

Principles of Programming Languages

Higher Order Functions in Scheme and Local Variables

We introduced *L1* which supported:

- **Atomic expressions** – numbers, booleans, primitive procedures
- **Special forms:** `define` expressions
- **Non-special forms:** `(<exp> ... <exp>)`

We then introduced *L2* which added:

- User-defined procedures (closures)
- **Special forms:** `lambda`, `if`, `cond`

Lastly, we introduced *L3* which adds two new values to our language: Pairs and Lists. We added:

- Primitive procedures: `cons`, `car`, `cdr`, `pair?`, `list?`, `equal?`
- The special literal for the empty list: `'()`
- Literal expressions: `'(<lit> . <lit>)` and `'(<lit> ... <lit>)`.

The possible computed values in $L3$ are therefore:

- Numbers
- Booleans
- Primitive procedures
- Closures
- Pairs
- Lists

Can we define higher-order functions in $L3$?

For all the procedures we will define in L_i from this point on, we will attach a contract in the form of formatted comments. The contract includes at least:

;; Signature:

;; Type:

;; Purpose:

Within these comments, we use a type language to define the type of expressions which is extremely close to the TypeScript type language - but restricted to the data types that we have defined in our language - Pair and List instead of Arrays and Maps.

For function types, we write

$[T1 * T2 \rightarrow T3]$ for a function taking 2 parameters of type $T1$ and $T2$ and returning a value of type $T3$. We denote type variables by using type names that start with a capital T . In the next Chapter, we will formalize this type language and implement meta-programs operating over it (type checker and type inference systems).

Let us try to define our favorite higher-order functions:

```
;; Signature: map(f, s)  
;; Type: [[T1 -> T2] * List(T1) -> List(T2)]  
;; Purpose: Apply f to all elements in s  
;;           and return the list of the results.  
(define map  
  (lambda (f s)  
    (if (empty? s)  
        '()  
        (cons (f (car s))  
              (map f (cdr s))))))
```

The key part to verify according to the evaluation rules - is that the invocation of `(f (car l))` will be evaluated as expected according to the evaluation rules of *L3*.

filter

```
;; Signature: filter(pred, s)  
;; Type: [[T -> Boolean] * List(T) -> List(T)]  
;; Purpose: Return the list of elements  
;;           in s that satisfy pred.  
(define filter  
  (lambda (pred s)  
    (if (empty? s)  
        '()  
        (if (pred (car s))  
            (cons (car s) (filter pred (cdr s)))  
            (filter pred (cdr s))))))
```

reduce

```
;; Signature: reduce(reducer, initial, s)  
;; Type: [[T1 * T2 -> T2] * T2 * List(T1) -> T2]  
;; Purpose: Combine all the values of s using reducer  
;; Example: (reduce + 0 '(1 2 3)) => (+ 1 (+ 2 (+ 3 0)))  
(define reduce  
  (lambda (reducer initial s)  
    (if (empty? s)  
        initial  
        (reducer (car s)  
                  (reduce reducer initial (cdr s))))))
```

Compare this first definition of **reduce** with this one:

reduce

```
;; Signature: reduce2(reducer, initial, s)  
;; Type: [[T1 * T2 -> T2] * T2 * List(T1) -> T2]  
;; Purpose: Combine all the values of s using reducer  
;; Example: (reduce2 + 0 '(1 2 3)) => ?  
(define reduce2  
  (lambda (reducer initial s)  
    (if (empty? s)  
        initial  
        (reduce2 reducer  
                  (reducer (car s) initial)  
                  (cdr s)))))
```

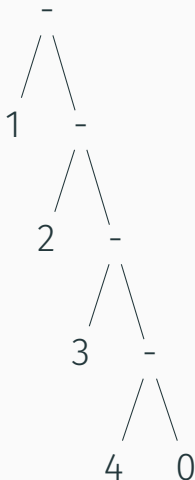
Are the two functions `reduce` and `reduce2` equivalent?

