An introduction to Prolog and wamcompiler

(Content not (yet) checked for grammar and spelling errors)

Vernon Sipple   ed500ac   Marcus   Lucas

Chapter 1

Emacs commands

Emacs is the text editor that Lisp programmers prefer. To give you a head start, this chapter lists the basic commands you need to edit a file.

In the following cheat sheet, `Spc` denotes the space bar, `C-` is the `Ctrl` prefix, `M-` represents the `Alt` prefix and κ can be any key.

Basic commands.   `C-`κ – Press and release `Ctrl` and `κ` simultaneously

> `C-s` then type some text, and press `C-s` again – search a text.
> `C-r` then type some text, and press `C-r` again – reverse search.
> `C-k` – kill a line.
> `C-h` – backspace.
> `C-d` – delete char forward.
> `C-Spc` then move the cursor – select a region.
> `M-w` – save selected region into the kill ring.
> `C-w` – kill region.
> `C-y` – insert killed or saved region.
> `C-g` – cancel minibuffer reading.
> `C-a` – go to beginning of line.
> `C-e` – go to end of line.
> `C-b` – move backward one character.
> `C-f` – move forward one character.
> `C-n` – move cursor to the next line.
> `C-p` – move cursor to the previous line.
> `C-l` – refresh screen.
> `C-u` – undo.
> `C-v` – forward page.
> `INS` – toggle overwrite mode.
> `←` `→` `↑` `↓` – arrows move cursor.

Control-x commands.   `C-x C-`κ – Keep $\boxed{\text{Ctrl}}$ down and press $\boxed{\text{x}}$ and $\boxed{\kappa}$

     `C-x C-f` – open a file into a new buffer.
     `C-x C-w` – write file with new name.
     `C-x C-s` – save file.
     `C-x C-c` – exit Emacs.
     `C-x i` – insert file at the cursor.
     `C-h b` – lists all key strokes.

Window commands.   One can have more than one window on the screen. Below, you will find commands that deal with this situation.

- `C-x` κ – Press and release $\boxed{\text{Ctrl}}$ and $\boxed{\text{x}}$ together, then press $\boxed{\kappa}$

     `C-x b` – next buffer.
     `C-x C-b` – list buffers.
     `C-x k` – kill current buffer.
     `C-x =` – code for char under the cursor.
     `C-x 2` – split window into cursor window and other window.
     `C-x o` – jump to the other window.
     `C-x 1` – close the other window.
     `C-x 0` – close the cursor window.

- `Esc` κ – Press and release the $\boxed{\text{Esc}}$ key, then press the $\boxed{\kappa}$ key.

     `Esc >` – go to the end of buffer.
     `Esc <` – go to the beginning of buffer.
     `Esc f` – word forward.
     `Esc b` – word backward.

- `M-`κ – Keep the $\boxed{\text{Alt}}$ key down and press the $\boxed{\kappa}$ key.

     `M-b` – move backward one word.
     `M-f` – move forward one word.
     `M-g` – go to the line given in the minibuffer.
     `M-n` – activate the next buffer.
     `M-r` – query replace.

Query replace. If you press $\boxed{\text{Esc}}$ $\boxed{\%}$, the computer enters into the query replace mode. First, Emacs prompts for the text snippet S that you want to replace. Then it prompts for the replacement R. When you type the R text and press the $\boxed{\text{Enter}}$ key, the cursor jumps to the first occurence of S, and Emacs asks whether you want to replace it. If the answer is `y`, Emacs will replace S with `R` and jump to the next occurence. If the answer is `n`, Emacs will jump to the next occurrence of S without performing the replacement.

Go to line. When you try to compile code containing errors, the compiler usually reports the line number where the error occurred. If you press `C-u 3` $\boxed{\text{Esc}}$ $\boxed{\text{g}}$ $\boxed{\text{Esc}}$ $\boxed{\text{g}}$, the cursor jumps to the line where the error occurred, for instance, to line 3.

In order to know the position of the cursor, press the `C-x =` command, and Emacs gives information concerning the character under the cursor, the corresponding code, the line, and the character position in the text.

List bindings. If you press the `C-h b` command, the computer lists all Emacs commands. Then you can check whether I forgot one or other keybinding in this tutorial. In order to issue this command, keep the $\boxed{\text{Ctrl}}$ key down and press $\boxed{\text{x}}$, then release both keys and press the $\boxed{?}$ key.

Nia Vardalos. In order to get acquainted with editing and scripting, let us accompany the Canadian programmer Nia Vardalos, while she explores Emacs. Disclaimer: Nia, the coder, is not related to Nia, the actress.

When text is needed for carrying out a command, it is read from the minibuffer, a one line window at the bottom of the screen. For instance, if Nia presses `C-s` for finding a text, the text is read from the minibuffer, and while Nia is still typing, Emacs starts the search. When she finds what she is looking for, Nia strikes the $\boxed{\text{Enter}}$ key to stop the interactive search. Nia can press `C-s` again to find other instances of the text.

To transport a region from one place to another, Nia presses `C-Spc` to start the selection process and move the cursor to select a region. Then she presses `C-w` to kill her selection. Finally, she moves the cursor to the insertion place, and presses the `C-y` shortcut. To copy a region, Nia presses `C-Spc` and moves the cursor to select the region. Then she presses `M-w` to save the selection into the kill ring. Finally, she takes the cursor to the destination, where the copy is to be inserted and issues the `C-y` command.

Nia noticed that there are two equivalent ways to issue an `M-`κ command. She can press and release the $\boxed{\text{Esc}}$ key, then strike the κ key. Alternatively, she can keep the $\boxed{\text{Alt}}$ key down and press the $\boxed{\kappa}$ key.

Calculations with Lisp. Emacs offers a Lisp dialect to perform calculations and text processing. Let us assume that you want to find out how many students graduate from medical schools in California. You type the Lisp addition command, then press `C-x C-e` with the cursor in front of it:

> (+ 190 180 170 160 120 100 100 90)`C-x C-e`

In the above expression, the first thing that follows the open parenthesis is the + sum operation identifier. After the name of the operation, there is a list of the arguments to be added together. Emacs shows the result of the addition as soon as Nia presses the `C-x C-e` command:

    1110

The rule that worked for the (+ x_1 x_2 ...) sum operation also works for the other arithmetic function too.

- Multiplication is performed by the ∗ operation.

  > (∗ 1 2 3 4 5)`C-x C-e`
  120

- Division is obtained through the / operation.

  > (∗ 1 2 4)`C-x C-e`
  0.125

  The above operation performed the chain division 1/2/3.

- Subtraction has the syntax shown below.

  > (- 1 2 4)`C-x C-e`
  -5

- Mixed operations. One can nest subexpressions within an outer expression. For instance, to get the average student number graduating from medical school in California, one can calculate the expression:

  > (/ (+ 190 180 170 160 120 100 100 90) 8)`C-x C-e`

Command execution. When you press shortcut keys, Emacs calls a command written in C or in Lisp. You can get a complete list of all Emacs commands by pressing the `C-x ?` shortcut.

There are commands that are not associated to shortcut keys. For instance, the `lisp-mode` function tells Emacs that you are editing Lisp code. In order to issue such a command, keep the `Alt` key down and press the `x` key. The `M-x` prompt will be placed on the minibuffer. Type `lisp-mode` in front of the prompt:

```
M-x lisp-mode
```

When typing a command at the `M-x` prompt, you often don't remember the complete name of the operation. In this case, type press the `Tab` key, and Lisp will list the options available to complete the name.

```
backspace          backspace
C-a                move-beginning-of-line
C-b                backward-char
C-d                delete-char
C-e                move-end-of-line
C-f                forward-char
C-h                help-command
C-k                kill-line
C-l                recenter-top-bottom
C-n                next-line
C-p                previous-line
C-r                isearch-backward
C-space            set-mark
C-s                isearch-forward
C-/                undo
C-v                scroll-up-command
C-w                kill-region
C-x 1              delete-other-windows
C-x 2              split-window-below
C-x C-b            list-buffers
C-x C-c            save-buffers-kill-terminal
C-x C-f            find-file
C-x C-s            save-buffer
C-x =              cursor-position
C-x C-w            write-file
C-x i              insert-file
C-x k              kill-buffer
```

```
C-x o           other-window
C-y             yank
down            next-line
esc a           apropos
esc B           backward-word
esc b           backward-word
esc <           beginning-of-buffer
esc d           kill-line
esc >           end-of-buffer
esc esc         show-version
esc F           forward-word
esc f           forward-word
esc G           goto-line
esc g           goto-line
esc i           yank
esc k           kill-region
esc l           list-bindings
esc m           set-mark
esc n           next-buffer
esc o           delete-other-windows
right           forward-character
up              previous-line
```

Chapter 2

Installation

In order to program in Prolog, you will need to install three programs:

1. Emacs – a text editor

2. Steel Bank Common Lisp, or sbcl for short

3. A Prolog compiler that works in tandem with Lisp, such as `cl-prolog` that links Lisp with a popular implementation of Prolog, or a Lisp based Prolog, such as the `wamcompiler.lisp` virtual machine.

However, you must learn many things before actually being able to make these three pieces of software work together. Unless you are well versed in programming, I suggest that you call a geek who has majored in Computer Science to perform the installations for you.

Installation of Emacs is quite easy. You will find binary distributions for practically any machine. Even so, you will need help to configure the editor and perform the setup of useful plug-ins, such as slime, which will help you with Lisp programming.

Installation of sbcl is slightly more difficult. The best approach is to install roswell, which is a Lisp implementation installer/manager and launcher. Search the Internet for the roswell page and follow the instructions with the help of the Computer Science major. Use the keywords *Common Lisp roswell* to find the github repository for roswell.

Finally you will need to download the `wamcompiler.lisp` file. The `wamcompiler` was developed by Daiki Matsunaga, who publish under the nickname of `matsud224`. Therefore, a search with the following keywords should land on the wamcompiler page: `matsud224 wamcompiler`.

Besides Prolog, this book will deal with Common Lisp and shell script, thus if you go through the whole text, you will not need help from the Computer Science major pretty soon.

2.1   Ready

If you installed Emacs and sbcl, then performed the download of the wam-compiler, you are now ready for action!

It seems that people prefer money to sex. After all, almost everybody says no to sex on one occasion or another. But I have never seen a single person refusing money. Therefore, let us start this tutorial talking about money.

If one wants to calculate a running total of bank deposits, she will make a column of numbers and perform the addition.

What I mean to say is that, if you need to perform the addition $8 + 26 + 85 + 3$ with pencil and paper, you will probably stack the numbers the same way you did before taking pre-algebra classes in highschool:

$$+ \quad \begin{array}{r} 8 \\ 26 \\ 85 \\ 3 \\ \hline 122 \end{array}$$

Subtraction is not treated differently:

$$- \quad \begin{array}{r} 358 \\ 216 \\ \hline 142 \end{array}$$

This chapter contains an introduction to the Cambridge prefix notation, which is only

Fig. 2.1: Running total

slightly different from that learned in elementary school: The operation and its arguments are put between parentheses. In doing so, one does not need to draw a line under the bottom number, as you can see below:

$$(+ \quad \begin{array}{r} 8 \\ 26 \\ 85 \\ 3) \\ 122 \end{array}$$

The Cambridge prefix notation can be applied to any operation, not only to the four arithmetic functions.

2.2 Cambridge prefix notation

Let us summarize what we have learned until now. In pre-algebra, students learn to place arithmetic operators (+, -, × and ÷) between their operands; e.g. 347+45. However, when doing additions and subtractions on paper, they stack the operands.
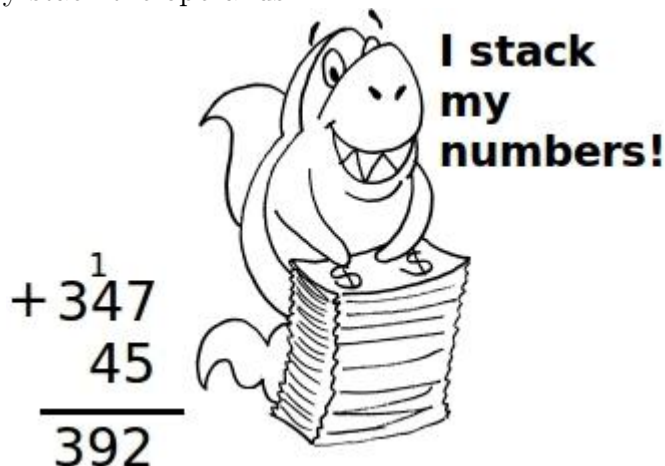
Lisp programmers put operation and operands between parentheses. The right parenthesis separate the operands from the result, instead of drawing a line under the last operand.

2.3 Pictures

Here is the story of a Texan who went on vacation to a beach in Mexico. While he was freely dallying with the local beauties, unbeknowest to him a blackmailer took some rather incriminating photos. After a week long gallivanting, the Texan returns to his ranch in a small town near Austin. Arriving at his door shortly after is the blackmailer full of bad intentions.

Unaware of any malice, the Texan allows the so called photographer to enter and sit in front of his desk. Without delay, the blackmailer spread out

a number of photos on the desk, along with his demands: "For the photo in front of the hotel, that will cost you \$ 25320.00. Well, the one on the beach that's got to be \$ 56750.00. Finally, this definitively I can't let go for less than \$ 136000.00."

Once finished with his presentation, the blackmailer snaps up the photos, and looks to the Texan with a sinister grin, awaiting his reply.

Delighted with the selection of pics, the Texan in an elated voice says: "I thought I would have no recollection of my wonderful time. I want 3 copies of the hotel shot, half a dozen of the beach. And the last one, I need two copies for myself, and please, send one to my ex-wife. Make sure you leave me your contact details; I might need some more.

Mixed calculations.   In order to calculate how much the Texan must pay his supposed blackmailer, his bookkeeper needs to perform the following operations:

$$3 \times 25320 + 6 \times 56750 + 2 \times 136000 + 136000$$

It should be remembered that multiplications within an expression take priority over additions and subtractions. The bookkeeper must therefore calculate the first two products 3×25320 and 6×56750, before performing the first addition. In the Cambridge prefix notation, the internal parentheses pass priority over to the multiplications.

The Texan's bookkeeper started Emacs in order to perform the calculations. The text editor creates a memory buffer that mirrors the file contents. By convention, the file and the buffer have the same name.

Initially, Emacs will place the bookkeeper on a *scratch* buffer. The bookkeeper issues the `M-x` command by keeping the Alt key down and pressing the `x` key. Emacs will put the bookkeeper in the minibuffer with the `M-x` prompt. Then, the bookkeeper will type the `shell` command, which we will study in one of the future chapters. Here is how to start the shell:

`M-x shell` Enter

The above command will create a kind of chat box between the bookkeeper and the computer. The bookkeeper types the `ros run` command to start Lisp. Listing 2.2 shows what an interaction with shell and sbcl looks like. After typing an expression, while obeying the Cambridge prefix notation syntax, the bookkeeper moves the cursor to the front of the most external right parenthesis. To perform the calculation, she presses the Enter key. Then, the sbcl Read Eval Print Loop transforms the command to machine language, evaluates it, and inserts the result into the buffer.

To save the buffer, the bookkeeper keeps the [Ctrl] key down, and presses the [x] and [s] keys in sequence. To exit the editor, she maintains the [Ctrl] key pressed and hits the [x] and [c] keys one after the other. If there are unsaved files, the editor warns that modified buffers exist, and requires confirmation before quitting. But wait, if you are shadowing the actions of the bookkeeper, don't leave the editor yet. Let us test the wamcompiler before doing so.

```
~/wamcompiler/wamdocs master ×
> ros run
* (+  (* 3 25320)
      (* 6 56750)
      (* 2 136000)
    136000 )[Enter]
824460
```

Listing 2.2: Chat with Lisp

The Texan's accountant decided to check whether the wamcompiler is working on her machine. Therefore, she loads the `wamcompiler.lisp` file and starts a Prolog Read Eval Print Loop, or repl for short.

```
* (load "~/wamcompiler/wamcompiler.lisp")
(load "~/wamcompiler/wamcompiler.lisp")
T
* (repl)
(repl)
> ?- X is 3*25320 + 6*56750+ 2* 136000 + 136000.
?- X is 3*25320 + 6*56750+ 2* 136000 + 136000.
X = 824460

yes.
> ?- halt.

yes.
```

The Lisp prompt is usually an asterisk, therefore be careful not to confuse the prompt with the multiplication symbol. The wamcompiler prompt is the *greater-than* > character, but you need to prefix all queries with the ?- symbol, as you can see in the example.

2.4   Time value of money

Suppose that you wanted to buy a $ 100,000 red Ferrari, and the forecourt salesperson in his eagerness to close a deal gives you the following two payment options:

- Pay $ 100,000 now **or**

- pay the full $ 100,000 after a three year grace period.

I am sure that you would choose to pay the $ 100,000 after three years of grace has finished, although you have the money in a savings account waiting for a business opportunity. But why is this? After all, you will need to pay the debt one way or the other. However, if you keep the money in your power during the grace period, you can earn a few months of fuel from the interest.

You may not know for sure how much interest you will earn in three years, but since the salesperson is not charging you for deferring the payment, whatever you gain is yours to keep.

Unfortunately for you, the above scenario would more than probably not happen in real life. The right to delay payment until some future date is a merchandise with a price tag, which *is called interest by those who think it lawful, and usury by those who do not* (William Blackstone's Commentaries on the Laws of England). Therefore, unless the salesperson is your favorite aunt, the actual offer is like jumping into a tank full of sharks as in the classic James Bond films. It requires a little forethought and understanding of how interest works, before making a decision. Here is a more realistic real estate sales offer:

- $ 100,000 now **or**

- $ 115,000 at the end of three years.

What to do when facing an increase in price to cover postponement of payment? The best policy is to ask your banker how much interest she is willing to pay you over your granted grace period.

Since the economy performance is far from spectacular, your banker offers you an interest rate of 2.5%, compound annually. She explains that compound interest arises when interest is paid on both the principal and also on any interest from past years.

2.5 Future value

The value of money changes with time. Therefore, the longer you keep control of your money, the higher its value becomes, as it can earn interest. Time Value of Money, or TVM for short, is a concept that conveys the idea that money available now is worth more than the same amount in the future.

If you have $ 100,000.00 in a savings account now, that amount is called *present value*, since it is what your investment would give you, if you were to spend it today.

