

An introduction to Low Level Lisp

(Content not checked for grammar and spelling errors)



Eduardo Costa

Vernon Sipple

Chapter 1

Emacs commands

In the next chapter, you will learn how to install Emacs. 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  prefix, **M-** represents the  prefix and κ can be any key.

Basic commands. **C- κ** – Press and release  and  simultaneously

C-s then type some text – search a text.

C-r then type some text, – reverse search.

C-k – kill a line.

 **backspace** – 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-/ – undo.

C-v – forward page.

 – toggle overwrite mode.

    – arrows move cursor.

Control-x commands. C-x C- κ – Keep Ctrl down and press x and κ

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 c – describes a key stroke.
 M-x **shell** – shell command. After M-x, if one types **shell** at the prompt, the computer will enter the shell of an ANSI terminal. It is possible to use many other command just like the **shell** given as example.

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

- C-x κ – Press and release Ctrl and x together, then press κ
 - C-x b – switch to buffer. User arrows to choose the buffer
 - C-x C-b – list buffers.
 - C-x k – kill current buffer.
 - C-x = – char at cursor position.
 - 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 Esc key, then press the κ 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 Alt key down and press the κ key.
 - M-b – move backward one word.
 - M-f – move forward one word.
 - M-g **goto-line** – go to the line given in the minibuffer.
 - C-x → – activate the next buffer.
 - C-x ← – go to previous buffer.
 - C-Spc C-Spc – set a return position.
 - C-u C-Spc – go to return position after a long jump.

Query replace. If you press `[Esc] x` **replace-string** RETURN, type the search string and press RETURN again, then provide the replacement string, computer enters in 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 `[Enter]` key, the cursor jumps to the first occurrence 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 occurrence. 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 `[Esc] x` then **goto-line**, emacs prompts for a line number. As soon as you type the number and press the `[Enter]` key, the cursor jumps to the given line.

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 key bindings. On the other hand, the **C-h c** command prompts you for a key sequence and describes it.

Nia Vardalos. In order to get acquainted with editing, let us accompany the Canadian programmer Nia Vardalos, while she explores emacs.

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. After localizing the text, Nia presses **Enter** to interrupt the search. For abandoning an ongoing command, she may press **C-g**. To search the same text again, Nia presses **C-s C-s** twice.

To transport a region from one place to another, Nia presses **C-Spc** to start the selection process and move the cursor to select the 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 `[Esc]` key, then strike the κ key. Alternatively, she can keep the `[Alt]` key down and press the `[κ]` key.

Calculations with Lisp. Emacs offers a Lisp dialect to perform calculations and text processing. Let us assume that you want to find how many students graduate from medical schools in California. You press `[Esc][x]` and type **ielm** at the **M-x** prompt, and emacs will place you in a Lisp chat box.

```
ELISP> (+ 190 180 170 160 120 100 100 90) [Enter]
```

In the above expression, the first thing after 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. As soon as Nia presses the `[Enter]` key, emacs shows the result of the addition:

```
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.

```
[Esc][;]  
ELISP> (* 1 2 3 4 5) [Enter]  
120
```

- Division is obtained through the `/` operation.

```
[Esc][;]  
ELISP> (/ 1 2 4) [Enter]  
0.125
```

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

- Subtraction has the syntax shown below.

```
[Esc][;]  
ELISP> (- 1 2 4) [Enter]  
-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:

```
[Esc][;]  
ELISP> (/ (+ 190 180 170 160 120 100 100 90) 8) [Enter]
```

Command execution. When you press shortcut keys, femto emacs call a command written in C or in Lisp. You can get a complete list of all emacs commands by pressing the `C-h` `b` shortcut.

There are commands that are not associated to shortcut keys. For instance, the `goto-line` command is obtained by pressing `M-x` then typing `goto-line` at the minibuffer prompt. Below you will find a fairly complete list of emacs commands.

<code>backspace</code>	<code>backspace</code>
<code>C-a</code>	<code>beginning-of-line</code>
<code>C-b</code>	<code>backward-character</code>
<code>C-d</code>	<code>forward-delete-char</code>
<code>C-e</code>	<code>end-of-line</code>
<code>C-f</code>	<code>forward-character</code>
<code>C-k</code>	<code>kill-line</code>
<code>C-n</code>	<code>next-line</code>
<code>C-p</code>	<code>previous-line</code>
<code>C-r</code>	<code>search-backward</code>
<code>C-space</code>	<code>set-mark</code>
<code>C-s</code>	<code>search-forward</code>
<code>C-/</code>	<code>undo</code>
<code>C-w</code>	<code>kill-region</code>
<code>C-x 1</code>	<code>delete-other-windows</code>
<code>C-x 2</code>	<code>split-window</code>
<code>C-x C-b</code>	<code>list-buffers</code>
<code>C-x C-c</code>	<code>exit</code>
<code>C-x C-f</code>	<code>find-file</code>
<code>C-x C-<Right></code>	<code>next-buffer</code>
<code>C-x C-<Left></code>	<code>previous-buffer</code>
<code>C-x C-s</code>	<code>save-buffer</code>
<code>C-x =</code>	<code>what-cursor-position</code>
<code>C-x C-w</code>	<code>write-file</code>
<code>C-h c</code>	<code>describe-key</code>
<code>C-x i</code>	<code>insert-file</code>
<code>C-x k</code>	<code>kill-buffer</code>
<code>C-x o</code>	<code>other-window</code>
<code>M-x shell</code>	<code>shell-command</code>
<code>C-y</code>	<code>yank</code>
<code><down></code>	<code>next-line</code>
 <code>esc b</code>	 <code>backward-word</code>

esc <	beginning-of-buffer
esc w	copy-region
esc d	kill-word
esc >	end-of-buffer
esc f	forward-word
M-x goto-line	goto-line
C-h b	list-bindings
C-Spc	set-mark
esc @	mark-word
esc <	beginning-of-buffer
esc w	copy-region
esc x	execute-command
home	beginning-of-line
INS	toggle-overwrite-mode
<left>	backward-character
<right>	forward-character
<up>	previous-line

Chapter 2

Installation

A text editor, like Emacs, 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 frontends. Some frontends work with a raster graphics image, which is a dot matrix data structure that represents a grid of pixels. There are also frontends that are specialized in showing letters and other characters via an appropriate display media. The latter sort of frontend 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, emacs offers both ncurses, which is a text-based user interface, and also GUI. Search the Internet for a distribution that works in your computer, download and install it according to the instructions of the vendor.

You will also need Bigloo. Search the Internet for Bigloo scheme, download and install it, as you did in the case of Emacs.

Both Bigloo and Emacs, like any piece of software, are quite difficult to install. It is possible that you will not find a binary distribution. In this case, you will need to build the package, before performing the installation.

If computer science is not your thing, ask for help. In the United States, in the neighborhood of large universities, it is quite easy to find very capable students that will install Bigloo and Emacs for food.



2.1 Ready

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 high school:

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

Subtraction is not treated differently:

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



Fig. 2.1: Running total

This chapter contains an introduction to the Cambridge prefix notation, which is slightly different from the one you 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:

```
(+   8
    26
    85
    3)
122
```

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 separates the operands from the result, instead of drawing a line under the last operand. This syntax is used both by Bigloo, Emacs Lisp and WebAssembly. It was discovered by a logician whose name is Jan Łukasiewicz, for this reason it is called Polish prefix notation, after Łukasiewicz nationality.

The Polish notation has an interesting mathematical property: It is the simplest language that is expressive enough to represent mathematical and logical well formed formulas. Let us see the consequences of this mathematical syntax.

2.2.1 Lisp and Mathematics

Computer languages have, in general, a very complex syntax. A Python programmer, for instance, must learn many syntactical variants for calling up an expression. The function that calculates the logarithm has a prefix notation, while arithmetic operations obey infix syntax rules.

```
>>> math.log(3,4)
0.7924812503605781
>>> (3+4)*(5+6+7)*8
1008
```

Instead of accepting an arbitrary syntax, Lisp adopted symbolic expressions, which evolved from a mathematical notation proposed by Łukasiewicz in 1920. In mathematics, basic constructions and transformation rules are kept to a minimum. This LEX PARSIMONIÆ has many consequences. The first being that few rules can classify all symbolic expressions:

1. Numerals have traditional notation: 3, 3.1416, -8, etc.
2. Symbols are sequences of characters that cannot be interpreted as numbers and do not contain brackets: `sin`, `x`, `y`, `*ops*`, etc.
3. Quoted lists such as `'(a b e)` represent sequences of objects.
4. Unquoted lists such as `(log 8 2)` denote function calls.
5. Abstractions such as `(lambda(x y) (log (abs x) y))` define functions. The sequence of symbols `(x y)` is called a binding list, and the expression `(log (abs x) y)` is the body of the abstraction.

Any variable that appears in the binding list is said to be bound. Free variables occur in the body of the abstraction but not in the binding list.

Another consequence of its mathematical foundations is that Lisp has a clear and simple semantics. The rules of computation are based on a Mathematical system called the Lambda Calculus.

α -conversion Changing the name of a bound variable produces equivalent expressions. Therefore, $(\lambda(x)x)$ and $(\lambda(y)y)$ are equivalent.

β -reduction $((\lambda(x)E)v)$ can be reduced to $E[x := v]$.

Lisp has a Read Eval Print Loop (REPL) that performs β -reduction over symbolic expressions in order to simplify their forms. In the β -reduction process, the first element of a list can be considered as a function or a macro.

In Lisp, all functions follow the list syntax, without a single exception: The first element of the list is the function identifier, and the other elements are arguments. In order to start Bigloo Scheme, a dialect of Lisp, execute the command `M-x shell` from Emacs (keep the `Alt` down and press the `A` key, then type `shell`). From the terminal call the Bigloo interpreter, as shown below.

```
~/LLDocs$ bigloo -s

1:=> (expt 3 4)
81
1:=> (* (+ 3 4) (+ 5 6 7) 8)
1008
1:=> (exit)
~/LLDocs$
```

One immediate advantage of its mathematical foundations is that Lisp is unlikely to become obsolete. This means that one may expect that a language like Python or Fortran to suffer changes without backward compatibility. One can even expect that Python or Fortran would be phased out. However, Lisp code written many decades ago can be easily run in a modern computer. Besides this, since Lisp does not change, computer scientists can work on Lisp compilers for many decades, which results in fast and robust code.

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, unbeknownst 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 ellated 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 needs to 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 visit the `finance.scm` file. The text editor creates a memory buffer that mirrors the file contents. By convention, the file and the buffer have the same name.

Initially, the `finance.scm` buffer is empty, but the bookkeeper will fill it with expressions and the corresponding calculations. Here is how to start the text editor:

```
~/wrk/wasm$ emacs -nw
```

From emacs, type `M-x shell` command, then `bigloo -s` to start the interactive interpreter. Listing 2.2 shows what an interaction with the Bigloo buffer looks like.

<pre>1:=>(+ (* 3 25320) (* 6 56750) (* 3 136000))</pre> <div style="border: 1px solid black; display: inline-block; padding: 2px;">Enter</div>
824460
M-x shell

Listing 2.2: The `finance.scm` buffer

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:

$$FV = PV \times (1 + i)^n \quad (2.1)$$

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.

<pre>1:=> (* 100000.00 (expt (+ 1.0 0.025) 3))</pre>	Enter
107689.06	

Listing 2.3: Command from Bigloo interpreter

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.

Let us summarize what you have learned until now. When you open the emacs editor, there is a one line minibuffer at the bottom of the screen. If you keep the **Alt** key pressed, and strike the **x** key, the editor will prompt you for command. Type **shell** to open a terminal. Then call the Bigloo interpreter with **bigloo -s** and type prefix expressions for the computer to calculate.

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 a medical 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 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 a 8% rate over \$ 10,000.00. Press `Ctrl``o` and enter formula 2.1 in the minibuffer:

```
1:=> (* (expt (+ 1.0 0.08) 2) 10000)
11664.0000000000002
```

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

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 2.4. Start the editor from a text terminal:

```
~$ mkdir financing
~$ cd financing
~/financing$ emacs -nw fvalue.scm
```

After typing the definition shown in listing 2.4, you must issue the `Ctrl``x` `Ctrl``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 scheme, type `(load "fvalue.scm")` in the Read Eval Print Loop (REPL) of the interpreter, as shown in listing 2.4. The interpreter is called REPL because it is continuously reading, evaluating and printing symbolic expressions (`sexpr`), that is how scheme and WebAssembly expressions are called.

In listing 2.4, text between a semicolon and the end of a line is considered a comment. Therefore, `;; File: fvalue.scm` is a comment. Likewise, `;;end define` is a comment. Comments are ignored by scheme, and have the simple function of helping people.

```
;; File: fvalue.scm

(define (future-value pv i n)
  (* (expt (+ (/ i 100.0) 1) n)
     pv)
) ;;end define
```

```
~/LLDocs$ bigloo -s

1:=> (load "fvalue.scm")
fvalue.scm
1:=> (future-value 10000 8 2)
11664.000000000002
```

Listing 2.4: Future Value Program

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 scheme 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 2.5.

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

```
;; File: pmt.scm

(define (pmt p ;; present value
            i ;; interest
            n ;; number of periods
            #!optional (r (/ i 100.0))
                      (rn (expt (+ 1.0 r) n)) )
  (/ (* r p rn)
      (- rn 1)))

1:=> (load "pmt.scm")
pmt.scm
1:=> (pmt (* 10500000.00 0.7) (/ 10.0 12) (* 20 12))
70929.09091293965
```

Listing 2.5: Mortgage for a God

You certainly noticed that the `pmt` function, defined in listing 2.5, has two parameters `r` and `rn`, which the user does not need to express. Scheme itself assigns `(/ i 100.0)` to `r` and `(expt (+ 1.0 r) n)` to `rn`.

2.8 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: (**operation** $x_1 x_2 x_3 \dots x_n$). This means that Lisp commands, macros, procedures and functions can be written as a pair of parentheses enclosing an operation followed by its arguments. This syntax is also known as symbolic expression, or *sexpr* for short.

