How to make advanced packages

Simon Tournier

November, 10th, 2023

Abstract

This tutorial is dedicated to review what can be done when that's not enough to list dependencies and/or declare a build system. The aim is to introduce various mechanisms for adapting the base Guix recipe. The prerequisite is the reading of the section "Defining Packages" from the manual and the goal of this tutorial is to provide some ingredients for making it sound. We propose to first introduce a Scheme/Guile Swiss-knife toolbox, then to cover how to modify upstream source code (field `origin`) and how to customize the build system parameters or phases (field `arguments`). If time allows, we will introduce the meaning of cryptic symbols as the sequence `#~(#$(`

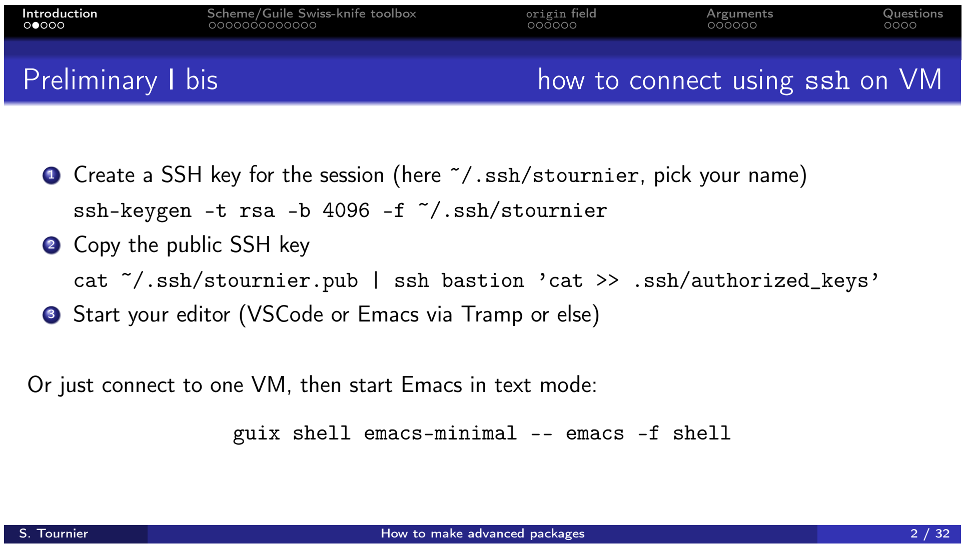
Do not forget that packaging is a craft, so there is **no magic but only practise.**

(Terms using sans serif font are hyperlinks.)

This tutorial for introducing key components to make advanced packages for Guix is its first version. Do not take all as written in stone; it is based on my experience and I do not consider being a packaging expert. If I might, my two only advises are:

1. Dive into existing packages and confront with Guix manual.
2. Most is about **a lot of** practise. Quoting rekado,

I wish I had anything to say about this other than
"try again, give up, forget about it, remember it, ask for pointers, repeat"
`#guix-hpc` on 2023-10-13.