Future value of an investment is the amount you have today plus the interest that your investment will bring at the end of a specified period.

The relationship between the present value and the future value is given by the following expression:

$$FV = PV \times (1+i)^n \tag{2.1}$$

where FV is the future value, PV is the present value, i is the interest rate divided by 100, and n is the number of periods.

In the case of postponing the payment of a $ 100,000.00 car for 3 years, at an interest rate of 0.025, the future value of the money would be 107,689.06; therefore, I strongly recommend against postponing the payment.

2.6 Compound interest

Our Texan decides he needs a break. Thus he walks into a New York City bank and asks for the loan officer. He tells a story of how through his doctor's recommendation he was taking it easy at his property in the south of France for two whole years and for such an emergency he needs a $ 10,000.00 loan.

The loan officer said that the interest was a compound 8% a year, but the bank would need some collateral for the loan.

"Well, I have a 60 year old car that I like very much. Of course, I cannot take it with me to France. Would you accept it as collateral?"

Unsure whether or not the old car was worth the amount of the loan, the officer summons the bank manager. The manager inspects the vehicle that was parked on the street in front of the bank. After a close examination, he gives a nod of approval: "It's a Tucker Torpedo. Give him the loan."

The Texan quite willingly signed his heirloom over. An employee then drove the old car into the bank's underground garage and parked it. From time to time, the employee would go, and turn over the engine, to keep the car in good running condition, and gave it an occasional waxing just to maintain it in pristine condition. Two years later the Texan returned, and asked how much he owed the bank. The loan officer started Emacs and Lisp, and calculated the total debt as $ 11,664.00.

Interest Calculation.   Let us follow the calculation of the value accumulated in the first year of compound interest at an 8% rate over $ 10,000.00. Enter formula 2.1 in the Lisp Read Eval Print Loop:

```
* (* (expt (+ 1.0 0.08) 2) 10000)
11644.00
```

Observe that it was necessary to divide the interest rate by 100, which produces 0.08.

```
;; File: fvalue.lisp

(defun  future-value(pv i n)
   (* (expt (+ (/ i 100.0) 1) n)
      pv)
);;end defun
```
```
* (load "fvalue.lisp")[Enter]
T
* (future-value 10000 8 2)
11664.001
```

Listing 2.3: Future Value Program

It is possible to define an operation that calculates formula 2.1 given the arguments `fv`, `i` and `n`. The definition is given in listing 2.3.

After typing the definition shown in listing 2.3, you must issue the `C-x C-s` command to save the buffer. In order to do this, keep the [Ctrl] key pressed down and hit the [x] and [s] keys in sequence.

In order to load the program into Lisp, keep the [Alt] down and press the
x key. The prompt M-x will appear in the minibuffer; type shell at this
prompt, as shown below.

```
;; File: fvalue.lisp

(defun  future-value(pv i n)
    (* (expt (+ (/ i 100.0) 1) n)
       pv)
);;end defun
```
```
M-x shell[Enter]
```

When the shell chat starts, run the `ros run` command to start the Common
Lisp compiler:

```
> ros run
* (load "fvalue.lisp")
T
* (future-value 10000 8 2)
11664.001y
```

In listing 2.3, any text between a semicolon and the end of a line is
considered a comment. Therefore, `;; File: fvalue.lisp` is a comment.
Likewise, `;;end defun` is a comment. Comments are ignored by Lisp, and
have the simple function of helping users and programmers.

Now, let us see what Prolog adds to Common Lisp, I mean, what Lisp
programmers will gain by having a Prolog compiler embedded into Common
Lisp. Here one can see a Prolog program for calculating future values:

```
%% File: ?- consult('fvalue.pl')

period_of_time(2).
period_of_time(4).
period_of_time(8).
period_of_time(10).

futval(PV, I, N, FV) :- period_of_time(N),
                FV is PV*(1+I/100)**N.
```

In Prolog, comments are introduced by the % percent symbol, not by a
semicolon, as in Lisp. Variables are capitalized; thus, in the definition of
`futval/4`, the symbols PV, I, N, and FV are variables. Below you will see

an example of how to compile and execute the `'fvalue.pl'` program. The
predicate `period_of_time/1` has four clauses. When this predicate is called
from `futval/4` with the `period_of_time(N)` pattern, it unifies `N` with the
first pattern, which makes `N= 2`. To make a long story short, there are many
options for the period of time `N`, and Prolog starts with the first choice.

The `FV` future value is calculated for this choice of `N`, and the result is
shown to the user. If the first result is satisfactory, the user types `yes`, and
Prolog stops searching for solutions. If the user presses the semicolon key,
Prolog backtracks to the `period_of_time(N)` predicate, and chooses `N=4`,
which is the second option. By repeatedly pressing the semicolon, the user
is able to obtain all solutions.

```
> rlwrap ros run
* (load "wamcompiler.lisp")
T
* (repl)
> ?- consult('fvalue.pl').

yes.
> ?- futval(10000, 8, N, FV).
N = 2
FV = 11664.001
?;
N = 4
FV = 13604.892
?;
N = 8
FV = 18509.309
?;
N = 10
FV = 21589.258

yes.
```

2.7   Mortgage

A man with horns, legs and tail of a goat, thick beard, snub nose and pointed
ears entered a real estate agency, and expressed his intention of closing a deal.
People fled in all directions, thinking that Satan himself was paying them a
visit. Deardry seemed to be the only person who remained calm. "What a

ignorant, narrow minded, prejudiced lot! I know you are not the devil. You are the Great God Pan! What can I do for you?"

"I would like to buy a house in London. I know that the city is made up of 32 Boroughs and where I buy will make an enormous difference to price, quality of life, and chances of increase in capital value of the property."

"Well, Sir, with our experience you can rest assured that you will secure your ideal property. Firstly we must decide on what type of property fits your needs and where you want to be. You may consider somewhere like Fulham, Chelsea, Knightsbridge, Kensington or Mayfair. I have properties in all these places. "

"Chelsea! I like the name."

"Chelsea is arguably one of the best residential areas in London. It has benefited from it's close proximity to the west end of the city and is highly sought after by overseas buyers looking to be located in one of the most popular areas in central London. At the moment, I can offer you a fantastic living space in a four bedroomed flat situated behind King's Road. The price is £10,500,000."

"I don't have this kind of cash with me right now, but can get it in a few months of working in a circus. Meanwhile, can you arrange a mortgage for me? "

"Yes, Yes I can help arrange mortgages in all 32 Boroughs of London. Non-residents can have mortgages up to 70% of the purchase price. Do you have £3,150,000 pounds for the down payment? In mortgage agreements, down payment is the difference between the purchase price of a property and the mortgage loan amount."

"I know what down payment is. And yes, I can dispose of £3,150,000 pounds."

"A mortgage insurance is required for borrowers with a down payment of less than 20% of the home's purchase price. That is not your case. Therefore, the balance of the purchase price after the down payment is deducted is the amount of your mortgage. Let us write a program in Lisp to find out your estimated monthly payment. The loan amount is £7,350,000 pounds. The interest rate is 10% a year. The length of the mortgage is 20 years. That is the best I can do for you, I am afraid. You are going to pay £97131.00 pounds a month for ten years. After that, the balance of your debt will be zero. But your visit is a surprise!" The broker exclaimed. " I never thought I would ever see a true living god wanting to do property deals in London."

With that, Pan replied: "If property prices were not so high, and interest rate so steep, we would definitely be up for business more often."

Loan Amortization.   Amortization refers to the gradual reduction of the loan principal through periodic payments.

Suppose you obtained a 20-year mortgage for a $ 100,000.00 principal at the interest rate of 6% a year. Calculate the monthly payments.

Amortization Formula.   Let p be the present value, n the number of periods, and r be the interest rate, i.e., the interest divided by 100. In this case, the monthly payment for full amortization is given by 2.2. This formula is implemented in listing 2.4.

$$\frac{r \times p \times (1+r)^n}{(1+r)^n - 1} \tag{2.2}$$

```
;; File: (load "pmt.lisp")

(defun pmt(
        p ;; present value
        i ;; interest
        n ;; number of periods
        ;; optional parameters
        &optional  (r (/ i 100.0))
            (rn (expt (+ 1.0 r) n))
    );;end of parameter list
(/ (* r p rn)
 (- rn 1))
);;end of defun
```
```
(load "pmt.list") Enter
T
(pmt (* 10500000.00 0.7) (/ 10.0 12) (* 20 12)) Enter
70929.00
```

Listing 2.4: Mortgage for a God

You certainly noticed that the pmt function, defined in listing 2.4, has two parameters r and rn, which the user does not need to express. Lisp itself assigns (/ i 100.0) to r and (expt (+ 1.0 r) n) to rn. Now, let us repeat the magic in Prolog. Here is the program:

```
%% File: ?- consult("pmt.lisp").

property(fulham, 6500000).
property(chelsea, 10500000).
property(mayfair, 9000000).
property(kensington, 4000000).

pmt(Local, Price, Down, I, NY, Payment) :-
    property(Local, Price),
    P is 0.7 * Price,
    Down is 0.3 * Price,
    R is I/1200,
    N is NY * 12,
    RN is (1 + R)** N,
    Payment is R * P * RN / (RN - 1).
```

The Prolog program will consult a data base of property prices, calculate the down payment and the monthly payments needed to be made, and present the values to the appreciation of the client. By pressing the semicolon key, the client can refuse an offer and ask to see another option.

```
> rlwrap ros run
* (load "wamcompiler.lisp")
T
* (repl)
> ?- consult('pmt.pl').

yes.
> ?- pmt(Local, Price, Down, 10, 20, Payment).
Local = fulham
Price = 6500000
Down = 1950000.1
Payment = 43908.496
?;
Local = chelsea
Price = 10500000
Down = 3150000.3
Payment = 70929.12
?
```

2.8   Functions

I am sure that you know what a function is. It is possible that you do not know the mathematical definition of function, but you have a feeling for functions, acquired by the use of calculators and computer programs or by attending one of those Pre-Algebra courses.

A function has a functor, which is the name of the function, and arguments,e.g., `S is sin(X)` is a function. Another function is `M is mod(X,Y)` that gives the remainder of the division of `X` by `Y`. When you want to use a function, you substitute constant values for the variables, or arguments. For instance, if you want to find the remainder of 13 divided by 2, you can type

```
?- X is mod(13,2).
```

in the Prolog read eval print loop to get the answer. One could say that functions are maps between possible values for the argument and a value in the set of calculation results.

The domain is the set of possible values for the argument, whereas the image is the set of calculation results. In the case of the sinus function, the elements of the domain are real numbers between $-\pi/2$ and $\pi/2$ radians. There is an important thing to remember: Mathematicians insist that a function must produce only one value for a given argument. Therefore, if a calculation produces more than one value, it is not a function. For instance, $\sqrt{4}$ can be 2 or -2. Then the square root of a number cannot be a function.

What about functions of many arguments? For example, the function `max(X,Y)` takes two arguments, and returns the greatest one. In this case, you can consider that it has only one argument, that being a tuple of values. Thus, the argument of `max(5,2)` is the `(5,2)` pair. Mathematicians say that the argument of such a function is the Cartesian product $R \times R$. The Cartesian product was named after René Descartes, a French philosopher and scientist. In Latin, the family name of Descartes translates as Cartesius, therefore, the adjective Cartesian.

There are functions whose functor is placed between its arguments. This is the case of the arithmetic operations, where one often writes `5+7`, instead of `+(5,7)`. At this point, it is necessary to praise Lisp, since the designers of this elegant language chose a uniform syntax for all operations, including application of functions. In Lisp, the `sin(X)` function is represented by `(sin X)`, and addition is notated as `(+ X Y Z ...)`. In other words, all well formed formulas in Lisp are lists, where the operator is the first element.

2.9 Predicates

Predicates are functions whose domain is the set `{true,false}`. There are a few predicates that are well known to any person who tried his/her hand at programming, or even by a student who is taking Pre-Algebra:

- $X > Y$ is true if X is greater than Y, false otherwise.

- $X < Y$ is true if X is less than Y, false otherwise. 8

- $X = Y$ is true if X is equal to Y, false otherwise.

A predicate with one argument tells of an attribute or feature of its argument. One can tell that this predicate acts like an adjective. In Lisp, `(not X)` is true if `X` is false, and `false` otherwise. In other computer languages there are predicates equivalent to this one.

In the story of the Texan, you have seen an example of a one slot predicate:

```
period_of_time(2).
period_of_time(4).
period_of_time(8).
period_of_time(10).
```

A predicate with more than one argument shows that there exists a relationship between its arguments. In the case of X=Y, the relationship is *equal*.

Prolog predicates have a feature that one cannot see in other computer languages: they not only check whether constant arguments have a feature or obey a relationship, but they actively search for values that satisfy a set of requisites. For example, the futval(PV, 8, N, FV) searches for values of N and FV that satisfy the definition of a future value.

```
> rlwrap ros run
* (load "wamcompiler.lisp")
T
* (repl)
> ?- consult('fvalue.pl').

yes.
> ?- futval(10000, 8, N, FV).
N = 2
FV = 11664.001
?;
N = 4
FV = 13604.892
?;
N = 8
FV = 18509.309
?;
N = 10
FV = 21589.258

yes.
```

2.10 Lists

Every human and computer language has syntax, i.e., rules for combining components of sentences. In general, a computer language syntax is complex, and the programmer must study it for years before becoming proficient.

Lisp has the simplest syntax of any language. All Lisp constructions are expressed in Cambridge prefix notation: ($\mathtt{operation}$ x_1 x_2 $x_3 \ldots x_n$). This means that all Lisp procedures can be written as a pair of parentheses enclosing an operation followed by its arguments.

A linked list is a representation for a $\mathtt{xs}$ sequence in such a way that it is easy and efficient to push a new object to the top of $\mathtt{xs}$.

```
> (defparameter aList '(S P Q R))
(S P Q R)
> (cons 'Rome aList)
(Rome S P Q R)
> aList
(S P Q R)
```

The '(S P Q R) list of letters is prefixed by a quotation mark because in Lisp lists and programs have the same syntax: The Cambridge prefix notation. The single quotation mark tags the list so the computer will take it at face-value, and thus will not try to be evaluated.

Through the repeated application of the $\mathtt{cons}$ function, one can build lists of any length by adding new elements to a core.

```
> (defparameter xs '(S P Q R))
(S P Q R)
> (cons 5 (cons 4 (cons 3 (cons 2 (cons 1 xs)) ) ))
(5 4 3 2 1 S P Q R)
> xs
(S P Q R)
```

In the above example, you have noted that repeated application of nested functions cause right parentheses accumulate at the tail of the expression. It is good practice in programming to group these bunches of right parentheses in easily distinguishable patterns.

```
> (cons 'S (cons 'P (cons 'Q '(R)) ))  ;4 right parentheses
(1 2 3)
> (+ 51 (* 9 (+ 2 (- 3 (+ 1 2 3)) ) )) ;5 right parentheses
42
> (+ 6 (* 9 (+ 2 (- 3 (+ 4 (- 3))) ))) ;6 right parentheses
42
```

Another way to create easily recognizable patterns is to reorganize the expression in order to interrupt the sequence of right parentheses:

```
> (+ 51 (* (+ 2 (- 3 (+ 1 2 3))) 9))
42
```

The head of a list is its first element. For instance, the head of `'(S P Q R)` is the `'S` element. The tail is the sublist that comes after the head. In the case of `'(S P Q R)`, the tail is `'(P Q R)`. Lisp has two functions, `(car xs)` and `(cdr xs)` that selects the head and the tail respectively.

As one can see below, it is possible to reach any element of a list with a sequence of `car` and `cdr`.

```
> (defparameter xs '(2 3 4 5 6.0))
(2 3 4 5 6.0)
> xs
(2 3 4 5 6.0)
> (car xs)
2
> (cdr xs)
(3 4 5 6.0)
> (car (cdr xs))
3
> (cdr (cdr xs))
(4 5 6.0)
> (car (cdr (cdr xs)))
4
> (car (cdr (cdr (cdr xs)) ))
5
> (car (cdr (cdr (cdr (cdr xs)) ) ))
6.0
```

2.11   Going through a list

In listing 2.5, the `avg` function can be used to calculate the average of a list of numbers. It reaches all elements of the list through successive applications of `(cdr s)`. The list elements are accumulated in `acc`, while the `n` parameter counts the number of `(cdr s)` applications. When `s` becomes empty, the `acc` accumulator contains the sum of `s`, and `n` contains the number of elements. The average is given by `(/ acc n)`.

In order to fully understand the workings of the `avg` function, we should have waited for the introduction to the `cond-form` given on page 77. The only

reason for presenting a complex definition like `avg` so early in this tutorial is to show a grouping pattern of five right parentheses. As was discussed previously, when the number of close parentheses is greater than 3, good programmers distribute these into small groups, so that an individual who is trying to understand the program can see that the expression is proprely closed at a glance.

For the impatient learner. However, for the sake of the impatient reader, let us give a preview on how the cond-form works.

```
(defun (avg s &optional (acc 0) (n 0))
  (cond ( (and (null s) (= n 0)) 0)
        ( (null s) (/ acc n) )
        (T (avg (cdr s)
                (+ (car s) acc)
                (+ n 1.0)) ) ))
```
```
(load "average.lisp") Enter
T
(avg '(3 4 5 6)) Enter
4.5
```

Listing 2.5: Going through a list

The cond-form consists of a sequence of clauses, where each clause has two components: A condition and an expression. The value of the cond-form will be given by the first clause that has a true condition. In listing 2.5, the clauses are:

1. `( (and (null s) (= n 0)) 0)` – The condition will be true when `s` is the empty `()` list and `n=0`. This happens when the user types `(avg ())`. In such a case, the cond-form returns 0.

2. `( (null s) (/ acc n) )` – This clause will be activated when the `s` list is `()` empty, but `n` is greater than 0. If `n` were 0, the first clause would prevent the evaluator from reaching the second clause. The value of the second clause is `(/ acc n)`, which is the average of the list.

3. `(T (avg (cdr s) (+ (car s) acc) (+ n 1.0)) )` – If the user executes the `(avg '(4 5 6))` expression, the evaluator will visit this clause with `s= (4 5 6)`, `s= (5 6)` and `s= (6)`. Every time the evaluation falls into the third clause, `(cdr s)` is executed, and `s` looses an element, until it becomes empty and is captured by the second clause, which produces the answer.

