1 Introduction

1. **Computers and computation.** Computers, like the mindbrain, have internal states, which are, in the standard case, patterns of voltages over a huge number of interconnected transistors. A computational process is a certain sequence (or evolution) of such states.[1] Not every sequence of states is a computational process, but every computational process is some such sequence. What distinguishes a computational process from a random sequence of states is that in the former there are "meaningful" relations between some states in the sequence.

2. **Programs.** A computer program is a body of expressions (also called 'code') that manipulates the states – and thereby the behavior – of a computer. Programs are written in programming languages. Our choice of programming language is **Common Lisp**. We will use the expression '`LISP`' to refer to it.[2]

3. **`SBCL`.** During the term, we will write programs, many of them. For our purposes, the computer that we manipulate via our programs is a machine that is capable of translating the code written in `LISP` into instructions that are executable by the physical computers sitting on our desks or laps. The machine is called `SBCL` (an acronym for Steel Bank Common Lisp). It is not a physical machine; it is an abstract machine, which itself is actually a program. Computer science, like mathematics, thrives on abstractions; if you are new to these subjects, it will take some time to get used to this. Work hard and be patient.

4. **Top-level.** Let us start interacting with the computer. First we need to turn it on by typing '`rlwrap sbcl`' in the command-line, without the quotes. If you see a `*`, everything is in order; you are at the **top-level**. Why we call it that will be clarified in a minute.

5. **Structure of the computer.** To make things a little easier for the start, we will assume that our computer is structured into two components: a **workspace** and an **environment**. We will first clarify the function of the workspace; the discussion on the environment will come a little later.

6. **Workspace.** This can be thought of as the place where all the computations are done; it's a sort of dynamic component of the computer. What we called the top-level – the `*` sign – is an access point to the workspace of `SBCL`. One of the simplest things you can do is to put a number on the workspace through the top-level:[3]

```
1 * 17
2 17
3 *
```

   What's going on here is simply this: You put the number 17 on the workspace (line 1). `SBCL` works on this number – actually it automatically works on whatever you put on the workspace. As there is nothing to work on a single number, the result of the work is the number itself. `SBCL` displays this result to you (line 2). Finally it clears the workspace (line 3).

7. **Procedure calls.** If you want more interesting action on the workspace, you need to occupy `SBCL` with a **procedure call** – something to really work on. Here is an extremely simple one:

[1] In parallel processes, the sequence consists of several threads in parallel.

[2] Lisp is a family of languages – or dialects if you like. When you consult the web for this course, keep in mind that the particular dialect we will use is Common Lisp.

[3] You need to hit `RETURN` after typing the number; and `SBCL` leaves an empty line after the first line, which we omit to save space.

```
* (+ 17 3)
20
*
```

8. **Application.** In the technical parlance the expression `(+ 17 3)` is an **application**,[4] where `+` is the **operator** and the numbers 17 and 3 are **operands**. The operator stands for a **procedure** and the operands provide the **arguments** to the procedure. Here the procedure is arithmetic addition and arguments are the integers 17 and 3.

   You put the simple combination `(+ 17 3)` on the workspace, `SBCL` works on it, computes the result, displays it to you, and finally clears the workspace.

9. **Prefix notation.** In `LISP` you always put the operators before their operands. This allows you to increase the number of operands without repeating the operator:

```
* (+ 17 3 42 8)
70
```

10. **Evaluation.** For `SBCL`, to work on something put on the workspace is to **evaluate** it; it computes the **value** of the expression.

10.1. Numbers evaluate to themselves – `SBCL` gives back exactly what it takes;

10.2. Applications (=procedure calls) evaluate to the result of applying the procedure to its arguments.

11. **REPL.** The other name of the top-level is REPL – Read-Evaluate-Print Loop. It should be obvious why.

12. **Complex applications.** Applications can have other applications as their operands:

```
* (+ 17 (* 3 4))
29
```

   There is no theoretical limit to the complexity of an application.

13. **The Great Rule of Evaluation.**

13.1. Evaluate the operator to get the procedure;

13.2. Evaluate the operands one-by-one, from left to right to get the arguments;

13.3. Apply the procedure to the arguments to compute the value of the application.

14. Looking at 13, you may wonder how does `SBCL` evaluate an operator like `+`? And what is the value obtained?

   The operator `+` is a **symbol**. It is the symbol that names the addition operation (or procedure). The procedure itself is stored somewhere in our computer. Evaluating an operator symbol is to look-up and fetch the procedure stored under the name of that symbol. When `SBCL` finds the application `(+ 17 (* 3 4))` on the workspace, it runs the Great Rule of Evaluation on this application. It first evaluates the `+` symbol to get the addition procedure, then evaluates `17` to `17`, then evaluates the application `(* 3 4)`. In order to perform this last step, it has to run the Great Rule again, this time on this smaller application – get the procedure stored under the name of `*`, evaluate `3` to `3`, `4` to `4`, apply the procedure to the arguments to get `12`. Now this twelve becomes the second argument of the procedure that was fetched as the value of `+`. Finally applying the addition procedure to `17` and `12` gives the value `29`, which gets printed to the screen by the top-level.

[4] Also called “combination”.

15. **Environment.** We said above that our computer has two components: workspace and environment. We saw a little about the workspace, now it's time to talk about the environment. The environment is a collection of storage locations called **variables** – you can think of them as pigeonhole mail boxes you find in the hall of apartments or the secretarial offices of academic departments. All sorts of things can be stored in variables – numbers and procedures are the examples we encountered so far. You can also think of the environment as a sort of dictionary that contains all the words known to the computer at that point.

    Besides the objects stored in them, some variables may also have names. For instance, the variable that hosts the addition procedure has the name `+`. When `SBCL` evaluates the symbol `+` it consults the environment and finds the box that is named `+` and brings a copy[5] of the procedure stored in that box to the workspace, so that it can add the arguments.[6]

    Ask `SBCL` about `(sqrt 9)` or `(max 8 21 13)`; you'll get what you expected, because `SBCL` has variables named `sqrt` and `max` in its environment by default – it "knows" the meaning of these symbols.

16. **Manipulating the environment.** You can ask `SBCL` to name a variable in the environment by:

```
(defvar area)
```

    This is the simplest way you can manipulate the current environment. You tell `SBCL` that you intend to use the symbol `area` as the name of a variable, without yet telling `SBCL` what to store in the variable. If you want to do that as well, then you need to do:

```
(defvar radius 13)
```

    Now you both named a variable and stored a number in it. You can recall that number by simply asking `SBCL` to evaluate the symbol `radius`, by entering it to the top-level.

17. **Special forms.** There are two important things about `DEFVAR` – actually also about all similar creatures: One, a `DEFVAR` form is not an application. If it were, you would get an error as `SBCL` would try to evaluate `RADIUS`, which, at that time, has no value to return. `DEFVAR` is an instance of what is called a **special form**. What makes special forms special is the fact that they do not follow the Great Rule of Evaluation. Every special form has its own ways; and therefore you have to learn them case-by-case.

    The second important thing about `DEFVAR` is the fact that it always *evaluates* its second operand and stores the *value* obtained in a variable that is named by the first operand. For instance, the following move has an identical effect as the above in terms of environment manipulation:

```
(defvar radius (sqrt 169))
```

    since `DEFVAR` evaluates the application `(SQRT 169)` to `13` and stores this value, rather than the application itself, in the variable named `RADIUS`.

18. **`SETF`** `DEFVAR` is a one-shot tool. You define a variable with it, either storing an initial value in it or not. If you want to manipulate the value stored in the variable afterwards, you need `SETF`. For instance,

[5] This, like almost all the things we deal with at the moment, is a metaphor, which will be very useful when we study recursion.

[6] Of course, also the truth about the environment is more complicated than we assume at this stage. We will gradually learn more about it.

```
(setf area (* pi (* radius radius)))
```

does this for the variable named by the symbol AREA. SETF also is a special form. It first evaluates its second operand and puts the value obtained into the variable named by the symbol it received as its first operand.

19. **Side-effects versus Values.** Now we come to perhaps the most vital distinction in programming, that between **side-effects** and (return) **values**. It is a fairly simple distinction, but if you do not understand this distinction absolutely clearly and internalize it, you will find programming very hard and confusing. This is especially so for programming languages like LISP.

    Apparently DEFVAR and SETF are quite similar special forms in terms of what they are doing. You might have, however, noticed that they diverge in what they return as values: DEFVAR returns a symbol – namely the one you used to name the variable, while SETF returns the value it obtained after evaluating its second operand. How do we know? We know because when we enter them to the top-level, that's what we get in return; remember that the top-level – or REPL – reads, evaluate and prints. These behaviors are hard-wired into these special forms. That's what I meant when I said you have to learn them case-by-case.

    Now let us further sharpen our terminology. We have seen two types of expressions so far: 1. applications like `(+ 3 (/ 8 2))` and 2. special forms like `(DEFVAR X 8)`. Looking at expressions, it is impossible to tell whether they are applications or special forms – what about `(CONS C (* 8 7))` for instance – you simply need to know their meaning. Let us abstract above the level of meaning and call applications and special forms collectively as **forms**.[7] There is a crucial behavior that unites forms in LISP. Let us put this in bold face:

    Every form in LISP has (=evaluates to) a value.

