Contents

1 noweb

2 SICP [14/362]

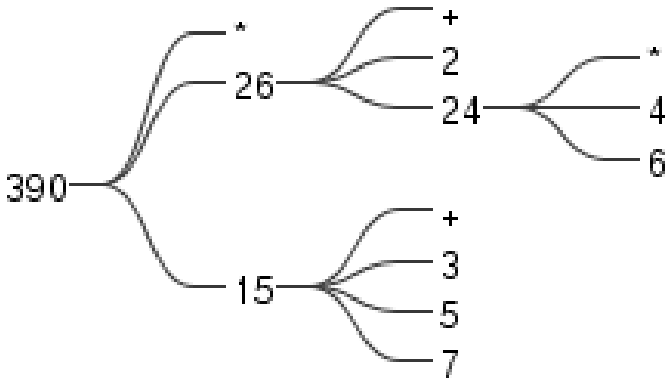
`header-args: :noweb yes`

1. **TODO** Chapter 1: Building abstractions with procedures [14/47]

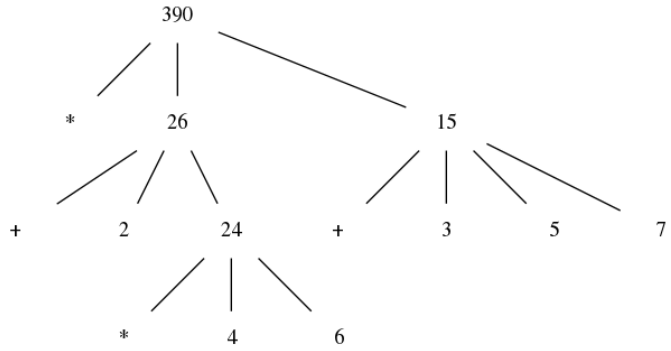
   (a) Snippet language=Lisp,label= ,caption= ,captionpos=b,numbers=none
   (* (+ 2 (* 4 6)) (+ 3 5 7))

   (b) Thought Tree accumulation is the process of computing a thing by traversing a tree.

   (c) **DONE** Figure 1.1 **CLOSED:** *[2019-08-20 Tue 14:35]*
   For the sake of pedagogical clarity, I have formatted it as a picture.



   ;



(d) Snippet #+name square language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (square x) (* x x)) (define (sum-of-squares x y) (+ (square x) (square y))) (sum-of-squares 3 4)

(e) **DONE** Exercise 1.1 **CLOSED:** *[2019-08-20 Tue 14:23]* language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (disp sexp) (display sexp) (newline)) (disp 10) (disp (+ 2 3 4)) (disp (- 9 1)) (disp (/ 6 2)) (disp (+ (* 2 4) (- 4 6))) (define a 3) (define b (+ a 1)) (disp (+ a b (* a b))) (disp (= a b)) (disp (if (and (> b a) (< b (* a b))) b a)) (disp (cond ((= a 4) 6) ((= b 4) (+ 6 7 a)) (else 25))) (disp (+ 2 (if (< b a) b a))) (disp (* (cond ((> a b) a) ((< a b) b) (else -1)) (+ a 1)))

(f) **DONE** Exercise 1.2 **CLOSED:** *[2019-08-20 Tue 14:25]* language=Lisp,label= ,caption= ,captionpos=b,numbers=none (/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5))))) (* 3 (- 6 2) (- 2 7)))

(g) **DONE** Exercise 1.3 **CLOSED:** *[2019-08-20 Tue 14:35]* language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (sum-of-squares x y) (+ (square x) (square y))) (import (srfi 95)) (define (sum-of-two-max a b c) (let ((num$_l ist(sort(listabc)(lambda(ab)(if(> ab)ab)))))(sum-of-squares(carnum$_l$ist)(cadrnum$_l$ist))))(sum-of-two-max123)

(h) **DONE** Exercise 1.4 **CLOSED:** *[2019-08-20 Tue 14:39]* language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (a-plus-abs-b a b) ((if (> b 0) + -) a b)) (disp (a-plus-abs-b 3 4)) (disp (a-plus-abs-b 3 -4))

(i) **DONE** Exercise 1.5 **CLOSED:** *[2019-08-20 Tue 14:50]*

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (p) (p)) (define (test x y) (if (= x 0) 0 y)) (test 0 (p))

On my interpreter this code goes into an infinite recursion, which makes sense, I guess, since the second argument to (test) is evaluated before executing (test). However, if we only substitute p into the application of test and try to traverse the tree depth-first, this code should be able to terminate successfully?