2.12   Prolog Lists

In his book about Lisp, Patrick Winston says: *A list is an ordered sequence of elements, where ordered means that the order matters.* In Prolog, a list is put between brackets, and is supposed to have a head and a tail.

List	Head	Tail
`[3,4,5,6,7]`	`3`	`[4,5,6,7]`
`[wo, ni, ta]`	`wo`	`[ni, ta]`
`[4]`	`4`	`[]`
`[3.4, 5.6, 2.3]`	`3.4`	`[5.6, 2.3]`

You can match a pattern of variables with a list. For instance, if you match

```
[X|Xs]
```

with the list `[42,666,13]`, you will get `X=42` and `Xs=[666,13]`, i.e., `X` matches the head of the list, and `Xs` matches the tail. Of course, you can use other variables instead of `X` and `Xs` in the pattern `[X|Xs]`. Thus, `[A|B]`, `[X|L]`, `[First|Rest]` and `[P|Q]` are equivalent patterns.

Pattern	List	X	Xs
`[X|Xs]`	`[42,666,13]`	`42`	`[666,13]`
`[X|Xs]`	`[666,13]`	`666`	`[13]`
`[X|Xs]`	`[13]`	`13`	`[]`
`[X|Xs]`	`[]`	no match	no match

Listing 2.6 shows a Prolog program that calculates the average of a list. A Prolog definition is a set of clauses, not very different from the cond-form.

As you learned previously, when Prolog finds a solution through one of the clauses, it offers the user another solution, obtained by the use of the next clause in the definition. Computer scientists say that Prolog backtracks to the point of choice, in order to find multiple solutions. However, in the definition of `avg/4`, I know that if Prolog finds a solution through the second clause, it will be useless to try the third clause, since the empty `[]` list will never unify with the `[X|Xs]` pattern of the third clause. Therefore, I cut the search tree by placing a exclamation mark in the first and in the second clause. This is equivalent to a declaration that `avg/4` has only one solution.

Whenever Prolog finds the exclamation mark, it interrupt the search for new solutions. The exclamation mark is called *cut* by computer scientists. Since mathematical functions have only one solution, the definition of a function often requires a cut.

Let us test the predicate `avg/4` given in listing 2.6. By the way, Prolog is not very efficient in numerical computations. Since we have Common Lisp available for number crunching, there is no meaning in using Prolog for arithmetic calculations.

```
> rlwrap ros run
* (load "wamcompiler.lisp")
T
* (repl)
> ?- consult('average.pl').

yes.
> ?- avg([3,4,5,6], 0, 0, Average).
Average = 4.5

yes.
```

```
%% ?- consult('average.pl').

avg([], Acc, 0, 0) :- !.
avg([], Acc, N, A) :- N > 0, !, A is Acc/N.
avg([X|Xs], Acc, N, A) :- S is Acc + X,
          N1 is N + 1, avg(Xs, S, N1, A).
```

Listing 2.6: Going through a list in Prolog

2.13 How to solve it in Prolog

The title of this section is a homage to Helder Coelho, the first author of the best book on Prolog ever written. This book was printed by the National Laboratory of Civil Engineering, where Helder Coelho works in Portugal.

Coelho's book is a collection of short problems, proposed and solved by great programmers and computer scientists. The whole thing is organized as a kind of FAQ. The problems are interesting, and the solutions are illuminating. The more important, the book provides a lot of fun.

PROBLEM 1.   Verbal statement: Check whether H is a member of a list L; search all members of a list L.

Prolog is the only language, where two lines code can be interesting. The definition of `memb` has two clauses. The first clause says that `H` is member of any list that can be represented by the pattern `[H|_]`, since it is the first element of the list. The second clause says that `H` is a member of a list represented by the `[_|T]` pattern, if it is member of the tail T.

```
%% ?- consult('utilities.pl').


memb(H, [H|_]).
memb(H, [_|T]) :- memb(H, T).
```

As one would expect from the logic of the code, Prolog answers yes when confronted with the `?-memb(2, [3,2,4])`, since 2 is member of the `[3,2,4]` list. What is magic in Prolog is that, when one submit the query `?-memb(X,[3,2,4])`, the inference engine finds all members of the list.

The inference engine starts at the first clause, and unify `X` with the head of `[3,2,4]`, what produces `X=3`. If the user does not like this first solution, the second clause changes the goal to `memb(X,[2,4])` by eliminating the refused answer. With the new goal, Prolog tries the first clause again, which produces `X=2`, the second answer.

```
> ros run
* (load "../../wamcompiler.lisp")
T
* (repl)
> ?- consult('utilities.pl').

yes.
> ?- memb(3, [2,3,4]).

yes.
> ?- memb(X, [2,3,4]).
X = 2
?;
X = 3
?;
X = 4
?;

no
```

PROBLEM 7. Specify the relation append (list concatenation) between three lists, which holds if the last one is the result of appending the first two.

```
%% ?- consult('utilities.pl').

app([], L, L).
app([H|T], L, [H|U]) :- app(T, L, U).
```

This problem is an all time favorite. If you run the goal `?-app(X,Y,[2,3,4,5])`, Prolog will find and print all ways of cutting the list `[2,3,4,5]`:

```
> rlwrap ros run
* (load "../../wamcompiler")
T
* (repl)
> ?- consult('utilities.pl').

yes.
> ?-app(X,F,[2,3,4,5]).
X = NIL
F = [2,3,4,5]
?;
X = [2]
F = [3,4,5]
?;
X = [2,3]
F = [4,5]
?;
X = [2,3,4]
F = [5]
?;
X = [2,3,4,5]
F = NIL
```

One can say that both Prolog and lisp use clauses to process alternatives and options. The difference is that, in Lisp, the choice is final. However, when the choice reveals itself as inadequate, Prolog can backtrack to the point of bifurcation and select another path.

Consider the definition of app, for instance. The first alternative for the `?-app(X,F,[2,3,4,5])` goal is the first clause, which makes `X=[]`. However, if the user is not happy with this solution, Prolog backtracks to the point of choice, and selects the second clause, where the `[X|T]` pattern can accept

lists of one or more elements, but not empty lists. Let us suppose that the
tail of the second clause matches the first clause, i.e., `app(T,L,U)` matches
`app([],L,L)`. In this case, the query will produce a second answer, `X=[2]`.
However, the predicate `app(T,L,U)` can skip the first clause, and match the
second, which would make `T=[3]`, and `X=[2,3]`, and so on.

PROBLEM 65.   [EMDEN, 1980] Write a program for the game of Nim
defined as follows.

A position of the game of Nim can be visualized as a set of heaps of
matches. Two players, called *us* and *them*, alternate making a move. As
soon as a player, whose turn is to move, is unable to make a move, the game
is over and that player has lost; the other player is said to have won. A move
consists of taking at least one match from exactly one heap.

Ten years ago, I published a book called *Visual Prolog for Tyros*, where
I tackled this and a few other problems proposed by Coelho. Here is my
solution for the Nim problem:

```
%% ?-consult('nim.pl').

ap([], L, L).
ap([H|T], L, [H|U]) :- ap(T, L, U).

tk([X],V,V).
tk([X,X1|Y],V,[[X1|Y]|V]).
tk([X|T],V,Y) :- tk(T,V,Y).

mv(X,Y) :- ap(U,[X1|V],X),
           tk(X1,V,R),ap(U,R,Y).

us(X,Y) :- mv(X,Y), \+ them(Y,Z).
them(X,Y) :- mv(X,Y), \+ us(Y,Z).
```

Coelho wrote: *A position of the game of Nim can be visualized as a set of
heaps of matches.* Let us represent each heap as a list of integers:

```
[1,1,1,1,1]
```

Therefore, the set of heaps will be a list of lists; e.g.

```
[  [1,1,1,1,1],
   [1,1],
   [1,1,1,1]]
```

Two players, *us* and *them*, alternate making moves. As soon as a player is unable to make a move, the game is over, and that player has lost. A move consists of taking at least one match from exactly one heap. In the example, if you take three matches from the third heap, you make a valid move for us, and the board becomes:

```
[  [1,1,1,1,1],
   [1,1],
   [1]]
```

where you have removed three matches from the third heap. To implement this project, we will use a programming technique called *incremental development of systems*. This technique is discussed in the first chapter of *On Lisp*, a book by Paul Graham, that I strongly recommend for your edification.

First, you will implement and test a program that append two lists. Since you are going to use this program both to split and to concatenate lists, you need to test both possibilities.

```
> ?-ap([1,1,1],[2,2,2,2],L).
L = [1,1,1,2,2,2,2]

yes.
> ?-ap(X,Y,[1,1,2,2]).
X = NIL
Y = [1,1,2,2]
?;
X = [1]
Y = [1,2,2]
?;
X = [1,1]
Y = [2,2]
?;
X = [1,1,2]
Y = [2]
?;
X = [1,1,2,2]
Y = NIL
```

The first query gives the result of concatenating two lists. The lines that follow shows the many possibilities of splitting a list. Them the first program seems to be working properly.

The next step is to write and test a program that takes at least one match from a heap; of course, it can take more than one match. After remove one

or more matches from the heap, `tk/3` insert the modified heap into a set of
heaps. If you test the program, it will produce the following solutions:

```
> ?- tk([1,1,1,1],[[1,1]],Resp).
Resp = [[1,1,1],[1,1]]
?;
Resp = [[1,1],[1,1]]
?;
Resp = [[1],[1,1]]
?;
Resp = [[1,1]]
?;

no.
```

N.B. The first clause of `tk/3` makes sure that the predicate will not insert
an empty heap into the set of heaps.

I will leave the test of `mv/2` to your discretion. Below, you will find the
result of a game between a human player and the computer.

```
> rlwrap ros run
* (load "wamcompiler.lisp")
T
* (repl)
> ?-consult('nim.pl').

yes.
> ?-us([[1,1,1],[1,1]],Move).
Move = [[1,1],[1,1]]
?y

yes.
> ?-us([[1],[1,1]],Move).
Move = [[1],[1]]
?y

yes.
> ?-us([[1]],Move).
Move = NIL
?y

yes.
```

Chapter 3

Ninety-Nine Prolog Programs

Werner Hett says in his article Ninety-Nine Prolog Programs:

> The purpose of this problem collection is to give you the op-
> portunity to practice your skills in logic programming. Your goal
> should be to find the most elegant solution of the given problems.
> Efficiency is important, but logical clarity is even more crucial.
> Some of the (easy) problems can be trivially solved using built-in
> predicates. However, in these cases, you learn more if you try to
> find your own solution.
>
> Every predicate that you write should begin with a comment
> that describes the predicate in a declarative statement. Do not
> describe procedurally, what the predicate does, but write down a
> logical statement which includes the arguments of the predicate.
> You should also indicate the intended data types of the arguments
> and the allowed flow patterns.
>
> The problems have different levels of difficulty. Those marked
> with a single asterisk (*) are easy. If you have successfully solved
> the preceeding problems you should be able to solve them within
> a few (say 15) minutes. Problems marked with two asterisks
> (**) are of intermediate difficulty. If you are a skilled Prolog
> programmer it shouldn't take you more than 30-90 minutes to
> solve them. Problems marked with three asterisks (***) are more
> difficult. You may need more time (i.e. a few hours or more) to
> find a good solution.

I did not followed Hett's advice to the letter. In roder to check for bugs
in the compiler, I used cuts everywhere to prevent backtracking. It is easy
to restore the logic purity of the programs: Just eliminate the cuts.

```prolog
%% ?- consult('p99.pl').

%% P01 (*): Find the last element of a list
%% > ?-mylast(X, [1,2,3,4,5]).
%% X = 5

mylast(X, [X]) :- !.
mylast(X, [_|L]) :- mylast(X, L).

%% P02 (*): Find the last but one element of a list
%% > ?-last_but_one(X, [1,2,3,4,5]).
%% X = 4

last_but_one(X, [X,_]) :- !.
last_but_one(X, [_,Y|Ys]) :- last_but_one(X, [Y|Ys]).

% P03 (*): Find the K'th element of a list.
% The first element in the list is number 1.
%% > ?-element_at(X, [1,2,three,4,5], 3).
%% X = three
element_at(X,[X|_],1) :- !.
element_at(X,[_|L],K) :- K > 1,
   K1 is K - 1, element_at(X,L,K1).

% P04 (*): Find the number of elements of a list.
%% > ?-length([1,2,3,4,5], L).
%% L = 5

length([],0) :- !.
length([_|L],N) :- length(L,N1),
                   N is N1 + 1.

% P05 (*): Reverse a list.
%% > ?-reverse([1,2,3,4],L).
%% L = [4,3,2,1]

reverse(L1,L2) :- my_rev(L1,L2,[]).

my_rev([],L2,L2) :- !.
my_rev([X|Xs],L2,Acc) :- my_rev(Xs,L2,[X|Acc]).
```

```
% P06 (*): Find out whether a list is a palindrome
%% > ?-is_palindrome([s,u,b,i,d,u,r,a,a,r,u,d,i,b,u,s]).


%% Ignore spaces in Latin palindromes: Romans didn't use them:
%  subi dura a rudibus -- endure rudeness from peasants
%  roma tibi subito motibus ibit amor --
%    in Rome quickly with its bustle you will find love

is_palindrome(L) :- reverse(L,L).



% P07 (**): Flatten a nested list structure.
%% > ?-flatten([3,4,[a,[b]], 5], L).
%% L = [3,4,a,b,5]

append([],L,L) :- !.
append([H|T], L, [H|U]) :- append(T,L,U).

flatten([],[]) :- !.
flatten([X|Xs],Zs) :- !, flatten(X,Y),
   flatten(Xs,Ys), append(Y,Ys,Zs).
flatten(X, [X]).

% P08 (**): Eliminate consecutive duplicates of list elements.
%% > ?-compress([2,3,3,3,4,4,5], L).
%% L = [2,3,4,5]
compress([],[]) :- !.
compress([X],[X]) :- !.
compress([X,Y|Xs],Zs) :-  X==Y, !, compress([X|Xs],Zs).
compress([X,Y|Ys],[X|Zs]) :-  compress([Y|Ys],Zs).

% P09 (**):  Pack consecutive duplicates of list elements.
%% > ?-pack([2,3,3,3,4,4,5], L).
%% L = [[2],[3,3,3],[4,4],[5]]

pack([],[]) :- !.
pack([X|Xs],[Z|Zs]) :- transfer(X,Xs,Ys,Z), pack(Ys,Zs).

transfer(X,[],[],[X]) :- !.
transfer(Y,[X|Xs],Ys,[X|Zs]) :- X == Y, !, transfer(X,Xs,Ys,Zs).
transfer(X,[Y|Ys],[Y|Ys],[X]).
```

```
% P10 (*):  Run-length encoding of a list
%% > ?-encode([a,a,a,b,b,b,b,b,c,d], Ans).
%% Ans = [[3,a],[5,b],[1,c],[1,d]]

encode(L1,L2) :- pack(L1,L),
            transform(L,L2).

transform([],[]) :- !.
transform([[X|Xs]|Ys],[[N,X]|Zs]) :- length([X|Xs],N),
                                transform(Ys,Zs).

% P11 (*):  Modified run-length encoding
%% > ?-encode_modified([1,1,2,2,2,2,2,3,3,4], G).
%% G = [[2,1],[5,2],[2,3],4]

encode_modified(L1,L2) :- encode(L1,L),
                     strip(L,L2).

strip([],[]) :- !.
strip([[Num,X]|Ys],[X|Zs]) :- Num == 1, !,
                        strip(Ys,Zs).
strip([[N,X]|Ys],[[N,X]|Zs]) :- N > 1,
                            strip(Ys,Zs).


% P12 (**): Decode a run-length compressed list.
%% > ?-encode([a,a,a,b,b,b,b,b,c,d], Ans), decode(Ans, G).
%% Ans = [[3,a],[5,b],[1,c],[1,d]]
%%G = [a,a,a,b,b,b,b,b,c,d]

notlist(X) :- atom(X), !.
notlist(X) :- number(X).

decode([],[]) :- !.
decode([[Num,X]|Ys],[X|Zs]) :- Num == 1, !, decode(Ys,Zs).
decode([[N,X]|Ys],[X|Zs]) :- N > 1, !,
          N1 is N - 1,
          decode([[N1,X]|Ys],Zs).
decode([X|Ys],[X|Zs]) :- notlist(X),
                        decode(Ys,Zs).
```

```
% P13 (**): Run-length encoding of a list (direct solution)
%% > ?-encode_direct([1,1,2,2,2,2,2,3,3,4], G).
%% G = [[2,1],[5,2],[2,3],4]

encode_direct([],[]) :- !.
encode_direct([X|Xs],[Z|Zs]) :- count(X,Xs,Ys,1,Z),
                       encode_direct(Ys,Zs).

count(X,[],[],1,X) :- !.
count(X,[],[],N,[N,X]) :- N > 1, !.
count(X,[Y|Ys],[Y|Ys],1,X) :- X \= Y.
count(Y,[X|Xs],Ys,K,T) :- X == Y, !,
   K1 is K + 1, count(X,Xs,Ys,K1,T).
count(X,[Y|Ys],[Y|Ys],N,[N,X]) :- N > 1.

% P14 (*): Duplicate the elements of a list
%% > ?-dupli([1,2,2,3,3,3,5], L).
%% L = [1,1,2,2,2,2,3,3,3,3,3,3,5,5]

dupli([],[]) :- !.
dupli([X|Xs],[X,X|Ys]) :- dupli(Xs,Ys).

% P15 (**): Duplicate the elements of a list agiven number of times
%% > ?-dupli([1,2,2,3,3,3,5], 4,  L).
%% L = [1,1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,3,3,3,3,5,5,5,5]

dupli(L1,N,L2) :- dupli(L1,N,L2,N).
dupli([],_,[],_) :- !.
dupli([_|Xs],N,Ys, K) :- K == 0, !, dupli(Xs,N,Ys,N).
dupli([X|Xs],N,[X|Ys],K) :- K > 0, K1 is K - 1,
                dupli([X|Xs],N,Ys,K1).

% P16 (**):  Drop every N'th element from a list
%% > ?-drop([1,2,3,4,5,6], 3, R).
%% R = [1,2,4,5]

drop(L1,N,L2) :- drop(L1,N,L2,N).
drop([],_,[],_) :- !.
drop([X|Xs],N,[X|Ys],K) :- K > 1, !,
       K1 is K - 1, drop(Xs,N,Ys,K1).
drop([_|Xs],N,Ys, K) :-  drop(Xs,N,Ys,N).
```

```
% P17 (*): Split a list into two parts
%% > ?-split([2,5,3,8,1], 3, X, Y).
%% X = [2,5,3]
%% Y = [8,1]

split([X|Xs],N,[X|Ys],Zs) :- N > 0, !, N1 is N - 1,
                   split(Xs,N1,Ys,Zs).
split(L,0,[],L).

% P18 (**):  Extract a slice from a list
%% > ?-slice([1,2,3,4,5,6,7], 2,4, R).
%% R = [2,3,4]

slice([X|_],1,1,[X]) :- !.
slice([X|Xs],1,K,[X|Ys]) :- K > 1, !,
   K1 is K - 1, slice(Xs,1,K1,Ys).
slice([_|Xs],I,K,Ys) :- I > 1,
   I1 is I - 1, K1 is K - 1, slice(Xs,I1,K1,Ys).

% P19 (**): Rotate a list N places to the left
%% > ?-rotate([1,2,3,4,5,6,7], 2, R).
%% R = [3,4,5,6,7,1,2]

rotate(L1,N,L2) :- N >= 0, !,
   length(L1,NL1), N1 is N mod NL1, rotate_left(L1,N1,L2).
rotate(L1,N,L2) :-
   length(L1,NL1), N1 is NL1 + (N mod NL1), rotate_left(L1,N1,L2).

rotate_left(L1,N,L2) :- N > 0, !,
    split(L1,N,S1,S2), append(S2,S1,L2).
rotate_left(L,0,L).

% P20 (*): Remove the K'th element from a list.
%% > ?-remove_at(X, [1,2,3,4,5,6], 3, L).
%% X = 3
%% L = [1,2,4,5,6]

remove_at(X,[Y|Xs],K,[Y|Ys]) :- K > 1, !,
   K1 is K - 1, remove_at(X,Xs,K1,Ys).
remove_at(X,[X|Xs],1,Xs).
```

```
% P21 (*): Insert an element at a given position into a list
%% > ?-insert_at(a,[1,2,3,4,5], 3, L).
%% L = [1,2,a,3,4,5]

insert_at(X,L,K,R) :- remove_at(X,R,K,L).


% P22 (*):  Create a list containing all integers within a given range.
%% > ?-range(4,9,L).
%% L = [4,5,6,7,8,9]

range(I,K,[I]) :- I >= K, !.
range(I,K,[I|L]) :- I < K,
    I1 is I + 1, range(I1,K,L).


% P23 (**): Extract a given number of randomly
% selected elements from a list.
%% (load "rnd.lisp") the Lisp function
%% that generates random numbers.
%% Study file "rnd.lisp" to learn how to
%% implement new primitives.
%% > ?- rnd_select([1,2,3,4,5,6], 3, L).
%% L = [6,2,4]

rnd_select(_,K,[]) :- K < 1, !.
rnd_select(Xs,N,[X|Zs]) :- N > 0,
    length(Xs,L),
    random(R, L), I is R + 1,
    remove_at(X,Xs,I,Ys),
    N1 is N - 1,
    rnd_select(Ys,N1,Zs).


% P24 (*): Lotto: Draw N different random numbers from  1..M
%% > ?-lotto(6,49,L).
%% L = [44,36,39,5,28,34]

lotto(N,M,L) :- range(1,M,R), rnd_select(R,N,L).
```

```
% P25 (*):  Generate a random permutation of the elements of a list
%% > ?-rnd_permu([1,2,3,4,5], L).
%% L = [2,4,3,1,5]

rnd_permu(L1,L2) :- length(L1,N), rnd_select(L1,N,L2).
```

