1 Cognition, computation, computers, programs

1.1 Cognition and computation

1. Cognitive Science: mind/brain is analogous to software/hardware.

1.1. Various names: "the computational view of mind," "information processing psychology," "the computational theory of mind," or simply "computationalism."

1.2. The analogy is usually far from strict and there are many varieties.

1.3. For instance: "mind/cognition is computation" versus "mind/cognition is computable."

2. Why we need computationalism?

2.1. Compare the tasks:[1]

- Predicting the trajectory of celestial bodies, say the motion of the earth in the next six hours.
- Predicting the next move of a chess player at a given state of the game.

2.2. Crane (2003:103): "The planets do not 'compute' their orbits from the input they receive: they just move."

2.3. In the case of chess, physical description and physical laws are helpless. The source of helplessness is twofold: (i) The state of a chess game has infinitely many physical realizations, and whether a physical state is a game state is dependent on the whether or not some cognitive agents interpret the "scene" as a chess game or not. Therefore a function from physical description to game state is at best extremely complex. (ii) Even if we find a way from physical description to game state, the physical description of rule-based behavior would be a function of the particular realization function (the mapping from physics-to-chess states) at that particular occasion (Pylyshyn 1984).

3. Computation and cognition share an essential property: both operate on rules and representations, yet are based on (instantiated by) physical causal systems. What is happening in a computer and mind/brain are quite similar.

   Levels (more on this below):

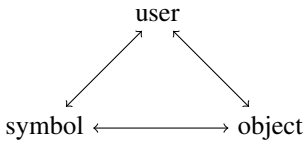
   i. physical (device)
   i. symbolic (syntactic)
   iii. intentional (semantic)

[1] We gloss over a potential objection to the legitimacy of this comparison, given the huge difference between the well-definedness of the questions.

1.2 What is (symbolic) computation?

1. Symbols (or signs) and signification are central concepts in language, logic, and computation.

1.1. Take signification as a three-part relation:



- A user uses a sign to **refer** to an object.
- A sign **denotes** an object.
- A user has certain **intentions** about an object.

1.2. A sequence of binary digits (bits) is a symbolic representation:

$$001011010100111000000010$$

2. A computational process is a sequence of manipulations performed over symbolic representations.

2.1. Example: the process by which you flip the digits of the representation above, one at a time is a computational process.

1.3 Abstract machines (optional)

- A Turing Machine (TM) is specified as a quintuple $\langle \Sigma, Q, q_0, q_H, \omega \rangle$, where

Σ	is an alphabet of admissible symbols (including a symbol for blank cells);
Q	is a finite set of internal states;
$q_0 \in Q$	is the starting state;
$q_H \in Q$	is the halting state;
ω	is the state transition function of the form:

$$\langle q, \sigma \rangle \mapsto \langle q', \sigma', M \rangle$$

where q and q' are states, σ and σ' are symbols, and $M \in \{-1, 0, 1\}$, for 'move left', 'don't move', and 'move right', respectively.

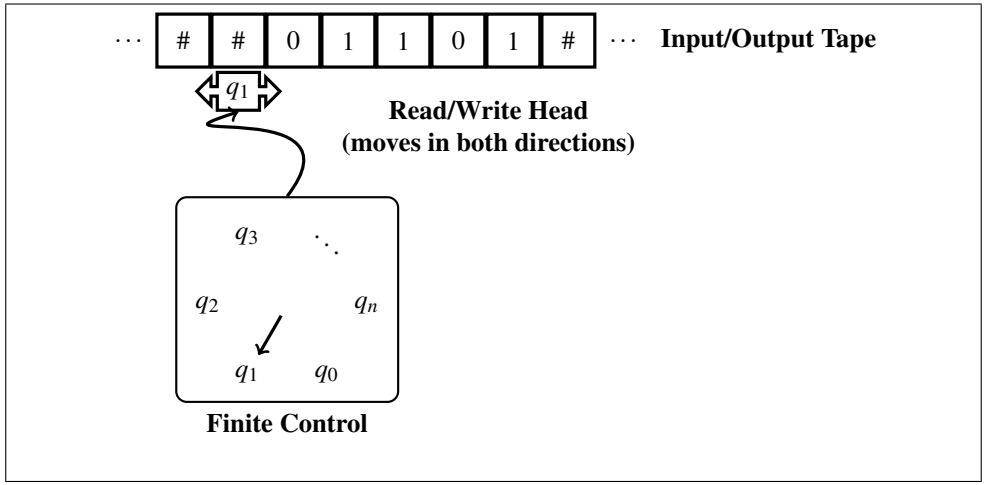


Figure 1: A Turing Machine

Example 1.1
A TM that decides whether its input is a palindrome. Let # be the blank symbol, q_0 the initial, and q_8 the halting state.

