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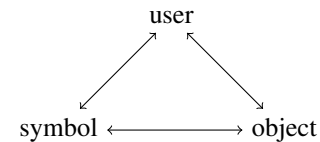
   - A user uses a sign to **refer** to an object.
   - A sign **denotes** an object.
   - A user has certain **intentions** about an object.

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

$$00101101010011110000000010$$

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.
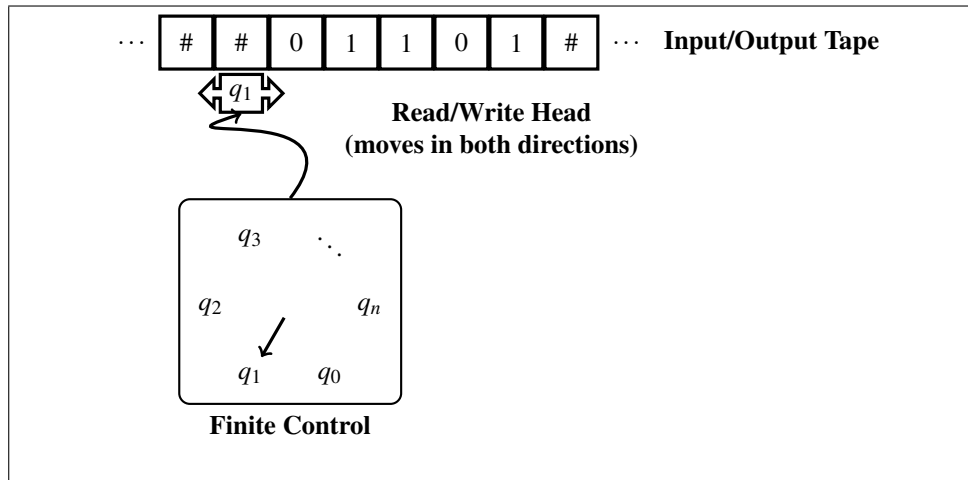
Figure 1: A Turing Machine

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.

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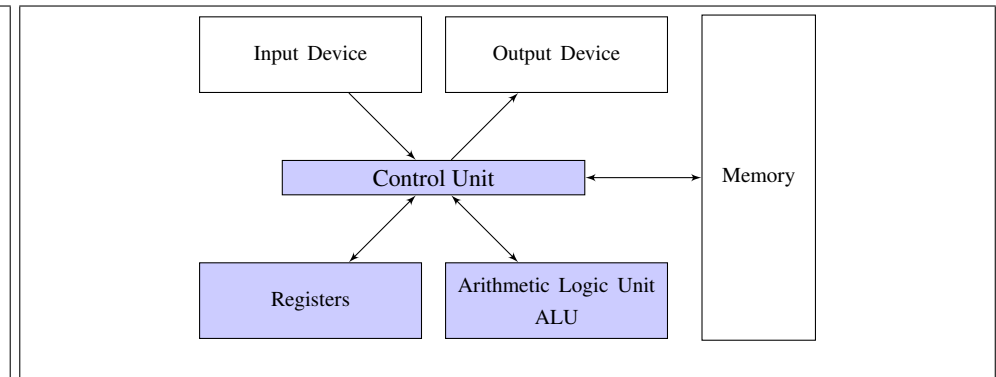\end{array}
$$

$\square$

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):