Chapter 4

Shell

Nia, a Greek young woman, has an account on a well known network. She visits her friends' postings on a daily basis, and when she finds an interesting picture or video, she presses the *Like*-button. However, when she needs to discuss her upcoming holidays on the Saba Island with her Argentinian boyfriend, she uses the live chat box. After all, hitting buttons and icons offers only a very limited interaction tool, and does not produce a highly detailed level of information that permits the answering of questions and making of statements.

Using a chat service needs to be very easy and fun, otherwise all those teenage friends of Nia's would be doing something else. I am telling you this, because there are two ways of commanding a computer. The first is called Graphical User Interface (GUI) and consists of moving the cursor with a mouse or other pointing device and clicking over a menu option or an icon, such as the *Like* button. As previously mentioned, a Graphical User Interface often does not generate adequate information for making a request to the computer. In addition, finding the right object to press can become difficult in a labyrinth of menu options.

The other method of interacting with the computer is known as Shell and is similar to a chat with one's own machine.

In your computer, there is a giant program, called the operating system, which controls all peripherals that the machine uses to stay connected with the external world: pointing devices, keyboard, video terminal, robots, cameras, solid state drives, pen drives, and other peripherals.

In a Shell interface, Nia issues written instructions that the operating system answers by fulfilling the assigned tasks. The language that Nia uses to chat with the operating system is called *Bourne-again shell*, or bash for short. This language has commands to go through folders, browser files, create new directories, copy objects from one place to the other, configure

the machine, install applications, change the permissions of a file, create groups to organize users and devices, etc. When accessing the operating system through a text-based terminal, a shell language is the main way of executing programs and doing work on a computer.

The shell interface derives its name from the fact that it acts like a shell surrounding all other programs being run, and controlling everything the machine performs.

To make a long story short, it is much faster to complete tasks using a shell than with graphical applications, icons, menus and mouse. Another benefit of the shell is that you can gain access to many more commands and scripts than with a Graphical User Interface.

In order not to scare off the feeble-minded, many operating systems hide access to the text terminal. In some distribution of Linux, you need to maintain the Alt key down, then press the F2 key to open a dialog box, where you must type the name of the terminal you want to open. If you are really lucky, you may find the icon of the terminal on the toolbar.

If the way of opening the text terminal is not obvious, you should ask for help from a student majoring in Computer Science. You can teach her Russian, Samskrit, Javanese or Ancient Greek, as a compensation for the time that she will spend explaining how to start a terminal in OS X or Linux. If you are afraid of paying for a few minutes of tutorial about the Bourne-again shell with hundred hours of classes on Russian, don't worry! The CS major will give up after barely starting the section on the alphabet. After all, Philology is much more difficult than Computer Engineering. For details, read the tale "The man who could speak Javanese" by Lima Barreto.

The prompt. The shell prompt is where one types commands. The prompt has different aspects, depending on the configuration of the terminal. In Nia's machine, it looks something like this:

```
~$ _
```

Files are stored in folders. Typically, the prompt shows the folder where the user is currently working.

The main duty of the operating system is to maintain the contents of the mass storage devices in a tree structure of files and folders. Folders are also called *directories*, and like physical folders or cabinets, they organize files.

A folder can be put inside another folder. In a given machine, there is a folder reserved for duties carried out by the administrator. This special folder is called home or personal directory.

Besides the administrator, a machine can have other rightful users, each with a personal folder. For instance, Nia's folder on her Mac OS X has the `/Users/nia` path. You will learn a more formal definition of path later on.

pwd # The `pwd` command informs the user's position in the file tree. A folder can be placed inside another folder. For example, in a Macintosh, Nia's home folder is inside the `/Users` directory.

One uses a path to identify a nest of folders. In a path, a subfolder is separated from the parent folder by a slash bar. If one needs to know the current folder, there is the `pwd` command.

```
~$ pwd          # shows the current folder. Enter
/Users/nia
```

When Nia issues a command, she may add comments to it, so her boyfriend that does not know Bourne-again shell (bash) can understand what is going on and learn something in the process. Comments are prefixed with the `#` hash char, as you can see in the above chat. Therefore, when the computer sees a `#` hash char, it ignores everything to the end of the line.

mkdir wrk # creates a `wrk` folder inside the current directory, where `wrk` can be replaced with any other name. For instance, if Nia is inside her home directory, `mkdir wrk` creates a folder with the `/Users/nia/wrk` path.

cd wrk # One can use the `cd <folder name>` commands to enter the named folder. The `cd ..` command takes Nia to the parent of the current directory. Thanks to the `cd` command, one can navigate through the tree of folders and directories.

Tab. If you want to go to a given directory, type part of the directory path, and then press `Tab`. The shell will complete the folder name for you.

Home directory. One can use a `~` tilde to represent the home directory. For instance, `cd ~` will send Nia to her personal folder. The `cd $HOME` has the same effect, i.e., it places Nia inside her personal directory.

echo #   One can use the `echo` command to print something. Therefore, `echo $HOME` prints the contents of the `HOME` environment variable on the terminal.

Environment variables store the terminal configuration. For instance, the `HOME` variable contains the user's personal directory identifier. One needs to prefix the environment variable with the `$` char to access its contents. Therefore, `echo $HOME` displays the contents of the `HOME` variable.

The instruction `echo "Work Space" > readme.txt` creates a `readme.txt` text file and writes the `"Work Space"` string there. If the `readme.txt` file exists, this command supersedes it.

The command `echo "Shell practice" >> readme.txt` appends a string to a text file. It does not erase the previous content of the `readme.txt` file. Of course, you should replace the string or the file name, as necessity dictates.

Below you will find an extended example of a chat between Nia and the OS X operating system.

```
~$ mkdir wrk   # creates a work space folder        Enter
~$ cd wrk      # transfers action to the wrk file   Enter
~/wrk$ echo "* Work space" > readme.txt   Enter
~/wrk$ echo "This folder is used" >> readme.txt   Enter
~/wrk$ echo "to practice the bash" >> readme.txt   Enter
~/wrk$ echo "commands and queries." >> readme.txt   Enter
~/wrk$ ls      # list files in current folder.   Enter
readme.txt
~/wrk$ cat readme.txt   # shows the file contents.   Enter


* Work space
This folder is used
to practice the bash
commands and queries.
```

ls #   By convention, a file name has two parts, the id and the extension. The id is separated from the extension by a dot. The `ls` command lists all files and subfolders present in the current folder. The `ls *.txt` prints only files with the `.txt` extension.

Wild card.   The `*.txt` pattern is called wild card. In a wild card, the `*` asterix matches any sequence of chars, while the `?` interrogation mark matches a single char.

ls -lia *.txt # prints detailed information about the .txt files, like date of creation, size, etc.

```
~/wrk$ ls -la  Enter


total 8
drwxr-xr-x    3 edu500ac  staff    102 Sep 18 16:29 .
drwxr-xr-x+ 321 edu500ac  staff  10914 Sep 18 14:40 ..
-rw-r--r--    1 edu500ac  staff     74 Sep 18 16:30 readme.txt
```

Files starting with a dot are called hidden files, due to the fact that the `ls` command does not normally show them. All the same, the `ls -a` option includes the hidden files in the listing.

In the preceding examples, the first character in each list entry is either a dash (-) or the letter d. A dash (-) indicates that the file is a regular file. The letter d indicates that the entry is a folder. A special file type that might appear in a `ls -la` command is the symlink. It begins with a lowercase l, and points to another location in the file system. Directly after the file classification comes the permissions, represented by the following letters:

- r – read permission.

- w – write permission.

- x – execute permission.

cp readme.txt lire.txt # makes a copy of a file. You can copy a whole directory with the **-rf** options, as shown below.

```
~/wrk$ ls
readme.txt
~/wrk$ cp readme.txt lire.fr
~/wrk$ ls
lire.fr  readme.txt
~/wrk$ cd ..
~$ cp -rf wrk wsp
~$ cd wsp/
~/wsp$ ls
lire.fr  readme.txt
```

rm lire.txt #   removes a file. The `-rf` option removes a whole folder.

```
~/wsp$ ls
lire.frreadme.txt
~/wsp$ rm lire.fr
~/wsp$ ls
readme.txt
~/wsp$ cd ..
~$ ls wrk
lire.frreadme.txt
~$ rm -rf wrk
~$ ls wrk
ls: wrk: No such file or directory
```

mv wsp wrp #   changes the name of a file or folder, or even permits the moving of a file or folder to another location.

```
~$ mv wsp wrk
~$ cd wrk
~/wrk$ ls
readme.txt
~/wrk$ mv readme.txt read.me
~/wrk$ ls
read.me
~/wrk$ cp read.me wrk-readme.txt
~/wrk$ ls
read.mewrk-readme.txt
~/wrk$ mv wrk-readme.txt ~/Documents/
~/wrk$ ls
read.me
~/wrk$ cat ~/Documents/wrk-readme.txt
Work space
This folder is used
to practice the bash
commands and queries.
```

Copy to pen drive.   In most Linux distributions, the pen drive is seen as a folder inside the `/media/nia/` directory, where you should replace `nia` with your user name. In the Macintosh, the pen drive appears at the `/Volume/` folder. The commands **cp**, **rm** and **ls** see the pen drive as a normal folder.

4.1 Package managers

A package manager is a tool that works with core libraries to handle, even though poorly, the installation and removal of software. Each operating system has its own package manager. Even if you stick to a single operating system, you will have to deal with many package managers and different philosophies concerning software installation.

I will teach you how to use three installation tools: apt-get, homebrew and git clone. However, this tutorial will not completely cover the many subtleties involved while you make your machine usable. You will really need the help of that girl who is majoring in Computer Engineering. Tell her that, if she teaches you how to configure apt-get install, you will teach her the ancient Javanese script.

4.2 Roswell

People say that Lisp is alien technology. Therefore, it is no surprise that one of the most popular Lisp installer is called *roswell*.

In order to find the page of *roswell*, I performed an web search with the following keywords: *Common Lisp roswell*. There, I learned that roswell could be installed in a Macintosh through a package manager called homebrew:

```
$ brew install roswell
```

If you search for *roswell* by using the *Common Lisp* and *roswell* keywords, you will certainly find a easy way of installing *roswell* in your machine.

The next step after installing *roswell* is to fix the `$PATH` environment variable to include *roswell*. In my machine, this is done by including a line to the `.zshrc` shell configuration file:

```
export PATH="$PATH:/usr/local/Cellar/roswell/19.09.12.102/bin"
export PATH="$PATH:/Users/ed/.roswell/bin"
```

I don't give more details because this kind of information changes from time to time, and also from machine to machine. You will need to discover the details by yourself. That said, here are the main commands of roswell:

```
$ ros install                # Available Lisp versions
$ ros install sbcl-bin       # default sbcl
$ ros install sbcl           # Newest released version of sbcl
$ ros install ccl-bin        # default prebuilt binary of ccl
$ ros list installed sbcl    # List the installed versions
$ ros run                    # Execute Lisp
```

Emacs is huge. If you want a compatible editor that consumes much less computational resources, here is the recipe to install the Lem editor:

```
$ ros install cxxxr/lem
```

4.3   github

In the jargon of tech-oriented people, a person who uses computers is called *user*, but engineers prefer the term *looser*, as the latter is an affectionate way of calling those individuals who need to work with a computer, but in fact have no skills. By the way, looser is a loser who can't spell "loser". Fortunately for us *losers*, there exists a repository called github. Let us see how to download wamcompiler from github:

```
git clone https://github.com/matsud224/wamcompiler
```

4.4   sudo

Installation often requires that the system copy files to the `/usr/local/` folder. However, you must give special permission for this operation to be performed, otherwise the possibility remains open for a virus to make similar copies in critical folders. The `sudo` command informs the machine that you are a superuser. To avoid being tricked by a virus or an unauthorized user,

the machine asks for the password, and if you type it correctly, the system performs the installation.

Let us see an example of sudo in action. If you decide to use the Lem editor, instead of Emacs, you may need to install a front end library called `libncurses`. Let us understand this issue. A text editor, like Emacs or Lem, has three components:

1. A frontend that inserts characters into a buffer. A buffer is a memory region with an image of a text file. One can see the buffer as a representation of a scratch pad. As such, the frontend provides tools to insert keystrokes at the point indicated by a movable cursor.

2. Editing operations that save buffers as files, delete and insert text, move the cursor around, copy items from one place to another, etc.

3. A scripting language for customizing the editor.

There is a large choice of front-ends. Some front-ends work with a raster graphics image, which is a dot matrix data structure that represents a grid of pixels. There are also front-ends that are specialized in showing letters and other characters via an appropriate display media. The latter sort of front-end is called text-based user interfaces, while the former is called graphical user interface, or GUI for short.

Text-based user interfaces are more comfortable on the eyes, since they provide sharper and crisp alphabetical letters. On the other hand, GUI allows for font customization. At present, the Lem editor offers only ncurses, which is a text-based user interface. Therefore, search the web for the distribution site and download the most recent version of this front-end.

Unpack and make the distribution archive as shown below:

```
~$ tar xfvz ncurses-6.0.tar.gz
~$ cd ncurses-6.0/
~/ncurses-6.0$ ./configure --enable-widec
~/ncurses-6.0$ make
~/ncurses-6.0$ sudo make install
```

Lem assumes that `libncursesw.a` resides in the following folder:

    /usr/local/lib/libncursesw.a

Therefore, copy the `libncursesw.a` file to `/usr/local/lib/` as shown below:

```
~/$ cd ncurses-6.0
~/ncurses-6.0$ cd lib
~/ncurses-6.0/lib$ sudo cp libncursesw.a /usr/local/lib/
```

I have to say this again: An installation task has so many details that, unless you are an engineer yourself, ask for help from a computer expert.

Chapter 5

The man who knew Javanese

In a doughnut shop the other day, I was telling my friend Castro how I tricked respectable members of the community not only to make a living, but also rise to a higher social position.

For instance, while I was living in Manaus, I had to hide my professional degree to win over the clients' trust. People, who never would come to a physician's practice or lawyer firm, flocked to my wizard and fortune teller's office. So, I was telling him this story.

My friend listened to me in silence, enraptured by my words, appreciating the account of picaresque adventures, that seemed as though they had been extracted from the Story of Gil Blas. During a conversational pause, after both drying our glasses, he remarked randomly:

"You've been living an extremely funny life, Castelo!"

"It's the only kind of life worth living... I could not immagine myself having a single occupation. It is extremely boring to go to work in the morning, come back in the evening, read the paper on the sitting room armchair, and go to bed at eleven fifteen. Don't you think? In fact, I can't imagine how I have been able to stand my job at the Consulate!"

"Yes, the existence in the civil service must be tedious. But that's not what amazes me. I wonder how have you been able to pass through so many adventures here, in this monotonous and bureaucratic Brazil."

"You will be surprized, my dear Castro, to learn that even here, in this country, one can write quite interesting pages on a memoir. Believe it or not, I was even a teacher of Javanese!"

