

Little Schemer Basics

Your final assignment is going out as Handout 38, and that assignment challenges you to extend a working expression evaluator that speaks a small subset of the Scheme language. The point of the assignment is to give you practice with programming languages and inheritance, but we don't need to learn the full Scheme language in order to do that. Instead, we'll focus on a core subset of the language—a subset small enough to learn in a day or two—and in the process see how phenomenally powerful such a small language can be. We're calling our language **Little Scheme**, and the interpreter you're building is being dubbed the **little-schemer**.

Getting Started

little-schemer, like virtually all Scheme environments, operates much like the expression evaluator I demoed in lecture, in that you type in expressions, hit return, and expect the expression's evaluation to be published in response.

You interact with the little environment by typing something in and then allowing **little-schemer** to read what you typed, evaluate it, and publish a result. Sometimes, what you type is so trivial that the result is precisely what you typed in.

```
little-schemer 1> 14
14
little-schemer 2> 4/5
4/5
little-schemer 3> 1803/2404
3/4
little-schemer 4> "CS106X"
"CS106X"
little-schemer 5> "Stanford University"
"Stanford University"
little-schemer 6> true
true
little-schemer 7> false
false
```

You're shouldn't be impressed yet, but there really is an intellectually compelling explanation for what just happened: Everything you type in has to evaluate to a single value. When the expressions we type in are constants, **little-schemer** spits them right back at you. Constants (be they integer, fraction, string, or Boolean) evaluate to themselves.

In general, you'll get the interpreter to do some work for you. By typing out more elaborate expressions involving a function call, Scheme evaluates the call and eventually prints its evaluation.

```

little-schemer 1> (add 1 2 3 4 5 6 7 8 9 10)
55
little-schemer 2> (multiply 1 2 3 4 5)
120
little-schemer 3> (subtract 100 17)
83
little-schemer 4> (add)
0
little-schemer 5> (multiply)
1
little-schemer 6> (add 32 (multiply 9/5 100))
212
little-schemer 7> (add 32 (multiply 9/5 (subtract 0 40)))
-40

```

Each of the seven examples above invoked some core mathematical built-in. All function names are required to be purely alphanumeric, which is why you see names like **add** and **multiply** instead of the more traditional **+** and *****. We don't support **divide** or the ability to type in negative fraction constants, because we're trying to keep the core as small as possible to just illustrate how an interpreted language like **little-scheme** works.

What **is** surprising is how we invoke a function in the first place. Function calls are expressed as lists, where function name and arguments are bundled in between open and close parentheses. All functions, including the ones like **add** and **subtract**, are invoked in prefix form. Each of the top-level items making up the function call gets evaluated, and once all sub-expressions have been recursively evaluated, the top-level expression gets evaluated too.

When you've had it and want to do something else, you leave the environment by calling **quit**.

```

little-schemer 1> (quit)

```

```

[Press <return> to close]

```

little-scheme Data Types

little-scheme is actually a little more sophisticated than we're going to make it out to be. In fact, we're going to pretend that **little-scheme** offers up only two types of data: **primitives** and **lists**. Primitives are the types that can't be subdivided: integers, rationals, strings, symbols, and Booleans. Lists comprise the generic abstraction **little-scheme** offers up for aggregating data. Full blown Scheme allows for arrays (called vectors), records, and even classes, but we're going to ignore everything else and pretend that lists

are all we have for bundling multiple pieces of information. It's not at all an unreasonable thing to do, because true Scheme programmers use the list more than all of the other aggregate types combined. (In fact, LISP is short for **L**ist **P**rocessing, and Scheme and LISP are fundamentally the same language.)

What primitives are there? For our purposes, not all that many. We're more interested in illustrating programming language techniques and computation that we are in representing characters and imaginary numbers, so we're sticking to a fairly small core of primitive types:

Numbers: The set of all numeric constants includes:

42, 201, 12345678, 45/2, 3/4, -22/7

As expressions, rational number constants evaluate to themselves.

