An introduction to femtolisp

(Content not checked for grammar and spelling errors)

Eduardo Costa Vernon Sipple

Preface

Femto is a tiny Emacs clone, originally derived from a minimalist editor called Atto. Although built upon the C Atto codebase, Femto can be customized using the femtolisp Scheme dialect.

The fundamental paradigm of Emacs is that it rests on a small core, but can be extended and customized through the Lisp programming language.

Istead of requiring from the programmer a compile, build, test and fix cycle — Lisp offers a Read, Eval, Print Loop (REPL).

1. **Read.** The REPL reads a symbolic expression such as `(+ 42 13 4)` from a text terminal. The symbolic expression – sexpr – always has the form: (operation a_1 $a_2 \ldots a_n$), where a_1, $a_2 \ldots a_n$ are the arguments of the operation.

2. **Eval.** The compiler translates the symbolic expression to machine language, then executes the generated code and calculates the corresponding value.

3. **Print.** The value produced in step 2 is printed.

4. **Loop.** The REPL returns to step 1.

Since Lisp is usually compiled, it is much faster than interpreted languages. However, to facilitate portability, femtolisp does not generate native code, but compiles symbolic expressions to a virtual machine language. As a consequence of this choice, femtolisp is not as fast as other Lisp dialects, such as sbcl and Racket. Notwithstanding, it is about four times faster than Python and forty times faster than Tiny Scheme.

The REPL approach enables fast prototyping of extensions as each idea can be tested individually on the command line inside the editor with an immediate result check.

For historical reasons, the reference GNU Emacs implementation uses a specialised version of lisp known as elisp. However, the elisp dialect is not standard lisp, being used only by the GNU Emacs editor and its derivatives.

Femtolisp was chosen as the extension language for FemtoEmacs for the following reasons:

1. Femtolisp is open source.
2. Femtolisp is small and written in C.
3. In the majority of situations Femtolisp is compatible with the standard Scheme dialect.
4. Femtolisp is fast.

Chapter 1

Intallation

A text editor, like Emacs or femto, 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 confortable on the eyes, since they provide sharper and crisp alphabetical letters. On the other hand, GUI allows for font customization. At present, femtoemacs offers only ncurses, which is a text-based user interface. Therefore, download the most recent version from the distribution site: `https://www.gnu.org/software/ncurses/`

MAC OS-X. Unpack and make the distribution archive as shown below:

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

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

```
/usr/local/lib/libncursesw.a
```

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

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

After installing libncursesw, enter the folder `Femto-Emacs/` and build the editor for Macistosh:

```
~/$ cd ~/Femto-Emacs/
~/Femto-Emacs$ make -f Makefile.macosx
```

Linux. Use apt-get to install the latest version of `libncursesw5-dev`. Since Linux repositories are usually old, download a recent source distribution of ncurses, build it, and install it on top of the apt-get version, thus:

```
~$ sudo apt-get install libncursesw5-dev
~$ wget http://ftp.gnu.org/gnu/ncurses/ncurses-6.0.tar.gz
~$ tar xfvz ncurses-6.0.tar.gz
~$ cd ncurses-6.0/
~/ncurses-6.0$ ./configure --enable-widec
~/ncurses-6.0$ make
~/ncurses-6.0$ sudo make install
```

After installing ncursesw, you can make the editor:

```
~/$ cd ~/Femto-Emacs/
~/Femto-Emacs$ make
```

Installation is achieved by copying the compiled files to `/usr/local/bin/` and the customization files to your home directory.

```
~/Femto-Emacs/femto$ sudo cp femto /usr/local/bin/
~/Femto-Emacs/femto$ sudo cp femto.boot /usr/local/bin/
~/Femto-Emacs/femto$ cp init.lsp $HOME/
~/Femto-Emacs/femto$ cp aliases.scm $HOME/
```

1.1 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.

Fig. 1.1: Running total

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

```
+      8
      26
      85
       3
-----------
     122
```

Subtraction is not treated differently:

```
-    358
     216
-----------
     142
```

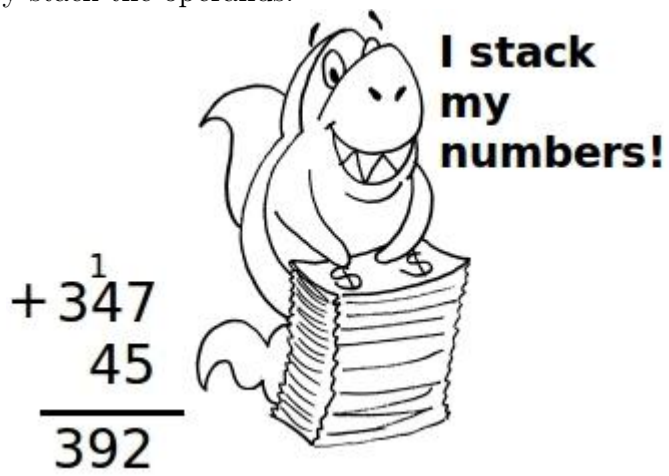
This chapter contains an introduction to the Cambridge prefix notation, which is slightly different from the one you learned in elementary school: The operation and its arguments are put between parentheses. In doing so, one does not need to draw a line under the bottom number, as you can see below:

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

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

1.2 Cambridge prefix notation

Let us summarize what we have learned until now. In pre-algebra, students learn to place arithmetic operators (+, -, × and ÷) between their operands; e.g. 347+45. However, when doing additions and subtractions on paper, they stack the operands.



