Chapter 8

Sussman and Steele make Scheme

The proliferation of Lisp systems was waiting for a reaction. Much like the relative austerity of neoclassicism followed the overload of frills from the baroque period, the never-ending addition of features to Lisp almost begged for someone to stomp their foot down and yell "enough is enough!"

Now, that did not *literally* happen, of course, but what did happen was the development of a new Lisp that would radically simplify and clean up the last decade-and-a-half worth of Lisp work. This is the Scheme language, created by MIT hackers Gerald Jay Sussman and Guy L. Steele.

Scheme started as a small Lisp interpreter to, among others, help understand the finer details of actor systems as described by Carl Hewitt in his paper introducing the concept [31]. Actors, in Hewitz' paper, are a highly precise concept but we can make do with a basic definition here: think of them as self-contained computations that are part of a bigger whole, run independently, and communicate through the exchange of messages. A bunch of tiny computers talking to each other. If you think that this is a bit like objects, you're right on the money but that's a story for another time.

In Sussman and Steele's 1975 paper, *SCHEME: An Interpreter For Extended Lambda Calculus* [77], they introduce the system as:

> Inspired by ACTORS, we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus, but extended for side effects, multiprocessing, and process synchronization. The purpose of this implementation is tutorial.

They go on to list some points that are important to them, which

mostly revolve around clarifying existing LISP constructs and simplifying these concepts to land at a language that was exceptionally well-suited for experimentation with programming semantics and style. Their first version of the language was called "Schemer" but ITS had a six character limit, so "Scheme" was the name that stuck.

This first version of Scheme was written in Maclisp in around 700 lines of code for the basic interpreter. This small bit of code contains enough to implement a multiprocessing system, including critical sections (`EVALUATE|UNINTERRUPTIBLY`), all hacked together on top of a Maclisp function, `ALARMCLOCK` that can send timed interrupts to Lisp programms on ITS and Multics. We should be fine using the term "hack" here—in the 1970s it did not have the pejorative meaning that it carries today; rather, it was a badge of honor.

Closures, actors: all the same thing

When Sussman and Steele implemented Scheme, actors were front-of-mind and Scheme contained a little actor system on top of the machinery they built in Maclisp to support all that. One of the things they added, to make things nice and transparent, was a function called `ALPHA` that looked the same as `LAMBDA` but returned an anonymous actor instead of an anonymous function or "closure". Then, "invoking" this thing would just send the passed in arguments as a message to the actor, which would return a message that was the return value. A neat idea to make the step from "function-oriented" programming to "actor-oriented" programming simpler.

Much to their surprise, when implementing `ALPHA`, they found that the code was identical to `APPLY` in every sense, except for the lower-level primitive it used! They stumbled upon somewhat of a discovery that implicated that closures and actors are actually the same thing. Not only proved this helpful in further understanding actors, but it also validated their hypothesis that a small, lexically scoped language like Scheme would make for a very powerful programming language and computing research platform.

Tail recursion

However neat the multiprocessing, the first Scheme presentation focuses on computation, and specifically on *recursrive* computation. It uses our friend the factorial function, which has a simple definition in scheme:

```
(define fact (lambda (n)
  (if (= n 0)
```

```
    1
    (* n (fact (- n 1))))))
```

Even without an introduction to Scheme, this should be pretty clear. Now, if we run this manually, tracing the execution steps, we'd get an ever-larger expansion, as it were. Quoting from the paper:

☞
```
> (fact 3)
(if (= 3 0) 1 (* 3 (fact (- 3 1))))
```

Essentially, what we're doing here is treating the function as a function in Church' lambda calculus, and we simply expand by substitution. Next, executing the conditional, the multiplication, and the subtraction gives:

```
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
```

