An introduction to femtolisp

Vernon Sipple

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, that is a text-based user interface. Therefore, let us learn how to install ncurses, in case your machine still does not have it. You must download the most recent version from the distribution site:

```
https://www.gnu.org/software/ncurses/
```

Unpack the distribution archive as shown below:

```
~$ tar xfvz ncurses-6.0.tar.gz
```

The next step consists of entering the distribution directory, and then perform the usual `./configure`, `make`, `make install` cycle.

Diacritics. Some people add marks or glyphs to certain letters. For instance, they write “café” in lieu of “cafe”. If you need or like diacritics, you must compile the libncurses library in this manner:

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

MAC OS-X. Femtoemacs assumes that you have placed `libncursesw.a` in the following directory:

```
    /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. Do not use apt-get to install ncurses, as the repositories are old. Instead, download a recent source distribution and build it thus:

```
~$ 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 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-k` – kill a line.
  `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-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.

- `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.

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.

2 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 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.

```
; 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")
```

Figure 1: Source file and Interation Buffer

Let us recall what happened until now. Nia pressed `C-x 2` to split the buffer window, as shown in figure 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)` 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 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)
```

Figure 2: 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, 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 2, a list element can be a nested sublist. In this case, the `car` part of a pair can point down to a sublist branch. The diagram below shows that one can reach any part of a nested list following a chain of `car` and `cdr`. In such a chain, the `cdr` operation advances one pair along the list, and the `car` operation goes down into a sublist. The `cdr` operation is equivalent to a right → arrow, while the `car` operation is acts like a down ↓ arrow.

```
      →        →
[*|*]cdr-[*|*]cdr---[*|*]cdr-()
car      car        car↓
 |        |          |   →
 |        |         [*|*]cdr-[*|*]cdr-[*|*]cdr-()
 |        |         car      car↓     car
 |        |          |        |        |
 |        |          |        |        B
 |        |          |        |
 |        |         AND      [*|*]cdr-[*|*]cdr-()
 |        |                  car      car
 |        |                   |        |
 |        |                  NOT       A
 |        |
 OR      [*|*]cdr-[*|*]cdr-[*|*]cdr-()
          |        |        |
         AND       A       [*|*]cdr>[*|*]cdr>()
                            |        |
                           NOT       B
```

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 `(cdr (cdr (car (cdr (car xs)) )))` to reach the goal.

4 let binding

The let-form introduces local variables through a list of `(id value)` pairs.

```
(define (zr ymd)
  (let* ( (y (car ymd))
          (m (cadr ymd))
          (d (caddr ymd))
          (ax (quotient (- 14 m) 12))
          (mm (+ m (* 12 ax) -2))
          (yy (- y ax))
          (c (- (quotient yy 100)))
          (ly (quotient yy 4)) )
    (wday (mod (+ yy
             (quotient (- (* 13 mm) 1) 5)
             ly
             c
             (quotient yy 400) d) 7))))
```

In the basic let binding, a variable cannot depend on previous variables that appear in the local list. In the `let*` (let star) binding, one can use previous variables to calculate the value of a variable, as one can see in the above definition.

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.

A common error when dealing with the let binding is to forget the parentheses that group local variables together. Besides this, learners often leave out the parentheses that associate each variable with its value. Therefore, one is advised to study the syntax of the let binding carefully, to avoid these two types of error.

The above program calculates the day of the week through Zeller's congruence. The `ymd` variable points to a list containing the year, the month and the day. Therefore, the pair `(y (car ymd))` establishes that the variable `y` is initialized with the result of the first element of the `ymd` list, which is the year. On the same token, `(m (cadr ymd))` initializes the `m` (month) variable, and `(d (caddr ymd))` identifies the `d` with the day of the month. The `quotient` function performs the integer divisions required by the algorithm.

As an example, `(zr '(31 8 2016))` returns a number between 0 and 6, corresponding to the days `"Sunday"`, `"Monday"`, `"Tuesday"`, `"Wednesday"`, `"Thursday"`, `"Friday"` and `"Saturday"`.

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 (wday n)
   (cond ( (equal? n 0) "Sunday")
         ( (equal? n 1) "Monday")
         ( (equal? n 2) "Tuesday")
         ( (equal? n 3) "Wednesday")
         ( (equal? n 4) "Thursday")
         ( (equal? n 5) "Friday")
         ( (equal? n 6) "Saturday")
         ( else "unknown")))
```

The cond-form is used to control 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 `(equal? n 0)` predicate. If the predicate `(equal? n 0)` returns `#t` (true) for Sunday index, the function `wday` returns `"Sunday"`. If `n= 1`, the second condition holds, and the function produces the string `"Monday"`. And so on.

The idea is to select a string such as `"(2016 8 31)"` and save it on the clipboard with the `M-w` command. 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 `zr` function needs a list. Therefore, Nia needs a procedure to read a list from a string of characters.

```
(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, except 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 for 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 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:

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

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

(define (wday n)
   (cond ( (equal? n 0) "Sunday")
         ( (equal? n 1) "Monday")
         ( (equal? n 2) "Tuesday")
         ( (equal? n 3) "Wednesday")
         ( (equal? n 4) "Thursday")
         ( (equal? n 5) "Friday")
         ( (equal? n 6) "Saturday")
         (else "unknown")))

(define (zr ymd)
   (let* ( (y (car ymd))
           (m (cadr ymd))
           (d (caddr ymd))
           (ax (quotient (- 14 m) 12))
           (mm (+ m (* 12 ax) -2))
           (yy (- y ax))
           (century (- (quotient yy 100)))
           (ly (quotient yy 4)) )
       (wday (mod (+ yy
                     (quotient (- (* 13 mm) 1) 5)
                     ly
                     century
                     (quotient yy 400) d) 7)) ))
```

```
;; Shortcut definitions
(define (keyboard key)
   (cond
      ( (equal? key "C-c z")
        (insert (zr (read-string (cutregion)) ))
        (insert " is the day of the week."))
      ( (equal? key "C-c a")
        (insert "<p> </p>")
        (backward 5))
      ( (equal? key "C-c b")
        (beginning-of-line)
        (insert "<h1> </h2>")
        (beginning-of-line)
        (forward 4))
      ( (equal? key "C-c c")
        (end-of-line)
        (insert "<p>-")
        (insert (clipboard))
        (insert "-</p>"))
      (else (insert key)) ))

(newlanguage ".scm" ";" "#|" "|#")
(keyword "define")
(keyword "cond")
(keyword "else")

(newlanguage ".lsp" ";" "#|" "|#")
(keyword "define")
(keyword "cond")
(keyword "else")

(newlanguage ".c" "//" "/*" "*/")
(keyword "auto")
(keyword "break")
(keyword "case")
(keyword "char")
(keyword "const")
(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")
```

6 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.

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*.

References

[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.