Lisp programmers put operation and operands between parentheses. The right parenthesis separate the operands from the result, instead of drawing a line under the last operand.

1.3 Pictures

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

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

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

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

Delighted with the selection of pics, the Texan in an 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 femtoemacs 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:

```
~/Femto-Emacs$ femto finance.scm
```

Listing 1.2 shows what an interaction with a femto buffer looks like.

```
(+  (* 3 25320)
    (* 6 56750)
    (* 2 136000)
    136000 )Esc ]
824460
```

Listing 1.2: The finance.scm buffer

After typing an expression obeying the Cambridge prefix notation syntax, the bookkeeper moves the cursor to the front of the most external right parenthesis. To perform the calculation, he presses and releases the Esc key, then presses and releases the] key. The femtolisp interpreter inserts the result into the buffer. To save the buffer, the bookkeeper keeps the Ctrl key down, and presses the x and s keys in sequence. To exit the editor, he maintains the x pressed and hits the x and c keys one after the other. If there are unsaved files, the editor warns that modified buffers exist, and requires confirmation before quitting.

1.4 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.

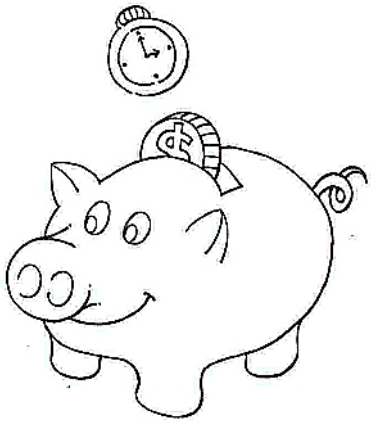
Unfortunately for you, the above scenario would more than probably not happen in real life. The right to delay payment until some future date is a merchandise with a price tag, which *is called interest by those who think it lawful, and usury by those who do not* (William Blackstone's Commentaries on 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.

1.5 Future value



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

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

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

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

$$FV = PV \times (1 + i)^n \tag{1.1}$$

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.

Postponed payment. 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 stongly recomend against postponing the payment. You have learned how to calculate an expression by pressing the Esc] command. However, there is another way of evaluating an expression.

When you open the femtoemacs editor, there is a one line buffer at the bottom of the screen. If you keep the Ctrl key pressed, and strike the o key, the editor will read and evaluate a Cambridge prefix expression from the minibuffer.

```
(+  (* 3 25320)
    (* 6 56750)
    (* 2 136000)
    136000 )Esc ]
824460
```

```
> (* 100000.00 (expt (+ 1.0 0.025) 3))Enter
107689.06
```

Listing 1.3: Command from the minibuffer

1.6 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 recomendation 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 femtoemacs, 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 1.1 in the minibuffer:

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

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

It is possible to define an operation that calculates formula 1.1 given the arguments `fv`, `i` and `n`. The definition is given in listing 1.4. Start the editor from a text terminal:

```
~$ mkdir financing
~$ cd financing
~/financing$ femto fvalue.scm
```

After typing the definition shown in listing 1.4, 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 femtolisp, press Ctrl o and type `(load "fvalue.scm")` in the minibuffer, as shown in listing 1.4. After the loading operation, you can press the Ctrl o command again, and type

```
(future-value 10000 8 2)
```

in the minibuffer.

```
;; File: fvalue.scm

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

```
(load "fvalue.scm")Enter
#<function>
(future-value 10000 8 2)Enter
11644.00
```

Listing 1.4: Future Value Program

In listing 1.4, 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 femtolisp, and have the simple function of helping people.

1.7 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 femtolisp 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 1.2. This formula is implemented in listing 1.5.

$$\frac{r \times p \times (1+r)^n}{(1+r)^n - 1} \tag{1.2}$$

```
;; File: pmt.scm

(define (pmt
             p ;; present value
             i ;; interest
             n ;; number of periods
             ;; optional parameters
                  (r (/ i 100.0))
                  (rn (expt (+ 1.0 r) n))
        );;end of parameter list
  (/ (* r p rn)
     (- rn 1))
);;end of define
```

```
(load "pmt.scm")Enter
#<function>
(pmt (* 10500000.00 0.7) (/ 10.0 12) (* 20 12))Enter
70929.00
```

Listing 1.5: Mortgage for a God

You certainly noticed that the `pmt` function, defined in listing 1.5, has two parameters `r` and `rn`, which the user does not need to express. Femtolisp itself assigns `(/ i 100.0)` to `r` and `(expt (+ 1.0 r) n)` to `rn`.

1.8 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 becomming proficient.

Lisp has the simplest syntax of any language. All Lisp constructions are expressed in Cambridge prefix notation: `(operation` x_1 x_2 $x_3 \ldots x_n$`)`. This means that Lisp commands, macros, procedures and functions can be written as a pair of parentheses enclosing an operation followed by its arguments. Remember, there is no exception to this rule.

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`.

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

The `'(S P Q R)` list of letters is prefixed by a quotation mark because in Lisp lists and programs have the same syntax: The Cambridge prefix notation. The single quotation mark tags the list so the computer will take it at face-value, and 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.

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

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

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

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

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

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

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

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

1.9 Going through a list

In listing 1.6, 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 35. The only reason for presenting a complex definition like `avg` so early in this tutorial is to show a grouping pattern of five right parentheses. As was discussed previously, when the number of close parentheses is greater than 3, good programmers distribute these into small groups, so that an individual who

is trying to understand the program can see that the expression is proprely closed at a glance.

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

