Introduction to Scheme/Guile

unwrap what's in package

Simon Tournier

INSERM US 53 - CNRS UAR 2030
simon.tournier@u-paris.fr

November 29, 2022
https://hpc.guix.info

We will discuss about

```
$ guix edit gsl
```

```
(define-public gsl
  (package
...
    (arguments
     (let ((...))
       `(#:configure-flags (list "--disable-static")
         #:phases
         (modify-phases %standard-phases
           ,@(cond
...
                 '((add-after 'unpack 'force-bootstrap
                     (lambda _
...
                (or (string-prefix? "aarch64" system)
...
                      (substitute* "spmatrix/test.c"
...
                (else '()))))))
```

```
$ guix edit gsl
```

What does it mean?

keyword	`define-public`, `let`, `lambda`
record	`package`
convention	`%something`, `something?`, `something*`
symbol	quote (’), backquote (‘), comma (,), comma at (,@), underscore (_)

Resources (links)

Manual Guile « Hello Scheme »

Mini-tuto « A Scheme Primer »

Mini-tuto « Unlock Lisp / Scheme's magic: beginner to Scheme-in-Scheme in one hour » (video 1h)

Talk « A tour of the Guix source tree » (video 40min)

Book « How to Design Programs » (famous course about Racket, another Scheme; syntactic sugar varies)

Book « Structure and Interpretation of Computer Programs » (famous course from MIT)

Post Whitespace to Lisp (wisp) « Going from Python to Guile Scheme »

Post « How to implement (basic) automatic differentiation using Guile »

First things first

S-expression: atom or expression of the form `(x y ...)`

atom: `+`, `*`, `list`, etc.

expression: `(list 'one 2 "three")`

Call (evaluation) with parenthesis

e.g., apply the atom `list` to the rest

`(list 'one 2 "three")` returns the list composed by the elements `(one 2 "three")`

Quote protects from the call

e.g., `'one` returns plain one

`'(list 'one 2 "three")` returns `(list 'one 2 "three")`

Second things first

variable `(define some-variable 42)`

procedure `(lambda (argument) (something argument))`

Define a procedure

```
(define my-name-procedure
  (lambda (argument1 argument2)
    (something-with argument1)))
```

```
(define (my-name-procedure argument1 argument2)
  (something-with argument1))
```

Call `(my-name-procedure 1 "two")`

`define-public` is sugar to `define` and export (see « Creating Guile Modules (link)) »

Local variables = `let`

Independent local variables

```
(define (add-plus-2 x y)
  (let ((two 2)
        (x+y (+ x y)))
    (+ x+y two)))
```

Dependant local variables

```
(define (add-plus-2-bis x y)
  (let* ((two 2)
         (x+two (+ x two))
         (result (+ y x+two)))
    result))
```

Conventions

predicate ends with question mark (?), return boolean (`#t` or `#f`)

e.g., `(string-prefix? "hello" "hello-world")`

variant ends with star mark (*)

e.g., `let*`

Optional argument `define*` (see manual here)

```
(define* (add-plus-something x y #:optional (value 2))
  (+ x y value))
```

keyword starts with sharp colon (`#:`) (see manual here)

e.g., `#:optional`, `#:configure-flags`, `#:phases`

"global" starts with percent (%)

e.g., `%standard-phases`

Quote, quasiquote, unquote

quote	do not evaluate (keep as it is)	quote ’
quasiquote	postpone evaluation	backquote ‘
unquote	evaluates at compile time	coma ,

```
(define (if-then-else predicate then else)
  (if predicate
      then
      else))
```

```
scheme@(guile-user)> (if-then-else #f
                                   (1+ "failure")
                                   'else)
BOUM!
```

The both arguments are evaluated.

We would like to post-pone...

Macro

```
(define-macro (if-then-else predicate then else)
  `(if ,predicate
       ,then
       ,else))
```

```
scheme@(guile-user)> (if-then-else #f
                                   (1+ "failure")
                                   'else)
$28 = else
```

The `then` branch is never evaluated.

- S-expression syntax and macro allow to easily create Domain-Specific Language (DSL)
- The concept is behind the concept of G-exp (see manual (link))
  where #~ or #$ are similar as quasiquote and unquote for "code".

Record

How to create new data types?

C programmer would think about `struct`.

```
(define-record-type <pkg>                ;name, convention <>
  (pkg name version)                     ;constructor
  pkg?                                   ;predicate
  (name pkg-name)                        ;accessor
  (version pkg-version))
```

```
scheme@(guile-user)> (pkg "hello" "1.2")
$29 = #<<pkg> name: "hello" version: "1.2">
```

Questions ?

guix-science@gnu.org

https://hpc.guix.info/events/2022/café-guix/