(j) **DONE** Exercise 1.6 **CLOSED:** *[2019-08-21 Wed 14:05]*

The problem with this Alyssa's (new-if) is that both arguments would be computed, so this (new-if) would be either very inefficient or even not working at all in the case when one of the arguments is infeasible. Consider:

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (import (chibi ast)) (import (chibi show)) (define (disp sexp) (display sexp) (newline)) (define (new-if predicate then-clause else-clause) (cond (predicate then-clause) (else else-clause))) (define a 1) (define b 0) (disp (if (not (= b 0)) (/ a b) a)) (new-if (not (= b 0)) (/ a b) a)

However, this issue can be solved using scheme macros.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (import (chibi ast)) (import (chibi show)) (define (disp sexp) (display sexp) (newline)) (define-syntax new-if (syntax-rules () ((new-if predicate then-clause else-clause) (cond (predicate then-clause) (else else-clause))))) (define a 1) (define b 0) (disp (if (not (= b 0)) (/ a b) a)) (disp (new-if (not (= b 0)) (/ a b) a))

The code above works as expected, because the macro does not evaluate its arguments, and (cond) is a special form.

(k) **DONE** Exercise 1.7 **CLOSED:** *[2019-08-22 Thu 12:52]*

This exercise is a very misleading one. On the first glance is seems that this is just about formulating a good criterion. Make no mistake, practically solving this task means really writing all this code carefully.

The function we are interested in is:

$$f(x) = \sqrt{x} \tag{1}$$

The code given in the chapter before is equivalent to the following Newton's method formula, where f_i denotes the next guess:

$$f_{i+1} = \frac{f_i + \frac{x}{f_i}}{2} \tag{2}$$

How on Earth does this formula even appear? Let's remember some mathematics, namely, the Taylor series (variables unbound):

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + o(x) \tag{3}$$

Let us call 'true' value of $\sqrt{x} = f$. Let us call our first guess f_0. What is the value of the difference (error) between them? Clearly, $f - f_0$. Well, the problem is — we don't know f. But we do know f^2. Therefore $f^2 - f_0^2$ is a number we know. What will be the error on the next step of the algorithm? Let's find f_1 as $f_1 = f_0 + \delta$. If δ is not too big, we can use the Taylor expansion from ref:eq:1 δ.

$$E = f^2 - f_0^2 = f^2 - (f_0 + \delta)^2 \approx f^2 - f_0^2 - 2f_0\delta \tag{4}$$

Be careful. What I expanded here is not the function value. It is the error value. Now, clearly we want our error to be as small as possible, desirably as little as machine precision would allow. So assuming $E = 0$, we get an equation to solve:

$$E = 0 \leftrightarrow f^2 - f_0^2 - 2f_0\delta = 0 \tag{5}$$

$$\delta = \frac{f_0^2 - f^2}{2f_0} \tag{6}$$

Remember though that we don't need just δ here. We actually need f_1. But f_1 is just $f_0 + \delta$.

$$f_1 = \frac{f^2 - f_0^2}{2f_0} + f_0 \tag{7}$$

Now if you rearrange this formula, you will get exactly the formula ref:eq:1.

The code below is copied from SICP verbatim and implements the algorithm above.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (sqrt-iter guess x) (if (good-enough? guess x) guess (sqrt-iter (improve guess x) x)))

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (improve guess x) (average guess (/ x guess)))

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (good-enough? guess x) (< (abs (- (square guess) x)) 0.001))

(define (improve guess x) (average guess (/ x guess))) (define (average x y) (/ (+ x y) 2)) (define (sqrt x) (sqrt-iter 1.0 x))

#+name simple-newton language=Lisp,label= ,caption= ,captionpos=b,numbers=none (import (chibi ast)) (import (chibi show)) (define (disp sexp) (display sexp) (newline))

(define (sqrt-iter guess x) (if (good-enough? guess x) guess (sqrt-iter (improve guess x) x))) (define (good-enough? guess x) (< (abs (- (square guess) x)) 0.001)) (define (improve guess x) (average guess (/ x guess))) (define (average x y) (/ (+ x y) 2)) (define (sqrt x) (sqrt-iter 1.0 x))

(sqrt 9)

An example of how this fails on small numbers:

language=Lisp,label= ,caption= ,captionpos=b,numbers=none

(square (sqrt 0.0004))

An example of why this fails on big numbers I didn't manage to craft. Perhaps chibi-scheme has some clever way to deal with rounding? Anyway — here is the code: language=Lisp,label= ,caption= ,captionpos=b,numbers=none

(square (sqrt 9999999999.0))

Why exactly this is not very good algorithms is a good question. The derivative of the square is well-defined near the 0, although the derivative of the square root is not. Therefore, the equation ref:eq:8 become very imprecise. As we see, big number seem to be working fine in my scheme implementation.

