SICP sections 2.3.1 - 2.3.2

August 30, 2007 at 05:17 **Tags** SICP

The solutions for these sections are coded in Common Lisp.

Section 2.3.1

First of all, a few words on equality. Until this point, all the data we worked on in the book were numbers, which are compared with the `=` operator, both in Common Lisp and Scheme. Now, we'll also have symbolic data and lists to consider for equality.

The equality operators in Common Lisp and Scheme are quite different. I've written in length about the various equality operators in CL. For example, the best candidate for an equivalent to Scheme's `eq?` that's used in the definition of `memq` is the CL function `eql` (which compares symbols, numbers and characters). Here is `memq`:

```
(defun memq (item x)
  (cond ((null x) nil)
        ((eql item (car x)) x)
        (t (memq item (cdr x)))))
```

Exercise 2.53

```
(list 'a 'b 'c)
==> (A B C)

(list (list 'george))
==> ((GEORGE))

(cdr '((x1 x2) (y1 y2)))
==> ((Y1 Y2))

(cadr '((x1 x2) (y1 y2)))
==> (Y1 Y2)

(consp (car '(a short list)))
==> NIL

(memq 'red '((red shoes) (blue socks)))
==> NIL

(memq 'red '(red shoes blue socks))
==> (RED SHOES BLUE SOCKS)
```

Exercise 2.54

The `equal?` wanted here is the `equal` function of CL. Here's an implementation, as requested:

```
(defun equal? (la lb)
  (cond
    ((and (symbolp la) (symbolp lb))
      (eql la lb))
    ((symbolp la) (symbolp lb))
    ((symbolp lb) (symbolp la))
    ((null la) (null lb))
    ((null lb) (null la))
    (t (and
        (equal? (car la) (car lb))
        (equal? (cdr la) (cdr lb))))))
```

Exercise 2.55

```
(car ''abracadabra)
==> QUOTE
```

As footnote 34 in the book says, the quote character is just an abbreviation for the special operator `quote`. In fact, in the environment it is just syntactic sugar that is translated to`quote` internally. So we can see the statement entered above as equivalent to:

```
(car '(quote abracadabra))
```

Which, unsurprisingly, returns `QUOTE`.

Section 2.3.2

As in the picture language from section 2.2.4, this section presents an example of a beautifully designed system. To quote the authors:

> To embody these rules in a procedure we indulge in a little wishful thinking, as we did in designing the rational-number implementation. If we had a means for representing algebraic expressions, we should be able to tell whether an expression is a sum, a product, a constant, or a variable. We should be able to …

This *wishful thinking* is one of the most powerful methods to design software systems (any systems, really)[1].

Here is an implementation of the basic derivation, without simplification:

```
(defun variable? (x)
  (symbolp x))

(defun same-variable? (v1 v2)
  (and
    (variable? v1)
    (variable? v2)
    (eql v1 v2)))

(defun make-sum (a1 a2)
  (list '+ a1 a2))

(defun make-product (m1 m2)
  (list '* m1 m2))

(defun sum? (x)
  (and (consp x) (eql (car x) '+)))

(defun addend (s)
  (cadr s))

(defun augend (s)
  (caddr s))

(defun product? (x)
  (and (consp x) (eql (car x) '*)))

(defun multiplier (s)
  (cadr s))

(defun multiplicand (s)
  (caddr s))

(defun deriv (expr var)
  (cond ((numberp expr) 0)
        ((variable? expr)
          (if (same-variable? expr var) 1 0))
        ((sum? expr)
          (make-sum (deriv (addend expr) var)
                    (deriv (augend expr) var)))
        ((product? expr)
          (make-sum
            (make-product (multiplier expr)
                          (deriv (multiplicand expr) var))
            (make-product (deriv (multiplier expr) var)
                          (multiplicand expr))))
        (t
          (error "unknown expression type -- DERIV ~A" expr))))
```

And these are the simplification functions:

```
(defun =number? (expr num)
  (and (numberp expr) (= expr num)))

(defun make-sum (a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (numberp a1) (numberp a2)) (+ a1 a2))
        (t (list '+ a1 a2))))

(defun make-product (m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (numberp m1) (numberp m2)) (* m1 m2))
        (t (list '* m1 m2))))
```