No. For example:

`(reduce - 0 '(1 2 3 4)) ;; ==> -2`

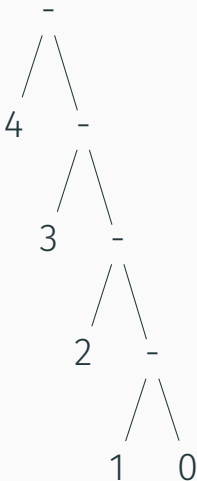
`(reduce2 - 0 '(1 2 3 4)) ;; ==> 2`

reduce creates this tree:



reduce

While **reduce2** creates this tree:



Let us observe the process of defining functional abstractions through abstraction of repeated patterns of code.

Procedures as Parameters

Consider a procedure that computes the sum of the numbers in a range $[a, b]$:

```
;; Signature: sum-integers(a, b)  
;; Type: [Number * Number -> Number]  
;; Purpose: Compute the sum of  
;;           all integers a to b  
(define sum-integers  
  (lambda (a b)  
    (if (> a b)  
        0  
        (+ a (sum-integers (+ a 1) b)))))
```

Procedures as Parameters

Let us then define a procedure to compute the sum of the cubes of all the numbers in $[a, b]$:

```
(define cube (lambda (x) (* x x x)))
```

```
;; Signature: sum-cubes(a, b)
```

```
;; Type: [Number * Number -> Number]
```

```
;; Purpose: Compute the sum of the cube
```

```
;;           of all integers a to b
```

```
(define sum-cubes  
  (lambda (a b)  
    (if (> a b)  
        0  
        (+ (cube a)  
            (sum-cubes (+ a 1) b)))))
```

Consider yet a third function - used to compute a numerical approximation of π using the formula:

$$\frac{1}{a(a+2)} + \frac{1}{(a+4)(a+6)} + \frac{1}{(a+8)(a+10)} + \dots$$

This formula converges to $\frac{\pi}{8}$ when starting with $a = 1$.

Procedures as Parameters

```
;; Signature: pi-sum(a,b)  
;; Type: [Number * Number -> Number]  
;; Purpose: compute the approximation of pi/8  
(define pi-sum  
  (lambda (a b)  
    (if (> a b)  
        0  
        (+ (/ 1 (* a (+ a 2)))  
            (pi-sum (+ a 4) b)))))
```

Let us now try to **generalize** the structure of these 3 procedures - that is, define a functional abstraction that describes the commonality between these 3 functions.

We observe the following repeated pattern:

```
(define <name>
  (lambda (a b)
    (if (> a b)
        0
        (+ (<term> a)
            (<name> (<next> a) b)))))
```

Procedures as Parameters

Based on this observation, we define a functional abstraction:

```
;; Signature: sum(term, a, next, b)  
;; Type: [[N -> N] * N * [N -> N] * N -> N]  
;; Purpose: Compute the sum:  
;;          (term a) + (term (next a)) + ... + (term n)  
;;          where n = (next (next (... (next a)))) <= b  
;;          and (next n) > b.  
(define sum  
  (lambda (term a next b)  
    (if (> a b)  
        0  
        (+ (term a)  
            (sum term (next a) next b)))))
```

Procedures as Parameters

We can now redefine the 3 procedures above using the new **sum** abstraction:

```
(define sum-integers
  (lambda (a b)
    (sum identity a add1 b)))

(define sum-cubes
  (lambda (a b)
    (sum cube a add1 b)))

(define pi-sum
  (lambda (a b)
    (sum pi-term a pi-next b)))

(define pi-term
  (lambda (x)
    (/ 1 (* x (+ x 2)))))

(define pi-next
  (lambda (n) (+ n 4)))
```

Given this new functional abstraction **sum** - we can define other functions. For example, a numerical approximation of the definite integral of a numerical function using the formula:

$$\int_a^b f(x)dx = \left[\sum_{n=1}^{\frac{b-a}{dx}} f\left(a + n \cdot dx + \frac{dx}{2}\right) \right] dx$$

Procedures as Parameters

As we spot a sum in this formula, the `sum` abstraction is relevant for the implementation:

```
(define dx 0.001)
(define add-dx (lambda (x) (+ x dx)))
```

```
;; Signature: integral(f, a, b)
;; Type: [[N -> N] * N * N -> N]
;; Purpose: Compute an approximation of the
;;          definite integral of f in [a, b].
```

```
(define integral
  (lambda (f a b)
    (* (sum f (+ a (/ dx 2)) add-dx b) dx)))
```

Up to this point, we only defined global variables using the **define** special form.