Let us write a better sqrt-iter?.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (sqrt-iter guess x) (let ((better-guess (improve guess x))) (if (good-enough? guess (square better-guess)) better-guess (sqrt-iter better-guess x))))

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (import (chibi ast)) (import (chibi show)) (define (disp sexp) (display sexp) (newline))

(define (sqrt-iter guess x) (let ((better-guess (improve guess x))) (if (good-enough? guess (square better-guess)) better-guess (sqrt-iter better-guess x)))) (define (good-enough? guess x) (< (abs (- (square guess) x)) 0.001)) (define (improve guess x) (average guess (/ x guess))) (define (average x y) (/ (+ x y) 2)) (define (sqrt x) (sqrt-iter 1.0 x))

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (import (chibi ast)) (import (chibi show)) (define (disp sexp) (display sexp) (newline))

(define (sqrt-iter guess x) (let ((better-guess (improve guess x))) (if (good-enough? guess (square better-guess)) better-guess (sqrt-iter better-guess x)))) (define (good-enough? guess x) (< (abs (- (square guess) x)) 0.001)) (define (improve guess x) (average guess (/ x guess))) (define (average x y) (/ (+ x y) 2)) (define (sqrt x) (sqrt-iter 1.0 x))

(square (sqrt 0.0004))

Works faster and gives a better result. Seemingly. QED[1].

(l) **DONE** Exercise 1.8 **CLOSED:** *[2019-08-22 Thu 17:36]*

This exercise is not very hard. The only difference is that the 'improve' function is not derived from a derivative of a square but rather from a derivative of a cube.

language=Lisp,label=orge4c07f7,caption= ,captionpos=b,numbers=none (define (cube x) (* x x x))

language=Lisp,label=orgcf8e623,caption= ,captionpos=b,numbers=none (define (cube-improve guess x) (/ (+ (/ x (* guess guess)) (* 2 guess)) 3))

language=Lisp,label=orgf09b74f,caption= ,captionpos=b,numbers=none (define (cube-good-enough? guess x) (< (abs (- (cube guess) x)) 0.001))

language=Lisp,label=org1bd49a1,caption= ,captionpos=b,numbers=none (define (cube-root-iter guess x) (let ((better-guess (cube-improve guess x))) (disp better-guess) (if (cube-good-enough? better-guess (cube guess)) better-guess (cube-root-iter better-guess x))))

language=Lisp,label=orgdc60fc9,caption= ,captionpos=b,numbers=none (import (chibi ast)) (import (chibi show)) (define (disp sexp) (display sexp) (newline)) (define (cube x) (* x x x)) (define (cube-improve guess x) (/ (+ (/ x (* guess guess)) (* 2 guess)) 3)) (define (cube-good-enough? guess x) (< (abs (- (cube guess) x)) 0.001)) (define (cube-root-iter guess x) (let ((better-guess (cube-improve guess x))) (disp better-guess) (if (cube-good-enough?

[1] This exercise took me 7 hours.

better-guess (cube guess)) better-guess (cube-root-iter better-guess x)))) (cube-root-iter 1.0 27.0)

(m) **TODO** Exercise 1.9 UNCLEAR

I didn't find (inc) and (dec) in my scheme, so I define them myself.

I still don't want to overload the "+" and "-" symbols, so I will call them 'plus' and 'minus'.

language=Lisp,label=org784f3c5,caption= ,captionpos=b,numbers=none (import (chibi ast)) (define (inc x) (+ 1 x)) (define (dec x) (- x 1)) (define-syntax plusF (syntax-rules () ((plusF a b) (if (= a 0) b (inc '(plusF (dec ,a) ,b)))))) (macroexpand '(plusF 4 5))

language=Lisp,label=org4accaad,caption= ,captionpos=b,numbers=none (define (plusS a b) (if (= a 0) b (plusS (dec a) (inc b))))

So my first attempt to write this in Scheme failed, so I resort to the other lisp, which seems to be able to do what I want it to do. Note that (unlike Chibi-scheme), Emacs Lisp does have the increment and decrement operations, they are called (1+) and (1-).