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

```
1:=> (define aList '(S P Q R))
aList
1:=> (cons 'Rome aList)
(Rome S P Q R)
1:=> 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, or *sexpr*. The single quotation mark tags the list so the computer will take it at face-value, and so not to be evaluated.

Through the repeated application of the **cons** function, one can build lists of any length by adding new elements to a core.

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

Repeated application of nested functions cause right parentheses accumulate at the tail of the *sexpr*. It is good practice in programming to group these bunches of right parentheses in easily distinguishable patterns.

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

```
1:=> (+ 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.

```
1:=> (define xs '(2 3 4 5 6.0))
(2 3 4 5 6.0)
1:=> xs
(2 3 4 5 6.0)
1:=> (car xs)
2
1:=> (cdr xs)
(3 4 5 6.0)
1:=> (car (cdr xs))
3
1:=> (cdr (cdr xs))
(4 5 6.0)
1:=> (car (cdr (cdr xs)))
4
1:=> (car (cdr (cdr (cdr xs)) ))
5
1:=> (car (cdr (cdr (cdr (cdr xs)) ) ))
6.0
```

2.9 Going through a list

In listing 2.6, 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 87. 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 properly closed at a glance. However, for the sake of the impatient reader, let us give a preview on how the `cond`-form works.

```
(define (avg s #!optional (acc 0) (n 0))
  (cond [ (and (null? s) (= n 0)) 0]
        [ (null? s) (/ acc n) ]
        [else (avg (cdr s) (+ (car s) acc) (+ n 1.0)) ] ))

1:=> (load "average.scm")
average.scm
1:=> (avg '(3 4 5 6))
4.5
```

Listing 2.6: Going through a list

The `cond`-form consists of a sequence of clauses. Each clause has two components: A condition and an expression. The value of the `cond`-form is given by the first clause that has a true condition. In the case of listing 2.6, the clauses are:

1. `[(and (null? s) (= n 0)) 0]` – The condition `(null? s)` will be true when `s` is the empty `'()` list. If the user types `(avg '())`, the `(null? s)` expression is true, and `n` is equal to 0. The condition `(and (null? s) (= n 0))` is true, and 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 only way to reach the second clause is to go through the list by repeatedly evaluating the third clause. The value of the second clause is `(/ acc n)`, which is the average of the list.
3. `[else (avg (cdr s) (+ (car s) acc) (+ n 1.0))]` – This clause will be evaluated when none of the previous conditions are true. If the user executes the `(avg '(3 4 5 6))` expression, the evaluator will visit this clause with `s = (3 4 5 6)`, `s = (4 5 6)`, `s = (5 6)` and `s = (6)`. Every time the evaluator falls into the third clause, `(cdr s)` is executed, and `s` loses one element.

The third clause. Let us follow the evaluation of `(avg '(3 4 5 6))` as it repeatedly descends through the `cond`-form conditions until it reaches the third clause.

- `(avg '(3 4 5 6))` goes to the `else`-clause, that will call:

`(avg (cdr s) (+ (car s) acc) (+ n 1))`

with `s = (3 4 5 6)`, `acc = 0` and `n = 0`. Since:

`(cdr s) = (4 5 6)`, `(+ (car s) acc) = 3` and `(+ n 1) = 1`,

the `else`-clause expression reduces to `(avg '(4 5 6) 3 1)`.

- `(avg '(4 5 6) 3 1)` goes to the `else`-clause again, that calls:

`(avg (cdr s) (+ (car s) acc) (+ n 1))`

with `s = (4 5 6)`, `acc = 3` and `n = 1`. Since:

`(cdr s) = (5 6)`, `(+ (car s) acc) = 7` and `(+ n 1) = 2`,

the `else`-clause expression reduces to `(avg '(5 6) 7 2)`.

- `(avg '(5 6) 7 2)` visits the `else`-clause once more:

`(avg (cdr s) (+ (car s) acc) (+ n 1))`

with `s = (5 6)`, `acc = 7` and `n = 2`. Since:

`(cdr s) = (6)`, `(+ (car s) acc) = 12` and `(+ n 1) = 3`,

the `else`-clause expression reduces to `(avg '(6) 12 3)`.

- `(avg '(6) 12 3)` evaluates the expression

`(avg (cdr s) (+ (car s) n) (+ n 1))`

for the fourth time, which calls `(avg '() 18 4)`.

- `(avg '() 18 4)` matches the `(null? s)` clause, that calculates the averaging `(/ acc n)` expression with `acc = 18` and `n = 4`.

Chapter 3

Shell

Nia, a Greek young woman, has an account on a well known social 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, Sanskrit, 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 
```

```
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 **n**ia 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.

Chapter 4

Might Scheme

As you will learn in the following chapters, there is an Instruction Set Architecture (ISA) for compiling code that can run in most modern browsers. Therefore, a web designer can customize the Internet pages to fit the necessity of a given client. To summarize the situation, the web page contains code that can be executed by the browser, which the client uses to view the page, and this code can add tailored material and modify the existing text in order to suit the interests of the reader. This kind of active web content is called Single Page Application, or SPA.

When Lisp was created, and Scheme standards were established, there was no safe method for sending compiled code over the Internet for execution in the client's machine. Since the evolution of the web did not keep up with the development of compilers, Scheme implementors did not have out of the box solutions for writing Single Page Applications (SPAs). As a consequence, most Scheme implementations do not have a porting to the virtual machine that people use for delivering SPAs. Thus a group of web designers created Might Scheme, a bare-bones implementation of the Scheme language that one can use to generate Single Page Applications. The idea is to discontinue the project as soon as Standard Scheme compilers offer tools for generating Single Page Applications.

Might Scheme was so named after Matthew Might, a Computer Scientist and pioneer in Precision Medicine, who laid the foundations onto which the Might Scheme was built. Might Scheme generates code for a low level language called C. Then a C compiler produces the final application that the client can execute in his or her machine.

This chapter discusses a collection of simple examples that were designed both as tests for checking whether the compiling is working correctly, and as introductory material on Might Scheme.

4.1 The letrec-form

The letrec-form introduces local identifiers and variables through a list of (id value) pairs. When using a letrec 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. Let us examine a very simple example to make things clear. Use emacs to type the program of listing 4.1.

```
;; File: ~/wasCm/s03-to-c/ids.scm

(letrec [ (str "Rose of Luxemburg")
          (chr #\c)
          (cns '(3 4 5))
          (n 42)
          (fl 42.0) ]
  (display "Check whether str is a string")
  (display (string? str))
  (display "Check whether cns is a pair")
  (display (pair? cns))
  (display "Check whether str is a chair")
  (display (char? chr))
  (display "Check whether n is integer")
  (display (integer? n))
  (display "Check whether n is float")
  (display (float? n))
  (display "Check whether fl is float")
  (display (float? fl)) )
```

Listing 4.1: A few identifiers

Before trying to understand the program of listing 4.1, let us see whether it works, since there is no point in studying buggy code.

The compilation process aims at transforming the scheme program depicted on listing 4.1 into machine code. As previously mentioned, this task will be accomplished in two steps, first translate Scheme to C, second compile the C code.

The program that translates from Might Scheme to C is written in Bigloo, therefore you will need Bigloo Scheme to compile it. From emacs, type the command `M-x shell` to open a term and startup the Bigloo interpreter (keep the `Alt` key down, then press `x`; at the `M-x` prompt, type `shell`).

From the shell, enter the Bigloo interpreter with the command `bigloo -s` as shown in listing 4.2. From Bigloo, load the `scm2c.scm` program, which will perform the translation from Might Scheme to C. The command

```
(compit "ids.scm" "ids.c")
```

will generate the C code. Below, you can see how to compile and run the C code that the `compit` command generated.

```
~/wasCm/s03-to-c$ bigloo -s

1:=> (load "scm2c.sch")
scm2c.sch
1:=> (compit "ids.scm" "ids.c")
#unspecified
1:=> (exit)
~/wasCm/s03-to-c$ gcc -O3 ids.c -o ids.x
~/wasCm/s03-to-c$ ./ids.x
Check whether str is a string
#t
Check whether cns is a pair
#t
Check whether str is a chair
#t
Check whether n is integer
#t
Check whether n is float
#f
Check whether fl is float
#t
```

Listing 4.2: Compiling and running a program

Listing 4.1 creates five ids, which are `str`, `chr`, `cns`, `n` and `fl`. The two lines below check whether `str` id represents a string.

```
(display "Check whether str is a string")
(display (string? str))
```

Since `str` is indeed a string, the result of `(display (string? str))` is `#t`, which is the Boolean value `true`.

The predicate `(pair? cns)` returns `#t` if the `cns` id is a list, otherwise it produces the Boolean value `#f` that stands for false. Later on, you will see that `(pair? x)` is a very important predicate, since it tells us whether a process reached the end of a list or not. By the way, predicate is a function that returns `#t` (true) or `#f` (false), to indicate whether a relation holds or a feature is present. Thus, `(char? x)` verifies whether `x` is a char, `(integer? i)` returns `#t`, when `i` is an integer, and `(float? f)` checks whether `f` is a floating point number.

4.2 First test of Might Scheme

The program `examples/test01.scm` shows that the value of an identifier can be a function.

```
(letrec [ (fn (lambda(x y) (*fx x x y y)))
          (cns '(3 4 5))
          (n 42)
          (fl 42.0)]
  (display (pair? cns))
  (display (integer? n))
  (display (float? fl))
  (display (fn 3 9)) )

~/wasCm/s03-to-c$ bigloo -s
1:=> (load "scm2c.sch")
1:=> (compit "examples/test01.scm" "test01.c")
1:=> (exit)
~/wasCm/s03-to-c$ gcc -O3 test01.c -o test01.x
~/wasCm/s03-to-c$ ./test01.x
#t
#t
#t
729
```

Listing 4.3: `examples/test01.scm`

In this case, the arguments of the function are introduced by a lambda form. The arguments of the `fn` function are represented by the `x` and `y` variables.

When one applies `(fn 3 9)` to the values 3 and 9, `x` is replaced by 3 and `y` is replaced by 9 in the expression `(*fx x x y y)`. The function `*fx` that

appears in this expression multiplies integer numbers. Since `x` is equal to 3 and `y` is equal to 9, one should expect that `(*fx x x y y)` produces 729 as a result.

4.3 Floating point operations

Below, you can see the `examples/test02.scm` program, which is one of the tests that the developers of Might Scheme used to verify whether the compiler is working properly.

```
~/wasCm/s03-to-c$ cat examples/test02.scm
;; Test arithmetic operators with many args
;; Floating point operations
(letrec
  [(ifun (lambda(x y)
    (if (=fx x 5)
      (/ (+ 42.0 42.0 42.0 84.0) 2.0 3.0)
      (*fx x y)) )])
  (display (ifun 5 9)) )

~/wasCm/s03-to-c$ bigloo -s

1:=> (load "scm2c.sch")
scm2c.sch
1:=> (compit "examples/test02.scm" "test02.c")
#unspecified
1:=> (exit)
~/wasCm/s03-to-c$ gcc -O3 test02.c -o test02.x
~/wasCm/s03-to-c$ ./test02.x
35.0000
```

You certainly will have noticed that Might Scheme uses a set of operators to work with integer numbers and a different set to perform floating point calculations. In the case of integers, one uses `*fx`, `+fx` and `-fx` for multiplication, addition and subtraction, respectively. The function `quot` produces the quotient of integer division, while `mod` calculates the remainder.

Besides arithmetic operations, integer numbers also have a separate set of predicates: `(=fx p q)` checks whether `p` is equal to `q`, `(>fx p q)` produces `#t` if `p` is greater than `q`, `(<fx p q)` verifies whether `p` is less than `q`, `(<=fx p q)` checks whether `p` is less or equal `q`, and finally `(>=fx p q)` checks whether `p` is greater or equal to `q`.

In the case of floating point numbers, the arithmetic calculations are performed by the following operators: `*` for multiplication, `+` for addition, `-` for subtraction and `/` for division.

4.4 Integer comparison

The `examples/test03.scm` program tests all predicates that one can use to perform comparisons between integers.

```
~/wasCm/s03-to-c$ cat examples/test03.scm
```

```
(letrec
  [ (f03 (lambda(x y) (if (<fx x 5) 42 (*fx x x y)) ))
    (f04 (lambda(x y) (if (<=fx x 5) 42 (*fx x y)) ))
    (f05 (lambda(x y) (if (>=fx x 5) 42 (*fx x y)) ))
    (f06 (lambda(x y) (if (>fx x 5) 42 (*fx x y)) ))
    (f07 (lambda(x y) (if (<=fx x 5) 42 (mod x y)) ))
    (f08 (lambda(x y) (if (>fx x 15) 42 (quot x y 2)) ))]
  (display (f03 5 9))
  (display (f04 5 9))
  (display (f05 5 9))
  (display (f06 5 9))
  (display (f07 29 6))
  (display (f08 15 3)))
```

```
~/wasCm/s03-to-c$ bigloo -s
```

```
1:=> (load "scm2c.sch")
scm2c.sch
1:=> (compit "examples/test03.scm" "test03.c")
#unspecified
1:=> (exit)
~/wasCm/s03-to-c$ gcc -O3 test03.c -o test03.x
~/wasCm/s03-to-c$ ./test03.x
225
42
42
45
5
2
```

4.5 cons pair

In mathematics and computer science, an ordered pair is a structure containing two objects. The order in which the objects appear in the pair matters, which means that the pair `(cons a b)` is different from the pair `(cons b a)`.