Local variables are a programming language feature which encourages programmers to limit the scope within which variables are known, and to avoid *hidden dependencies* between global variables and functions that refer to them.

When defining a local variable, we must provide:

- A way to name the variable
- How to initialize the variable
- A way to decide what is the scope of the variable - that is, what is the part of the program where this variable is visible.

Up to this point, we observed variables in only two contexts:

- In `define` expressions:
`(define <var> <exp>)`
- In `lambda` expressions:
`(lambda (<var> ...) <exp> ...)`

We first define the notions of **scope**, and **bound** and **free** variable occurrences before introducing local variables in the language.

A **lambda** form includes parameters and a body. Within a **lambda** expression, the body is the **scope** of the parameters - this means that all the variables that occur within the body are intended to be **bound** to the parameters. This is the case even if there is preceding definition of a variable with the same name.

For example:

```
(define x 1)
(lambda (x) (* x x))
```

Occurrences of **x** in the body refer to the parameter variable **x** and **not** to the global variable **x**.

In such a case, we say that the occurrences of x in the body of the lambda expression are **bound occurrences**. Other variables are said to occur **free**.

Parameters, Scope, Bound and Free Variable Occurrences

For example, in the expression:

```
(lambda (f a b dx)
  (* (sum f
        (+ a (/ dx 2.0))
        (lambda (x) (+ x dx))
        b)
    dx))
```

the variables **f**, **a**, **b**, **dx** all occur **bound** - while
the variable **sum** occurs **free**.

Consider the computation of the following function:

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$$

Deriving the let Shortcut Notation

Let us implement a Scheme procedure to compute this function:

```
(define f
  (lambda (x y)
    (+ (* x
          (square (+ 1 (* x y))))
       (* y
          (- 1 y)))
    (* (+ 1 (* x y))
       (- 1 y)))))
```

Deriving the `let` Shortcut Notation

We observe that the same sub-expressions are repeated: `(+ 1 (* x y))` and `(- 1 y)`.

This not only makes it more difficult to read the code - it also makes the computation slower - because these expressions will actually be computed twice each time the function is invoked.

One way to avoid these repetitions is to **abstract away** the repeated sub-expressions - in exactly the same way we abstracted away the repeated sub-expressions when defining the sum abstraction.

Deriving the `let` Shortcut Notation

Our abstraction mechanism involves defining new functions:

```
(define a  
  (lambda (x y)  
    (+ 1 (* x y))))
```

```
(define b  
  (lambda (y)  
    (- 1 y)))
```

Deriving the `let` Shortcut Notation

Using these two helper functions, we can rewrite `f` as:

```
(define f
  (lambda (x y)
    (+ (* x (square (a x y)))
       (* y (b y))
       (* (a x y) (b y)))))
```

The problem is that we need to define names for these helpers - and define new functions for each one.

Alternatively - we can proceed as follows:

Let $a = 1 + xy$, $b = 1 - y$, and define f as

$$f(x, y) = xa^2 + yb + ab$$

Deriving the `let` Shortcut Notation

Which in Scheme is implemented as:

```
(define f
  (lambda (x y)

    ((lambda (a b)
      (+ (* x (square a))
         (* y b)
         (* a b))))

    (+ 1 (* x y))
    (- 1 y))))
```

Deriving the `let` Shortcut Notation

This structure is convenient - it allows us to define local variables **a** and **b** which are defined within the scope of **x** and **y** and are only used within a single expression.

The syntactic form of the definition of local variables is not readable - because the value which initializes the local variables (**a** and **b**) are far away from their declaration.