    Besides the values they return, DEFVAR and SETF manipulate the environment. These are simple examples of **side-effects.** Applications usually only return values; whereas special forms may have side-effects in additon to returning a value. Whenever you encounter a new form in LISP, first thing is to get clear about whether it's a special form or an application, so that you can learn what will be its return value and side-effects, if there is any.

20. Let us see what we had so far:

```
(defvar radius 13)
(defvar area)
(setf area (* pi (* radius radius)))
```

    The symbols DEFVAR, SETF, PI and * are all known by SBCL by default; they are always present in the environment. The symbols RADIUS and AREA are added to the environment by DEFVAR. In the third line, when the SETF form is put on the workspace, SBCL evaluates `(* pi (* 13 13))` and stores the resulting value in the variable named AREA. This storing action is the side-effect of evaluating the SETF form. The value returned is some number close to 530.9291, which will be printed at the top-level.

    At this point, it is also important to notice the difference between the occurrence of the symbol radius. The first occurence is at an operand position of a special form and does not get evaluated. The second and third occurrences are at operand positions of an application, and therefore they get evaluated – in this case to the value stored in the variable they name.

[7] Another common name is "S-expression".

21. **Abstraction with DEFUN.** The symbols like radius and area above are names of **variables**. You can think of variables as stores that you can put things inside and recall these things by using the names of these stores. We could have computed the number we computed above without the help of the variable we named as radius:

```
(defvar area)
(setf area (* pi (* 13 13)))
```

The only advantage the previous version has over this one is that we would be able to reach the value of the radius if we wanted to – it was stored in the variable named radius; with the second version, the value 13 is not stored anywhere.

The real importance of using named variables, however, comes from the fact that, besides variables, you can also give names to **procedures**. (You can think of variables as nouns and procedures as verbs of the programming language). For instance, the way take the square of a number appears too clumsy, let us define a more concise way of doing it:

```
(defun square (x) (* x x))
```

What we do here basically is to "teach" the computer a new word by giving its meaning as a recipe for a certain action. The action is simply taking something and multiplying it with itself. (Notice that we do not need defvar here.)

Using names to refer to variables and procedures are fundamental abstractions in programming.

21.1. The general structure of a DEFUN form is:[8]

(defun <procedure-name> (<parameters>) <body>)

21.2. The procedure name and the names of parameters are totally upto whoever defines the procedure. Therefore the following definitions are as good as the one given above

```
(defun sqr (x) (* x x))
(defun square (y) (* y y))
```

21.3. DEFUN is a special form.

21.3.1. It's side-effect is manipulating the environment: It causes the name <procedure-name> to get associated with a variable that holds the definition of the procedure. You can think of this definition as a piece of program that is executable by the computer.

21.3.2. It's return value is the procedure name.

21.4. The body of a procedure is a sequence of zero or more forms. In SQUARE there is only one form.

21.5. A DEFUN'ed procedure is called as an application; it is no different than other procedures recognized by SBCL by default. For instance:

```
(square (- 4 1))
```

is an ordinary application, which is evaluated in accord with the Great Rule of Evaluation. First the symbol SQUARE is evaluated to give the program that takes squares, than the only operand is evaluated to give the argument 3; then the procedure is applied to the argument to give the result 9.

[8] Throughout we use <. . .> to mark meta variables; the variables we use in talking about components of LISP code.

22. **If/when you get confused about DEFUN.** There is an alternative way of thinking about DEFUN'ed procedures. Assume we defined a squaring procedure named SQUARE as above. Now, SBCL knows about the definition. When we put `(square (- 4 1))` on the top-level and therefore on the workspace, you can think what SBCL does as follows: It checks whether a procedure named SQUARE is in the environment. Then if it finds it, fetches this definition to the workspace, replacing the parameters in the definition with the operands *without evaluating them*. According to this model, SBCL translates `(square (- 4 1))` to,

```
(* (- 4 1) (- 4 1))
```

Now we have a form on the workspace that consists of applications headed by procedures that are known to SBCL by default – which we will call built-in procedures. At this point the Great Rule of Evaluation runs in the normal fashion and computes the value 9.

22.1. It needs to be emphasized that there is no distinction between DEFUN'ed procedures and the built-in procedures like *, MAX, etc. All applications actually work according to the Great Rule. The substitution model I gave above aims to make how procedure definitions work clearer. In the beginning you may consult to this method to understand how things work; later on you will drop it entirely.

23. **Further abstraction.** Having defined a procedure for squaring numbers, we can change the SETF form of area computation to:

```
(setf area (* pi (square radius)))
```

But why stop here? Let us define a procedure that computes the area of a circular region on the basis of its radius, assuming that we already DEFUN'ed a procedure named SQUARE:

```
(defun area (radius) (* pi (square radius)))
```

Let's trace the computation starting with the procedure call via the application `(area 13)`.

```
(area 13)
(* pi (square 13))
(* pi (* 13 13))
(* pi 169)
530.929
```

24. **Example: Sum of cubes.** Here is a procedure definition that computes the sum of the cubes of two numbers:

```
(defun sum-of-cubes (x y) (+ (cube x) (cube y)))
```

When you try to enter this definition to the top-level, SBCL will warn you that it does not know a function named CUBE. In order for SUM-OF-CUBES to work, you need to define CUBE as well. Here is how:

```
(defun cube (x) (* x x x))
```

25. **Programs.** So far we have been entering all our forms at the top-level. There are some drawbacks of this. One is that all the definitions you make get lost when you quit the top-level (which you can do either by entering `(exit)` or pressing the key sequence Ctrl-D. Another is that you might want to see what you have defined so far; this might be hard to do when entering definitions one by one, expecially

when you have many of them. There is another drawback which is hard to realize at the moment, as we are entering very simple forms. As our forms get complicated we will need certain typographical aids to make our codes easier to read. In order to deal with these drawbacks, we collect our definitions in a file. When we load the file at the top-level, the effect will be entering each definition at the top-level with a single stroke. For instance our circular area procedure can be kept in a file, together with the squaring procedure, which it uses:

```
(defun square (x)
  (* x x))

(defun area (radius)
  (* pi (square radius)))
```

We will call such collections of code **programs**.

Exercise 1.1
For this course you need to learn to use a terminal (=command-line). You can launch a terminal by clicking the terminal icon – the thing that looks like a screen. First, complete the tutorials One and Two here. Then, create a folder named `lisp`, change to it and invoke your terminal application to edit a program file by typing `subl test.lisp`. Type the following code into your file and save it. Then open another terminal, change to the same directory and start `SBCL` by `rlwrap sbcl`. Now you can load your program file by typing `(load "test.lisp")`. Finally, change the name of the file of your program and reload it.

```
; This is my first program

(defvar radius 8)
(defvar area)
(setf area (* pi (* radius radius)))
```

□

Exercise 1.2
Write the following operation in `LISP` – you can assume `(POWER A B)` is an application that raises `A` to the power `B`.

$$\frac{x^n}{7-y/2} \times \frac{y^{2/3}+17}{4}$$

□

Exercise 1.3
What would be the values stored in the variables named by `X`, `Y` and `Z` after entering the following forms to the top-level – first answer, then check with the top-level.

```
(defvar x 8)
(defvar y)
(defvar z)

(setf y (setf z (* x 2)))
```

□

Exercise 1.4
Define a procedure that takes two numbers and returns their average.

□

Exercise 1.5
Define a procedure that computes the absolute value of the difference between two numbers. Define two versions: one using `-`, `MIN` and `MAX`; and another using `-` and `ABS`, where the latter is the `LISP` implementation of the absolute value function mathematically defined as:

$$Abs(x) = \begin{cases} x, & \text{if } x \geq 0; \\ -x, & \text{otherwise.} \end{cases}$$

□

2 Making decisions

1. **Predicates.** Procedures that take an argument and return `T` or `NIL`, that is "true" or "false", or "yes" or "no" are predicates.

1.1. All sorts of numeric comparisons are predicates. Try and observe the following at the top-level:

```
(defvar k 0)
(defvar s 8)
(= k s)
(< k s)
(<= s 8)
(zerop k)
(numberp s)
```

2. The most basic special form to make a decision is `IF`, which has the form:

```
(if <test> <success> <failure>)
```

2.1. **test** – any `LISP` form can be a test; we will say that a test **succeeds**, if the evaluation of the test returns a value other than `NIL` – e.g. the symbol `T` or a number, and that it **fails**, otherwise. Note that not only predicates but any `LISP` form can stand as a test.

2.2. **success** – an expression that will be *evaluated and returned* in case the test succeeds;

2.3. **failure** – an expression that will be *evaluated and returned* in case the test fails.

3. Let us observe `IF` in action. Here is a procedure definition which takes an integer and halves it if it's even, and multiplies it by 3 and adds 1, if it's odd. One mathematical fact useful here is that if an integer is not even, it must be odd.

```
(defun change (n)
  (if (evenp n)
    (/ n 2)
    (+ (* 3 n) 1)))
```

3.1. Our procedure is meant to operate on integers, as the concepts evenness and oddness apply to integers. If you try your procedure with a floating point number ("float" for short) like `3.8`, you will get an error. Try it, see the error, and return to the top-level by `Ctrl-D` or `0`.

3.2. Let us improve our procedure a little. Our plan is to first check whether the input is an integer, if not round it to the nearest integer, and then do what we want.

```
(defun change (n)
  (if (integerp n)
    (if (evenp n)
      (/ n 2)
      (+ (* 3 n) 1))
    (if (evenp (round n))
      (/ (round n) 2)
      (+ (* 3 (round n)) 1))))
```

Now our procedure can handle floats by rounding them to the nearest integers – `8.3` give `4`, `8.7` gives `28` as result.