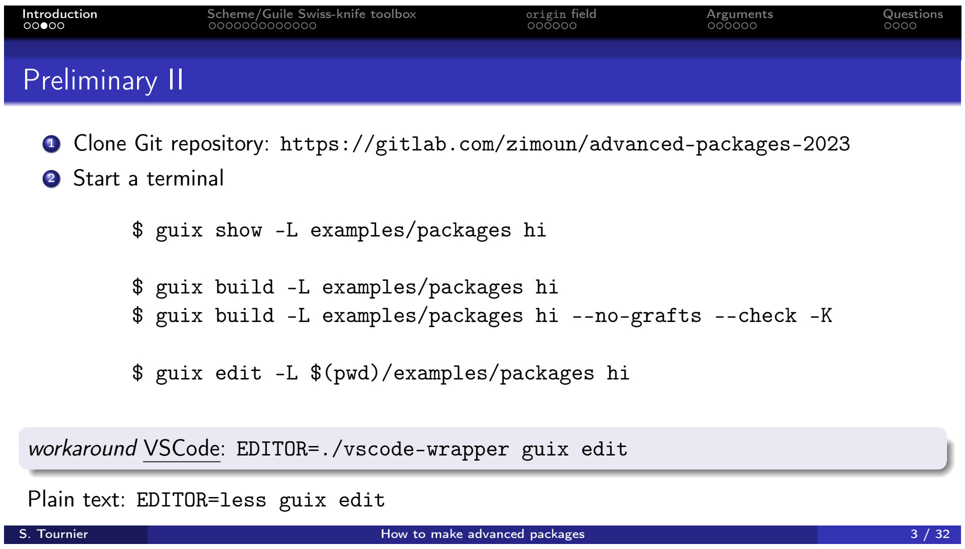


Preliminary I bis — how to connect using ssh on VM

1. Create a SSH key for the session (here `~/.ssh/stournier`, pick your name)
   `ssh-keygen -t rsa -b 4096 -f ~/.ssh/stournier`
2. Copy the public SSH key
   `cat ~/.ssh/stournier.pub | ssh bastion 'cat >> .ssh/authorized_keys'`
3. Start your editor (VSCode or Emacs via Tramp or else)

Or just connect to one VM, then start Emacs in text mode:

```
guix shell emacs-minimal -- emacs -f shell
```



Preliminary II

1. Clone Git repository: `https://gitlab.com/zimoun/advanced-packages-2023`
2. Start a terminal

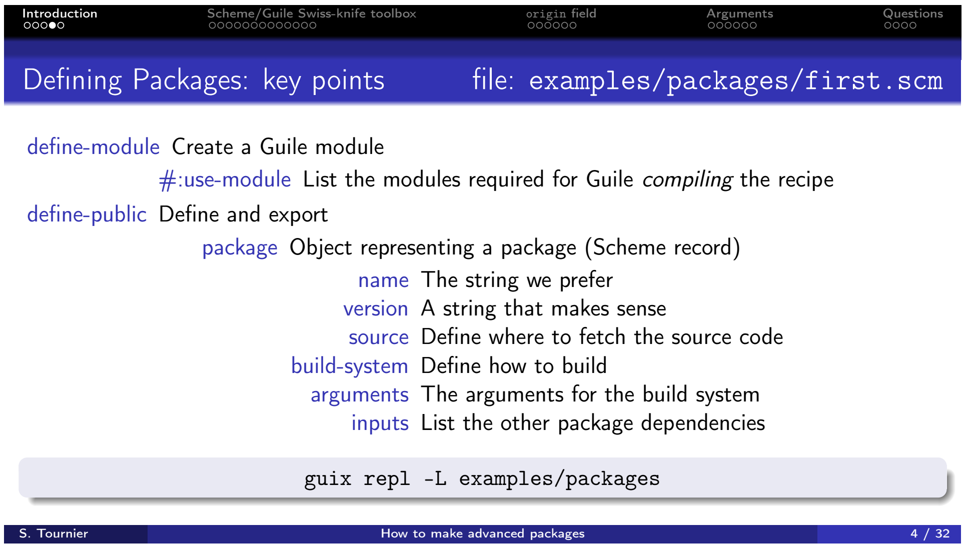
```
$ guix show -L examples/packages hi

$ guix build -L examples/packages hi
$ guix build -L examples/packages hi --no-grafts --check -K

$ guix edit -L $(pwd)/examples/packages hi
```