In order to work with pairs, one needs three functions. The first of these functions is the `cons` procedure, which builds the pair. The `(car (cons x y))` produces the first element of the pair, which is `x`, while `(cdr (cons x y))` calculates the second element. Examples `test04.scm` and `test05.scm` can be used to verify whether Might Scheme works with an ordered pair properly.

```
~/wasCm/s03-to-c$ cat examples/test04.scm
(letrec
  [(xx (cons 3 5))
   (fn (lambda(x y) (if (>fx x 9) 42 (+fx x y)) ))]
  (display (fn (fn (car xx) 9) 5) ))
```

```
~/wasCm/s03-to-c$ cat examples/test05.scm
(letrec
  [(xx (cons 3 4))
   (fn (lambda(x y) (if (>fx x 9) 42 (+fx x y)) ))]
  (display (fn (fn (car xx) (cdr xx)) 9)))
```

```
~/wasCm/s03-to-c$ bigloo -s
```

```
1:=> (load "scm2c.sch")
scm2c.sch
1:=> (compit "examples/test04.scm" "test04.c")
#unspecified
1:=> (compit "examples/test05.scm" "test05.c")
#unspecified
1:=> (exit)
~/wasCm/s03-to-c$ gcc -O3 test04.c -o test04.x
~/wasCm/s03-to-c$ gcc -O3 test05.c -o test05.x
~/wasCm/s03-to-c$ ./test04.x
42
~/wasCm/s03-to-c$ ./test05.x
16
```

Printing ordered pairs. Lisp has a tree representation for an ordered pair, which it uses internally to perform computations, and an external rep-

resentation, that is used to facilitate communication with people. In the external representation, a list is shown by the sequence of its elements placed between parentheses. In order to implement this external representation, one needs a function, which can read a list from a string, and the `(display s)` function, which prints the list `s`. Example `test06` checks whether `display` is working correctly.

```
~/wasCm/s03-to-c$ cat examples/test06.scm
(letrec
  [(zz (cons 4 #f))]
  (display zz))
~/wasCm/s03-to-c$ bigloo -s

1:=> (load "scm2c.sch")
scm2c.sch
1:=> (compit "examples/test06.scm" "test06.c")
#unspecified
1:=> (exit)
~/wasCm/s03-to-c$ gcc -O3 test06.c -o test06.x
~/wasCm/s03-to-c$ ./test06.x
(4)
```

In Lisp, a list of elements is implemented as a chain of pairs, each pair containing a point to a list element and another pointer to the next pair. Let us assume that `zz` points to the list `(2 3 4 5)`. This list is constituted of the following chain of pairs:

```

[*|*]--->[*|*]--->[*|*]--->[*|*]--->()
|         |         |         |
2         3         4         5
```

The examples `test06.scm` and `test07.scm` check whether Might Scheme can deal with such a representation of lists.

In Might Scheme, the `NULL` pointer, which represents the empty list, substitutes the Boolean value `#f` in applications such as `(cons 5 #f)`.

As mentioned before, the `test06.scm` example just checks whether the `display` procedure is working properly. Example `test07.scm` verifies whether Might Scheme is mature enough for implementing an algorithm, which can process all elements of a list, as discussed in section 2.9, page 20.

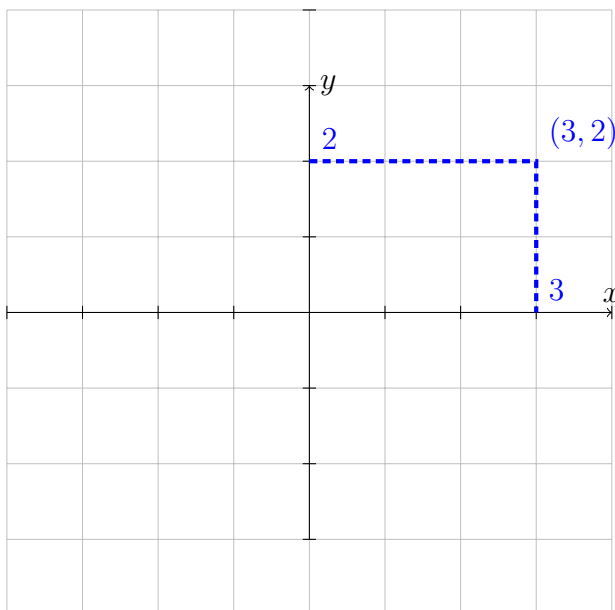
In order to go through a list, one needs a predicate `(null? x)` that checks whether the algorithm reached the end of the sequence. One needs also an `if`-statement that produces a trivial solution, when `(null? x)` is `#t` true,

and breaks the original list into shorter sublists, when it is not. Such a strategy of processing lists is said to be recursive, and will be discussed in detail later in this book.

```
~/wasCm/s03-to-c$ cat examples/test07.scm
(letrec
  [(zz (cons 2 (cons 3 (cons 4 (cons 5 #f)))))
   (add (lambda(x) (if (null? x) 0
                        (+fx (car x) (add (cdr x))))))]
  (begin (display (add zz))))
~/wasCm/s03-to-c$ bigloo -s

1:=> (load "scm2c.sch")
scm2c.sch
1:=> (compit "examples/test07.scm" "test07.c")
#unspecified
1:=> (exit)
~/wasCm/s03-to-c$ gcc -O3 test07.c -o test07.x
~/wasCm/s03-to-c$ ./test07.x
14
```

Cartesian plane. In linear algebra, numbers are called scalars. An ordered pair of scalars is called a 2-dimensional vector. Historically, two dimensional vectors are very important, because the French philosopher René Descartes famously described their use for determining a geometric point on a plane.



4.6 Low level program

There is a semantic gap between the the concepts that a programmer uses to reason and the machine that will execute code. High level languages, such as Scheme and Lisp, offer bridges to cross this gap. For instance, there is no ordered pair in the instruction set architecture that your computer uses to execute code, but Might Scheme provides the tools to build pairs with the operations that are available to the computer. Sometimes, however, one needs to write procedures that talk directly to the computer through operations, which map closely to machine instructions. In Might Scheme, these procedures must carry an `env` argument that provides the environment structure, which all functions must receive. In Might Scheme, these primitive functions are invariably created through the *define-form*.

```
;; File: examples/test10.scm
(define (fadd env fx fy)
  (+ fx fy))

(display (fadd 17.0 3.0))
```

A usual method of passing data to a compiled program is through command line arguments. Consider the program below, which was called with the `Ed` argument.

```
~/wrk/wasCm/s03-to-c$ cat examples/test12.scm
(begin (display "Hello") (display (arg 0)) (display (arg 1)))

~/wrk/wasCm/s03-to-c$ ./test12.x Ed
Hello
./test12.x
Ed
```

The command `(display (arg 0))` displays the identifier of the executable file, which is `test12.x` in the present case.

The command `(display (arg 1))` prints the first argument, which is `Ed`, the name of the person that executed the program.

4.7 Loops

As you learned in the previous section, Might Scheme possesses primitive functions, which use operations that map closely to machine instructions.

These operations are the arithmetic calculations and predicates. In a *define-form*, identifiers that start with f, g and h represent machine floating point numbers, while identifiers prefixed with i, j and k indicate integer numbers.

```
;;; File: examples/test11.scm
(define (floopsum env ix sx facc)
  (if (null? sx) facc
      (floopsum env (- ix 1) (cdr sx)
                  (+ (f(car sx)) facc)) ))

(define (floop env ix fy)
  (if (< ix 1) fy
      (floop env (- ix 1) (+ fy 0.1)) ))

(define (fadd fx fy) (+ fx fy))

(letrec
  [(xx 41.0)]
  (begin (display (fadd xx 12.5))
         (display (floop 1000000 0.0)) ))

~/wasCm/s03-to-c$ bigloo -s

1:=> (load "scm2c.sch")
scm2c.sch
1:=> (compit "examples/test11.scm" "test11.c")
#unspecified
1:=> (exit)
~/wasCm/s03-to-c$ gcc -O3 test11.c -o test11.x
~/wasCm/s03-to-c$ ./test11.x
53.5000
100000.0000
```

Listing 4.4: Primitive function with a loop

Primitive functions are used mainly to define efficient loops, or repetitions. In `examples/test11.txt`, the `(floop env ix fy)` performs the accumulation `(+ fy 0.1)` one million times, as one can see in listing 4.4.

The designers of Might Scheme could easily have removed the `env` argument in the definition of `floop` and `floopsum`. However, they decided leave

an explicit `env` variable to make it clear that primitive functions have calling conventions that are different from functional applications.

Lists can be built as a sequence of nested `cons`s. However, Might Scheme offers a more convenient method of creating a list, which is to prefix the list with the single quote symbol. In file `examples/consin.scm`, the `floopsum` function adds the elements of the `'(4 5 6 7)` list, which was defined through the single quote prefix. In Might Scheme, a preprocessor replaces this list with:

```
(cons 4 (cons 5 (cons 6 (cons 7 #f)) ))
```

where there is a `cons` call for each element of the list. An important observation: In Might Scheme, the semantic of lists prefixed by the single quote symbol is different from Standard Scheme, where identifier symbols are not replaced by its value. Might Scheme, tries to replace any identifier by its value, and produces an error if the identifier is not defined.

In primitive functions, pairs and lists are represented by the same type of values that one uses in normal functions. However, Might Scheme uses the internal representation of the underlying machine for integer and floating point numbers. This means that if a primitive function needs to perform an arithmetic operation in a list element, it must convert the list element to the machine representation. This conversion is performed by the `f` function. In the `examples/consin.scm` program, `(+ (f(car sx)) facc)` converts the `(car sx)` element to machine representation, before adding it to `facc`.

In the snippet `examples/cons.scm`, the value of the `xx` variable is 42. When Might Scheme builds the list `'(xx (80 90 45.5) 56)`, it substitutes the value 42 for `xx`, and prints `(42 (80 90 45.5) 56)`. This is not the normal behavior of standard Scheme, but makes it easier to represent the structure of web documents as nested lists. Besides this, the semantic of Might Scheme is simpler than the semantic of other dialects of Lisp, since Might Scheme only accepts symbols with well defined values, and always replaces the symbol by its value.

Readme.md

WebAssembly is an Instruction Set Architecture (ISA) for a stack machine that can run in most modern browsers. Therefore, WebAssembly allows the web designer to customize the experience of the client, who is using his or her browser to visit an Internet page. Besides this, WebAssembly is blazing fast, when compared to Javascript, the only other programming language that the World Wide Web Consortium recommended to run natively on Internet browsers.

One thing that makes WebAssembly exciting is the use of the Cambridge Prefix Syntax, the same that makes Lisp so interesting and powerful. Since Lisp is designed to handle this kind of syntax, I invite you to help me in the construction of a compiler, which translates Low Level Lisp into WebAssembly. By using Lisp as the implementation language, we can get the tokenizer and the parser for free.

Implementing Lisp onto WebAssembly is so obvious that you may wonder why somebody did not have this idea long ago. In fact, a team at Google is working on the implementation of Schism, a dialect of Lisp, on WebAssembly. The members of this team intend to outfit Schism with all powerful tools for list processing and reflection that made Lisp famous. This is not what the author of Low Level Lisp intends to do. The power of Lisp comes with a price: Lisp compilers are very difficult to implement. Compilers for Chez Scheme, Bigloo, Racket, Gambit and Sbcl required years of work from bright people, such as Manoel Serrano, Kent Dybvig, Marc Feeley and Matthew Flatt.

I am fully aware that it would be difficult and unnecessary to compete with the Google team that is constructing a WebAssembly scheme. It will be unnecessary because they will eventually succeed, and everybody will have an open source Scheme dialect to use and deploy. However, besides a high level language, such as Lisp or Prolog, one needs low level languages to develop algorithms for gaining a better understanding of the underlining computer and performing things such as memory manipulation.

WebAssembly uses the Cambridge Prefix Notation, as Lisp, but it does

not accept those concise expressions, which make Lisp programmers so productive. The idea is to design a Low Level Lisp (LLL) compiler that accepts succinct expressions, as in Scheme, but implement only those operations that can be translated directly into WebAssembly.

Let us examine a concrete example. People often use a naive definition of the Fibonacci's function to perform benchmarks. In WebAssembly, such a definition is given below.

```
(func $fib (param $n i32) (result i32)
  (if (result i32)
    (i32.lt_s (get_local $n) (i32.const 2))
    (then (i32.const 1))
    (else (i32.add (call $fib (i32.sub (get_local $n)
                                         (i32.const 1)))
                  (call $fib (i32.sub (get_local $n)
                                         (i32.const 2)))))))
```

In Low Level Lisp, the same definition would not require tags on constants or even on local variables, therefore it could become compact and more to the point, as shown below.

```
(define (fib n)
  [if (<fx n 2) 1
    (+fx (fib (-fx n 2))
         (fib (-fx n 1))) ])
```

Although succinct, the Low Level Lisp definition of naive Fibonacci has all the necessary elements to reconstruct the wat code. For instance, since the `-fx` function that performs fixed point addition accepts only `i32` numbers, the Low Level Lisp compiler can infer that the arguments of `(-fx n 2)` should be `(get_local $n)` and `(i32.const 2)`.

In a typical WebApplication, three languages would be at play. On the server side, a Bigloo program would take care of list processing and Artificial Intelligence. On the browser, WebAssembly generated by the LLL compiler would deal with text processing and numerical calculations, while Javascript could handle input output operations and insertion of the LLL generated html snippets into the document.

In the documentation of this project, the interested reader will find detailed descriptions of miniature web pages designed pursuant to the LLL philosophy. The documentation also provides a brief tutorial of Bigloo, one of the languages supported by Manoel Serrano's Hop web programming environment. Finally, there is the ongoing work on an LLL compiler.

Chapter 5

WebAssembly

WebAssembly (abbreviated **wasm**) is a set of instructions for a stack-based virtual machine. Wasm offers a portable binary-code format that a virtual machine in a browser can execute, and also a corresponding symbolic expressions (sexpr) that people can use to generate programs. Like Lisp, WebAssembly definitions are written in Cambridge Prefix Notation.

To work with WebAssembly, you will need a **wat2wasm** compiler that generates wasm binary-code from source code written in the WebAssembly text (wat) format. You should search the internet for a distribution of the **wat2wasm** compiler and install it in your machine. Again, if you are not proficient in Computer Science, ask for help from students of the local college. I will not provide instructions about the installation of the **wat2wasm** compiler, because they change with time and from computer to computer. You will also need the installation of the node interpreter for Javascript.

