Programming with Wisp, Quick start from zero to best practices

Wisp is an extension to Guile Scheme that implements Scheme request for implementation 119. It uses fewer parentheses than typical Scheme-code but has the same expressive power. Here I want to show you how to use it and then build a more complex application to show it in real use.

This is unfinished and got replaced by the conceptually leaner Naming & Logic: programming essentials with Wisp

Contents

Install

If you're on Guix, just use `guix install guile-wisp`.

Otherwise install Guile (`http://gnu.org/s/guile`) and Mercurial (`http://www.mercurial-scm.org`), then get and install wisp:

```
hg clone https://hg.sr.ht/~arnebab/wisp
cd wisp && autoreconf -i && ./configure && make && sudo make install
```

More detailed instructions are available on the Wisp-Website.

Commands in a Wisp-shell

To start a wisp-shell, just run `wisp`

```
wisp
```

Now you can type wisp-commands. To run the commands you typed, add two empty lines.

```
display "Hello World!\n"


```

```
Hello World!
```

Use `;;` to comment a line. I will use this to show results of commands:

```
display "Hello World!\n"
;; => Hello World!
```

Call existing functions: the overpowered pocket calculator

Just name the function you use (here "fun") and add the arguments (here "arg1" and "arg2"), separated by spaces:

```
fun arg1 arg2
```

Hit enter thrice to get the result. Hit tab to get a list of functions you can call:

```
wisp@(guile-user)> {TAB}
Display all 1958 possibilities? (y or n) y
* + - ... / < <= = => > >= @ @@ _ $sc-dispatch 1+ 1- ...
```

There's an awful lot of math in these first of the functions, so let's put them to use in an example.

Overpowered Pocket Calculator (OPC)

All those math operators are functions. You can call them as commands to get an overpowered pocket calculator (OPC™):

```
+ 1 2
;; => 3
```

If you have a lot of arguments, you can put them on the next line by starting the line with `. ` (period and space):

```
+ 1 2 3
  . 4 5
```

```
;; => 15
```

You can use a sub-command as argument by putting it on its own line with indentation, without period at the start:

```
+ 1 2
  * 3 4
  . 5
;; => 20
```

Instead of putting a command as argument on its own line, you can also enclose it by parentheses `()`:

```
+ 1 2 (* 3 4) 5
;; => 20
```

If you prefer infix-math, you can use braces `{}` instead of parentheses:

```
+ 1 2 {3 * 4} 5
;; => 20
```

You can also get the equivalent of parentheses closed at the end of the line by using a colon sourrounded by whitespace (`:`). Note that later arguments must be put on the next line:

```
+ 1 2 : * 3 4
  . 5
;; => 20
```

If you're following this guide in a wisp-shell (as you should!) and you're growing tired of typing enter three times, just end a line with ` .` (space period) and hit enter once to execute what you already typed:

```
* 1 2 3
  . 4 5 .
;; => 120
```

An incomplete list of math-functions:

```
+ - * / abs max min modulo modulo-expt
sqrt exp expt sin cos tan asin acos atan log log10
floor ceiling round floor/ ceiling/ round/ ash
```

To see their meaning, just use the `help` command (note the `.` at the end):

```
help modulo-expt .
;; `modulo-expt' is a procedure in the (guile) module.
;;
;; - Scheme Procedure: modulo-expt n k m
;;      Return N raised to the integer exponent K, modulo M.
;;
;;           (modulo-expt 2 3 5)
;;              ==> 3
;;
```

You can find more math functions and the arguments they take in the Guile manual under Arithmetic and Scientific (also available offline from the Guile manual via the Info reader and Emacs with info-mode: `C-h i`).

There are many more commands you can use right from the shell without doing anything else. Have a look at your wisp shell to see them.

But there is more: you can import modules to get specialized procedures.

Import from modules

Once you need more than the directly available functions (called “procedures” in the manual), you’ll need to import them.

Let’s say that you want to know the current date formatted as ISO-8601 date. For this you can import SRFI-19 to get date and time functions.

```
;; import all date and time functions
import : srfi srfi-19 ;; date and time functions as SRFI
;; use the ones you need.
date->string : current-date
             . "~1"
;; => "2021-03-14"
```

Let’s look at the more complex case where you want to know the date tomorrow formatted as ISO-8601 date.

This still looks complicated. We will make this easier when we get to defining variables for re-use.

Read the following example starting from the deepest indentation (marked with `;; HERE is the deepest indentation`):

```
;; import all date and time functions
import : srfi srfi-19 ;; date and time functions as SRFI
```

```
;; use the ones you need.
date->string ;; get the date as string with a given format
  time-utc->date ;; convert time back to a date
    add-duration ;; add to the time
      date->time-utc ;; convert date to time in UTC
        ;; get the current date:
        current-date ;; HERE is the deepest indentation
      make-time time-duration ;; define a time-duration to add
         . 0 ;; nanoseconds
         * 24 60 60 ;; 1 day: 24h in seconds
  . "~1" ;; format for the date as string; ~1 means ISO-8601 date
;; => "2021-03-15"
```

You can find a complete list of procedures in the Procedure-Index.

SRFIs are “Scheme Requests for Implementation”, community-created and curated standards, spoken as “surfies”, available in many implementations of Scheme. To get accomodated with what they offer, have a look at the list of SRFIs at srfi.schemers.org.

. . . this concludes the first step. If you don’t want to wait until I get to write and add the others, have a look at the **wisp-tutorial in With Guise and Guile**.

TODO Conditionals: decision branching

To act differently depending on some criterion, use `cond`:

```
cond
  : equal? 1 2
    format #t "Math is broken!\n"
  else
    format #t "The universe might continue to exist.\n"
;; => The universe might continue to exist.
;;    (I hope - please tell me if you get Something different.
;;     Then you might be on to an interesting anomaly :-))
```

TODO Binding variables

TODO Defining functions

TODO Save Code in Files

TODO