$$
\begin{array}{llll}
\langle q_0,\#\rangle \mapsto \langle q_8,1,0\rangle & & \langle q_4,\#\rangle \mapsto \langle q_8,1,0\rangle \\
\langle q_0,0\rangle \mapsto \langle q_1,\#,1\rangle & & \langle q_4,0\rangle \mapsto \langle q_7,\#,-1\rangle \\
\langle q_0,1\rangle \mapsto \langle q_2,\#,1\rangle & & \langle q_4,1\rangle \mapsto \langle q_5,\#,-1\rangle \\
\langle q_1,\#\rangle \mapsto \langle q_3,\#,-1\rangle & & \langle q_5,\#\rangle \mapsto \langle q_8,1,0\rangle \\
\langle q_1,0\rangle \mapsto \langle q_1,0,1\rangle & & \langle q_5,0\rangle \mapsto \langle q_5,1,-1\rangle \\
\langle q_1,1\rangle \mapsto \langle q_1,1,1\rangle & & \langle q_5,1\rangle \mapsto \langle q_5,1,-1\rangle \\
\langle q_2,\#\rangle \mapsto \langle q_4,\#,-1\rangle & & \langle q_6,\#\rangle \mapsto \langle q_0,\#,1\rangle \\
\langle q_2,0\rangle \mapsto \langle q_2,0,1\rangle & & \langle q_6,0\rangle \mapsto \langle q_6,0,-1\rangle \\
\langle q_2,1\rangle \mapsto \langle q_2,1,1\rangle & & \langle q_6,1\rangle \mapsto \langle q_6,1,-1\rangle \\
\langle q_3,\#\rangle \mapsto \langle q_8,1,0\rangle & & \langle q_7,\#\rangle \mapsto \langle q_8,0,0\rangle \\
\langle q_3,0\rangle \mapsto \langle q_5,\#,-1\rangle & & \langle q_7,0\rangle \mapsto \langle q_7,\#,-1\rangle \\
\langle q_3,1\rangle \mapsto \langle q_7,\#,-1\rangle & & \langle q_7,1\rangle \mapsto \langle q_7,\#,-1\rangle
\end{array}
$$

□

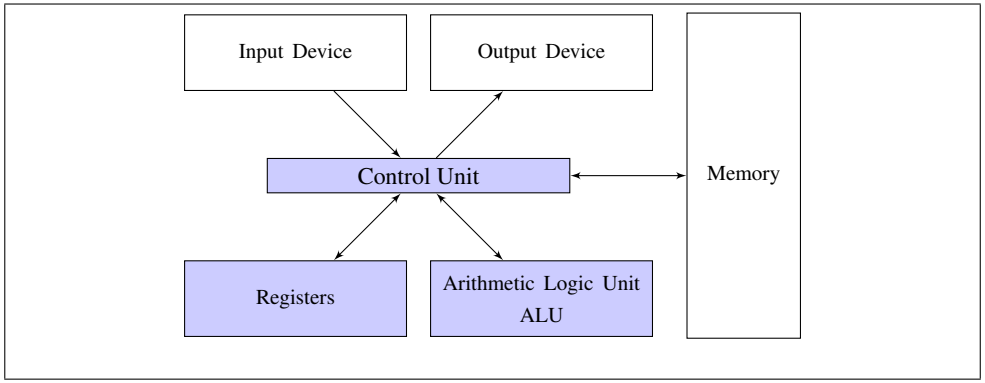


Figure 2: Von Neumann architecture

1.4 Hardware/software

1. Most computers we have around are based on von Neumann[2] architecture.
2. VNA consists of **memory**, **central processing unit**, and I/O devices.
3. Memory consists of a sequence of **cells** (also called **words**).

3.1. Each cell is of a fixed capacity.

3.1.1. What we mean by capacity is how many binary digits (bits) a cell can hold.

3.1.2. Bits are thought of in groups of 8's; 8 bits = 1 **byte**.

3.1.3. A single byte can hold 2^8 different values, or symbols, if you like.

3.2. Each cell of memory has a unique **address**, itself a binary number.

4. The two basic components, CPU and memory, communicate through three channels:
   i. A collection of wires called **address bus**;
   ii. Another collection of wires called **data bus**;
   iii. A single wire called **R/W line**, the status of which signals whether CPU wants to write to or read from the memory.
5. The address bus and the R/W line are one way channels. The value is always dictated by the CPU. The data bus is a two-way channel.
6. The two basic interactions between CPU and Memory go as follows:

[2] Named after the mathematician and physicist John von Neumann.