3.3. It is impossible to miss the duplication of effort here. Almost exactly the same `IF` form is repeated twice; once with `N` and once with `(ROUND N)`. Such repetitions are clear calls for abstraction. Let us separate the part of our procedure that checks integerhood from the rest; that way we can keep our original `CHANGE`, but call it with the argument given, if it's an integer, or the rounded version of it, if it's a floating point number. Here is how to do it:

```
(defun change (n)
  (if (evenp n)
    (/ n 2)
    (+ (* 3 n) 1)))

(defun changer (n)
  (if (integerp n)
    (change n)
    (change (round n))))
```

Now, things look tidier. Notice that we defined `CHANGE` before `CHANGER`. This is not absolutely necessary. Why we did it this way is to avoid a **warning** from `SBCL`, which processes procedure definitions in order, and it would get puzzled if the order was reversed, as, when processing the dinition of `CHANGER` it would bump into a procedure name, `CHANGE`, which it doesn't know at that moment. Such warnings are safe to ignore. But if you don't want to see them, define your procedures in the "correct" order. You can also silence the warnings of `SBCL` if you want to, but this is not advised, as some warnings are useful in correcting some errors in your program.

3.4. We still have some problem. We have been assuming that if something is not an integer, it must be a float. Of course this is not true; try to call the procedure with something that is not an integer *and* not a float, say `(CHANGER T)`, using the built-in symbol `T` that stands for truth.

3.5. Here is an improved version that checks for numberhood as well:

```
(defun change (n)
  (if (evenp n)
    (/ n 2)
    (+ (* 3 n) 1)))

(defun changer (n)
  (if (numberp n)
    (if (integerp n)
      (change n)
      (change (round n)))
    nil))
```

Now we have a definition for `CHANGER` that works for any sort of input whatsoever; when provided a non-number input, it simply returns `NIL`. Note that the failure expression of the inner `IF` is `NIL`, this is perfectly OK, since `NIL` and `T`, like numbers, evaluate to themselves.

3.6. Also note that the third operand of `IF` is optional; if you do not provide it, `NIL` is returned if the test fails; therefore the above could equivalently be written without the final `NIL`.

4. **Negation.** In constructing your tests, there will be times you would like to have the opposite of the outcome of the test. For instance assume you want to see whether something is *not* a number. In such a case the test to use is simply `(not (numberp n))`. This will return `T` if `n` is not a number, and `NIL`, if it is. In other words, `NOT` forms are single operand applications.

5. **Multi-way decisions.** `IF` is perfect for decisions involving a limited number of options. When things get more complicated, heavily nested `IF` forms become impossible to read. The special form `COND` is used in such cases. Its syntax may appear a little complicated in comparison to `IF`, but one gets used to it after a few times.

   Assume a variant of the changer task: We halve an even number, we do what we did with odd numbers this time with numbers divisible by 3, and we leave the number as it is otherwise. This problem is a little more complex than the original. The reason is that, in the original, once we see that a number is not even, then, as we are sure that it is odd, we were able to apply the "three times *n* plus one" procedure; but now, non-evenness does not guarantee the applicability of "three times *n* plus one". If you want to stick with `IF`, you will need to nest another `IF` at the failure slot of the evenness test. The task can be handled without nesting `IF`s as follows:

```
(defun changer-cond (n)
  (cond ((not (numberp n)) nil)
        ((evenp n) (/ n 2))
        ((zerop (rem n 3)) (+ (* 3 n) 1))
        (t n)))
```

6. **Procedures can call themselves.** Here is a further improved version of `CHANGER-COND`, which can handle floats as well as integers. If it discovers an input that is a number but not an integer, it rounds it and calls another instance of the same procedure to handle the rounded float (= integer).

```
(defun changer-cond (n)
  (cond ((not (numberp n)) nil)
        ((not (integerp n)) (changer-cond (round n)))
        ((zerop (rem n 3)) (+ (* 3 n) 1))
        (t n)))
```

7. A `COND` clause that only has a test, returns the value of the test, if it succeeds. Therefore, you can have the final clause of the above program simply as `(n)`. It is, however, advisable to avoid such shortcuts, they may impair the readability of your program.

8. Everything that can be done by `COND` can also be done by `IF` and `PROGN` (which you haven't seen yet), but `COND` makes life easier for complex decisions.

9. `AND` and `OR` form sequences of expressions with special evaluation algorithms:

9.1. `AND`: Evaluate the expressions from left to right until you reach either the end or an expression that evaluates to `NIL`; return the value of the last expression.

9.2. `OR`: Evaluate the expressions from left to right until you reach either the end or an expression that evaluates to something other than `NIL`; return the value of the last expression.

Exercise 2.1
Define your own absolute value procedure (see Ex. 1.5 above).

□

Exercise 2.2
Define a procedure that takes three numbers and gives back the second largest of them.

□

Exercise 2.3
Define a procedure that takes three numbers and gives back the sum of the squares of the larger two.

□

Exercise 2.4
Solve Ex 2.2 using only `IF` and comparison predicates.

□

Exercise 2.5
Rewrite `(AND X Y Z W)` by using cond `COND`.[9]

□

Exercise 2.6
Write `COND` statements equivalent to: `(NOT U)` and `(OR X Y Z)`.[10]

□

Exercise 2.7
The following definition is meant to mimic the behavior of `IF` using `AND` and `OR`.

```
1 (defun custom-if (test succ fail) ; wrong!
2   (or (and test succ) fail))
```

But it is unsatisfactory in one case, what is it? Define a better procedure which avoids this failure.[11]

□

---

[9] Touretzky 1990:ex. 4.19.
[10] Winston and Horn 1984:Problem 4-6.
[11] Touretzky 1990.

3 Repetition

1. Computation frequently involves repeating a task with different inputs and conditions. As the number of repetitions depend on the input, one needs a mechanism that repeats the task until certain conditions are fulfilled. When designing a repetition you need to decide on roughly three things: 1. when to stop; 2. what to do/return when stopped; 2. how to continue.

   Let us take the task of computing the remainder of a dividing an integer m by n. At first glance, you may not look at the task as a repetition task; this must be something that can be solved in a single step, you might think. There is, however, a very nice way to solve the problem that involves repetition. First take the following mathematical definition,

$$Rem(m,n) = \begin{cases} 0, & \text{if } m = n, \\ m, & \text{if } m < n, \\ Rem((m-n),n), & \text{otherwise} \end{cases} \tag{1}$$

   You may find this a little puzzling at first sight, so it is important to spend some time on this to get used to it. The first two cases come from the definition of what is it to divide an integer by another. To convince yourself about the validity of the third step, just observe that, *if the first two conditions do not hold*, you cannot change the result of $Rem(m,n)$ by turning into $Rem((m-n),n)$; whatever the result of the first is, the result of the second will be the same number.

   Now we can directly express the Definition 1 in LISP:

```
(defun remain (m n)
  (cond ((= m n) 0)
        ((< m n) m)
        (t (remain (- m n) n))))
```

2. Now a similar example. Assume we are to add two numbers together and we cannot use +; all we are allowed to use is incrementing and decrementing an integer by 1. Again, we first construct a definition:

$$Add(m,n) = \begin{cases} n, & \text{if } m = 0, \\ Add((m-1),(n+1)) & \text{otherwise} \end{cases} \tag{2}$$

   Yes, this is even stranger than the definition for remainder. But, again, looking at the second clause, it simply tells the truth – the result of adding 3 to 4 cannot be anything different than the result of adding 2 to 5. It is straightforward to implement the definition in LISP,

```
(defun add (m n)
  (cond  ((zerop m) n)
         (t (add (- m 1) (+ n 1)))))
```

3. Here is another perspective on the same problem, with a slightly different definition:

$$Add(m,n) = \begin{cases} n, & \text{if } m = 0, \\ 1 + Add((m-1),n) & \text{otherwise} \end{cases} \tag{3}$$

   This translates into LISP as,

```
(defun add2 (m n)
  (cond ((zerop m) n)
        (t (+ 1 (add2 (- m 1) n)))))
```

Exercise 3.1
Define a procedure that multiplies two integers using only addition as a primitive arithmetic operation.

□

Exercise 3.2
The factorial of a non-negative integer is defined as follows:

$$F(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times F(n-1), & \textit{otherwise} \end{cases} \tag{4}$$

Define a procedure that computes the factorial of a given integer.

□

Exercise 3.3
Collatz' Conjecture says that starting with any positive integer, by running the function `C` defined below, you will eventually reach number 1.

$$C(n) = \begin{cases} 1, & \text{if } n = 1 \\ C(n/2), & \text{if } n \text{ is even} \\ C(3n+1), & \text{if } n \text{ is odd} \end{cases} \tag{5}$$

Define a procedure `COLL` that implements the function C. Before running your procedure, type `(trace coll)` at the top-level and observe how your initial input approaches to 1.

□

Exercise 3.4
Define a procedure that takes two integers, say x and y, and returns the sum of all the integers in the range including and between x and y.

□

Exercise 3.5
Define a factorial procedure that uses an accumulator.