5.1 Types

Let us create a new directory, where you can follow my efforts to learn how WebAssembly works.

```
~$ mkdir wrk
~$ cd wrk
~/wrk$ mkdir wasm
~/wrk$ cd wasm
~/wrk/wasm$
~/wrk/wasm$ wat2wasm arith.wat
~/wrk/wasm$ node arithclient.js
Answer to the Ultimate Question of Life
42
```

In the listing above, I compiled the program of listing 5.1, then ran it through the Javascript client given in listing 5.2.

```
;; File arith.wat
(module
;; Add two numbers together
(func $sum (param $lhs i32) (param $rhs i32) (result i32)
  (i32.add
    (local.get $lhs)
    (local.get $rhs)))
  (export "sum" (func $sum)))

;; Answer to the Ultimate Question of Life
(func $hitchHike (param $d i32) (result i32)
  (i32.div_u
    (call $sum
      (i32.const 84)
      (i32.const 42))
    (local.get $d))) ;;divide
  (export "answer" (func $hitchHike)))
```

Listing 5.1: Type i32

```
const { readFileSync } = require("fs");

const run = async () => {
  const buffer = readFileSync("./arith.wasm");
  const module = await WebAssembly.compile(buffer);
  const instance = await WebAssembly.instantiate(module);
  console.log("Answer to the Ultimate Question of Life");
  console.log(instance.exports.answer(3));};

run();
```

Listing 5.2: Javascript client

As I said before, wasm uses the same syntax as Lisp, however, wasm requires that you declare the types of variables and functions. The `$hitchHike`

function has a single parameter `d` of 32 bits and produces a result of 32 bits as well. The type system of `wasm` requires that even constants, such as `(i32.const 42)`, receive a tag, which declares the type of the constant.

```
<!DOCTYPE html>
<html><head>
  <meta http-equiv="content-type"
    content="text/html; charset=UTF-8"/>
  <title>WASM test with XHR</title> </head>

  <body>
    <h2>-Answer to the Ultimate Question of Life-</h2>
    <p id="tag"> </p>

    <script>
      var importObject = {
        imports: { imported_func: function(arg) {
          console.log(arg);} } };

      function exec(obj) {
        document.getElementById("tag").innerHTML=
          obj.instance.exports.answer(3);}

      request = new XMLHttpRequest();
      request.open('GET', 'arith.wasm');
      request.responseType = 'arraybuffer';
      request.send();

      request.onload = function() {
        var bytes = request.response;
        WebAssembly.instantiate(bytes, importObject).then(exec);};
    </script>
  </body></html>
```

Listing 5.3: Wasm in the Web

The first thing you learned in this chapter was that all functions and parameters of `wasm` require a type declaration. You also learned that you need a Javascript client to execute a WebAssembly Program. You can run the client through the *node* runtime, as shown in figure 5.2, or through the World Wide Web. Figure 5.3 shows a client that can be installed as a `html`

page. You must place both the `arith.wasm` file and the page of figure 5.3 in the same directory of your html server.

I don't know much about Javascript, except that it is one of the four languages to run natively in browsers, the other two being HTML, CSS and WebAssembly, of course. However, from listing 5.2, I can see that a Javascript program needs a `run()` function, which receives the wasm program. The Javascript compiles that buffer into a module, instantiate the module, and only then can start calling wasm functions.

5.2 Memory sharing

Since WebAssembly does not have means of printing results, or inserting html snippets into a document, it must rely on Javascript to perform these tasks. In the previous section, you learned that Javascript can receive and process numerical results from WebAssembly functions. Now, you will see that a linear array of bytes can be shared between Javascript and WebAssembly, which makes it possible to exchange vast amounts of data between WebAssembly and the browser.

```
;; File mem.wat

(module
  (memory $mem 1 2)
  (export "mem" (memory $mem))

  (func $setmem (param $n i32) (param $v i32) (result i32)
    (i32.store (i32.mul (local.get $n) (i32.const 4))
              (local.get $v))
    (local.get $v))

  (export "set" (func $setmem))

  (func $ndx (param $n i32) (result i32)
    (i32.load (i32.mul (local.get $n) (i32.const 4) )))

  (export "ndx" (func $ndx)) )
```

Listing 5.4: Test memory sharing: mem.wat

Vector of integers. Function `i32.store` can store a number in a given position of a byte array. The position must be given in number of bytes. Therefore, if I want to create a vector of 32 bit integers, the index of such a vector must be multiplied by 4, in order to find the position. This is exactly what has happened in function `$setmem` that I have defined in figure 5.4.

```
const { readFileSync } = require("fs");

const run = async () => {
  const buffer = readFileSync("./mem.wasm");
  const module = await WebAssembly.compile(buffer);
  const instance = await WebAssembly.instantiate(module);
  const mem = new Int32Array(instance.exports.mem.buffer);
  console.log("Test memory");
  mem[1]= 67; mem[2]= 68; mem[3]= 69;
  console.log(instance.exports.set(0, 666));
  console.log(instance.exports.ndx(1));
  console.log(instance.exports.ndx(2));
  console.log(instance.exports.ndx(0));
  console.log("=== End test ===");};

run();
```

Listing 5.5: Js code that shares memory with wasm - `memclient.js`

Below, I have used the **node** runtime to test the memory sharing wasm program.

```
~/wrk/wasm$ wat2wasm mem.wat
~/wrk/wasm$ node memclient.js
Test memory
666
67
68
666
=== End test ===
```

As you can see, the `set` function, exported by wasm, sees the linear memory `mem` as a vector of integer. In the example, it was used to store the number of the beast at index 0 of the vector. Further into the test, `mem[0]` was correctly retrieved by the `ndx(0)` call.

5.3 printing operations

Since we already know how to insert WebAssembly generated html snippets into a document, printing operations are not necessary. However, let us see how this could be done.

```
(module ;; File wrt.wat
(import "js" "emit" (func $pr (param i32)))
(memory $mem 1 2) (export "mem" (memory $mem))

(func $prt (param $v i32) (result i32)
  (call $pr (local.get $v))
  (local.get $v))
(export "prt" (func $prt)))
```

Below is the Javascript program that you can use to perform the testing of the `wrt.wasm` application:

```
const { readFileSync } = require("fs");
const run = async () => {
  var importObj = {js: {
emit: (x) => { var s = ''; for (var i= 0; i< 4; i++)
                s += String.fromCharCode(x);
                console.log(s)},
greet: () => console.log("Hello!") }};
  const buffer = readFileSync("./wrt.wasm");
  const module = await WebAssembly.compile(buffer);
  const instance = await WebAssembly.instantiate(module,
                                                importObj);
  const mem = new Int32Array(instance.exports.mem.buffer);
  instance.exports.prt(65);
  instance.exports.prt(66);};

run();
```

Here is how to compile and run the application above.

```
~/wrk/wasm$ wat2wasm wrt.wat
~/wrk/wasm$ node wrtclient.js
AAAA
BBBB
```

5.4 Strings

Strings are sequence of chars. WebAssembly implements strings practically out of the box. In figure 5.6, you can see a simple implementation of strings in WebAssembly.

```
(module
  (memory (export "mem") 1)

  (data (i32.const 1004) "String demo.")
  (data (i32.const 1204) "Hello, you Guys!")

  (func $initstr (result)
    (i32.store (i32.const 1000) (i32.const 12))
    (i32.store (i32.const 1200) (i32.const 15)))

  (export "initstr" (func $initstr))

  (func $setmem (param $n i32) (param $v i32) (result)
    (i32.store (local.get $n) (local.get $v)))

  (export "set" (func $setmem))

  (func $getsz (param $n i32) (result i32)
    (i32.load (local.get $n)))

  (export "getsize" (func $getsz))

  (global (export "pos") i32 (i32.const 1000)))
```

Listing 5.6: Strings: WebAssembly side – `str.wat`

The `data` declaration is used to specify where a string starts as well as its contents. In listing 5.6, there are two strings, one starting at position 1004 and the other at 1204. Function `$initstr` stores the sizes of these two strings at positions 1000 and 1200 respectively. This function should be called from Javascript, in order to initialize the strings with their sizes: In the example, the first string contains 12 bytes, which are stored at position 1000, while the second contains 15 bytes, value that is stored at position 1200.

The `$getsz` function is exported Javascript, in order to retrieve the size of a string. If the user wants to retrieve only part of the string, she can use

the function `$setmem` to store a smaller value at the string size position.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
  <h2 id="htag"></h2>
  <p id="tag"> </p>

  <script>
function run(wasm) {
  const m  = wasm.exports.mem;
  wasm.exports.initstr();
  const p = wasm.exports.pos;
  function sz(i) { return wasm.exports.getsize(i);}
  wasm.exports.set(1000, 11);
  const str = new Uint8Array(m.buffer, p+204, sz(p+200));
  const msg = new Uint8Array(m.buffer, p+4, sz(p));
  const s = new TextDecoder('utf8').decode(str);
  const zmsg = new TextDecoder('utf8').decode(msg);
  document.getElementById("tag").innerHTML= s;
  document.getElementById("htag").innerHTML= zmsg;}
fetch("str.wasm").then(reponse =>
  reponse.arrayBuffer()
).then(bytes =>
  WebAssembly.instantiate(bytes, {})
).then(result =>
  result.instance
).then(run);
  </script>
</body>
</html>
```

Listing 5.7: Strings: Javascript side

5.5 Lists

The last thing that we need to implement in WebAssembly are singly linked lists, the kind of lists that exists in Lisp. In order to implement lists, one needs only three operations – `cons`, `car` and `cdr`, which are implemented in listing 5.9 and 5.8.

```
<!DOCTYPE html>
<html lang="en">
<head> <meta charset="UTF-8">
    <title></title>
</head>
<body>
    <h2 id="htag"></h2>
    <p id="tag"></p>
<script>
function run(wasm) {
    const m    = wasm.exports.mem;
    wasm.exports.initstr();
    const p = wasm.exports.makelist();
    function fst(i) {return wasm.exports.car(i);}
    function rst(i) {return wasm.exports.cdr(i);}
    function sz(i) {return wasm.exports.getsize(i);}
    const s = new Uint8Array(m.buffer, fst(p)+4, sz(fst(p)));
    const zs= new TextDecoder('utf8').decode(s);
    const msg= new Uint8Array(m.buffer, fst(rst(p))+4,
                               sz(fst(rst(p))));
    const zmsg = new TextDecoder('utf8').decode(msg);
    document.getElementById("htag").innerHTML= zmsg;
    document.getElementById("tag").innerHTML= zs;}
fetch("consdemo.wasm").then(reponse => reponse.arrayBuffer()
    ).then(bytes => WebAssembly.instantiate(bytes, {}))
    ).then(result => result.instance
    ).then(run);
</script>
</body> </html>
```

Listing 5.8: List selectors: Javascript side

```

(module (memory (export "mem") 1)
  (data (i32.const 1004) "Anne")
  (data (i32.const 1204) "Maria")
  (func $getsz (param $n i32) (result i32)
    (i32.load (local.get $n)))
  (export "getsize" (func $getsz))

  (func $newcons (result i32)
    (i32.store (i32.const 2000)
      (i32.add (i32.load (i32.const 2000)) (i32.const 10)) )
    (i32.load (i32.const 2000)))
  (func $cons (param $hd i32) (param $tail i32) (result i32)
    (local $n i32) (local.set $n (call $newcons))
    (i32.store (local.get $n) (local.get $hd))
    (i32.store (i32.add (local.get $n) (i32.const 4))
      (local.get $tail))
    (local.get $n))

  (func $car (param $xs i32) (result i32)
    (i32.load (local.get $xs)))
  (export "car" (func $car))
  (func $cdr (param $xs i32) (result i32)
    (i32.load (i32.add (local.get $xs) (i32.const 4)) ))
  (export "cdr" (func $cdr))

  (func $makelist (result i32)
    (call $cons (i32.const 1000)
      (call $cons (i32.const 1200) (i32.const 0)) ))
  (export "makelist" (func $makelist))

  (func $initstr (result)
    (i32.store (i32.const 2000) (i32.const 2000))
    (i32.store (i32.const 1000) (i32.const 4))
    (i32.store (i32.const 1200) (i32.const 5)))
  (export "initstr" (func $initstr))
  (global (export "pos") i32 (i32.const 1000)))

```

Listing 5.9: List constructor and selectors

5.6 Why do I need Low Level Lisp?

As you could see, dealing with strings requires a lot of redtape. Low Level Lisp can spare you from this manual handling of memory.

```
(module
  (mem 2)
  (data 1000 100 "Hello, World!"
    "Hi, Suzy"))
```

Listing 5.10: LLL source code: scmstr.web

The line below shows how to compile the program of listing 5.10.

```
~/src/wasCm$ scheme/wascm.x scmstr
~/src/wasCm$ wat2wasm scmstr.wat
```

Figure 5.11 shows an html page that deploys the program of listing 5.10. Program of listing 5.10 requires only what is strictly necessary to store and retrieve strings, to wit, the position where the string memory starts and the maximum size that a string can have, which is 100 bytes, in the example.

The compiler for Low Level Lisp is written in Bigloo scheme. You will find the compiler in the same repository, where you found this documentation. After installing Bigloo, in order to build the `wascm.scm` source file, all you need is to type the line below.

```
~/src/wasCm/scheme$ bigloo wascm.scm -o wascm.x
```

Building Bigloo projects is quite easy, but people who are not familiar with the operating system can get lost in the labyrinth of paths and permissions. Therefore, if you are not a hacker yourself, my advice is to engage the help of someone who is conversant with a computer terminal.

```
<!DOCTYPE html>
<html lang="en"><head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
  <h2>Test string access</h2>
  <p id="tag"> </p>
<script>
function run(scmstr) {
  const m  = scmstr.exports.mem;
  scmstr.exports.initstr();
  function p(i) {return scmstr.exports.pos(i)+4;}
  function sz(i) {return scmstr.exports.getsize(i);}
  console.log(p(0));
  const str = new Uint8Array(m.buffer, p(1), sz(1));
  const s = new TextDecoder('utf8').decode(str);
  document.getElementById("tag").innerHTML= s;}
fetch("scmstr.wasm").then(reponse =>
  reponse.arrayBuffer()
).then(bytes =>
  WebAssembly.instantiate(bytes, {})
).then(result =>
  result.instance
).then(run);
</script>
</body>
</html>
```

Listing 5.11: LLL source code: scmstr.web

Chapter 6

Simple programs in LLL

Let us see two simple programs in Low Level Lisp, a language that we will call LLL from now on.

```
;; File: fib.web
(module

  (define (fib.n i)
    (if (< i 2) 1
        (+ (fib.n (- i 1))
            (fib.n (- i 2))) )))
```

Listing 6.1: Fibonacci of fixnum integer

Below there is a client for the program of listing 6.1.

```
const { readFileSync } = require("fs");

const run = async () => {
  const buffer = readFileSync("./fib.wasm");
  const module = await WebAssembly.compile(buffer);
  const instance = await WebAssembly.instantiate(module);
  console.log("Fibonacci(40)=", instance.exports.fib(40));
};

run();
```

Listing 6.2: fib client

Computers operate on many numerical types, such as i32 bit integers and float numbers, which are approximations of real numbers. For each type, there is a different set of instructions. Low Level Lisp can infer which instruction it should use, but the programmer must inform the type of variables and the return type of used defined functions, which can be done through the `.n` suffix, in the case of 32 bit integers. Besides this, the identifiers `i`, `j`, `k` and `n` are reserved to i32 integers.

Below, you will see how to compile and run the program of listing 6.1. The first step of the compilation process is to translate from Low Level Lisp to WebAssembly Text, or **wat**. This is done by a Bigloo program that you compiled previously, and placed the executable in the directory `scheme`. The `fib.wat` file that LLL generates needs to be translated into binary code by the `wat2wasm` compiler.