- CPU sets the address bus and R/W line to W. In that case it also sets the data bus. Obviously, this amounts to dictating to write the data to the given address in memory.
- CPU sets the address bus and R/W line to R. In this case it is the memory that sets the data bus in accord with the data located at the address provided by CPU; this is reading from memory.

7. CPU itself also has some local memory slots. These are called **registers**. Access to registers is faster than access to memory. But there is a reason to keep the size of the CPU small, therefore there are a limited number of registers, which are used to store intermediate results of computations and some frequently used information.

8. The computation unit of CPU is **arithmetic logic unit** (ALU, for short.) ALU is responsible for arithmetic and logical operations.

9. CPU feeds on **instructions**. Some typical types of instructions are:

   i. Store the number at the register X at the memory address Y;
   ii. Fetch the number at the address X and store it at the register Y;
   iii. Add the number at address X to the number at address Y, and store the result at address Z;
   iv. Compute the bitwise *and* of the numbers at addresses X and Y, and store the result at the address Z;
   v. If the contents of the register X and Y are not identical, go to address Z;
   vi. Jump to the address X;
   vii. and so on.

10. **Stored-program** concept

    a. Not only data but also instructions are represented as numbers;
    b. Programs are stored in memory; they can be read and written just like data.

11. A central aspect of a computational system is **flow of control**. Some instructions have *go to* or *jump*, which sends the control to another instruction. But other instructions lack such a mechanism for flow of control.

12. There is a special register called **program counter**, where the address of the next instruction is kept.

13. CPU operates through a sequence of cycles. Each cycle begins by fetching a binary description of what to do, an **instruction** from an address in Memory. Following this, CPU understands, or technically speaking, **decodes** the instruction. The next step is to **execute** the instruction. This completes a single cycle, which is followed by reading the next instruction from Memory.[3]

1.5 Levels of programming

1. Computers operate on numbers, in the sense that the functional architecture of the machines get their instructions as binary numbers organized into expressions of **machine code**.

2. Given a functional architecture, say VNA, a straightforward specification would be to code the memory byte-by-byte (writing **machine code**); so that CPU "knows" what to do in each possible state of the process.

3. Example of an addition instruction:

```
000000 10001 10010 01000 00000 1000000
```

4. A machine code instruction is organized into **fields** with different meanings.

5. Some levels up the machine code is the **assembly** language. The above machine code takes the form:

```
add $s1,$t1,$t2
```

6. Assembly level lies between high-level languages and machine code.

   A simple while loop in C:

```
while (save[i] == k) {
        i = i + 1;
        }
```

   Assuing that `i` and `k` are stored in registers `$s3` and `$s5`, and the array `save` starts at the memory address stored in `$s6`, the above C code is **compiled** to the following assembly code (for MIPS):

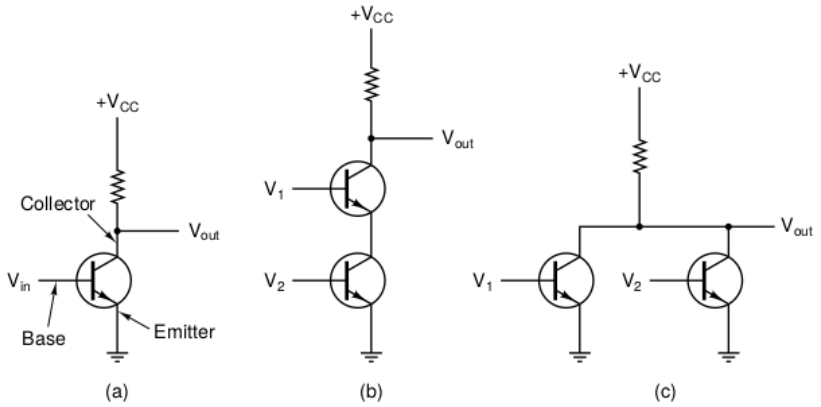
[3] Cycles are called 'fetch-decode-execute' or 'instruction cycle'.

```
Loop:  sll $t1,$s3,2
       add $t1,$t1,$s6
       lw  $t0,0($t1)
       bne $t0,$s5,Exit
       add $s3,$s3,1
       j   Loop
Exit:
```

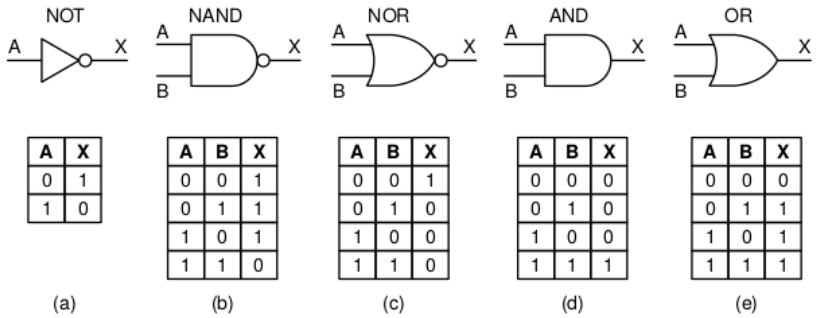
7. A computational process can be specified at various levels.

