix given for promises in Goblins):

```
;; Interaction on machine B, communicating with fork-motors on A
REPL> define car-vow
_____   <- fork-motors 'make-car "Explorist" "blue"
```

So we have a *promise* to a future car reference, but not the reference itself. We would like to drive the car as soon as it rolls off the lot of the factory, which of course involves sending a message to the car.

Without promise pipelining, making use of the tools we have already shown (and following the pattern most other distributed programming systems use), we would end up with something like:

```
;; Interaction on machine B, communicating with A
REPL> on car-vow                    ; B->A: first resolve the car-vow
_____   lambda (our-car)            ; A->B: car-vow resolved as our-car
_____     on (<- our-car 'drive)    ; B->A: now we can message our-car
_____       lambda (val)            ; A->B: result of that message
_____         format #t "Heard: ~a\n" val
; prints (eventually):
;   Heard: *Vroom vroom!*  You drive your blue Fork Explorist!
```

With promise pipelining, we can simply message the promise of the car directly. The first benefit can be observed from code compactness, in that we do not need to do an `on` of `car-vow` to later message `our-car`, we can simply message `car-vow` directly:

```
;; Interaction on machine B, communicating with A
REPL> on (<- car-vow 'drive)        ; B->A: send message to future car
_____   lambda (val)                ; A->B: result of that message
_____     format #t "Heard: ~a\n" val
; prints (eventually):
;   Heard: *Vroom vroom!*  You drive your blue Fork Explorist!
```

While clearly a considerable programming convenience, the other advantage of promise pipelining is a reduction of round-trips, whether between our event-loop *vats* or across machines on the network.

This can be understood by looking at the comments to the right of the two above code interactions.

The message flow in the first case looks like:

```
B => A => B => A => B
```

The message flow in the second case looks like:

```
B => A => B
```

In other words, machine B can say to machine A: "Make me a car, and as soon as that car is ready, I want to drive it!"

With this in mind, the promise behind Mark Miller's quote at the beginning of this section is clear. If two objects are on opposite ends of the planet, round trips are unavoidably expensive. Promise pipelining both allows us to make plans as programmers and allows for Goblins to optimize carrying out those steps as bulk operations over the network.

3.2.7. When schemes go awry: failure propagation through pipelines

```
Thy wee bit heap o' leaves an' stibble,
 Has cost thee mony a weary nibble!
 Now thou's turn'd out, for a' thy trouble,
 But house or hald,
 To thole the winter's sleety dribble,
 An' cranreuch cauld!

 But, Mousie, thou art no thy-lane,
 In proving foresight may be vain;
 The best-laid schemes o' mice an' men
 Gang aft agley,
 An' lea'e us nought but grief an' pain,
 For promis'd joy!

 Still thou art blest, compar'd wi' me
 The present only toucheth thee:
 But, Och! I backward cast my e'e.
 On prospects drear!
 An' forward, tho' I canna see,
 I guess an' fear!

 — From "To a Mouse, on Turning Her Up in Her Nest With the Plough" by Robert
Burns, 1785
```

Unexpected behavior can cause a cascade of failures. In a synchronous call-return system with exceptions, raising an exception causes not only the current procedure invocation to fail, but further invocations up the chain until the exception is caught (and if uncaught, possibly by allowing the program as a whole to fail). While potentially frustrating to encounter as a programmer or user, the alternative of proceeding without mitigating unhandled behavior could be equally disastrous. Still, if we can interpret each procedure as voluntarily "sending a message to its caller" that something has gone awry, we can see the great service that each callee performs for its caller (such a pattern is common when a language does not provide implicit exception support), allowing the caller to make new plans, or at least not move forward under assumptions that no longer hold. Even unhandled exceptions, observed by the programmer, can be an opportunity to study and make new plans so that things may work better next time.

In a highly asynchronous networked environment, the likellihood of unanticipated failures grows substantially. Even with the most well implemented, bug-free *locally implemented* code (itself usually less likely a possibility than its authors may think), network connections are fickle, and remote objects may misbehave. As such, if a promise is broken, a pipelined message to that promise

will have nowhere to go. This too should be interpreted as a failure and handled correctly.

As an example of this, consider this broken implementation of a car factory:

```
define (^borked-car-factory bcom company-name)
  define (^car bcom model color)
    methods                       ; methods for the ^car
      (drive)                     ; drive the car
       format #f "*Vroom vroom!*  You drive your ~a ~a ~a!"
              . color company-name model
  ;; methods for the ^car-factory instance
  methods                         ; methods for the ^car-factory
    (make-car model color)        ; create a car
      error "Your car exploded on the factory floor!  Ooops!"
      spawn ^car model color
```

What would happen if we tried making a car using this factory and then pipeline a message to drive it?

```
REPL> define forked-motors
_____   spawn ^borked-car-factory "Forked"
REPL> define car-vow
_____   <- forked-motors 'make-car "Exploder" "red"
REPL> define drive-noise-vow
_____   <- car-vow 'drive
REPL> on drive-noise-vow
_____    lambda (val)
_____      format #t "Heard: ~a\n" val
_____    #:catch
_____    lambda (err)
_____      format #t "Caught: ~a\n" err
; prints (eventually):
;   Caught: <error...>
```

Even though it is `car-vow` which is initially broken, its exception propagates to `drive-noise-vow`. Since there would be no useful way to drive a broken promise of a car anyhow, this is the correct design, and the situation can be detected and dealt with.

3.3. Security as relationships between objects

> Cooperation between independent agents depends upon establishing a degree of security. Each of the cooperating agents needs assurance that the cooperation will not endanger resources of value to that agent. In a computer system, a computational mechanism can assure safe cooperation among the system's users by mediating resource access according to desired security policy. Such a mechanism, which is called a security kernel, lies at the heart of many operating systems and programming environments.
>
> – Jonathan A. Rees, A Security Kernel Based on the Lambda Calculus

In Capability security as ordinary programming we demonstrated how a programming language which uses lexical scoping and is strict about removing ambient authority is already likely an excellent foundation for a capability secure architecture. In A taste of Goblins we saw Goblins' powerful *transactional* distributed object programming system. This section shows the union of the two: that the relationships between Goblins objects is an excellent, expressive, and sufficient security model for networked programs.

To make this clear we will present a common tutorial: a blogging style system[14] (in our case, used by a community newspaper of an imagined town) with different users cooperating and performing different roles. Unlike most such tutorials, this is accomplished without an access control list: resources are protected from misuse without relying on checking the identity of the performing agent. Despite this, we will manage to introduce accountability and revocation features, the protection of misuse from unauthorized parties, and even the demonstration of a multiple-stakeholder cooperation pattern which has no direct parallel in an access control system.

3.3.1. Making and editing a blogpost

Lauren Ipsdale has decided to run a newspaper for her local community. The first thing Lauren will need is a way to construct individual posts which can be widely read, but edited only by trusted editors.

Lauren creates a new post:

```
REPL> define-values (day-in-park-post day-in-park-editor)
_____   spawn-post-and-editor
_____     #:title "A Day in the Park"
_____     #:author "Lauren Ipsdale"
_____     #:body "It was a good day to take a walk..."
```

(We will show implementation details of these blogposts below, but first we will focus on narrative and use.)

`spawn-post-and-editor` returned two capabilities:

- `day-in-park-post`, which grants the authority to read Lauren's blogpost, but not to make changes to it.
- `day-in-park-editor`, which grants the authority to modify the blogpost.

Lauren wants the feedback of her friend Robert, but wants to decide whether or not to make or accept any changes herself. She shares `day-in-park-post` with Robert. Robert is able to view the post by running:

```
REPL> display-post day-in-park-post
```

Which prints out:

```
A Day in the Park
=================
  By: Lauren Ipsdale

It was a good day to take a walk...
```

Robert tells Lauren that he likes the blogpost, but that "a fine day" might sound more pleasant than "a good day" for the article's opening, and that maybe the name of the post should be "A Morning in the Park". Robert, not having access to `day-in-park-editor`, cannot make the changes himself.

Lauren deliberates on this feedback and decides that she agrees with the suggestion to change "good" to "fine" but that she thinks her title is good as-is. Lauren makes the change:

```
REPL> $ day-in-park-editor 'update
```

14 This is not meant to be a "production-ready system", but an illustrative one. As one example limitation, the blog we will build is runtime-only and does not persist between processes to disk. However, the general ideas described are the foundation from which a more serious system could be built, and even persistence could be accomplished through the mechanisms described in Safe serialization and upgrade.

```
_____   #:body "It was a fine day to take a walk..."
```

3.3.1.1. Implementation

Since the "blog rendering" code is not essential to the demonstration of these security properties, that code is not shown in this section. However, it is available in Appendix: Utilities for rendering blog examples.

The final header we'll be using for this module will look like so:

```
define-module : goblins-blog
  #:use-module : goblins
  #:use-module : goblins actor-lib methods
  #:use-module : ice-9 match
  #:use-module : srfi srfi-9
  #:use-module : srfi srfi-9 gnu
  #:use-module : simple-sealers
  #:use-module : method-cell
  #:export (spawn-post-and-editor spawn-blog-and-admin
          new-spawn-blog-and-admin spawn-adminable-post-and-editor
          ^logger spawn-logged-revocable-proxy-pair
          spawn-post-guest-editor-and-reviewer
          display-post-content display-blog-header
          display-post display-blog)
```

The implementation of the post and editor pairs is fairly simple:

```
define* (spawn-post-and-editor #:key title author body)
  ;; The public blogpost
  define (^post _bcom)
    methods
      ;; fetches title, author, and body, tags with '*post* symbol
      (get-content)
      define data-triple            ; assign data-triple to
        $ editor 'get-data          ;   the current data
      cons '*post* data-triple      ; return tagged with '*post*

  ;; The editing interface
  define (^editor bcom title author body)
    methods
      ;; update method can take keyword arguments for
      ;; title, author, and body, but defaults to their current
      ;; definitions
      (update #:key (title title) (author author) (body body))
      bcom : ^editor bcom title author body
      ;; get the current values for title, author, body as a list
      (get-data)
      list title author body

  ;; spawn and return the post and editor
  define post : spawn ^post
  define editor : spawn ^editor title author body
  values post editor     ; multi-value return of post, editor
```

This procedure takes three optional keyword arguments, the initial title, author, and body of the post.[15] (If not supplied, they will default to `#f`, meaning "false".) It returns two values, the `post` (which is the object which represents the readable blogpost), and the `editor`, which allows for editing what viewers of the `post` see.

15 Guile's `define` does not support keyword arguments, but `define*` does.

In this system, the `editor` is the more powerful object. It contains two methods:

- `update`: Allows for changing the data associated with the post. The `bcom` operation calls `^editor` again, producing new behavior with the same `bcom` capability but updated (or not) versions of the `title`, `author`, and `body`.
- `get-data`: Retrieves the current title, author, and body associated with this post.

The `post` is considerably less powerful, and only has one method, `get-content`. Curiously, `get-content` is a thin wrapper around the `editor`'s `get-data`, merely tagging the returned data with the symbol `'*post*`.

3.3.1.2. Analysis

With ordinary Goblins programming and a safe language environment, Lauren is able to construct separate post and editor capabilities which refer to the same blogpost. Lauren is able to choose who she hands these out to. Since Lauren shares the `post` capability with Robert but not the `editor` capability, Robert is able to read the blogpost, but there is no way for him to change its contents.

All of this is accomplished without any attention by the underlying system to the identities of Lauren and Robert who are using the software, using ordinary reference passing behaviors. This is important, because in Capability security as ordinary programming we demonstrated that an identity-centric authority model is unsafe due to ambient authority and confused deputy problems. The solution we demonstrated of a capability security as ordinary argument passing extends into Goblins in a natural way. Since Goblins' object model is entirely built around behavior constructed from enclosed procedures, an object can only make use of the references to other objects it possesses in its scope.

We have also chosen in this example to have `post` be a comparatively thin object to `editor`, mostly proxying information which `editor` is in charge of, with a small type-tagging symbol added. This demonstrates how one less powerful object can achieve most of its functionality by attenuating a more powerful object.

3.3.2. A blog to collect posts

Of course, a blogpost on its own is not itself a blog or newspaper. Lauren wants a collection of updated posts, not just a singular entry. Time to make the blog!

Lauren invokes `spawn-blog-and-admin`:

```
REPL> define-values (maple-valley-blog maple-valley-admin)
_____   spawn-blog-and-admin "Maple Valley News"
```

`spawn-blog-and-admin` returns two capabilities. The first is for the blog itself, which Lauren has locally bound to the variable `maple-valley-blog`, and which only grants read access to the current set of posts. `maple-valley-admin` provides the ability to curate the set of posts itself. Lauren has a certain vision and standard of post quality she'd like to see held for Maple Valley News but would like it to be widely read, and thus she will share and encourage wide dissemination of the former capability but will more carefully guard the latter capability.

Since `maple-valley-blog` has just been initialized, it unsurprisingly reports having no posts:

```
REPL> $ maple-valley-blog 'get-posts
; => ()
```

Since Lauren is now happy with `day-in-park-post`, she can add it via `maple-valley-`

`admin`, and `maple-valley-blog` will now report the new post's addition:

```
REPL> $ maple-valley-admin 'add-post day-in-park-post
REPL> $ maple-valley-blog 'get-posts
; => (#<local-object ^post>)
```

The blog can now also be read with `display-blog`:

```
REPL> display-blog maple-valley-blog
```

Which prints the following:

```
**********************
** Maple Valley News **
**********************

A Day in the Park
=================
  By: Lauren Ipsdale

It was a fine day to take a walk...
```

Robert tells Lauren he'd love to make an article of his own, and Lauren says she'd love to read it and see about including it. Robert pens a new post:

```
;; Run by Robert:
REPL> define-values (spelling-bee-post spelling-bee-editor)
_____  spawn-post-and-editor
_____    #:title "Spelling Bee a Success"
_____    #:author "Robert Busyfellow"
_____    #:body "Maple Valley School held its annual spelling bee..."
```

Robert sends this to Lauren for review. Lauren says that it's good, but could use a catchier title. Robert's years of community newspaper reporting leaves him with exactly the right idea for a change:

```
;; Run by Robert:
REPL> $ spelling-bee-editor 'update
_____   #:title "Town Buzzing About Spelling Bee"
```

Lauren checks the post and decides it's ready to go. She adds it to the blog:

```
REPL> $ maple-valley-admin 'add-post spelling-bee-post
```

Now `maple-valley-blog` is starting to look like it's got some real content going!

```
REPL> display-blog maple-valley-blog
**********************
** Maple Valley News **
**********************

Town Buzzing About Spelling Bee
===============================
  By: Robert Busyfellow

Maple Valley School held its annual spelling bee...


A Day in the Park
=================
  By: Lauren Ipsdale
```

```
It was a fine day to take a walk...
```

3.3.2.1. Implementation

Here is the core implementation of spawn-blog-and-admin:

```
;; Blog main code
;; ==============

define (spawn-blog-and-admin title)
  define posts
    spawn ^cell '()

  define (^blog _bcom)
    methods
      (get-title)
      . title        ; return the title, as a value
      (get-posts)
      $ posts 'get   ; fetch and return the value of posts

  define (^admin bcom)
    methods
      (add-post post)
      define current-posts
        $ posts 'get
      define new-posts
        cons post current-posts  ; prepend post to current-posts
      $ posts 'set new-posts

  define blog : spawn ^blog
  define admin : spawn ^admin
  values blog admin
```

Here we see how lexical scope becomes a powerful feature for capability systems. `posts`, a cell which stores the current state of which articles are valid posts for this blog, is within the scope of the code for both `blog` and `admin`, which both utilize it within the scopes of their constructors `^blog` and `^admin` internally. However, while `blog` and `admin` are returned directly from `spawn-blog-and-admin`, `posts` never directly leaves the closure. Thus `posts` becomes a fully encapsulated coordination point between `blog` and `admin`.

3.3.2.2. Analysis

The similarity between the patterns of `spawn-post-and-editor` and `spawn-blog-and-admin` is mostly clear, but what is interesting is in where they differ. While both return two capabilities, one effectively for reading and one effectively for writing, `spawn-post-and-editor` accomplished its job by having posts mostly proxy a subset of behavior of editors. In `spawn-blog-and-admin`, the roles are completely separated, and instead the encapsulated object of `posts` serves as the intermediary data structure that the two other objects both use to coordinate reading current information (with `blog`) and writing current information (with `admin`).

3.3.3. Group-style editing

One implication from the way this code is currently written is that the blog is mostly a kind of aggregator of posts. While Lauren added Robert's post to Maple Valley News's collection of blogposts, since Robert did not share the edit capability with Lauren, Lauren cannot edit the post if

she discovers a problem.

This can be an acceptable design, but Lauren has decided that she would like to ensure that any posts that are on the blog are editable by her or any other admins she gives access to. She also does not want to have to keep track of which edit capability is associated with which post: if she is looking at a post and catches an error, she wants to be able to jump straight into correcting it. Lauren wants to make sure her blogging administration software helps her ensure she is only adding objects which uphold these properties.

Under this rearchitecture, the admin interface is directly involved in constructing new posts and editors:

```
REPL> define-values (bumpy-ride-post bumpy-ride-editor)
_____   spawn-adminable-post-and-editor
_____     . maple-valley-admin
_____     #:title "Main Street's Bumpy Ride"
_____     #:author "Lauren Ipsdale"
```

Using this approach, Lauren could edit `bumpy-ride-post` using `bumpy-ride-editor`, but she does not need to since she can also use `maple-valley-admin` to edit:

```
REPL> $ maple-valley-admin 'edit-post
_____   . bumpy-ride-post
_____   #:body "Anyone who's driven on main street recently..."
```

This new code also provides an assurance that any blogposts which are added are created through the internals of the code which runs "Maple Valley News". It will not be possible for any other object to spoof being a post which will not grant a user of `maple-valley-admin` the ability to edit the post and still be added to the blog.

3.3.3.1. Pre-Implementation: Sealers and unsealers

This example relies on a concept called "sealers and unsealers". *Sealers* and *unsealers* have an analogy with public key cryptography, where sealing resembles encryption, and unsealing resembles decryption. A third component, a *brand check predicate*, can check whether or not a sealed object was sealed by its corresponding sealer, and with a bit of work, we will show it can operate as the equivalent of signature verification. What is astounding is that all three of these operations can work without any cryptography at all, implemented purely in programming language abstractions. (The details of implementing sealers and unsealers can be seen in Appendix: Implementing sealers and unsealers.)

To make this clearer, let us imagine a scenario where we are sealing lunchtime meals using sealers and unsealers. Our rival, who wishes to sabotage us, does the same:

```
REPL> define-values (our-lunch-seal our-lunch-unseal our-can?)
_____   make-sealer-triplet
REPL> define-values (rival-lunch-seal rival-lunch-unseal rival-can?)
_____   make-sealer-triplet
```

We give our customer the unsealer, the delivery driver the brand predicate, and we keep the sealer privately to ourselves.

The contents of sealed cans are private:

```
REPL> our-lunch-seal 'fried-rice
; => #<seal>
```

Our customer wants some chickpea salad, so we seal some for them:

```
REPL> define chickpea-lunch
_____   our-lunch-seal 'chickpea-salad
```

Thankfully our truck driver is able to check that the food they are to deliver really is from us. (We have a reputation to uphold!)

```
REPL> our-can? chickpea-lunch
; => #t (true)
REPL> our-can?
_____   rival-lunch-seal 'melted-ice-cream
; => #f
```

And the customer is able to open it just fine:

```
REPL> our-lunch-unseal chickpea-lunch
; => 'chickpea-salad
```

Whew!