"When was that? Here, after you retired from your job at the Consulate?"

"No, it was a long time ago. What's more, I got my job at the Consulate as a consequence of my teaching duties."

"That story I want to hear! In the mean time, let us have a few more drinks?"

"Yes, I accept another glass of beer."

We sent for another bottle, filled our glasses and I continued:

"I had just arrived in Rio and I was literally broke. My life was consumed fleeing from one boarding house to the other on rent day, without idea of how to earn a living.

Of course, I didn't have money to pay for breakfast. Even so, I was in a café, but only to read through the job openings section of the newspaper left on a table for the customers' convenience. Then, I came across the following advertisement:

"Needed: A JAVANESE TEACHER. Interviews at, etc."

So, I told myself: "This certainly is a position where you won't find many applicants. If only I could comprehend a few words of Javanese, I'd apply."

I left the coffee shop in a waking dream, where imagined myself in a brave new world, as a Javanese teacher, earning more than just a decent living, driving my car, without unhappy meetings with debt collectors.

When I awoke from my daydream, I found myself in front of the public library. I went in and gave my cap to a person that looked like an attendant, since I knew about the custom of the upper class to leave their hats at the hat stand. I did not have a hat, so I left my cap. Of course, I never saw my cap again.

I didn't really know what book I was going to ask for. In any case, this was irrelevant, because my name was nationally blacklisted, since I never returned the books to the highschool library. Nevertheless, I climbed the stairs to browse an Encyclopedia, volume J and look up the entry on Java and the Javanese dialect. In a few minutes, I knew that Java was a big island of the Greater Sunda Islands, at the time a Dutch colony.

I also learned that Javanese is an agglutinative language of the Malayo-Polynesian family. It has a noteworthy literature written in characters derived from the old Hindu alphabet. The Javanese alphabet is a modern variant of the Kawi script, a Brahmic script developed in Java around the ninth century. In the past, it was widely used in religious literature, which was written on palm-leaves. This kind of manuscript came to be known as lontar. The Encyclopedia gave a list of books for further reading and I had no doubt of what I needed to do, and consulted one of them. I made a copy of the alphabet and its phonetic transcription and left. I wandered the streets walking aimlessly chewing over the letters as they were gum.

Hieroglyphs danced in my head. From time to time, I would look at my notes. Then I would go into Tijuca park, and wrote those strange looking characters with a stick in the sand to keep them vividly in my memory and accustom my hands in writing them.

I would enter my building late at night in an attempt to avoid coming

across the landlord, who could ask for the rent. Even in my bedroom I kept chewing over my Malayan A-B-C. My determination was such that I knew it all by heart with the sunrise. I convinced myself that Malay was the easiest language in the world, and that Javanese was as close to Malay as Portuguese to Spanish, which simply is not true. The reality is that Malay is not so easy, and it is as far from Javanese as English from German.

I tried to leave home as early in the morning as possible to avoid my landlord. But to my dismay, it was not early enough, for I couldn't escape meeting the man in charge of asking for the room rent.

"Mr. Castelo, when are you going to pay your rent bill?"

I aswered him with the most heart filled hope:

"Soon. . . Wait just a little longer. . . Be patient. . . I will be appointed as a Javanese teacher and. . . " At that moment, the man cut in into my stuttering excuses.

"What in hell is that, Mr. Castelo?"

I enjoyed the amusement and proceeded to invest in the man's patriotism:

"It's a language from the distant reaches of Timor. Do you know where it is?"

Oh, simple minded fellow! The man forgot all about my overdue bill and told me in a strong accent from Portugal:

"I am not sure, Mr. Castelo. If I remember rightly, Portugal has or had a colony with that name close to Macau. But do you really know such a thing, Mr. Castelo?"

Incentivated by this initial result from my Javanese studies, I started to fulfil the requisite of the advertisement. I decided to offer my services as a teacher of that transoceanic idiom. I composed the letter of acceptance, then I went to the newspaper office to left the document there. With that, I returned to the library to resume my Javanese studies. I didn't make any progress that day. In fact, I felt that the alphabet was the only knowledge required from a Javanese teacher. Perhaps, I was overly indulged with the bibliography and history of the language that I was going to teach.

Two days later I received an answer to my correspondence, in which was stated that I should go to the residence of a certain Dr. Manuel Feliciano Soares Albernaz, Baron of Jacuecanga, Conde de Bonfim street. I can't recall the house number. Remember that I was extremely focused on Malayan studies to register the details for history, as for example the house number. But don't be fooled into thinking that my lack of memory for particularities, such as house numbers or names of historical figures, renders the story as a figment of my immagination. More than one scholar told me that the name of a certain Prince Kalunga is Aji Saka. There you are.

Besides the alphabet, I knew the names of a few authors. I also could

say, "How are you?", and knew a few grammar rules. All this knowledge was supported on around twenty words.

You can't imagine how hard I tried to get the money for the bus fare! Javanese is easier than loaning money, you can be sure. Failing to raise the necessary funds for the trip from one neighborhood to the other, I walked. Of course, I arrived in a state of dripping sweat. With a motherly affection, the old mango trees standing in front of the baron's house received, harbored and cooled me down with the fresh air of shade. It was the first time in all my life that I felt sympathy for Nature.

The house was huge, but ill kept. However, it was that way more from despondency and weariness of living than from its owner's poverty. The walls hadn't been painted for years, and what was left of their coats was peeling away. The eaves around the edges of the roof were made up of old fashioned tiles, some of which were missing, creating as effect of decaying false teeth. In the garden, the flatsedge and other weeds had expelled the begonias. However, the more resistent crotons continued to bloom with their purplish leaves.

I knocked. It was awhile before an aged African Negro came to the door. His white beard and hair gave him an appearance of fatigue and suffering.

In the parlor there was a row of gold framed portraits, which showed bearded gentlemen and profiles of sweet ladies holding huge fans and dresses that looked like balloons ready to ascend in the air. From among the many items, to which the dust gave more antiquity and respect, what impressed me most was a big, beautiful, porcelain vase from China or India, I never learned to differentiate their origins. Anyway, the purity of the material, its fragility, the naiveté of its contour and its dim lustre that appeard to be as the moonlight suggested to me that the artwork was fashioned by the hands of a dreaming child for the enchantment of old men's disillusioned eyes.

As I described above, I was inspecting the furniture, while waiting for the master of the house. He delayed quite a long time. Then, he came. Full of respect, I watched the elderly man approach haltingly, a large handkerchief in his hand, inhaling an old fashioned peppermint essence venerably.

Although I wasn't sure he was my pupil to be, I felt that it would have been wicked to deceive the aged gentleman, whose ancient aspect made me think of something august, sacred. I hesitated for a moment. Should I invent an excuse and leave? But finally decided to wait and see.

"I am," I said to break the ice, "The Javanese teacher you asked for."

"Sit down," answered the old gentleman. "Are you from Rio?"

"No, Sir. I'm from Canavieiras."

"What?" He said. "Speak louder, my hearing is impaired."

"I am from Canavieiras," I repeated putting emphasis upon each word.

"Where did you go to school?"

"In the city of Salvador, in the great state of Bahia."

"And where did you learn Javanese?" he asked with that dodgedness so common among the elderly.

I was not expecting for such a question, but I made up a lie about a Javanese father, who had come to Bahia on a freighter. Liking what he saw, my phony father decided to settle there in Canavieiras as a fisherman. He married, prospered and taught me Javanese.

"Did he believe you?" asked my friend who up to that moment had remained silent. "And your features?"

"I'm not very different from a Javanese" I contested. "With my thick, straight, black hair and tanned skin, I could very well pass for a Indonesian halfbreed. You know very well that among us there are all kinds of nationalities – Indians, Malayans, Tahitians, Madagascans, Guanches[1] and even Goths. Our people are an amalgamation of races and types to make the whole world envy us."

"O.K.." My friend agreed. "Please, go on."

"The old gentleman," I resumed, "listened intently and examined my physique for a long while. Then he concluded that I was indeed the son of a Malayan, and asked me softly:

"Well, do you really want to teach me Javanese?"

The answer came unwittingly: "Yes."

"You must be astounded," added the Baron of Jacuecanga, "that I, at my age, should still have a wont for learning."

"I'm not surprized at all. There have been noteworthy examples of late learners, as Socrates, who started flute at seventy."

"What I really want, Mr.. . . ?"

"Castelo," I supplied.

"What I really want, Mr. Castelo, is to fulfill a family pledge. I don't know whether you realize that I'm the grandson of State Secretary Albernaz, the same man who accompanied Peter, the first emperor of Brazil, into exile, after his abdication. When he came back from London, the Secretary brought with him a book written in a strange language, onto which he bestowed great value.

The old volume had been given to my grandfather by an Indian or a Siamese sailor in return for what received favor I do not know.

Before he died, my grandfather called my father to his deathbed and told him the story that follows.

---

[1]Guanches were the Berber aboriginal inhabitants of the Canary Islands.

"Son, do you see this book here, written in Javanese? Well, the person who gave it to me believed that it would bring its owner happiness and deliverance from evil. I don't know whether you should believe in such a legend or not. In any case, keep the book, and if the good omens prophesized by the oriental wiseman come true, teach your son to read it, so that our family line should prosper."

"My father," proceeded the old baron, "didn't take the story to heart. Nevertheless, he maintained the book in safe keeping. When he lay sick and suspected that the end of his life was near[2], he gave it to me repeating the dying words of his own father."

In the beginning, I paid little attention to the book. I threw it in a corner, and got on with life. I even forgot it existed. Lately, so many misfortunes have befallen me, that I remembered the old family heirloom. I must read and understand the book, if I want to preserve my last days from witnessing the total ruin of my posterity. Of course, to read the book, I need to master the Javanese language. There you have all of it."

He became silent. I noticed his eyes glistened with tears. He wiped them discreetly and asked me if I would care to see the book. I gave a yes. He summoned the maid, passed her the instructions to find and fetch the book, while he told me how he had lost all his children, nephews and nieces. Only one married daughter remained alive. From her numerous offspring, only one son survived, a weak and infirm boy.

The book arrived, an old, large volume, presented in quarto bound in leather and printed in huge letters on yellowed paper.

Since the title page was missing, I was unable to determine the date of publication. Fortunately, the preface was written in English. There I read that the book contained the stories of a Prince Kalunga, a renowened writer.

I explained this entirety to the baron. Ignorant of the fact that I arrived at this deliverance of knowledge from the English preface, he was very impressed with my erudition on Javanese culture. I browsed the pages with the look of someone familiar with that language that most people would not be able to tell apart from Sanskrit or Hindi.

Before parting company, the baron and I agreed upon my fees and the class schedules. To fulfill my part in the contract, I should teach my old pupil to read the book in a year.

A few days later, I gave the gentleman the first private lesson, but the old man was not diligent, not even at my level of interest. At the end of the class, he could not distinguish one character of the abugida from the other,

---

[2]Before the Scholar among my readers accuse me of plagiarism, I avow that I am fluent in Ancient Greek, and the first book I read in my life was the Anabasis.

or trace the five first letters of the hanacaraka sequence.

Traditionally, the Javanese syllabary is taught through a poem of 4 verses narrating the myth of Aji Saka.

ꦲꦤꦕꦫꦏ – There were two messengers

ꦢꦠꦱꦮꦭ – with mutual hatred.

ꦥꦝꦗꦪꦚ – One was as mighty as the other.

ꦩꦒꦧꦛꦔ – Here are the corpses.

In the poem, each syllable is written with a different letter. After learning the first verse, the student has learned five letters. Five more letters are presented in each sebsequent verse.

It took a whole month for the Baron of Jacuecanga to learn the first and second verse. What is worse, only one day was enough for him to forget everything. So, both teacher and student started a long cycle of learning and forgetting.

I think that the Baron's daughter and her husband didn't know anything about the book until they become curious of my activities in their home. When I explained the matter to them, they took it lightly. They laughed about the behavior of their elderly relative, said that he probably had dementia, and that learning languages could slow the progress of the disease.

You won't believe it, my dear Castro, but the son-in-law came to admire and respect the teacher of Javanese. He kept saying: "It is amazing how my father-in-law's tutor could grasp a culture so different from our own! Yet, he is so young. If I were as knowledgeable as he, I would be a scholar at Cambridge or at the Sorbonne.

The husband of Mrs. Maria da Gloria, that was the name of the Baron's daughter, was a judge, a powerful and well connected man. Even so, he never tired of showing his admiration for my knowledge of Javanese.

The Baron also seemed happy with me. But after two months he gave up learning the language. Instead of reading the book himself, he concluded that it would be enough to understand its contents in order to fulfill the pledge he made to his dying father. Certainly, the powerful spiritual forces behind the book would not be opposed to the humble request for the services of a translator. He would hear my rendition of the story of Aji Saka, avoiding the effort that his old brain was not in shape to bear.

You are certainly aware that even today I don't know Javanese. But as every learned man in this country, I had heard the story of King Aji Saka,

or Prince Kalunga, as Brazilians prefer to call the hero. In my rural village, at night, in my room illuminated by candles and kerosene lamps, my mother used to read a translation of the Indonesian legend into Latin until I would fall asleep. Therefore, it was not very difficult to patch together memories from my childhood, fill the gaps, and present the result to the old man as coming directly from the book. He would hear those myths in ecstasy, as though an angel were providing the rendition. And my reputation increased among the members of the Baron's family and its circle of acquaintainces. The Baron raised my salary, and I started living an easy life.

An unexpected fact contributed to my prestige. The Baron received an inheritance from a Portuguese relative that he didn't even know existed. Of course, he assigned the happy event to the Javanese book. I myself almost believed that this was so.

As time went by, I stopped myself from feeling remorse and guilty due to my deceiving the naive man and his family. In any case, I was terrified at the prospect of coming across some horrible person that could speak the Javanese dialect. My terror increased when the Baron wrote a letter to the Viscount of Caruru suggesting my name for an international affairs career. I advanced every kind of objection to the idea. "I am very ugly, uncouth and gaunt. My knowledge of French is faulty. I am not elegant. I don't know the protocols."

He insisted: "Physical aspect doesn't matter, young man. Go ahead! Everybody speaks French, German and English. But only you know Javanese."

So, I went to the interview with the Viscount, who sent me to the Ministry of Foreign Affairs with more than one recommendation letter. After all, everybody wanted to have the honor of giving me a recommendation letter. I was a huge success when I arrived at the office of the director of fucking nothing[3].

The director called all the department heads: "Look at this! The man knows Javaness. What a prodigy!"

The heads of various departments introduced me to clerks and amanuensis. One of these lesser officials of the Ministry gave me a look with more envy and hatred than admiration. But everybody else kept saying: "Do you really know Javanese? Is it difficult? I believe that nobody else speaks that language here."

The clerk who had looked at me with undeserved hatred approached the group of admirers and interrupted the applauses with a cold comment: "That is true, nobody knows Malay or Indonesian here. But I speak Kanak,

---

[3]A literal translation of the Portugese acronym for ASPONE – Assessor de Porra Nenhuma.

a language of the New Caledonia. Do you know Kanak?

I told him I didn't and proceeded to the Minister's office.

The high authority got up, adjusted the glasses on his nose, and asked point blank: "So, do you speak Javanese?" My answer was a loud *yes*. He wanted to know where I learned the language. I told him about my Javanese father. The Minister was sincere and direct. "You can't go into the Diplomatic Service. Brazilian Diplomats must be blond with blue eyes. Of course, there is the affirmative action for blacks and indians. But although your skin is dark, you cannot apply, since you are Javanese, not Indian. We could send you to a consulate in Asia or Oceania, but I am afraid that there is no opening now. However, if a position appears, you will get it. Meanwhile, you will be attached directly to my Ministry. By the way, I want you to go to Basel sometime next year, in order to represent our country in a congress of Linguistics. Read the books of Hove-Iacque, Max Müller, and other good authors in the field."

Can you follow me? I didn't know Javanese and could barely ask for lunch in French or English, but have a good job, and would represent my country in a meeting of scholars!

The old Baron came to pass, and the book went to his son-in-law, with the intention that it be given to his grandson when the boy come to age. The deceased Baron also left me a sum in his will.

I cannot say that I didn't try to learn tha Malayo-Polynesian languages, but to no avail. The effort of learning those languages is way out of my reach. Besides this, I had more pressing business to take care of: Eating well, dressing elegantly, and reading Comics.

Anyway, even if you don't read much, it is advisable to have shelves of books and many volumes of journals in your office. Therefore, I subscribed to the Revue Anthropologique et Linguistique, the Proceeding of the English Oceanic Association, and the Archivo Glottologico Italiano. Of course, subscribing to those scholarly journals without reading them did add anything to my learning of languages.

Yet, my reputation did not stop increasing. People who I met on the street would greet me with the comment:

"There goes the man who knows Javanese."

In book stores, grammarians often would consult me about the position of pronouns in the dialect spoken on the island of Sonda. Scholars sent me letters from all over the world and newspapers would write articles about me. On an occasion, I had to refuse a group of well to do young students who wanted to learn Javanese at any cost.

Do you remember that commerce paper where I found the advertisement? The editor asked me to write an article on the classical and modern Javanese

literature. I obliged.

"How could you, since you avow to be ignorant of the language, let alone the literature?" asked Castro.

"That was not very difficult. I started with a detailed description of the island of Java. Dictionaries and geography textbooks provided me with the necessary information. Then, I quoted American, German and French authors."

"Did anybody ever put your knowledge in question?" My friend asked.

"Never, although I was almost caught out on a particular occasion. The police arrested an individual, a sailor, a brown fellow who spoke a strange language. They called a number of different interpreters, but to no avail. Finally the police got in touch with me, the Javanese scholar, with all the respect my position deserved. I delayed in attending the request, but in the end I could no longer escape the compromise. Fortunately, the man had been released, thanks to the intervention by the Dutch consul to whom the sailor could explain his case through the use of half a dozen or so Dutch words. The sailor was indeed Javanese."

The moment finally arrived for my attending the congress. So, there I was on my way to Europe. It was marvelous! I attended to the opening ceremony and the invited paper presentations. The organizers registered me in the Tupi-Guarani section. After a couple of days at going from presentation to presentation, I concluded that the whole thing was not for me, and left for Paris. Before taking the train, however, I was careful enough to convince a journalist to publish my portrait and a short article about me. Thus, I had a way to prove that I was in Basel, not in Paris.