1.6 More into hardware

1. The basic building block of modern computer is the transistor.[4] Transistors are organized into logic gates.

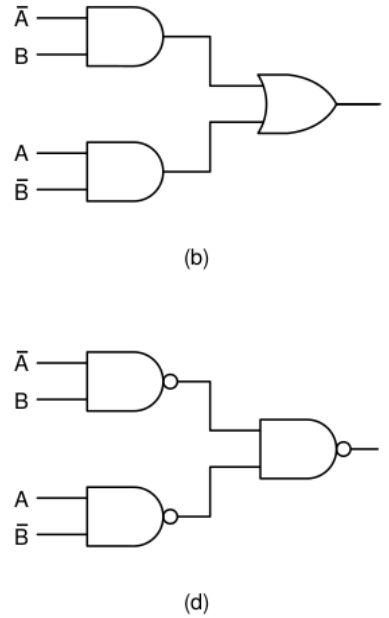


[4]The figures in this subsection are taken from Tanenbaum 1999.



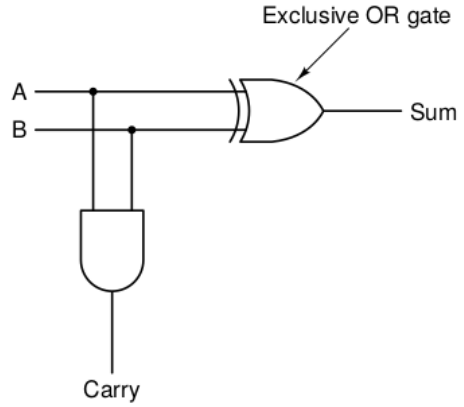
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

(a)

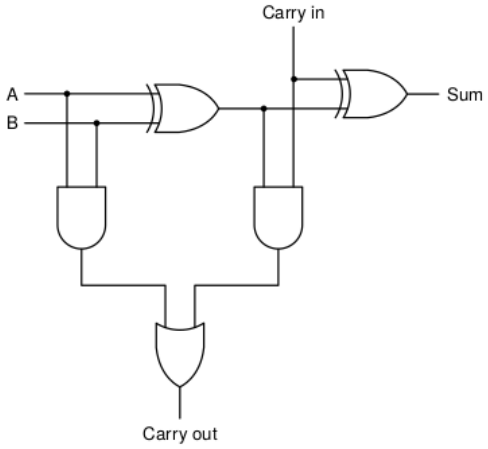


2. Mechanical addition with circuits:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



(a) (b)

1.7 Further reading

Pohl and Shaw (1981) is an introduction to computer science, for non-computer scientists. Tanenbaum (1999) (or a newer edition) is an accessible book on computer architecture. It also provides a short and nice history of computers. Pylyshyn (1984) is a classic on cognition and computation, from the representational camp. Consult Churchland and Sejnowski (1992) for a connectionist approach to the same topic. The last two books are NOT introductory level. For a general introduction, see Crane (2003).

2 Conses and Lists

2.1 Building blocks

1. We start with 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.

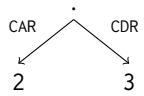
2. A fourth type of entity is a **cons**, a concatenation of two addresses, which is stored in a cell.

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:[5]



2.2 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 separate by white space and enclosed in parentheses:

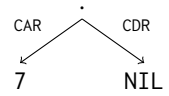
```
(1 3 9)
(RED GREEN BLUE)
(1 GREEN)
()
(7)
```

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

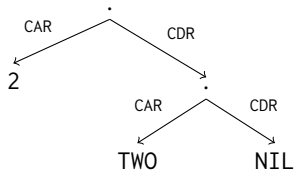
3. The empty list is represented in two equivalent ways: besides `()`, also as `NIL`.

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:

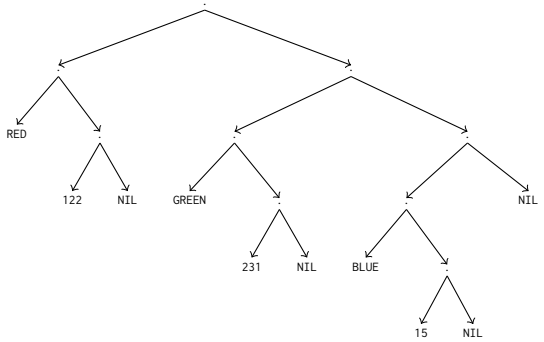


and so on...