So let's analyze Alyssa's form number 1: language=elisp,label=orgb683a77,caption= ,captionpos=b,numbers=none (defmacro plusF (a b) (if (= a 0) b '(1+ (plusF ,(1- a) ,b)))) (macroexpand-all '(plusF 4 5))

language=elisp,label=org153edee,caption= ,captionpos=b,numbers=none (defmacro plusS (a b) (if (= a 0) b '(plusS ,(1- a) ,(1+ b)))) (macroexpand-all '(plusS 4 5))

Okay, this is not in Scheme. But we can clearly see the difference. The first macro is genuinely recursive, it expands to a series of calls, and needs to keep the information about this calls on the stack. The second one is actually iterative. The macro call only happens as the last step, and no information is kept, as the return value will be just the last result, so this macro is expanded until it's just a number.

(n) **DONE** Exercise 1.10 **CLOSED:** *[2019-08-25 Sun 18:31]*

Let's run the demos first: language=Lisp,label=orgbe345e1,caption= ,captionpos=b,numbers=none (import (chibi ast)) (import (chibi show)) (define (disp sexp) (display sexp) (newline)) (define (A x y) (cond ((= y 0.0) 0.0) ((= x 0.0) (* 2.0 y)) ((= y 1.0) 2.0) (else (A (- x 1.0) (A x (- y 1.0)))))) (disp (A 1 10)) (disp (A 2 4)) (disp (A 3 3))

```
1024.0
65536.0
65536.0
```

The values of these expressions are listed above.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (f n) (A 0 n)) (define (g n) (A 1 n)) (define (h n) (A 2 n)) (define (k n) (* 5 n n))

The mathematical expressions for these formulae are:

$$f(n) = 2y \tag{8}$$
$$g(n) = 2^y \tag{9}$$
$$h(n) = 2^{2^n} \tag{10}$$
$$k(n) = 5n^2 \tag{11}$$
$$\tag{12}$$

Actually this is not the Ackermann's function as it is most often defined, for example, see `http://mathworld.wolfram.com/AckermannFunction.html`. But the recurrent relation is the same. This version of the Ackermann's function seems to be equivalent to the powers tower.

I may have lied with the coefficients, but essentially, the Ackermann's function with parameters n and m works by applying the n-the hyperoperator m times to 2. A hyperoperator is a generalization of the standard matematical operator sequence '+', '*', '$\hat{}$', see `https://googology.wikia.org/wiki/Hyper_operator`

(o) **DONE** Exercise 1.11 **CLOSED:** *[2019-08-25 Sun 19:25]*

$$f(n) = \begin{cases} n & : \quad n < 3 \\ f(n-1) + 2f(n-2) + 3f(n-3) & : \quad \geq 3 \end{cases} \tag{13}$$

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (f-recursive n) (cond ((< n 3) n) (else (+ (f-recursive (- n 1)) (* 2 (f-recursive (- n 2))) (* 3 (f-recursive (- n 3))))))) (f-recursive 7)

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (f-iter m n fn-1 fn-2 fn-3) (let ((fn (+ fn-1 (* 2 fn-2) (* 3

fn-3)))) (cond ((= m n) fn) (else (f-iter m (+ n 1) fn fn-1 fn-2)))))
(define (f-iterative n) (cond ((< n 3) n) (else (f-iter n 3 2 1 0))))
(f-iterative 7)

(p) **DONE** Exercise 1.12 **CLOSED:** *[2019-08-25 Sun 19:42]*

```
                1
            1       1
        1       2       1
    1       3       3       1
1       4       6       4       1
            .   .   .
```

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (define (pascal-number line-number column-number) (cond ((= line-number 1) 1) ((= line-number 2) 1) ((= column-number 1) 1) ((= column-number line-number) 1) (else (+ (pascal-number (- line-number 1) (- column-number 1)) (pascal-number (- line-number 1) column-number))))) (pascal-number 5 3)

(q) **DONE** Exercise 1.13 **CLOSED:** *[2019-08-25 Sun 23:04]*

$$\mathrm{Fib}(n) = \begin{cases} 0 & : \quad n = 0 \\ 1 & : \quad n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & : \quad \text{otherwise} \end{cases} \tag{14}$$

Abelson and Sussman define $\varphi = (1+\sqrt{5})/2$ and $\psi = (1-\sqrt{5})/2$. Knowing that $\mathrm{Fib}(n) = (\varphi^n - \psi^n)/\sqrt{5}$ is almost all the problem done, because ψ is clearly less than 1, so for large n it will be exponentially close to 0, and this is where the "closest integer" comes from.