```
~/src/wasCm/examples$ cat fib.web
;; File: fib.web
(module

  (define (fib.n i)
    (if (< i 2) 1
        (+ (fib.n (- i 1))
            (fib.n (- i 2))))))

~/src/wasCm/examples$ ../scheme/wascm.x fib.web
~/src/wasCm/examples$ wat2wasm fib.wat
~/src/wasCm/examples$ node fibclient.js
Fibonacci(40)= 165580141
```

The command `wat2wasm fib.wat` translate the WebAssembly Text, or **wat** for short, into binary code that is placed into the `fib.wasm` file, and can be executed by the WebAssembly virtual machine. There are two main ways to execute a **wasm** file. One of them is to publish it on a web page, so people can enjoy the application on their browsers.

A faster and simpler method of testing a **wasm** program is through the node run time. Either way, you will need to write a Javascript client program, in order to call the **wasm** compiled binary code. The first step is to load the **wasm** file and create a module for it. Then you can instantiate the module and call any function that it exports.

```
const buffer = readFileSync("./fib.wasm");
const module = await WebAssembly.compile(buffer);
const instance = await WebAssembly.instantiate(module);
console.log(instance.exports.fib(40));
```

6.1 Floating point

Besides i32 integers, another important numerical type is floating point numbers. LLL considers that any identifier represents f64 floating point numbers, unless it is reserved for integers. The names reserved for integers are the `i`, `j`, `k`, `n` variables and the identifiers, which have the `n` suffix.

```
;; File: ffib.web
(module
(define (fib x)
  (if (< x 2.0) 1.0
      (+ (fib (- x 1.0)) (fib (- x 2.0))) )))
```

Listing 6.3: Fibonacci of a floating point number

Listing 6.3 shows that the decimal point is mandatory for floating point constants, since it is one of the hints that the type inference system uses to select the correct arithmetic and logic instructions.

```
// File: ffibclient.js
const { readFileSync } = require("fs");

const run = async () => {
  const buffer = readFileSync("./ffib.wasm");
  const module = await WebAssembly.compile(buffer);
  const instance = await WebAssembly.instantiate(module);
  console.log("Fib(40)=", instance.exports.fib(40.0));};

run();
```

Listing 6.4: Float Fibonacci's function client

The floating Fibonacci client is not very different from the `fixnum` version. You must load the binaries, compile and instantiate the module, then call whatever function the module exports. The execution of the program through the node runtime is identical in the two cases.

```
~/src/wasCm/examples$ ../scheme/wascm.x ffib.web
~/src/wasCm/examples$ wat2wasm ffib.wat
~/src/wasCm/examples$ node ffibclient.js
Fib(40)= 165580141
```

6.2 Tail Recursion Elimination

The only way that Low Level Lisp has to repeat an algorithm is through recursive functions. A recursive function breaks down the data into smaller parts, then solve them straightforwardly, if the solution is trivial, or apply the same function to each part, and combine the Results. The resume of this strategy is to define a function in terms of itself.

Recursion seems almost like magic, but there is an important difference between recursion and magic. People see magic as a way of getting something for free, without paying the tag price established by the laws of nature, such as conservation of energy or entropy increase. On the other hand, recursion is very expensive computationally. To see this point, let us examine the definition of the Fibonacci's function:

```
(define (fib.n i)
  (if (< i 2) 1
      (+ (fib.n (- i 1))
         (fib.n (- i 2))) )))
```

For i larger than 1, `(fib.n i)` must go to the beginning of the algorithm twice, in order to calculate `(fib.n (- i 1))` and `(fib.n (- i 2))`, then come back to the expression `(+ (fib.n (- i 1)) (fib.n (- i 2)))` to calculate the addition.

In order to reduce the cost of recursion, the `fib.n` function should perform the addition before jumping to the beginning of the algorithm. Thus, the repetition of the algorithm would not need to store a return address to a pending calculation, as there would be no addition or any other calculation wrapping the recursive call of `fib.n`.

```
(module
  (define (fib.n i f1.n f2.n)
    if (< i 2) f1.n
        (fib.n (- i 1) (+ f1.n f2.n) f1.n))

  (define (ffib i f1 f2)
    if (< i 2) f1
        (ffib (- i 1) (+ f1 f2) f1)))
```

Listing 6.5: Tail Call Elimination

When `fib.n` goes to the beginning of the algorithm, it should carry the addition already finished in one of its arguments, and all other arguments

as well, so there will be no need of coming back. The process of writing a repetition, which does not require that the recursive function return to a pending calculation is called tail call elimination.

Figure 6.5 shows how to use tail call elimination to calculate the Fibonacci's numbers. In order to make it clear that tail call elimination is vastly different from normal recursion, Low Level Lisp doesn't wrap the `if-then-else` form in parentheses. This syntax, besides making clear the difference between normal and tail recursion, has the additional advantage of avoiding the accumulation of parentheses. However, from the next chapter onwards, you will learn standard Lisp, where there is no difference between normal recursion and tail recursion, therefore you must examine the recursive calls carefully, to check whether they are wrapped in parentheses.

```
const { readFileSync } = require("fs");

const run = async () => {
  const buffer = readFileSync("./tailfib.wasm");
  const module = await WebAssembly.compile(buffer);
  const instance = await WebAssembly.instantiate(module);
  console.log("=== Tail Recursion Elimination ===");
  console.log(instance.exports.fib(42, 1,1));
  console.log(instance.exports.ffib(5, 1.0, 1.0)); };

run();
```

Listing 6.6: A client for tailfib: `tailfibclient.js`

Figure 6.6 shows the Javascript program that calls the `fib.n` and `ffib` functions. These two functions should serve as model for efficient loop implementation, where *loop* means repetition achieved through tail call elimination. In fact, Low Level Lisp uses loops to implement tail call elimination.

I will assume that you have the Low Level Lisp compiler, which is written in Bigloo scheme installed in the `~/src/wasCm/scheme/` folder of your home directory. Let us assume that you have managed to install the Bigloo scheme compiler. Here is how to build the LLL compiler:

```
~/src/wasCm/scheme$ bigloo wascm.scm -o wascm.x
~/src/wasCm/scheme$ cd ..
~/src/wasCm$ _
```

Compilation and deployment. Now, you are ready to translate the LLL source code into WebAssembly Text, or *wat* for short. The `wascm.x` LLL compiler is in the `~/src/wasCm/scheme/`, where we have built it through the Bigloo compiler. However, the source code is in the `~/src/wasCm/examples/` folder. Therefore, you need to provide the complete path from one folder to the other, in order to call the compiler.

```
~/src/wasCm/examples$ ../scheme/wascm.x tailfib.web
declaration - module
define - (fib.n . 3)
define - (ffib . 3)
((ffib . 3) (fib.n . 3))

~/src/wasCm/examples$ wat2wasm tailfib.wat
~/src/wasCm/examples$ node tailfibclient.js
=== Tail Recursion Elimination ===
433494437
8
~/src/wasCm/examples$
```

Although WebAssembly has a `return_call` command that was designed especially for tail call optimization, not all browsers implement it, therefore the designers of Low Level Lisp decided to avoid it.

In this chapter, you have learned that there are two ways of running programs compiled to WebAssembly. The first way is through a Javascript program embedded into an html page, that you can open with a browser. The other is to use the node runtime. Listing 6.6 was designed for the node option.

Chapter 7

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 imagine 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 I 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 answered 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 imagination. 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 an effect of decaying false teeth. In the garden, the flatsedge and other weeds had expelled the begonias. However, the more resistant 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 appeared 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, Guanches¹ 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.

¹Guanches 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 near², 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 renowned 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,

²Before 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.

ᮊ ᮓ ᮒ ᮑ ᮓ ᮓ – There were two messengers

ᮓ ᮓ ᮒ ᮓ ᮓ ᮓ – with mutual hatred.

ᮓ ᮓ ᮒ ᮓ ᮓ ᮓ – One was as mighty as the other.

ᮓ ᮓ ᮓ ᮓ ᮓ ᮓ – 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 subsequent 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 acquaintances. 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 nothing³.

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,

³A literal translation of the Portuguese 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 the 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 *Archivio 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 absence 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 Polynesian 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 8

Arithmetic operations

In this chapter, you will see that the emacs editor is designed for writing programs, as well as executing them.

Nia entered the editor and pressed `C-x C-f` to create the `celsius.scm` buffer. She typed the program below into the buffer.

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

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

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

Function `c2f` converts Celsius readings to Fahrenheit. The `define` 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 `M-x shell` to create a terminal window

for interacting with Scheme. After this action, there are two windows on the screen, one showing the `celsius.scm` buffer and the other showing the terminal. The `C-x o` command makes the cursor jump from one window to the other. From the bottom window, Nia types the `bigloo -s` command to create the `Read Eval Print Loop` that provides interaction with scheme.

On the `REPL` buffer, Nia can load the `celsius.scm` program and call any application that she has defined herself, or which comes with Lisp. The first operation is loading the `celsius.scm` file, as you can see in figure 8.1.

Let us recall what happened until now. Nia pressed `M-x shell` command to split the buffer window and create a terminal, as shown in figure 8.1. On the terminal window, she issues the command `bigloo -s` to start the scheme `REPL` interaction buffer. 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.

The command `C-x C-s` saves the Lisp source file.

On the `REPL` window, the `(load "celsius.scm")` command loads the source file. After loading the source file, Nia types `(c2f 100)` on the interaction buffer to convert 100 Celsius degrees to Fahrenheit degrees. 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.

```

; File: celsius.scm

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

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

~/LLLdocs/src$ bigloo -s

1:=> (load "celsius.scm")
celsius.scm
1:=> (c2f 100)
212.0

> (load "celsius.scm")

```

Listing 8.1: Source file and Iteration Buffer

8.1 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 8.2. When using a `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.

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.

```
(define (winter m)
  (quotient (- 14 m) 12))

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

(define (zr y m day)
  (let* [ (roman-month (roman-order m))
         [roman-year (- y (winter m))]
         [century (quotient roman-year 100)]
         [decade (remainder roman-year 100)) ]
    (remainder(+ day decade
                  (quotient (- (* 13 roman-month) 1) 5)
                  (quotient decade 4)
                  (quotient century 4)
                  (* century -2)) 7)) )
```

```
1:=> (load "zeller.scm")
zeller.scm
1:=> (zr 2016 8 31)
3
```

Listing 8.2: Source file and Iteration Buffer

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 ninth 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 eighth; November, the ninth 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.

```
(define (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.

```
(define (winter m)
  (quotient (- 14 m) 12))
```

For months from 3 to 12, (- 14 m) produces a number less than 12, and (quotient (- 14 m) 12) is equal to 0. 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 8.2 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 8.2 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 8.2, expressions such as `(zr 2016 8 31)` return a number between 0 and 6, corresponding to Sunday, Monday, Tuesday, Wednesday, Thursday, Friday and Saturday.

8.2 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. Scheme 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.

```
(define (zeller y m day)
  (let* [ (roman-month (if (< m 3) (+ m 10) (- m 2)))
        (roman-year (if (< m 3) (- y 1) y))
        (century (quotient roman-year 100))
        (decade (remainder roman-year 100))]
    (remainder (+ day decade
                  (quotient (- (* 13 roman-month) 1) 5)
                  (quotient decade 4)
                  (quotient century 4)
                  (* century -2)) 7)))
```

```
1:=> (load "zel.scm")
zel.scm
1:=> (zeller 2016 8 31)
3
```

Listing 8.3: Making decisions

You probably know, in all computer languages, *Boolean* is a data type that can have just two values: `#t` for “true” and `#f` for “false”. The procedure `(< m 3)` returns `#t` if `m` is less than 3, and `#f` 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 `(quotient decade 4)` to the formula.

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 `(remainder 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 `(quotient 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

```
(define (acc i) (- (quotient (- (* 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.

Chapter 9

Sets

A set is a collection of things. You certainly know that collections do not have repeated items. I mean, if a guy or girl 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.

9.1 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} : $\{0, 1, 2, 3, 4 \dots\}$.

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

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 Argentines are somewhat conceited. *You know what is the best possible deal? It is to pay a fair price for Argentines, 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** ($> \text{ x } 2$) ($< \text{ x } 5$)) in Lisp notation.

Disjunction. The expression $(x < 2) \vee (x > 5)$ means (**or** ($< \text{ x } 2$) ($> \text{ x } 5$)) in Lisp notation

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

$$\mathbb{Q} = \left\{ \frac{p}{q} \mid p \in \mathbb{Z} \wedge q \in \mathbb{Z} \wedge q \neq 0 \right\}$$

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 scheme, 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.


```

(define f (cons 2 3))

(define g (cons 4 5))

(define (add x y)
  (let ( (numerator (+ (* (car x) (cdr y))
                        (* (car y) (cdr x)) ))
        (denominator (* (cdr x) (cdr y)) ))
    (cons numerator
          denominator)))

~/LLLdocs/src$ bigloo -s

1:=> (load "cartesian.scm")
cartesian.scm
1:=> (add f g)
(22 . 15)

```

Listing 9.1: 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:

$$\frac{X_a}{X_d} + \frac{Y_a}{Y_d} = \frac{X_a \times Y_d + Y_a \times X_d}{X_d \times Y_d} \quad (9.1)$$

In the dotted pair notation, one has $X_a = (\text{car } X)$, $X_d = (\text{cdr } X)$, $Y_a = (\text{car } Y)$ and $Y_d = (\text{cdr } Y)$. By replacing these values in equation 9.1, 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 9.1.

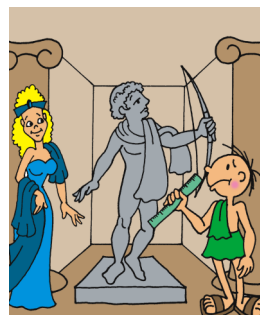
9.2 Irrational Numbers

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 how 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.

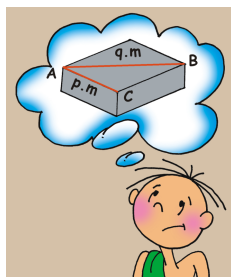
If the Greeks were right, I would be able to find a meter, possibly a very small one, that produces an integer measure for the diagonal of the square, and another integer measure for the side. Suppose that p is the result of measuring the side of the square, and q is the result of measuring the diagonal.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

$$p^2 + p^2 = q^2 \therefore 2p^2 = q^2 \quad (9.2)$$

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 9.2, 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 9.2.



$$2 \times p^2 = q^2 = (2 \times n)^2 = 4 \times n^2 \therefore 2 \times p^2 = 4 \times n^2 \therefore p^2 = 2 \times n^2 \quad (9.3)$$



Equation 9.2 shows that q is even; equation 9.3 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 9.2, 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, scheme and all other programming languages replace sets with the concept of type. In scheme, \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 scheme, 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
(expt  $x$   $y$ ) –  $x^y$ 
(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
```

Integers are distinguished from inexact numbers by the `(integer? x)` predicate. Two important integer functions are `(quotient x y)` and `(remainder 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.

