A Scheme Primer

Christine Lemmer-Webber and the Spritely Institute

Table of Contents

The following is a primer for the Scheme family of programming languages. It was originally written to aid newcomers to technology being developed at The Spritely Institute but is designed to be general enough to be readable by anyone who is interested in Scheme.

1. Introduction

In all the world of computer programming, there are few languages as simple, clean, comprehensive, powerful, and extensible as Scheme. The introduction to the R5RS edition of Scheme's standardization[1] explains its philosophy well:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

This minimalism means that the foundations of Scheme are easy to learn. The R5RS introduction continues with:

> Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

1 R5RS stands for "Revised(5) Report on the Algorithmic Language Scheme". It is not the only edition of Scheme, but it is very small, clean, and minimal. It may be too small, in some ways: implementations of R5RS Scheme are notorious for being incompatible with each other when it comes to actual libraries in use, not least of which because R5RS did not even include a library system! Nonetheless, R5RS is short and easy to read. If you enjoy this document, we would encourage reading either R5RS (or R7RS-small for an only slightly longer but more inter-compatible modern report on Scheme).

With just a few rules and an incredibly simple syntax, Scheme manages to be able to handle any language paradigm you can throw at it. Its minimal base and strong support for extensibility means that it is beloved by and frequently used as a foundation for academic language research.

But a strong reputation in one domain can also be a weak reputation in another. Scheme's association with being an academic language has also lead it to be frequently interpreted as too difficult for the "average" programmer to adopt.

In reality, there is nothing difficult about Scheme, and any programmer can learn it in a very short amount of time (even, maybe especially, children and non-programmers).[2]

At the Spritely Institute, we decided to base our core technology, Spritely Goblins, on Scheme. We found that while there were excellent in-depth writings on Scheme, and some simple and short Scheme tutorials, what was lacking was a middle-of-the-road introduction. The following is somewhere between a brief and comprehensive overview of Scheme. A shallow read of the following text is sufficient to begin being productive with Scheme, but the enthusiastic reader will find much depth (especially by reading the footnotes).

We begin with no assumption of programming experience, but such experience will help speed the reader through some of the early chapters. The further we go in the tutorial, the more advanced topics become. We will end with a real whammy: how to write a Scheme interpreter in Scheme in a mere 30 single lines of code.

2. Setting up

You will need to choose a Scheme implementation to work with, as well as an editor. There are many choices for each, but we will narrow our suggestions to two paths:

- Guile Scheme + GNU Emacs + Geiser: this is what this tutorial was written using, and is a powerful option (which also opens the door to working with Guix, one of the most interesting Scheme projects out there). However, it is also a path with a considerable learning curve.
- Racket, which comes with a built-in IDE called DrRacket. This is an easy path to get started with.

Some code examples are preceded by `REPL>`. *REPL* stands for "Read Eval Print Loop", which here means an interactive scheme prompt to experiment with entering expressions.

3. Hello Scheme!

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

2 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.)

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.

```
(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.

4. 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:

```
#t                              ; boolean representing "true"
#f                              ; boolean representing "false"
"Pangalactic Gargleblaster"     ; string (text)
'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
```

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).

5. 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:[3]

```
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".[4] 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")
```

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

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

```
; 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:
;   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!)

6. 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?`:[5]

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

5 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 `?`.

(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.)[6]

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:[7]

6 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` 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).

7 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
```

```
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)
                "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:[8]

```
(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

8 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.

identity (as defined by the language's runtime).[9] 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)
; => #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!"
```

7. 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.[10] 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

9 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

10 *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.

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)
```

For very historical reasons,[11] 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"):[12]

```
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).[13]

11 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

12 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.

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

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"))
         (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. 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.

9. 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")))
                '("ice cream" "fudge" "cookies"))
; prints:
;   I just love ICE CREAM!!!
;   I just love FUDGE!!!
;   I just love COOKIES!!!
```

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 tree-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 `list` 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 ((words '("carrot" "potato" "pea" "celery"))
                 (num-words 0)
                 (num-chars 0))
        (if (eq? words '())
            (format #f "We found ~a words and ~a chars!"
                    num-words num-chars)
            (loop (cdr words)
                  (+ num-words 1)
                  (+ num-chars (string-length (car words))))))
; => "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.

10. 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.[14] 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.[15] Another example is mutable vectors and `vector-set!`:

14 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`.

15 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.

```
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.

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.[16]

11. 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)
```

16 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 not 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.

```
  (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.[17]

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.[18] We can also notice that `body ...` appears in both the

17 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.

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

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

(define-syntax when
```

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 STRAWBERRIES!!!
;   I just love BANANAS!!!
;   I just love GRAPES!!!
```

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)
```

```
  (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 ...)
          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.

```
                        "!!!\n")))
; prints:
;   I just love STRAWBERRIES!!!
;   I just love BANANAS!!!
;   I just love GRAPES!!!
```

It works! This is much easier to read.[19]

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?)
```

19 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 ) {
//      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")))
```

```
; => #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 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 family 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.

12. 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!)[20]

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 square a number:

```
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
```

20 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.

```
           (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
        `((+ . ,+)
          (= . ,=)))
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.[21] 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))
```

21 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.

```
            (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)
;     (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.[22] 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 `vals` from the procedure invocation.[23]

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 items, 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

22 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!)

23 This is where the choice is made between lexical and dynamic scoping. The environment which is extended is not the environment of the caller, but the environment of the previous expression in which this lambda was defined… this is how we achieve the functionality described in the `closures` section.

In a sense, "lexical scope" is "scope is defined by capturing where it is written" and is capability secure, is explicit, and preserves the "if you don't have it, you can't use it" property. "Dynamic scope" is instead "scope is defined by the invoker" (and their parent invokers, all the way up the stack of current invocations) and is implicit, but also ambient including in the ambient authority sense, making dynamic scope suffer more serious security issues (and not coincidentally, more strongly resemble access control list systems).

borrows from C's standard library).[24] 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 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![25]

24 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`.

25 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!