Let us prove the rest by induction.

$$\frac{\varphi^{n-1} - \psi^{n-1} + \varphi^{n-2} - \psi^{n-2}}{\sqrt{5}} = \frac{\varphi^n - \psi^n}{\sqrt{5}} \quad (15)$$

$$\varphi^{n-1} - \psi^{n-1} + \varphi^{n-2} - \psi^{n-2} = \varphi^n - \psi^n \quad (16)$$

$$(\varphi + 1)\varphi^{n-2} - (\psi + 1)\psi^{n-2} = \varphi^n - \psi^n \quad (17)$$

$$(\varphi + 1 - \varphi^2)\varphi^{n-2} = (\psi + 1 - \psi^2)\psi^{n-2} \quad (18)$$

$$(\frac{1+\sqrt{5}}{2} + 1 - (\frac{1+\sqrt{5}}{2})^2)\varphi^{n-2} = (\frac{1-\sqrt{5}}{2} + 1 - (\frac{1-\sqrt{5}}{2}))\psi^{n-2} \quad (19)$$

$$(\frac{2+2\sqrt{5}}{4} + \frac{4}{4} - \frac{1+2\sqrt{5}+5}{4})\varphi^{n-2} = (\frac{2-2\sqrt{5}}{4} + \frac{4}{4} - \frac{1-2\sqrt{5}+5}{4})\psi^{n-2} \quad (20)$$

$$0 = 0 \quad (21)$$

This proves that the recurrent relation for $\frac{\varphi^n - \psi^n}{\sqrt{5}}$ is the same as for the Fibonacci sequence. Then if we prove that there exist such n and $n-1$ so that $\text{Fib}(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}}$, then we're done.

Indeed, let's have a look at $n = 1$: $\frac{1+\sqrt{5}}{2\sqrt{5}} - \frac{1-\sqrt{5}}{2\sqrt{5}} = 1$; and $n = 0$: $\frac{1-1}{\sqrt{5}} = 0$.

(r) **DONE** Exercise 1.14 **CLOSED:** *[2019-08-26 Mon 00:40]*

Theoretically, this exercise should be done in Scheme, but again, I don't understand how to do something like (macroexpand-all form), so I will have to resort to the tool using which I do know how to do it.

language=elisp,label=orgf5203ce,caption= ,captionpos=b,numbers=none
(defmacro cc (amount kinds-of-coins) (cond ((= amount 0) 1) ((or (< amount 0) (= kinds-of-coins 0)) 0) (‘(+ (cc ,amount ,(- kinds-of-coins 1)) (cc ,(- amount (first-denomination kinds-of-coins)) ,kinds-of-coins))))) (defun first-denomination (kinds-of-coins) (cond ((= kinds-of-coins 1) 1) ((= kinds-of-coins 2) 5) ((= kinds-of-coins 3) 10) ((= kinds-of-coins 4) 25) ((= kinds-of-coins 5) 50))) (pp (macroexpand-all ’(cc 11 5)))

The space complexity of the algorithm will be dominated by the depth of the tree — that is the value to be changed, as there is no need to keep any additional information.

The time complexity can be estimated as follows: for every additional value the algorithm will have to go through all passes of the

algorithm without an additional denomination, times the amount divided by the value of an additional denomination. We can consider the additional denomination value as a constant, and the amount of steps for the simplest case of only one denomination is the amount. Therefore, the algorithm is linear in amount and exponential in the number of denominations.