```
(define (avg s (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)) ) ))
```

```
(load "average.scm")Enter
#<function>
(avg '(3 4 5 6))Enter
4.5
```

Listing 1.6: Going through a list

The cond-form consists of a sequence of clauses. In a function definition, a clause has two components: A condition and an expression. The value of the cond-form will be given in the first clause that has a true condition. A concrete example will make this statement clear. In the case of listing 1.6, 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` looses an 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)` maches the `(null? s)` clause, that calculates the averaging `(/ acc n)` expression with `acc= 18` and `n= 4`.

Chapter 2

Femtoemacs commands

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

- `C-`κ – Press and release Ctrl and κ simultaneously

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

- `C-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-s` – save file.
  `C-x C-c` – exit femto.

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

- `C-x` κ – Press and release Ctrl and x together, then press κ

  `C-x n` – switch buffers.
  `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.

- `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` – go to the line given in the minibuffer.
  `M-n` – activate the next buffer.
  `M-r` – query replace.

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

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 typing the text that she wants to localize, Nia presses `C-s` again to start the search. For abandoning an ongoing command, she may press `C-g`.

To transport a region to another place, Nia presses `C-Spc` to start selection and move the cursor to select the region. Then she presses `C-w` to kill the 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.

Chapter 3

Arithmetic operations

The femto editor is designed for writing programs. Therefore, Nia decided to learn how to use it through its own scripting language, that is femtolisp.

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 `C-x 2` to create a window for interacting with

Lisp. After this action, there are two windows on the screen, both showing the `celsius.scm` buffer. The `C-x o` command makes the cursor jump from one window to the other. From the bottom window, Nia presses `C-x C-f` to create the `repl.scm` interaction file.

On the `repl.scm` buffer, Nia can load the `celsius.scm` program and call any application that she has defined herself, or which comes with Lisp. The `C-o` command reads an operation from the minibuffer and executes it. The first operation is loading the `celsius.scm` file, as you can see in figure 3.1. Be sure that you are on the `repl.scm` interaction window when you perform the loading, otherwise you will introduce the `#<function>` extraneous text into the source file.

Let us recall what happened until now. Nia pressed `C-x 2` to split the buffer window, as shown in figure 3.1. Thus she has two buffers in front of her, each with its own window. One of the buffers has a Lisp source file, while the other contains a Lisp interaction. In order to switch from one buffer to the other, Nia press the `C-x o` command.

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

The `C-o` command followed by `(load "celsius.scm")` on the minibuffer loads the source file.

One can also load the source file by typing `(load "celsius.scm")` on the interaction buffer, then press the `Esc ]` command. After loading the source file either with `C-o` or with the `Esc ]` method, Nia types `(c2f 100)`

```
; 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
```

```
#<function>

(c2f 100)
212.0
```

```
> (load "celsius.scm")
```

Listing 3.1: Source file and Interation Buffer

on the interaction buffer. Then she presses `Esc ]` to insert the calculation of the Fahrenheit reading into the interaction buffer.

Nia often works with a single window. In this case, after typing and saving the source file with `C-x C-s`, she does not split the window before opening the interaction buffer with the `C-x C-f` command. In this case, only one buffer is visible. Nia switches between the interaction buffer and the source file by pressing the `C-x n` command.

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

3.1 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 3.2. 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. Therefore, one must open two parentheses in front of the first pair, one for the list of pairs and the other for the first pair.

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

- *decimal* – base ten.
- *decimeter* – tenth part of a meter.
- *decimate* – the killing of every tenth Roman soldier that performed badly in battle.

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

Rome was purportedly founded by Romulus, who designed a calendar. March was the first month of the year in the Romulus calendar. Therefore, if Nia wants the order for a given month in this mythical calendar, she must subtract 2 from the order in the Gregorian calendar. In doing so, September

becomes the seventh month; October, the eigth; November, the nineth and December, therefore, the tenth month.

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

```
(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))
```

```
(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)) )
     (mod (+ day
             (quotient (- (* 13 roman-month) 1) 5)
             decade
             (quotient decade 4)
             (quotient century 4)
             (* century -2)) 7)) )
```

```
> (load "zeller.scm")
#<function>
> (zr 2016 8 31)
3
```

Listing 3.2: Source file and Interation Buffer

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 3.2 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 3.2 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 3.2, expressions such as `(zr 2016 8 31)` return a number between 0 and 6, corresponding to Sunday, Monday, Tuesday, Wednesday, Thursday, Friday and Saturday.

3.2 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.

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.