Booleans: Scheme has a strong data type for the Boolean, just like C++ and Java. The Scheme equivalents of true and false are **true** and **false**, although everything other than **false** is understood to be logically equivalent to **true**. Of course, Boolean expressions evaluate to Boolean values and influence what code is executed when and how many times. Scheme provides the full bag of logical operators for compound expressions: **and**, **or**, and **not**. As expressions, Boolean constants evaluate to themselves..

Strings: String constants are linear sequences of zero or more characters, and are delimited by double quotes. As expressions, string constants evaluate to themselves (with the double quotes). Examples of string constants include:

**"Eric Lovett likes little-scheme",
"1234567890!@#\$%^&*()",
"He said, \"Hey there!!\"", and
"".**

We're downplaying the role of strings in the assignment, and including them just so we have another participant in the **Expression** class hierarchy. If you want, you can add functions that concatenate, compare, and find substrings. Not required, though.

Symbols: Symbols are sequences of characters (without the double quotes) that serve as general-purpose identifiers. Examples include:

**add, subtract, multiply, cons, car, cdr, list
lambda, define, apply, or, not, x, y, z**

double, reverse, factorial, fibonacci, find

In a nutshell, a symbol is a glorified variable typically associated with another expression. Symbols are bound to these expressions using the **define** built-in, and symbols generally evaluate to whatever expression they're associated with. To help simplify scanning, we require that all symbols be alphanumeric and begin with a letter.

Here's enough code to illustrate how symbols work in our little language.

```

little-schemer 1> (define x 142)
x
little-schemer 2> x
142
little-schemer 3> (define name "Keith Schwarz")
name
little-schemer 4> name
"Keith Schwarz"
little-schemer 5> (define y x)
y
little-schemer 6> y
142
little-schemer 7> (define x 194)
x
little-schemer 8> x
194
little-schemer 9> y
142

```

Lists

A list is a sequence of zero or more expressions, where each sub-expression can be either a primitive or another list. In ASCII form, a list is represented by a left parenthesis ' (' followed by the ASCII forms of all the elements separated by spaces, followed by a right parenthesis ') '. Here are a few examples:

```

(1 2 3)
(1 "not a number" 22/7)
("hi" (1 2 3 4) 42/5)
("hi" (((1 2 3 (4)))) 42/7)

```

All four of these lists contain three elements, although only the first one is homogenous. Each of the sub-expressions can be pretty much anything you want, as long as each of the atoms are spelled out properly and all of those parentheses are balanced and properly nested.

Function Evaluation

Function calls are also expressed using the list syntax. When you type in a list at the prompt and hit return, the expression evaluator reads it, and by default interprets it as a function call. It takes the first expression to be the function and assumes the remaining expressions are the arguments.

```

little-schemer 1> (add 1 2 3)
6
little-schemer 2> (hiccup (list 1 2 3 4))
(1 1 2 2 3 3 4 4)
little-schemer 3> (factorial 4)
24
little-schemer 4> (fibonacci 20)
6765

```

Only the **add** function is a built-in; the other three functions aren't built in but were defined by me. We'll figure out how to define our own functions soon.

An unfortunate feature of the language (all dialects of Scheme, really—not just ours) is that list constants look the same as function calls, so if you type, say, **(1 2 3)** in at the interpreter prompt, it throws a tantrum and prints the following:

```

little-schemer 1> (1 2 3)
Error: Function evaluation requires a procedure be present
in the front position.

```

Now, you probably just want to construct a list of three numbers when you type something like that. But **little-schemer** is looking at that **1** to fill the same role that **add**, **subtract**, and **multiply** did in earlier examples. The interpreter follows its own rules so closely that it looks for a procedure named **1**, but when it realizes that **1** is a rational and not something that can identify a function very easily, it freaks out a bit and throws an error message your way.

That means that if we want to construct a list of pure numbers (or strings, or Booleans, or sublists, or some combination), then we need to use the **list** built-in to do that. Look here:

```

little-schemer 1> (list 1 2 3)
(1 2 3)
little-schemer 2> (list "hey" 45/10 (list true false (list)))
("hey" 9/2 (true false ()))

```

Note that one of the inner lists is empty (**list** can even take zero arguments), and that there's no homogeneity requirement.