$$C = \Theta(n^a) \tag{22}$$

(s) **TODO** Exercise 1.15

(t) **TODO** Exercise 1.16

(u) **TODO** Exercise 1.17

(v) **TODO** Exercise 1.18

(w) **TODO** Exercise 1.19

(x) **TODO** Exercise 1.20

(y) **TODO** Exercise 1.21

(z) **TODO** Exercise 1.22

() **TODO** Exercise 1.23

() **TODO** Exercise 1.24

() **TODO** Exercise 1.25

() **TODO** Exercise 1.26

() **TODO** Exercise 1.27

() **TODO** Exercise 1.28

() **TODO** Exercise 1.29

() **TODO** Exercise 1.30

() **TODO** Exercise 1.31

() **TODO** Exercise 1.32

() **TODO** Exercise 1.33

() **TODO** Exercise 1.34

() **TODO** Exercise 1.35

() **TODO** Exercise 1.36

() **TODO** Exercise 1.37

() **TODO** Exercise 1.38

() **TODO** Exercise 1.39
() **TODO** Exercise 1.40
() **TODO** Exercise 1.41
() **TODO** Exercise 1.42
() **TODO** Exercise 1.43
() **TODO** Exercise 1.44
() **TODO** Exercise 1.45
() **TODO** Exercise 1.46

2. **TODO** Chapter 2: Building abstractions with Data [0/97]

(a) **TODO** Exercise 2.1
(b) **TODO** Exercise 2.2
(c) **TODO** Exercise 2.3
(d) **TODO** Exercise 2.4
(e) **TODO** Exercise 2.5
(f) **TODO** Exercise 2.6
(g) **TODO** Exercise 2.7
(h) **TODO** Exercise 2.8
(i) **TODO** Exercise 2.9
(j) **TODO** Exercise 2.10
(k) **TODO** Exercise 2.11
(l) **TODO** Exercise 2.12
(m) **TODO** Exercise 2.13
(n) **TODO** Exercise 2.14
(o) **TODO** Exercise 2.15
(p) **TODO** Exercise 2.16
(q) **TODO** Exercise 2.17
(r) **TODO** Exercise 2.18
(s) **TODO** Exercise 2.19
(t) **TODO** Exercise 2.20
(u) **TODO** Exercise 2.21

(v) **TODO** Exercise 2.22

(w) **TODO** Exercise 2.23

(x) **TODO** Exercise 2.24

(y) **TODO** Exercise 2.25

(z) **TODO** Exercise 2.26

() **TODO** Exercise 2.27

() **TODO** Exercise 2.28

() **TODO** Exercise 2.29

() **TODO** Exercise 2.30

() **TODO** Exercise 2.31

() **TODO** Exercise 2.32

() **TODO** Exercise 2.33

() **TODO** Exercise 2.34

() **TODO** Exercise 2.35

() **TODO** Exercise 2.36

() **TODO** Exercise 2.37

() **TODO** Exercise 2.38

() **TODO** Exercise 2.39

() **TODO** Exercise 2.40

() **TODO** Exercise 2.41

() **TODO** Exercise 2.42

() **TODO** Exercise 2.43

() **TODO** Exercise 2.44

() **TODO** Exercise 2.45

() **TODO** Exercise 2.46

() **TODO** Exercise 2.47

() **TODO** Exercise 2.48

() **TODO** Exercise 2.49

() **TODO** Exercise 2.50

() **TODO** Exercise 2.51

() **TODO** Exercise 2.52

() **TODO** Exercise 2.53
() **TODO** Exercise 2.54
() **TODO** Exercise 2.55
() **TODO** Exercise 2.56
() **TODO** Exercise 2.57
() **TODO** Exercise 2.58
() **TODO** Exercise 2.59
() **TODO** Exercise 2.60
() **TODO** Exercise 2.61
() **TODO** Exercise 2.62
() **TODO** Exercise 2.63
() **TODO** Exercise 2.64
() **TODO** Exercise 2.65
() **TODO** Exercise 2.66
() **TODO** Exercise 2.67
() **TODO** Exercise 2.68
() **TODO** Exercise 2.69
() **TODO** Exercise 2.70
() **TODO** Exercise 2.71
() **TODO** Exercise 2.72
() **TODO** Exercise 2.73
() **TODO** Exercise 2.74
() **TODO** Exercise 2.75
() **TODO** Exercise 2.76
() **TODO** Exercise 2.77
() **TODO** Exercise 2.78
() **TODO** Exercise 2.79
() **TODO** Exercise 2.80
() **TODO** Exercise 2.81
() **TODO** Exercise 2.82
() **TODO** Exercise 2.83

() **TODO** Exercise 2.84
() **TODO** Exercise 2.85
() **TODO** Exercise 2.86
() **TODO** Exercise 2.87
() **TODO** Exercise 2.88
() **TODO** Exercise 2.89
() **TODO** Exercise 2.90
() **TODO** Exercise 2.91
() **TODO** Exercise 2.92
() **TODO** Exercise 2.93
() **TODO** Exercise 2.94
() **TODO** Exercise 2.95
() **TODO** Exercise 2.96
() **TODO** Exercise 2.97