workaround VSCode: `EDITOR=./vscode-wrapper guix edit`

Plain text: `EDITOR=less guix edit`

1 Introduction

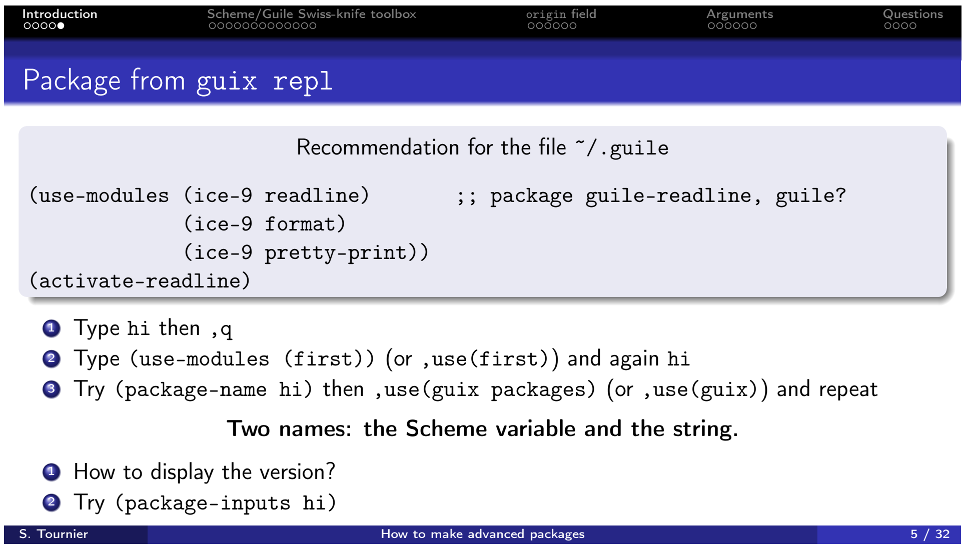
As a warmer, let re-read the section Defining Packages from the Guix manual. The aim of this ≈ 1*h*30 tutorial is to help in packaging when Invoking `guix import` is not enough.
Quoting Package Naming:
A package actually has two "names" associated with it. First, there is the name of the Scheme variable, the one following 'define-public'. By this name, the package can be made known in the Scheme code, for instance as input to another package. Second, there is the string in the 'name' field of a package



Defining Packages: key points file: examples/packages/first.scm

define-module	Create a Guile module
#:use-module	List the modules required for Guile *compiling* the recipe
define-public	Define and export
package	Object representing a package (Scheme record)
name	The string we prefer
version	A string that makes sense
source	Define where to fetch the source code
build-system	Define how to build
arguments	The arguments for the build system
inputs	List the other package dependencies

```
guix repl -L examples/packages
```



Package from `guix repl`

Recommendation for the file `~/.guile`

```
(use-modules (ice-9 readline)       ;; package guile-readline, guile?
             (ice-9 format)
             (ice-9 pretty-print))
(activate-readline)
```

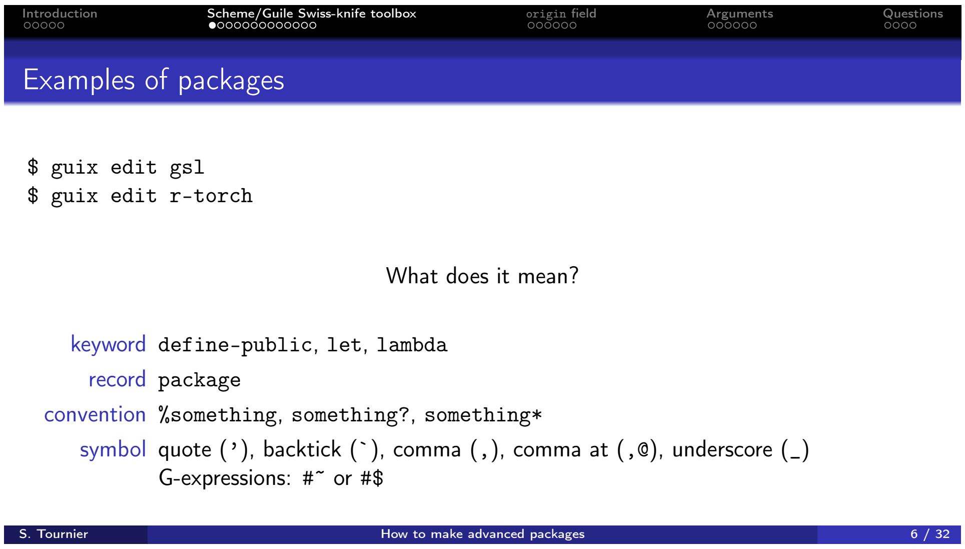
1. Type `hi` then `,q`
2. Type `(use-modules (first))` (or `,use(first)`) and again `hi`
3. Try `(package-name hi)` then `,use(guix packages)` (or `,use(guix)`) and repeat