Eventually, I returned to Basel for the closing ceremony. The organizers of the congress assumed that my abscence was due to their mistake of assigning me to the Guarani section. I accepted the apology, but I still was not able to write that treatise on Javanese that I promised them.

After the congress, I paid for the publication of German and Italian translations of the article from the Basel paper. The readers of these translations offered a banquet in my honor. The banquet was presided by Senator Gorot. My contribution cost me all the money the Baron left me in his will.

The money was well spent. After all, I became a celebrity, and after six months I was appointed Consul in Havana, where I lived for many years, in order to perfect my proficiency of the Polinesian languages.

"It is fantastic," observed Castro, holding tightly to his glass of beer.

"Look! Do you know what I would like to be right now?"

"What?"

"A famous genetic or computer engineer. Let us go for it?"

"I am in."

Chapter 6

Arithmetic operations

Nia accepted the idea of using Prolog in Lisp, therefore she decided to learn Lisp. To this end, she entered the editor and pressed `C-x C-f` to create the `celsius.lisp` buffer, where she typed the program below, and saved it with `C-x C-s` (she kept the Ctrl key down and pressed x followed by s).

```
;; File: celsius.lisp
;; Comments are prefixed with semicolon
;; Some people use two semicolons to
;; make comments more visible.

(defun c2f(x)
(- (/ (* (+ x 40) 9) 5.0) 40)
);;end defun

(defun f2c(x)
(- (/ (* (+ x 40) 5.0) 9) 40)
);;end defun
```

Function `c2f` converts Celsius readings to Fahrenheit. The `defun` macro, which defines a function, has four components, the function id, a parameter list, an optional documentation, and a body that contains expressions and commands to carry out the desired calculation. Comments are prefixed with a semicolon. In the case of `c2f`, the id is `c2f`, the parameter list is `(x)`, and the body is `(- (/ (* (+ x 40) 9) 5.0) 40)`.

Lisp programmers prefer the prefix notation: open parentheses, operation, arguments, close parentheses. Therefore, in `c2f`, `(+ x 40)` adds `x` to 40, and `(* (+ x 40) 9)` multiplies $x + 40$ by 9.

In order to perform a few tests, Nia must initially save the program with `C-x C-s`. Then she presses `C-x 2` to create a window for interacting with

61

Lisp. After this action, there are two windows on the screen, both showing
the `celsius.lisp` buffer. The `C-x o` command makes the cursor jump from
one window to the other. From the bottom window, Nia presses `M-x shell`
to startup the shell (she keeps the [Alt] key down and presses `x`, then the
types `shell` at the `M-x` prompt in the minibuffer). At the shell, Nia launch
Common Lisp through roswell:

```
~/wamcompiler/wamdocs/lisp-src master ×
> ros run
* (* 3 4 5)
60
* (* 3 4 5)
60
* (load "celsius.lisp")
T
* (c2f 100)
212.0
* (f2c 212)
100.0
```

From the Read Eval Print Loop, which we will call simply REPL from
now on, Nia can load the `celsius.lisp` program and call any application
that she has defined herself, or which comes with Lisp. The first operation
in the REPL is loading the `celsius.lisp` file, as you can see in figure 6.1.

```
;; To compile: (load "celsius.lisp")

(defun c2f(x)
(- (/ (* (+ x 40) 9) 5.0) 40)
);;end defun

(defun f2c(x)
(- (/ (* (+ x 40) 5.0) 9) 40)
);;end defun
```
```
* (c2f 100)
212.0
```
```
> (load "celsius.scm")
```

Listing 6.1: Source file and REPL

Let us recall what happened until now. Nia pressed `C-x 2` to split the
buffer window, as shown in figure 6.1. Thus she has two buffers in front of

her, each with its own window. One of the buffers has a Lisp source file, while the other contains a Lisp interaction. In order to switch from one buffer to the other, Nia press the `C-x o` command. Nia is a smart girl, who cultivates the habit of saving the buffer with `C-x C-s` after any modification.

Nia often works with a single window. In this case, after typing and saving the source file with `C-x C-s`, she does not split the window before opening the interaction buffer with the `M-x shell` command. In this case, only one buffer is visible. Nia switches between the interaction buffer and the source file by pressing the `C-x b` command, and using the up and down arrow to choose the destination buffer.

Text editing is one of these things that are easier to do than to explain. Therefore, the reader is advised to learn Emacs by experimenting with the commands.

6.1 let-binding

The let-form introduces local variables through a list of (`id value`) pairs. In the basic let-binding, a variable cannot depend on previous values that appear in the local list. In the `let*` (let star) binding, one can use previous bindings to calculate the value of a variable, as one can see in figure 6.2. When using let binding, the programmer must bear in mind that it needs parentheses for grouping together the set of (`id value`) pairs, and also parentheses for each pair. Therefore, one must open two parentheses in front of the first pair, one for the list of pairs and the other for the first pair.

You certainly know that the word 'December' means the tenth month; it comes from the Latin word for ten, as attested in words such as:

- *decimal* – base ten.

- *decimeter* – tenth part of a meter.

- *decimate* – the killing of every tenth Roman soldier that performed badly in battle.

October is named after the Latin numeral for *eighth*. In fact, the radical OCT- can be translated as *eight*, like in *octave*, *octal*, and *octagon*. One could continue with this analysis, placing September in the seventh, and November in the nineth position in the sequence of months. But everybody and his brother know that December is the twelfth, and that October is the tenth month of the year. Why did the Romans give misleading names to these months?

Rome was purportedly founded by Romulus, who designed a calendar. March was the first month of the year in the Romulus calendar. Therefore, if Nia wants the order for a given month in this mythical calendar, she must subtract 2 from the order in the Gregorian calendar. In doing so, September becomes the seventh month; October, the eigth; November, the nineth and December, therefore, the tenth month.

Farming and plunder were the main occupations of the Romans, and since winter is not the ideal season for either of these activities, the Romans did not care much for January and February. However, Sosigenes of Alexandria, at the request of Cleopatra and Julius Cæsar, designed the Julian calendar, where January and February, the two deep winter months, appear in positions 1 and 2 respectively. Therefore, a need for a formula that converts from the modern month numbering to the old sequence has arisen.

```lisp
(defun roman-order(m) (+ m (* (winter m) 12) -2))
```

The above formula will work if `(winter m)` returns 1 for months 1 and 2, and 0 for months from 3 to 12. The definition below satisfies these requisites.

```lisp
(defun winter(m)  (floo (- 14 m) 12))
```

```lisp
(defun winter(m)  (floor (- 14 m) 12))

(defun roman-order(m)  (+ m (* (winter m) 12) -2))

(defun zr(y m day)
  (let* ( (roman-month (roman-order m))
          (roman-year (- y (winter m)))
          (century (floor roman-year 100))
          (decade (mod roman-year 100)) )
     (mod (+ day decade
             (floor (- (* 13 roman-month) 1) 5)
             (floor decade 4)
             (floor century 4)
             (* century -2)) 7)) )
```
```
> (load "zeller.lisp")
T
> (zr 2016 8 31)
3
```

Listing 6.2: Source file and REPL

For months from 3 to 12, `(- 14 m)` produces a number less than 12, and `(floor (- 14 m) 12)` is equal to 0, since `floor` is a strange name for the integer division. Therefore, `(* (winter m) 12)` is equal to 0 for all months from March through December, and the conversion formula reduces to `(+ m -2)`, which means that March will become the Roman month 1, and December will become the Roman month 10. However, since January is month 1, and February is month 2 in the Gregorian calendar, `(winter m)` is equal to 1 for these two months. In the case of month 1, the Roman order is given by `(+ 1 (* 1 12) -2)`, that is equal to 11. For month 2, one has that `(+ 2 (* (winter 2) 12) -2)` is equal to 12.

The program of figure 6.2 calculates the day of the week through Zeller's congruence. In the definition of `(zr y m day)`, the let-star binds values to the local variables `roman-month`, `roman-year`, `century` and `decade`.

Figure 6.2 shows that the let binding avoids recalculating values that appear more than once in a program. This is the case of `roman-year`, that appears four times in the `zr` procedure. Besides this, by identifying important subexpressions with meaningful names, the program becomes clearer and cleaner. After loading the program of figure 6.2, expressions such as `(zr 2016 8 31)` return a number between 0 and 6, corresponding to Sunday, Monday, Tuesday, Wednesday, Thursday, Friday and Saturday.

6.2 True or False

So far, the only tool that Nia has used for programming is combination of functions. However, this is not enough to write programs that reach a decision or make a choice. Lisp has expressions to say "if this is true, do one thing, else do another thing". Nia could have used one of these decision making expressions to calculate the roman order:

```
(if (< m 3) (+ m 10) (- m 2))
```

The if-else expression says: if the Julian numbering `m` is greater than 2, then subtract 2 from `m`, else add 10 to `m`. The `if-else` expression is much simpler to understand than all that confusing talk about deep winter months.

Besides correcting the month, the `(winter m)` expression was used to correct the year. If the month `m` is 1 or 2, one must subtract 1 from the Gregorian year in order to obtain the Roman year.

In Lisp, *Boolean* is a data type that can have just two values: `T` for "true" and `nil` for "false". The procedure `(< m 3)` returns `T` if `m` is less than 3, and `nil` otherwise. On the other hand, `(if (< m 3) (+ m 10) (- m 2))`

calculates the expression `(+ m 10)` if and only if `(< m 3)` produces the value
`T`. Otherwise, the `if` form calculates the `(- m 2)` expression.

Now, let us analyze the terms of the Zeller's congruence. A normal year
has 365 days and 52 weeks. Since 365 is equal to `(+ (* 7 52) 1)`, each
normal year advances the day of the week by 1. Therefore, the formula has
a `decade` component. Every four years February has an extra day, so it is
necessary to add `(floor decade 4)` to the formula, where `floor` plays the
role of integer division.

A century contains 100 years. Therefore, one would expect 25 leap years
in a century. But since turn of the century years, such as 1900, are not leap
years, the number of leap years in a century reduces to 24. So, in a century
the day of the week advances by 124. But `(mod 124 7)` produces 5, that is
equal to `(- 7 2)`. Then one needs to subtract 2 for each century. This is
done through the term `(* century -2)`.

But every fourth century year is a leap year. Therefore, Zeller needed to
add the term `(floor century 4)` to the formula.

Starting from March, each month has 2 or 3 days beyond 28, that is the
approximate duration of a lunar month: 3, 2, 3, 2, 3, 3, 2, 3, 2, 3, 3, 0. Nia
noticed that definition

```
(defun acc(i) (- (floor (- (* 13 i) 1) 5) 2))
```

returns the accumulated month contribution to the week day. However, the
day chosen to start the week cycle is irrelevant. Therefore, there is no need
to subtract 2 from the accumulator.

```
(defun zeller(y m day)
  (let* ( (roman-month (if (< m 3) (+ m 10) (- m 2)))
          (roman-year (if (< m 3) (- y 1) y))
          (century (floor roman-year 100))
          (decade (mod roman-year 100)) )
    (mod (+ day decade
            (floor (- (* 13 roman-month) 1) 5)
            (floor decade 4)
            (floor century 4)
            (* century -2)) 7)) )
* (load "wday.lisp")
T
* (zeller 2016 8 31)
3
```

Listing 6.3: Making decisions

Chapter 7

Sets

A set is a collection of things. You certainly know that collections do not have repeated items. I mean, if a guy or gall has a collection of stickers, s/he does not want to have two copies of the same sticker in his/her collection. If s/he has a repeated item, s/he will trade it for another item that s/he lacks in his/her collection.

It is possible that if your father has a collection of Greek coins, he will willingly accept another drachma in his set. However, the two drachmas are not exactly equal; one of them may be older than the other.

7.1   Sets of numbers

Mathematicians collect other things, besides coins, stamps, and slide rules. They collect numbers, for instance; therefore you are supposed to learn a lot about sets of numbers.

$\mathbb{N}$ **is the set of natural integers.** Here is how mathematicians write the elements of $\mathbb{N}$: $\{0, 1, 2, 3, 4 \ldots\}$.

$\mathbb{Z}$ **is the set of integers, i.e.,** $\mathbb{Z} = \{\ldots -3, -2, -1, 0, 1, 2, 3, 4 \ldots\}$.

Why is the set of integers represented by the letter $\mathbb{Z}$? I do not know, but I can make an educated guess. The set theory was discovered by Georg Ferdinand Ludwig Philipp Cantor, a Russian whose parents were Danish, but who wrote his *Mengenlehre* in German! In German, integers may have some strange name like *Zahlen*.

You may think that set theory is boring; however, many people think that it is quite interesting. For instance, there is an Argentinean that scholars consider to be the greatest writer that lived after the fall of Greek civilization. In

few words, only the Greeks could put forward a better author. You probably heard Chileans saying that Argentineans are somewhat conceited. *You know what is the best possible deal? It is to pay a fair price for Argentineans, and resell them at what they think is their worth.* However, notwithstanding the opinion of the Chileans, Jorge Luiz Borges is the greatest writer who wrote in a language different from Greek. Do you know what was his favorite subject? It was the Set Theory, or *Der Mengenlehre*, as he liked to call it.

When a mathematician wants to say that an element is a member of a set, he writes something like

$$3 \in \mathbb{Z}$$

If he wants to say that something is not an element of a set, for instance, if he wants to state that -3 is not an element of $\mathbb{N}$, he writes:

$$-3 \notin \mathbb{N}$$

Let us summarize the notation that Algebra teachers use, when they explain set theory to their students.

Vertical bar. The weird notation $\{x^2 | x \in \mathbb{N}\}$ represents the set of x^2, *such that* x is a member of $\mathbb{N}$, or else, $\{0, 1, 4, 9, 16, 25 \dots\}$

Conjunction. In Mathematics, you can use a symbol $\wedge$ to say **and**; therefore $x > 2 \wedge x < 5$ means `(and (> x 2) (< x 5))` in Lisp notation.

Disjunction. The expression $(x < 2) \vee (x > 5)$ means `(or (< x 2) (> x 5))` in Lisp notation

Using the above notation, you can define the set of rational numbers:

$$\mathbb{Q} = \{\frac{p}{q} | p \in \mathbb{Z} \wedge q \in \mathbb{Z} \wedge q \neq 0\}$$

In informal English, this expression means that a rational number is a fraction

$$\frac{p}{q}$$

such that p is a member of $\mathbb{Z}$ and q is also a member of $\mathbb{Z}$, submitted to the condition that q is not equal to 0.

In Lisp, one can represent a rational number as a Cartesian pair of integers. The expression `(cons p q)`, where `p` and `q` are integers, builds such a pair. For instance, `(cons 5 3)` produces the `'(5 . 3)` pair. N.B. there are spaces before and after the dot.

```
(defparameter f (cons 2 3))

(defparameter g (cons 4 5))

(defun add(x y)
   (let ( (numerator (+ (* (car x) (cdr y))
                        (* (car y) (cdr x)) ))
          (denominator  (* (cdr x) (cdr y)) ))
      (cons numerator denominator)))
```
```
* (load "cartesian.lisp")
T
* (add f g)
(22 . 15)
* (add '(3 . 7) '(2 . 5))
(29 . 35)
```

Listing 7.1: Cartesian pairs

In the dotted pair notation of a rational number, the `(car '(5 . 3))` operation retrieves the first element of the pair, i.e., 5. On the other hand, the `(cdr '(5 . 3))` operation returns the second element, which is 3.

Let us assume that Nia wants to add two rational numbers. From elementary arithmetic, Nia knows that the addition of two rational numbers P and Q is given by the following expression:

$$\frac{X_a}{X_d} + \frac{Y_a}{Y_d} = \frac{X_a \times Y_d + Y_a \times X_d}{X_d \times Y_d} \tag{7.1}$$

In the dotted pair notation, one has $X_a=$ `(car X)`, $X_d =$ `(cdr X)`, $Y_a=$ `(car Y)` and $Y_d=$ `(cdr Y)`. By replacing these values in equation 7.1, one arrives at the following expressions for the numerator and the denominator of the addition:

```
(+ (* (car x) (cdr y))     ;; numerator
   (* (car y) (cdr x)))

(* (cdr x) (cdr y))        ;; denominator
```

Nia used the above expressions for calculating the numerator and denominator of $X + Y$ in listing 7.1.

7.2   Irrational Numbers

At Pythagora's time, the Greeks claimed that any pair of line segments is commensurable, i.e., you can always find a meter, such that the lengths of any two segments are given by integers. The following example will show the Greek theory of commensurable lengths at work. Consider the square that the Greek in the figure at the bottom of this page is evaluating.

   If the Greeks were right, I would be able to find a meter, possibly a very small one, that produces an integer measure for the diagonal of the square, and another integer measure for the side. Suppose that p is the result of measuring the side of the square, and q is the result of measuring the diagonal. The Pythagorean theorem states that $\overline{AC}^2 + \overline{CB}^2 = \overline{AB}^2$, i.e.,

$$p^2 + p^2 = q^2 \therefore 2p^2 = q^2 \tag{7.2}$$

   You can also choose the meter so that p and q have no common factors. For instance, if both p and q were divisible by 2, you could double the length of the meter, getting values no longer divisible by 2. E.g. if $p = 20$ and $q = 18$, and you double the length of the meter, you get $p = 10$, and $q = 9$. Thus let us assume that one has chosen a meter so that p and q are not simultaneously even. But from equation 7.2, one has that q^2 is even. But if q^2 is even, q is even too. You can check that the square of an odd number is always an odd number. Since q is even, you can substitute $2 \times n$ for it in equation 7.2.
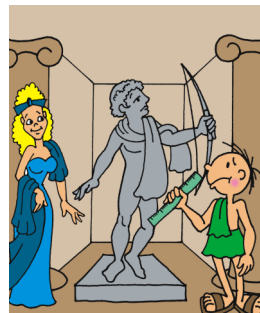
$$2 \times p^2 = q^2 = (2 \times n)^2 = 4 \times n^2 \therefore 2 \times p^2 = 4 \times q^2 \therefore p^2 = 2 \times n^2 \tag{7.3}$$
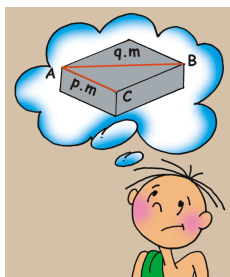
Equation 7.2 shows that q is even; equation 7.3 proves that p is even too. But this is against our assumption that p and q are not both even. Therefore, p and q cannot be integers in equation 7.2, which you can rewrite as

$$\frac{p}{q} = \sqrt{2}$$

The number $\sqrt{2}$, that gives the ratio between the side and the diagonal of any square, is not an element of $\mathbb{Q}$, or else, $\sqrt{2} \notin \mathbb{Q}$. It was Hypasus of Metapontum, a Greek philosopher, who proved this. The Greeks were a people of wise

men and women. Nevertheless they had the strange habit of consulting with an illiterate peasant girl at Delphi, before doing anything useful. Keeping with this tradition, Hyppasus asked the Delphian priestess— the illiterate girl— what he should do to please Apollo. She told him to measure the side and the diagonal of the god's square altar using the same meter. By proving that the problem was impossible, Hypasus discovered a type of number that can not be written as a fraction. This kind of number is called irrational.