The Scheme language defines a **syntactic abbreviation** which is internally turned into the same lambda application form called the `let` form to encourage programmers to use this construct.

The `let` Abbreviation

The structure of the `let` expression is:

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      ...
      (<varn> <expn>))
  <body1> ...)
```

The let Abbreviation

Internally, this form is replaced by the equivalent syntactic form:

```
((lambda (<var1> ... <varn>)
  <body1> ...)
 <exp1> ... <expn>)
```

Hence its evaluation rule is already defined as per the rules of *L3* evaluation.

The `let` Abbreviation

So, for our example, the code would look like this:

```
(define f
  (lambda (x y)
    (let ((a (+ 1 (* x y)))
          (b (- 1 y)))
      (+ (* x (square a))
         (* y b)
         (* a b)))))
```

The `let` Abbreviation

Still, it is useful to remember the way `let` forms are evaluated:

- The `let` variables have scope over the body of the `let` only.
- Each `<vari>` is bound to the value of each `<expi>`.
- All the bindings are performed *simultaneously* - they do not depend on each other.

The `let` Abbreviation

- The initial values `<expi>` are computed *outside the scope* of the `let`.
- The evaluation of the `let` form involves the creation of closure value and its application to the initial values.

We have seen so far examples of higher order procedures that receive procedures as parameters. Let us now review cases where procedures return computed procedures.

Let us first work out the mechanics of procedures returning procedure values:

We first start from an expression that has a concrete value (a number) and abstract it away on each of the variables that occur in the expression:

Returning a Closure

```
(define x 3)
```

```
(define y 0)
```

```
(+ x y y)
```

```
;; => 3
```

```
(lambda (y) (+ x y y))
```

```
;; => <Closure (y) (+ x y y)>
```

```
(lambda (y) (lambda (x) (+ x y y)))
```

```
;; => <Closure (y) (lambda (x) (+ x y y))>
```

Returning a Closure

Now, let us apply these closure values:

```
((lambda (x) (+ x y y)) 5)
;; => 5
((lambda (y) (lambda (x) (+ x y y))) 2)
;; => <Closure (x) (+ x 2 2)>
(((lambda (y) (lambda (x) (+ x y y))) 2) 5)
;; => 9
```

The important point to notice is that when a Closure value is constructed, the value of the free variables is substituted in the body of the closure. For example, in the second expression above, the closure value has no free variable `y`

- but the occurrences of `y` have been substituted by 2.

Derivative Example

We design a function which given a numeric function as a parameter, returns a new function which computes a numerical approximation of the derivative of the function - using the formula:

$$f'(x) = \frac{f(x + dx) - f(x)}{dx}$$

Derivative Example

```
;; Signature: derive(f, dx)  
;; Type: [[N -> N] * N -> [N -> N]]  
;; Purpose: Construct a function that computes  
;;          a numerical approx. of the  
;;          derivative of f with resolution dx.  
(define derive  
  (lambda (f dx)  
    (lambda (x)  
      (/ (- (f (+ x dx))  
            (f x))  
         dx))))
```

Derivative Example

When this function is evaluated, it returns a Closure value which *remembers* the values of `f` and `dx` that were used.

```
(let ((f1 (deriv square 0.001)))  
  (f1 2)) ;; => 4.00099999999999699
```

Derivative Example

We can iterate this procedure to compute the
nth derivative of a numerical function:

```
;; Signature: nth-deriv(f, n)  
;; Type: [[N -> N] * N -> [N -> N]]  
;; Purpose: construct a function that computes  
;;           a numerical approx.  
;;           of the nth derivative of f  
(define nth-deriv  
  (lambda (f n)  
    (lambda (x)  
      (if (= n 0)  
          (f x)  
          ((nth-deriv (derive f 0.0001) (- n 1)) x)))))
```

Derivative Example

Consider an alternative definition:

```
(define nth-deriv-early
  (lambda (f n)
    (if (= n 0)
        f
        (derive (nth-deriv-early f (- n 1)) 0.0001))))
```

The time at which the closures are created is different. Trace the behavior and comment on which version is more desirable.