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))`:[6]



[5] Think of it as an ordered pair in set theory.

[6] 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.

5. Two basic constructors for lists: `CONS` and `LIST`.

5.1. Make sure you totally understand how they differ.

5.2. See Touretzky 1990:59 on how `LIST` works.

3 Evaluation

1. Open up `SBCL` by typing `rlwrap sbcl` in the command-line and hit return.

2. If you see the prompt `*`, you are at the **top-level**.

3. Whatever you enter at the top-level will be **read**, **evaluated** and the result of the evaluation will be **printed**. This is called REPL, read-evaluate-print-loop.

4. `LISP` has strict rules on how to evaluate what you give to it.

4.1. Numbers and the symbols `NIL` and `T` evaluate to themselves – you get what you give.

4.2. Lists, on the other hand, are evaluated as follows:

4.2.1. The first element of the list is treated as naming a function – `LISP` will bitterly complain if it fails to find a function with that name.[7]

4.2.2. If the first step is successful, `LISP` will evaluate the rest of the elements, from left to right.

4.2.3. If all goes well, it will apply the function stored in the first element to the values obtained by the evaluation of the rest.

5. What do you expect from?

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

6. There is a way to tell `LISP` not to evaluate certain expressions; as in natural language, the device is quotation. The following would behave well:

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

7. There is a shorthand for `QUOTE`:

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

8. Note that `'(a b c)` is not the same as `('a 'b 'c)`.

4 Defining functions

1. To associate a function with a symbol:

```
(defun square (n)
    (* n n))
```

2. Does `DEFUN` obey the rule of evaluation?

3. Now you can use `SQUARE` as the first element of an unquoted list to invoke the square function:

```
(square 7)
```

4. At this point it is natural to think that `SQUARE` is a symbol (or name) of a function, so giving it to top-level should return some representation of this function, after all `LISP` should be familiar with the name `SQUARE`. Try it.

5 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`:

[7] This "name" business is slightly more complicated than stated here, we will get back to it later.

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

3. Note that SETF also does not obey the rule of evaluation.
4. SETF establishes 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 Making decisions

1. Predicates are expressions that take arguments and return T or NIL, that is "true" or "false".
2. Built-in predicates usually end in letter p: `symbolp`, `listp`, `zerop`, but `atom` and `null`.
3. Testing for equality: there are more than one ways for testing equality, but to keep things simple for now, use the two argument predicate `equal`.
4. Comparison predicates: <, >, <=, >=.
5. A predicate applied to its functions is a test. E.g. `(< 3 2)` is a test that evaluates to NIL.
6. The simplest structure to make a decision is IF which takes three arguments:

6.1. a test;
6.2. an expression that will be evaluated and returned in case the test succeeds;
6.3. an expression that will be evaluated and returned in case the test fails.

7. NULL checks whether its argument evaluates to NIL or not, here is a function to return human readable indications of whether a list is empty or not:

```
(defun emptyp (lst)
  (if (null lst)
    'empty
    'not-empty))
```

What happens if you try this with a non-list argument, say the number 4?

Here is another way to test the emptiness of a list:

```
(defun emptyp (lst)
  (if (endp lst)
    'empty
    'not-empty))
```

Now try this one with an empty and non-empty list; why does it work?

```
(defun emptyp (lst)
  (if lst
    'not-empty
    'empty))
```

8. The third argument of `if` is optional; if you do not provide it NIL is returned in case the test fails.

6.1 More on SETF

1. Second argument of SETF can make use of the first:

```
(defvar k)
(setf k 8)
(setf k (+ k 3))
```

2. At first glance SETF might seem to be a simple function which takes a symbol and a value and associates the value-binding of the symbol with the value. Actually it is more clever than that. Observe the following interaction with SBCL:

```
* (defvar k)

K
* (setf k '(A B C))

(A B C)
* (setf (car k) 'Z)

Z
* k

(Z B C)
*
```

3. SETF takes two *expressions* and make sure that they evaluate to the same thing.

4. Also notice that SETF itself has a return value – can you see what?

5. Therefore, SETF does two things at the same time: manipulates some bindings in the memory and returns a value. The first is called a "side effect" of SETF – a little bit confusing given the daily usage of the expression.