```
(define (zr 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)) )
     (mod (+ day
             (quotient (- (* 13 roman-month) 1) 5)
             decade
             (quotient decade 4)
             (quotient century 4)
             (* century -2)) 7)) )
```

```
> (load "zeller.scm")
#<function>
> (zr 2016 8 31)
3
```

Listing 3.3: Making decisions

Chapter 4

Sets

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

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

4.1 Sets of numbers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In femtolisp, 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 pair 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)))
```

```
> (load "cartesian.scm")
#<function>
> (add f g)
(22 . 5) (add '(3 . 7) '(2 . 5))
(29 . 35)
```

Listing 4.1: 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 addition of two rational numbers P and Q is given by tho following expression:

$$\frac{X_a}{X_d} + \frac{Y_a}{Y_d} = \frac{X_a \times Y_d + Y_a \times X_d}{X_d \times Y_d} \tag{4.1}$$

In the dotted pair notation, X_a= `(car X)`, X_d = `(cdr X)`, Y_a= `(car Y)` and Y_d= `(cdr Y)`. Replacing these values in equation 4.1, we get 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
```

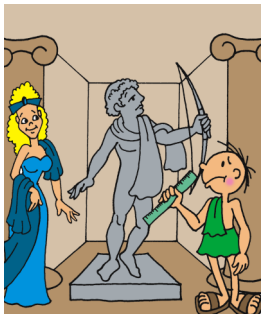
Nia used the above expressions for calculating the numerator and denominator of $X + Y$ in figure 4.1.

4.2 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.

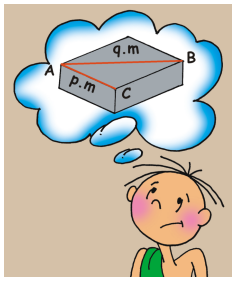
If the Greeks were right, I would be able to find a meter, possibly a very small one, that produces an integer measure for the diagonal of the square, and another integer measure for the side. Suppose that p is the result of measuring the side of the square, and q is the result of measuring the diagonal.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

$$p^2 + p^2 = q^2 \therefore 2p^2 = q^2 \tag{4.2}$$



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 4.2, 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 4.2.

$$2 \times p^2 = q^2 = (2 \times n)^2 = 4 \times n^2 \therefore 2 \times p^2 = 4 \times q^2 \therefore p^2 = 2 \times n^2 \tag{4.3}$$



Equation 4.2 shows that q is even; equation 4.3 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 4.2, 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 with an transfinite number of elements. Therefore, femtolisp and all other programming languages replace sets with the concept of type. In femtolisp, $\mathbb{Z}$ 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, that is subset of $\mathbb{R}$.

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

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

`(+` $x_1\ x_2\ \ldots x_n$`)` – addition
`(*` $x_1\ x_2\ \ldots x_n$`)` – multiplication
`(-` $x_1\ x_2\ \ldots x_n$`)` – subtraction
`(/` $x_1\ x_2\ \ldots x_n$`)` – division
`(pow` $x\ y$`)` – x^y
`(sin` x`)` – $\sin(x)$
`(asin` x`)` – $\arcsin(x)$
`(cos` x`)` – $\cos(x)$
`(acos` x`)` – $\arcsin(x)$
`(tan` x`)` – $\tan(x)$
`(atan` x`)` – $\arctan(x)$
`(log` x`)` – natural logarithm
`(log2` x`)` – logarithm in base 2
`(log10` x`)` – logarithm in base 10

Integers are distinguished from inexact numbers by the `(integer? x)` predicate. Two important integer functions are `(quotient x y)`, that produces the integer division of `x` by `y`, and `(mod x y)` that returns the remainder of the division. You have used these two functions to calculate the Zeller's congruence.

4.3 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 character. 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.

4.4 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`

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

Functions have a parameter, 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',...}`. 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= #\a`*,* `x= #\a`*,* `x= #\a`*,* `x= #\a` *or* `x= #\a`*?* If the answear 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)) ))
```

```
> (load "fibonacci.scm")
#<function>
> (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 characteres 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 $p_2 \ldots p_n$) form evaluates the sequence p_1, $p_2 \ldots p_n$ until it reaches p_n or one of the previous p_i returns `#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 $p_2 \ldots p_n$) 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 wether `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)))
```

4.5 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 (mod (+ day
                       (quotient (- (* 13 roman-month) 1) 5)
                       decade
                       (quotient decade 4)
                       (quotient century 4)
                (* century -2)) 7)) )
```

```
> (load "zeller.scm")
#<function>
> (zeller 2018 8 31)
Friday
```

Listing 4.2: Day of the week

The cond-form controls conditional execution, based on a set of clauses. Each clause has a condition followed by a sequence of actions. Lisp starts with the top clause, and proceeds in decending order. It executes the first clause whose condition produces a value different from `#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.

4.6 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)))
```

```
(load "lists.scm")Esc ]
(or (and A (not B)) (and (not A) B))

(car xor-structure) Esc ]
or
(cdr xor-structure)Esc ]
((and A (not B))
 (and (not A) B))
(car (cdr xor-structure))Esc ]
(and A (not B))
(car (cdr (car (cdr xor-structure)) ))Esc ]
A
(car (cdr (cdr (car (cdr xor-structure)) )))Esc ]
(not B)
```

Listing 4.3: 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 26), each pair containing a pointer to a list element, and another pointer to the next pair. Let us assume that `xs` points to the list `'(S P Q R)`. This list corresponds to the following chain of pairs:

```
[*|*]--->[*|*]--->[*|*]--->[*|*]--->()
 |        |        |        |
 S        P        Q        R
```

The first pair has a pointer to `S`, and another pointer to the second pair. The second pair contains pointers to `P` and to the third pair. The third pair points to `Q` and to the fourth pair. Finally, the fourth pair points to `R` and to the `()` empty list.

```
[*|*]---> next pair
 |
 S
```