Manipulating Lists

Since most everything is expressed as a list, we need to know how to break them down into their constituent parts and build up new ones. We'll soon see that even function definitions are expressed as lists, so you need to be fluent in the core subset of built-ins that help you manipulate them. Here's an overview of Lists 101.

cons, car, and cdr

cons is the basic list construction function. **cons** takes an element and a list and constructs a new list, one longer than the old one, which has the element pushed on the front of the list. Check this out:

```
little-schemer 1> (cons 1 (list 2 3))
(1 2 3)
little-schemer 2> (cons "meenie" (cons "miney" (cons "mo" (list))))
("meenie" "miney" "mo")
little-schemer 3> (cons (list 1 2) (list 1 2))
((1 2) 1 2)
little-schemer 4> (cons "1" (list 2 (list "three")))
("1" 2 ("three"))
```

While **cons** builds lists out of elements, **car** and **cdr** pull elements out of lists. **car** returns the first element of a list, and **cdr** returns the list without the **car**. The argument not only needs to be a list, but it needs to be a non-empty one, else you should expect more tantrums.

```
little-schemer 1> (car (list 1 2 3))
1
little-schemer 2> (cdr (list false "1" (list "2" "3") "4"))
("1" ("2" "3") "4")
little-schemer 3> (cons (car (list 1 2 3)) (cdr (list 1 2 3)))
(1 2 3)
little-schemer 4> (car (list "singleton"))
"singleton"
little-schemer 5> (cdr (list "singleton"))
()
little-schemer 6> (car 1)
Error: car expects its one argument to be a list
little-schemer 7> (cdr "two")
Error: cdr expects its one argument to be a list.
```

By cascading calls to **car** and **cdr**, you can extract individual items and sublists. If you need the fourth element of a list, **cdr** three times, and take the **car**.

```
little-schemer 1> (car (cdr (cdr (cdr (list 1 2 3 4 5 6 7)))))
```

Other combinations lead to different elements, as illustrated here:

```

little-schemer 1> (cdr (cdr (cdr (cdr (cdr (list 5 10 15 20 25
50 99))))))
(50 99)
little-schemer 2> (cdr (car (list (list 1 2) (list 3 4 5))))
(2)

```

if expressions

The **if** form in Scheme is most similar to the ternary **?:** operator in C and C++. **if** takes the form of an expression, as if **if** is some function being invoked. **if** requires three expressions as arguments: the first is the test, the second is the expression to be evaluated if the test passes (or, more specifically—if the result of the first expression is anything other than **false**), and the third is the expression to be evaluated if the test fails. All three expressions are required—in particular, the third expression is required, because the entire **if** expression needs to evaluate to something in the event that the test fails. Interestingly enough, the two expressions needn't evaluate to the same data type.

```

little-schemer 1> (define x 7)
x
little-schemer 2> (define y 10)
y
little-schemer 3> (if (equal x y) "fifty" 999)
999
little-schemer 4> (if (less x y) "fifty" 999)
"fifty"

```

Defining Functions

little-schemer wouldn't be of much use to us if all we ever did were compare rational numbers and construct lists. Like all programming languages, Scheme allows us to build our own procedures and add them to the set of existing ones. Very few implementations of Scheme even distinguish between the built-in functions and the user-defined ones.

Our dialect of Scheme exposes a reliance on the notion of a **lambda function**, which is little more than a parameterized expression that looks like a function definition without a name. Here's a simple lambda expression framed in terms of one open argument named **n**:

```

little-schemer 1> (lambda (n) (add n 1))
<lambda>

```

lambda is a gesture to the lambda calculus, which is the operative theory of functions and function evaluation that backs the design of most programming languages. The above

expression is equated with the expression that adds **1** to **n** once **n** has been bound to a value.