6. Knowing the side effects and return values of constructs is crucial.

6.1. Assume you want to write a function that takes a number and a list, and replaces the first element of the list with the number, in case the number is odd. Here is one way to do it:

```
(defun replace-if-odd (a-number a-list)
  (if (oddp a-number)
    (setf (car a-list) a-number)))
```

6.2. Now also assume that you have some reason to check the length of the list after the replacement. You can check the length of a list by the built-in LENGTH. You do this:

```
(length (replace-if-odd 3 '(1 2 3)))
```

and you get an error – can you see why?

6.3. Here is a way to make our function return the modified list, so that we can immediately check its length:

```
(defun replace-if-odd (a-number a-list)
  (if (oddp a-number)
    (cons a-number (cdr a-list))
    a-list))
```

6.4. This time, however, we lost the modified list; we will see a way to both manipulate the list (in-place) and return a value.

6.2 Back to decisions

1. COND is helpful when you have more than two way decisions. Here is a function that adds a number to a set and returns the extended set – remember that sets do not allow repetitions:

```
(defun set-add-number (n numbers)
  (cond ((not (numberp n)) numbers)
        ((member n numbers) numbers)
        (t (cons n numbers))
        )
  )
```

2. Everything that can be done by COND can also be done only by IF.

```
(defun set-add-number (n numbers)
  (if (not (numberp n))
    numbers
    (if (member n numbers)
      numbers
      (cons n numbers)
      )
    )
  )
```

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

3.1. AND Evaluate the expressions until you reach either the end or an expression that evaluates to NIL.

3.2. OR Evaluate the expressions until you reach either the end or an expression that evaluates to something other than NIL.

4. Exercises:

4.1. Write a function HOWCOMPUTE taking 3 numbers, telling the basic arithmetic operation that is used to compute the third number from the first two – it should say so if it cannot find it (Touretzky 1990:ex. 4.13).

4.2. Write (AND X Y Z W) by using cond COND (Touretzky 1990:ex. 4.19).

4.3. It is possible to write IF in terms of AND and OR. A tempting trial would be:

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

But it is unsatisfactory in one case – can you see which? Write a better function which avoids this failure (Touretzky 1990).

7 Recursion

1. Let us start with the factorial function:

$$F(n) = \begin{cases} n = 0, & 1 \\ n \in \mathbb{Z}^+, & n \times F(n-1) \\ \text{otherwise}, & \text{undefined} \end{cases} \quad (1)$$

2. This definition can be directly turned into an algorithm:

function FACTORIAL(n)
  if $n = 0$ **then**
    return 1
  else
    return $n*$ FACTORIAL($n-1$)

3. Here is the factorial function in LISP:

```
(defun factorial (n)
  (if (zerop n)
    1
    (* n (factorial (- n 1)))
    )
  )
```

4. Here is a "safer" version:

```
(defun factorial (n)
  (cond ((or (not (integerp n)) (< n 0)) (error "Factorial undefined!"))
        ((zerop n) 1)
        (t (* n (factorial (- n 1))))
        )
  )
```

5. A custom LENGTH

```
(defun c-length (lst)
  (if (endp lst) 0 (+ 1 (c-length (cdr lst))))
  )
```

6. A custom MEMBER:

```
(defun c-member (item lst)
  (cond ((endp lst) nil)
        ((equal item (car lst)) lst)
        (t (c-member item (cdr lst)))
        )
  )
```

7. A power function:[8]

[8] The corresponding built-in is EXPT; another built-in EXP is a one-place function that raises Euler's constant e to the power of the given argument.

```
(defun power (x y)
  (cond ((zerop y) 1)
        ((= 1 y) x)
        (t (* x (power x (- y 1))))
        )
  )
```

8. Range function, "unsafe" and "safe" versions:

```
(defun range (n)
  (if (zerop n)
    nil
    (append (range (- n 1)) (list (- n 1)))
    )
  )

(defun s-range (n)
  (cond ((or (not (integerp n)) (< n 0)) (error "Range undefined!"))
        ((zerop n) nil)
        (t (append (range (- n 1)) (list (- n 1))))
        )
  )
```

7.1 Recursion with accumulators

1. The examples we saw so far were simple recursions. A slightly more complicated recursive strategy for solving problems is to keep an accumulator (or store) that gets build as we "recurse" down (or up) the problem.