Two names: the Scheme variable and the string.

1. How to display the version?
2. Try `(package-inputs hi)`

definition. This name is used by package management commands such as ‘guix package’ and ‘guix build’.
Pick the same for both is welcome but not mandatory. And several packages for the same version have the same string name but not the same symbol; except if they are defined in different modules.

2 Scheme/Guile Swiss-knife toolbox



Introduction | Scheme/Guile Swiss-knife toolbox | origin field | Arguments | Questions

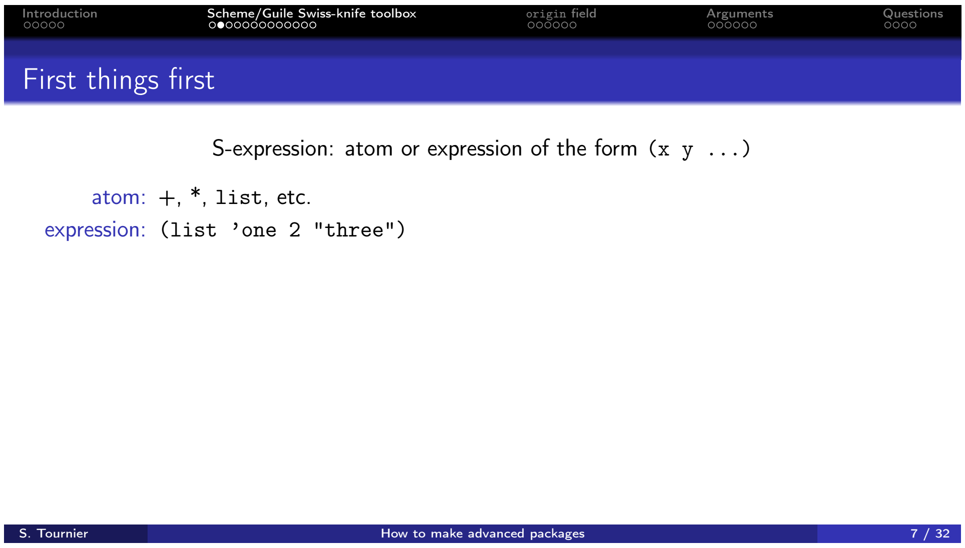
Examples of packages

```
$ guix edit gsl
$ guix edit r-torch
```

What does it mean?

keyword	`define-public`, `let`, `lambda`
record	`package`
convention	`%something`, `something?`, `something*`
symbol	quote (`'`), backtick (`` ` ``), comma (`,`), comma at (`,@`), underscore (`_`) G-expressions: `#~` or `#$`

S. Tournier — How to make advanced packages — 6 / 32



Introduction | Scheme/Guile Swiss-knife toolbox | origin field | Arguments | Questions