An irrational number is not a crazy, or a stupid number; it is simply a number that you cannot represent as *ratione* (fraction, in Latin).

The set of all numbers, integer, irrational, and rational is called $\mathbb{R}$, or the set of real numbers.

Computers are not able to deal with sets that hold a transfinite number of elements. Therefore, Lisp and all other programming languages replace sets with the concept of type. In Lisp, $\mathbb{Z}$ is called **integer**, although the **integer** type does not cover all integers, but enough of them to satisfy your needs. Likewise, a floating point number belongs to the **number** type, which is a subset of $\mathbb{R}$.

If $x \in \mathbf{Int}$, Lisp programmers say that x has type **integer**. They also say that r has type **number** if $r \in \mathbf{Real}$.

In Lisp, rational and irrational numbers are inexact data types. Here are a few functions that produce inexact results:

(+ x_1 x_2 ... x_n) – addition
(* x_1 x_2 ... x_n) – multiplication
(- x_1 x_2 ... x_n) – subtraction
(/ x_1 x_2 ... x_n) – division
(pow x y) – x^y
(sin x) – $\sin(x)$
(asin x) – $\arcsin(x)$
(cos x) – $\cos(x)$
(acos x) – $\arcsin(x)$
(tan x) – $\tan(x)$
(atan x) – $\arctan(x)$
(log x) – natural logarithm
(log xb) – logarithm to the base b

Integers are distinguished from inexact numbers by the (integerp x) predicate. Two important integer functions are (floor x y) and (mod x y). The first function produces the integer division of x by y, while the other returns the remainder of the division. You should remember having used these two functions earlier to calculate the Zeller's congruence.

7.3   Data types

There are other types besides **integer**, and **number**. The `class-of` function provides the type of any object:

```
* (class-of 10)
#<BUILT-IN-CLASS COMMON-LISP:FIXNUM>
* (class-of 10.0)
#<BUILT-IN-CLASS COMMON-LISP:SINGLE-FLOAT>
```

Here is a list of primitive types:

integer — Integer numbers between -2147483648 and 2147483647. The `(integerp x)` function returns `T` if `x` is an integer, and `nil` otherwise.

float — represents rational and irrational numbers, albeit approximately. The `(floatp x)` function returns `T` for floating point numbers.

number — The `(numberp x)` function returns `T` for any type of numbers, and `nil` for non-numeric objects.

string — A quoted string of characters: `"3.12"`, `"Hippasus"`, `"pen"`, etc. The `(stringp x)` function answers `T` if `x` is a string, and `nil` otherwise. A few important functions that operate on strings are:

- `(length s)` returns the number of chars of the `s` string
- `(subseq "Hello, World" 2 5)` operation returns `"llo"`

char — Chars have the `#\` prefix: `#\A`, `#\b`, `#\3`, `#\Space`, `#\Newline`, etc. The `(characterp x)` function returns `T` when `x` is a char. Let `s` be a string such as `"Hello"`. Then, `(aref "Hello" 0)` returns the first char of `s`, `(aref "Hello" 1)` returns the second char, and so on. One can compare chars with the following functions:

```
(char= x #\g) – T if x= #\g
(char> x #\g) – T if x comes after the #\g char.
(char< x #\g) – T if x comes before the #\g char.
(char>= x #\g) – T if x is equal to #\g or comes after it.
(char<= x #\g) – T if x is equal to #\g or comes before it.
```

cons — The cons data type can represent both Cartesian pairs and lists; `(consp x)` returns `T` if `x` is a cons, otherwise it returns `nil`. The `()` empty list is often used as the second element of a cons pair to form lists. The `(null x)` returns `T` when `x` is the `()` empty list.

7.4 Functions

A function is a relationship between an argument and a unique value. Let the argument be $x \in B$, where B is a set; then B is called domain of the function. Let the value be $f(x) \in C$, where C is also a set; then C is the range of the function. Functions can be represented by tables, or clauses. Let us examine each one of these representations in turn.

Tables

Let us consider a function that associates `T` (*true*) or `nil` (*False*) to the letters of the Roman alphabet. If a letter is a vowel, then the value will be `T`; otherwise, it will be `nil`. The range of such a function is `{T, nil}`, and the domain is $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$.

Domain	Range	Domain	Range	Domain	Range	Domain	Range
a	T	g	nil	m	nil	t	nil
b	nil	h	nil	n	nil	u	T
c	nil	i	T	o	T	v	nil
d	nil	j	nil	p	nil	w	nil
e	T	k	nil	q	nil	x	nil
f	nil	l	nil	r	nil	y	nil
				s	nil	z	nil

7.4.1 Clauses

From the last section, you certainly noted that it is pretty tough to represent a function using a table. You must list every case. There are also functions, like (sin x), whose domain has an infinite number of values, which makes it impossible to list all entries. Even if you were to try to insert only a finite subset of the domain into the table, it wouldn't be easy. Notwithstanding, in the past, people used to build tables. In fact, tables were the only way to calculate many useful functions, like (sin x), (log x), (cos x), etc. In 1814 Barlow published his Tables which give factors, squares, cubes, square roots, reciprocals and hyperbolic logs of all numbers from 1 to 10000.

In 1631, Briggs published tables of sin functions to 15 places and tan and sec functions to 10 places. I heard the story of a mathematician who published a table of sinus, and made a mistake. Troubled by the fact that around a hundred sailors lost their way due to his mistake, that mathematician committed suicide. This story shows that the use of tables may be hazardous to your health.

In order to understand how to represent a function with clauses, let us revisit the vowel table. Using clauses, that table becomes

```
(defun vowelp(x)
   (cond ( (or (char= x #\a)(char= x #\e)(char= x #\i)
               (char= x #\o)(char= x #\u)) T)
         (T nil)))
```

Expressions like (or p_1 $p_2 \ldots p_n$) has the same meaning as $p_1 \vee p_2 \ldots \vee p_n$.

Functions have a parameter list that represents an element of the domain. Below the name of the function, and its parameter list, one finds a set of clauses. Each clause has a condition followed by an expression, that gives the value, if that clause applies. Consider the first clause:

```
(or (char= x #\a)(char= x #\e)(char= x #\i)
            (char= x #\o)(char= x #\u)) T)
```

asks the question: *Is* x= #\a*,* x= #\a*,* x= #\a*,* x= #\a *or* x= #\a*?* If the answer is yes, the clause value is T (last expression of the clause).

Now, let us consider the Fibonacci function, that has such an important role in the book "The Da Vinci Code". Here is its table for the first 9 entries:

0	1		3	5		6	21
1	2		4	8		7	34
2	3		5	13		8	55

Notice that a given functional value is equal to $f_n = f_{n-3} + 2 \times f_{n-2}$. Assume that $n = 6$. Then, $f_6 = f_3 + 2 \times f_4$.

Of course, the expression $f_n = f_{n-3} + 2 \times f_{n-2}$ is true only for $n > 2$, since there are not three precedent values for $n = 0$, $n = 1$ and $n = 2$. The below program shows how you can state that a rule is valid only under certain conditions.

```
(defun fib(n)
   (cond ( (< n 2) (+ n 1))
         ( (= n 2) 3)
         (T  (+ (fib (- n 3))
                (* (fib (- n 2)) 2)) )))
```
```
* (load "fibonacci")
T
* (fib 5)
13
* (fib 50)
32951280099
```

Predicates. A predicate is a function that gives true and false for output. In Lisp, the only false value is `nil`, but any value that is different from `nil` is considered true. A predicate can be used to discover whether a property is true for a given value. For instance, a sequence of elements such as `'(S P Q R)` is called list. There is also an empty list, that has no elements at all, and is represented by `()` or by `nil`. Let `xs` be a list. Then, the predicate `(null xs)` returns `T` if `xs` is the empty list.

There are predicates designed for performing comparisons. For instance, `(> m 2)` returns `T` for `m` greater than 2, and `nil` otherwise. Here is a more or less complete list of comparison predicates:

- `(> m n)` – `T` for `m` greater than `n`.

- `(< m n)` – `T` for `m` less than `n`.

- `(>= m n)` – `T` for `m` greater or equal to `n`.

- `(<= m n)` – `T` for `m` less or equal to `n`.

- `(= m n)` – `T` if `m` is a number equal to `n`.

The inputs `m` and `n` must be both numbers, if you want the above predicates to work.

A string is a sequence of characters represented between double quotation marks. For instance, `"Sunday"` is a string. Below, there is a short list of string predicates.

- `(string= s z)` – `T` if `s` and `z` are equal.

- `(string> s z)` – `T` if `s` comes after `z` in the alphabetical order.

- `(string< s z)` – `T` if `s` comes before `z` in the alphabetical order.

- `(string<= s z)` – `T` if `s` precedes or is equal to `z`.

- `(string>= s z)` – `T` if `s` follows or is equal to `z`.

The prefix `#\` identifies isolated characters. Let us assume the following definition:

```
(defparameter s "Nia Vardalos")
```

Then, the chars `#\N` `#\i` `#\a` and `#\Space` were retrieved by the expressions `(aref s 0)`, `(aref s 1)`, `(aref s 2)` and `(aref s 3)` respectively.

The blank space character, control characters, non-graphic characters and all other non-printable characters can be represented by identifiers, such as `#\space`, `#\tab` and `#\newline`.

Character have their own set of predicates, as one should expect.

- (char= #\A #\a) – is nil since #\A and #\a are not equal.

- (char> #\z #\a) – is T since #\z comes after #\a in the alphabetical order.

- (char< #\a #\z) – is T since #\a comes before #\z in the alphabetical order.

- (char<= #\a #\c) – is T since the order of #\a is less or equal to the order of #\c.

- (char>= #\a #\c) – nil since the order of #\a greater or equal to the order of #\c.

Lisp often uses special forms to make decisions. The (and p_1 $p_2 \ldots p_n$) form evaluates the sequence p_1, $p_2 \ldots p_n$ until it reaches p_n or one of the previous p_i returns nil. In fewer words, the and-form stops at the first p_i that returns nil and, if no argument is false, it stops at p_n. The form returns the value of the last p_i that it evaluates. The and-form returns a value different from nil if and only if all p_i predicates produce values different from nil (false).

Like the and-form, the or-form also stops as soon as it can. In the case of the or-form, this means returning true as soon as any of the arguments is true. Remember that true is anything different from nil. The (or p_1 $p_2 \ldots p_n$) form returns the value of the first true p_i predicate.

The (not P) form returns T if P produces nil, and evaluates to nil if P is true, i.e., anything different from nil.

With the and, or and not forms, Nia can combine primitive functions to define many interesting predicates. For instance, the predicate (digitp d) determines whether d is a digit.

```
(defun digitp(d)
    (and (char>= d #\0)
         (char<= d #\9)))
```

The predicate deep-winterp checks if m is 1 or 2:

```
(defun deep-winterp(m)
    (or (= m 1) (= m 2)))
```

7.5 The cond-form

Now that Nia knows how to find an integer between 0 and 6 for the day of the week, she needs a function that produces the corresponding name.

```
(defun weekday(n)
  (cond ( (= n 0) 'Sunday)
     ( (= n 1) 'Monday)
     ( (= n 2) 'Tuesday)
     ( (= n 3) 'Wednesday)
     ( (= n 4) 'Thursday)
     ( (= n 5) 'Friday)
     ( T 'Saturday)))

(defun zeller(y m day)
  (let* ( (roman-month (if (< m 3) (+ m 10) (- m 2)))
          (roman-year (if (< m 3) (- y 1) y))
          (century (floor roman-year 100))
          (decade (mod roman-year 100)) )
    (weekday (mod (+ day decade
                     (floor (- (* 13 roman-month) 1) 5)
                     (floor decade 4)
                     (floor century 4)
                     (* century -2)) 7)) ))
> (load "zel.lisp")
T
> (zeller 2018 8 31)
Friday
```

Listing 7.2: Day of the week

The cond-form controls conditional execution, based on a set of clauses. Each clause has a condition followed by a sequence of actions. Lisp starts with the top clause, and proceeds in decending order. It executes the first clause whose condition produces a value different from `nil`. For instance, the first clause condition is the (= n 0) predicate. If the predicate (= n 0) returns T for the Sunday index, the function `weekday` returns `'Sunday`. If n= 1, the second condition holds, and the function produces the symbol `'Monday`. And so on.

7.6   Lists

In Lisp, there is a data structure called list, that uses parentheses to represent nested sequences of objects. One can use lists to represent structured data. For instance, in the snippet below, `xor-structure` shows the structure of a combinatorial circuit through nested lists.

```
(defparameter *x* 42)

(setf  (symbol-function 'xor)
  (lambda (A B)
     (or (and A (not B))
         (and (not A) B)) ))

(defparameter xor-structure
  '(or (and A
           (not B))
      (and (not A)
           B)))
```

Although Nia does not know what a combinatorial circuit is, she noticed that the structure defined as `xor-structure` is prefixed by a single quotation mark, that in Lisp parlance is known as quote. Since Common Lisp uses the same notation for programs and structured data, the computer needs a tag to set data apart from code. Therefore, quote was chosen to indicate that an object is a list, not a procedure that needs to be executed by the computer.

The `defparamete` form creates global variables. In the above source code, the variable `*x*` is an id for the number 42, while `xor` is the id for a program that calculates the output of a two input exclusive or gate. Of course, Nia could define the `xor` gate as shown below:

```
(define *x* 42)

(defun xor(A B)
  (or (and A (not B))
      (and (not A) B)) )

(defparameter xor-structure
  '(or (and A
           (not B))
      (and (not A)
           B)))
```

From the examples, Nia discovered that there are two ways of defining the `xor` gate as a combination of `A` input, `B` input, `and` gate, `or` gate and `not` gate. In the first and most popular style, one uses the `(defun xor(A B) ...)` format. The advantage of this method is that it spares one nesting level.

In the second style of defining functions and gates, the abstraction appears explicitly in the defining expression:

```
(setf  (symbol-function 'xor)
   (lambda (A B)
      (or (and A (not B))
          (and (not A) B)) ))
```

This second style is quite remarkable because the abstraction that defines the operation is treated exactly like any other data type existent in the language. In fact, there is no formal difference between the definition of `*x*` as an integer constant, and `xor` as a functional abstraction.

```
;; File: list-example.lisp

(defparameter xor-structure
   '(or (and A (not B))
        (and (not A) B)))
```
```
(load "list-example.lisp") Enter
T

(car xor-structure) Enter
OR
(cdr xor-structure) Enter
((AND A (NOT B)) (AND (NOT A) B))
(car (cdr xor-structure)) Enter
(AND A (NOT B))
(car (cdr (car (cdr xor-structure)) )) Enter
A
(car (cdr (cdr (car (cdr xor-structure)) ))) Enter
(NOT B)
```

Listing 7.3: List selectors

Although Lisp represents programs and data in the same way, it does not use the same methods to deal with source code and lists. In the case of

source code, the compiler translates it into a virtual machine language that the computer can easily and efficiently process.

Data structures, such as lists, have an internal representation with parts. In particular, a list is implemented as a chain of pairs (vide page 68), each pair containing a pointer to a list element, and another pointer to the next pair. Let us assume that `xs` points to the list `'(S P Q R)`. This list corresponds to the following chain of pairs:
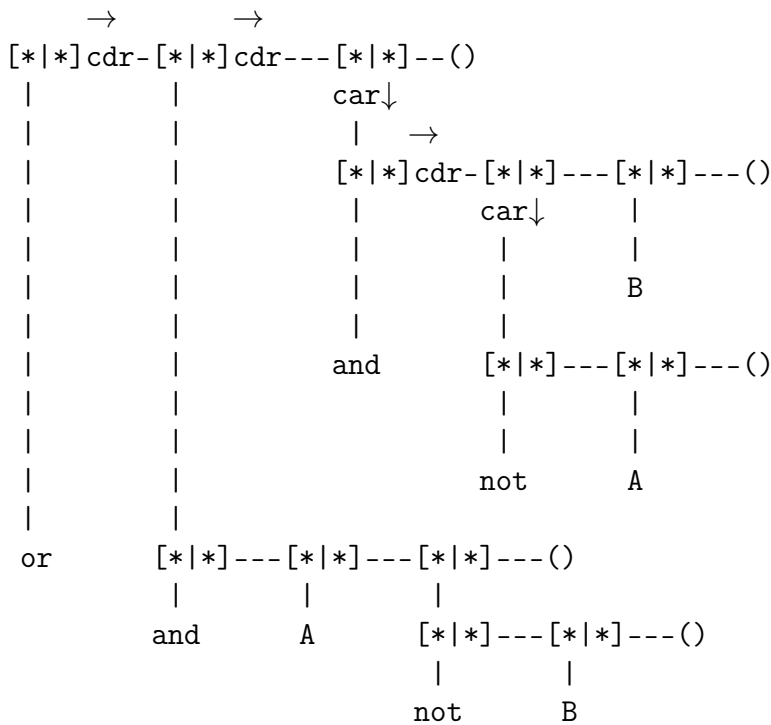
```
[*|*]--->[*|*]--->[*|*]--->[*|*]--->()
 |        |        |        |
 S        P        Q        R
```
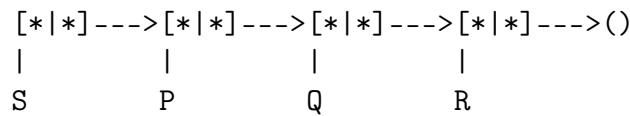
The first pair has a pointer to `S`, and another pointer to the second pair. The second pair contains pointers to `P` and to the third pair. The third pair points to `Q` and to the fourth pair. Finally, the fourth pair points to `R` and to the `()` empty list.

```
[*|*]---> next pair
 |
 S
```
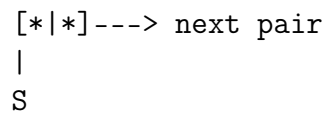
The operation `(car xs)` produces the first pointer of the `xs` chain of pairs. In the case of the `'(S P Q R)` list, `(car xs)` returns `S`. On the other hand, `(cdr xs)` yields the pair after the one pointed out by `xs`, i.e., `'(P Q R)`. A sequence of `cdr` applications permits the user to go through the pairs of a list.

```
(defparameter spqr
    '(S P Q R))
```
```
(load "spqr.lisp") Enter
(S P Q R)

(cdr spqr)
(P Q R)

(cdr (cdr spqr))
(Q R)

(cdr (cdr (cdr spqr)))
(R)
```

If all one needs is to represent sequences, then contiguous memory elements could be more practical than pairs. However, as one can see in figure 7.3, a list element can be a nested sublist. In this case, the `car` part of a pair can point down to a sublist branch. The diagram below shows that one can reach any part of a nested list following a chain of `car` and `cdr`. In such a chain, the `cdr` operation advances one pair along the list, and the `car` operation goes down into a sublist. The `cdr` operation is equivalent to a right → arrow, while the `car` operation is acts like a down ↓ arrow.

```
          →           →
   [*|*]cdr-[*|*]cdr---[*|*]--()
     |        |        car↓
     |        |         |    →
     |        |        [*|*]cdr-[*|*]---[*|*]---()
     |        |         |       car↓        |
     |        |         |        |          |
     |        |         |        |          B
     |        |         |        |
     |        |        and      [*|*]---[*|*]---()
     |        |                  |          |
     |        |                  |          |
     |        |                 not         A
     |        |
    or      [*|*]---[*|*]---[*|*]---()
              |       |       |
            and       A      [*|*]---[*|*]---()
                               |       |
                              not      B
```

The above example shows the chains and subchains of pairs that one uses to represent the logic circuit below:

```
(defparameter xs
   '(or
       (and  A
            (not B))
       (and (not A)
            B)))
```

Let us assume that Nia wants to retrieve the `(NOT A)` part of the circuit. Considering that each right → arrow is equivalent to a `cdr` operation, and each down ↓ arrow can be interpreted as a `car` operation, she must perform `(car (cdr (car (cdr (cdr xs)) )))` to reach the goal.