□

Exercise 3.6
The Fibonacci numbers are defined for non-negative integers as follows:

$$Fib(n) = \begin{cases} n, & \text{if } n < 2 \\ Fib(n-1) + Fib(n-2), & \text{otherwise} \end{cases} \tag{6}$$

Define a procedure that gives the Fibonacci number of given integer. Define,

- a version without using accumulator(s);
- a version with accumulator(s); here you can have two versions, one where you use a separate counter, and another where you use n itself as the counter.

□

Exercise 3.7
Here is a method, known as Newton's Method, for finding square roots:
In order to find the square root of x,

1. Start with a guess $y = 1$.
2. Check if y^2 is close enough to x; say the difference is not larger than 0.00001.
3. If yes, stop and report y as the result; if no, update your guess by replacing y with $\frac{\frac{x}{y}+y}{2}$ and return to step 2.

Write a program that computes the square root of a given number by this method. Divide your program into procedure definitions. You need to use `1.0` as the initial guess, otherwise `LISP` will generate fractions rather than floating point numbers.

---

4 Conses and Lists

4.1 Conses

1. Three building blocks: **symbols**, **numbers**, **addresses** and memory locations called **cells**.

1.1. Numbers include, but are not limited to, integers $\{\ldots -3, -2, -1, 0, 1, 2, 3 \ldots\}$.

1.2. Symbols include, but are not limited to, combinations of numbers, alphabetical characters and the '-', which are not numbers.

1.3. An address is a binary number (typically 32bit) that points to a cell, where symbols and numbers are stored.

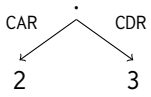
2. A fourth type of entity is a **cons**, a concatenation of two addresses, which is stored in a cell.

2.1. Therefore, an address, by virtue of uniquely identifying a cell, can point to a symbol, a number, or a cons.

2.2. Or, equivalently, cells host symbols, numbers and conses.

2.3. The two parts forming a cons are called `CAR` and `CDR`, respectively.

2.4. A cons is used to pair two objects residing in the cells pointed to by the `CAR` and the `CDR` of the cons:[12]



3. You can construct a cons of two objects by applying the procedure `CONS`; for instance,

```
(cons 2 3)
```

will return a cons of 2 and 3. Witness that

```
(defvar k  (cons 2 3))
(consp k)
```

will return `T`.

You can reach the components of a cons by the built-ins `CAR` and `CDR`:[13]

```
(car k) ==> 2
(cdr k) ==> 3
```

4.2 Lists

1. List is the central data structure in LISP, which stands for Lis(t) P(rocessor).

1.1. Lists are represented as sequences of objects separated by white space and enclosed in parentheses:

[12] Think of it as an ordered pair in set theory.
[13] Note that instead of giving full top-level interaction, we designate the return value by ==>, which is not part of `LISP`.

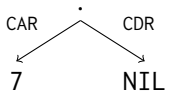
```
(1 3 9)
(RED GREEN BLUE)
(1 GREEN)
()
(7)
```

2. We call an object in a list its element.

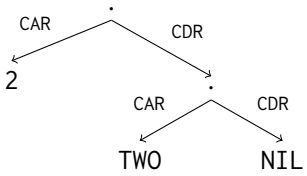
3. The empty list is represented in two equivalent ways: `NIL` and `()` – both evaluate to themselves.

4. Lists are represented internally as conses. The `CAR` of the cons points to the first element of the list, while the `CDR` of the cons points to the list but its first element.

4.1. The single element list `(7)` is represented as:



4.2. The two element list `(2 TWO)` is represented as:

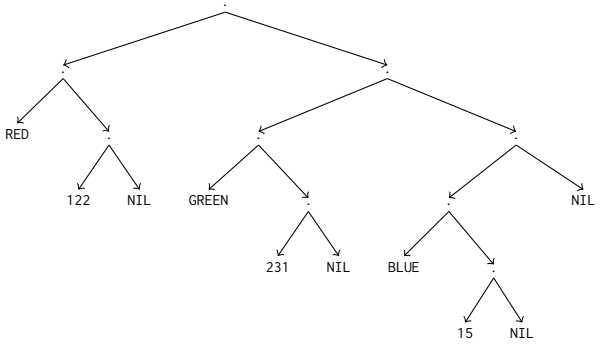


and so on...

4.3. The only exception to cons-based representation of lists is the empty list, which is the object `NIL`, which is not a cons.

4.4. Lists can be nested (= lists as elements of lists).

4.4.1. Take for example `((RED 122) (GREEN 231) (BLUE 15))`:[14]



5. **QUOTE:** Notice that trying to get the first element of the list `(1 2 3)` by `(car (1 2 3))` results in error. The reason is that `SBCL` attempts to interpret the form `(1 2 3)` as an application, seeking a procedure definition under the name `1`. This is so because `CAR` is an application, its argument needs to get evaluated. But in this case we mean a list to be just a list and would not want `LISP` to evaluate that expression as a procedure call. The way to tell `LISP` not to evaluate an expressions and leave it as it is is putting the expression in quotes, as we do in natural language. The following would behave well:

[14] From here on we adopt the convention that the left branch leaving a cons is `CAR` and the right branch is `CDR`, to be able to omit the labels on the edges.

```
(car (quote (1 2 3)))
(cdr (quote (a b c)))
```

5.1. As one needs so frequently to quote expressions, there is a shorthand for QUOTE:

```
(car '(1 2 3))
(cdr '(a b c))
```

(list (quote a) (quote b) (quote c))))

6. **Quoting symbols:** Preventing evaluation by quoting applies to symbols as well. This enables one to use symbols as data; a quoted symbol evaluates to the symbol itself.

   Note that `'(a b c)` is not the same as `('a 'b 'c)`, make sure that you understand why.

7. **Constructing lists:** The most basic constructor for lists is CONS, it adds an object (CAR) to the front of a list (CDR). Another constructor is LIST, it simply evaluates and collects its operands into a list. Yet another is APPEND, which joins together lists. CONS is always two argument, while LIST and APPEND take as many arguments as they like, including zero. Try and observe the difference between the following:

```
(cons '(1 2) '(3))
(list '(1 2) '(3))
(append '(1 2) '(3))
```

8. **Dotted lists:** In a normal list, as you break down the list into its conses, you will always find a "bottom" cons where an object is consed with the empty list NIL. In a dotted list this "bottom" cons does not involve NIL. For instance you can cons two non-NIL objects and form a dotted list:

```
(cons 'a 'b) ==> (A . B)
```

   You can go on consing to this dotted list:

```
(cons 'x (cons 'a 'b)) ==> (X A . B)
```

   SBCL will visually indicate a dotted list by a dot before the CDR. Normal lists and dotted lists differ in their final CDRs.

9. **Accessing list elements:** We already know how to access the first element, namely by CAR. What about other elements? One way is to combine CARs and CDRs. For instance having,

```
(defvar k '(1 (2 3) 4))
```

   to get the list `(2 3)` you can make `(car (cdr k))`; or to get 2 `(car (car (cdr k)))`; or to get 3 `(car (cdr (car (cdr k))))`.

   LISP provides the built-ins FIRST to TENTH. Some people like to use FIRST for CAR and REST for CDR.

   You can combine the successive CARs and CDRs as in CDDR or CADAR up to a certain level. The convention is to match the nested operators from right to left and inside to outside. For instance `(cadr x)` is equivalent to `(car (cdr x))`, which is equivalent to `(second x)`.

   Finally, the built-in NTH takes a non-negative integer and a list, and gives the element at that index position, where indexing starts with 0 – an index out of the range of the list returns NIL.

4.3 Procedures on lists

1. **Example (membership):** LISP has a built-in membership procedure called MEMBER for lists. Let's define our own:

```
(defun memberp (item lst)
  (cond ((endp lst) nil) ; test if lst is empty or not
        ((equal item (car lst)) t)
        (t (memberp item (cdr lst)))))
```

2. **List predicates:**

2.1. **NULL vs. ENDP:** Note that the test for emptiness of a list is ENDP. There is another predicate NULL which behaves quite like ENDP, giving T for NIL. The two predicates diverge in their behavior for nonNIL inputs. NULL gives NIL for anything that is not NIL; but ENDP gives an error if its argument is not a list.

2.2. **LISTP vs. CONSP:** Any cons will answer T to CONSP. All lists are conses with the only exception of NIL. Therefore, NIL is the only thing that will answer NIL to CONSP and T to LISTP. Yes, a little complicated; make sure you get it right.

There is also ATOM, which is a predicate[15] that answers T to expressions that answer NIL to CONSP.

3. **Example (length):** Now our own version of the built-in LENGTH, first without an accumulator:

```
(defun len (lst)
  (if (endp lst)
    0
    (+ 1 (len (cdr lst)))))
```

and now with an accumulator:

```
(defun len-acc (lst &optional (counter 0))
  (if (endp lst)
    counter
    (len-acc (cdr lst) (+ counter 1))))
```

Exercise 4.1
Define a procedure that computes the sum of a list of numbers; with and without an accumulator. □

Exercise 4.2
Define a procedure AFTER-FIRST that takes two lists and inserts all the elements in the second list after the first element of the first list. Given (A D E) and (B C), it should return (A B C D E). □

Exercise 4.3
Define your own version of NTH (don't use NTHCDR). □

Exercise 4.4
Define a procedure MULTI-MEMBER that checks if its first argument occurs more than once in the second. □

Exercise 4.5
Define your own procedure APPEND2 that appends two list arguments into a third list. You are not allowed to use APPEND, LIST and REVERSE – use just CONS.