3.3.3.2. Implementation

We will have to re-architect our post/editor and blog/admin tooling to enable this new functionality, adding support for sealers and a few new methods.

Our new version of post/editor spawning will no longer be used directly by users, so we also update its name, adding a `-internal` suffix.

```
define* (spawn-post-and-editor-internal blog-sealer #:key title author body)
  ;; The public blogpost
  define (^post _bcom)
    methods
      ;; fetches title, author, and body, tags with '*post* symbol
      (get-content)
      define data-triple            ; assign data-triple to
        $ editor 'get-data          ;   the current data
      cons '*post* data-triple      ; return tagged with '*post*
      ;; *New*: get a sealed version of the editor from anywhere
      (get-sealed-editor)
      blog-sealer : list '*editor* editor
      ;; *New*: get a sealed version of self for self-attestation
      (get-sealed-self)
      blog-sealer : list '*post-self-proof* post

  ;; The editing interface
  define (^editor bcom title author body)
    methods
      (update #:key (title title) (author author) (body body))
      bcom : ^editor bcom title author body
      (get-data)
      list title author body

  ;; spawn and return the post and editor
  define post : spawn ^post
  define editor : spawn ^editor title author body
  values post editor
```

There are actually only three changes from our prior implementation, `spawn-post-and-editor`:

- This version takes one required argument, `blog-sealer`, which will be passed in by the admin object which creates the post/editor pair.

- We add two new methods to `post`:
  - `get-sealed-editor`: Uses `blog-sealer` to seal the corresponding `editor` object, allowing a relevant `admin` object to be able to unseal any post straight from the post itself (analogous to encryption). The `'*editor*` symbol is stored within the seal as a type tag indicating the *purpose* of the seal.
  - `get-sealed-self`: Uses `blog-sealer` to seal the post itself to attest to the `admin` that it was indeed created by the blog/admin code itself (analogous to a cryptographic signature). Like the previous method, it also stores a type tag within the seal indicating its purpose, here `'*post-self-proof*`.

We must also update our blog/admin spawning code so that it will be able to cooperate with the post/editor code we have just defined:

```
define (new-spawn-blog-and-admin title)
  ;; *New:* sealers / unsealers relevant to this blog
  define-values (blog-seal blog-unseal blog-sealed?)
    make-sealer-triplet

  define posts
    spawn ^cell '()

  define (^blog _bcom)
    methods
      (get-title)
      . title
      (get-posts)
      $ posts 'get

  define (^admin bcom)
    methods
      ;; *New:* A method to create posts specifically for this blog
      (new-post-and-editor #:key title author body)
      define-values (post editor)
        spawn-post-and-editor-internal
          . blog-seal
          #:title title
          #:author author
          #:body body
      list post editor

      ;; *Updated:* check that a post was made (and is updateable)
      ;;   by this blog
      (add-post post)
      ;; (This part is the same as in the last version)
      define current-posts
        $ posts 'get
      define new-posts
        cons post current-posts  ; prepend post to current-posts
      ;; *New*: Ensure this is a post from this blog
      ;; This is accomplished by asking the post to provide the sealed
      ;; version "of itself".  The `blog-unseal` method will throw an error
      ;; if it is sealed by anything other than `blog-seal
      define post-self-proof
        $ post 'get-sealed-self
      match : blog-unseal post-self-proof
        ('*post-self-proof* obj)  ; match against tagged proof
          unless : eq? obj post   ; equality check: same object?
            error "Self-proof not for this post"
```

```
      ;; Checks out, let's update the set of posts
      $ posts 'set new-posts

      ;; *New:* A method to edit any post associated with this blog
      (edit-post post #:rest args)
      define sealed-editor
        $ post 'get-sealed-editor
      define editor
        match : blog-unseal sealed-editor
          ('*editor* editor)  ; match against tagged editor
            . editor
      apply $ editor 'update args

  values
    spawn ^blog
    spawn ^admin
```

Here we see several new additions:

- The blog calls `make-sealer-triplet` to instantiate `blog-seal` (the sealer), `blog-unseal` (the unsealer), and `blog-sealed?` (the brand-check predicate).
- `^admin` receives three key changes:
  - New method: `new-post-and-editor` is used to create post/editor pairs by running `spawn-post-and-editor-internal` (which was defined by the previous code block).
  - Updated method: `add-post` now checks that this is a post made by the blog itself. This is accomplished by asking the post for its supplied self-proof. This self-proof is returned sealed and must be unsealed by `blog-unseal`, which will throw an exception if not sealed by `blog-seal`, ensuring this is a post created by (and thus editable in the future by) the blog. The unsealed value should be a list tagged with the purpose of `'*post-self-proof*` and the object to check, the latter of which should have the same identity (compared via the identity-comparison procedure `eq?`) as `post`.
  - New method: `edit-post` allows for editing a post even without access to its corresponding editor object. This is accomplished by calling the `'get-sealed-editor` method on a post. The admin interface uses the `blog-unsealer` to extract the type-tagged editor. It uses `apply` to take the remaining arguments passed into edit-post and passes them along to the unsealed editor.

Finally, this last bit is some convenience for consistency in our examples, since actors cannot return multiple values from their behavior

```
define (spawn-adminable-post-and-editor admin . args)
  define post-and-editor
    apply $ admin 'new-post-and-editor args
  match post-and-editor
    (post editor)          ; match against list of post and editor
      values post editor   ; return as values for consistency in examples
```

3.3.3.3. Analysis

An administrator encountering a blogpost which is worth editing will want to edit it immediately. In an access control list style system, the way to accomplish this would be to assign users to an "editor" group, but we are building a system which aims to avoid the security problems associated

with traditional access control list and related identity-centric authority systems.

Instead, we take an approach called *rights amplification*: a sealed capability is attached to the post, giving access to the more powerful editor object, but this object can only be used through the corresponding unsealer. The only object empowered to make use of the unsealer is the blog's admin object, and so only by going through the admin is editing from the post possible.

3.3.4. Revocation and accountability

Lauren decides that it may be time for her to not be the only person running things, but she wants to make sure that she can hold anyone she gives access to accountable for the decisions they make and, if something inappropriate happens, revoke that access.

Lauren realizes she can extend her system to accommodate this plan *without rewriting any of the existing code*. Instead she will define some new abstractions that compositionally extend the system that exists.

The first thing she will need is a logger.

```
REPL> define admin-log
_____   spawn ^logger
```

Robert has been a great collaborator and has expressed interest in helping run things. Lauren decides it's time to take him up on it.

Lauren uses a new utility, `spawn-logged-revocable-proxy-pair`, which can proxy any object and log actions associated with a username meaningful to Lauren:

```
REPL> define-values (admin-for-robert roberts-admin-revoked?)
_____   spawn-logged-revocable-proxy-pair
_____     . "Robert"            ; username Lauren holds responsible
_____     . maple-valley-admin  ; object to proxy
_____     . admin-log           ; log to write to
```

The first of the two returned capabilities, `admin-for-robert`, is the one she sends Robert. The second, `roberts-admin-revoked?`, is the cell which defaults to false, but Lauren can set to be true at any time, at which point messages from Robert will no longer pass through.

Robert thanks Lauren for the capability and soon decides that Lauren's post would be better with a different title:

```
REPL> <- admin-for-robert 'edit-post bumpy-ride-post
_____    #:title "Main Street Takes Some Bumps"
```

Later, Lauren suddenly notices with irritation that her blogpost isn't named what she remembered it being. She checks the log:

```
REPL> $ admin-log 'get-log
; => ((*entry*
;      user "Robert"
;      object #<local-object ^admin>
;      args (edit-post #<local-object ^post>
;            #:title "Main Street Takes Some Bumps")))
```

Lauren decides that Robert shouldn't be editing her or anyone else's posts on the blog until they've had a serious conversation.

```
REPL> $ roberts-admin-revoked? 'set #t
```

Robert tries to make another edit to the blogpost and notices that it didn't go through. He sees a

frustrated message in his inbox from Lauren and apologizes. The two of them agree on what the proper etiquette for editing someone else's post should be in the future and Lauren feels satisfied enough to renew Robert's access.

```
REPL> $ roberts-admin-revoked? 'set #f
```

3.3.4.1. Implementation

The logger should look fairly familiar by now:

```
define (^logger _bcom)
  define log
    spawn ^cell '()   ; log starts out as the empty list

  methods
    ;; Add an entry to the log of:
    ;;  - the username accessing the log
    ;;  - the object they were accessing
    ;;  - the arguments they passed in
    (append-to-log username object args)
      define new-log-entry
      list '*entry* 'user username 'object object 'args args
      define current-log
      $ log 'get
      define new-log
      cons new-log-entry current-log  ; prepend new-log-entry
      $ log 'set new-log

    (get-log)
      $ log 'get
```

The revocable proxy pair takes the associated username, object to proxy, and log to write to:

```
define (spawn-logged-revocable-proxy-pair username object log)
  ;; The cell which keeps track of whether or not the proxy user's
  ;; access is revoked.
  define revoked?
    spawn ^cell #f

  ;; The proxy which both logs and forwards arguments (if not revoked)
  define (^proxy _bcom)
    lambda args
      ;; check if access has been revoked
      when ($ revoked? 'get)
      error "Access revoked!"
      ;; If not, first send a message to log the access
      $ log 'append-to-log username object args
      ;; Then proxy the invocation to the object asynchronously
      apply $ object args

  define proxy
    spawn ^proxy

  values proxy revoked?
```

It returns two cells, the proxy, and the cell which is used to control whether or not access is revoked.

3.3.4.2. Analysis

Since Robert is never given access to the admin object directly, he has to operate using the `admin-`

`for-robert` object which Lauren gives him. This object reports Robert's actions to a log which Lauren controls and will only operate if Lauren decides not to flip the `revoked?` cell to be true. Lauren is able to resume access through the capability should she so choose by flipping the `revoked?` cell's value back to false.

Nothing is preventing Robert from sharing `admin-for-robert` with anyone else, but Lauren will hold Robert accountable for any actions taken with the `admin-for-robert` capability. This is a feature, and we will see it extended in the next section.

3.3.5. Guest post with review

Some time has passed and Maple Valley News is doing well. Robert and Lauren have been knocking out a lot of well celebrated articles covering their community. Lauren is busy figuring out next steps for the newspaper, but Robert is exhausted and needs to go on the vacation he has long promised his family they would take. But Robert has an idea for a guest post article that could be published in his absence without having to interrupt Lauren.

Robert has a friend who works at the local school, Maple Valley Elementary, and has told Robert about how a young student named Matilda Sample won a distinguished prize in the regional science fair, assisted with the mentorship of her science teacher Mx. Beaker. Robert thinks this would be a great idea for a story. Robert asks if Matilda would be willing to write a story about her experience and whether Mx. Beaker would be willing to review and determine if and when the article would be good enough to publish.

Everyone agrees, so Robert sets everything up. Robert runs the following:

```
;; Robert's interactions
REPL> define-values (science-fair-post science-fair-editor science-fair-
reviewer)
_____   spawn-post-guest-editor-and-reviewer "Matilda Sample" admin-for-robert
```

Robert now sends out a message to Matilda and another to Mx. Beaker with the capabilities they will need:

- `science-fair-post` is given to both Matilda and Mx. Beaker and allows either of them to read the current state of the post.
- `science-fair-editor` is given to Matilda only; this allows Matilda to edit and author the post. This capability only allows Matilda to change the title and body, but *not* the author (which Robert has already set to "Matilda Sample"). However, this capability *does not* give Matilda the authority to publish the post.
- `science-fair-reviewer` is given to Mx. Beaker; this allows Mx. Beaker to approve and publish the post (but also will prevent future edits in the process). However, this capability *does not* give Mx. Beaker the authority to modify the post.

Matilda begins writing the post:

```
;; Matilda's interactions
REPL> <- science-fair-editor 'set-body
_____    . "My name is Matilda and I am twelve. I won the science fair..."
```

Matilda asks Mx. Beaker if it's good enough to publish. Mx. Beaker tells Matilda, not yet! The post needs a title, and Matilda's teacher explains how to make the post tell a more engaging and personal narrative.

Matilda updates the title and rewrites the body:

```
;; Matilda's interactions
REPL> <- science-fair-editor 'set-title
_____    . "Winning the Middle School Science Fair: A Personal Account"
REPL> <- science-fair-editor 'set-body
_____    . "At twelve years old, winning the local science fair has been..."
```

After another prompt for review, Mx. Beaker decides that the post now looks great and will be a great representation of both Matilda and the school. Feeling proud of their student, Mx. Beaker presses approve:

```
;; Teacher's interactions
REPL> <- science-fair-reviewer 'approve
```

And the post goes live!

Readers of the blog will see the new post, and will be able to share it widely:

```
;; Widely runnable (by blog readers and those they share it with)
REPL> display-blog maple-valley-blog
```

Robert, still on vacation, receives a message from Lauren. "Hey, I just saw that blogpost go live! It looks great! But I see in the log that you posted it… didn't you promise you weren't going to work while you left on vacation?"

Robert smiles and types up a response. "Funny thing that… nice things can happen that serve multiple peoples' interests, and if you think far ahead enough, sometimes when you aren't even around…"

3.3.5.1. Implementation

Robert's clever solution is custom code. He was able to write it without even having to change anything about how the core blogging code worked:

```
;;; Guest post with review
;;; ======================

;; The restricted-editor user can only change the title and body, but
;; not their name.
;; They cannot conspire with their teacher to be someone else on the
;; newspaper.
;;
;; The teacher cannot do anything but approve the student's post to
;; go live.  They cannot change the student's choice of language,
;; only ask them to change it before approval.

define (spawn-post-guest-editor-and-reviewer author blog-admin)
  define-values (post editor)
    spawn-adminable-post-and-editor
      . blog-admin
      #:author author

  define submitted-already?
    spawn ^cell #f

  define (ensure-not-submitted)
    when : $ submitted-already? 'get
      error "Already submitted!"

  define (^reviewer _bcom)
    methods
      (approve)
```

```
    ensure-not-submitted
    $ blog-admin 'add-post post
    $ submitted-already? 'set #t

define (^restricted-editor _bcom)
  methods
    (set-title new-title)
    ensure-not-submitted
    $ editor 'update #:title new-title
    (set-body new-body)
    ensure-not-submitted
    $ editor 'update #:body new-body

define reviewer : spawn ^reviewer
define restricted-editor : spawn ^restricted-editor
values post restricted-editor reviewer
```

This uses patterns we have already seen. The code above has an encapsulated `post` and `editor` but only exports `post` directly. The post is already configured at `spawn-post-and-editor` time with the relevant author. `restricted-editor` is configured to allow changing the title and the body, but not the author.

Once `reviewer`'s `'approve` method is called, the encapsulated `blog-admin` will be invoked to add the post to the blog. This also flips the encapsulated `submitted-already?` cell will be flipped to true. At this point, `reviewer` and `restricted-editor` will be revoked, throwing an error if someone tries to use them.

3.3.5.2. Analysis

While we have seen variants of all the techniques shown in this example, the astounding thing is that the way they are arranged permits cooperation between multiple parties:

- Lauren wishes to hold Robert responsible for any updates to the blog Robert makes. Since Robert uses his admin capability, he is still held accountable for whatever actions are taken.
- Robert wishes to have interesting new content added to the blog while both he and Lauren are unavailable to actively participate. By bringing multiple stakeholders to the table, he feels confident that quality both he and Lauren would feel comfortable with will be maintained.
- Matilda wants to be able to talk about her experiences, and wants to be able to tell them in her own words and not be misrepresented. She is willing to receive mentorship from her teacher and apply this feedback to produce an improved article, even though she wants to write the article herself.
- Mx. Beaker wants a quality article that reflects well on their school, their student, and themselves. However, Mx. Beaker can only approve the post, meaning that they must convince Matilda of any changes they would like made.
- Robert is assured that neither Mx. Beaker nor Lauren can post the article on the blog falsely claiming authorship from someone else.
- When Lauren and Robert return from being busy, they will both still be able to use their admin capabilities to edit the post should they feel it appropriate (though Lauren will still hold Robert accountable for his changes).

But the most astounding thing of all: this entire arrangement was possible without changing any of the pre-existing blog code. Robert was able to encode an arrangement that kept everyone's interests

in play, without having to even be present!

3.3.6. Lessons learned

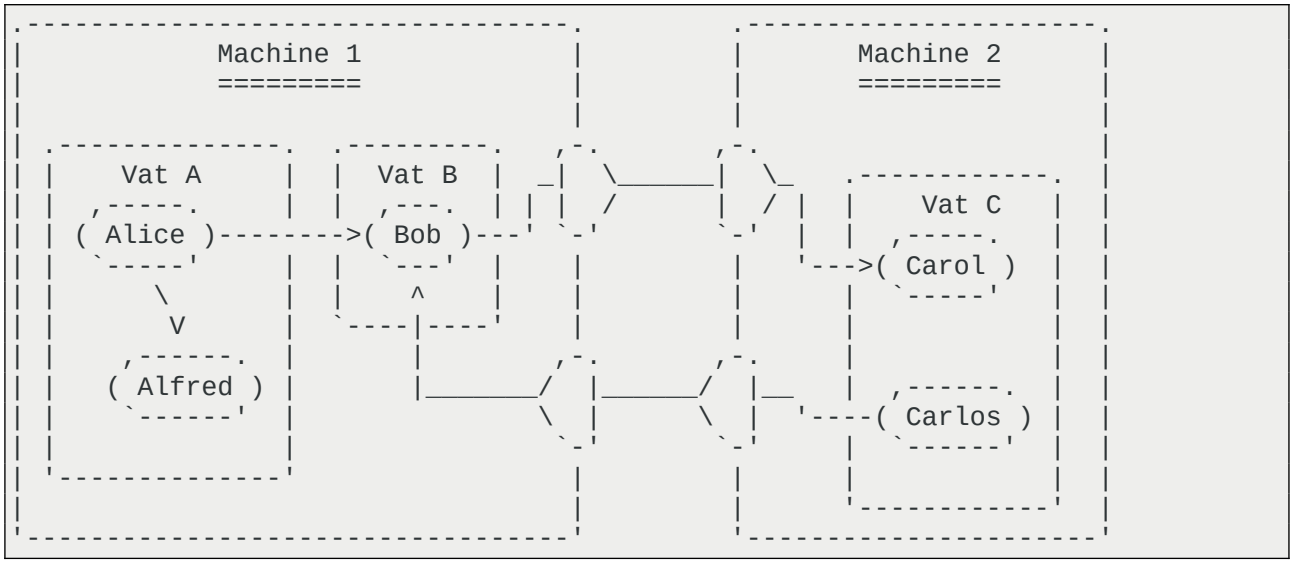
We have skipped over some important steps intentionally: we have not shown how to set up the network connections between parties, we have not shown how to produce capability references which can be passed along offline, and we have not shown how these posts might be persisted to long-term storage or upgraded.

Nonetheless, we have seen evidence of some powerful things:

- Distributed objects defined by *behavior* and bound together through *capabilities* are sufficient to represent sophisticated and useful social interactions between multiple parties.
- Our authorization mechanism relies on capability references and follows the "if you don't have it, you can't use it" philosophy. Sharing access remains as simple as reference passing. Everything is understandable as ordinary code.
- Despite the fact that our authorization mechanism itself is ambivalent about the identity of its participants, we are able to encode attribution of actions into the system. Combined with a revocation mechanism, this permits accountability. We have also added broader group-style access to administer certain objects. All this without needing an access control list mechanism or the inherent ambient authority and confused deputy risks associated with such an approach.
- We are able to encode rich, multi-stakeholder arrangements that benefit everyone. Through the guest post with a review example, we have demonstrated that special use cases like this can occur layered on top of an existing system rather than requiring a messy rewrite of existing behavior.