Exercise 2.56

```
(defun make-exponentation (base exp)
  (cond ((=number? exp 0) 1)
        ((=number? exp 1) base)
        ((and (numberp base) (numberp exp))
          (expt base exp))
        (t (list '** base exp))))

(defun exponentation? (x)
  (and (consp x) (eql (car x) '**)))

(defun base (s)
  (cadr s))

(defun exponent (s)
  (caddr s))

(defun deriv (expr var)
  (cond ((numberp expr) 0)
        ((variable? expr)
          (if (same-variable? expr var) 1 0))
        ((exponentation? expr)
          (make-product
            (make-product
              (exponent expr)
              (make-exponentation
                      (base expr)
                      (1- (exponent expr))))
            (deriv (base expr) var)))
        ((sum? expr)
          (make-sum (deriv (addend expr) var)
                    (deriv (augend expr) var)))
        ((product? expr)
          (make-sum
            (make-product (multiplier expr)
                          (deriv (multiplicand expr) var))
            (make-product (deriv (multiplier expr) var)
                          (multiplicand expr))))
        (t
          (error "unknown expression type -- DERIV ~A" expr))))
```

Note that I've also added a test in `make-exponentation` that checks if both the base and the exponent are number and computes the result directly calling CL's built-in function `expt`.

Exercise 2.57

First let's see how to define sums (products are very similar):

```
(defun make-sum (&rest nums)
  (append (list '+) nums))
```

Here I'm using CL's `&rest` argument. It passes all the arguments it receives as a list into the function. For example:

```
(defun foo (&rest args) args)
(foo 1 2 3)
==> (1 2 3)
```

Therefore, it is possible to create sums as follows:

```
(make-sum 1 'x 4)
==>
'(+ 1 x 4)
```

Let's see the other building blocks of the representation:

```
(defun sum? (x)
  (and (consp x) (eql (car x) '+)))

(defun addend (s)
  (cadr s))

(defun augend (s)
  (let ((aug (cddr s)))
    (if (= (length aug) 1)
      (car aug)
      (append (list '+) aug))))
```

`sum?` and `addend` remain unchanged. `augend` changes, however. As the authors proposed, it returns the sum of the rest of the terms (when there's more than two terms). Note that I use `append` directly, and not `make-sum` inside `augend`. There is a reason for it, and it's subtle. `make-sum` is using `&rest` which means it accepts multiple arguments and turns them into a list. But multiple arguments are not the same as a list, and when we pass a list into it, we don't get what we expect. This is easily demonstrable:

```
(defun foo (&rest args) args)
(foo 1 2 3)
==> (1 2 3)

(foo '(1 2 3))
((1 2 3))
```

As you see, the two calls to `foo` give different results, as expected. Since in `augend` we only have a list to pass, we can't use `make-sum` and must use `append` directly.

And this is the code for products (note that this code is without simplification, which is considerably more difficult when each expression has multiple operands):

```
(defun make-product (&rest nums)
  (append (list '*) nums))

(defun product? (x)
  (and (consp x) (eql (car x) '*)))

(defun multiplier (s)
  (cadr s))

(defun multiplicand (s)
  (let ((m (cddr s)))
    (if (= (length m) 1)
      (car m)
      (append (list '*) m))))
```

Inded, `deriv` can remain unchanged.

Exercise 2.58

a.

This part is actually quite easy. If we assume input correctness[2], the only thing that has changed is the ordering of operands inside expressions. In the earlier versions, the operator came first. Now it's in the middle.

These are the new accessors (`deriv` and everything else stays unmodified):

```
(defun make-sum (a1 a2)
  (list a1 '+ a2))

(defun make-product (m1 m2)
  (list m1 '* m2))

(defun sum? (x)
  (and (consp x) (eql (cadr x) '+)))

(defun addend (s)
  (car s))

(defun augend (s)
  (caddr s))

(defun product? (x)
  (and (consp x) (eql (cadr x) '*)))

(defun multiplier (s)
  (car s))

(defun multiplicand (s)
  (caddr s))
```

b.

This is quite a bit trickier. Parsing arbitrary infix expressions, respecting the rules of operator

precedence and parenthesizing is far from being simple. I'm pretty sure this is not what the authors of SICP meant, but to solve this exercise I wrote a full blown recursive-descent parser with a stack of token streams. The most important thing is that I had loads of fun writing it :-)

What we have here is a simple formal grammar that can be described in BNF notation:

```
;;
;; <sum>    -> <term> + <sum>
;;           | <term>
;;
;; <term>   -> <factor> * <term>
;;           | <factor>
;;
;; <factor> -> symbol
;;           | number
;;           | (<sum>)
;;
```