First things first

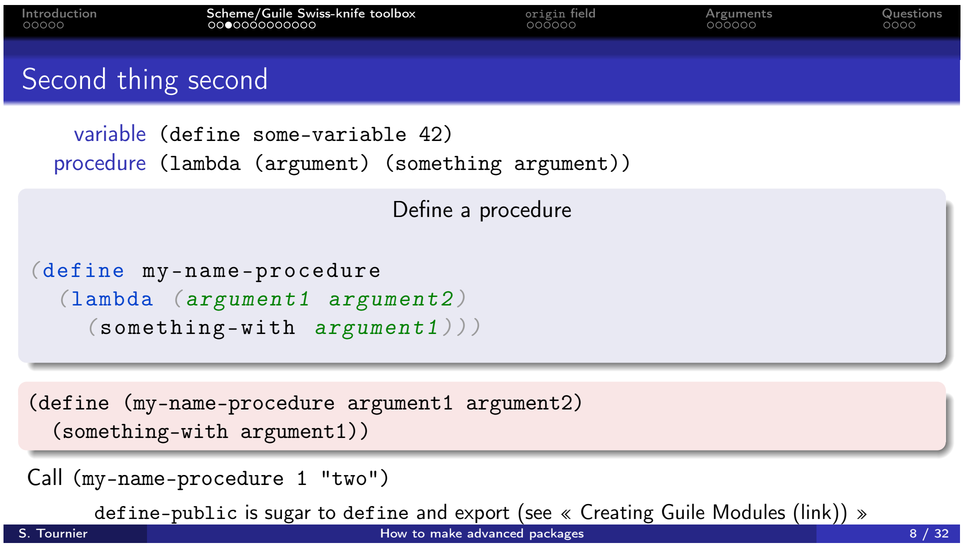
S-expression: atom or expression of the form `(x y ...)`

atom:	`+`, `*`, `list`, etc.
expression:	`(list 'one 2 "three")`

S. Tournier — How to make advanced packages — 7 / 32

"quoted" data remains unevaluated, and provides a convenient way of representing Scheme programs. This is one of the big payoffs of Lisp's simple syntax: since programs themselves are lists, it is extremely simple to represent Lisp programs as data. Compare the simplicity of quoted lists with the ML datatype that we used to represent ML expressions.
This makes it simple to write programs that manipulate other programs — it is easy to construct and transform programs on the fly.

Note that names in Lisp programs are translated into symbols in quoted Lisp expressions. This is so that quoted names can be distinguished from quoted strings; consider the difference between the following two expressions:


Introduction
Scheme/Guile Swiss-knife toolbox
origin field
Arguments
Questions

Second thing second

variable `(define some-variable 42)`
procedure `(lambda (argument) (something argument))`