3.4. Spritely Goblins as a society of networked objects

The relationship between Spritely Goblins' abstracted distributed object layers can be understood visually. Consider the following relationship graph representing communicating objects:



In the above diagram, we see:

- Two machines (`Machine 1` and `Machine 2`, running separately from each other, but connected to each other over the network via OCapN and CapTP.

- `Vat A` and `Vat B` are event loops which live on `Machine 1`, and `Vat C` is an event loop which lives on `Machine 2`.
- The individual objects (represented by circles) live in *vats*, aka event loops which contain objects. `Alice` and `Alfred` live in `Vat A`, `Bob` lives in `Vat B`, and `Carol` and `Carlos` live on `Vat C`. (While we've given these objects human-like names, they're just Goblins objects.)
- The arrows between the objects represent references these objects have to each other. `Alice` has references to both `Alfred` and `Bob`. `Bob` has a reference to `Carol`. `Carlos` has a reference to `Bob`.
- Two objects which are in the same vat are considered *near* each other, and thus can invoke each other synchronously, whereas any objects not in the same vat are considered *far* from each other. Any objects can invoke each other by asynchronous message passing… assuming they have a reference to each other.
- Not pictured: each vat has an *actormap*, an underlying *transactional* heap used for object communication. This is what permits *transactionality* and time travel. (*Actormaps* can also be used independently of vats for certain categories of applications.)

Another way to think about this is via the following abstraction nesting dolls:

```
(machine (vat (actormap {refr: object-behavior})))
```

- **Machines**, which are computers on the network, or more realistically, operating system processes, which contain…
- **Vats**, which are communicating event loops, which contain…
- **Actormaps**, transactional heaps, which contain…
- A mapping of **References** to **Object Behavior**.

3.5. The vat model of computation

Goblins follows what is called the *vat model* of computation. A *vat* is simply an event loop that manages a set of objects which are *near* to each other (and similarly, objects outside of a vat are *far* from each other).

Objects which are *near* can perform synchronous call-return invocations in a manner familiar to most sequential programming languages used by most programmers today. Aside from being a somewhat more convenient way to program, sequential invocation is desirable because of cheap *transactionality*, which we shall expand on more later. In Goblins, we use the `$` operator to perform synchronous operations.

Both *near* and *far* objects are able to invoke each other asynchronously using asynchronous message passing (in the same style as the *classic actor model*). It does not generally matter whether or not a *far* object is running within the same OS process or machine or one somewhere else on the network for most programming tasks; asynchronous message passing works the same either way. In Goblins, we use the `<-` operator to perform asynchronous operations.

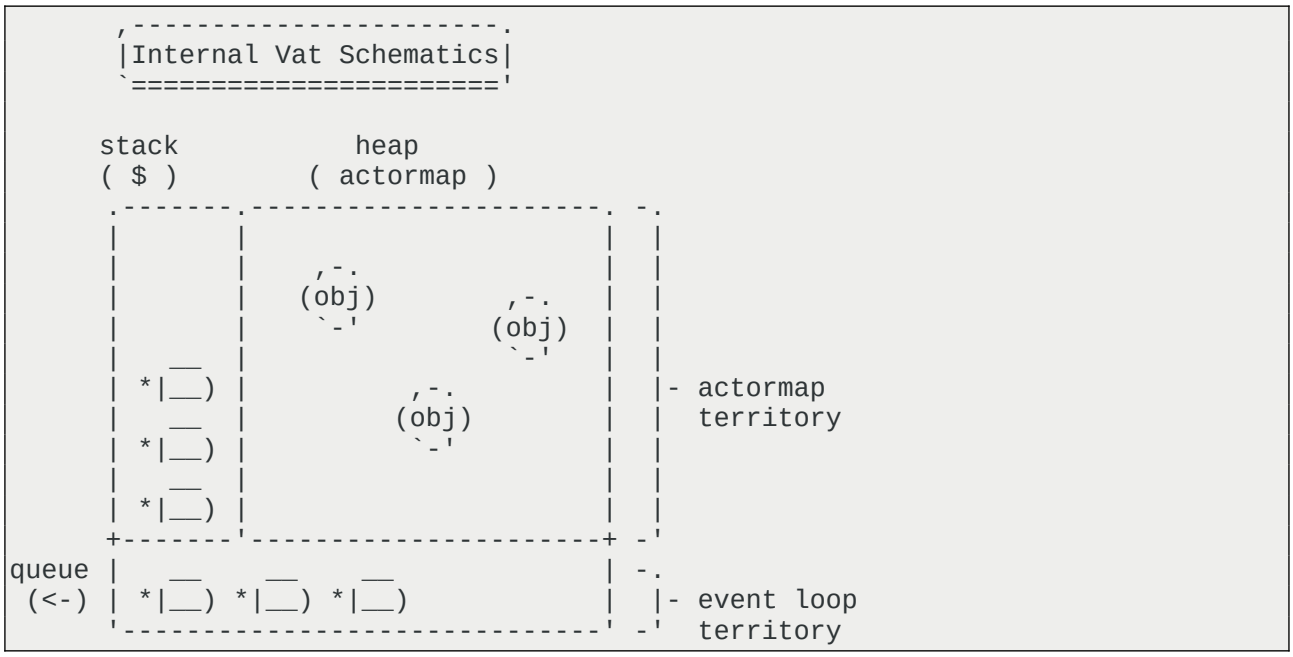
For both programmer convenience and for network efficiency, Goblins supports *promise pipelining*: messages can be sent to promises which have not yet been resolved, and will be forwarded to the target once the promise resolves. The sender of the message is handed back a promise to which it can supply callbacks, listening for the promise to be fulfilled with a value or broken (usually in case of an unexpected error).

As usual in the vat model of computation, individual message sends to a vat (event loop) are queued and then handled one *turn* at a time, akin to the way board game players take turns around a table (which is indeed the basis for the term *turn*).

The message, addressing a specific object, is passed to the recipient object's current behavior. This object may then invoke other *near* objects (residing within the same vat), which may themselves invoke other near objects in a synchronous and sequential call-return style familiar to most users of most contemporary programming languages. Any of these invoked objects may also change their state/behavior (behavior changes appear purely functional in Goblins; invocations of other actors do not), spawn new objects, invoke ordinary expressions from the host language, or send asynchronous messages to other objects (which are only sent if the *turn* completes successfully).

While the *vat model* of computation is not new (it originates in the E programming language and can trace some of its ideas back to E's predecessor Joule, and has since reappeared in systems such as Agoric's SwingSet kernel), Goblins brings some novel contributions to the table in terms of *transactionality* and time-travel debugging, enhancing an already powerful distributed programming paradigm.

3.6. Turns are cheap transactions



Special to Goblins is the transactional nature of *vat turns:* unhandled errors result in a *turn* being rolled back automatically (or more accurately, simply never being committed to the root transactional heap), preventing unintended data corruption. This cheap transactionality means that errors in Goblins are much less eventful and dangerous to deal with than in most asynchronous programming languages. Significantly less effort needs to be spent on cleanup when time is reverted to a point where a mess never occurred.[16]

16 It is well known that the introduction of time and the introduction of local state are the same, introducing both benefits and costs. *Purely functional* systems model local state without introducing *side effects* by using *monads*, which re-introduces the benefits of time without being locked into changes which have occured. In other words: functional programming with monads grants freedom from time. Monads are powerful and beautiful constructs but are notorious for being difficult to learn to use (though learning to use them sometimes becomes a programmer point of pride), introducing enormous amounts of explicit plumbing outward to the user, threaded manually through a user's code. Goblins' design can be perceived as having an *implicit monad* which grants the user the benefits of time-travel without the explicit plumbing, allowing the user to focus on the core object behavior aspects of their program. The ability to be productively oblivious to the above is a goal: most users will never even know or consider the idea

3.7. Time-travel distributed debugging

The same transactional-heap design of Goblins can be used for other purposes. A distributed debugger inspired by E's Causeway is planned, complete with message-tracing mechanisms. This will be even more powerful when combined with already-demonstrated time travel features,[17] allowing programmers to debug a program in the state of an error when it occured.

3.8. Safe serialization and upgrade

```
Do you, Programmer,
 take this Object to be part of the persistent state of your application,
 to have and to hold,
 through maintenance and iterations,
 for past and future versions,
 as long as the application shall live?

  —Arturo Bejar
```

Processes crash or close and must be resumed. Behavior changes and representations must change to accommodate such change. Goblins has an integrated serialization mechanism which simplifies serialization and upgrade.

The need for state persistence and upgrade is hardly unique to Goblins programs. Much of programming traditionally involves reading and writing the state of a program to a more persistent medium, generally files on a disk or some specialized database. Web applications in particular spend an enormous amount of effort moving between database representations and runtime behavior, but translating between runtime behavior and persistent state is typically disjoint and its solution space complicated.

Since Goblins' security model is encoded within the underlying runtime graph, manually scribing and restoring this structure would be a Sisyphean task in terms of labor and, should we naively trust objects' own self-descriptions, an entry point for vulnerability.

As an example, consider a multiplayer fantasy game might have to keep track of many rooms, the inhabitants of those rooms including various monsters and players, players' inventory, and many clever other objects and mechanisms which might even be defined while the game is running. Ad-hoc serialization of such a system would be too hard to keep our heads on straight about, and so we would like some way of having our system do the serialization of our process for us. Asking the objects to self-describe or manipulate the underlying database could also be dangerous, as objects could claim to have authority that they do not… for example, in our game, we would not want player-built objects to be able to claim or dispense in-game currency or grant themselves powers which they did not originally have on restoration.

One option would be to use an underlying language runtime serialization system (many Lisp and Smalltalk systems have supported this for decades). However, this is wasteful; most serialized systems can be restored from a recipe of their construction rather than their current state at a fraction of the storage cost. Furthermore, the structure of our objects will be subject to change over time, and language-based process persistence misses out an opportunity to treat restoration as an opportunity for upgrade.

that Goblins contains an *implicit monad* unless they enjoy reading footnotes of architectural papers.

17 One early demonstration of this idea was shown in the runs-in-your-terminal space shooter game Terminal Phase, built as a demo to show off Spritely Goblins. The entire core game was built before even considering that time travel would be an easy feature to add, and a time travel demonstration was added within less than three hours changing no core game code but merely wrapping the toplevel of the program; its design fell out naturally from what Goblins already provided in the way it was used.

Spritely Goblins' solution is a serialization mechanism which asks objects how they would like to be serialized, but only allows objects to provide self-portraits utilizing the permissions they already have.[18,19] Goblins' serializer starts with root objects and calls a special serializer method on each object, asking each object for its self-portrait. This serialization mechanism is *sealed* off from normal usage; only the serializer can *unseal* it, preventing objects from interrogating each other for information or capabilities they should not have access to.[20]

Since walking the entire object graph is expensive, we can take advantage of reading turn-transaction-delta information to only serialize objects which have changed, making our serialization system performant.

The system is restored by walking the graph in reverse and applying each self-portrait to its build recipe. Restoring an object ends up being a great time to run upgrade code and as we build out Goblins we plan to capture many upgrade patterns into a common library.

The serialized graph can be used for another purpose: we can use it to create a running visualization of a stored ocap system, further helping programmers debug systems and understand the authority graph of a running system.

3.9. Distributed behavior and why we need it

In general when we have spoken so far in this paper of distributed objects, we have been referring to objects with one specific "location". But many systems are actually more complicated than this. For example, Alisha and Ben might both be in the same chatroom and there may be a distinct address for Alisha and Ben's personas; if we ask whether or not Carol means the same Alisha as Ben, she should have no problem saying "yes, this is the same person", and this can be as simple as address comparison.[21] Alisha and Ben may have their own local representations of Carol with their own local behavior and state, particular to the interests, needs, or knowledge of Carol as she is known locally to each networked participant.

The Unum Pattern is a conceptual framework that encompasses the idea of a distributed abstract object with many different presences. One difference between the framing provided by the *unum pattern* and most other distributed pattern literature is that the *unum pattern* is particularly interested in *distributed behavior* rather than *distributed data*. Distributed data may be emergent from distributed behavior, but it is only one application. In the *unum pattern*, many different presences cooperate together performing different roles, sometimes even responding to messages in a manner semi-opaque to each other.

Consider a teacup sitting on a table in a virtual world. Where does it live? On the server? What about its representation in your client? What about the representation on another player's client?

18 The ideas for our serialization/upgrade mechanism stem from comments by Jonathan A. Rees about "uneval" and "unapply" and the E programming language's Safe Serialization Under Mutual Suspicion paper (along with discussions between Randy Farmer and Mark S. Miller while at Electric Communities which preceded this).

19 Originally we had built this system as a separate mechanism we called *Aurie*, symbolized by a character made out of fire which was continuously extinguished and re-awakened like a phoenix. However we discovered that many programs, and even many of the standard library pieces which Goblins ships with, were in want of such a system, so Aurie's flame became folded into Goblins itself.

20 This is a common ocap pattern called *rights amplification*, explored in Group-style editing.

21 Actually, saying that this is "as simple as address comparison" is the greatest misleading statement in this entire paper. Object identity through address comparison, frequently referred to as `EQ` based on the operator borrowed from Lisp systems, is one of the most complicated talks debated in the object capability security community. See also the erights.org pages on Object Sameness and the Grant Matcher Puzzle. These are just the tip of the iceberg of `EQ` discussion and debate in the ocap community, and it's no surprise why: when identity is handled *incorrectly* it can accidentally behave as a *Access Control List* (*ACL*) or inherit their problems of *ambient authority* and *confused deputies*. This is part of the value of finding patterns, to help prevent users from falling into these traps.

What about in your mind? While there is one *unum*, or "conceptual object", of the teacup, there are likely many *presences* representing it. Information and authority pertaining to the teacup may also be asymmetric;[22] you might know that the teacup has a secret note sealed inside it and I might not. While there may be one object which is the *canonical presence*, possibly serving as a source of shared identifier to refer to the object, the *canonical presence* is still a *presence*.[23]

Presences in Goblins typically correspond to Goblins objects.[24] The *unum pattern* is typically implemented via several messaging patterns: the reply pattern, the point-to-point pattern, the neighbor pattern, and the broadcast pattern. Keen observers might notice that a subset of the *unum pattern*, applied to data, is a publish-subscribe (PubSub) system, which is common in social media architecture design (ActivityPub is more or less a glorified data-centric publish-subscribe classic actor model implementation designed for social media on the web). For large-scale distribution of messages, the Amphitheater Pattern will be supported.

However, in recent times there have been advancements in convergent information architectures with research on conflict-free replicated data types. Goblins plans on implementing a standard library of CRDT patterns which can be thought of as a "unum construction kit".

4. OCapN: A Protocol for Secure, Distributed Systems

Here, in brief, we discuss *OCapN* (the *Object Capability Network*), which Spritely Goblins implements, and which we aim to eventually standardize. What *OCapN* provides is a set of layered abstractions so that very little code needs to be aware of "where" objects live for asynchronous programming, fully capable of functioning with no central authorities, even on *peer-to-peer* networks with the default assumption of hostile participants. While *OCapN* is already supported by Spritely Goblins, the protocol is general and could be broadly implemented across programming languages, providing interoperable networked cooperation.

The layers of OCapN are:

- **CapTP:** The *Capability Transport Protocol* (also known as *CapTP*) provides a distributed, secure networked object programming abstraction. *CapTP* provides familiar message passing patterns with no distinction between asynchronous programming against local vs remote objects and features:
  - **Distributed garbage collection:** Servers can cooperate to free resources which are no longer needed.
  - **Promise pipelining:** Massive parallelization and network optimization. Provides convenience of sequential programming without round trips.
- **Netlayers:** CapTP sits on top of the *netlayers* abstract interface, which allows for

22 Exploiting *asymmetric authority* is the very definition of the *confused deputy problem*. Its cause is usually emergent from ambient authority. Phishing attacks are an example of confused deputy problems where the confused deputy is a human being. Most object capability programming does not have confused deputy issues because to have a reference to a capability, in the general case, means to have authority to it. However, `EQ` and rights amplification (which bottoms out in a kind of `EQ`) both can re-introduce asymmetry, permitting confused deputies in careless designs, even to ocap systems. One might suggest removing identity comparison altogether from such systems, and for many ocap programs this is possible. However a *social system* is not very useful without identity, so we must develop patterns that treat identity with care.

23 The above explanation is modified directly from Chip Morningstar's explanation of the Unum. Chip Morningstar co-founded both Lucasfilms Habitat and Electric Communities (with EC Habitat), both of which are enormous influences on Spritely's design.

24 Outside of Goblins, presences still may exist; it is still acceptable to consider your conception of a teacup to be a presence. Barring significant advancements in biomechanical integration, presences in your mind of a teacup probably are not represented directly by a Goblins object.

establishing secure connections between two parties. The *netlayers* abstraction provides:

  - **Transport protocol agnosticism:** Multiple types of netlayers are supported. Fully *peer-to-peer* networks such as Tor Onion Services, I2P, and libp2p can work alongside more contemporary networks such as DNS + TLS. Even encrypted *sneakernets* are possible.
  - **Temporal connection abstraction:** Both live sessions for high-performance socketed connections and high-delay, intermittently offline/online store-and-forward systems are supported.
- **URI structure and certificates:** Entrance to the network must be bootstrapped and object locations identified; a unification of URI schemes provides the information an OCapN-aware language/library can use to engage connectivity. Certificates provide similar functionality but with different tradeoffs: less simplicity in sharing, but also less vulnerability to leakage.

5. Application safety, library safety, and beyond

> Users have faced an impossible choice: between the full authority to get your work done and destroy your machine or authority so puny that you can't do anything useful with it. And if you grant full authority you are *toast!* Object capabilities enable you at many different scales to create easy-to-understand secure cooperation.
>
> If your cooperation has no security you will quickly find that the number of people you dare to cooperate with is limited. Unless you have security, you can only cooperate with your closest friends. By making this cooperation secure, we enable you to cooperate with people whom you do not fully trust. So if you want to do cooperation, you do indeed care about security.
>
> – Marc Stiegler, From Desktop to Donuts: Object-Caps Across Scales

While all our examples in this paper follow object capability security discipline, we have hand-waved past one critical detail. Even if Goblins follows object capability security discipline, Goblins is implemented as a library. Goblins can provide capability security properties in the network through *OCapN*, but we would like more:

- We want to trust the security environment that Goblins itself runs on, so that our Goblins-enabled programs will not be subverted in the security properties they are designed to provide. In other words, we need a **trusted computing base**.
- We would also like to be able to preserve the safety of code which runs *on top of Goblins* (both externally potentially malicious or buggy vulnerable code, but even preserving the safety of our own code, to help reduce bugs which manifest as vulnerabilities), which means we need a **safe evaluation environment**.

There are many layers of a *trusted computing base*, and we would like to provide as many as we can:

- **User experience level safety:** The end user experiences of everyday users should uphold the users' intuitions of security through the interfaces they use for their work, entertainment, social communication, and community interactions. (This is the topic of our forthcoming paper, Spritely for Secure Applications and Communities.)
- **Network cooperation level safety:** We wish to be able to cooperate with objects hosted across the network and preserve capability passing semantics at the network abstraction

level. We should be able to cooperate with objects on another host, but another host should be able to hold no more dangerous authority over us than the capabilities which have been granted it (by us, or by those who have delegated capabilities to it). Thankfully, Goblins is able to provide this layer through *OCapN* already, so we can consider this part of our *trusted computing base* (assuming, of course, lower components have not been subverted).