□

[15] Note that NULL and ATOM do not follow the general convention of ending predicate names with 'p'.

Exercise 4.6
Define a procedure HOW-MANY? that counts the top-level occurrences of an item in a list.

```
(how-many? 'a '(a b r a c a d a b r a))
5
```

□

Exercise 4.7
The built-in REVERSE reverses a list. Define your own version of reverse. □

Exercise 4.8
Define a predicate that tells whether its argument is a dotted list or not. □

Exercise 4.9
Define a recursive procedure D-HOW-MANY? that counts all not only top-level occurrences of an item in a list. For instance (D-HOW-MANY? 'A '((A B) (C (A X)) A)) should return 3.

□

Exercise 4.10
In Ex 4.7 you defined a list reversing procedure. Now alter that definition so that it not only reverses the order of the top-level elements in the list but also reverses any members which are themselves lists.

```
* (reverse0 '(a (b c (x d)) k))
(K ((D X) C B) A)
```

□

Exercise 4.11
Define a three argument procedure REMOVE-NTH, which removes every *n*th occurrence of an item from a list.

□

Exercise 4.12
Define a procedure that takes a list of integers and returns the *second* largest integer in the list.

□

Exercise 4.13
Define a procedure that takes a list of integers and an integer *n*, and returns the *n*th largest integer in the list.

□

Exercise 4.14
Define a procedure UNIQ that takes a list and removes all the repeated elements in the list keeping only the first occurrence. For instance:

```
* (uniq '(a b r a c a d a b r a))
(A B R C D)
```

Don't use REMOVE (built-in or in-house), you may use MEMBER.

□

Exercise 4.15
Solve Ex 4.14 by keeping the last occurrence rather than the first. □

Exercise 4.16
Define a procedure REMLAST which removes the last occurrence of an item from a list. Do not use MEMBER or REVERSE. □

Exercise 4.17
Define a procedure `REMOVEX` that takes an element and a list; and returns a list where all the occurrences of the element that are preceded by the symbol `X` are removed from the list. □

Exercise 4.18
Define a recursive function `FLATTEN`, which takes a possibly nested list and returns a version where all nesting is eliminated. E.g. `((1 (2) 3) 4 (((5) 6) 7))` should be returned as `(1 2 3 4 5 6 7)`. □

Exercise 4.19
Define a recursive procedure `SUBSTITUTE`, with 3 arguments, say `old new exp` such that every occurrence of `old` at the top-level of `exp` is replaced by `new`. By "top-level" we mean the function should not check embedded levels in lists. E.g. `(substitute 'x 'k '(x (x y) z))` should return `(k (x y) z)`. □

Exercise 4.20
Modify `SUBSTITUTE` to `D-SUBS` (for "deep substitute"), so that it does the replacement for *all* occurrences of `old`, no matter how deeply embedded. □

Exercise 4.21
Define a recursive procedure that counts the non-nil atoms in a list. For instance, an input like `((a b) c)` should return 3, `(a ((b (c) d)))` should return 4, and so on. Remember that the built-in `ATOM` returns `NIL` for all lists except `NIL`; `NULL` returns `T` only for `NIL`; `ENDP` is like `NULL`, except that it gives an error if its input happens to be something other than a list. Your function should use a counter/accumulator – it will be a two argument function.

□

Exercise 4.22
Define a procedure `BRING-TO-FRONT` (or `BFT` for short), that takes an item and a list and returns a version where all the occurrences of the item in the given list are brought to the front of the list. For instance, `(bring-to-front 'a '(a b r a c a d a b r a))` would return `(A A A A A B R C D B R)`; and `(bring-to-front` would return `(B B A R A C A D A R A)`. You are NOT allowed to count the occurrences of the item in the given list or use `REMOVE`.

□

Exercise 4.23
Define a procedure that groups the elements in a list putting consecutive occurrences of items in lists. For instance, `(group '(a a b c c c d d e))` should give `((A A) (B) (C C C) (D D) (E))`. Note that you should NOT bring together non-consecutive repetitions; a call like `(group '(a b b c b b c))` should return `((A) (B B) (C) (B B) (C))`.

□

5 More on symbols

1. Symbols are more complicated than they first appear:

Name
Value Binding
Function Binding
Package Binding

2. To set the Value Binding for a symbol, say K:

```
(setf k (random 10))
```

3. Note that SETF also does not obey the rule of evaluation.

4. SETF and LET establish the value binding of a symbol.

5. DEFUN establishes the function binding of a symbol.

6. A symbol gets evaluated to its value binding, unless it appears as the first element of an unquoted list, in which case it gets evaluated to its function-binding.

7. If you want to access the function binding of a symbol elsewhere, you need to prefix the symbol with #'. More on these rules below.

6 Higher order procedures

1. Assume you have a list of integers that you would want to turn into a list of their, say, factorials in a single stroke. And further assume that you would like to have a more general tool, which does the same trick not only with factorial but any unary function you provide to it, e.g. cube, square root, etc. What you need is a function that takes a function and a list as arguments, apply the function one by one to the elements of the list while storing the results in another list. Let us call this function MAPP – note the double 'P' not to clash with the built-in function MAP.

1.1. This is a task that would be straightforwardly implemented with recursion. Assume MAPP is given a function f and some list. If the list is empty, than its MAPP should be empty; if the list is non-empty the MAPP of it is simply the value obtained by applying f to the CAR of the list consed with the value obtained by calling MAPP with f and the rest of the list.

1.2. Here is a definition that attempts to achieve this:

```
(defun mapp (func lst) ;; WRONG!
  (if (endp lst)
    nil
    (cons (func (car lst)) (mapp func (cdr lst)))))
```

1.3. The problem with the above definition is that it expects LISP to bind the function provided by the parameter func to the func that occurs in the function definition. LISP is designed *not* to do this;[16] when you call this function, say with (mapp factorial '(1 2 3 4)), or (mapp 'factorial '(1 2 3 4)), the func in the definition will not get replaced by the parameter you provided for the argument named func. Such replacements are done only with non-initial elements of lists – even not for all such elements, see 1.7. below.

1.4. What we want is achievable with the functions FUNCALL or APPLY.

1.5. FUNCALL wants its first argument to be something that would *evaluate* to:

i. a symbol with a function binding; or

ii. a function.

1.6. The rest of its arguments are treated as arguments of the function provided via the first argument; the function is applied to its arguments and the value is returned.

1.6.1. Assuming the definition of FACTORIAL is loaded,

[16] A dialect of LISP, called Scheme, does this.

```
(funcall factorial 8)
```

would lead to an error. The reason is that before getting fed into FUNCALL the argument FACTORIAL gets evaluated. Remember the rule of evaluation, which says that if a symbol is encountered at a non-initial position in a list, it gets evaluated to its value-binding. In this case LISP cannot find anything in the value-binding. The above specification of FUNCALL says that the first argument should be something that would *evaluate* to a symbol with a function binding. Therefore the correct form is:

```
(funcall 'factorial 8)
```

This way FUNCALL gets a symbol – that's what 'FACTORIAL gets evaluated to – with a function-binding. This function is retrieved and applied to the remaining arguments – we have only one in the present case.

1.6.2. You can use the built-in procedures as follows:

```
(funcall '+ 8 7 29)
(funcall 'member 'a '(z c a t))
```

1.6.3. The specification of FUNCALL in 1.5. has a second clause, which says that one can also provide an argument that evaluates to a function. Given a symbol, you can access its function-binding in two ways:

```
(function factorial)
(symbol-function 'factorial)
```

note the quote in the second form. The first form is the frequent one and like the QUOTE function, it has an abbreviated form:

```
#'factorial
```

Therefore you can directly send a function as an argument to FUNCALL as,

```
* (funcall #'+ 8 7 29)
* (funcall #'factorial 9)
```

1.7. Now you might think that with the below code we could get what we want, for instance with
(mapp factorial '(1 2 3 4))
we get (1 2 6 24).

```
1 (defun mapp (func lst) ; STILL WRONG!
2   (if (endp lst)
3     nil
4     (cons (funcall (function func) (car lst)) (mapp func (cdr lst)))))
```

but again FUNCTION – QUOTE is no different – prevents the argument FUNC from getting bound to the parameter provided to the MAPP.
The correct code is:

```
1 (defun mapp (func lst)
2   (if (endp lst)
3     nil
4     (cons (funcall func (car lst)) (mapp func (cdr lst)))))
```

and you can call the function in two ways, both are fine:

```
(mapp 'factorial '(1 2 3 4))
(mapp #'factorial '(1 2 3 4))
```

2. **MAPCAR.** LISP has a built-in called MAPCAR that does what our MAPP does and a little more (see Ex. 6.6).

3. **Global variables.** In most applications, data is read from a file, entered by the user, or provided by a stream over a network. We will come to these topics, but for now, we will assume that data is stored in a **global** variable in our program. One way to declare and assign a global variable is DEFPARAMETER. For instance, let us assume we have a set of grades:

```
(defparameter *grades*
    '(86 98 79 45 0 75 96 83 91 90 0 70 85 82 91 47 0 70))
```

3.1. What makes a global variable global is that you can access the variable from within any function definition you have in the same file.

3.2. The asterisk characters * around the word "grades" have no special meaning for LISP. It is a convention among LISP programmers to name global variables as such; thereby you can recognize that a variable is global only by looking at its name.