Define a procedure

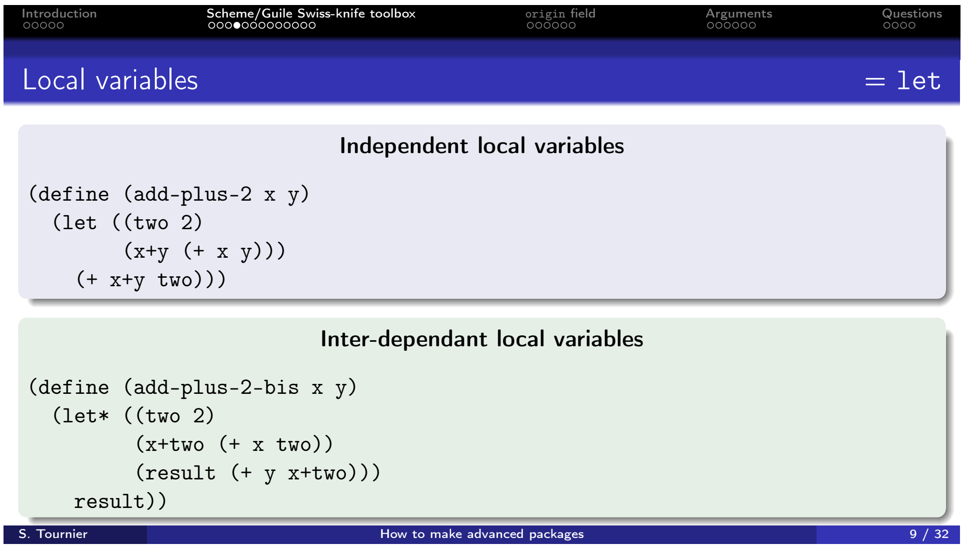
```
(define my-name-procedure
  (lambda (argument1 argument2)
    (something-with argument1)))
```

```
(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)) »

S. Tournier How to make advanced packages 8 / 32


Introduction
Scheme/Guile Swiss-knife toolbox
origin field
Arguments
Questions

Local variables = let

Independent local variables

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

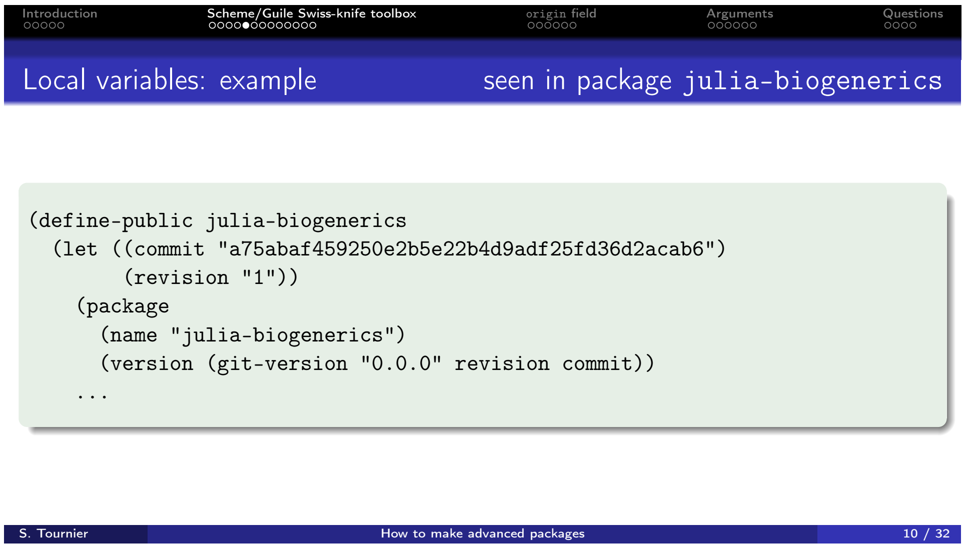
Inter-dependant local variables

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

S. Tournier How to make advanced packages 9 / 32

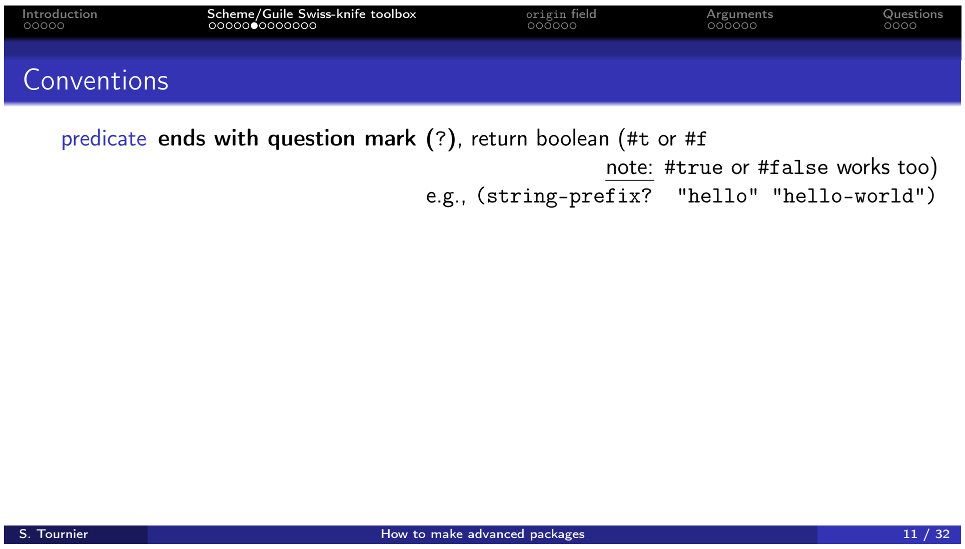
Let build something!

Before, we need to fetch something, isn't it?



Local variables: example seen in package julia-biogenerics