- **Library level safety:** All modules are untrusted by default. Loading a module doesn't mean it can do dangerous things. Instead of libraries being able to "reach out" and grab access to whatever dangerous operations they would like (such as accessing the filesystem, the network, etc), libraries should have to be passed explicit capabilities to do these things, not unlike how we pass capabilities into the invocation of a function.
- **Language level safety:** Related to the above, we want our language to uphold the security properties we encode in our programs, and we want the runtime itself to be well programmed and to have good object capability enabling semantics. Generally, for a language to be an *object capability programming language*, it should uphold the following properties: no ambient authority, no global mutable state, lexical scoping with reference passing being the primary mechanism for capability transfer, and importing a library should not provide access to interesting authority.
- **Application level safety:** All programs are untrusted by default. Loading a program doesn't mean it can do dangerous things. Individual applications should be sandboxed to begin with no interesting authority, and we should have the ability to launch new sandboxed applications. Access to the filesystem, network, system clocks, etc should also be capabilities passed in at this layer.[25]
- **Operating system level safety:** The operating system itself should be programmed with *object capability security* in mind. It should have a secure and auditable kernel. Access to external devices should be contained and managed on a capability level.
- **Hardware level safety:** The hardware itself should not be a path to violating the integrity of our system, as free of side-channel attacks as possible, tamper-resistant, auditable and controllable by the end user, and understandable with well published specifications.
- **Supply chain level safety:** We should be able to be sure that hardware produced matches the hardware security specifications laid out, that the production facilities are auditable, and that backdoors are not inserted.
- **Cryptographic level safety:** We should have fundamental cryptographic operations which have understandable abstractions.
- **Physics and mathematics level safety:** We should be certain that the physics and the mathematics of our universe actually function in the manner described so that all of our abstractions are possible.[26]

This is a tall order (especially that last one). Listing these out can make the process of building a fully secure system feel like an impossible task. Thankfully, things are better than they appear: while layers lower on the stack are able to subvert the integrity of layers higher on the stack, at any

25 We should note that sandboxing alone is insufficient. Running in an enclosed environment where all available capabilities are defined at launch time is insufficient; this will result in too narrowly available a range of capabilities, and users will drive a sledgehammer through the walls by handing too-large of a bundle of capabilities by default. Instead, operating systems must provide the ability to "pass in" capabilities as a system is run, not only at initialization time.

26 If we are living in a simulation, we ask that those running simulation politely not tamper with the abstraction barriers we have come to rely on unless we are to be given access to the parent environment in which our simulation runs.

layer of operation we benefit from protection. For example, if a user is running a web browser in what we consider to be a generally insecure operating system, if the execution of untrusted code is contrained from accessing the user's file system, we have still protected the user from some levels of vulnerability.

Spritely, aiming to provide a *trusted computing base* which users can rely on, is interested in secure implementations of every one of these layers. However, for the purpose of upholding Goblins' abstractions most especially, the most obvious layer of importance is on the *library level safety* and *language level safety* layers. To this end, the choice of Guile for this task is not a coincidence: while more work needs to be done, Guile has the right fundamental operations of sandboxed evaluation which are needed to build a secure environment.[27] The demonstration of such an *object capability programming language* with Goblins running on top of it will be the focus of a future Spritely Institute paper.

6. Portable encrypted storage

> Every seller of cloud storage services will tell you that their service is “secure”. But what they mean by that is something fundamentally different from what we mean. What they mean by “secure” is that after you’ve given them the power to read and modify your data, they try really hard not to let this power be abused. This turns out to be difficult! Bugs, misconfigurations, or operator error can accidentally expose your data to another customer or to the public, or can corrupt your data. Criminals routinely gain illicit access to corporate servers. Even more insidious is the fact that the employees themselves sometimes violate customer privacy out of carelessness, avarice, or mere curiosity. The most conscientious of these service providers spend considerable effort and expense trying to mitigate these risks.
>
> What we mean by “security” is something different. *The service provider never has the ability to read or modify your data in the first place: never.*
>
> — The Tahoe-LAFS manual on "provider-independent security"

How do we keep information alive even when computers drop from the network? Is there a way to keep information alive and not beholden to the liveness of a particular hosting provider without sacrificing the privacy and security of our users? Can robust and private data storage be achieved in a way that upholds the same level of capability security properties we have demonstrated in this paper so far?

In Security as relationships between objects we provided an example of implementing a blog purely in terms of behavior. We handwaved past several details, mostly notably how to construct OCaPN *URIs* so that live connections to blogposts can be bootstrapped from out-of-band, how to persist the running object graph to long-term storage via safe serialization, how to encode a more sophisticated markup language (eg HTML or Markdown) to allow for rich document formatting, or any example of embedding (potentially large) static media within said documents.

Nonetheless, our blogpost resembles contemporary blogs served over HTTP in the following way: access to these documents requires a live reference to a particular entity on a particular machine and is retrieved via a live interaction over a live connection. While this was useful for demonstrating that a capability system with interesting interactions can be constructed out of a *behavior-oriented* system rather than a *data-oriented* system, the blogposts themselves are fundamentally *data-*

27 It should be seen as a good sign that the previously linked sandboxed evaluations in Guile page references A Security Kernel Based on the Lambda Calculus, which we have mentioned several times throughout this paper.

oriented and could be stored as useful portable documents.

Unfortunately, this means that an interesting document is subject to the bandwidth (and to a smaller degree, processing) availability and uptime of a single machine on the network. Hosting costs for producing a useful resource can grow, and usually fall on the shoulders of that particular resource. Should this machine no longer be available on the network, pointers to documents hosted by it can disappear. This is the general state of the web today, and is a major drive towards centralization and general bitrot of useful and historical information.

The solution to this problem is to support *portable encrypted storage*, which must fulfill the following properties:

1. Documents must be **content addressed** and **location agnostic**. In other words, the name of the particular resource is based on information stemming from the content itself rather than a particular network location. Generally this name is the hash of the corresponding document in the case of *immutable* documents and a public key (or hash thereof) in the case of *mutable* documents.
2. Both **immutable** and **mutable** documents must be supported, with the latter generally being built upon the former.
3. Documents must be **encrypted** such that the documents can be stored in locations that are oblivious to their actual contents. Only those possessing read capabilities should be able to access the documents' contents.
4. Documents should be **chunked** so that they are not vulnerable to *size-of-file attacks*.
5. Reading (and, in the case of mutable documents, writing) documents must be accessed through abstract **capabilities**.
6. Files must be *network agnostic*, meaning that they are not only *location agnostic* but agnostic even to a specific network structure. *peer-to-peer*, *client-to-server*, and *sneakernet* networks all should be supported with the same object *URIs* between them.

Many systems have been written which supply some of these properties.

IPFS is the most popular but does not provide the privacy and encryption requirements listed above, although it can be used as a foundation on which those layers are based. We have written our own toy examples that satisfy all of the above requirements with Magenc and Crystal, as well as an example applied to a social network with Golem. Freenet and Tahoe LAFS were the first systems coming close to fulfilling most (but not all) of the above requirements, and laid the foundations for understanding what these requirements are and how to fulfill them. Currently Encoding for Robust Immutable Storage (ERIS) and Distributed Mutable Containers (DMC) appear to be the most promising directions for fulfilling these requirements.

This paper is primarily designed to discuss *behavior-oriented* systems rather than *data-oriented* systems; Spritely Goblins does not itself implement a solution for *portable encrypted storage* as described above, but can be a good backend for a transport by which they may be distributed, and can compose nicely with the *distributed object programming* features that Goblins does provide. However, given that the purpose of this paper is to describe essential infrastructure, we believed it was important to demonstrate why in the long run *portable encrypted storage* will provide. Live distributed object programming without *portable encrypted storage* is capable in the short term of building full social network systems, but secure long-lived document storage is important to the preservation of the cultural artifacts we build together and to provide scalability friendly towards *peer-to-peer* networks without undue pressure towards centralization. Fuller expansion of this topic will be the subject of future papers.

7. Conclusions

Despite early ambitions of internet architecture, networked technologies of the last two decades have primarily been built by, and around the needs of, large and centralized institutions. Spritely's vision of re-architecting individual and community experiences on the internet requires a different approach where radically decentralized and participatory secure networked applications are the default result of programming.

Spritely Goblins meets these goals by building on established distributed programming lessons from the object capability community. Goblins further integrates these designs with theoretical approaches from the Lisp/Scheme and functional programming world, building a system that hybridizes actors and the lambda calculus. Many complicated considerations, otherwise relegated to the fringes of an explosion of domain specific languages and protocols, unify under a single model. While implemented on Scheme (for being a strong and natural fit), these ideas are written as a library general enough to be ported to most language environments with first class functions and lexical scoping.

The end result delivers great power to the user. Security analysis moves towards the intuitions of ordinary programming paradigms of reference passing. The vat model of computation synthesizes both synchronous programming against highly localized objects and asynchronous programming against objects which can live anywhere. Turn-based transactionality means that failures do not cause corruption of state in most circumstances. Time travel plus distributed debugging allows the user to more easily pin down problems and analyze them from the point of view of the system at the time where the errors occurred. An integrated safe serialization mechanism allows for objects to describe how they should be persisted using no more authority than that which they have been already granted and, upon being restored, also allows for the possibility of upgrade. And most importantly, Goblins' integration with OCapN (the Object Capability Network) and its implementation of CapTP (the Capability Transport Protocol) provides a unified distributed programming protocol with powerful features such as distributed debugging and efficient promise pipelining.

With all these features combined, Goblins provides a foundation where not only is building a future as robust as Spritely's vision requires possible, it is also comfortable and comprehensible.

8. Appendix: A small-ish scheme and wisp primer

This paper (and Goblins itself) was written in the Guile implementation of Scheme, itself a *dialect* of *Lisp*. (A Racket version also exists, but is not the subject of this paper. The two versions are very similar.) This choice was made for many reasons, most notably of which was the flexibility, fast iteration time, and extensibility of the underlying language.

The usual *surface syntax* for *Scheme* and other languages like it in the *Lisp* family is "parenthetical", like so:

```
(define (greet name)
  (string-append "Hello " name "!"))
```

In a *parenthetical* representation of *symbolic expressions* (also known as *s-experessions* or *sexps*), the parentheses show where the beginning and end of each "expression" are very clearly. The parenthetical syntax is also highly minimal, but is robust enough that any (really!) programming language can be represented using this kind of Lisp "parenthetical symbolic expression" syntax.

In general, Scheme/Lisp programmers' editors do the work of managing parentheses for them, and most code is read by indentation rather than by the parenthetical grouping. In other words, Lisp

programmers usually don't spend much time thinking about the parentheses at all. However, since most programming languages *don't* use syntax like this, experienced programmers sometimes find parenthetical Lisp style syntax intimidating. (In general, students totally new to programming have an easier time learning traditional Lisp syntax than seasoned programmers unfamiliar with Lisp do.)[28]

To keep *experienced programmers* from feeling intimidated, we've chosen to use Wisp, which looks like so:

```
define (greet name)
  string-append "Hello " name "!"
```

Compare to the previous `greet` example:

```
(define (greet name)
  (string-append "Hello " name "!"))
```

The structure of the language is the same in each of these, only the *surface syntax* has changed. Wisp derives its expression structure from indentation, but the end result is still symbolic expressions, just not expressed parenthetically. Wisp can be converted to parenthetical s-expressions, and vice versa.

We will return to an overview of Wisp's syntax transformation rules, but first let us get an overview of Scheme itself using its parenthetical syntax, and then look at how to convert between the two. This will give us a more precise vision of the language.

8.1. A brief-ish Scheme tutorial

The following is somewhere between a brief and comprehensive overview of Scheme. The further we go in the tutorial, the more advanced topics become. A shallow read of the following text is sufficient to read this paper in general, but the enthusiastic reader will gain much by reading the entire thing.

8.1.1. Hello Scheme!

Here's the familiar "hello world", written in Scheme:

```
(display "Hello world!\n")
```

This prints "Hello world!" to the screen. (The `"\n"` represents a "newline", like if you pressed enter after typing some text in a word processor.)

If you are familiar with other programming languages, this might look a little bit familiar and a little bit different. In most other programming languages, this might look like:

```
display("Hello world!\n")
```

In this sense, calling functions in Scheme (and other Lisps like it) is not too different than other languages, except that the function name goes inside the parentheses.

28 We've found that in running workshops introducing programming, students learning programming for the first time don't find Lisp syntax intimidating once they start programming, but experienced programmers do because Lisp's syntax looks alien at first sight if you know most other languages. We have even found that in teaching both Scheme (through Racket) and Python in parallel, many students with no programming background whatsoever (the workshops were aimed at students with a humanities background) expressed a strong preference for parenthetical Lisp syntax because of its clarity and found it easier to write and debug given appropriate editor support (Racket makes this easy with its newcomer-friendly IDE, DrRacket). For more about this phenomenon, see the talk Lisp, but Beautiful; Lisp for Everyone.

8.1.2. Basic types, a few small functions

Unlike in some other languages, math expressions like `+` and `-` are prefix functions just like any other function, and so they go first:

```
(+ 1 2)         ; => 3
(/ 10 2)        ; => 5
(/ 2 3)         ; => 2/3
```

Most of these can accept multiple arguments:

```
(+ 1 8 10)   ; equivalent to "1 + 8 + 10" in infix notation
```

Procedures can also be nested, and we can use the "substitution method" to see how they simplify:

```
(* (- 8 (/ 30 5)) 21)   ; beginning expression
(* (- 8 6) 21)          ; simplify: (/ 30 5) => 6
(* 2 21)                ; simplify: (- 8 6)  => 2
42                      ; simplify: (* 2 21) => 42
```

A variety of types are supported. For example, here are some math types:

```
42          ; integer
98.6        ; floating point
2/3         ; fractions, or "rational" numbers
-42         ; these can all also be negative
```

Since Scheme supports both "exact" numbers like integers and fractions, and does not have any restriction on number size, it is very good for more precise scientific and mathematical computing. The floating point representation is considered "inexact", and throws away precision for speed.

Here are some more types:

```
'foo                          ; symbol
'(1 2 3)                      ; a list (of numbers, in this case)
(lambda (x) (* x 2))          ; procedure (we'll come back to this)
'(lambda (x) (* x 2))         ; a list of lists, symbols, and numbers
#t                            ; boolean representing "true"
#f                            ; boolean representing "false"
"Pangalactic Gargleblaster"   ; string (text)
```

Symbols are maybe the strangest type if you've come from non-Lisp programming languages (with some exceptions). While symbols look kind of like strings, they represent something more programmatic. (In *Goblins*' `methods` syntax, we use symbols to represent method names.) Curiously, if a Lisp expression itself is quoted with `'`, as in the quoted `lambda` expression above, the symbols inside are also automatically quoted.

We will devote some time to discussing lists in Lists and "cons". The combination of lists and symbols is featured very prominently in many Lisps, including Scheme, because they lie at the heart of Lisp's extensibility: code which can write code. We will see how to take advantage of this power in On the extensibility of Scheme (and Lisps in general).

8.1.3. Variables and procedures

We can assign values to variables using `define`:

```
REPL> (define name "Jane")
REPL> (string-append "Hello " name "!")
; => "Hello Jane!"
```

However, if what follows `define` is wrapped in parentheses, Scheme interprets this as a procedure

definition:

```
(define (greet name)
  (string-append "Hello " name "!"))
```

Now that we have named this procedure we can invoke it:

```
REPL> (greet "Samantha")
; => "Hello Samantha!"
```

Note that *Scheme* has *implicit return*. By being the last expression in the procedure, the result of the `string-append` is automatically returned to its caller.

This second syntax for `define` is actually just *syntactic sugar*. These two definitions of `greet` are exactly the same:

```
(define (greet name)
  (string-append "Hello " name "!"))

(define greet
  (lambda (name)
    (string-append "Hello " name "!")))
```

`lambda` is the name for an "anonymous procedure" (ie, no name provided). While we have given this the name `greet`, the procedure would be usable without it:

```
REPL> ((lambda (name)
       (string-append "Hello " name "!"))
       "Horace")
; => "Hello Horace!"
```

There is also another way to name things aside from `define`, which is `let`, which allows for a sequence of bound variables and then a body which is evaluated with those bindings. `let` has the form:

```
(let ((<VARIABLE-NAME> <VALUE-EXPRESSION>) ...)
  <BODY> ...)
```

(The `...` in the above example represents that its previous expression can be repeated multiple times.)

Here is an example of `let` in use:

```
REPL> (let ((name "Horace"))
      (string-append "Hello " name "!"))
; => "Hello Horace!"
```

Clever readers may notice that this looks very similar to the previous example, and in fact, `let` is *syntax sugar* for a lambda which is immediately applied with arguments. The two previous code examples are fully equivalent:

```
REPL> (let ((name "Horace"))
      (string-append "Hello " name "!"))
; => "Hello Horace!"
REPL> ((lambda (name)
       (string-append "Hello " name "!"))
       "Horace")
; => "Hello Horace!"
```

`let*` is like `let`, but allows bindings to refer to previous bindings within the expression:[29]

29 There is also `letrec`, which allows for bindings to recursively refer to each other (or themselves). Both `let*` and

```
REPL> (let* ((name "Horace")
           (greeting
            (string-append "Hello " name "!\n")))
      (display greeting))   ; print greeting to screen
; prints: Hello Horace!
```

It is possible to manually apply a list of arguments to a procedure using `apply`. for example, to sum a list of numbers, we can use `apply` and `+` in combination:

```
REPL> (apply + '(1 2 5))
; => 8
```

As the inverse of this, it is possible to capture a variable-length set of arguments using "dot notation".[30] Here we show this off while also demonstrating Guile's `format` (which when called with `#f` as its first argument returns a formatted string as a value, and when called with `#t` as its first argument prints to the screen, the latter of which is what we want here):

```
REPL> (define (chatty-add chatty-name . nums)
      (format #t "<~a> If you add those together you get ~a!\n"
            chatty-name (apply + nums)))
REPL> (chatty-add "Chester" 2 4 8 6)
; Prints:
;   <Chester> If you add those together you get 20!
```

While not standard in Scheme, many Scheme implementations also support optional and keyword arguments. Guile implements this abstraction as `define*`:

```
REPL> (define* (shopkeeper thing-to-buy
                     #:optional (how-many 1)
                     (cost 20)
                     #:key (shopkeeper "Sammy")
                     (store "Plentiful Great Produce"))
      (format #t "You walk into ~a, grab something from the shelves,\n"
            store)
      (display "and walk up to the counter.\n\n")
      (format #t "~a looks at you and says, "
            shopkeeper)
      (format #t "'~a ~a, eh? That'll be ~a coins!\n"
            how-many thing-to-buy
            (* cost how-many)))
REPL> (shopkeeper "apples")
; Prints:
;   You walk into Plentiful Great Produce, grab something from the shelves,
;   and walk up to the counter.
;
;   Sammy looks at you and says, '1 apples, eh? That'll be 20 coins!'
REPL> (shopkeeper "bananas" 10 28)
; Prints:
;   You walk into Plentiful Great Produce, grab something from the shelves,
;   and walk up to the counter.
;
;   Sammy looks at you and says, '10 bananas, eh? That'll be 280 coins!'
REPL> (shopkeeper "screws" 3 2
              #:shopkeeper "Horace"
              #:store "Horace's Hardware")
; Prints:
```

`letrec` theoretically have some overhead, but a sufficiently advanced compiler can notice when either of these is equivalent to `let` and optimize appropriately. If Scheme were to be specified from scratch, it might be more sensible to just have one `let` which absorbs both `let*` and `letrec`. Alas, history is history.