1.1. Writing a recursive length function was straightforward, a more complicated one would keep track of the count of members along recursion and return it when the recursion hits bottom. Let us see a version where the counter is expected to be provided by the user, with the default value of 0:

```
(defun c-length (lst counter)
  (if (endp lst)
    counter
    (c-length (cdr lst) (+ counter 1))
    )
  )
```

1.2. However, it would be more appropriate to have a function that expects only the list, where the counter is kept behind the scenes. One way to do it is to write a second function which is not called by the user, but invoked only by the function that the user interacts with:

```
(defun count-length-user (lst)
  (count-length lst 0)
  )
```

1.3. There is still a better way. `LISP` allows us to define optional parameters to functions, which are not necessary to be provided in function calls. In such cases the parameter gets a default value, if provided, and `NIL` otherwise. Here is how to write a length function with an optional counter parameter.

```
(defun c-length (lst &optional (counter 0))
  (if (endp lst)
    counter
    (c-length (cdr lst) (+ counter 1))
    )
  )
```

2. In the case of length, writing an alternative with a counter was quite unnecessary. Here is a case where an accumulator is desirable for efficiency concerns. We wrote above a function `RANGE` that produces a list starting from 0 up to one less than the given argument. This function is fine for small numbers of input. It is an extremely "low-performance" function, however. Try it with big numbers, say `(range (factorial 8))`; the factorial of 8 is 40320, not a very big number, anyway; try it with `(factorial 9)`, most probably you will not be able to get a result before you run out of resources.

3. The reason `RANGE` performs so poorly is that it uses `APPEND`. Whenever you want `LISP` to append two lists it creates a copy of the first and makes its last cdr point to the cons cell of the second list rather than to `NIL`. For this reason `APPEND` should be avoided if you expect your programs to append long lists and for many times. But what could be the alternative? Here is another range function, `F-RANGE` ('f' for 'fast'), that makes use of a store:

```
(defun f-range (n &optional (store nil))
    (if (zerop n)
      store
      (f-range (- n 1) (cons (- n 1) store))
      )
    )
```

4. We will now see how dramatic is the improvement; try `(f-range (factorial 9))`, you should be able to get a long list of numbers. In order not to wait for the numbers to get printed on the screen, do this:

```
(and (f-range (factorial 9)) t)
```

it should be quite fast to see the `T` on the screen; meaning the range is computed successfully. The function breaks down in factorial 12,[9] hitting the memory limit – no more room for a longer list.[10]

5. `SBCL` allows you to inspect the time and space resources used by your programs by the built-in `TIME`. To see the difference between two versions of the range function run these:

```
(time (and (range (factorial 8)) t))
(time (and (f-range (factorial 8)) t))
```

6. Let us do more recursion exercises; first a custom reverse function operating over lists:

```
(defun c-reverse (lst)
  (if (endp lst)
    nil
    (append (c-reverse (cdr lst)) (list (car lst)))
    )
  )
```

7. Now, let us write the same function – same in the sense of mapping same inputs to same outputs – with an accumulator; and compare their performances.

[9] The actual limit is around 50,100,000, which is larger than factorial 11 = 39,916,800. You can gain around a 4 fold further improvement by invoking `SBCL` with the `--dynamic-space-size 2048` option.

[10] An iterative version would perform identically, therefore the limit is not related to recursion here; but it was, for the former `RANGE`.

```
(defun f-reverse (lst &optional (store nil))
  (if (endp lst)
    store
    (f-reverse (cdr lst) (cons (car lst) store))
    )
  )
```

8 Delayed function calls

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;[11] 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.

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

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,

```
(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. 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 functions 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)) to get (1 2 6 24).

```
(defun mapp (func lst) ; STILL WRONG!
  (if (endp lst)
    nil
    (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:

```
(defun mapp (func lst)
  (if (endp lst)
    nil
    (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))
```

9 Applicative programming

1. Lists are the basic data structures in LISP and in many other programming languages. Once you have your data in the form of a list, then you can do various transformations and/or checks on your data.

2. In most applications, data is read from a file, entered by user, or provided by a stream over a network. We will come to these topics, but for now, we will assume that data is given 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. 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.

4. Having all the grades stored in a list, let us write a very simple function that would compute the sum of them. What about this one?

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

5. 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.

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

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

7. Sometimes, one needs to transform a list into another list with an equal length and with a specific correspondence between the elements of the two lists. This is a mapping. LISP provides the built-in MAPCAR for this purpose. For instance assume you would – for some strange reason – take the square root of the grades, and collect them in a list. LISP has the built-in SQRT that can be applied to numbers. To apply this wholesale on a list, say *grades*, just do:

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

8. There is no limit to the complexity of the functions you can use with MAPCAR. Let us first define a function that gives the letter grade corresponding to the numerical value. You can easily do this by COND. Give it the name LETTER. Doing (mapcar #'letter *grades*) should give you:

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

9. Or, you can define a function APPEND-LETTER, which pairs each grade with the letter grade, returning everything in a list. What you get in the end should look like:

```
((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))
```

10. 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*))
```

11. 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*)
```

12. 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)))))
```

13. The code is inefficient as we compute the very same thing twice. There is a very useful construct to store values that are to be used more than once. The construct is LET and here is its syntax:

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

where <body> is just an ordinary function body.

14. Here is the mean function with LET binding:

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

15. 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)
    (let ((new-value (if (evenp n)
                         (/ n 2)
                         (+ (* n 3) 1))))
      (cons n (collatz-generate new-value)))))
```

16. 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))
```

17. 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.[12]

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

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

19. Check the maximum Collatz length in this range:

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

20. 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:

[12] 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)))))
```

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

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

22. 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).