```
(define-public julia-biogenerics
  (let ((commit "a75abaf459250e2b5e22b4d9adf25fd36d2acab6")
        (revision "1"))
    (package
      (name "julia-biogenerics")
      (version (git-version "0.0.0" revision commit))
    ...
```



Conventions

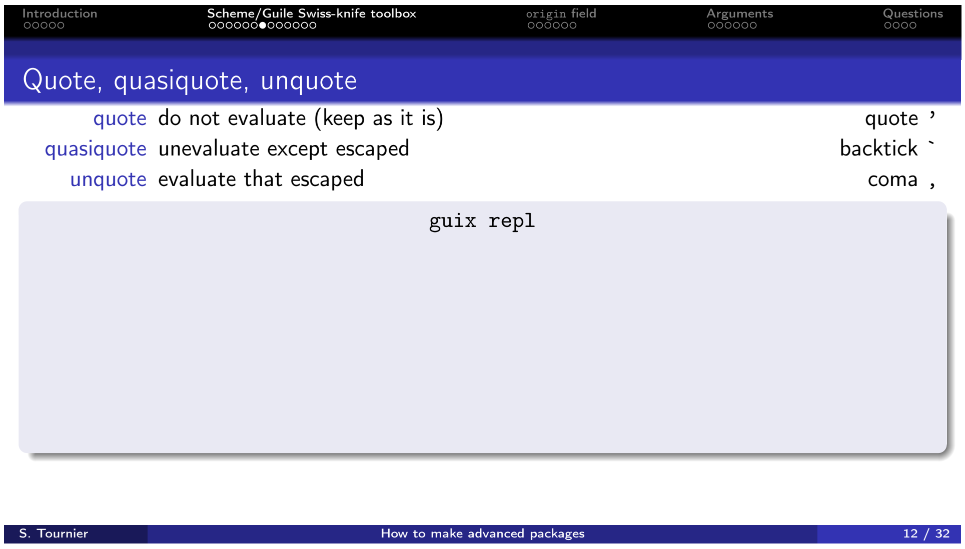
predicate **ends with question mark (?)**, return boolean (`#t` or `#f`
note: `#true` or `#false` works too)
e.g., `(string-prefix? "hello" "hello-world")`

3 `origin` field

4 Arguments

`%standard-phases`

Except one, all package build systems implement a notion of Build Phases: a sequence of actions that the build system executes, when you build the



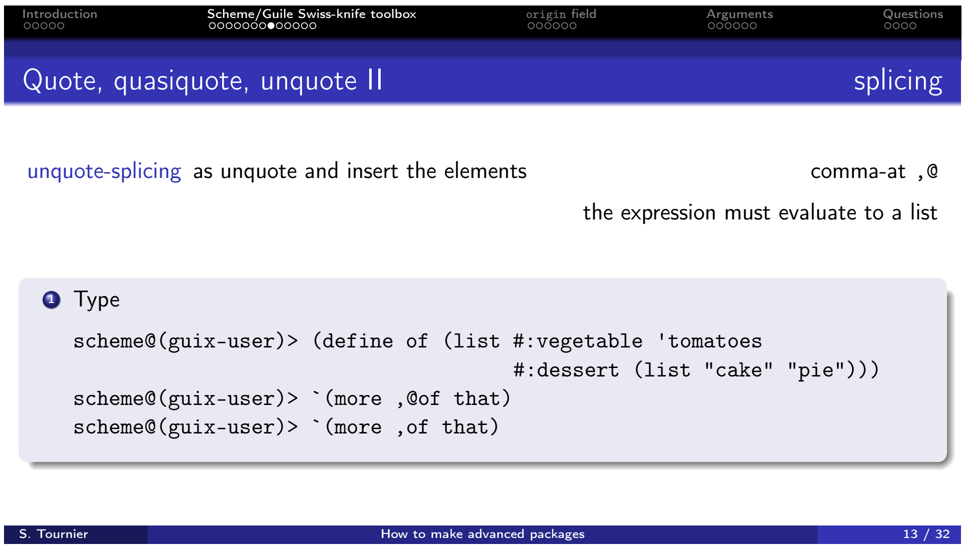
Introduction | **Scheme/Guile Swiss-knife toolbox** | origin field | Arguments | Questions

Quote, quasiquote, unquote

quote	do not evaluate (keep as it is)	quote '
quasiquote	unevaluate except escaped	backtick `
unquote	evaluate that escaped	coma ,