9.3 Data types

There are other types besides **integer**, and **number**. Here is a list of primitive types:

integer — Integer numbers between -2147483648 and 2147483647 . The `(exact? x)` function returns `#t` if `x` is an integer, and `#f` otherwise.

inexact — the inexact type represents both rational and irrational numbers, albeit approximately. The `(inexact? x)` function returns `#t` for inexact numbers.

number — A number can be either exact or inexact. The `(number? x)` function returns `#t` for numbers, and `#f` for any other type of object.

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

- `(string-length s)` returns the number of chars of the `s` string.
- `(substring "Hello, World" 2 5)` operation returns the `"llo"` substring. `(substring "Hello, World" 0 4)` returns the `"Hell"` substring. And so on.

Char — Chars have the `#\` prefix: `#\A`, `#\b`, `#\3`, `#\space`, `#\newline`, etc. The `(char? x)` function returns `#t` when `x` is a char. Let `s` be a string such as `"Hello"`. Then, `(string-ref "Hello" 0)` returns the first char of `s`, `(string-ref "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.
```

Pair — One can use a pair data type both to represent Cartesian pairs and lists. The `(pair? x)` returns `#t` if `x` is a pair, otherwise it returns `#f`. The `'()` empty list is a very important object that one can use as the second element of a pair. The `(null? x)` returns `#t` when `x` is the `'()` empty list.

9.4 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 **#f** (*False*) to the letters of the Roman alphabet. If a letter is a vowel, then the value will be **#t**; otherwise, it will be **#f**. The range of such a function is **{#t, #f}**, 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	#f	m	#f	t	#f
b	#f	h	#f	n	#f	u	#t
c	#f	i	#t	o	#t	v	#f
d	#f	j	#f	p	#f	w	#f
e	#t	k	#f	q	#f	x	#f
f	#f	l	#f	r	#f	y	#f
				s	#f	z	#f

9.4.1 Clauses

From what you have seen in the last section, you certainly notice 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

```
(define (vowel x)
  (cond ( (or (char=? x #\a)(char=? x #\e)(char=? x #\i)
              (char=? x #\o)(char=? x #\u)) #t)
        (else #f)))
```

Expressions like $(\text{or } p_1 p_2 \dots p_n)$ has the same meaning as $p_1 \vee p_2 \dots \vee p_n$.

Functions have a parameter, also called variable, that represents an element of the domain. Thus, the vowel function has a parameter **x**, that represents an element of the set $\{'a', 'b', 'c', 'd', 'e', 'f', \dots\}$. Below the name of the function, and its variable, 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. The expression:

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

asks the question: *Is $x = \text{\#a}$, $x = \text{\#e}$, $x = \text{\#i}$, $x = \text{\#a}$ or $x = \text{\#a}$?* If the answer is yes, the clause value is **#t**; in proper functions, the clause value is given by the second clause expression.

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 6 entries:

0	1	3	3
1	1	4	5
2	2	5	8

Notice that a given functional value is equal to the sum of the two precedent values, i.e., $f_{n+1} = (+ f_n f_{n-1})$. Assume that $n = 5$. Then, $f_6 = (+ f_5 f_4)$.

Of course, the expression $f_{n+1} = (+ f_n f_{n-1})$ is true only for $n > 0$, since there are not two precedent values for $n + 1 = 0$, or $n + 1 = 1$. The below programa shows how you can state that a rule is valid only under certain conditions.

```
(define (fib n fn fn-1)
  (cond ((< n 2) fn)
        (else (fib (- n 1) (+ fn fn-1) fn))))

1:=> (load "fibonacci.scm")
fibonacci.scm
1:=> (fib 5 1 1)
8
```

Predicates. A predicate is a function that gives true and false for output. In Scheme, the only false value is `#f`, but any value that is different from `#f` 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 `'()`. 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 `#f` 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:

```
(define s "Nia Vardalos")
```

Then, the characters `#\N` `#\i` `#\a` and `#\space` were retrieved from `s` by the expressions `(string-ref s 0)`, `(string-ref s 1)`, `(string-ref s 2)` and `(string-ref 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 `#f` 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)` – `#f` since the order of `#\a` greater or equal to the order of `#\c`.

Besides predicates, Scheme uses special forms to make decisions. The (and $p_1 p_2 \dots p_n$) form evaluates the sequence $p_1, p_2 \dots p_n$ until it reaches p_n or one of the previous p_i returns `#f`. In fewer words, the and-form stops at the first p_i that returns `#f` 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 `#f` if and only if all p_i predicates produce values different from `#f` (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 `#f`. The (or $p_1 p_2 \dots p_n$) form returns the value of the first true p_i predicate.

The (not P) form returns `#t` if P produces `#f`, and evaluates to `#f` if P is true, i.e., anything different from `#f`.

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

```
(define (digit? d)
  (and (char>=? d #\0)
       (char<=? d #\9)))
```

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

```
(define (deep-winter? m)
  (or (= m 1) (= m 2)))
```


9.5 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.

```
(define (week-day n)
  (cond [ (= n 0) 'Sunday]
        [ (= n 1) 'Monday]
        [ (= n 2) 'Tuesday]
        [ (= n 3) 'Wednesday]
        [ (= n 4) 'Thursday]
        [ (= n 5) 'Friday]
        [ else 'Saturday]))

(define (zeller y m day)
  (let* [ (roman-month (if (< m 3) (+ m 10) (- m 2)))
          (roman-year (if (< m 3) (- y 1) y))
          (century (quotient roman-year 100))
          (decade (remainder roman-year 100)) ]
    (week-day (remainder (+ day decade
                           (quotient (- (* 13 roman-month) 1) 5)
                           (quotient decade 4)
                           (quotient century 4)
                           (* century -2)) 7)) ))
```

```
1:=> (load "zellerCond.scm")
zellerCond.scm
1:=> (zeller 2018 8 31)
Friday
```

Listing 9.2: 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 descending order. It executes the first clause whose condition produces a value different from `#f`. For instance, the first clause condition is the `(= n 0)` predicate. If the predicate `(= n 0)` returns `#t` for the Sunday index, the function `week-day` returns `'Sunday`. If `n = 1`, the second condition holds, and the function produces the symbol `'Monday`. And so on.

9.6 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.

```
(define *x* 42)

(define xor
  (lambda(A B)
    (or (and A (not B))
        (and (not A) B)) ))

(define 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 Scheme 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 `define` 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)

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

(define 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 `(xor A B)` format that is a mirror of the gate application. The advantage of this method is that it spares one nesting level, and shows clearly how to use the definition.

In the second style of defining functions and gates, the id of the abstraction appears immediately after the `define` keyword. The arguments and the body of the definition are introduced by a lambda form:

```
(define 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: lists.scm

(define xor-structure
  '(or (and A (not B))
       (and (not A) B)))

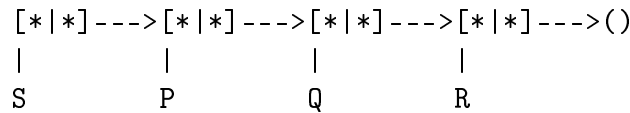
1:=> (load "lists.scm")
lists.scm
1:=> (car xor-structure)
or
1:=> (cdr xor-structure)
((and A (not B)) (and (not A) B))
1:=> (car (cdr xor-structure))
(and A (not B))
1:=> (car (cdr (car (cdr xor-structure))))
A
1:=> (car (cdr (cdr (car (cdr xor-structure)))))
(not B)
```

Listing 9.3: List selectors

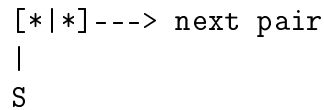
Although Scheme represents programs and data in the same way, it does not use the same methods to deal with source code and lists. In the case of

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 78), 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:



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.

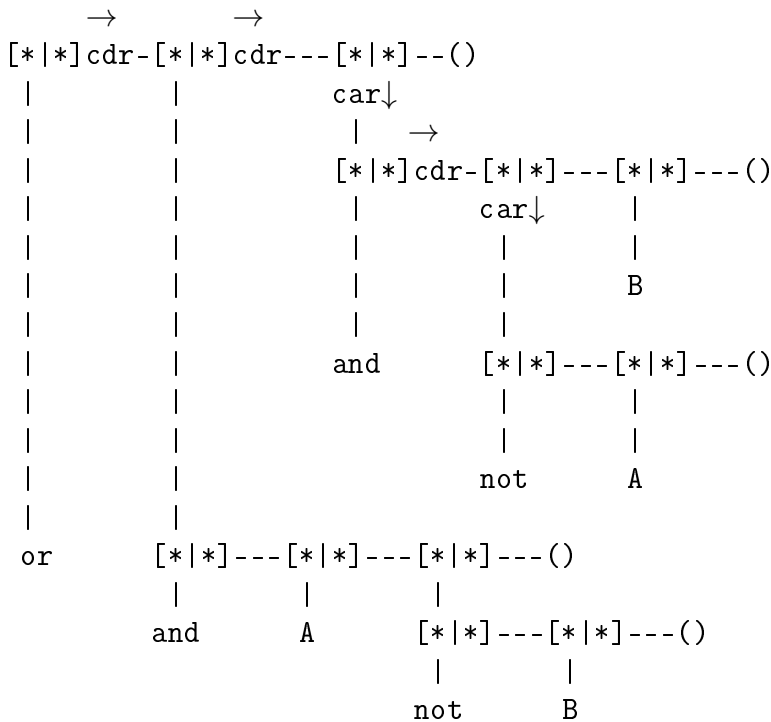


The operation `(car xs)` produces the first pointer of the `xs` chain of pairs. In the case of the `'(S P Q R)` list, `(car xs)` returns `S`. On the other hand, `(cdr xs)` yields the pair after the one pointed out by `xs`, i.e., `'(P Q R)`. A sequence of `cdr` applications permits the user to go through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

<pre> (define spqr '(S P Q R)) </pre>
<pre> 1:=> (load "spqr.scm") spqr 1:=> spqr (S P Q R) 1:=> (cdr spqr) (P Q R) 1:=> (cdr (cdr spqr)) (Q R) 1:=> (cdr (cdr (cdr spqr))) (R) </pre>

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 9.3, 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 \rightarrow arrow, while the `car` operation is acts like a down \downarrow arrow.



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

```

(define 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 \rightarrow arrow is equivalent to a `cdr` operation, and each down \downarrow arrow can be interpreted as a `car` operation, she must perform `(car (cdr (car (cdr (cdr xs)))))` to reach the goal.

9.7 Processing text from the editor

Let us learn how to retrieve a selection such as "(2016 8 31)" from emacs, transform it into the '(2016 8 31) list, and calculate the corresponding day of the week.

To select the string, press **C-Spc** at the first character, and move the cursor onto the last character.

The problem is that the clipboard contains a string like "(2016 8 31)", and the `zeller` function defined in needs a list. Therefore, Nia needs a procedure to read a list from a string of characters.

All languages open ports for reading things from files. Lisp is no exception, but for the fact that one can open objects like strings, and read complex objects such as a whole list. These powerful features can be seen in the above definition. The `let*` was necessary in the definition of `read-string` because Nia needs the result of `(read port)`, and cannot close the port before finishing the reading. The solution for this problem is to execute `(read port)`, the `(close-input-port port)`, and finally return the produce of `(read port)`, which is stored in the variable `res`.

To solve the proposed problem, you will need to load the `readfromstring` file and also the definition of Zeller's congruence, given on page 87.

```
;; File readfromstring.scm

(define (rdstr str)
  (let* [ (port (open-input-string str))
          (res (read port))]
    (close-input-port port) res) )

#|
1:=> (load "readfromstring.scm")
readfromstring.scm
1:=> (load "zellerCond.scm")
zellerCond.scm
1:=> (zeller 2016 8 31)
Wednesday
1:=> (apply zeller (rdstr "(2016 8 31)"))
Wednesday
|#
```

As you can see above, you can place text between `#|` and `|#` to mark it as comment. From now on, this book will show how to use a program by placing an example between comment marks.

9.8 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))  
(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 78, 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.