30 This notation is directly related to the design of `cons` cells, which we will discuss more in Lists and "cons".

```
;   You walk into Horace's Hardware, grab something from the shelves,
;   and walk up to the counter.
;
;   Horace looks at you and says, '3 screws, eh? That'll be 6 coins!'
```

Finally, Scheme's procedures can do something else interesting: they can return multiple values using… `values`! As a particularly silly example, perhaps we would like to compare what it's like to both add and multiply two numbers:

```
REPL> (define (add-and-multiply x y)
      (values (+ x y)
              (* x y)))
REPL> (add-and-multiply 2 8)
; => 10
; => 16
REPL> (define-values (added multiplied)
      (add-and-multiply 3 10))
REPL> added
; => 13
REPL> multiplied
; => 30
```

As you can see, we can capture said values with `define-values`, as shown above. (`let-values` and `call-with-values` can also be used, but that's enough new syntax for this section!)

8.1.4. Conditionals and predicates

Sometimes we would like to test whether or not something is true. For instance, we can see whether or not an object is a string by using the `string?`:[31]

```
REPL> (string? "apple")
; => #t
REPL> (string? 128)
; => #f
REPL> (string? 'apple)
; => #f
```

(Remember that `#t` represents "true" and `#f` represents "false".)

We can use this in combination with `if`, which has the form:

```
(if <TEST>
    <CONSEQUENT>
    [<ALTERNATE>])
```

(The square brackets around `<ALTERNATE>` means that it is optional.)[32]

31 Procedures which test for truth / falseness are called *predicates* in Scheme. This is a bit confusing given the more broad definition of predicates used across natural languages and mathematics where a *predicate* is something that demonstrates a relationship. Technically, a test that gives a boolean value does demonstrate a relationship related to that test, so this is not wrong, but it may be counter-intuitive depending on the reader's background.
Scheme *predicates* traditionally have a `?` suffix attached to them. The `?` suffix is conventionally pronounced "huh?", and thus "string-huh?". In some other Lisps, a `-p` suffix is used in the same way Scheme uses `?`.

32 In standard scheme, `<ALTERNATE>` is technically not required. However, there is a separate procedure named `when`, provided by many Schemes by default, which has the form:

```
(when <TEST>
  <BODY> ...)
```

In *Racket*, an `if` without `<ALTERNATE>` (called a "one legged `if`") is not allowed, and even outside of *Racket*, `when`

So, we could write a silly function that excitedly reports on whether or not an object is a string or not:

```
REPL> (define (string-enthusiast obj)
      (if (string? obj)
          "Oh my gosh you gave me A STRING!!!"
          "That WASN'T A STRING AT ALL!! MORE STRINGS PLEASE!"))
REPL> (string-enthusiast "carrot")
; => "Oh my gosh you gave me A STRING!!!"
REPL> (string-enthusiast 529)
; => "That WASN'T A STRING AT ALL!! MORE STRINGS PLEASE!"
```

As we can see, unlike in some other popular languages, `if` also returns the value of evaluating whichever branch is chosen based on <TEST>.

Scheme also ships with some mathematical comparison tests. > and < stand for "greater than" and "less than" respectively, and >= and <= stand for "greater than or equal to" and "less than or equal to", while = checks for numerical equality:[33]

```
REPL> (> 8 9)
; => #f
REPL> (< 8 9)
; => #t
REPL> (> 8 8)
; => #f
REPL> (>= 8 8)
; => #t
```

If we wanted to test for multiple possibilities, we could use nested `if` statements:

```
REPL> (define (goldilocks n smallest-ok biggest-ok)
      (if (< n smallest-ok)
          "Too small!"
          (if (> n biggest-ok)
```

is preferred in such a situation. This is also because in a *purely functional programming language*, there is no such thing as calling a conditional where one possibility returns nothing of interest. In other words, it only ever makes sense to use `when` (or a "one legged `if`") for a *side effect*. Distinguishing between these cases is thus useful for the reader to observe. We will revisit `when`, including how to write it ourselves, in On the extensibility of Scheme (and Lisps in general).

33 Admittedly, this is a place where Lisp's prefix notation falls short of an infix notation choice, since there is a visual notation of size inherent in angle-bracket notation of greater-than / less-than. Some Schemes support SRFI-105: Curly infix expressions which is a bit easier to read. Compare:

```
REPL> (> 8 9)
; => #f
REPL> (< 8 9)
; => #t
REPL> (> 8 8)
; => #f
REPL> (>= 8 8)
; => #t
```

vs:

```
REPL> {8 > 9}
; => #f
REPL> {8 < 9}
; => #t
REPL> {8 > 8}
; => #f
REPL> {8 >= 8}
; => #t
```

```
           "Too big!"
           "Just right!")))
REPL> (goldilocks 3 10 20)
; => "Too small!"
REPL> (goldilocks 33 10 20)
; => "Too big!"
REPL> (goldilocks 12 10 20)
; => "Just right!"
```

However, there is a much nicer syntax named `cond` which we can use instead which has the following form:[34]

```
(cond
 (<TEST>
  <THEN-BODY> ...) ...
 [(else <ELSE-BODY> ...)])
```

Compare how much nicer our `goldilocks` procedure looks with `cond` instead of nested `if` statements:

```
;; Nested "if" version
(define (goldilocks n smallest-ok biggest-ok)
  (if (< n smallest-ok)
      "Too small!"
      (if (> n biggest-ok)
        "Too big!"
        "Just right!")))

;; "cond" version
(define (goldilocks n smallest-ok biggest-ok)
  (cond
   ((< n smallest-ok)
    "Too small!")
   ((> n biggest-ok)
    "Too big!")
   (else
    "Just right!")))
```

Scheme also provides some different ways to compare whether or not two objects are the same thing. The shortest, simplest (but not comprehensive) summary of the zoo of equality predicates is that `equal?` compares based on content equivalence, whereas `eq?` compares based on object identity (as defined by the language's runtime).[35] For example, `list` constructs a fresh list with a new identity every time, so the following are `equal?` but not `eq?`:

```
REPL> (define a-list (list 1 2 3))
REPL> (define b-list (list 1 2 3))
REPL> (equal? a-list a-list)
; => #t
REPL> (eq? a-list a-list)
; => #t
REPL> (equal? a-list b-list)
; => #t
REPL> (eq? a-list b-list)
```

34 As we will see in On the extensibility of Scheme (and Lisps in general), Scheme permits us to implement new forms of syntax. A Scheme implementation only needs one primitive form of syntax, since `if` can be written as a simplified version of `cond`, and `cond` can be written as a nested series of `if` statements.

35 The most understated footnote in computer science appears in The Art of the Propagator by Gerald Jay Sussman and Alexey Radul, which simply says:

> Equality is a tough subject

```
; => #f
```

Finally, in Scheme, anything that's not `#f` is considered true. This is sometimes used with something like `member`, which looks for matching elements and returns the remaining list if anything is found, and `#f` otherwise:

```
REPL> (member 'b '(a b c))
; => (b c)
REPL> (member 'z '(a b c))
; => #f
REPL> (define (fruit-sleuth fruit basket)
      (if (member fruit basket)
          "Found the fruit you're looking for!"
          "No fruit found! Gadzooks!"))
REPL> (define fruit-basket '(apple banana citron))
REPL> (fruit-sleuth 'banana fruit-basket)
; => "Found the fruit you're looking for!"
REPL> (fruit-sleuth 'pineapple fruit-basket)
; => "No fruit found! Gadzooks!"
```

8.1.5. Lists and "cons"

```
"My other CAR is a CDR"
  — Bumper sticker of a Lisp enthusiast
```

For structured data, Scheme supports lists, which can contain any other type.[36] Here are two ways to write the same list:

```
REPL> (list 1 2 "cat" 33.8 'foo)
; => (1 2 "cat" 33.8 foo)
REPL> '(1 2 "cat" 33.8 foo)
; => (1 2 "cat" 33.8 foo)
```

One difference between the two above is that in the latter quoted example, the symbol "foo" did not need to be quoted, since the outer list's quoting implicitly quoted it.

There is a "special" list known as "the empty list", which is a list with no elements, simply designated `'()` (also known as *nil*, and which is the only object which will return `#t` in response to the predicate `null?` in standard Scheme). Lists in *Scheme* are actually "linked lists", which are combinations of pairs called "cons cells" that terminate in the empty list:

```
REPL> '()
; => ()
REPL> (cons 'a '())
; => (a)
REPL> (cons 'a (cons 'b (cons 'c '())))
; => (a b c)
```

The latter of which is equivalent to either:

```
REPL> (list 'a 'b 'c)
; => (a b c)
REPL> '(a b c)
; => (a b c)
```

36 *Scheme* also has built-in support for *vectors*, which are like lists but which provide the benefit of constant-time access, but are not as useful for functional programming since they cannot easily have new elements prepended to them. Many *Scheme* languages also support and provide other interesting data types, including hashmaps and user-defined records, an unusual application of which is demonstrated in Appendix: Implementing sealers and unsealers.

For very historical reasons,[37] accessing the first element of a cons cell is done with `car` and the second element of a cons cell with `cdr` (pronounced "could-er"):[38]

```
REPL> (car '(a b c))
; => a
REPL> (cdr '(a b c))
; => (b c)
REPL> (car (cdr '(a b c)))
; => b
```

The second member of `cons` does not have to be another cons cell or the empty list. If not, it is considered a "dotted list", and has an unusual-for-lisp infix syntax:

```
REPL> (cons 'a 'b)
; => (a . b)
```

Notice how this is structurally different from the following:

```
REPL> (cons 'a (cons 'b '()))
; => (a b)
```

It's easy to get caught up on piecing apart `cons` cells (arguably schemers do far too often, but `cons` is also elegantly powerful).[39]

In a sense, this subsection is a digression. We intentionally do not use `cons` too much in this paper, and we have entirely kept `car` and `cdr` out of the main text. This may lead to the question, why contain this subsection on lists at all?

The reason is that we are building up to something we will explore further shortly, the extensibility of Scheme. Scheme is written in its core data types, and is modifiable as such. We will get to this more shortly, but as an example, we can quote any expression, transforming code into data:

```
REPL> (+ 1 2 (- 8 4))
; => 7
REPL> '(+ 1 2 (- 8 4))
; => (+ 1 2 (- 8 4))
REPL> (let ((name "Horace"))
      (string-append "Hello " name "!"))
; => "Hello Horace!"
REPL> '(let ((name "Horace"))
```

37 The name `cons` sensibly refers to "constructing" a pair, but the names `car` and `cdr` are a fully historical detail of Lisp's first implementation, the former referring to "contents of the address register" and the latter the "contents of the decrement register". It's amazing how long terms stick around, for better or worse.

For some interesting Lisp history, see:

- History of Lisp by John McCarthy
- The Evolution of Lisp by Guy L. Steele and Richard P. Gabriel
- History of LISP by Paul McJones

38 Since `car` and `cdr` are such "historical details", it's tempting to try to replace them with better names. If one is just using lists, `first` and `rest` are very good aliases:

```
REPL> (first '(a b c))
; => a
REPL> (rest '(a b c))
; => (b c)
REPL> (first (rest '(a b c)))
; => b
```

However, in cons cells that are simply pairs like `(cons 'a 'b)`, this makes less sense… `rest` returns a single element, rather than a sequence. So it is, and the names `car` and `cdr` live on.

39 Much else can be said about `cons`, "the magnificent" (as well as how to develop an intuitive sense of recursion) read The Little Schemer.

```
      (string-append "Hello " name "!"))
; => (let ((name "Horace")) (string-append "Hello " name "!"))
```

This last example is especially curious: we finally see the reason for symbols in Scheme to be important, as the function and syntax names become captured as symbols upon being quoted. In this sense, Lisp (including Scheme) is written in Lisp: there is little distinction between the representation the programmer sees and the representation the compiler sees, as see in On the extensibility of Scheme (and Lisps in general).

By the way, the apostrophe quote is just a shorthand for `(quote <EXPR>)`:

```
;; these two are the same
'foo
(quote foo)

;; and these two are the same
(lambda (x) (* x 2))
(quote (lambda (x) (* x 2)))
```

Lists can also be used as an associative mapping between keys and values, called *alists* (association lists). A variety of procedures for convenient lookup exist, such as `assoc`, which returns the pair if found or `#f` if not:

```
REPL> (define animal-noises
      '((cat . meow)
        (dog . woof)
        (sheep . baa)))
REPL> (assoc 'cat animal-noises
; => (cat . meow)
REPL> (assoc 'alien animal-noises)
; => #f
```

Association lists are easy to implement, look nice enough in Scheme's printed representation, and are easy to use with functional programming. (Want to add more to an alist? Just cons on another cons cell!) This means they tend to be popular with schemers. However, they are not always efficient. While `assoc` is fine for small alists, an alist that is one thousand elements long will take one thousand steps to find a key-value pair buried at its bottom. Other datastructures, such as hashmaps which provide constant-time average lookups, are commonly provided in many Scheme implementations, and are sometimes a better choice.

Aside from quote, it is also possible to use quasiquote, which uses the backtick to begin a quasiquote, and the comma to unquote. In this way we can move quickly between the world of data and code. For example, using a somewhat apocryphal metric for converting cat years to human years:

```
REPL> (define (cat-years years)
      (cond
       ((<= years 1)       ; first year equivalent to 15
        (* years 15))
       ((<= years 2)
        (+ 15 (* 9 (- years 1))))   ;      second year 9
       (else
        (+ 24 (* 4 (- years 2))))))) ; years after that 4
REPL> (define (cat-entry name age)
      `(cat (name ,name)
            (age  ,age)
            (cat-years-age ,(cat-years age))))
REPL> (cat-entry "Missy Rose" 16)
; => (cat (name "Missy Rose")
```

```
;         (age 16)
;         (cat-years-age 80))
REPL> (cat-entry "Kelsey" 22)
; => (cat (name "Kelsey")
;         (age 21)
;         (cat-years-age 104))
```

Wow! Those are some old cats!

8.1.6. Closures

Recall our earlier definition and use of `goldilocks`:

```
REPL> (define (goldilocks n smallest-ok biggest-ok)
      (cond
       ((< n smallest-ok)
        "Too small!")
       ((> n biggest-ok)
        "Too big!")
       (else
        "Just right!")))
REPL> (goldilocks 3 10 20)
; => "Too small!"
REPL> (goldilocks 33 10 20)
; => "Too big!"
REPL> (goldilocks 12 10 20)
; => "Just right!"
```

Entering the same values for `smallest-ok` and `biggest-ok` over and over again is tedious. Goldilocks' range of preferences are unlikely to change from invocation to invocation. Is there a way we could produce a version of Goldilocks with a kind of memory so we only have to pass in `smallest-ok` and `biggest-ok` once but still test against multiple versions of `n`? Indeed there is… *closures* to the rescue!

```
(define (make-goldilocks smallest-ok biggest-ok)
  (define (goldilocks n)   ; make a procedure which encloses
    (cond                  ;  smallest-ok and biggest-ok so
     ((< n smallest-ok)    ;  that only the n argument needs
      "Too small!")        ;  to be passed in
     ((> n biggest-ok)
      "Too big!")
     (else
      "Just right!")))
  goldilocks)              ; return goldilocks procedure
```

We can now invoke `make-goldilocks`, which returns the *enclosed* `goldilocks` procedure.

```
REPL> (make-goldilocks 10 30)
; => #<procedure goldilocks (n)>
```

Now we can call the inner `goldilocks` over and over again.

```
REPL> (define goldi
      (make-goldilocks 10 30))
REPL> (goldi 7)
; => "Too small!"
REPL> (goldi 256)
; => "Too big!"
REPL> (goldi 22)
; => "Just right!"
```

The outer procedure "closes over" the inner procedure, giving it access to (and a memory of) `smallest-ok` and `biggest-ok`.

Notably, this is the same pattern *Goblins* uses to implement constructors for its objects: the outer procedure is the constructor, the inner procedure is the behavior of the object. (The primary difference is indeed that *Goblins* objects spawned with `spawn` get a *bcom* capability which they can use to change their behavior!)

Beautifully, we can also build our own cons cells out of pure abstraction using this same technique.

```
REPL> (define (abstract-cons car-data cdr-data)
      (lambda (method)
        (cond
         ((eq? method 'car)
          car-data)
         ((eq? method 'cdr)
          cdr-data)
         (else (error "Unknown method:" method)))))
REPL> (define our-cons (abstract-cons 'foo 'bar))
REPL> (our-cons 'car)
; => foo
REPL> (our-cons 'cdr)
; => bar
```

In this sense, closures are also datastructures built from the code flow of the program itself.

Closures are a property of *lexical scoping*. We take advantage of this in Goblins: the capabilities an object has access to is merely the capabilities it has within its behavior's scope.

8.1.7. Iteration and recursion

Much of programming involves sequences of operations, especially on datastructures which contain other information. One especially useful procedure for functional programming which operates on lists is `map`, which applies its first argument's procedure to each element in its series.

For example, `string-length` gives the number of characters which exist in a given string:

```
REPL> (string-length "cat")
; => 3
REPL> (string-length "gorilla")
; => 7
```

So, using `map`, we could easily construct a list representing the length of each of its strings:

```
REPL> (map string-length '("cat" "dog" "gorilla" "salamander"))
; => (3 3 7 10)
```

We could also supply a procedure we define:

```
REPL> (define (symbol-length sym)
      (string-length (symbol->string sym)))
REPL> (map symbol-length '(basil oregano parsley thyme))
; => (5 7 7 5)
```

In fact, there is no requirement that we name the procedure… we can use `lambda` to construct an anonymous procedure which we pass to `map` directly:

```
REPL> (map (lambda (str)
           (string-append "I just love "
                          (string-upcase str)
                          "!!!"))
```

```
        '("strawberries" "bananas" "grapes"))
; => ("I just love STRAWBERRIES!!!"
;     "I just love BANANAS!!!"
;     "I just love GRAPES!!!")
```

`map` performs some extra work by building up a list of results every time. But what if we wanted to simply display our love of some food to the screen using `display` and did not care about operating on the data any further? We could use `for-each`, which has the same structure as `map` but does not build a result:

```
REPL> (for-each (lambda (str)
             (display
              (string-append "I just love "
                             (string-upcase str)
                             "!!!\n")))
           '("strawberries" "bananas" "grapes"))
; prints:
;   I just love ICE CREAM!!!
;   I just love FUDGE!!!
;   I just love COKIES!!!
```

NOTE! The following text in this subsection, indeed in the rest of the Scheme tutorial, is beyond anything required to understand the main body of our paper "The Heart of Spritely"! However, it will significantly advance a newcomer's understanding of Scheme.

Scheme has the surprising property that iteration is actually defined in terms of recursion!

Here is what we mean. We could define our own version of `for-each`:

```
(define (for-each proc lst)
  (if (eq? lst '())  ; End of the list?
      'done          ; We're done, so simply return "done"
      (let ((item (car lst)))  ; Otherwise... let's fetch this item
     (proc item)            ; Call the procedure with this item
     (for-each proc (cdr lst)))))  ; Iterate with the remaining work
```

This calls `proc` successively with each item from `lst` until it runs out of items. If you have experience with other programming languages, your expectation would probably be that this design could accidentally "blow the stack". However, Scheme is smart: it sees that there is no more work left to be done within the current version of `for-each` once we reach the last line… in other words, where `for-each` calls itself is in the "tail position". Because of this, Scheme is able to skip allocating a new frame on the stack and "jump" back to the beginning of `for-each` again with the new variables allocated. This is called `tail call elimination` and all iteration facilities are actually defined this way in terms of recursion in Scheme.