7.7   The list constructor

Since the `car` and `cdr` operations select the two parts of a pair, they are called *selectors*. Besides the two selectors, lists have a constructor: The operation (`cons x xs`) builds a pair whose first element is x, and the remaining elements are grouped in xs.

```
(cons 'and '(A B)) Enter
(AND A B)
```

One has learned previously that lists must be prefixed by the special quote operator, in order to differentiate them from programs. To make a long story short, quote prevents the evaluation of a list or symbol.

When you first heard about pairs on page 68, it was stated that the first element of a pair is separated from the second by a dot. However, the dot can be omitted if the second element is itself a Cartesian pair or the empty list. Then, the much neater `'(3 4)` list syntax is equivalent to the dotted Cartesian `'(3 . (4 . ()))` pair.

A backquote, not to be confused with quote, also prevents evaluation, but the backquote transforms the list into a template. When there appears a comma in the template, Lisp evaluates the expression following the comma and replaces it with the result. If a comma is combined with @ to produce the `,@` at-sign, then the expression following the at-sign is evaluated to create a list. This list is spliced into place in the template. Templates are specially useful for creating macros, as you will learn below.

Macros are programs that brings a more convenient notation to a standard Lisp form. The syntax of Lisp, that unifies data and programs, makes it possible to create powerful macros that implement Domain Specific Languages (DSLs), speed up coding or create new software paradigms. In fact, the macro system is one of the two features that places Lisp asunder from other languages, the other being its Mathematical Foundations. For learning macros in depth, the reader is referred to Dough Hoyte's book on the subject[4]. Here is a small example of macro:

```
(defmacro while-do(c r . bdy)
    `(do () ((not ,c) ,r) ,@bdy))
```

```
* (load "whiledo")
T
* (let ( (s ()) (i 0))
    (while-do (< i 5) s
       (setf s (cons i s) i (+ i 1)) ))
(4 3 2 1 0)
```

In the above macro definition, the `bdy` variable, which comes after a dot, groups all remaining macro parameters. The syntax of Lisp requires that a blank space is inserted before and after the dot.

The `(setf i (+ i 1))` operation destructively updates the value of the local variable `i`. For instance, if `i` is equal to 3, the value of `i` is replaced with `(+ i 1)`, what makes `i` equal to 4. On the same token, `(setf s (cons i s))` replaces the value of `s` with `(cons i s)`. Then, if s= `(2 1 0)` and i= 3, `(setf s (cons i s))` updates `s` to `(cons 3 '(2 1 0))= '(3 2 1 0)`.

Destructive updates are not considered good programming practice. The `while-do` macro is not very useful, since Lisp programmers adopt the functional style, which abhor the use of setf. A much more interesting macro is the Curry operator:

```
(defmacro curry(fn x)
   (let ( (g (gensym)))
      `(lambda(,g) (,fn ,g ,x)) ))
```

Here are a few examples of how to use the curry macro:

```
* (load "curry.lisp")
T
* (setf (symbol-function 'twice) (curry * 2))
#<FUNCTION (LAMBDA (#:G0)) {225F6FFB}>
* (twice 21)
42
* (funcall (curry + 2) 39)
41
```

In order to understand this macro, you must learn the concept of lambda abstraction. You have learned how to define functions that have names. However, Lisp allows the creation of anonymous functions. Suppose that you want to filter the elements of a list, leaving only those that are greater than a given number. You can use the following expression:

```
(remove-if-not (lambda(x) (> x 4)) '(3 2 4 1 6 9 8))
(6 9 8)
```

Of course, you could have defined a `g4` function and use it as shown below.

```
(defun g4(x) (>  x 4))
```

```
(remove-if-not #'g4 '(3 2 4 1 6 9 8))
(6 9 8)
```

This solution has a flaw: It requires a definition for every number that you
want to filter, which is simply impossible. Fortunately, the lambda abstrac-
tion comes to your rescue, since it creates a predicate on the fly for every
number that you want to filter. The Curry macro is a handier method to
define a lambda abstraction:

```
(remove-if-not (curry > 4) '(3 2 4 1 6 9 8))
(6 9 8)
```

Perhaps you are not convinced of the necessity of defining the curry macro.
After all, who needs to filter a list for elements greater than a given number?
I will try to give an answer to this question.

Let us assume that you have a list of words and needs to sort it to build
a spell checker. The sorting program is given below.

```
(defmacro curry(fn x)
   (let ( (g (gensym)))
      `(lambda(,g) (,fn ,g ,x)) ))

(defun quick-sort(s)
  (cond ((null s) s)
        ((null (cdr s)) s)
        (T (append
              (quick-sort (remove-if-not
                              (curry string< (car s))
                              (cdr s)))
              (list (car s))
              (quick-sort (remove-if-not
                              (curry string>= (car s))
                              (cdr s))) ) )))
```
```
* (load "curry.lisp")
T
* (quick-sort '("Caecilia" "Anna" "Priscilla"))
("Anna" "Caecilia" "Priscilla")
```

Listing 7.4: The filter function

In listing 7.4, the definition of `quick-sort` calls `quick-sort` itself. When
such a thing happens, computer scientists say that the function is recursive.
The `avg` function that you learned in section 2.11, page 24 is another example
of recursive function.

The `append` function appends its arguments, that are supposed to be lists. Let us assume that the `quick-sort` function was called by the following expression:

```
> (quick-sort '("g" "a" "c"  "h" "n" "b"))
```

The (`remove-if-not (curry string< (car s)) (cdr s)`) expression will filter all strings that comes before "g" and return the ("a" "c" "b"). A recursive call to `quick-sort` will sort this list, producing the first argument of `append`, to wit, ("a" "b" "c").

The second argument of `append` is given by the (`list (car s)`) expression. Since `s= '("g" "a" "c"  "h" "n" "b")`, and the `list` function builds a list from its arguments, one has (`list (car s)`)= '("g").

The third argument of `append` will be the sorted list of all strings in `s` that are greater or equal to "g", i.e., ("h" "n").

The value of the (`append '("a" "b" "c") '("g") '("h" "n")`) expression produces the final result, that is ("a" "b" "c" "g" "h" "n").

Chapter 8

Recursion

The mathematician Peano invented a very interesting axiomatic theory for natural numbers:

1. Zero is a natural number.

2. Every natural number has a successor: The successor of 0 is 1, the successor of 1 is 2, the successor of 2 is 3, and so on.

3. If a property is true for zero and, after assuming that it is true for n, you prove that it is true for n+1, then it is true for any number.

Peano's insight can be applied to many other situations. When they asked Myamoto Musashi, the famous Japanese Zen assassin, how he managed to kill a twelve year old boy protected by his mother's 37 samurais[1], he answered:

> *I defeated one of them, then I defeated the remaining 36. To defeat 36, I defeated one of them, then I defeated the remaining 35. To defeat 35, I defeated one of them, then... To defeat 2, I defeated one of them, then I defeated the other.*

A close look will show that the predicate `app/3` of Listing 8.1 acts like Musashi. The first clause of `app([],L,L)` states that appending an `[]` empty list to `L` produces `L`. The second clause of `app/3` states:

> `ap([H|T],L,[H|U]) :- ap(T,L,U).`
> The result of appending `[H|T]` to `L` is `[H|U]`,
> if the result of appending `T` to `L` is `U`.

---

[1]The boy's father had been killed by Musashi. His uncle met the the same fate. His mother hired her late husband's students to protect the child against Musashi.

In listing 8.1, the second clause of the `ap/3` predicate changes the goal repeatedly from `ap([H|T],L,[H|U])` to `:-ap(T,L,U)`, until a new subgoal unifies with the first clause and produces a solution. By the way, the `:-` symbol can be read as *if*, thus the clause `ap([H|T],L,[H|U]) :- ap(T, L, U)` means:

> `app([H|T],L,[H|U])` if `ap(T,L,U)`

Let us pick a more concrete instance of the problem. Nia evaluated `?-ap([2,3], [4,5],R)`. Here are the inference steps to solve this query:

1. The query `?-ap([2,3],[4,5],L)` matches to the `ap([H|T],L,[H|U])` head of the second clause, with `H=2`, `T=[3]` and `T=[4,5]`. The second clause changes the goal to:

   > subgoal 1 – `:-ap([3],[4,5],U)`

   after eventually finding that the subgoal 1 produces `U=[3,4,5]`, the inference engine will mount `H=2` and `U=[3,4,5]` into the `[H|U]` pattern to obtain the final result: `[2,3,4,5]`.

2. Subgoal 1 – which is `:-ap([3],[4,5],U)` – fails to unify with the first clause, but matches the second clause with `H=3`, `T=[]` and `L=[4,5]`. The second clause changes the goal again, this time to

   > subgoal 2– `:-ap([],[4,5],U)`.

   Subgoal 2 unifies with the first clause, and produces `U=[4,5]`. Then, the inference engine mount `H=3` and `U=[4,5]` into the `[H|U]` pattern of the second clause, and thus solves subgoal 1 –`ap([3],[4,5],[3,4,5])`.

3. Now that the solution of subgoal 1 is known, the answer to the original query can be found simply by mounting `H=2` and `U=[3,4,5]` into the pattern `[H|U]`, which produces `[2,3,4,5]`

```
ap([], L, L).
ap([H|T], L, [H|U]) :- ap(T, L, U).
```
```
> ?-ap([1,2,3],[4,5],Resp).
Resp = [1,2,3,4,5]
```

Listing 8.1: Append

8.1 Classifying rewrite rules

Typically a recursive definition has two kinds of clauses:

1. Trivial cases, which can be resolved using primitive operations and unification, which is a kind of pattern match.

2. General cases, which can be broken down into simpler subgoals.

Let us classify the two clauses of (`App xs s`):

`ap([],L,L)`	The first clause is certainly trivial
`ap([H|T],L,[H|U])` `:-    ap(T, L, U)`	The second clause can be broken down into simpler subgoals and operations: Appending two lists with one element removed from the first, and then inserting the element left out into the result.

8.2 Named-let

In Scheme, named-let permits the creation of functions with initialized arguments, avoiding the definition of auxiliary functions that require manual initialization. Unfortunately, Common Lisp does not have named-let, but there is nothing that a good macro cannot resolve.

```
(defmacro nlet(n letargs &rest body)
  `(labels ((,n ,(mapcar #'car letargs) ,@body))
      (,n ,@(mapcar #'cadr letargs)) ))

(defun fibo(i)
  (nlet fn+1 ( (n i) (fn 2) (fn-1 1) )
    (if (< n 2) fn (fn+1 (- n 1) (+ fn fn-1) fn)) ))
```
```
* (load "nlet.lisp")
T
* (fibo 5)
13
```

Listing 8.2: One argument Fibonacci function

The named-let binding is used mainly to loop. In this sense, it is not different from looping facilities that one can find in languages like C or Python.

However, the named-let binding has an important advantage over other loop facilities: One can give a meaningful name to Lisp loops. For instance, since the loop of figure 8.2 is used to calculate the `fn+1` iteration, Nia put the `fn+1` tag on it.

There is another way to get rid of auxiliary parameters: One can initialize the parameters of a definition with optional values, as shown in figure 8.3.

```
;; Example of optional parameters

(defun avg(s &optional (sum 0) (n 0))
  (cond ( (and (null s) (= n 0)) 0
        ( (null s) (/ sum n) )
        (T (avg (cdr s) (+ (car s) sum) (+ n 1.0)) )))
```
```
* (load "defaults.scm")
T
* (avg '(3 4 5 6))
4.5
```

Listing 8.3: Accumulator parameters

Doug Hoyte[4] gives an illuminating explanation about how the named-let works, therefore I feel in no obligation of dealing with the subject here. In any case, let us echo one of Hoyte's advices below, for ready reference:

> To understand a macro, it helps to *macroexpand* an example of use of this macro. To do that, provide a list representing this macro invocation to the `macroexpand` function. Notice that `macroexpand` will not expand nested macro invocation for you.
>
> ```
> * (macroexpand
>     '(nlet fn+1 ( (n i) (fn 2) (fn-1 1) )
>         (if (< n 2) fn
>             (fn+1 (- n 1) (+ fn fn-1) fn)) ))
> (LABELS ((FN+1 (N FN FN-1)
>             (IF (< N 2)
>                 FN
>                 (FN+1 (- N 1) (+ FN FN-1) FN))))
>     (FN+1 I 2 1))
> ```

From the example, you can see that Lisp has labels, which is a tool equivalent to named-let. However, labels has a more complex syntax, and therefore is more difficult to deploy.

8.3   Reading data from files

Procedure (`rdLines fname`), defined in figure 8.4, reads the contents of a file line by line and inserts the corresponding strings into a list. Procedure (`getcomments fname`) filters the list, leaving only comments.

```
;; File: (load "rdfile.lisp")

(defmacro nlet(n letargs &rest body)
  `(labels ((,n ,(mapcar #'car letargs) ,@body))
       (,n ,@(mapcar #'cadr letargs)) ))

(defun iscomment(s)
  (and (> (length s) 0)
       (char= (aref s 0) #\;))))

(defun rdLines(fname)
  (with-open-file (stream fname)
    (nlet next-line
       ( (line (read-line stream nil)) )
       (if (null line) nil
 (cons line
    (next-line (read-line stream nil))) )) ))

(defun getsemi(fname)
  (remove-if-not #'iscomment (rdLines fname)))
```
```
> rlwrap ros run
* (load "rdfile.lisp")
T
* (getsemi "rdfile.lisp")
(";; File: (load \"rdfile.lisp\")")
```

Listing 8.4: Lines from file

Lisp has a cleverly designed input system. The first argument of the `with-open-file` macro is a list containing a stream name and a file name. The body of the macro uses the stream to process the contents of the file. For instance, (`read-line stream nil`) reads a line from the stream. If it fails to read the line, it returns its second argument, where `nil` can be useful to stop the reading operation at the end of the file.

8.4   Writing data to files

Let us assume tha Nia needs a program that translates markdown to html, in
order to publish poems in the internet. I hope you will be able to understand
the code by yourself.

```
;; File: (load "wrthtml.lisp")
;;uncomment if nlet is not loaded yet
; (defmacro nlet(n letargs &rest body)
;     `(labels ((,n ,(mapcar #'car letargs) ,@body))
;         (,n ,@(mapcar #'cadr letargs)) ))

(defun tit(s) (and (> (length s) 0) (char= (aref s 0) #\#)))


(defun tohtml(xs fname)
  (with-open-file (out fname :direction :output
                            :if-does-not-exist :create
  :if-exists :append) ;; or :supersede
    (nlet next-line ( (s xs) )
       (cond ( (null s) T)
      ( (tit (car s)) (format out "<h2>~a</h2>~%" (car s))
        (next-line (cdr s)))
      ( T (format out "~a<br/>~%" (car s))
 (next-line (cdr s)) )) )))
```

```
> rlwrap ros run
* (load "rdfile.lisp")
T
* (load "wrthtml.lisp")
T
* (defparameter txt (rdLines "hellen.md"))
TXT
* (tohtml txt "hellen.html")
T
```

```
    # To Helen by Edgard Alan Poe

    Helen, thy beauty is to me
    Like those Necéan barks of yore,
    That gently, o'er a perfumed sea,
    The weary, way-worn wanderer bore
    To his own native shore.
```

Bibliography

[1] Harold Abelson, Gerald Jay Sussman and Julie Sussman. Structure and Interpretation of Computer Programs. The MIT Press, Second Edition, 1996.

[2] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi. How to Design Programs. Available from `http://www.htdp.org`

[3] Dorai Sitaram. Teach yourself Scheme in Fixnum Days. Available from `http://ds26gte.github.io/tyscheme/`

[4] Doug Hoyte. Let Over Lambda. Hoytech, 2008. ISBN: 978-1-4357-1275-1

[5] A. Church, A set of postulates for the foundation of logic, Annals of Mathematics, Series 2, 33:346–366 (1932).

[6] Alonzo Church. The Calculi of Lambda Conversion. Princepton University Press, 1986.

[7] R. Kent Dybvig. The Scheme Programming Language. Available from `http://www.scheme.com/tspl4/`

Index