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

```
(define spqr
   '(S P Q R))
```

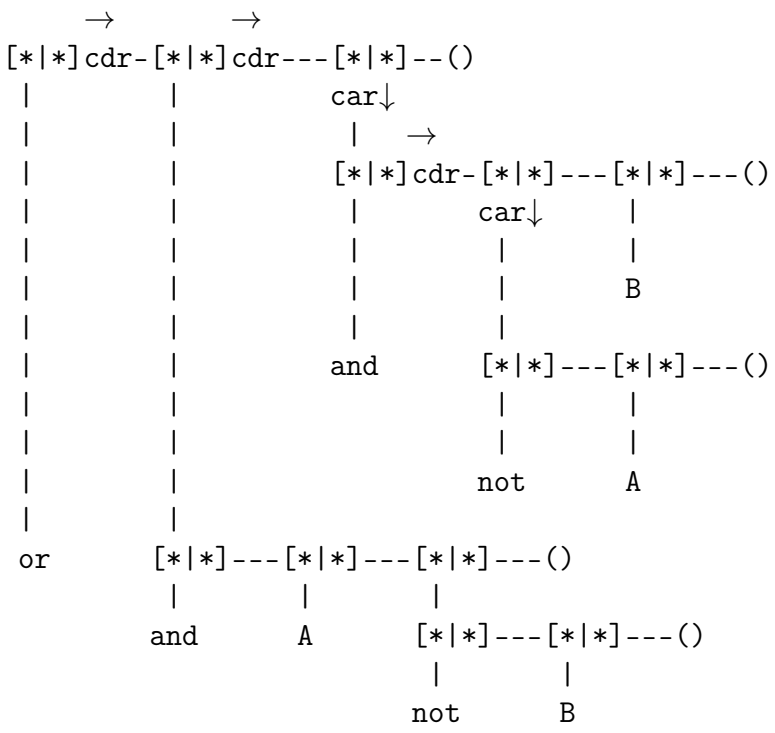
```
(load "spqr.scm")Esc ]
(S P Q R)

(cdr spqr)
(P Q R)

(cdr (cdr spqr))
(Q R)

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

If all one needs is to represent sequences, then contiguous memory elements could be more practical than pairs. However, as one can see in figure 4.3, a list element can be a nested sublist. In this case, the `car` part of a pair can point down to a sublist branch. The diagram below shows that one can reach any part of a nested list following a chain of `car` and `cdr`. In such a chain, the `cdr` operation advances one pair along the list, and the `car` operation goes down into a sublist. The `cdr` operation is equivalent to a right → arrow, while the `car` operation is acts like a down ↓ arrow.



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

4.7 Processing text from the editor

Let us learn how to retrieve a selection such as `"(2016 8 31)"` from femtoemacs, 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.

```
(load (home "aliases.scm"))

(define (read-string str)
    (let ( (port (open-input-string str)))
      (begin0 (read port)
      (close-input-port port)) ))
```

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 `begin0` form evaluates a sequence of expressions, and returns the first one as the result of the whole sequence. The `begin0` 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)`.

The `week-day` function transform a number `n` between 0 and 6 into strings corresponding to the seven days of the week:

```
(define (week-day n)
  (symbol->string
    (cond ( (= n 0) "Sunday")
          ( (= n 1) "Monday")
          ( (= n 2) "Tuesday")
          ( (= n 3) "Wednesday")
          ( (= n 4) "Thursday")
          ( (= n 5) "Friday")
          ( (= n 6) "Saturday")
          (else "This should not happen")) ))
```

Calculating the day of the week. Zeller congruence, as given in figure 4.2 and repeated below for ready reference, can be used to calculate the day of the week.

```
(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 (mod (+ (quotient (- (* 13 roman-month) 1) 5)
                       decade
                       (quotient decade 4)
                       (quotient century 4)
                       (* century -2)
                       day) 7)) ))
```

The last step is to program a command that calculates the day of the week into the editor. Here Nia uses the cond-form again, to select one of many commands that the user may press.

The keys `C-c a`, `C-c b`, until `C-c z` are reserved for customization, which means that you can hook any Lisp program onto them. For instance, if Nia wants to wrap the clipboard contents with the html-tags `<p>` and `</p>`, besides calculating the day of the week, she may define the following keyboard:

```
(define (keyboard key)
   (cond
    ( (equal? key "C-c z")
      (let ( (date (read-string (cutregion)) ))
        (insert (apply zeller date)) ))
      ( (equal? key "C-c p")
        (insert "<p> </p>")
        (backward-char 5))
      ( (equal? key "C-c c")
        (end-of-line)
        (insert "<p>-")
        (insert (clipboard))
        (insert "-</p>"))
      (else (insert key)) ))
```

The `(apply zeller date)` operation applies `zeller` to the three elements of the `date` list, which contains the year, month and day of interest.

;;Syntax highlight.