```
(define-syntax while-do  
  (syntax-rules ()  
    [ (kwd condition the-return bdy ...)   
      (do () ((not condition) the-return) bdy ...)]))
```

```
(define (reverse-iota m)  
  (let [(s '()) (i 0)]  
    (while-do (< i m) s  
      (set! s (cons i s))  
      (set! i (+ i 1)) )))
```

```
#|
```

```
1:=> (load "whiledo.scm")
```

```
whiledo.scm
```

```
1:=> (reverse-iota 5)
```

```
(4 3 2 1 0)
```

```
|#
```

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.

In the above macro definition, the **syntax-rules** have the form

```
(syntax-rules (keyword ...) clause ...)
```

Clauses have the form (**pattern template**), where each pattern shows one possible syntax. The templates specify how the pattern should be expanded to create a valid Lisp expression. A pattern may contain keywords, variables and ellipses. An ellipsis ... indicates that the previous expression can be followed by other expressions with the same shape. The syntax of Lisp requires that a blank space is inserted before and after an ellipsis. A keywords is an identifier that only matches itself.

Chapter 10

Recursion

The mathematician Peano invented a very interesting axiomatic theory for natural numbers. I cannot remember the details, but the idea was something like the following:

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 natural number.

Did you get the idea? For this very idea can be applied to many other situations. When they asked Miyamoto Musashi, the famous Japanese Zen assassin, how he managed to kill a twelve year old boy protected by his mother's 37 samurais, 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 I defeated the remaining 34...

...

To defeat 2, I defeated one of them, then I defeated the other.

A close look will show that the function **App** of Listing 10.1 acts like Musashi. The first clause of (**App** **xs** **s**) states that appending a (**null?** **xs**) list to **s** produces **s**. The second clause of (**App** **xs** **s**) states: To append a **xs** list to an **s** list, rewrite the calling (**App** **xs** **s**) expression as

```
(cons (car x) (App (cdr xs) s))
```

Let us pick a more concrete instance of the problem. The calling pattern `(App '(1 2 3) '(4 5))` matches to `(App xs s)` with `xs= '(1 2 3)` and `s= '(4 5)`. Since `(null? xs)` is false, the second clause rewrites the calling pattern as:

```
(cons (car xs) (App (cdr xs) s))
```

which the inference engine reduces to

```
(cons 1 (App '(2 3) '(4 5)))
```

 (10.1)

The `(App '(2 3) '(4 5))` subpattern matches once again with `(App xs s)`, this time making `xs= '(2 3)` and `s= '(4 5)`. Since `(null? xs)` is false, the second clause holds, and the `(App '(2 3) '(4 5))` is rewritten as

```
(cons 2 (App '(3) '(4 5))).
```

Therefore, equation 10.1 becomes:

```
(cons 1 (cons 2 (App '(3) '(4 5))))
```

 (10.2)

The calling pattern `(App '(3) '(4 5))` matches to `(App xs s)` with `xs= '(3)` and `s= '(4 5)`. For the last time, the second clause is chosen, and expression 10.2 becomes:

```
(cons 1 (cons 2 (cons 3 (App '() '(4 5)))))
```

 (10.3)

The pattern `(App '() '(4 5))` matches to `(App xs s)` with `xs= '()` and `s= '(4 5)`. This time, `(null? xs)` is true, and the calling pattern reduces to `s`, which is equal to `'(4 5)`, and expression 10.3 can be rewritten as:

```
(cons 1 (cons 2 (cons 3 '(4 5))))= '(1 2 3 4 5)
```

<pre>(define (App xs s) (cond [(null? xs) s] [else (cons (car xs) (App (cdr xs) s))]))</pre>
<pre>1:=> (App '(1 2 3) '(4 5)) (1 2 3 4 5)</pre>

Listing 10.1: Append

10.1 Classifying rewrite rules

Typically a recursive definition has two kinds of clauses:

1. Trivial cases, which can be resolved using primitive operations.
2. General cases, which can be broken down into simpler cases.

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

<code>[(null? xs) s]</code>	The first clause is certainly trivial
<code>[else (cons (car xs) (App (cdr xs) s))]</code>	The second equation can be broken down into simpler operations: Appending two lists with one element removed from the first, and then inserting the element left out into the result.

10.2 Quick Sort

Although Hoare's Quick Sort algorithm is of little practical use in these days of BTree everywhere, it gives a good illustration of recursion. The problem that Hoare solved consists of sorting a list.

The `(smaller xs p)` function returns all elements of `s` that are smaller than the `p` pivot. The `(greater xs p)` produces the list of the `s` elements that are greater or equal to `p`. Therefore the `(quick xs)` function partitions `xs` into three lists, then sorts and appends these lists:

- `(smaller (cdr s) (car s))`— numbers smaller than `(car s)`
- `(list (car s))`
- `(greater (cdr s) (car s))`— numbers greater or equal to `(car s)`.

If you sort, then concatenate these three lists, you will end up sorting the original list. A concrete case will make this fact clear. Let `s` be the list `'(4 3 1 5 2 8 7)`. The expression `(quick (smaller (cdr s) (car s)))` returns `'(1 2 3)`. The expression `(quick (greater (cdr s) (car s)))` generates `'(5 7 8)`. Finally, the `(append '(1 2 3) '(4) '(5 7 8))` application returns `'(1 2 3 4 5 7 8)`.

In the quicksort algorithm, there are two trivial cases, the empty `'()` list and lists with a single element, either of which do not require sorting.

Lists with two or more elements are sorted by breaking them into smaller sublists. Since they are closer to the trivial cases, one can assume that the smaller sublists are easier to sort. Figure 10.2 shows a complete implementation of the quicksort algorithm for a list of numbers.

```
;; File: quicksort.scm

(define (smaller xs p)
  (cond [ (null? xs) xs]
        [ (< (car xs) p)
          (cons (car xs)
                 (smaller (cdr xs) p))]]
        [else (smaller (cdr xs) p)] ))

(define (greater xs p)
  (cond [ (null? xs) xs]
        [ (>= (car xs) p)
          (cons (car xs)
                 (greater (cdr xs) p))]]
        [else (greater (cdr xs) p)] ))

(define (quick s)
  (cond [ (null? s) s]
        [ (null? (cdr s)) s]
        [else (append
                   (quick (smaller (cdr s) (car s)))
                   (list (car s))
                   (quick (greater (cdr s)
                                   (car s)))) ] ))

#|
1:=> (load "quicksort.scm")
quicksort.scm
1:=> (quick '(8 4 6 3 9 3))
(3 3 4 6 8 9)
|#
```

Listing 10.2: The quicksort algorithm

10.3 Named-let

The Fibonacci function on page 84 has the undesirable characteristic of requiring two auxiliary arguments. Nia cannot forget to initialize these dummy arguments, otherwise she will get an error message, instead of a result. The named-let permits the creation of functions with initialized arguments, avoiding anomolous definitions that require manual initialization.

```
(define (fibo i)
  (let fn+1 [ (n i) (fn 1) (fn-1 1) ]
    (if (< n 2) fn
        (fn+1 (- n 1) (+ fn fn-1) fn))))

;; 1=> (load "Fibonacci.scm")
;; Fibonacci.scm
;; 1=> (fibo 5)
;; 8
```

Listing 10.3: 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 scheme loops. For instance, since the loop of figure 10.3 is used to calculate the `fn+1` iteration, Nia put the `fn+1` tag on it.

On page 20, you learned how to calculate the average of a list of numbers. Below, you can see another method of performing the same task, this time using a named let.

```
(define (avg xs)
  (let nxt [(s xs) (acc 0) (n 0)]
    (cond [ (and (null? s) (= n 0)) 0]
          [ (null? s) (/ acc n) ]
          [else (nxt (cdr s) (+ (car s) acc)
                      (+ n 1.0))] )))

;; 1=> (load "average.scm")
;; average.scm
;; 1=> (avg '(3 4 5 6))
;; 4.5
```

10.4 Reading data from files

Figure 10.4 reads and prints a file line by line. However, before printing a string, (`port->lines p`) prefixes it with a commented line number. The program should not print commented lines and empty lines.

```
(define (port->lines p)
  (let next-line ( (i 1) (x (read-line p)) )
    (cond [ (eof-object? x) #t]
          [ (< (string-length x) 1)]
          [ (char=? (string-ref x 0) #\;)]
          [else (display "#| ")
                (display (number->string i))
                (cond [ (< i 10) (display " |# ")]
                      [else (display " |# ")]
                (display x) (newline)
                (next-line (+ i 1) (read-line p)))] )))

(define (rdLines filename)
  (call-with-input-file filename port->lines))

;; 1:=> (load "prtFile.scm")
;; prtFile.scm
;; 1:=> (rdLines "average.scm")
;; #| 1  |# (define (avg xs)
;; #| 2  |#   (let nxt [(s xs) (acc 0) (n 0)]
;; #| 3  |#     (cond [ (and (null? s) (= n 0)) 0]
;; #| 4  |#       [ (null? s) (/ acc n) ]
;; #| 5  |#       [else (nxt (cdr s) (+ (car s) acc)
;; #| 6  |#         (+ n 1.0))] )))
```

Listing 10.4: Lines from file

Lisp has a cleverly designed input system. The `call-with-input-file` procedure has two arguments. The first argument is the file name. The second argument is a single parameter function such as `port->lines`. In the example of figure 10.4, Lisp opens a file, and pass the file descriptor to the `port->lines` procedure.

The definition of (`port->lines p`) assigns a line that it reads from port `p` to the variable `x`. Then `x` is printed, and loop proceeds to read the next line. The iteration stops when end of file is reached.

10.5 Writing data to files

Let us assume that Nia needs a Scheme program that translates markdown to html. She wants to use the program to publish poems in the internet. Here an example of markdown:

```
# 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.

On desperate seas long wont to roam,
Thy hyacinth hair, thy classic face,
Thy Naiad airs have brought me home
To the glory that was Greece,
And the grandeur that was Rome.

Lo! in yon brilliant window-niche
How statue-like I see thee stand,
The agate lamp within thy hand!
Ah, Psyche, from the regions which
Are Holy-Land!
```

The program of listing 10.5 reads a markdown file line by line. If the line starts with the `##` prefix, it will be wrapped in the `<h2>...</h2>` html tag. If its first char is the `#` prefix, but the second char is different from `#`, it will be wrapped in the `<h1>...</h1>` html tag.

The simplified version of the html generator of listing 10.5 treats only titles, line breaks and paragraphs. However, the interested reader will be able to add other html elements.

You have already learned how the procedure `call-with-input-file` works. If you don't remember, take a look at page 100.

The procedure `call-with-output-file` can be applied to a file name and a function with an output port as argument. One could define the output port function, but usually people use a lambda abstraction to perform the magic. Read again the explanation about the lambda abstraction on pages 89. File input/output must be mastered to perfection.

```

(define get-line read-line)

(define (tag i <h> x </h> )
  (let [ (Len (string-length x))]
    (string-append <h>
      (substring x i (- Len 1)) </h> "\n")))

(define (subtitle? x)
  (and (> (string-length x) 4)
    (char=? (string-ref x 0) #\#)
    (char=? (string-ref x 1) #\#)))

(define (title? x)
  (and (> (string-length x) 3)
    (char=? (string-ref x 0) #\#)))

(define (convert in out)
  (let loop ( (x (get-line in)) )
    (cond [ (eof-object? x) #t]
      [ (subtitle? x)
        (display (tag 2 "<h2>" x "</h2>") out)
        (loop (get-line in))]
      [ (title? x)
        (display (tag 1 "<h1>" x "</h1>") out)
        (loop (get-line in))]
      [ (> (string-length x) 1)
        (display x out)
        (display "<br/>\n" out)
        (loop (get-line in))]
      [ else (display "<p/>\n" out)
        (loop (get-line in))] )))

(define (copyFile inFile outFile)
  (call-with-output-file outFile
    (lambda(out) (call-with-input-file inFile
      (lambda(in) (convert in out)))) )))

;; 1:=> (load "prtFile.scm")
;; prtFile.scm
;; 1:=> (load "tohtml.scm")
;; tohtml.scm
;; 1:=> (copyFile "helen.md" "helen.html")

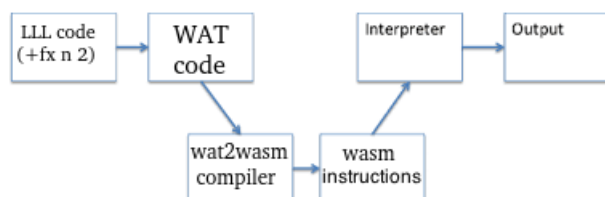
```

Listing 10.5: Writing data to a file

Chapter 11

Cloud computing

A compiler is a program that reads source code into a language that human beings can handle easily and translates the aforementioned code to an object language that computers can interpret. For instance, Nia writes source code in Low Level Lisp (LLL). The `wasCm` compiler translates it into WebAssembly Text, or *wat* for short. Then the `wat2wasm` compiler translates *wat* code into instructions that the computer can carry out efficiently.



The problem is that the `wat2wasm` compiler installed on Nia's machine deals with a language called *wat*, while Nia works with LLL. Nia can work in a language different from the compiler only because the `wasCm` preprocessor provides a bridge between LLL and the `wat2wasm` compiler.

In this chapter, you will learn how to write a web application in pure scheme. Thus, you can leap frog the translation from LLL to *wat*. But firstly, let us learn what cloud computing is.

Cloud computing is an Internet based computer that provides shared processing on demand for those people connected to a service provider. When a company adheres to cloud computing, it avoids infrastructure costs, such as purchasing servers, and keeping them in working order.

The <https://www.cloudorado.com> site shows that provider fees can be pretty steep. However, one can hire a normal hosting service and use it for creating pages that are able to run quite interesting applications.

When you hire a hosting service, you will receive a user name and will be requested to buy a domain name. Let us assume that Nia owns the

`medicina.tips` domain, where `MEDICINA` means health care in Latin. The domain may be any sequence of chars, but it is a good idea to choose names that raise interest in Web users. Since doctors usually know some Latin, if one wants to offer expert medical consultancy, `medicina.tips` is a good name. Now that Nia has a user name and a domain, she must install scheme servelets in her cloud machine. Here is how she enters the shell:

```
~$ ssh -p2222 nia@medicina.tips
nia@medicina.tips's password: ***
```

The 2222 port is used in the secure shell protocol that connects Nia to the cloud computer. She needs a password to complete the connection.

```
Last login: Sun Sep 25 18:03:00 2016 from 179.155.71.6
~$ cd public_html
~/public_html$ mkdir lsp
~/public_html$ cd lsp
~/public_html/lsp$ echo "Addhandler cgi-script .k .lsp" > .htaccess
```

The `.htaccess` file specifies an extension for running lisp scripts on the web.

The cloud terminal is called jailshell. The jailshell does not allow Nia to access the `/usr/local` folder. Therefore, she needs to send a statically linked Bigloo program to her cloud home directory.

```
~/Femto-Emacs/scheme$ scp -P2222 greet.k nia@medicina.tips:~
nia@medicina.tips's password: ***
```

An important note: The `scp` command sends files to the cloud. In the example, Nia sends the `greet.k` servlet to her `nia@medicina.tips` home folder. N.B. The `scp` command takes the `-P2222` port directive with uppercase P, but the `ssh` command requires lowercase p.

Again, all the above procedures are easier to perform than to explain. The best approach is to hire a private tutor who is majoring in Computer Science. Usually, two or three classes are enough to learn jailshell, `ssh` and `scp`. If you do not want to spend money with a tutor, you can ask for technical support from your hosting service. If you are a really lucky, the attendant answering your call is sufficiently well informed to help you.

11.1 Script

A program that generates web pages is called a script. Usually, an html page formats the text and presents it on your browser. A dynamic page can use lisp to add calculation to the document.

Nia needs to compile listing 11.1 statically, which means that the executable will not depend on any library. This is necessary, because the host service may not provide the libraries that Lisp needs to work. The line below compiles the `greet.scm` statically in the case of Bigloo.