Here's a lambda expression defined to evaluate to the larger of two variables:

```
little-schemer 1> (lambda (one two) (if (less one two) two
one))
<lambda>
```

If a lambda expression is levied against the corresponding number of argument expressions, then the lambda's parameters are bound to those arguments so that the inner expression can be fully evaluated.

```
little-schemer 1> (lambda (x) (add x 1))
<lambda>
little-schemer 2> (increment 99)
100
little-schemer 3> (lambda (one two) (if (less one two) two one))
<lambda>
little-schemer 4> ((lambda (one two) (if (less one two) two one)) 14
7)
14
little-schemer 5> ((lambda (one two) (if (less one two) two one)) 14
70)
70
```

The above amounts to a traditional function call, save for the fact that an nameless function has taken up residence where the name of a function normally lives. (We'll define **increment** is a second.)

Normally, we use the lambda notation to implement a new function, and then use the **define** built-in to bind a name to it.

```
little-schemer 1> (define increment (lambda (n) (add n 1)))
increment
little-schemer 2> (increment 49)
50
little-schemer 3> (define max (lambda (one two) (if (less one two)
two one)))
max
little-schemer 4> (max 17 43)
43
little-schemer 5> (max 63 (max 12 (increment 76)))
77
```

Once symbols like **increment** and **max** have been bound to **lambdas**, they're peers with other symbols like **add**, **less**, **list**, and **cdr**.

Recursion

With Scheme, recursion is really, recursion big. Its primary data structure is the list, the list is inductively defined, and where there's an inductive definition there's sure to be recursion. Other Scheme implementations support iteration as well, but we're just going to stick with pure recursion. Most of the algorithms that can be implemented iteratively can just as easily be implemented recursively, so we'll just pretend that iteration isn't an option and not bother supporting it.

Here're the obligatory **factorial** and **fibonacci** functions. They work well enough, albeit slowly in the case of the doubly recursive **fibonacci** procedure.¹

```

little-schemer 1> (define factorial (lambda (n)
                      (if (equal n 0) 1
                              (multiply n (factorial (subtract n 1))))))
factorial
little-schemer 2> (factorial 4)
24
little-schemer 3> (factorial 10)
3628800
little-schemer 4> (define fib (lambda (n)
                      (if (less n 2) n
                              (add (fib (subtract n 1))
                                  (fib (subtract n 2))))))
fib
little-schemer 5> (fib 5)
5
little-schemer 6> (fib 10)
55
little-schemer 7> (fib 11)
89

```

(Whenever a function returns 55 and 89, you know it's managing to compute Fibonacci numbers.)

You'll note that each of the two lambda expressions refer to symbols called **factorial** and **fib** before they're bound, but that's fine. The parameterized expressions under the jurisdictions of the **lambdas** aren't deeply evaluated. They're just scanned for structure and then built into a data structure that can be tied to the **factorial** and **fib** symbols. The fact that the lambda expressions are bound to **factorial** and **fib** and make use of

¹ Note that I use carriage returns and plenty of white space to spread the definition of a function over several lines. This is just done for clarity, as the entire function definition needs to be typed in on one line when typing it directly in to the interpreter.

those symbols in the implementation is consistent with what we always see with recursive implementations.

List Recursion

We've asserted the list to be the central aggregate data type, but we've ignored lists so far. Code that reads and otherwise manipulates a list needs to visit every single one of its elements. Schemers use what's called **car-cdr** recursion, which recursively **cdr's** down the list until there's no list left. Each **cdr** has its own **car**, and you grab the car with each call and let it contribute to the answer you're trying to build. Think of the **car** of the list as the first element, and the **cdr** as everything else.

Here's an intentionally unspectacular function, just to help illustrate the idiom.

```

little-schemer 1> (define hiccup (lambda (ls)
                          (if (null ls) ls
                              (cons (car ls)
                                      (cons (car ls)
                                             (hiccup (cdr ls)))))))
hiccup
little-schemer 2> (hiccup (list 1 2 3 4))
(1 1 2 2 3 3 4 4)
```

The recursion manages to drill down into successive **cdrs** of the list, and as the recursion unwinds, the answer of interest accumulates up through a chain of evaluation results.

Here's another example that returns the concatenation of two lists:

```

little-schemer 1> (define append (lambda (one two)
                                   (if (null one) two
                                       (cons (car one)
                                             (append (cdr one) two))))
append
little-schemer 2> (append (list false 1 "two") (list 8 "zero" 18/4))
(false 1 "two" 8 "zero" 9/2)
```