```
;; define syntax highlighting for scheme files
(newlanguage ".scm" ";" "#|" "|#")
(keyword "define")
(keyword "cond")
(keyword "else")

;; define syntax hughlighting for C code
(newlanguage ".c" "//" "/*" "*/")
(keyword "auto")
(keyword "break")
(keyword "case")
(keyword "char")
(keyword "continue")
(keyword "default")
(keyword "do")
(keyword "double")
(keyword "else")
(keyword "enum")
(keyword "extern")
(keyword "float")
(keyword "for")
(keyword "goto")
(keyword "if")
(keyword "int")
(keyword "long")
(keyword "register")
(keyword "return")
(keyword "short")
(keyword "signed")
(keyword "sizeof")
(keyword "static")
(keyword "struct")
(keyword "switch")
(keyword "typedef")
(keyword "union")
(keyword "unsigned")
(keyword "void")
(keyword "volatile")
(keyword "while")
```

4.8 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))Esc ]
(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 26, 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 the much neater `'(3 4)` list syntax is equivalent to the dotted Cartesian `'(3 . (4 . ()))` pair.

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

Macros are programs that brings a more convenient notation to a standard Lisp form. The syntax of Lisp, that unifies data and programs, makes it possible to create powerful macros that implement Domain Specific Languages (DSLs), speed up coding or create new software paradigms. In fact, the macro system is one of the two features that places Lisp asunder from other languages, the other being its Mathematical Foundations. For learning macros in depth, the reader is referred to Dough Hoyte's book on the subject[4]. The example below can be placed into the `init.lsp` file or into the `aliases.scm` file. Remember that these two configuration files must be copyed to your home directory.

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

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

Nia creates a `repl.scm` buffer and tests the `while-do` macro, as you can see in the following example:

```
(let ( (s '())
       (i 0))
   (while-do (< i 5) n
       (set! s (cons i s))
       (set! i (+ i 1)) )) Esc ]
(4 3 2 1 0)
```

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

Destructive updates are not considered good programming practice. In fact, the `set!` operation has an exclamation mark to remember you that it should not be used lightly. By the way, `set!` is pronounced as *set bang*.

The `while-do` macro is not very useful, since Scheme programmers adopt the functional style, which abhor the use of set bang. A much more interesting macro is the Curry operator, that comes pre-defined thus:

```
(define-macro (curry fn x)
  (let ( (g (gensym)))
    `(lambda(,g) (,fn ,g ,x)) ))
```

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

```
(filter (lambda(x) (> x 4)) '(3 2 4 1 6 9 8))
(6 9 8)
```

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

```
(define (g4 x) (> 4 x))

(filter g4 '(3 2 4 1 6 9 8))
(6 9 8)
```

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

```
(filter (curry > 4) '(3 2 4 1 6 9 8))
(6 9 8)
```

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

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

```
(define (quick-sort s)
   (cond ((null? s) s)
         ((null? (cdr s)) s)
         (else
          (append
             (quick-sort (filter (curry string>? (car s))
                                 (cdr s)))
             (list (car s))
             (quick-sort (filter (curry string<=? (car s))
                                 (cdr s))) ) )))
```

```
> (load "quick.scm")
#<function>
> (quick-sort '("Caecilia" "Anna" "Priscilla"))
(Anna Caecilia Priscilla)
```

Listing 4.4: The filter function

In listing 4.4, the deffinition of `quick-sort` calls `quick-sort` itself. When such a thing happens, computer scientists say that the function is recursive.

It is possible to understand how a recursive function works by following its execution step by step. In fact, this has been done for the `avg` function, in section 1.9, page 14. However, a much better approach is to study the function logically.

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

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

The `(filter (curry string>? (car s)) (cdr s))` expression will filter all strings that comes before `"g"` and return the `("a" "c" "b")`. A recursive call to `quick-sort` will sort this list, producing the first argument of `append`, to wit, `("a" "b" "c")`.

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

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

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

Chapter 5

Recursion

The mathematician Peano invented a very interesting axiomatic theory for natural numbers. I cannot remember the details, but the idea was something live 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 Myamoto Musashi, the famous Japanese Zen assassin, how he managed to kill a twelve year old boy protected by his mother's 37 samurais[1], he answered:

> *I defeated one of them, then I defeated the remaining 36. To defeat 36, I defeated one of them, then I defeated the remaining 35. To defeat 35, I defeated one of them, then 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 Append of Listing 5.1 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

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

`(cons (car x) (App (cdr xs) s))`. Let us pick a more concrete instance of the problem.

Nia evaluated `(App '(1 2 3) '(4 5))`. This calling pattern was matched to `(App xs s)` with `xs= '(1 2 3)` and `s= '(4 5)`. Since `(null? xs)` is false, the second clause rewrite the calling pattern as:

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

which the inference engine reduces to

```
(cons 1 (App '(2 3) '(4 5)))
```
(5.1)

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 5.1 becomes:

```
(cons 1 (cons 2 (App '(3) '(4 5))))
```
(5.2)

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 5.2 becomes:

```
(cons 1 (cons 2 (cons 3 (App '() '(4 5)))))
```
(5.3)

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 5.3 can be rewritten as:

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

```
(define (App xs s)
  (cond ( (null? xs) s)
        (else (cons (car xs) (App (cdr xs) s)) )))
```

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

Listing 5.1: Append

5.1 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)`:

`( (null? xs) s)`	The first clause is certainly trivial

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

5.2 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 5.2 shows a complete implementation of the quicksort algorithm for a list of numbers.