3. **TODO** Chapter 3: Modularity, Objects and State [0/82]

(a) **TODO** Exercise 3.1
(b) **TODO** Exercise 3.2
(c) **TODO** Exercise 3.3
(d) **TODO** Exercise 3.4
(e) **TODO** Exercise 3.5
(f) **TODO** Exercise 3.6
(g) **TODO** Exercise 3.7
(h) **TODO** Exercise 3.8
(i) **TODO** Exercise 3.9
(j) **TODO** Exercise 3.10
(k) **TODO** Exercise 3.11
(l) **TODO** Exercise 3.12
(m) **TODO** Exercise 3.13
(n) **TODO** Exercise 3.14
(o) **TODO** Exercise 3.15

(p) **TODO** Exercise 3.16

(q) **TODO** Exercise 3.17

(r) **TODO** Exercise 3.18

(s) **TODO** Exercise 3.19

(t) **TODO** Exercise 3.20

(u) **TODO** Exercise 3.21

(v) **TODO** Exercise 3.22

(w) **TODO** Exercise 3.23

(x) **TODO** Exercise 3.24

(y) **TODO** Exercise 3.25

(z) **TODO** Exercise 3.26

() **TODO** Exercise 3.27

() **TODO** Exercise 3.28

() **TODO** Exercise 3.29

() **TODO** Exercise 3.30

() **TODO** Exercise 3.31

() **TODO** Exercise 3.32

() **TODO** Exercise 3.33

() **TODO** Exercise 3.34

() **TODO** Exercise 3.35

() **TODO** Exercise 3.36

() **TODO** Exercise 3.37

() **TODO** Exercise 3.38

() **TODO** Exercise 3.39

() **TODO** Exercise 3.40

() **TODO** Exercise 3.41

() **TODO** Exercise 3.42

() **TODO** Exercise 3.43

() **TODO** Exercise 3.44

() **TODO** Exercise 3.45

() **TODO** Exercise 3.46

() **TODO** Exercise 3.47
() **TODO** Exercise 3.48
() **TODO** Exercise 3.49
() **TODO** Exercise 3.50
() **TODO** Exercise 3.51
() **TODO** Exercise 3.52
() **TODO** Exercise 3.53
() **TODO** Exercise 3.54
() **TODO** Exercise 3.55
() **TODO** Exercise 3.56
() **TODO** Exercise 3.57
() **TODO** Exercise 3.58
() **TODO** Exercise 3.59
() **TODO** Exercise 3.60
() **TODO** Exercise 3.61
() **TODO** Exercise 3.62
() **TODO** Exercise 3.63
() **TODO** Exercise 3.64
() **TODO** Exercise 3.65
() **TODO** Exercise 3.66
() **TODO** Exercise 3.67
() **TODO** Exercise 3.68
() **TODO** Exercise 3.69
() **TODO** Exercise 3.70
() **TODO** Exercise 3.71
() **TODO** Exercise 3.72
() **TODO** Exercise 3.73
() **TODO** Exercise 3.74
() **TODO** Exercise 3.75
() **TODO** Exercise 3.76
() **TODO** Exercise 3.77

() **TODO** Exercise 3.78
() **TODO** Exercise 3.79
() **TODO** Exercise 3.80
() **TODO** Exercise 3.81
() **TODO** Exercise 3.82

4. **TODO** Chapter 4: Metalinguistic Abstraction [0/79]

(a) **TODO** Exercise 4.1
(b) **TODO** Exercise 4.2
(c) **TODO** Exercise 4.3
(d) **TODO** Exercise 4.4
(e) **TODO** Exercise 4.5
(f) **TODO** Exercise 4.6
(g) **TODO** Exercise 4.7
(h) **TODO** Exercise 4.8
(i) **TODO** Exercise 4.9
(j) **TODO** Exercise 4.10
(k) **TODO** Exercise 4.11
(l) **TODO** Exercise 4.12
(m) **TODO** Exercise 4.13
(n) **TODO** Exercise 4.14
(o) **TODO** Exercise 4.15
(p) **TODO** Exercise 4.16
(q) **TODO** Exercise 4.17
(r) **TODO** Exercise 4.18
(s) **TODO** Exercise 4.19
(t) **TODO** Exercise 4.20
(u) **TODO** Exercise 4.21
(v) **TODO** Exercise 4.22
(w) **TODO** Exercise 4.23
(x) **TODO** Exercise 4.24

