## 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.** Procedure 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: (a) `(NOT U)`, `(AND X Y Z)`, `(OR X Y Z)`.[10]

□

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

```
(defun custom-if (test succ fail) ; wrong!
  (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:

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

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.