4. Having all the grades stored in a list, let us define a simple procedure that computes their sum. What about this one?

```
(defun total (lst) ; WRONG
  (+ lst))
```

Evaluating `(total *grades*)` would give an error. The reason this is unsuccessful is that the + operator works on numbers, but we provide a list as an argument. Using FUNCALL would not help as well. The construct to use in such situations is APPLY. It is like FUNCALL with the only difference that the arguments are provided in a list.[17]

```
(defun total (lst)
  (apply #'+ lst))
```

Now, evaluating `(total *grades*)` should give you the number 1188.

5. There is no limit to the complexity of the procedures you can use with MAPCAR. Let us first define a procedure that gives the letter grade corresponding to the numerical value. You can easily do this by COND:

[17] Actually, the rule for APPLY is more flexible than this, only the last argument to the it need be a list, but we will ignore this possibility for now.

```
(defun letter (grade)
  (cond ((> grade 89) 'AA)
        ((> grade 84) 'BA)
        ((> grade 79) 'BB)
        ((> grade 74) 'CB)
        ((> grade 69) 'CC)
        ((> grade 64) 'DC)
        ((> grade 59) 'DD)
        ((> grade 54) 'FD)
        (t 'FF)))
```

Now, doing,

```
(mapcar #'letter *grades*)
```

would give,

```
(BA AA CB FF FF CB AA BB AA AA FF CC BA BB AA FF FF CC)
```

Or, you can define a function APPEND-LETTER,

```
(defun append-letter (grade)
  (list grade (letter grade)))
```

which pairs each numerical grade with the letter grade, and doing,

```
(mapcar #'append-letter *grades*)
```

would yield,

```
((86 BA) (98 AA) (79 CB) (45 FF) (0 FF) (75 CB)
 (96 AA) (83 BB) (91 AA) (90 AA) (0 FF) (70 CC)
 (85 BA) (82 BB) (91 AA) (47 FF) (0 FF) (70 CC))
```

6. **LAMBDA.** In LISP you can define anonymous procedures with LAMBDA. An anonymous procedure is useful in cases where you need to define a procedure locally without giving it a name. The procedure APPEND-LETTER could be defined by LAMBDA as follows:

```
(lambda (grade)
  (list grade (letter grade)))
```

The entire LAMBDA expression can be thought as the name of the procedure. Therefore, you could obtain the letter-grade mapping by doing:

```
(mapcar #'(lambda (grade) (list grade (letter grade))) *grades*)
```

Usually LAMBDA procedures are written with short parameter names, like in the following,

```
(mapcar #'(lambda (x) (list x (letter x))) *grades*)
```

6.1. As a LAMBDA expression is a name of a procedure, you can use it in a procedure call. This is very useful in testing your LAMBDA procedures at the top-level. For instance try the following at the top-level:

```
((lambda (x) (* x x)) 8)
```

7. **REDUCE.** Remember that we had to use APPLY to get the sum of a list of numbers. Another way to do this is to use REDUCE. This is somewhat similar to MAPCAR. It takes a function with two arguments and a list; then it reduces the entire list to a single value, by applying the function first to the first two elements, then to the result of this application and the third element, then the result of this application and the fourth element, and so on until it reaches the end of the list. For instance, the following function calls will give you the sum and the mean of the grades, respectively.

```
(reduce #'+ *grades*)
(/ (reduce #'+ *grades*) (length *grades*))
```

8. Usually we would not want to include the 0 grades in computing the mean, especially if we know that these are coming from people who did not participate in the exam or the course. A useful pair of list filtering functions are REMOVE-IF and REMOVE-IF-NOT. To filter out zero grades, just do:

```
(remove-if #'zerop *grades*)
```

9. If we want to have a function that computes the mean of the grades, with first filtering the zero values, we might have the following:

```
(defun class-mean (grades-list)
  (float (/
           (reduce #'+ (remove-if #'zerop grades-list))
           (length (remove-if #'zerop grades-list)))))
```

10. The code is inefficient as we compute the very same thing twice. We can improve this with LET:

```
(defun class-mean (grades-list)
  (let ((real-grades (remove-if #'zerop grades-list)))
    (float (/ (reduce #'+ real-grades) (length real-grades)))))
```

11. Remember that LET has the following syntax:

```
(let ((<variable_1> <value_1>)
      (<variable_2> <value_2>)
          .
          .
          .
      (<variable_n> <value_n>))
      <body>
     )
```

where <body> is just an ordinary function body.

Exercise 6.1
Write LAMBDA expressions[18] that

- returns the greatest of two integers.
- given two integers, returns T if one or the other divides the other without remainder.
- given a list of integers, returns the mean.
- given a list of integers, returns the sum of their factorials – use your factorial solution.

□

Exercise 6.2
Define a procedure PAIR-PROD using MAPCAR and LAMBDA, which takes a list of two element lists of integers and returns a list of products of these pairs. E.g. an input like `((7 8) (1 13) (4 1))` should yield `(56 13 4)`. □

Exercise 6.3
Define your own REMOVE-IF. □

Exercise 6.4
Define a procedure APPLIER that takes a procedure `proc`, an input `input` and a count `cnt`; and gives the result of applying `proc` to `input` `cnt` times. For instance, `(APPLIER #'CDR '(1 2 3) 2)` should give `(3)` □

Exercise 6.5
Write a function REPLACE-IF, which takes three arguments: a list LST, an item ITEM and a function TEST, and replaces every element of LST that passes the TEST with ITEM. You may find using keyword arguments useful (see the lecture notes). Make use of MAPCAR, LAMBDA and FUNCALL in your solution. □

Exercise 6.6
MAPCAR can work on any number of lists; you only need to be careful to provide a function with the correct number of arguments. For instance

```
1 (mapcar #'(lambda (x y) (+ x y)) '(1 2 3) '(4 5 6))
```

gives `(5 7 9)`. Don't worry if lists are not of equal length, MAPCAR goes as far as the shortest list.

Define procedures that use MAPCAR and LAMBDA and

- zip two lists together – `(zip '(a b) '(1 2))` should give `((A 1) (B 2))`.
- take three lists: first two will be lists of integers, and the third is a list of functions. Apply the corresponding function to corresponding arguments.

□

---

[18] If needed, consult Graham on how LAMBDA works.

7 Applicative programming

1. Example: A Collatz sequence is obtained by starting with an integer n; if n is odd, add $3n+1$ to the sequence, if n is even add $n/2$ to the sequence. If you obtain 1 stop, otherwise go on as before with the lastly added integer. The Collatz Conjecture states that no matter which integer you start the sequence, you are guaranteed to reach 1 and stop after a finite number of iterations. Let us write a function that computes the Collatz sequence for a given integer.

```
(defun collatz-generate (n)
  (if (= n 1)
    '(1)
    (cons n (collatz-generate (if (evenp n)
                                  (/ n 2)
                                  (+ (* n 3) 1))))))
```

2. Let us define Collatz (sequence) length of an integer to be the number of steps needed to reach 1 from that integer. This is one less than the length of Collatz sequence. So,

```
(defun collatz-length (n)
  (- (length (collatz-generate n)) 1))
```

3. Knowing how numbers are correlated with their Collatz length would be interesting; does the length grow as the number grows, for instance? Or is there a fluctuating pattern? To do this let us first find a way of generating a sequence of integers within a given range; after this it would be easy to MAPCAR it to what we want to investigate. We already wrote a function for generating ranges; this time we will write a more sophisticated one. We will use keyword arguments in doing that. First the code, then we will look at it in detail.

```
(defun ranger (&key (start 0) (end 9) (step 1) (acc nil))
  (cond ((>= start end) (cons start acc))
        (t (ranger :acc (cons end acc)
                   :start start
                   :end (- end step)
                   :step step))))
```

this range function generates a range including its start and end points. Keyword arguments allow you to refer to the parameters by names – don't forget the ':' before the keywords – so that you do not have to remember the order of the parameters, as you should for optional parameters.[19]

4. Now we have enough machinery to list the Collatz lengths of the first one million integers:

```
(mapcar #'collatz-length (ranger :start 1 :end 1000000))
```

5. Check the maximum Collatz length in this range:

```
(apply #'max
    (mapcar #'collatz-length (ranger :start 1 :end 1000000)))
```

6. MAX breaks down for 1,000,000 range – it should respond 350 for 100,000. We define our own maximum function, which operates on lists:

[19] The function has a flaw; it may not work as expected for some ranges with step different than 1, can you see why? Any ideas to fix it?

```
(defun maxx (lst &optional (max nil))
  (cond ((endp lst) max)
        ((null max) (maxx (cdr lst) (car lst)))
        (t (maxx (cdr lst) (if (> (car lst) max)
                               (car lst)
                               max)))))
```

7. Now we can check the maximum Collatz length in a range of one million:

```
(maxx (mapcar #'collatz-length (ranger :start 1 :end 1000000)))
```

8. We can observe how the maximum Collatz length changes with the size of the range. Here is how to do it with LAMBDA:

```
(mapcar
    #'(lambda (range)
        (maxx (mapcar #'collatz-length
            (ranger :start 1 :end range))))
    '(10 100 1000 10000 100000 1000000))
```

giving (19 118 178 261 350 524).