What I wrote is a parser that converts expressions in infix notation to expressions in standard Lisp prefix notation, with two operands per operator. Essentially, a parse tree. This parse tree is exactly the format the original `deriv` wants, so problem solved. Here is the whole parsing code:

```
;; A stream is this case is a Lisp list,
;; with all operators in infix notation.
;; "Scanning" the stream is just popping
;; the next list element off the front.
;;
(defvar *stream* '() "Token stream")

(defun init-stream (stream)
  "Initializes the stream"
  (setq *stream* stream))

(defun next-token ()
  "Returns the next token of the stream"
  (car *stream*))

(defun scan ()
  "Scans the stream to 'find' the next token"
  (pop *stream*))

;; In order to parse nested infix expressions,
;; we must keep a stack of streams, since
;; the stream keeps state for the parsing calls.
;;
(defvar *stream-stack* '() "Stack of streams")

(defun push-stream (stream)
  "Push the current *stream* on stack, and set
  this stream as *stream*"
  (push *stream* *stream-stack*)
  (init-stream stream))
```

```
(defun pop-stream ()
  "Pop a stream from the stack and set it to
  *stream*"
  (init-stream (pop *stream-stack*)))

;;
;; BNF grammar:
;;
;; <sum>    -> <term> + <sum>
;;           | <term>
;;
;; <term>   -> <factor> * <term>
;;           | <factor>
;;
;; <factor> -> symbol
;;           | number
;;           | (<sum>)
;;
;;

(defun parse-factor ()
  (let ((tok (next-token)))
    (cond
      ((or (numberp tok) (symbolp tok))
        (scan)
        tok)
      ((listp tok)
        (push-stream tok)
        (let ((sum (parse-sum)))
          (pop-stream)
          (scan)
          sum))
      (t
        (error "Bad token in parse-atom -- ~A" tok)))))

(defun parse-term ()
  (let ((lfact (parse-factor)))
    (if (eq (next-token) '*)
      (progn
        (scan)
        (let ((rterm (parse-term)))
          (list '* lfact rterm)))
      lfact)))

(defun parse-sum ()
  (let ((lterm (parse-term)))
    (if (eq (next-token) '+)
      (progn
        (scan)
        (let ((rsum (parse-sum)))
          (list '+ lterm rsum)))
      lterm)))
```

```
      (term)))
```

The "main" function here is `parse-sum`. It converts an expression from infix to prefix notation. Now, to complete the derivation code for this representation, all I did was rename the function `deriv` to `deriv-prefix`, and created the new function:

```
(defun deriv-infix (expr var)
  (init-stream expr)
  (deriv-prefix (parse-sum) var))
```

Yeah, I know, I know that I should have changed the accessors and kept `deriv` intact. It's possible, but it's just more more, so since the example is very artificial anyway, this will do.

A few words on the parser implementation. Most of the code is very straightforward (your usual textbook RD parser). The only thing that isn't immediately obvious is why the stack of streams is needed. To understand this, recall how Lisp treats parenthesized expressions:

```
(car '(1 * (2 + 3) * 4))
=> 1

(caddr '(1 * (2 + 3) * 4)))
=> (2 + 3)
```

So, Lisp automatically treats parentheses not like other symbols (tokens), but rather as specifiers for internal lists.

Now, to match a parenthesized `sum` in `factor` (see the BNF above), I must first match a left paren '(', then a `sum`, then a right paren ')'. But there is no way to "match" a paren in a Lisp list because Lisp treats them in a special way !

There are a few approaches to solve this. I could convert the whole thing to a string and employ string processing. But why give up all the comfort of Lisp lists ? I could also insert special "open brace" and "end brace" symbols around the parens. This would be easier, but still quite kludgy.

The solution I eventually chose is IMHO the most elegant. I created a so-called *stack automata*. My parser walks the stream of tokens, and when it meets a sub-expression it parses it recursively. It does this by setting the sub-expression as a new stream to work on. However, the old stream must be preserved, because after we're finished with the sub-expression we must come back to the same place we left. This is why it is pushed on a stack, thus providing unlimited nesting capabilities.

[1] Sometimes I see negative comments about SICP here and there online. One of the most common complaints (apart from the one on using Scheme and not a "modern" language like Java, sigh) is that SICP doens't really teach how to program, but is just a bunch of unrelated trivia items. Those commenters must have never read SICP, it appears to me, since this book is just packed solid with amazingly thoughtful advice on how to design and write software. I wish there were other books like it.

[2] Assuming input correctness is a common trick used by programmers to make their life easier in exploratory and prototype programming.

For comments, please send me an email.

↑ Back to top