```
;; File: quicksort.scm


;; Output: elements of xs that are smaller than p

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


;; Output: elements of xs greater or equal to 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))) )) ))
```

```
> (quick '(4 3 1 5 2 8 7))
(1 2 3 4 5 7 8)
```

Listing 5.2: The quicksort algorithm

5.3 Named-let

Even before learning recursivity, Nia studied the Fibonacci function on page 32. The Fibonacci function 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 permit the creation of functions with initialized arguments, avoiding the definition of anomalous functions 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)) ))
```

```
> (load "Fibonacci.scm")
#<function>
> (fibo 5)
8
```

Listing 5.3: 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 5.3 is used to calculate the `fn+1` iteration, Nia put the `fn+1` tag on it.

There is another way to get rid of auxiliary parameters: One can initialize the parameters of a definition with default values, as shown in figure 5.4.

```
(define (avg s (sum 0) (n 0))
  (cond ( (and (null? s) (= n 0)) 0)
        ( (null? s) (/ sum n) )
        (else (avg (cdr s) (+ (car s) sum) (+ n 1.0)) )))
```

```
> (load "defaults.scm")
#<function>
> (avg '(3 4 5 6))
4.5
```

Listing 5.4: Accumulator parameters

5.4 Reading data from files

Figure 5.5 reads the contents of a file line by line and insert the corresponding strings into the buffer. However, before sending a string to the buffer, `(port->lines p)` prefixes it with a commented line number.

```
;; File: insFile.scm
;; The procedure (rdLines <filename>) reads
;; a Lisp file, and inserts its contents
;; into the current buffer.

(define (port->lines p)
  (let next-line ( (i 1) (x (read-line p)) )
    (cond ( (eof-object? x) #t)
          (else
            (insert "#") (insert "| ")
            (insert (number->string i))
            (cond ( (< i 10)  (insert "    |# "))
                  ( (< i 100) (insert "   |# "))
                  (else       (insert "  |# ")))
            (insert x) (insert "\n")
            (next-line (+ i 1) (read-line p))) )))

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

```
> (load "insFile.scm")
#<function>
> (rdLines "test.scm")
```

Listing 5.5: 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 5.5, 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 inserted in the buffer, and `loop` proceeds to read the next line. The iteration stops when end of file is reached.

5.5 Writing data to files

Let us assume tha Nia needs a femtolisp 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 5.6 reads a markdown file line by line. If the line starts with the `##` prefix, it will be wrapped in the `h1>...`/h1>| html tag. If its first char is the `#\#` prefix, but the second char is different from `#\#`, it will be wrapped in the `<h2>...</h2>` html tag.

The simplified version of the html generator of listing wrtFile 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 52.

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 37 and 44. File input/output must be mastered to perfection.

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

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

```
> (load "toHtml.scm")
#<function>
> (copyFile "toHelen.txt" "helen.html")
#t
```

Listing 5.6: Writing data to a file

Chapter 6

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 6.1.

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

Listing 6.1: Factorial function

The domain of the `factorial` function is $\mathbb{Z}$. In femtolisp, 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`, femtolisp will produce garbage. For instance, `(fac 20)` produces the correct result: `2432902008176640000`. However `(fac 21)` returns a meaningless sequence of digits. An interesting question is: How to know whether `(fac 20)` produces a correct result? Well, you 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 (i 1) (f 1) (next-f (* i f)))
   (assert (= (quotient next-f f) i))
   (if (>= i n) next-f
     (fact n (+ i 1) next-f) ))
```

```
> (load "factorial.scm")
#<function>
> (fact 15)
1307674368000
> (fact 21)
(assert-failed (= (quotient next-f f) i))
```

Listing 6.2: Factorial function

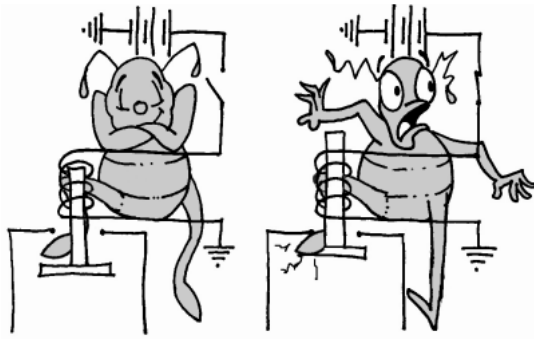
In order to avoid specification errors, it is necessary to use logical statements to write a contract between the computer engineer and the client. The contract has three assertions:

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 `(assert (= (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.

6.1 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 embarassment of failure being manifest on the client's terminal. Femtolisp does not possess such a kind of oracular compiler. It report errors only when you execute the code.

6.2 Missing parenthesis

There is only one kind of bug that femtolisp 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 6.3. The compiler issues the following error message:

> (load-error fvalue.scm (parse-error read: unexpected end of input))

The error is on line 4, where a right parenthesis is missing. Nevertheless, the compiler does not give the smallest hint on where the problem is. Furthermore, the error message is misleading, as it does not mention the parenthesis. It simply complains about an unexpected end of input.