It is also possible to build recursive procedures. The following (somewhat advanced, if you don't follow this it's ok) procedure builds a binary tree:

```
(define (build-tree depth)
  (if (= depth 0)
      '(0)
      (list depth
         (build-tree (- depth 1))
         (build-tree (- depth 1)))))

REPL> (build-tree 3)
; => (3 (2 (1 (0)
;            (0))
;          (1 (0)
;            (0)))
```

```
;       (2 (1 (0)
;             (0))
;          (1 (0)
;             (0))))
```

Or, better visualized:

```
       3
      / \
     /   \
    2     2
   / \   / \
  1  1   1  1
 /\  /\ /\  /\
0 0 0 0 0 0 0 0
```

However, unlike `for-each`, `build-tree` does *not* call itself in the tail position. There is no way to simply "jump" to the beginning of the procedure without allocating work to be done on the stack with the way this code is written: more work needs to be done, as cons sits waiting for its results. As such, unlike `for-each`, `build-tree` is recursive but not iterative.

Finally, come conveniences. Here are two variants on `let`, both useful for recursive and iterative procedures. The first is `letrec` which allows for procedures to call and refer to themselves or others defined by the `letrec`, regardless of definition ordering:

```
REPL> (letrec ((alice
            (lambda (first?)
              (report-status "Alice" first?)
              (if first? (bob #f))))
             (bob
            (lambda (first?)
              (report-status "Bob" first?)
              (if first? (alice #f))))
             (report-status
            (lambda (name first?)
              (display
               (string-append name " is "
                          (if first?
                              "first"
                              "second")
                          "!\n")))))
     (alice #t)
     (display "-----\n")
     (bob #t))
; prints:
;   Alice is first!
;   Bob is second!
;   -----
;   Bob is first!
;   Alice is second!
```

The second useful abstraction is the *named let* variant of `let`, where a looping name identifier appears as the first argument:

```
REPL> (let loop ((strings '("carrot" "potato" "pea" "celery"))
             (num-words 0)
             (num-chars 0))
     (if (eq? strings '())
         (format #f "We found ~a words and ~a chars!"
              num-words num-chars)
         (let* ((this-string (car strings))
```

```
              (rest-strings (cdr strings)))
            (loop rest-strings
                (+ num-words 1)
                (+ num-chars (string-length this-string))))))
; => "We found 4 words and 21 chars!"
```

What a *named let* does is define the named procedure (here named `loop`) and immediately invokes it with the initial bindings of the `let`. The procedure is available within the body of the `let` for convenient recursive (perhaps iterative) calls.

8.1.8. Mutation, assignment, and other kinds of side effects

This section is included for completeness. Notably, Goblins provides a different approach to much of this here which we will discuss towards the end.

Scheme ships with a way to reassign the current value of a variable using `set!`:

```
REPL> (define chest 'sword)
REPL> chest
; => sword
REPL> (set! chest 'gold)
REPL> chest
; => gold
```

This can even be combined with the techniques shown in Closures. For instance, here's an example of an object that counts down from an initial number `n` until it reaches its zero, and then always returns zero afterwards.

```
REPL> (define (make-countdown n)
      (lambda ()
        (define last-n n)
        (if (zero? n)
            0
            (begin
            (set! n (- n 1))
            last-n))))
REPL> (define cdown (make-countdown 3))
REPL> (cdown)
; => 3
REPL> (cdown)
; => 2
REPL> (cdown)
; => 1
REPL> (cdown)
; => 0
REPL> (cdown)
; => 0
```

There are several interesting things about this example:

- We have introduced time and change into our computations. Before the introduction of side effects such as assignment, calling a procedure with the same arguments will always produce the same result. But in the above example, `cdown` changes its response over time (even without being passed any arguments on invocation).
- Since we want to show the initial number the first time the procedure is called, we have to capture `last-n` *before* using `set!` to change `n`. If we accidentally reverse this order, we will introduce a bug where `cdown` would have started with `2` instead of `3` in the example above.

- Here we also see an interesting new piece of syntax: `begin`. `begin` executes several expressions in sequence, returning the value of the last expression.

This last one is interesting. Prior to introducing effects (such as the assignment shown above, displaying to the screen, logging to a file or database, etc), there is never any reason for `begin`. To understand this, recall the *substitution method* demonstrated at the beginning of this tutorial:

```
(* (- 8 (/ 30 5)) 21)   ; beginning expression
(* (- 8 6) 21)          ; simplify: (/ 30 5) => 6
(* 2 21)                ; simplify: (- 8 6)  => 2
42                      ; simplify: (* 2 21) => 42
```

Before effects, every procedure invoked is to compute a new part of the program. But since each branch of `if` only evaluates one expression, we must provide a way to sequence the *alternate* clause so that we can both `set!` and then return a value.[40] In other words, a *purely functional* program is really built to take a series of inputs and precisely compute a value, the same value, every time. This is a clean set of substitutions all the way up and down the evaluation. (In other worlds, before introducing time, we will have programs which are fully *deterministic*.)

However, by introducing *mutation* and *side effects*, we have introduced a powerful, but dangerous, new construct into our program: time. Our programs are no longer *purely functional*, time has made them *imperative*: do this, then do that. Time is change, and change requires sequences of events, not mere substitutions. And time means that the same programs and procedures run with the same inputs will not always produce the same outputs. We have traded a timeless world for one that changes.

Despite the caution, change can be desirable. We live in a world with time and change, and so too often do our programs. Scheme has a (somewhat inconsistent) naming convention for observing time and change: the addition of a `!` suffix, as we have seen with `set!`. The `!` can be seen as a kind of warning, as if the user is shouting about the possibility of mutation. (However, the runtime of Scheme provides no guarantee that the presence or absence of this suffix says anything about mutation whatsoever.)

However, `set!` is not the only form of change and mutation available in standard (and nonstandard) Scheme.[41] Another example is mutable vectors and `vector-set!`:

```
REPL> (define vec (vector 'a 'b 'c))
REPL> vec
; => #(a b c)
REPL> (vector-ref vec 1)
; => b
REPL> (vector-set! vec 1 'boop)
REPL> (vector-ref vec 1)
; => boop
REPL> vec
; => #(a boop c)
```

Both of these examples resemble mutation. However, we have already seen a different form of side effects in this tutorial, namely `display`, which writes to the screen. In fact, `display` itself builds on the idea of *ports*, which are mechanisms in Scheme for reading and writing from and to input and output devices.

40 Notably, `cond` does permit multiple expressions in its `<THEN-BODY>` / `<ELSE-BODY>` sections, but we can think of this as `cond` being written to contain `begin`.

41 The ambient availability of `set!` creates problems for Scheme programs which should be confined. The interested reader should observe the alternative approach in the *W7* variant of Scheme from A Security Kernel Based on the Lambda Calculus.

All of these carry the same challenges of `set!`. Put simply, the introduction of ambient time makes our programs less timeless. However, if turns out that we cannot remove *all* time and change from our computers, as illustrated in this nested set of quotes:

> As Simon Peyton Jones, a well-known functional programmer, likes to say, "All you can do without side effects is push a button and watch the box get hot for a while." (Which isn't technically true, since even the box getting hot is a side effect.)
>
> — From *Land of Lisp* by Conrad Barski, M.D.

As Simon and Conrad point out, the challenge with functional programming is that even though side effects can be dangerous, they are in a sense all the user really cares about. At some point, in order for a computer to be useful, input must be read from the user and output must be given back, and these are inherently side-effectful. Even using the radiant heat of a busy computer to warm your house is a side effect. At some point, we must both enter and leave the realm of pure mathlandia.

Functional programmers sometimes solve this with a clever trick called *monads*. Monads will not be covered in this tutorial, but they can be thought of as a clever and explicit form of handling time by threading a bundle of state through an otherwise stateless program. This provides enormous power: time exists, but in a deterministic manner: the programmer becomes a time lord. However, they come at some cost, by exposing plumbing outward to the programmer.

Goblins takes an alternate approach, as discussed in Turns are cheap transactions and Time-travel distributed debugging. By capturing the nature of change within turns, the programmer gains the ability to traverse time. With *language level safety* features, as discussed in Application safety, library safety, and beyond, fully deterministic and contained execution can be guaranteed. All this can be done by abstracting the details of managing change such that the user need think of them; the Goblins core kernel can take care of this for the user. This will be expanded upon in detail within a future paper.

8.1.9. On the extensibility of Scheme (and Lisps in general)

Let's say we'd like some new syntax. For instance, maybe we want to run multiple pieces of code in sequence when a condition is met. We could write:

```
(if (our-test)
    (begin
      (do-thing-1)
      (do-thing-2)))
```

But this is kind of ugly. What if we created some new syntax specifically for this purpose?

```
(when (our-test)
  (do-thing-1)
  (do-thing-2))
```

`when` cannot be built as a function because we do not want to execute `(do-thing-1)` or `(do-thing-2)` unless `(our-test)` passes. We need new syntax.

Could we build the new syntax ourselves? Remembering that we can "write Lisp in Lisp", the answer seems to be yes:

```
REPL> (define (when test . body)
        `(if ,test
             ,(cons 'begin body)))
REPL> (when '(our-test)
```

```
     '(do-thing-1)
     '(do-thing-2))
; => (if (our-test)
;         (begin
;           (do-thing-1)
;           (do-thing-2)))
```

This does build out the appropriate syntax! And it does demonstrate that our claim that Lisp can "write code which writes code" is indeed true.[42]

However, there are two obvious problems with this first attempt:

- We had to quote each argument passed to `build-when`. This is annoying to do.
- `build-when` does not actually run its code, it just returns the *quoted structure* that the code should expand to.

However, with just one tweak our procedure can be turned into a "macro": a special kind of procedure used by the compiler to expand code. Here is all we need to do:

```
(define-macro (when test . body)
  `(if ,test
       ,(cons 'begin body)))
```

All we needed to do was rename `define` to `define-macro`! Now Scheme knows it should use this for code expansion. This allows us to define new kinds of syntax forms.

`define-macro` shows very clearly what macros in Lisp and Scheme do: they operate on structure. Manually building up a list structure like this is how macros in Common Lisp work. However, this is not the general way to write macros in Scheme. Scheme macros look very similar though:

```
(define-syntax-rule (when test body ...)
  (if test
      (begin body ...)))
```

`define-syntax-rule` uses *pattern matching* to implement macros. The first argument to `define-syntax-rule` describes the pattern which the user will enter, and the second describes the template which will be expanded.[43] We can also notice that `body ...` appears in both the

42 Since Lisp's *abstract syntax tree* is written in same datastructures used by its users for programming and is visually represented in its *surface syntax* in those same structures, it is called *homoiconic*. *Homoiconicity* is one of the most distinguishing pieces of Lisp, leading to much of its extensibility.

43 Actually, `define-syntax-rule` is itself sugar. The following are equivalent:

```
(define-syntax-rule (when test body ...)
  (if test
      (begin body ...)))

(define-syntax when
  (syntax-rules ()
    ((when test body ...)
     (if test
       (begin body ...)))))
```

Indeed, we can build `define-syntax-rule` out of `define-syntax` and `syntax-rules`:

```
(define-syntax define-syntax-rule
  (syntax-rules ()
    ((define-syntax-rule (id pattern ...) template)
     (define-syntax id
       (syntax-rules ()
       ((id pattern ...)
```

pattern and the template; the `...` ellipsis in the pattern represents that multiple expressions will be captured from the user's input and the `...` in the template indicates where the repeating should occur. We can see that we do not need to manually quote things using this mechanism; Scheme cleverly takes care of it for us.

Ultimately, the Scheme version of syntax definitions is less obvious as to how it works under the hood than the `define-macro` version is. However, there is an issue that arrives in syntax transformation systems called *hygiene*: that a syntax form / macro not introduce unexpected temporary identifiers into the body of the form it expands into. We will not get into the debate in this primer, but both Common Lisp and Scheme's macros have significant tradeoffs, with Scheme being much more likely to be properly "hygienic", easier to write for simple syntax forms, but harder to write for more complicated ones, and less obvious as to how they work under the hood. For this reason, even though you will likely never use the `define-macro` approach in Scheme, it is a useful way to understand the idea behind "code that writes code".

Now that we know how to produce new syntax the Scheme way, let's see if we can make our life more convenient than before. Let's revisit our use of `for-each` from earlier:

```
REPL> (for-each (lambda (str)
             (display
              (string-append "I just love "
                             (string-upcase str)
                             "!!!\n")))
           '("strawberries" "bananas" "grapes"))
; prints:
;   I just love ICE CREAM!!!
;   I just love FUDGE!!!
;   I just love COKIES!!!
```

This works, but it is also unnecessarily tedious. That `lambda` is an unnecessary piece of detail! A small new syntax definition lets us clean things up:

```
(define-syntax-rule (for (item lst) body ...)
  (for-each (lambda (item)
              body ...)
            lst))
```

Let's give it a try:

```
REPL> (for (str '("strawberries" "bananas" "grapes"))
      (display
       (string-append "I just love "
                      (string-upcase str)
                      "!!!\n")))
; prints:
;   I just love STRAWBERRIES!!!
;   I just love BANANAS!!!
;   I just love GRAPES!!!
```

It works! This is much easier to read.[44]

```
        template))))))
```

There is a zoo of other syntax transformation syntax forms available in most Schemes, and many of them vary across Scheme implementations, though `define-syntax` and `syntax-rules` are part of the Scheme standard.

44 A fun exercise for the reader: try to implement a C-style for loop!
Here's the C version:

```
// C's version of for:
//   for ( init; condition; increment ) {
```

We need not stop here. The `methods` feature in Spritely Goblins is an example of a macro. Here is a simplified version:

```
(define-syntax-rule (methods ((method-id method-args ...)
                              body ...) ...)
  (lambda (method . args)
    (letrec ((method-id
              (lambda (method-args ...)
              body ...)) ...)
      (cond
       ((eq? method (quote method-id))
      (apply method-id args)) ...
       (else
      (error "No such method:" method))))))
```

We can both see here simultaneously how expressive Scheme style pattern matching examples are, but also how with multiple layers of ellipses (the `...`), it can be a bit challenging to see how the code expander is figuring out how to unpack things.

But let's not worry about that for now, and instead show an example of usage:

```
REPL> (define (make-enemy name hp)
      (methods
       ((get-name)
        name)
       ((damage-me weapon hp-lost)
        (cond
         ((dead?)
          (format #t "Poor ~a is already dead!\n" name))
         (else
          (set! hp (- hp hp-lost))
          (format #t "You attack ~a, doing ~a damage!\n"
                  name hp-lost))))
       ((dead?)
        (<= hp 0))))
REPL> (define hobgob
      (make-enemy "Hobgoblin" 25))
REPL> (hobgob 'get-name)
; => "Hobgoblin"
REPL> (hobgob 'dead?)
; => #f
REPL> (hobgob 'damage-me "club" 10)
; prints: You attack Hobgoblin, doing 10 damage!
REPL> (hobgob 'damage-me "sword" 20)
; prints: You attack Hobgoblin, doing 20 damage!
REPL> (hobgob 'damage-me "pickle" 2)
; prints: Poor Hobgoblin is already dead!
REPL> (hobgob 'dead?)
; => #t
```

We can go further. We can extend Scheme to include logic programming, we can add pattern

```
//      statement(s);
//   }
for (i = 0; i < 10; i = i + 2) {
  printf("i is: %d\n", i);
}
```

Try to make the following work:

```
(for ((i 0) (< i 10) (+ i 2))
  (display (string-append "i is: " (number->string i) "\n")))
```

matching, etc etc etc. Indeed, we will use a pattern matching system included in Guile's standard library in the next subsection.

Because of the syntactic extensibility of Lisp/Scheme, advanced programming language features can be implemented as libraries rather than as entirely separate sub-languages. Multiple problem domains can be combined into one system. For this reason, we say that languages in the Lisp language support *composable domain specific languages*.

It is also liberating. In other programming languages, users must pray at the altar of the programming language implementers for features to show up in the next official language release, features which would be only a few small and simple lines of code in the hand of a Lisp/Scheme user.

This is true power. But there is more. In the next section we will unlock Scheme itself, allowing us to configure and experiment with its underlying mechanisms, in a surprisingly compact amount of code.

8.1.10. Scheme in Scheme

Here is a working implementation of Scheme written in Scheme:

```
(use-modules (ice-9 match))

(define (env-lookup env name)
  (match (assoc name env)
    ((_key . val)
     val)
    (_
     (error "Variable unbound:" name))))

(define (extend-env env names vals)
  (if (eq? names '())
      env
      (cons (cons (car names) (car vals))
          (extend-env env (cdr names) (cdr vals)))))

(define (evaluate expr env)
  (match expr
    ;; Support builtin types
    ((or #t #f (? number?))
     expr)
    ;; Quoting
    (('quote quoted-expr)
     quoted-expr)
    ;; Variable lookup
    ((? symbol? name)
     (env-lookup env name))
    ;; Conditionals
    (('if test consequent alternate)
     (if (evaluate test env)
       (evaluate consequent env)
       (evaluate alternate env)))
    ;; Lambdas (Procedures)
    (('lambda (args ...) body)
     (lambda (. vals)
       (evaluate body (extend-env env args vals))))
    ;; Procedure Invocation (Application)
    ((proc-expr arg-exprs ...)
     (apply (evaluate proc-expr env)
```

```
          (map (lambda (arg-expr)
                 (evaluate arg-expr env))
               arg-exprs)))))
```

Without comments, blank lines, and the pattern matching import at the top (not necessary, but convenient), this is a mere 30 lines of code. This evaluator, while bare bones, is complete enough to be able to compute anything we can imagine. (You could even write another similar Scheme evaluator on top of this one!)[45]

Our `evaluator` takes two arguments, a Scheme expression `expr` and an environment `env`. Scheme's lispy structure is of great benefit here, since as we have learned we can easily quote entire sections of code. (Indeed, that is exactly what we are going to do.) The `env` of the second argument is an association list mapping symbols for names and their associated procedures.

Seeing is believing. Let's do some simple arithmetic, passing in some procedures to the default environment which can do some math:

```
(define math-env
  `((+ . ,+)
    (- . ,-)
    (* . ,*)
    (/ . ,/)))
```

As we can see, the first "substitution method" example we wrote works just fine using this environment:

```
REPL> (evaluate '(* (- 8 (/ 30 5)) 21)
           math-env)
; => 42
```

What do you know, that's the same answer we got in our own program!

We can also make a lambda and apply it. Let's make one that can perform square roots:

```
REPL> (evaluate '((lambda (x)
                    (* x x))
                  4)
           math-env)
; => 16
```

Nice, works perfectly.

Let's do something more advanced. Supplying only two operators, `+` and `-`, we are able to compute the Fibonacci sequence:

```
REPL> (define fib-program
      '((lambda (prog arg)   ; boot
          (prog prog arg))
        (lambda (fib n)      ; main program
          (if (= n 0)
            0
            (if (= n 1)
                1
                (+ (fib fib (+ n -1))
                   (fib fib (+ n -2))))))
        10))                 ; argument
REPL> (define fib-env
      `((+ . ,+)
```

45 This idea of implementing a language on top of a similar host language is called a "metacircular evaluator". It is popular amongst computer science researchers as a way to explore variants of programming language design without needing to reinvent the entire system.

```
       (= . ,=)))
REPL> (evaluate fib-program fib-env)
; => 55
```

This seems like magic. But it works! The evaluator really is performing the underlying computation, using merely addition (on both positive and negative numbers) and numeric equality check procedures, which we have provided.

The main program needs to be able to call itself, so the first procedure (labeled `boot`) takes a program and an argument and invokes the procedure with itself and that argument.[46] The second procedure (labeled `main program`) takes itself as the argument `fib` (supplied by our boot procedure) as well as an argument of `n` (also supplied by the boot procedure)… and it works! Our evaluator recursively builds up the Fibonacci sequence.