9. It may be hard to count 0's in specifying various sizes of ranges – it is easy to err. Let us handle that task with mapcar as well.

```
(mapcar #'(lambda (x) (expt 10 x)) '(1 2 3 4 5 6))
```

Now we can have

```
(mapcar #'(lambda (range)
            (maxx (mapcar #'collatz-length
                          (ranger :start 1 :end range))))
        (mapcar #'(lambda (x)
                    (expt 10 x)) '(1 2 3 4 5 6)))
```

10. We have gained some knowledge on Collatz sequences, but we still do not know about individual numbers. We can pair numbers with their Collatz lengths:

```
(mapcar #'(lambda (x) (list x (collatz-length x)))
    (ranger :start 1 :end 1000000))
```

11. Let us find the number with the maximum Collatz length. MAXX on its own is not helpful here; it justs finds the maximum number in a list of numbers. What we need is a way to find the pair(s) with the maximum second element. One way is to write a function similar to MAXX. Let us do this. But let us do this in a way that will have a general use. We will parametrize MAXX in such a way that, the caller/user of the function will decide where MAXX should look for items to compare. In the above version it looks at top most elements, by checking the CAR of the list in every recursion. Here is our new, more "customizable" MAXX.

```
(defun maxx (lst &key
                 (max nil)
                 (hook #'(lambda (x) x)))
  (cond ((endp lst) max)
        ((null max) (maxx (cdr lst) :max (car lst) :hook hook))
        (t (maxx
             (cdr lst)
             :max (if (> (funcall hook (car lst))
                         (funcall hook max))
                    (car lst)
                    max)
             :hook hook))))
```

12. With our new `MAXX` in our hands we can now find the number with the largest Collatz length in a given range, say 1000.

```
(maxx (mapcar #'(lambda (x)
                  (list x (collatz-length x)))
              (ranger :start 1 :end 1000)) :hook #'cadr)
```

Exercise 7.1
Find the numbers in a given range that have the same Collatz length using the techniques of this section. □

8 Iteration

1. Computing frequently involves repeating an action, each time with different inputs and/or conditions. There are mainly three perspectives on how to do this: (i) iterative, (ii) recursive, (iii) applicative. Which perspective is best mainly depends on the task at hand.

2. Let's take a list and print each element to the screen, no matter what.

3. But first we need to learn how to print stuff to screen; that's what we mean by "saying". Try this:

```
(print 'hello)
(print 8)
(print (* 2 4))
```

3.1. You'll see everything printed twice to screen. Why is that?

3.2. A list headed by PRINT is evaluated specially. The one and only argument to PRINT gets evaluated and is returned by PRINT. Besides this, PRINT also prints this value to the screen. Every LISP expression returns a value. PRINT not only returns a value, but also does something else – it has a **side effect**. We will talk more on this.

4. Now we are ready to write our function:

```
(defun printer (lst)
  (dolist (x lst 'done)
    (print x)))
```

5. DOLIST is an example of a **block structure**. Its job is to run for as many times as there are elements in LST. Runs are called **iterations**. In each iteration, the variable X holds the next element in LST. The symbol 'done is what will be returned when all the elements in the list are iterated over. The rest of the DOLIST list, namely its CDDR, is a possibly empty sequence of expressions, that are evaluated in order in each iteration of the DOLIST.

5.1. Observe that we see each element only once on the screen, nothing is repeated. The reason is that DOLIST does not care about values returned by the expressions in its body; the value it returns is always the third element in its first argument (= the list where you name your iteration variable). In the body of a DOLISTonly side-effects count, anything that doesn't have a side effect has no place in the body of a DOLIST.

5.2. For instance, imagine you wrote the code below to double a list, by adding each element to the end of the list in order:

```
(defun no-fun (lst)
  (dolist (x lst 'done)
    (append lst (list x))))
```

All this does is to iterate through the list; evaluate the APPEND form in each iteration, and finally evaluate 'DONE and return its value, which is the symbol DONE. The list we sent to the function stays totally intact, nothing added, nothing removed. The reason is that APPEND has no side effect, it just returns a value: the newly formed list obtained by bringing together two other lists. Therefore nothing happens from the perspective of DOLIST, it doesn't "see" these returned values.

6. Now we go one step further and try to achieve the task of computing the length of a list. So far, what we are doing is to simply iterate and wait for something to happen in each iteration. In the present task, while iterating, we need to keep track of how many elements we have seen so far. If we can do that, we can return that number when we reach the end of the list as the computed length of the list. "Keeping track of something" means storing its value somewhere that we can look up and update the value when we need. Such places are called **variables**. What we need here is to first create a variable that has the initial value of 0; then we will **increment** the variable by one in each iteration; and then we will return the final state of the variable.

7. Creating a variable is done by a `LET` construct and changing the state of the variable is done by `SETF`. Let's go over an example, type the following to top-level:

```
(let ((x 1)) (print x) (setf x 4) (print x))
```

7.1. `LET` consists of a list of variable declarations and a number of expressions that constitutes its **body**. The list of variable declarations is a list consisting of a number of variable declarations – as many as you like – where each declaration itself is a list of two elements: a variable name and an expression whose *value* will be stored in the variable just named. In our present case, we have only one such declaration, a very simple one. After this declarations list, comes the body of `LET`. The body can have as many expressions as you like – here we have three. `LET` evaluates each expression in its body one by one, and returns the value of the expression it evaluated last. In this case, the three expressions we put in the body of `LET` all have side-effects: first 1 is printed, then `X` is made to hold 4 instead of 1; finally 4, which is the value of `X` at that point, is printed. But you see an extra 4 as well. That's the value returned by the `LET` clause itself – remember that `LET` returns the value of the last expression in its body and top-level, also called REPL, reads an expression, evaluates it and prints the value. Therefore the first 4 you see is printed by `PRINT`, the second 4 you see is printed by the top-level.

7.2. Now save the expression above to a file, say `let.lisp`, and load the file at the top-level by doing:

```
(load "let.lisp")
```

Now you do not see the second 4, but a `T` instead. Can you see why? The reason is this: the top-level prints the value of the top-most expression it evaluated. In this case the top-most expression is a `LOAD` expression – which is, by the way, just another `LISP` list, standing for a function with a side-effect; its side effect is to load the file whose name is given to it. As every other `LISP` expression, `LOAD` as well returns a value: if everything goes fine with loading the file, it returns `T`. And it is this `T` that you see instead of the second 4.

8. Now we can write our function that computes the length of a given list:

```
(defun length2 (lst)
  (let ((counter 0))
    (dolist (x lst counter)
      (setf counter (+ 1 counter)))))
```

Let us have a closer look at what's going on here:

8.1. First, you see a strange warning message when you load the program. `SBCL` warns you that you have something named `X` in your `DOLIST`, which you do not use anywhere else. This is a style warning; something that is aimed to alert the programmer that there *may* be something wrong, but doesn't have to be. In our case there is nothing wrong; we need to iterate over a list, but we are not interested in the elements; all we care to know is how many of them there are. So, ignore the warning.[20]

[20] There are ways to "muffle" (=make silent) such warnings; but for now let them stay, sometimes they are helpful.

8.2. Second, there is no quote on `COUNTER` in the `DOLIST` variable specification, we want `DOLIST` to return the *value* of the symbol, not the symbol itself, as we did with `'DONE`.

8.3. Third, observe how we update the value of the variable `COUNTER`. `SETF` is again a special form, it does not obey the rule of evaluation, it has its own rule. It first evaluates its second argument; then stores the value obtained in its first argument; and finally *returns* the value stored. Again, like `PRINT`, it has both a side-effect (= change the value stored in a variable) and a return value (= the value itself).[21]

9. Similarly we can write a function that sums the numbers in a list:

```
(defun summer (lst)
  (let ((sum 0))
    (dolist (x lst sum)
      (setf sum (+ sum x)))))
```

10. Or, take a list of numbers and return another list that contains only the even ones in the original list:

```
(defun get-evens (lst)
  (let ((store nil))
    (dolist (k lst (reverse store))
      (if (evenp k)
        (setf store (cons k store))))))
```

We `SETF` only on the condition that the current iterated item is even; and we omit the final argument of `IF`, where `NIL` is returned when the test fails. This does no harm, because `DOLIST` does not care about return values anyway. Finally, observe that we do not return the `RESULT` itself, but its reverse as the value of `DOLIST`; otherwise we would have obtained the even numbers in the original list in the reverse order of their appearance.

11. So far we kept counters and stores that we conditionally increment or add elements along the iteration. Some problems require keeping a **flag**. A flag is a variable that holds a value that allows you to make decisions, usually boolean values like `T` and `NIL`.

11.1. Take the task of removing every odd occurrence of a symbol from a list – remove the first, third, fifth etc.

```
(defun remove-odd (elm lst)
  "remove odd occurrences of elm from lst"
  (let ((remove? t)
        (store nil))
    (dolist (a lst (reverse store))
      (if (equal a elm)
        (if remove?
          (setf remove? nil)
          (progn
            (setf store (cons a store))
            (setf remove? t)))
        (setf store (cons a store))))))
```

[21] `SETF` is actually capable of much more, but it's more than enough for now.