Whenever you receive a message of 'unexpected end of input', my advice is to look for a missing parenthesis. However, checking parentheses by hand

over a large file is time consuming and boring. Is it possible to write a new loader with error messages that are more helpful? The answer to this question is yes! In fact, it is very easy to customize Lisp.

```
;; 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))) )))

#|This loader fails at definition 3|#
```

```
(load "manydefs.scm")Enter
(load-error manydefs.scm
   (parse-error read: unexpected end of input))
```

Listing 6.3: Future Value Program

Listing 6.4 shows a loader that informs which expression has an unmatched parenthesis. Besides this, if you add postcondition expressions to the source file, the program will also report specification errors. For example, in listing 6.5, if the postcondition `(assert (= (fn+1 5) 8))` fails, the

`load-definition` command will inform which assertion did not meet the specifications. Here one has an example of a specification error. For the definition $f = \lambda(n)(n < 2)?1 : f(n-1) + f(n-2)$, one has that $f(5) = 5$. On the other hand, $f = \lambda(n)(n < 2)?1 : f(n-1) + f(n-2)$ produces $f(5) = 8$. Both alternatives are correct, but the postcondition shows that the client wants the second. Now that you know the effects, let us see how to perform the magic.

```
;; File: loader.scm

(define (rd i out)
  (trycatch (read out)
      (lambda(x) (insert "#") (insert "|")
         (insert "This loader failed at definition ")
         (insert (number->string i)) (insert "|#\n")) ))

(define (try-to-eval i sexpr)
   (trycatch (eval sexpr)
       (lambda(x) (insert "#") (insert "|")
          (insert "Error in expression ")
          (insert (number->string i))
          (insert "|#\n")) ))

(define (eval-them out)
   (let loop ( (i 1) (sexpr (rd 1 out)))
       (cond ( (eof-object? sexpr) #t)
             ( else (try-to-eval i sexpr)
                    (loop (+ i 1) (rd (+ i 1) out))) )))

(define (load-definitions source)
   (call-with-input-file source eval-them))
```

```
(load "loader.scm")Enter
#<function>
```

Listing 6.4: Reporting error position

In listing 6.4, a `(trycatch sexpr (lambda(x) ...))` tries to evaluate a Lisp `sexpr`. If the evaluation ends with success, `trycatch` returns the value of the `sexpr`. Otherwise, `trycatch` executes its second argument, which reports the error.

In listing 6.4, the `trycatch` form is used on two occasions. In the `(rd i out)` definition, `trycatch` reports unmatched parentheses.

In `(try-to-eval i sexpr)`, `trycatch` will report runtime errors. Since femtolisp defers error reporting to runtime, it is advisable to add `(assert P)` postconditions to the code, in order to detect specification errors.

The `(eval sexpr)` function-call evaluates a Lisp expression. Therefore, if one executes the `(eval '(* 2 21))`, s/he will get 42.

```
;; 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 1) (fn-1 1))
  (if (< n 2) fn
     (fn+1 (- n 1)
           (+ fn fn-1) fn ) ))

(assert (= (fn+1 5) 8))

(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))) )))
```

```
(load-definitions "manydefs.scm")Enter
#<function>
```

Listing 6.5: Checking for specification errors

Chapter 7

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 7.1 below illustrates a deterministic finite automaton using a state diagram. The aforementioned automaton recognizes words and puctuation 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.

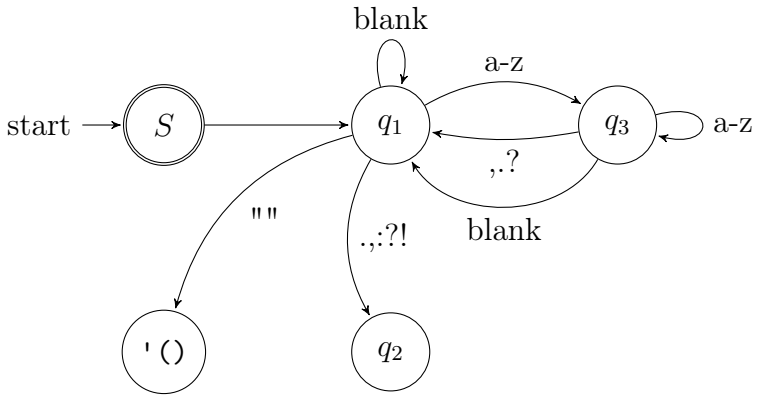


Fig. 7.1: Finite automaton that recognizes English words

In the automaton of figure 7.1, there are three states: q_1, q_2 and q_3. The automaton remains in the state q_1 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 returns `'()` and stops. If a punctuation mark appears in the stream, the automaton accepts it, and goes to the q_2 state.

```
;; Finite automaton

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

(define (punctuation-mark? c)
  (not (not (string.find ".;:,?!()" c)) ))

(define (tokenize s)
   (ignore-blank 0 s))

(define (ignore-blank n s)
   (cond ( (end-of-string? s n) '())
         ( (char-whitespace? (string-ref s n))
           (ignore-blank (+ n 1) s))
         ( (punctuation-mark? (string-ref s n))
             (cons (substring s n (+ n 1))
                   (ignore-blank (+ n 1) s)))
         (else (word n (+ n 1) s)) ))

(define (word i j s)
   (cond ( (end-of-string? s j)
           (list (substring s i j)))
         ( (char-whitespace? (string-ref s j))
             (cons (substring s i j)
                   (ignore-blank j s)))
         ( (punctuation-mark? (string-ref s j))
           (cons (substring s i j)
                 (ignore-blank j  s)))
         (else (word i (+ j 1) s)) ))
```

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. Available in `http://www.htdp.org`.

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

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

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

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

Index