Our evaluator can also be easily understood. Let us break it down section by section.

```
(define (env-lookup env name)
  (match (assoc name env)
    ((_key . val)
     val)
    (_
     (error "Variable unbound:" name))))
```

This one is easy. We are defining environments as association lists, so all `env-lookup` does is search for a matching name in the list. Newer additions will be found first, meaning that the same name defined in a deeper scope will *shadow* the parent scope. This can be seen by usage:

```
REPL> (env-lookup '((foo . newer-foo)
                    (bar . bar)
                    (foo . older-foo))
                  'foo)
; => 'newer-foo
```

The next one is a utility:

```
(define (extend-env env names vals)
  (if (eq? names '())
      env
      (cons (cons (car names) (car vals))
            (extend-env env (cdr names) (cdr vals)))))
```

`extend-env` takes an environment and a list of names and a parallel list of values. This is a convenience which we will use in procedure definitions later. Once again, easily understood by usage:

```
REPL> (extend-env '((foo . foo-val))
                  '(bar quux)
                  '(bar-val quux-val))
; => ((bar . bar-val)
;     (quux . quux-val)
```

46 This is a sophisticated example to demonstrate, with an interesting challenge to it: Fibonacci, as we have implemented it, is self-recursive! But self-recursion usually involves a feature like `letrec`, which we have not provided. To get around this, the main program (the second procedure) is passed to itself via the first procedure. Hence the comment calling the first procedure "boot".

This is kind of a cheap version of the Y combinator's bootstrapping technique. (No, not the company that starts startups, but now you can understand how that organization got its name.) The Y combinator performs the same trick but is more general. In many ways, evaluators like the one we have written have a lot in common with Y. The Why of Y is a lovely and concise article on the Y combinator and how one could derive it from a practical need, similar to the one we have demonstrated above. The Little Schemer also ends its lovely journey with writing a metacircular evaluator, similar to the one above, and explores how one might derive Y.

```
;     (foo . foo-val))
```

And now we are onto the evaluator. The shell of `evaluate` looks like so:

```
(define (evaluate expr env)
  (match expr
    (<MATCH-PATTERN>
     <MATCH-BODY> ...) ...))
```

`evaluate` takes two arguments:

- `expr`: the expression to evaluate
- `env`: the environment in which we will evaluate the expression

For the body of `evaluate`, we are dispatching our behavior depending on which patterns match `expr`. We are using `match` from Guile's pattern matching syntax (which came from our module import at the top). The short of it is though that if a `<MATCH-PATTERN>` matches, we will then stop searching for matches and evaluate `<MATCH-BODY>` (possibly with bindings set up from the `<MATCH-PATTERN>`).

So, now all we need to do is look at each pattern we support. The first is easy:

```
;; Support builtin types
((or #t #f (? number?))
 expr)
```

The `or` says we can match any one of its contained patterns. The first two are literally the true and false values from Scheme itself. The parentheses starting with a `?` symbol indicates that we will try matching against a predicate, in this case `number?`. If any of these match, we simply return the very same `expr` we are matching against… borrowing booleans and numbers straight from the underlying Scheme implementation.

In other words, the above powers:

```
REPL> (evaluate #t '())
; => #t
REPL> (evaluate #f '())
; => #f
REPL> (evaluate 33 '())
; => 33
REPL> (evaluate -2/3 '())
; => -2/3
```

That was easy! The next one is also easy:

```
;; Quoting
(('quote quoted-expr)
 quoted-expr)
```

Recall that `'foo` is just shorthand for `(quote foo)`, and likewise `'(1 2 3)` is shorthand for `(quote (1 2 3))`. In this pattern, we look for anything matching a list starting with the `'quote` symbol and a second element which is the expression to be quoted.

In other words, the above powers:

```
REPL> (evaluate ''foo '())
; => foo
REPL> (evaluate ''(1 2 3) '())
; => (1 2 3)
REPL> (evaluate (quote (quote (1 2 3))) '())
```

```
; => (1 2 3)
```

Those last two are the same. Note that we quote twice: once for quoting the entire program to be run, and once within the quoted program to say we want to quote an expression.

So far so good. The next one is still quite easy:

```
;; Variable lookup
((? symbol? name)
 (env-lookup env name))
```

The `(? symbol? name)` part binds `name` to the matching component. (In this case, `name` will be bound to the same value as the `expr` matched against, but this improves readability a little.)

As for the body… why, this is quite simple! We have already reviewed how `env-lookup` works. In other words if we see a symbol (not a quoted one of course, that has already been handled), we look up its corresponding value in the environment.

In other words, the above powers:

```
REPL> (evaluate 'x '((x . 33)))
; => 33
```

However, it will also empower variable lookups we define through lambda applications:

```
REPL> (evaluate '((lambda (x) x) 33) '())
; => 33
```

Of course, we have not yet gotten to `lambda`! But we are nearly there.

The next one, conditionals, also turns out to be fairly easy:

```
;; Conditionals
(('if test consequent alternate)
 (if (evaluate test env)
     (evaluate consequent env)
     (evaluate alternate env)))
```

In other words, a list starting with the symbol `'if` will be matched, with the three sub-expressions following `'if` bound to the variables `test`, `consequent`, and `alternate` in the match body. We use the underlying Scheme `if`, and first evaluate `test` against the current environment `env` (notice the recursion!), and the host Scheme's `if` helps us whether to evaluate the `consequent` or `alternate` inside of `env`, again using `evaluate` recursively.

Okay, now it's time to build procedures. This one is a little bit more complicated, but ultimately not too complicated either:

```
;; Lambdas (Procedures)
(('lambda (args ...) body)
 (lambda (. vals)
   (evaluate body (extend-env env args vals))))
```

The pattern here looks for a list starting with `'lambda`, with the second list member being the set of arguments, with the body being captured as, well, `body`. We then return a procedure which is ready to be evaluated with the same number of arguments.[47] The inner body of the procedure we return recursively calls `evaluate` against the `body` expression of the lambda we are matching against, but with a newly extended environment, binding together the names within `args` and the

47 If a user invokes this procedure with the wrong number of arguments, it will cause an error, but not a particularly useful one. Try figuring out how to give it a better one. (Hint: if `args` and `vals` aren't the same `length`, something is wrong!)

`vals` from the procedure invocation.

In other words, the above powers:

```
REPL> ((evaluate '(lambda (x y) x) '())
       'first 'second)
; => first
REPL> ((evaluate '(lambda (x y) y) '())
       'first 'second)
; => second
```

There is only one more piece left… application!

```
;; Procedure Invocation (Application)
((proc-expr arg-exprs ...)
 (apply (evaluate proc-expr env)
     (map (lambda (arg-expr)
            (evaluate arg-expr env))
          arg-exprs)))
```

This is the general-purpose procedure application piece of the puzzle! At this point, the pattern will match any list with one or more arguments, determining that this must mean a procedure applied to arguments. We evaluate the `proc-expr`, representing the procedure to be evaluated, within the current arguments, calling `evaluate` recursively with the current environment, `env`. We also gather all the `arg-expr` argument expressions passed to this procedure by calling `evaluate` recursively on each with the current environment, `env`.

In other words, the above powers:

```
REPL> (evaluate '(* (- 8 (/ 30 5)) 21)
           math-env)
; => 42
```

And with all pieces combined, we have enough power not only to compute the Fibonacci sequence, but any computable problem imaginable!

To be fair, this does borrow a portion of Scheme's underlying power, but not as much as it may appear… certainly less than many languages implemented on top of other languages do (certainly, certainly far less than Clojure borrows from Java, for instance, or nearly any popular language borrows from C's standard library).[48] And it is not so complete as to implement any of the Scheme standards. But without too much extra work, we could get there, and it is enough for demonstration. But we also get to *choose* how much power we give the language, by modifying the initial environment the code evaluates in.

It is also a capability-secure language! Aside from going into an infinite loop and consuming too many resources in terms of memory or CPU power, there is nothing particularly dangerous this language can do. However, we can decide how much power we would like to give it. If we choose, we can provide an environment with mutable cells, or one with access to the filesystem. The choice

48 Notably, the power of `quote` has made it so that we could write an interpreter without the need to parse textual syntax: all we need to do is quote our datastructures! Indeed, most textbooks on programming language design overly focus on the textual parsing of programming languages, giving the illusion that important parts of the language structure are related to the *surface syntax*, but this is never essentially true. Every programming language paradigm imaginable is representable in a simple symbolic expression representation such as is used by Scheme. Code and data are not truly as far apart as they may appear.

Of course, if we wanted to make this into a useful programming language, we would want to be able to read programs from source files on disk. `read` comes standard with Scheme (and really nearly any Lisp), so in general this work has already been done for us. However, implementing `read` in Scheme is also easy, and can be done in about the same amount of code as we implemented `evaluate`.

is ours.

And with tiny tweaks, our evaluator can operate in different and marvelous ways. We can add new syntax. We can add new syntax to add new syntax (macros)! We can change evaluation order, we can add static type analysis, we can do many things.

We promised that you would have learned Scheme from this tutorial. If you have reached this point, you have reached much more: you are no longer just a user of scheme, but a builder of Scheme. The power is yours![49]

9. Appendix: Following the code examples

TODO

10. Appendix: Utilities for rendering blog examples

```
;; Blogpost rendering utilities
;; ============================
define (display-post-content post-content)
  match post-content
    ('*post* post-title post-author post-body)
      let*
      : title : or post-title "<<No Title>>"
        title-underline : make-string (string-length title) #\=
        author : or post-author "<<Anonymous>>"
        body : or post-body "<<Empty blogpost!>>"
     display
       format #f "~a\n~a\n  By: ~a\n\n~a\n"
            . title title-underline author body

define (display-blog-header blog-title)
  define header-len
    + 6 (string-length blog-title)
  define title-stars
    make-string header-len #\*
  display
    format #f "~a\n** ~a **\n~a\n"
         . title-stars blog-title title-stars
```

49 You have made it to not only the final section, but the final footnote! At this point we must assume that you have a real passion for learning about these kinds of things, so we will provide you with even more resources:

- For more on how evaluators, like the one we have written above, work as well as its history, see William Byrd's incredible talk: The Most Beautiful Program Ever Written.
- Structure and Interpretation of Computer Programs (also known as SICP or "the wizard book") contains expanded versions of nearly everything we have covered here, as well as how to build an event loop, simple constraint solvers, evaluators like this one, logic programming evaluators, and compilers to more efficient machine code. There is no better way to understand the nature of computing than to study SICP. A good way to learn it is to switch between reading the source text and watching the 1980s lectures. SICP is available as a printed book, as an "info" manual conveniently readable in Emacs, or in HTML, but if you are going to read SICP from a web browser we strongly recommend this version.
- The Little Schemer is a fun book written in a conversational style. It has many useful lessons in it and a whimsical, puzzle-like quality to it, with delightful illustrations throughout. There are also many followup books in the series which explore other computer science topics in depth.
- Software Design for Flexibility takes many of the ideas from SICP and builds on them further. In many ways it can be thought of as SICP's true sequel.
- Not to mention that simply *using* Scheme or other Lisps are also a great way to learn about them. Contributing to Spritely or Guix are two excellent ways to put these skills to use!

Enjoy the journey!

```
define (display-post post)
  display-post-content
    $ post 'get-content

define (display-blog blog)
  display-blog-header
    $ blog 'get-title
  for-each
    lambda (post)
      display "\n"
      display-post post
      display "\n"
    $ blog 'get-posts
```

11. Appendix: Implementing sealers and unsealers

There are two ways to construct sealers and unsealers; one is the "coat check" pattern,[50] the other is the language-protected dynamic type construction pattern. The latter has less complications surrounding garbage collection and leads to some real "a-ha" moments, so we will show that one here.

To make this work, we will use dynamically constructed type-records (as taken from the SRFI-9 scheme extension). In Guile, we must import the following:

```
use-modules
  srfi srfi-9
```

To understand how these records work in the general case, here is an example of a srfi-9 record used to define a 2d positional object which we'll call `<pos>`:

```
define-record-type <pos>   ; <pos>: name of the type
  make-pos x y     ; make-pos: constructor, takes two arguments
  . pos?           ; pos?: brand-check predicate (is it a pos?)
  x pos-x          ; pos-x: accessor for x
  y pos-y          ; pos-y: accessor for y
```

Use of this pos is simple enough:

```
REPL> define our-pos
_____   make-pos 2 3
REPL> pos-x our-pos
;; => 2
REPL> pos-y our-pos
;; => 3
REPL> pos? our-pos
;; => #t
REPL> pos? 'something-else
;; => #f
```

We want to define our utilities so that they can be used from other modules. Let's start a new file which we'll call `simple-sealers.w` and define the following:

```
define-module : simple-sealers
  #:use-module : srfi srfi-9
  #:use-module : srfi srfi-9 gnu
```

50 The coat check pattern can be implemented and explained easily also: the coat is the value to be sealed, the sealer is the coat check desk, the ticket for later retrieval the sealed object, and the coat retrieval desk the unsealer. However this involves extra work to avoid garbage collection concerns amongst other issues; see "2.3.3 The Case for Kernel Support" in A Security Kernel Based on the Lambda Calculus.

```
  #:export (make-sealer-triplet)
```

Following from this, let's look at how `make-sealer-triplet` works:

```
;; Make a sealer, unsealer, and brand-check predicate using
;; dynamic type generation.
define (make-sealer-triplet)
  define-record-type <seal>
    seal val        ; constructor (sealer)
    . sealed?       ; predicate   (brand-check)
    val unseal      ; accessor    (unsealer)

  ;; Prevents snooping on contents at REPL, etc
  define (print-seal _rec port)
    display "#<sealed>" port
  set-record-type-printer! <seal> print-seal

  ;; Return sealer, unsealer, sealed? predicate
  values seal unseal sealed?
```

Within an invocation of `make-sealer-triplet`, we are defining a new `<seal>` type on the fly which will be completely distinct from any made during future invocations of `make-sealer-triplet`. The sealer is the constructor (accepting one argument, the sealed `val`), the brand-check is the type predicate, and the unsealer is the accessor of the sealed `val`. If running in a language argument which does not allow the user to piece apart a record without its corresponding accessor, there is no way to retrieve the associated value without the unsealer.

Note that upholding the above requires cooperation from the language runtime to not expose tools for deconstructing arbitrary record structures. "Unfortunately", Guile does provide tools readily with `record-accessor`, `record-constructor`, and so on. However, we put "unfortunately" in scare quotes because the situation is not so dire (or no more dire than the default situation in Guile, which already provides even more dangerous operations such as accessing the filesystem, the network, and so on). In Application safety, library safety, and beyond we describe how *language level safety* can be achieved. By not providing these record deconstructing tools to a constrained execution environment, the properties of sealers and unsealers as defined above can be upheld.

12. Appendix: Glossary

12.1. Goblins and capability terminology

- **Abstract Syntax Tree:** The abstracted programming language structure which the programming language operates at. See also *surface syntax*.
- **Actor:** A computational entity that operates only via asynchronous message passing. See also *actor model*, defined below. An *object* which only communicates via asynchronous message passing is usually considered an *actor*.
- **Actor model:** A programming paradigm where computation occurs between fully asynchronous message passing between computational entities named *actors*. An *actor* operating under the *classic actor model* processes one incoming message at a time defined by its current behavior, and in response may create new actors (obtaining their addresses in the process), send messages to other actors (including introducing them to actors this actor knows about in the process), or specify a change of behavior in regard to its next message. (To distinguish this core, original, and general subset of possible variants, we sometimes use

the term *classic actor model*.)

- **Actormap:** A transactional heap mapping *object* references to a set of *behaviors*.
- **Access Control List (ACL):** In contrast to *object capability security*, an *Access Control List* system relies on identity checks against approved operations. *ACL* systems tend to exhibit *ambient authority* and *confused deputy* vulnerabilities. See the paper ACLs Don't for an explanation of the many problems inherent to access control lists.
- **Ambient authority:** A source of vulnerabilities in many programs, particularly those operating under an *ACL* model of execution; ambient authority refers to authority that is implicitly available. Programs with *ambient authority* designs tend to be vulnerable to *confused deputy* attacks and usually fail to adhere to the *principle of least authority*, increasing the attack surface of a program dramatically. Since an *object capability* environment involves explicit use of references one holds and has access to, ambient authority risks are significantly smaller.
- **Behavior:** In Goblins, the *behavior* of the object is a procedure defining how it will currently react in response to an incoming message.
- **Behavior-oriented:** In contrast to a *data-oriented* system, a *behavior-oriented* system is primarily defined in terms of the *behaviors* of its participants and their relationships (which may both change over time). The mapping of *references* to *behavior* in Goblins is handled at a low level through the *actormap* (though this is a detail mostly hidden from users of Goblins). *Behavior-oriented* and *data-oriented* systems are duals, but the primary paradigm taken dramatically shapes the structure of the underlying architecture.
- **Capability:** See *object capability*.
- **CapTP:** Originally implemented in *E*, and now (as one layer of *OCapN*) implemented in *Goblins*, *CapTP* provides abstractions for distributed object programming which allow for programming against any object on the network to have the same ease and semantics as against locally hosted objects. Also provides some neat features such as *distributed garbage collection* and *promise pipelining*.
- **Causeway:** A distributed debugger implemented in *E* and a source of inspiration to *Goblins'* distributed debugger.
- **Classic actor model:** See *actor model*.
- **Client-to-server:** A network architecture where certain participants on the network have elevated, structurally central status, named *servers*, and *clients* connect to these as lighter, structurally less significant (and generally less addressable) status. Typically eventually results in a centralized topology. Contrast with *peer-to-peer* architectures.
- **Confused deputy:** A *confused deputy* is a kind of vulnerability which arises when one entity wishes to exploit the authority another entity has but which the former entity does not. Since the general *object capability* paradigm results in "if you can't have it, you can't use it", capability systems are (generally) free from such attacks. (Careless introduction of *identity* or *rights amplification* into an object capability system can re-introduce the possibility of such vulnerabilities, a topic of a future paper.) Originally described in The Confused Deputy (or why capabilities might have been invented) by Norm Hardy.
- **Constructor:** Within *Goblins*, a *constructor* is the procedure which, upon being invoked via *spawn*, returns the initial *behavior* of the newly constructed *object*. *spawn* passes the constructor both a *bcom* capability (for changing *behavior*) and all the remaining arguments passed to *spawn*, allowing for initial behavior to be tuned to the purpose of this particular

object and with other *capabilities* as references which allow the object to correctly operate.