(y) **TODO** Exercise 4.25

(z) **TODO** Exercise 4.26

() **TODO** Exercise 4.27

() **TODO** Exercise 4.28

() **TODO** Exercise 4.29

() **TODO** Exercise 4.30

() **TODO** Exercise 4.31

() **TODO** Exercise 4.32

() **TODO** Exercise 4.33

() **TODO** Exercise 4.34

() **TODO** Exercise 4.35

() **TODO** Exercise 4.36

() **TODO** Exercise 4.37

() **TODO** Exercise 4.38

() **TODO** Exercise 4.39

() **TODO** Exercise 4.40

() **TODO** Exercise 4.41

() **TODO** Exercise 4.42

() **TODO** Exercise 4.43

() **TODO** Exercise 4.44

() **TODO** Exercise 4.45

() **TODO** Exercise 4.46

() **TODO** Exercise 4.47

() **TODO** Exercise 4.48

() **TODO** Exercise 4.49

() **TODO** Exercise 4.50

() **TODO** Exercise 4.51

() **TODO** Exercise 4.52

() **TODO** Exercise 4.53

() **TODO** Exercise 4.54

() **TODO** Exercise 4.55

() **TODO** Exercise 4.56
() **TODO** Exercise 4.57
() **TODO** Exercise 4.58
() **TODO** Exercise 4.59
() **TODO** Exercise 4.60
() **TODO** Exercise 4.61
() **TODO** Exercise 4.62
() **TODO** Exercise 4.63
() **TODO** Exercise 4.64
() **TODO** Exercise 4.65
() **TODO** Exercise 4.66
() **TODO** Exercise 4.67
() **TODO** Exercise 4.68
() **TODO** Exercise 4.69
() **TODO** Exercise 4.70
() **TODO** Exercise 4.71
() **TODO** Exercise 4.72
() **TODO** Exercise 4.73
() **TODO** Exercise 4.74
() **TODO** Exercise 4.75
() **TODO** Exercise 4.76
() **TODO** Exercise 4.77
() **TODO** Exercise 4.78
() **TODO** Exercise 4.79

5. **TODO** Chapter 5: Computing with Register Machines [0/52]
   (a) **TODO** Exercise 5.1
   (b) **TODO** Exercise 5.2
   (c) **TODO** Exercise 5.3
   (d) **TODO** Exercise 5.4
   (e) **TODO** Exercise 5.5

(f) **TODO** Exercise 5.6

(g) **TODO** Exercise 5.7

(h) **TODO** Exercise 5.8

(i) **TODO** Exercise 5.9

(j) **TODO** Exercise 5.10

(k) **TODO** Exercise 5.11

(l) **TODO** Exercise 5.12

(m) **TODO** Exercise 5.13

(n) **TODO** Exercise 5.14

(o) **TODO** Exercise 5.15

(p) **TODO** Exercise 5.16

(q) **TODO** Exercise 5.17

(r) **TODO** Exercise 5.18

(s) **TODO** Exercise 5.19

(t) **TODO** Exercise 5.20

(u) **TODO** Exercise 5.21

(v) **TODO** Exercise 5.22

(w) **TODO** Exercise 5.23

(x) **TODO** Exercise 5.24

(y) **TODO** Exercise 5.25

(z) **TODO** Exercise 5.26

() **TODO** Exercise 5.27

() **TODO** Exercise 5.28

() **TODO** Exercise 5.29

() **TODO** Exercise 5.30

() **TODO** Exercise 5.31

() **TODO** Exercise 5.32

() **TODO** Exercise 5.33

() **TODO** Exercise 5.34

() **TODO** Exercise 5.35

() **TODO** Exercise 5.36

() **TODO** Exercise 5.37
() **TODO** Exercise 5.38
() **TODO** Exercise 5.39
() **TODO** Exercise 5.40
() **TODO** Exercise 5.41
() **TODO** Exercise 5.42
() **TODO** Exercise 5.43
() **TODO** Exercise 5.44
() **TODO** Exercise 5.45
() **TODO** Exercise 5.46
() **TODO** Exercise 5.47
() **TODO** Exercise 5.48
() **TODO** Exercise 5.49
() **TODO** Exercise 5.50
() **TODO** Exercise 5.51
() **TODO** Exercise 5.52

Emacs 26.2 (Org mode 9.1.9)