Let's continue to see where this lambda-calculus-like expansion leads us:

```
(* 3 (if (= 2 0 1 (* 2 (fact (- 2 1))))))
(* 3 (* 2 (fact (- 2 1))))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 (if (= 1 0) 1 (* 1 fact (- 1 1)))))
(* 3 (* 2 (* 1 (fact (- 1 1)))))
(* 3 (* 2 (* 1 (fact 0))))
(* 3 (* 2 (* 1 (if = 0 0) 1 (* 0 (fact - 0 1)))))
(* 3 (* 2 (* 1 (* 1 1))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Just simple substitution gave us the correct answer. Technically, we could have done the same substitution process for + and *, but this example is long enough. Note, though, that the expression gets longer and longer before we got rid of all the FACT function calls that kept appearing and we finally could start on a path towards simplification. However a computer calculates this recursion, it needs to do the same. It must keep all that state around, and that can be prohibitively expensive. Sussman and Steele came up with an alternative that iterates instead; however, it looks like just a more complex recursive function! Here it is:

```
(define fact
  (lambda (n)
```

```
(labels ((fact1
  (lambda (m ans)
    (if (= m 0)
      ans
      (fact1 (- m 1) (* m ans))))))
(fact1 n 1))))
```

LABELS defines a local function, pretty much the plural form of the LABEL function we already encountered. So let's go at it again, and just do expansions to see where it leads us. First, we essentially "jump" into the inner function:

☞
```
> (fact 3)
(fact1 3 1)
```

As you can see from the function definition, we now just proceed with calls to the inner FACT1 local function. Let's do a couple of steps:

```
(if (= 3 0) 1 (fact1 (- 3 1) (* 3 1)))
(fact1 (- 3 1) (* 3 1))
(fact1 2 3)
```

Interesting—we're back where we started, just with different arguments to the local function. Again:

```
(if (= 2 0) 3 (fact1 (- 2 1) (* 2 3)))
(fact1 (- 2 1) (* 2 3))
(fact1 1 6)
```

Still, nothing of the "growing and shrinking" behaviour we saw before. And indeed, if we keep doing this, we finish with what might be called a "flat profile":

```
(if (= 1 0) 6 (fact1 (- 1 1) (* 1 6)))
(fact1 (- 1 1) (* 1 6))
(fact1 0 6)
(if (= 0 0) 6 (fact1 (- 0 1) (* 0 6)))
6
```

Note that we did not do anything special—in "lambda calculus" style, we just substituted expressions in a loose following of Church's original rules. Somehow, the second factorial function is setup so that the state of the computation is not kept in an every-widening and then shrinking expression, but in the two variables.

The biggest difference between the two versions is in how and where they call themselves. The first one calls itself with this:

```
(* n (fact (- n 1)))
```

and the second one with this:

```
(fact1 (- m 1) (* m ans))
```

Can you spot the difference? If you worked with recursion before, the answer is simple: in the second version, the call-to-self is the *last* thing to happen; in the first one, we call ourselves but then have to take the answer and do a multiplication with it. It turns out that this mechanical substitution process turns into iteration if the recursion happens as the very last thing, and the term for this is *tail recursion*.

The beauty of this all: tail recursion can be detected by a smart parser and turned from recursion into iteration because we don't need to keep intermediate results. The process is called *tail call optimization* and was part of Scheme from day one. That you can arrive at such a result by just doggedly going through the steps of, essentially, the rules of applying the lambda calculus is nothing short of wonderful. Adding an accumulator to store intermediate results instead of relying on, in most cases, the computer's stack to do the same is a "trick" that can turn many recursive functions into properly tail-recursive ones so that they can be run iteratively, using less space.

Lexical scope

Another innovation was the elimination of dynamic scoping in favour of lexical scoping. In the section on LISP 1.5 we saw how the dynamic scoping of variables could lead to confusion; confusion that was dubbed "the FUNARG problem" and led to a lot of discussion in academic circles. To recap, Lisp's default way of looking up variables is to search in the execution environment. A quick example in a modern Lisp that uses dynamic variables (Common Lisp in this case, which uses dynamic binding by default for global variables defined with `DEFVAR`):

```
☞ > (defvar x 99)
  X
☞ > (defun getx () x)
  GETX
☞ > (getx)
  99
☞ > (let ((x 1)) (getx))
  1
☞ > (getx)
  99
```

The LET binding "overwrites" the value of the variable *X* before calling GETX, so during the invocation inside the LET binding the variable is assigned that value; after it, the old value pops back up.

Note that you can only figure out what will happen from the source code by tracing through it, like a sort of human single-stepping debugger. The GETX code itself does not give any indication where X could be set, it can happen all over the place. This is a trivial example, but this gets messy quick. It has some powerful uses, but its counterpart, lexical binding, is overall simpler to follow. To demonstrate:

```
☞ > (defvar bumpit nil)
  BUMPIT
☞ > (defvar getit nil)
  GETIT
☞ > (let ((y 1))
      (setq bumpit (lambda () (setq y (+ y 1))))
      (setq getit (lambda () y)))
  <FUNCTION (LAMBDA ()) {1003B53f3B}>
☞ > (funcall bumpit)
  2
☞ > (funcall bumpit)
  3
☞ > (funcall getit)
  3
☞ > y
  The variable Y is unbound.
```

The code is a bit more complex here, but essentially we store two lambda functions in BUMPIT and GETIT inside the LET binding. LET has a lexically scoped (local) variable Y, which these functions both access. Using FUNCALL, we invoke the two Lambda functions a couple of times seeing the value of Y change (why we need FUNCALL here is something we'll get to). Finally, we try to access Y and get an error—Y only exists inside the LET, not outside it. It is now trivial to figure out how Y can be changed: reading that little bit of code in the LET is all you need to do, anything else cannot possible access or, worse, change the value. The term "lexical" comes from greek *lexikos*, "pertaining to words". With lexical scoping, the words in the code tell you the scope of a variable and you can safely contain what happens to it. This is exactly the sort of thing you want when implementing actors, which are supposed to encapsulate their state much like objects do.

Shared namespace

Consider the Lisp code above, where we used `FUNCALL` to invoke a Lambda expression stored in a variable. Why is that? This was an example in Common Lisp, which has, like LISP 1.5 and most Lisps that followed it, different storage for functions and variables. In LISP 1.5, variable values are stored as APVALs on a single association list and functions as EXPRs. The system had some rules for where to look things up—depending on how a symbol was used, etcetera, but later systems like Common Lisp made this separation explicit: values are values and functions are functions, and they are treated separately. So, in Common Lisp:

```
☞ > (setq a 42)
  42
☞ > a
  42
☞ > (setq b (lambda () 42))
  <FUNCTION (LAMBDA ()) {53735B4B}>
☞ > b
  <FUNCTION (LAMBDA ()) {53735B4B}>
☞ > (b)
  debugger invoked on a UNDEFINED-FUNCTION….
☞ > (defun b () 42)
  B
☞ > (b)
  42
```

As you can see, we can't just assign `B` a lambda expression and then execute it as a function; conversely, we can use the same name `B` as a function and execute it. If we want to cross the realm from data to function, as it were, we can do that but need to make that explicit; this is where `FUNCALL` comes in:

```
☞ > (setq c (lambda () 42))
  <FUNCTION (LAMBDA ()) {53735E6B}>
☞ > (funcall c)
  42
```

There are tons of technical arguments around the pros and cons of separate namespaces, and Gabriel and Pitman did a quite technical review of the topic [24]; here, it suffices to say that Scheme took the simplest option, there is only one namespace and functions and variables share it:

```
☞ > (define a 42)
```

```
   a 42
☞ > (define (a) 42)
   a
   <procedure a ()>
☞ > (a)
   42
```

Hygienic macros

We saw before that Maclisp introduced `DEFMACRO` to help users write high-level code. Essentially, macros allow you to splice code into templates of sorts, and that comes with problems if the splicer and the splicee use the same names. Consider this code:

```
(defmacro my-or (a b)
  `(let ((tmp ,a))
     (cond
       (tmp tmp)
       (t ,b))))
```

This macro returns `A` if it is true, otherwise `B`. Because either clause can have side effects, it evaluates `A` once and keeps it in a local temporary variable. This way, `A` and `B` are both evaluated not more than once, which is what a caller would expect, and indeed, things work nicely:

```
☞ > (my-or (progn (print "A") t) (print "B"))
   "A"
   T
```

All good, nothing to see, you say? Well… what about this:

```
☞ > (let ((tmp "B"))
       (my-or (progn (print "A") nil) (print tmp)))
   "A"
   NIL
   NIL
```

Now, that wasn't expected! The first clause returns false (`nil`), so the second clause should print "B", not `NIL`. But it's not hard to spot the error when we ask `MACROEXPAND` to tell us what is going on:

```
☞ > (macroexpand '(my-or (progn (print ``A'') nil) (print tmp)))
   (LET ((TMP (PROGN (PRINT ``A'') NIL)))
     (COND (TMP TMP) (T (PRINT TMP))))
```

This code gets spliced into our `(let ((tmp "B")) …)` resulting in the complete expansion:

```
(LET ((TMP ``B''))
  (LET ((TMP (PROGN (PRINT ``A'') NIL)))
    (COND (TMP TMP) (T (PRINT TMP)))))
```

We used the same variable name as the macro and now it's getting all mixed up. Old Lispers probably called this code dirty and the name that stuck was "unhygienic macros". The problem of splicing code in code and getting names mixed up is as old as Lisp itself, and LISP 1.5 already had a solution for this: a function named `GENSYM` which generates a special symbol that can not be defined otherwise and returns a unique symbol every time it gets called. This way, macro writers can generate unique variable names and thus wash them clean and return them to proper hygience. For example:

```
(defmacro my-or (a b)
  (let ((tmp (gensym)))
    `(let ((,tmp ,a))
      (cond (,tmp ,tmp) (t ,b)))))
```

This works much better:

```
☞ > (let ((tmp "B")) (my-or (progn (print "A") nil) (print tmp)))
"A"
"B"
"B"
```

(Note that the second "B" is the return value of the print function which becomes the return value of the whole expression, it is still executed only once). This is what we want and `MACROEXPAND` shows us that with this version, there is no risk of confusion:

```
☞ > (macroexpand '(my-or (progn (print ``A'') nil) (print tmp)))
(LET ((#:G194 (PROGN (PRINT ``A'') NIL)))
  (COND (#:G194 #:G194) (T (PRINT TMP))))
```

The macro uses the generated symbol, in this case `#:G194`, and our calling code's `TMP` is left alone. All is good. However, if you followed along closely you can see that while Lisp changed in how it allows us to do this sort of programming, nasty surprises keep lurking around the corner and it is never completely safe or clean. You need to pay attention to these little gotchas.

* * *

Scheme emphasizes safety and cleanliness, as you can't do language research when the foundation is messy. While Sussman and Steele use plenty of Maclisp macros to implement their language, the language itself and the paper presenting it is silent on the subject. Scheme did not come with macros. Probably, people asked about this because in 1978, in the Revised Report on Scheme [38], macros are introduced. The revised report is also more normative than the original paper and starts talking about mandatory and optional parts of the language—clearly, Scheme was spreading its wings. Macros are introduced with a lot of optionality:

> A SCHEME implementation should have one or more ways for the user to extend the inventory of magic words. The methods provided will vary from implementation to implementation.

The revised version of Scheme comes with two macros: `SCHMAC` is system-dependent for the Maclisp implementation of Scheme on the PDP-10—while ostensibly a Scheme facility, the body of a `SCHMAC` macro is in Maclisp. This makes sense, probably, as Maclisp has access to a lot of (system-dependent) things that Scheme doesn't have, so Scheme can be extended in system-specific ways that are not in scope of a specification. The authors observe, though:

> It so happens, however, that SCHEME is a good meta-language for SCHEME, and so introducing meta-circularity provides no serious problems.

when introducing the Scheme macro definition `MACRO`. Both special forms are just introduced with their syntax and some syntax for quoting and quasiquoting, but no word about the issues above. For that, we need to step forward in time, again, to the Revised Revised Report on Scheme [1], now by a whopping 16 authors–it looks like Scheme was gaining more and more traction. We're in 1985 now, and we read:

> Scheme does not have any standard facility for defining new special forms.

The document goes on to list problems with macros, like the variable capture problem above, and recommends Scheme implementers to "continue to experiment with different solutions". A year later, the third revision is published, but the Revised^3 Report on Scheme [2] does not change its stance on macros. The experimentation continues until, finally, in 1991, the Revised^4 Report on the Algorithmic Language Scheme [3] comes with an appendix on Macros.

> With the appendix to this report Scheme becomes the first programming language to support hygienic macros, which permit the syntax of a block-structured language to be extended reliably.

Finally: hygienic macros. This fourth revision (colloquially abbreviated to R4RS) introduces a way to extend the language in a macro-like way while avoiding its pitfalls, by introducing a new set of special forms `define-syntax`, `let-syntax` and `letrec-syntax` to essentially be the macro versions of `define`, `let` and `letrec`. All of them invoke a pattern matching language called `syntax-rules` that matches up the code to be expanded with expansions. Even though it's a trivial example, let's go back to `my-or`. Here's how it looks like:

```
(define-syntax my-or
  (syntax-rules ()
    ((my-or c1 c2)
      (let ((tmp c1))
        (cond (tmp tmp) (t c2))))))
```

Essentially, what this says is that there's a new rule for the intepreter, and whenever a *pattern* like `(my-or c1 c2)` is encountered, it needs to be replaced with the `(let ....)` below it. That's all, and it works:

```
☞ > (let ((tmp "B"))
      (my-or (begin (display "A") #f) (display tmp)))
A
B
```

The body of `syntax-rules` looks a bit like Scheme, but isn't, and that is a good part of the reason why this works and the equivalent Lisp version doesn't. Syntax rules just pattern match and do a lexical expansion, with a little twist: anything created new, like `tmp` in our let binding, is a newly created unique object; in effect, a mechanism like Lisp's `GENSYM` is working under the hood to ensure that the various `tmp` variables in the example don't clash. However, unlike `GENSYM`, things don't get replaced with on-the-fly generated symbol names. If we expand our code using the mechanism that Scheme provides, something that should not work comes out:

```
☞ > (syntax->datum (expand
☞     '(let ((tmp "B")) (my-or
☞        (begin (display "A") #f) (display tmp)))))
'(let-values (((tmp) '"B"))
   (let-values (((tmp) (begin (#%app display '"A") '#f)))
     (if tmp (let-values () tmp)
```

```
(if '#t (let-values ()
          (#%app display tmp)) (#%app void)))))
```

There's some internals of the scheme implementation peeking through, but the most surprising thing is that all the `tmp` variables print as just `tmp`—for the reader, there is no way to keep them apart. But underneath, each `tmp` lives in its own scope and they are completely different, as the fact that the example works shows.

There's a lot going on around the "simple" pattern matching of Scheme's syntax rules; you can have multiple patterns, declare fixed elements in your syntax, have variable argument lists, etcetera. While not completely as powerful as regular Lisps' macro facilities, it is almost as powerful and usually powerful enough; the trade-off of a little bit of lost expressive power for a ton of safety and a clean macro model that doesn't rely on mixes of quoting, quasiquoting and unquoting is probably a good one for Scheme's purposes of simplicity and clarity.

Scheme in use

Scheme found widespread use in teaching and research, which was pretty much the stated goal of its creation. Reports with revisions have been coming out at a regular pace, and at the time of writing the 7th revision [25] is current—this revision is called "the small edition" as it trimmed a lot of bloat that the language accrued over the years; a "large edition" is in the works. On top of R7RS there is an extensive list of Scheme Requests for Implementation or SRFIs [73]. Scheme implementations will often implement a base standard and then a list of SRFIs.

Structure and Interpretation of Computer Programs

Probably still the most widely-known use of Scheme was MIT course 6.001. an introductory course for undergraduates into the structure and interpretation of computer programs that was taught for over twenty years. The course book, Structure and Interpretation of Computer Programs [4] shows on many more bookshelves than just MIT CompSci alumni—many professional developers who take a deep interest in their craft get the book and work through it.

To say that "SICP" and 6.001 are a fast-paced course is understating it. In the time of just one quick semester, students were taken from "`(+ 137 349)` equals `486`" to implementing a compiler for register machines. Even for software developers with a decade or more of professional experience under their belts, the book is fast, deep,

and more often than not, hard. But on top of all that shines elegance, where simple concepts like lexical closures are put to work. It's worth showing a little bit of the elegance that "SICP" teaches.

* * *

Most programming books start with mutating state. Let's say the aspiring coder picked up a BASIC book, there's a good chance that early on, something like this will happen:

```
10 FOR I = 1 TO 10
20 PRINT "Hello, World!"
30 NEXT I
```

The loop counter variable `I` here is assigned over and over again. It doesn't show immediately, but `FOR` is shorthand for:

```
10 LET I = 1
20 PRINT "Hello, World!"
30 I = I + 1
40 IF I <= 10 THEN GOTO 20
```

Remember how we "proved" that tail-recursion was iterative by simply applying substitution? That will not work here. Changing the value of a variable breaks the model of computation-by-substitution, it comes as it were with Dante's warning in his Inferno, "all hope abandon ye who enter here". A much more complex model of computation is needed, and SICP therefore does not treat the whole concept (packaged neatly in Scheme's `set!` function) until the reader is ready for somewhat advanced topics well into chapter 3 (out of the book's 5 chapters). By that time, the student has enhanced Scheme with a simple type system and implemented dispatching on types, then added a flexible way of computing with complex and rational numbers using such a system. All using functions that are *referentially transparent*, or, in other words, work with the substitution model.

Even then, the authors are careful to build something elegant and reusable. If you know BASIC, try to envision this example from the book in such a completely imperative language:

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

to show that it works, we fire up the Guile implementation of Scheme:

```
☞ > (define acct1 (make-withdraw 50))
☞ > (acct1 10)
  $5 = 40
☞ > (acct1 10)
  $6 = 30
☞ > (acct1 50)
  $7 = "Insufficient funds"
```

The mutated state here, of course, is the original `balance` argument to `make-withdraw`, and it remains visible as the closure of the returned lambda for the latter's lifetime. There is assignment involved, but it is encapsulated in a nicely wrapped package instead of being out in the open; the big benefit, of course, is that we can have as many accounts as we want:

```
☞ > (define acct2 (make-withdraw 100))
☞ > (define acct3 (make-withdraw 50))
☞ > (acct3 40)
  $8 = 10
☞ > (acct2 40)
  $9 = 60
☞ > (acct3 20)
  $10 = "Insufficient funds"
☞ > (acct2 20)
  $11 = 40
```

By turning something as, well, "basic" as mutable variables into an advanced problem that requires a newer, more complex model of computation (also explained in the book), the authors make sure that even seasoned developers think twice about such dangerous stuff.

Racket

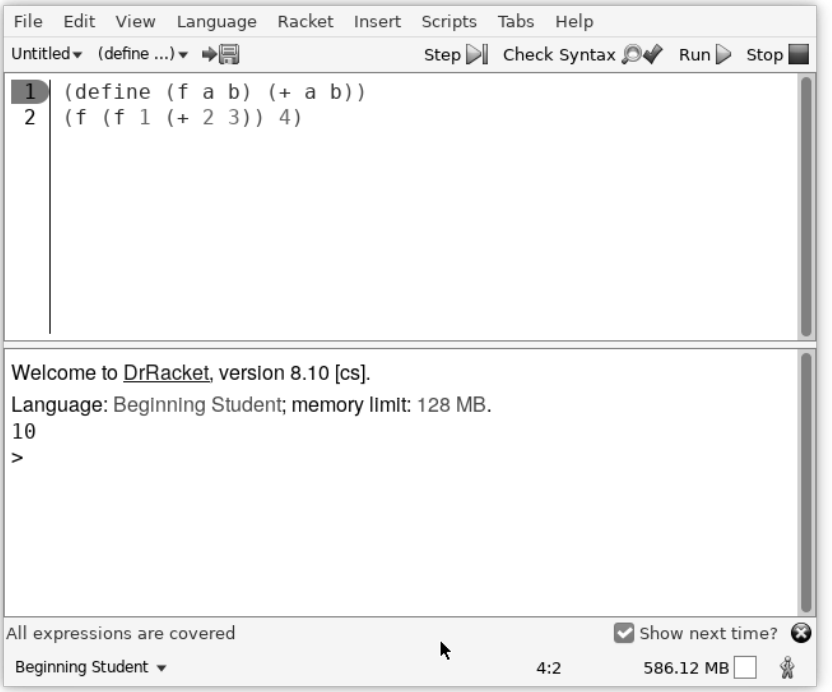
Scheme is like a scalpel, and Racket hones that scalpel to be a precise tool for language research, Scheme's original focus. Scheme, as a language, got something of a following in educational circles, and one of the implementations was PLT Inc.'s implementation that purely focused on producing pedagogical materials using Scheme as a teaching language. As the focus was on the materials, not the actual engine, PLT Scheme was essentially an environment built out of a collection of open source parts, like an existing Scheme library, the WX Widget library to add a graphical environment, and so on.

The engine was called "DrScheme" and the flavour of Scheme it implemented was called PLT Scheme. In 2010, the language was renamed "Racket" and the engine "DrRacket" and that is what we're talking about today. DrRacket is available for pretty much every platform under the sun, and gives an excellent and very user friendly integrated development environment for writing Lisp, both for educational purposes as well as general purpose programming. It is freely available from the Racket website [66], and you are encouraged to install it and follow along.

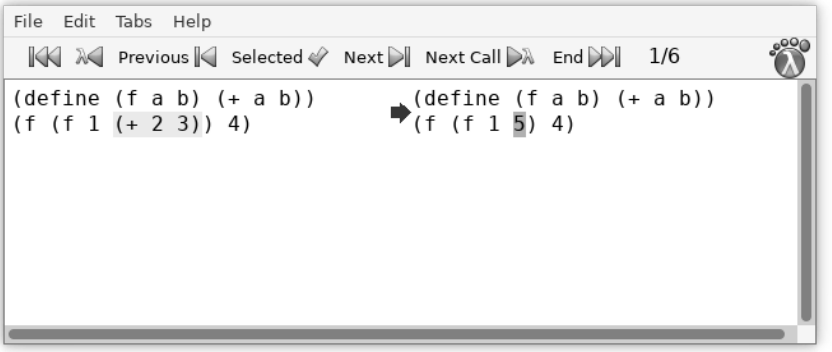
Scheme, like any Lisp, easily morphs into other things, so quickly new languages were added to support the pedagogical mission; for example, five levels of Scheme starting with a small Scheme for beginners building up all the way to advanced topics like "mutable state". On top of this, tools were added to help students understand what was happening, like an algebraic stepper that lets you, patiently like only a computer can do, step one by one through code showing exactly what is going on. Say, for example, we have a simple program:

```
(define (f a b) (+ a b)) (f (f 1 (+ 2 3)) 4)
```

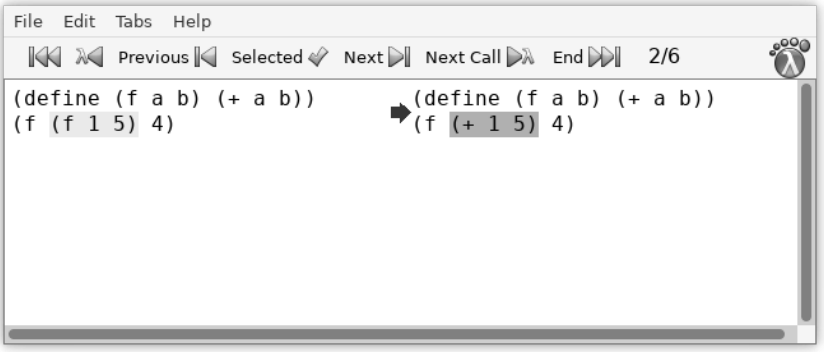
If we load this in the "Beginning Student" language and run it, we get "10". That's not extremely interesting, but the stepper allows us to see exactly how the answer is obtained, and that's a fun and useful thing to show:



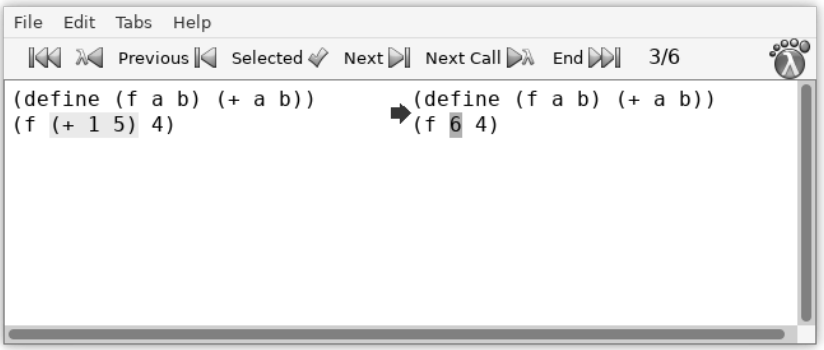
As you can see, we have the program entered, we ran it, we got 10, but now we hit the "Step" button in the toolbar:



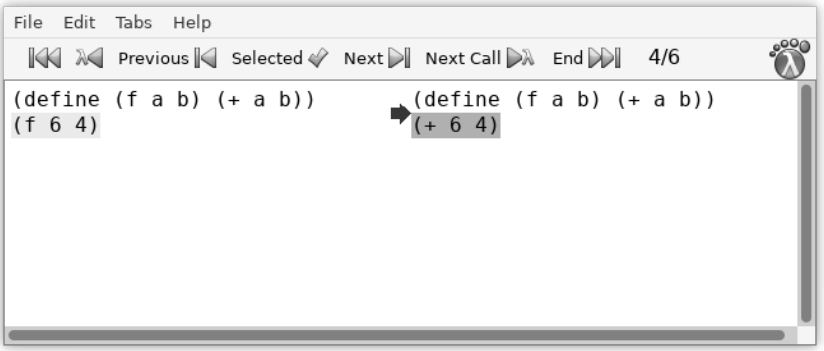
DrRacket opens up a new window and shows us that it started by evaluating `(+ 2 3)`, highlighted in green, and replaced that with `5`, the result of that expression. We hit "Next", and get:



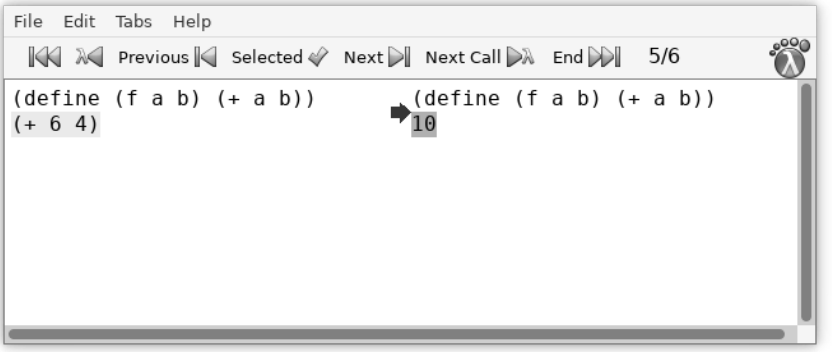
As you can see, Racket now expands the function definition—it replaces `(f 1 5)` with `(+ 1 5)`, which is the sort of mechanical replacement action we read about when discussing lambda calculus. Hitting "Next" again shows the expected replacement of that code with 6:



Which leaves us with just `(f 6 4)`, which another "Next" expands to the body of our little function:



And finally, we take the step to arrive at the expected answer, 10:



Racket's power is twofold: at every corner, it tries to be your guide. It comes with built-in debugging tools like the stepper, but also normally has much nicer error messages than most Lisps out there and comes with a development environment that does not require a PhD in computer science to operate. Next, it is not just a Scheme implementation but more of a language platform: not only can it run a whole list of variations on the Scheme theme, but it even comes with an Algol 60 implementation and domain-specific languages like "Video", for programmatically manipulating movies. It really epitomizes Lisp's core philosophy of "if you have a problem, first create a language to solve the problem, then solve the problem."

Where Scheme and its implementations like Racket are usually targeted at Lisp-in-the-small (although Racket, for example, comes with its own powerful module system to allow for building larger systems) and specific like a scalpel, some people want to build in the large and like Swiss Army knives better. The Swiss Army knife of Lisp is the system we'll look at next, Common Lisp.