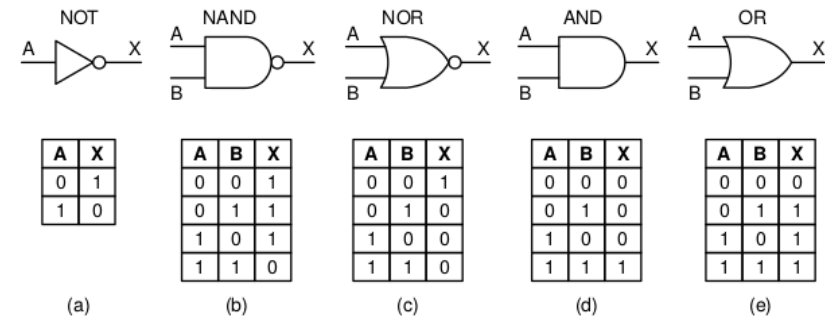[3]Cycles are called 'fetch-decode-execute' or 'instruction cycle'.

```
Loop:   sll $t1,$s3,2
        add $t1,$t1,$s6
        lw  $t0,0($t1)
        bne $t0,$s5,Exit
        add $s3,$s3,1
        j   Loop
Exit:
```
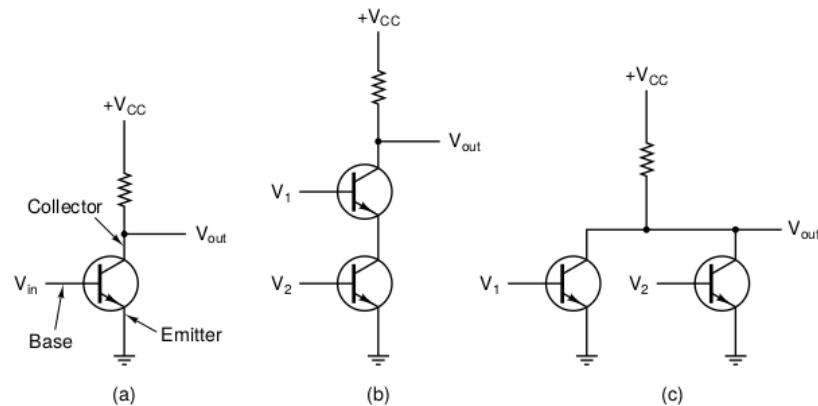
7. A computational process can be specified at various levels.
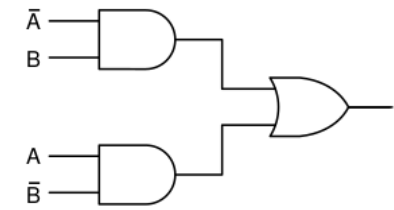

(a) (b) (c) (d) (e)

1.6 More into hardware

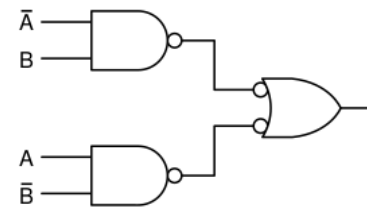1. The basic building block of modern computer is the transistor.[4] Transistors are organized into logic gates.

(a) (b) (c)

(a) (b)

(c) (d)

2. Mechanical addition with circuits:

---

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

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).

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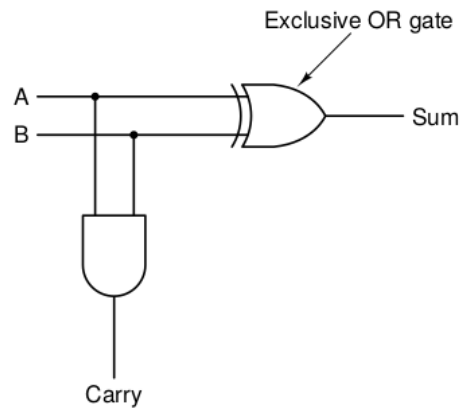
A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(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

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]
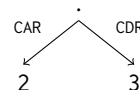

2.2 Lists

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:
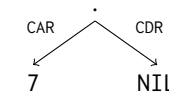
4.2. The two element list (2 TWO) is represented as:
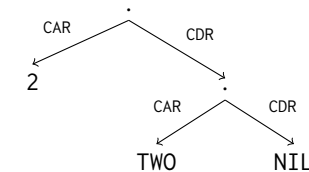
and so on...

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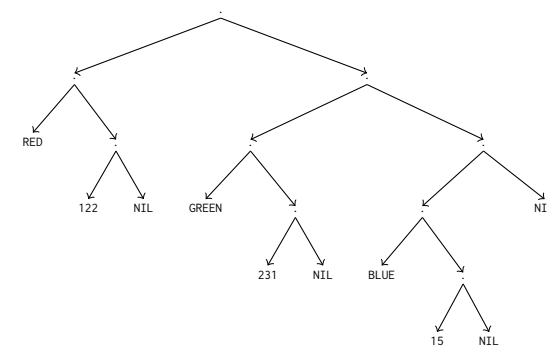
[6]From here on we adopt the convention that the left branch leaving a cons is CAR and the right branch is CDR, to be able to omit the labels on the edges.

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`:

---

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

```
(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:

```
(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:

```lisp
(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](#)).

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.