23. 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)))
```

24. 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))
```

25. 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 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))))
```

26. 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)
```

27. Finally, we would like to know the numbers in a given range that have the same Collatz length. The solution is left as an exercise.

10 Association lists

1. Table lookup, which consists in associating a "key" with a "value", is a basic component of programming. The most straightforward way of this in LISP is to store the key-value pairs in a list. For instance you can keep the grade thresholds for letter grades in such form:

```
(defparameter *letter-table*
    '((AA 90)
      (BA 85) (BB 80)
      (CB 75) (CC 70)
      (DC 65) (DD 60)
      (FD 55) (FF 1)
      (NA 0))
```

2. If you keep your data in such form, then you can search your data on the basis of keys. One way is to use the following LAMBDA function, which, when applied to a letter grade, gives the threshold.

```
(lambda (y) (find-if #'(lambda (x) (equal (car x) y))
    *letter-table*))
```

3. LISP has a build-in function ASSOC for such tasks. In its simplest usage ASSOC takes an object and a list of conses and returns the first cons in the list with the object as its car, if such a cons exists. For instance,

```
(assoc 'CC *letter-table*)
```

would give you (CC 70).

4. Any list of conses is called an **association list** and ASSOC will work for such a list.

5. What happens when you want to search by the second component of the pairs rather than the first. Let's say you want to see whether 70 is a threshold for any letter. One obvious way is to do

```
(assoc 70 (mapcar #'reverse *letter-table*))
```

But always remember that REVERSE is costly and, therefore, needs to be avoided if possible.

6. LISP provides RASSOC for searching by the cdr of each cons in the association list. Here, each cdr is the list of the threshold. Therefore the correct way to search for 70 is:

```
(rassoc '(70) *letter-table* :test #'equal)
```

Here we tell LISP to look at the cdr's in our association list and using EQUAL as the way of testing for matches.

7. In working with association lists, in order not to deal with the extra complication resulting from cdrs being lists themselves, one usually uses dotted pairs, instead of lists. A dotted pair is a cons where the cdr is not a list but an element. For instance (cons 'aa 90) will give you (AA . 90) rather than (AA 90), where the former pairs two symbols and the latter pairs a symbol and a list.

8. Let's keep our letter grade threshold information as a list of dotted pairs instead of lists, we change the name as well:

```
(defparameter *letters*
    '((AA . 90)
      (BA . 85) (BB . 80)
      (CB . 75) (CC . 70)
      (DC . 65) (DD . 60)
      (FD . 55) (FF . 1)
      (NA . 0))
```

Now we can directly do

```
(rassoc 70 *letters*)
```

and get what we want.

9. How can we use *LETTERS* to find the letter grade of a given numerical grade? Obviously, doing (rassoc 83 *letters*) would give NIL. We need to specify a test criterion different from equality. For the present case the right criteria is the "greater than or equal to" relation. If we call RASSOC with this relation as the test parameter, we will find the cons with the correct letter. As this will be a cons, we will need to take out the letter by a car in the end. Here is the LAMBDA function that gives the letter grade for a given numerical grade. Test it and completely understand how it works. One critical thing is that ASSOC and RASSOC return the *first* match they find. We made use of this property in computing the correct letter grade.

```
(lambda (x) (car (rassoc x *letters* :test #'>=)))
```

References

Churchland, P. S. and Sejnowski, T. J. (1992). *The Computational Brain*. MIT Press.

Crane, T. (2003). *The Mechanical Mind*. Routledge, New York.

Pohl, I. and Shaw, A. (1981). *The Nature of Computation: An Introduction to Computer Science*. Computer Science Press, Rockville, Maryland.

Pylyshyn, Z. (1984). *Computation and Cognition: Toward a foundation for Cognitive Science*. MIT Press, Cambridge, MA.

Tanenbaum, A. S. (1999). *Structured Computer Organization*. Prentice Hall, NJ, 4th edition.

Touretzky, D. S. (1990). *COMMON LISP: A Gentle Introduction to Functional Programming*. Benjamin/Cummings Publisihing Co., CA.