We keep a flag called REMOVE?, which keeps track of whether we have seen an even or an odd number of ELMs so far. When REMOVE? is T it means we have seen an even number of ELMs; therefore we start with T – zero is an even number. When REMOVE? is T, it means that we need to delete the next ELM we encounter. We first check whether the current item is ELM, and if not, we simply cons the current symbol to STORE. If the current symbol is ELM, we need to decide whether to take it into STORE or not. If our flag REMOVE? is T, all we need to do is to change the status of our flag – we could also have done `(setf remove? (not remove?))`; not consing the current element to STORE "deletes" it. You encounter a new construct, PROGN, in the failure clause of the internal IF. When the current item is equal to ELM and REMOVE? is NIL, we need to do two things: cons the current element to STORE and switch REMOVE? to T. As you will remember, IF is a three argument function, a test, an expression to be evaluated when the test succeeds, and an expression to be evaluated when the test fails. Therefore, there is room for only one expression for the failure case. But we have two SETFs to be evaluated. To solve this problem, we group the two expressions in a PROGN block. PROGN is a list that can have as many elements you like; what it does is to evaluate these elements one by one and return the value of the last one as the value of PROGN.

11.2. Here is a slightly simplified version of the same program.As an exercise, trace the execution of the program with pen and paper on a sample input.

```
(defun remove-odd (elm lst)
  "remove odd occurrences of elm from lst"
  (let ((remove? nil)
        (store nil))
    (dolist (a lst (reverse store))
      (if (equal a elm)
        (progn
          (setf remove? (not remove?))
          (if (not remove?)
            (setf store (cons a store))))
        (setf store (cons a store))))))
```

12. In iterative programs incrementing/decrementing a counter and updating a store variable by consing an element are quite common tasks. For this reason, LISP has some shortcuts for these operations.

    `(incf counter)` is equivalent to `(setf counter (+ counter 1))`.

    You can also specify the amount of increment by a second argument to INCF.

    `(incf counter 3)` is equivalent to `(setf counter (+ counter 3))`.

    DECF does the same thing, this time by subtracting instead of adding.

    For updates by consing elements,

    `(push x y)` is equivalent to `(setf y (cons x y))`.[22]

13. Let us now write a function that checks whether a given element is a member of a given list. Here is one way to do it,

[22] There is a dual of PUSH as well: POP. We will use and explain it later.

```
(defun elementp (item lst)
  (let ((answer nil))
    (dolist (k lst answer)
      (if (equal k item)
        (setf answer t)))))
```

So nice. But there is one efficiency issue. Imagine you are given a very long list and the item you are looking for is somewhere close to the front of the list. With the above strategy, you will have to wait untill `DOLIST` finishes iterating through the entire list in order to get the value – `T` or `NIL` – you want to know, even the value is settled long before you see the outcome. For such kind of situations, `LISP` has the construct `RETURN`. Let us first see it in action:

```
(defun elementp (item lst)
  (dolist (i lst)
    (if (equal i item)
      (return t))))
```

First notice that the first argument of `DOLIST` has two elements rather than three; the third argument, which specifies the return value of `DOLIST` is not given, and therfore it gets its default value `NIL`. This means that if `DOLIST` can reach the end of the list, it will return `NIL`. The only case that this will not happen is that the equality test of the `IF` in the body returns `T`, in which case the `DOLIST` iteration is aborted and the value `T` is returned as the value of the entire `DOLIST`, and thereby the value of the call to `ELEMENTP`. This is much shorter than the previous code; and it does not have to run till the end of the list, in case it finds the item somewhere before the end.

14. Now we will see another common programming technique. But first we need a problem that requires the use of it. Our task is to decide whether a given list is a sublist of another. In the previous tasks, we were keeping counters and flags; this time we will keep a **window**. A window is a fixed size list that is updated in each iteration. For instance, assume we are asked to see whether a three element list (let's call this the search list) is a sublist of another list (let's call this the mother list). What we need for this problem is a three element window. We will start with an empty window and will update it as we iterate through the mother list. The proper way to update the window is to add elements to the back of the window, while discarding the element at the front – otherwise our window will grow beyond the size of the search list. Apart from updating our window, we will also check whether our window equals to the search list. If we get the equality, we will `(RETURN T)`, otherwise we will go on iterating and updating our window. If no equality is catched until the end of the mother list, we will return `NIL`.

14.1. A critical step in our solution is to create a window of a given size. There is a very useful function `MAKE-LIST`, which constructs a list of a given length, initially holding `NIL`s.[23]

Here is the code:

[23] Our program will assume that `NIL` will not be the element of any search list. There is a way to avoid this shortcoming: namely telling `MAKE-LIST` to fill the list it constructs with symbols that are different from anything that can be in the input lists. We will see how to do this below.

```
(defun sublistp (search-list mother-list)
  (let ((window (make-list (length search-list))))
    (dolist (i mother-list)
      (setf window (append (cdr window) (list i)))
      (if (equal window search-list)
        (return t)))))
```

15. Another builtin iterative construct is DOTIMES. It's similar to DOLIST; instead of iterating a list, the iterative variable iterates over numbers from 0 up to 1 minus the given iteration parameter – where there was a list in DOLIST, there is a number.

15.1. Here is a function definition that takes its first argument to the power of its second argument.[24]

```
(defun exponent (base power)
  (let ((result 1))
    (dotimes (x power result)
      (setf result (* base result)))))
```

15.2. Here is another example with DOTIMES that computes the factorial of a given number. Factorial of an integer n is the number $n \times (n-1) \times (n-2) \times \ldots \times 1$, the factorial of 0 is 1 by definition.

```
(defun factorial (n)
  (let ((result 1))
    (dotimes (i n result)
      (setf result (* result (+ i 1))))))
```

In using DOTIMES, you need to bear in mind that the iteration starts with 0 and ends with one minus the given iteration limit.

16. **Iterating on more than one list in parallel.** One drawback of DOLIST and DOTIMES is that they cannot handle more than one list in parallel.

16.1. Assume you have two lists: one with student names and the other with grades, where lists are so nicely ordered that names and grades match position-wise. You want to merge these lists into a single list, where each element is itself a list of two elements, the first is a name and the second is the corresponding grade. To accomplish this, we will need a way to refer to the elements of a list by their positions. The builtin for this is NTH. It takes a number argument, which is the position, and a list. You should always keep in mind that positions start with 0 in LISP: the first element sits at position 0, the second at 1, and so on.

16.2. Here is the function that solves our name-grade pairing problem (and many similar others):

```
(defun pairlists (lst1 lst2)
  (let ((store nil))
    (dotimes (i (length lst1)  store)
      (push
        (list (nth i lst1) (nth i lst2))
        store))))
```

Notice how we broke down PUSH to increase readiblity of the code. A sample interaction of the program would look like:

[24] x to the power y, designated as x^y, is the number obtained by multiplying x by itself for y times.

```
* (pairlists '(john mary ted) '(82 74 42))

((TED 42) (MARY 74) (JOHN 82))
```

16.3. There is one thing we need to be careful about while pairing lists with this method; namely we need to decide what happens when the lists to be paired are not of equal length. One possible action to take is to see which list is shorter and use the length of that list with DOTIMES. In this strategy, the excess elements in the longer list will get discarded. Let us do the job of deciding on the smaller length with a separate function. This will make our program easier to understand and test. Also let us write our shorter-length-detector function in a general way, such that it will find the minimum length not only of a pair of lists but any number of lists we provide to it. Here is a way to do it:

```
(defun min-length (list-of-lists)
  (let ((minlen (length (car list-of-lists))))
    (dolist (x (cdr list-of-lists) minlen)
      (let ((len (length x)))
        (if (< len minlen)
          (setf minlen len))))))
```

16.4. Now we can update our list pairing function as:

```
(defun pairlists2 (lst1 lst2)
  (let ((store nil))
    (dotimes (i (min-length (list lst1 lst2)) store)
      (push
        (list (nth i lst1) (nth i lst2))
        store))))
```

17. **Changing the way iteration proceeds.** Another drawback of DOLIST and DOTIMES is that how they traverse a list or increment a number is fixed, namely one-by-one from left to right and increasing. And this drawback is not as easy to remedy as the other drawback. We will see how to do it later. But if you can't wait that long, go to Graham (1996) and study how to use DO.

In the exercises of this section you need to use iterative constructs as much as you can.

Exercise 8.1
Define a procedure APPEND2 that appends two lists. □

Exercise 8.2
Define an iterative procedure UNIQ that takes a list and removes all the repeated elements in the list keeping only the first occurrence. This is the expected behavior:

```
* (uniq '(a b r a c a d a b r a))
(A B R C D)
```

□

Exercise 8.3
Define a procedure that reverses the top-level elements of a list. □

Exercise 8.4
The mean of n numbers is computed by dividing their sum by n. A running mean is a mean that gets updated as we encounter more numbers. Observe the following input-output sequences:

```
* (run-mean '(3 5 7 9))
(3 4 5 6)
```

The first element 3 is the mean of the list `(3)`, the second element 4 is the mean of `(3 5)`, and so on. Implement `RUN-MEAN` by using `DOTIMES` and `NTH`. □

Exercise 8.5
Define a function `SEARCH-POS` that takes a list as search item, another list as a search list and returns the list of positions that the search item is found in the search list. As usual, positioning starts with 0. Use `DOTIMES`. A sample interaction:

```
* (search-pos '(a b) '(a b c d a b a b))
(6 4 0)
* (search-pos '(a a) '(a a a a b a b))
(2 1 0)
```

□

Exercise 8.6
Define a procedure that reverses the elements in a list including its sublists as well. □

Exercise 8.7
See the `PAIRLISTS` in lecture notes. Define a procedure that "pairs" an arbitrary number of lists. Here is a sample interaction:

```
* (pairlists '((a b) (= =) (1 2) (+ -) (3 9)))
((A = 1 + 3) (B = 2 - 9))
```

□