- **Data-oriented:** In contrast to *behavior-oriented*, *data-oriented* systems involve heavy analysis of data describing the system. *Data-oriented* systems tend to involve significant amounts of judgements upon data and narrative, and thus tend to encourage *ACL* type designs (and thus also their problems), but this is not universally the case. Many CRUD web applications reading and writing from an SQL database with separate logic for interpreting or modifying that data are often *data-oriented*, and so are many systems which focus on passed messages as descriptive information of updates rather than actions to execute.
- **Dialect:** A language variant, particularly a variant of *Lisp*.
- **Distributed object programming:** A programming style where asynchronous programming may occur against a network of interconnected object relationships, reducing the conceptual overhead of building secure, highly *peer-to-peer* networked programs.
- **Distributed garbage collection:** The cooperation of multiple machines to free resources which are no longer needed. Implemented by *CapTP*. More specifically, *cyclic distributed garbage collection* if cycles crossing *machine* boundaries are collected, and *acyclic distributed garbage collection* if not.
- **E:** A major influence on the design of *Goblins*, direct successor to *Joule*, innovator of many *object capability* security patterns, first implementer of the *vat model of computation*, and the source of the first iteration of *CapTP* (and *VatTP*, as part of *Pluribus*).
- **Eval/Apply:** The heart of most programming languages: *eval* gathers up the values of arguments to an expression and *apply* performs the execution of the expression's behavior against the evaluated arguments. Each calls the other until achieving a "fixed point" of computation (the result of the total program evaluation). Popular topic of conversation amongst *Scheme* programmers.
- **Functional programming:** Programming without side effects; freedom from time.
- **Goblins:** The very distributed object system described by this paper, and the heart of *Spritely*'s programming environment.
- **Guile:** A particular *dialect* of *Scheme*, on which *Goblins* has been implemented, and the implementation of focus in this paper.
- **Homoiconic, Homoiconicity:** The property, most notably in *Lisp*, of *surface syntax* being the same as the *abstract syntax tree*, under a datastructure which can be manipulated and used by the programmer. This permits easy language extensions in the language and makes the language much more general.
- **Identity:** An abstract signifier for some individual, resource, or concept. More mysterious a topic than it appears at surface level, and comparison of identity equality and equivalence particularly complicated. When *identity* is checked as the primary form of access control, becomes an *Access Control List*.
- **Joule:** A fully asynchronous programming language, and a direct predecessor to *E*.
- **Lambda:** Procedure abstraction, associated heavily with the Lambda Calculus, and generally considered the heart of *Scheme*. Composes *Goblins objects* both as the *constructor* and as the *behavior* of the *object*. Generally considered "The Ultimate".
- **Lisp:** A programming language family known for being highly extensible, easy to implement, and with many *dialects*. (The particular *dialect* of *Lisp* used in this paper is *Scheme*.) *Lisp* has a highly flexible abstract syntax which makes it easy to "write Lisp in

Lisp", even "writing code that writes code", making language extensions or variants trivial compared to most other languages. Its *surface syntax* is typically parenthetical but is not necessarily so; see *Wisp* for the indentation-oriented surface syntax used in this paper.

- **Machine:** Within CapTP, a computer or process available on the network which contains *objects* which may be communicated with.
- **Monad:** Something we try hard to not expose you to the details of in *Goblins*. Arguably, an implicit one exists in *Goblins*. The meaning of this entry is left as an exercise for the reader.
- **Near/Far:** *Near* objects are co-located in the same vat, otherwise they are *far*.
- **Netlayer:** Within *OCapN*, an individual *netlayer* implements the abstract *netlayer* interface, which is a way to implement a secure channel of communication between two *machines*. Different transport layers can be used as a *netlayer*, ranging from *peer-to-peer* networks to more contemporary client-server architectures. Originally called *VatTP* in *E*'s implementation of *Pluribus*.
- **Network:** An interconnected system of *machines*. See *OCapN*.
- **OCapN (the Object Capability Network):** The combined layered abstractions of *CapTP*, *Netlayers*, and *OCapN* specific *URIs*. Combined, these allow for the implementation of a fully *peer-to-peer distributed object* programming environment with most networked protocol concerns abstracted away from the developer.
- **Object:** A term with a lot of variant meaning, but which in the case of *Goblins* means a reference to an abstract resource whose *behavior* is fully encapsulated by the runtime or network. (*Goblins* does not mean anything about class hierarchies by the word *object*, should you be suffering from a Java PTSD induced aversion to the term).
- **Object capability (ocap):** An *object capability* based architecture (sometimes known simply as a *capability* architecture, though this term has prominent naming conflicts) is one where one's authority is based on references which one can invoke to perform computation and cause effects. Without a reference, one can't perform an action, leading to the slogan "if you don't have it, you can't use it." Used as an abstraction of security and favorable to the *principle of least authority*, though maintaining that pattern requires discipline.
- **Object capability programming language:** A programming language upholding *object capability security* properties. Generally has the following properties: no ambient authority, no global mutable state, lexical scoping with reference passing being the primary mechanism for capability transfer, and importing a library should not provide access to interesting authority.
- **Object graph:** The set of relationships between *objects*. In an *object capability programming language*, this is typically the set of other object references within the *behavior* of an object's scope.
- **Peer-to-peer:** A network architecture where a participant in the network has the same abstracted priority across the network routing fabric as any other participant on the network. Contrast with *client-to-server* architectures.
- **Pluribus:** The equivalent of *OCapN* in *E*. Made for a good pun: *E, Pluribus, Unum*.
- **Principle of least authority:** Design systems such that entities hold no more authority than they need in order to reduce the attack surface of an application and its subcomponents. Generally easy to pull off in *object capability* architectures, and hard to pull off in *access control list* architectures.

- **Promise:** A special type of *object* abstraction representing a computation yet to be completed, either fulfilled or broken.
- **Promise pipelining:** From a programming perspective, the ability to send messages to the objects promises will eventually designate before they are fulfilled. From a network perspective, provides an optimization allowing delivery of messages to the host *machine* queuing eventual delivery of messages once dependent promises are fulfilled, eliminating unnecessary round trips. In other words, simplifies dependency-based asynchronous plan construction. Propagates errors.
- **Quasi-functional:** *Goblins'* tricky "looks imperative from the perspective of invoking another actor and functional from the perspective of an object updating its own behavior" twist on kinda-sorta *functional programming*. Allows for powerful *transactional* programming with time-traveling features without having to expose *monad* plumbing directly to the user.
- **Racket:** Another *Scheme* which *Spritely Goblins* is also implemented on, but which is not the focus of this paper.
- **REPL:** Read Eval Print Loop, an interactive programming language shell.
- **Rights amplification:** To (mis-)quote Alan Karp, "combine two things to get access to another thing". Frequently used to provide group-like features in ocap systems. Frequently implemented using *sealers and unsealers*. Used carelessly, can accidentally re-introduce *confused deputy* vulnerabilities, but the patterns we show in this paper are free of such problems. Analysis of this phenomena hopefully the subject of a future paper.
- **Safe serialization:** Allowing objects to describe how they should be serialized, while still following the *object capability* motto of "if you don't have it, you can't use it". Implemented by *Goblins*, but originally in Safe Serialization Under Mutual Suspicion, which was inspired by *Uneval/Unapply*.
- **Sealers and unsealers:** The equivalent of public-key cryptography, but implemented in programming language abstractions instead. Frequently used to implement *rights amplification*.
- **Scheme:** A *lambda* heavy *dialect* of *Lisp*. The examples in this paper use a particular *Scheme*, *Guile*. Has some interesting history regarding the exploration of the *actor model*, but probably too long to cover in an already overly-verbose glossary appendix.
- **Surface syntax:** The representation of the programming language that programmers (usually humans) operate at. In *Lisp* derived languages, the *surface syntax* and *abstract syntax tree* are generally not very far apart, which is partly what makes *Lisp* languages so extensible.
- **Swingset:** Another interesting contemporary object capability programming language environment, this one layered on Javascript and produced by Agoric.
- **Sneakernet:** A network architecture where messages are delivered physically from participant to participant (perhaps even on foot, be that foot in wearing a sneaker or not).
- **Spritely:** An umbrella project to advance networked communities and decentralized networked programming abstractions.
- **Spritely Goblins:** See *Goblins*.
- **The Spritely Institute:** The nonprofit which is the fiscal steward and primary developer of *Spritely Goblins* amongst other things (and which produced this paper).

- **Syntactic sugar:** Syntax abstractions which make programming more convenient and (ideally) pleasant to read and write.
- **Transaction, transactionality:** A set of operations which are replied in a conceptually atomic manner: either all occur or none occur. Within *Goblins*, a *turn* is a *transaction* representing a delta of *behavior* changes to the *actormap* (including the introduction of new *near objects*), as well as a queue of messages to be sent. In the event of an error, the changes will not be committed and the messages will not be sent.
- **Turn:** A top-level event handled by a *Vat*, generally a message sent to a particular *object*. One unique feature of *Goblins* is that turns happen within *transactions*.
- **Unum/Presence:** The *unum* is an abstracted, conceptually and programmatically unified object, implemented by individual object *presences*.
- **Uneval/Unapply:** The abstract concept behind *safe serialization* and the inverse of *eval/apply*. Produces a program representing a graph of objects (using only the capabilities the *objects'* behavior had in scope) which can be later re-instantiated using a complimentary kind of *eval/apply*. Originally a remark from Jonathan A. Rees to Mark S. Miller leading to the Safe Serialization Under Mutual Suspicion paper. See also Rees's blogpost: Pickling, uneval, unapply.
- **URI (Universal Resource Identifier):** A type of digital identifier indicating a networked resource. *OCapN* defines several of these to designate *machines* and *distributed objects*.
- **Vat, Vat model:** An event loop which contains a set of objects, designed to be able to communicate with objects in other event loops. Objects within the vat are considered *near* to each other may perform both synchronous and asynchronous programming against each other, whereas objects *far* from each other may only provide asynchronous programming against each other.
- **W7:** The subset of *Scheme* implemented (on top of Scheme48) for Jonathan A. Rees's PhD dissertation, A Security Kernel Based on the Lambda Calculus. Highly influential to Spritely Goblins in demonstrating clearly that a pure lexically scoped language (such as a strict subset of scheme) with no mutable toplevel scope or other sources of ambient authority is already a viable *object capability programming language*.
- **Wisp:** An indentation-sensitive surface-level *Lisp* syntax, and the one used in this paper. *Wisp* determines its expression boundaries based on whitespace. Compatible with most *Lisp* implementations. Defined under the SRFI-119 specification.

12.2. Core goblins operations

- `spawn`: The `spawn` operator in *Goblins*.
- `$`: The synchronous call-return operator in *Goblins*.
- `<-`: The asynchronous message passing operator in *Goblins*. Returns a *promise*.
- `on`: Set up a callback to be handled with the resolution of a *promise* (possibly returning its own promise related to said resolution).
- `bcom`: Pronounced "become", in *Goblins* `bcom` is the conventional name given to a *capability* relevant to a particular object which permits, and is used to, indicate the next *behavior* of the particular object. Passed by `spawn` (through Goblins' abstract kernel) to the object's *constructor*. Technically implemented as a *sealer*, allowing for a *functional* substrate for updating *behavior*.

12.3. Portable encrypted storage specific terminology

- **Portable encrypted storage:** A document storage system where files are not tied to any particular machine location (via *content addressed storage*) and are encrypted in such a way that hosting content does not provide the ability to read or modify the underlying contents of hosted files.
- **Content addressed (storage):** A document storage system where documents are named and verifiably retrieved by their content rather than by a particular network location.
- **Immutable/mutable:** Immutable objects and files do not change or update, mutable objects and files do.
- **Size-of-file attack:** Statistically determining likeliness that a file contains particular content based on its file size.
- **Chunked:** Split into consistently sized pieces to be latter reassembled, so as to avoid *size-of-file attacks* or for storage and retrieval optimizations.
- **Location agnostic:** Not tied to a particular location on the network.
- **Network agnostic:** Not tied to a particular network configuration or transport.

13. Appendix: Acknowledgments

An enormous number of people reviewed and provided feedback to this paper. Thank you to: Alan Karp, Baldur Jóhannsson, Chris Hibbert, Dan Connolly, Dan Finlay, Douglas Crockford, Jessica Tallon, Jonathan A. Rees, Jonathan Frederickson, Mark S. Miller, Stephen Webber, Robin Templeton, Leilani Gilpin, Kate Sills, and Ludovic Courtès. (**NOTE:** if you think you should/shouldn't be on this list, let us know and we'll edit appropriately!)

Thank you to Mark S. Miller who personally spent enormous amounts of time walking Christine through object capability ideas through the years and provided guidance on how to properly represent granovetter diagrams (which, as applied to object capability systems, really are a powerful but underdocumented visual language). Thank you to Jessica Tallon who actively used Spritely Goblins during the production of this paper, allowing for feedback from direct experience, including many suggestions for improvements in the examples. Thank you to Arne Babenhauserheide, who developed the Wisp syntax for Lisp used in this paper.

14. Appendix: ChangeLog

14.1. [2022-07-01 Fri]

- Incorporated feedback from Jakob L. Kreuze:
  - Add a detailed footnote explaining a bit more about why promise pipelining makes things cleaner and the risks of re-entrancy attack when coroutines provide *splitchronous* operations which appear *synchronous*
  - Various grammar nits
  - Finish explanation of Alisha and Bob bob having their own Carol representations (was a confusing intro to that section without)
  - Rework homoiconic/homoiconicity footnote, glossary entry

14.2. [2022-06-30 Thu]

- Incorporated feedback from Leilani Gilpin:
  - Added definitions in glossary for *peer-to-peer*, *client-to-server*, *sneakernet*, and italicize usage of those
  - Italicize usage of *transaction* and friends
  - Clarify: it's not a requirement to read the other (as of this moment: to-be-written) Spritely whitepapers to read this one
- Incorporated feedback from Robin Templeton:
  - Clarify that Unix isn't the origin of ACLs, it popularized them
  - Have a consistent comment syntax in first examples (and make it clear which ones are the `SHELL>` vs `REPL>`)
  - Capitalized Lisp/Smalltalk/Scheme
  - Explain more clearly that it's not that Goblins itself is the essential toolkit to build something like Spritely, it's that Goblins *provides* the essential features to be a worthy toolkit
  - Various garammar nitpicking
  - Give the motivation for Portable encrypted storage sooner in its section
  - Don't say "social network" in introduction, too captured a term
  - Consistency around "**New:** comments
  - Clearer introduction of what we mean by Unum in its first introduction
  - Call "Lisp types" -> "Lisp dialects"
  - Clarify which machine is local vs remote in `fork-motors` example

14.3. [2022-06-28 Tue]

- Many duplicate words removed

14.4. [2022-06-27 Mon]

- Add PDF export
- Explain which parts of the syntax are keywords
- Fix missing equals sign on the `fork-motors` section (a whole bunch of reviewers caught this, thanks to everyone else for being so careful ;))

14.5. [2022-06-26 Sun]

- Incorporated many grammar fixes suggested by Alan Karp
- Guix manifest updated: we're now at the point where anyone with access to the repo can easily build and verify all documents with the following:

```
SHELL> guix shell        # sets up dependencies
GUIX-SHELL> make         # builds papers, extracts all code
```

```
GUIX-SHELL> make check   # runs unit tests on examples
```

14.6. [2022-06-24 Fri]

- Add tests for blog examples
- Simplify proxy code / explanation by using `$` instead of `<-`
- Switch Matilda / the Teachers' invocations to use `<-` instead of `$` to show off these do work fully async
- Add explanation of `define-values`, `let-values`

14.7. [2022-06-23 Thu]

- Add tests for sealers/unsealers
- Fix some examples in sealers/unsealers section

14.8. [2022-06-22 Wed]

- Export `^cell` in `spritely-core.w` for tests
- Various bugfixes to interactive examples found while writing tests
- More information in predicate / conditional section
- Add unit tests for Taste of Goblins section
- Add makefile rule to run unit tests

14.9. [2022-06-21 Tue]

- Both Wisp and Scheme files are now automatically extracted when the user runs `make`
- Fix `format`, was using Racket's version
- Provide and use `method-cell.scm` for importing methods

14.10. [2022-06-20 Mon]

- Incorporating suggestions from Jessica Tallon
  - Fixed some renames of `eval-expr` (old name for metacirculator evaluator) to `evaluate` (thanks for the catch, Jessica Tallon!)
  - Rename some comments before lambda / procedure and procedure invocation / application examples
  - Make it clear that the methods macro does get complicated to figure out what's happening to the ellipses… it's not just you, dear reader!
- Rename `for-list` macro to `for`, keep it simpler

14.11. [2022-06-18 Sat]

- Add a bit more about the Y Combinator (no, not the company) to a footnote in Scheme in

Scheme.

- Many tyops caught by spelcheckr
- Couple of small grammar suggestions from Baldur

14.12. [2022-06-17 Fri]

- Refactor introduction to language stuff, add On language and syntax choice with a mini "how to convert wisp to parenthetical syntax in your head" explainer
- Security as relationships between objects written in full!
  - Guest post with review written!
  - Lessons learned written!

14.13. [2022-06-16 Thu]

- Adding to Appendix: A small-ish scheme and wisp primer
  - Add explanations of `letrec` and *named lets* to Iteration and recursion
  - Show symbols earlier when showing "some more types".
  - Finish metacircular footnote.
  - Explain that `'foo` is just shorthand for `(quote foo)`, etc
  - Add On the extensibility of Scheme (and Lisps in general)
  - Preview that we'll show how to write our own `when` in the first footnote which mentions
  - Fully explain how the evaluator works in Scheme in Scheme
  - Use `format` sooner

14.14. [2022-06-15 Wed]

- Adding to Appendix: A small-ish scheme and wisp primer
  - Added Closures
  - Add a footnote to Conditionals and predicates explaining that both `cond` and `if` can be written in terms of each other. Also distinguish between `<THEN-BODY>` and `<ELSE-BODY>` in the syntactic explanation of `cond`.
  - Eliminate `newline` from examples… one less procedure to explain!
  - Explain variable arguments, `define*`, `values`
  - Added Mutation, assignment, and other kinds of side effects
  - Added Scheme in Scheme and hoo boy, it's awesome.
  - Add alist and quasiquote examples to Lists and "cons"

14.15. [2022-06-14 Tue]

- Most of Appendix: A small-ish scheme and wisp primer written.

- Correct footnote… we *do* explore rights amplification in this paper :)

14.16. [2022-06-11 Sat]

- Add Makefile, README, instructions for building HTML and extracting output

14.17. [2022-06-10 Fri]

- Finished incomplete sandboxing footnote
- Include explanations of how to build module files explicitly
- Rename section: Application safety, library safety, and beyond (formerly "Application and library safety (and beyond)")
- Some updates to Appendix: Implementing sealers and unsealers
  - Show example of `pos?` predicate in use
  - Explain necessity of language runtime participating
  - Move coat check pattern footnote to this section (which is where it was supposed to be once the appendix was added, whoops)
- Reorder some of the appendices
- Started writing Appendix: A small scheme and wisp primer

14.18. [2022-06-09 Thu]

- Added Application safety, library safety, and beyond
- Added glossary definition for `on`
- Add Portable encrypted storage section and relevant glossary terms
- Made changelog and glossary subsections into actual reified, linkable-by-fragment subsections
- Added Conclusions

14.19. [2022-06-08 Wed]

- Added Spritely Goblins as a society of networked objects
- Remove `^revoker` from revocation pair example since it isn't used (the cell is though)
- Added When schemes go awry: failure propagation through pipelines
- Fleshed out the Appendix: Glossary in no small amount of detail.

14.20. [2022-06-07 Tue]

- Added OCapN section
- `.w` wisp files now extracted from spritely-core.org (source of this document) via org-babel. You can view them at:
  - taste-of-goblins.w

    - goblins-blog.w
    - simple-sealer.w
- Cleaned up several code examples.
- Switched to new Wisp syntax adjustment (after discussion with Wisp upstream): lines starting with keywords no longer require dot to continue previous line. Change likely to be incorporated in future wisps.

14.21. [2022-04-02 Sat]

- Promise pipelining examples added to A Taste of Goblins. This section was already planned but raised much interest in pre-review.
- Make tagging list with cons in `^post` a bit easier to understand
- First batch of the smaller of the changes suggested by Alan Karp (a whole bunch, should iterate…)
- Incorporated feedback from Jessica Tallon
    - Explained how Solitaire gets access to keyboard and mouse
    - Switched reference from `^mcell` to `^cell`… oops, that's what I get from copy-pasting code from another document
    - Renamed `our-cgreeter` to `julius` in example
    - Fixed expected displayed message in "heard back" part
    - No longer use named let but the `^editor` constructor, reflect that in surrounding text
    - Mention cons prepends to a list where appropriate
    - Fixed "Run by Robert" which had mistakenly said it was run by Lauren
    - Make it clearer that Lauren will hold Robert responsible for **anyone** who uses `admin-for-robert` (including someone Robert delegates authority to).
    - Moved sealers and unsealers implementation details to Appendix: Implementing sealers and unsealers