```
bigloo -static-all-bigloo -copt "-pthread -static" \  
  greet.scm -o greet.k
```

```
(module inss)

(sprintf
  "Content-Type:text/html;charset=utf-8

<html>
  <meta http-equiv=\"Content-Type\"
    content=\"text/html;charset=utf-8\" />
<body>
  <h3>Hello</h3>
  <form name='form' method='get'>
    Name: <input type='text' name='xnome'> <br/>
  </form>
  <h3>Greetings</h3>
  Hello, ~a!
  <h2>Raw Data:</h2>
  <p>~s</p>
</body></html>
"
  (car (fmt (getenv "QUERY_STRING")))
  (getenv "QUERY_STRING"))

(define (fmt s)
  (pregexp-split "&" ;; Split fields
    (pregexp-replace* "\\+|x[a-z]+= "
      (url-decode s) " ") ))
```

Listing 11.1: `greet.scm`

You cannot be sure whether the method to generate static applications is the same for all computer languages, not even whether it will remain the same for Bigloo as the time goes by. Therefore you must discover how to do the magic in the case of your programming environment and host service.

Listing 11.1 shows the structure of the lisp script. The script must print the `Content-type: text/html` header followed by two newline chars.

The `printf` function has templates with slots to be filled. A slot is represented by the `~s` or the `~a` directives. In the case of the example, there are two slots. The first slot will be filled by the following calculation:

```
(car (fmt (getenv "QUERY_STRING")))
```

Let us examine how the `fmt` function works. For ready reference, I will repeat its definition below.

```
(define (fmt s)
  (pregexp-split "&" ;; Split fields
    (pregexp-replace* "\\+|x[a-z]+= "
      (url-decode s) " ") ))
```

There are hundreds of scripts in the world. When somebody fills in a form in scripts such as Chinese ideograms or Cyrillic letters, the script is codified as a sequence of 2, 3 or 4 bytes. This codification is called utf8. The function application (`url-decode s`) will translate utf8 to the normal character, that the browser can render. The `pregexp-split` and `pregexp-replace*` are regular expression processing tools, and they are used for removing irrelevant information from the form.

The script of listing 11.1 does its magic on the web-server where your page is hosted. It would be a good idea to perform tasks such as utf8 processing on the client browser. WebAssembly was invented for performing such tasks.

Chapter 12

Bugs

It is possible to divide computer errors into three groups:

1. Syntax errors. Code or data do not obey the grammar rules of the language chosen for writing the program.
2. Type errors. The arguments of a function are not of the right type.
3. Specification errors. The code works, but fails to carry out its original specification. The client does not get what he paid for.

In Lisp, forgetting to close parentheses is about the only way to commit a syntax error. After all, the Cambridge prefix notation can be resumed in nested lists. An example will help you to understand what a type error is. The factorial of n is the product of all integers between 1 and n . For instance, the factorial of 5 is given by $1 \times 2 \times 3 \times 4 \times 5$. Let us build a table for the first eight entries of the factorial function.

0	1	2	2	4	24
1	1	3	6	5	120

If you take a careful look at this table, you will note that it has an interesting property: The factorial of n is equal to $n \times \text{factorial}(n-1)$. Then, the factorial of 5 is equal to $5 \times \text{factorial}(4)$. This property is used in the definition of the factorial function given in listing 12.1.

```
(define (fac n #!optional (i 1) (f 1) (next-f (*fx i f)))  
  (if (>=fx i n) next-f  
      (fac n (+fx i 1) next-f) ))
```

Listing 12.1: Factorial function

The domain of the `factorial` function is \mathbb{Z} . For `fx` operations, the \mathbb{Z} set is approximated by the `integer` type, for which the elements must lie in the interval from -9223372036854775808 to 9223372036854775807.

If the result of `(fac n)` is greater than 9223372036854775807, scheme will produce garbage. For instance, `(fac 5)` produces the correct result: 120. However `(fac 21)` returns a meaningless sequence of digits. How to know whether `(fac 19)` produces a correct result? One must design clever tests. For instance, the expression `(quotient (fib n) (fib (- n 1)))` must return `n`. If it does not, the computer is generating garbage.

Since the expression `(= (quotient (fib n) (fib (- n 1))) n)` is true for all values of the `n` variable, it is called *invariant* by people who investigate methods for the writing of correct programs.

```
(define (fact n #!optional (i 1) (f 1) (nxt (*fx i f)))
  (when (not (= (quotient nxt f) i))
    (error 'overflow n 'assert-failed))
  (if (>=fx i n) nxt
      (fact n (+fx i 1) nxt) ))
```

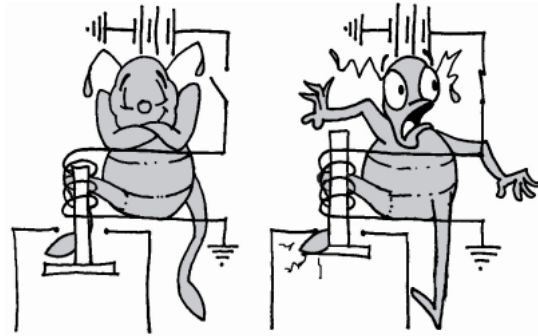
```
1:=> (load "src/factorial.scm")
src/factorial.scm
1:=> (fact 15)
1307674368000
1:=> (fact 21)
(assert-failed (= (quotient next-f f) i))
```

Listing 12.2: Factorial function

In order to avoid specification errors, it is necessary to use logical statements to write a contract between the computer engineer and the client. The contract has three assertions:

1. A precondition that must be true of the parameters for calling the contracted function.
2. A postcondition that is true after running the evaluator. The result returned by the contracted function must be what the client desires.
3. An invariant that is true before and after the execution of the code. The `(not (= (quotient next-f f) i))` expression interrupts the computation if the invariant ever becomes false, which means that there is an error in the program.

12.1 Bugs



The first computer was constructed by Konrad Zuse, a German civil engineer, and his assistant, Ms. Ursula Walk, née Hebekeuser. The first computers, like those of Zuse and Walk, were based on relays. These are bulky electrical devices, typically incorporating an electromagnet, which is activated by a current in one circuit to turn on or off

another circuit. Computers made of such a contrivance were enormous, slow, and unreliable. Therefore, on September 9th, 1945, a moth flew into one of the relays of the Harvard Mark II computer and jammed it. From that time on, *bug* became the standard word to indicate an error that prevents a computer from working as intended.

Due to bugs, compilers of languages like Clean and Haskell frequently return error messages, instead of generating code and running the corresponding programs. The Steel Bank Common Lisp language does not interrupt code generation when the compiler spots a bug, all the same it does issue warnings that help find the problem before the embarrassment of failure being manifest on the client's terminal. Scheme does not possess such a kind of oracular compiler. It report errors only when you execute the code.

12.2 Missing parenthesis

There is only one kind of bug that scheme reports at compile time: missing parenthesis. This bug is annoying, because the compiler does not report its position accurately. For example, try to compile the code of Listing 12.3. The compiler issues the following error message:

```
File "src/manydefs.scm", line 12, character 159:
#(define (fn+1 n fn fn-1)
#^
*** ERROR:read
Unexpected end-of-file -- Unclosed list
1. load, src/manydefs.scm:8
```

The error is on line 15, where a right parenthesis is missing. However, the error message is misleading, as it points to the wrong line and does not mention the parenthesis.

Whenever you receive a message of ‘unexpected end-of-file’, my advice is to look for a missing parenthesis.

```
;; File: fvalue.scm

(define (future-value pv i n)
  (* (expt (+ (/ i 100.0 1) n)
      pv))
) ;;end define

(define (fat n)
  (if (< n 1) 1
      (* (fat (- n 1)) n)))

(define (fn+1 n fn fn-1)
  (if (< n 2) fn
      (fn+1 (- n 1)
              (+ fn fn-1) fn)))

(define (quick xs)
  (if (null? xs) xs
      (append
        (quick (filter (lambda(x) (< x (car xs)))
                        (cdr xs)))
        (list (car xs))
        (quick (filter (lambda(x) (>= x (car xs)))
                        (cdr xs))) )))

1:=> (load "src/manydefs.scm")
File "src/manydefs.scm", line 12, character 159:
#(define (fn+1 n fn fn-1)
#^
*** ERROR:read
Unexpected end-of-file -- Unclosed list
```

Listing 12.3: Future Value Program

Chapter 13

Finite Automaton

A deterministic finite automaton is a finite state machine that accepts or rejects finite strings of chars and only produces a unique computation for each input string. Figure 13.1 below illustrates a deterministic finite automaton using a state diagram. The aforementioned automaton recognizes words and punctuation marks from a text using the Roman alphabet, which is the same as in English. Therefore, this automaton can accept a stream of words and punctuation from an English text.

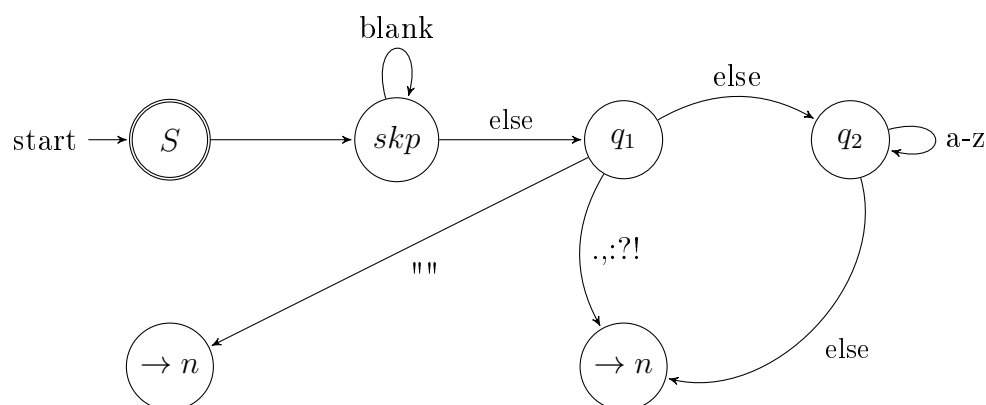


Fig. 13.1: Finite automaton that recognizes English words

In the automaton of figure 13.1, there are three states: skp , q_1 and q_2 . The automaton remains in the state skp as long as the stream produces blank chars, i. e., `#\space`, `#\newline` or `#\tab`. If the computation reaches the end of the string, the automaton stops. If a punctuation mark appears in the stream, the automaton accepts it, and returns $\rightarrow n$. If the q_1 state receives an alphabetical letter, the automaton goes to state q_2 and remains there as long as the reader keeps feeding letters.

```

(define (end? s n) (<= (string-length s) n))

(define (pct? s n)
  (string-char-index ".,?" (string-ref s n)))

(define (blk? s n)
  (string-char-index " \n" (string-ref s n)))

(define (letter? s n)
  (and (not (pct? s n)) (not (blk? s n)) ))

(define (skp n s)
  (if (or (end? s n) (not (blk? s n))) n
      (skp (+ n 1) s)))

(define (q1 n s)
  (cond ( (end? s n) n)
        ( (pct? s n) (+ n 1) )
        (else (q3 n s)) ))

(define (q3 n s )
  (cond ( (end? s n) n)
        ( (letter? s n) (q3 (+ n 1) s))
        (else n)))

(define (tkz s #!optional (n 0)
          (i (skp n s))
          (j (q1 i s)))
  (cond ( (end? s i) '())
        ( (end? s j) (list (substring s i j)))
        (else (cons (substring s i j) (tkz s j)) ) ))

```

Listing 13.2: Finite Automaton in Scheme

```

1:=> (load "src/auto-one.scm")
src/auto-one.scm
1:=> (tkz "She walks in beauty, like the night")
(She walks in beauty , like the night)

```

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. Princeton University Press, 1986.
- [7] R. Kent Dybvig. The Scheme Programming Language. Available from <http://www.scheme.com/tspl4/>

Index

- Aji Saka, 63
- Arithmetic
 - Elementary School, 8
- Bugs, 107
 - missing parenthesis, 109
- call-with-input-file, 100
- call-with-output-file, 101
- cond
 - clauses, 87
- Conditional execution
 - cond, 21, 87
 - if-else, 75
 - True and False, 75
- define, 16, 18, 110
- File, 100
- Finite automata, 111
- Future Value, 14
- Home directory, 25
- Input from file, 100
- Interest
 - Compound, 15
- Japanese, 61
- Japanese alphabet, 67
- Konrad Zuse, 109
 - first computer, 109
 - relays, 109
 - Ursula Walk, 109
- lambda, 89
- let-binding, 73
 - basic let-binding, 73
 - let-star binding, 73
- List
 - at-sign, 93
 - backquote, 93
 - comma, 93
 - cons, 93
 - selector: car, 20
 - selector: cdr, 20
- Local variables, 73
 - let-binding, 73
- Loop
 - named-let, 99
- Macros
 - define-syntax, 93
- Named-let, 99
 - meaningful name for loop, 99
- Output to file, 101
 - call-with-output-file, 101
- Pen drive, 28
- Permissions, 27
- Predicates
 - char?, 82
 - exact?, 82
 - inexact?, 82
 - integer?, 81
 - null?, 21
 - number?, 82
 - pair?, 82
 - string?, 82

- Prefix notation, 9
- Present value, 14
- Prince Kalunga, 63
- Recursion, 95
 - Append
 - Definition, 95
 - Classifying rules, 97
 - General case, 97
 - Peano's axioms, 95
 - Trivial case, 97
- Recursion in LLL, 58
- Recursion in Scheme, 95
- Shell, 23
 - change dir, 25
 - cp, 27
 - echo, 26
 - ls, 26
 - mkdir, 25
 - mv, 28
 - prompt, 24
 - pwd, 25
 - rm, 28
 - tab, 25
 - wild card, 26
- store command, 46
- String, 82
 - string, 111
 - string-length, 111
 - string.find, 111
 - strings-ref, 111
- Time value of money, 13
- Type, 81
 - Char, 82
 - Int, 81
 - String, 82
- Unwrapped if, 59
- Ursula Hebekeuser, 109
- Use of memory in recursion, 58
- WebAssembly, 43
 - i32, 44
 - Javascript client, 44
 - node, 60
 - Recursion, 58
 - Tail Call Elimination, 58
 - Unwrapped if, 59
 - web page, 45
- Zuse, 109