```
guix repl
```

S. Tournier | How to make advanced packages | 12 / 32



Introduction | **Scheme/Guile Swiss-knife toolbox** | origin field | Arguments | Questions

Quote, quasiquote, unquote II — splicing

unquote-splicing	as unquote and insert the elements	comma-at ,@

the expression must evaluate to a list

1. Type

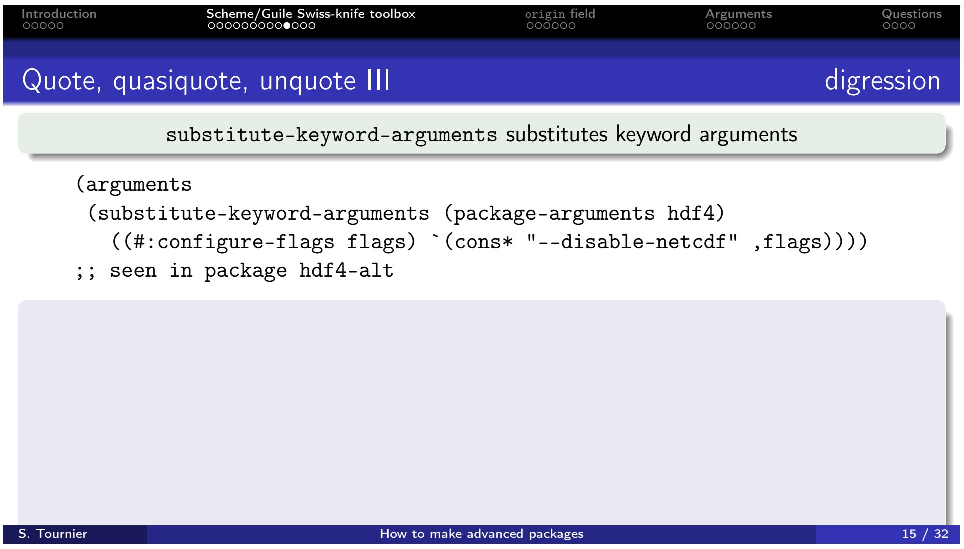
```
scheme@(guix-user)> (define of (list #:vegetable 'tomatoes
                                     #:dessert (list "cake" "pie")))
scheme@(guix-user)> `(more ,@of that)
scheme@(guix-user)> `(more ,of that)
```

S. Tournier | How to make advanced packages | 13 / 32

package. For instance, these actions might be `unpack`, `configure`, `build`, `check`, `install`, etc.

4.0.1 List and compare `%standard-phases`

For each build system, this sequence of actions is stored by `%standard-phases`. Please, note that this sequence of actions is build system depends, i.e., `%standard-phases` is defined per build system. For example, the `%standard-phases` for the GNU build system is defined by the module `(guix build-system gnu)`. And the `%standard-phases` for the Python build system is defined by

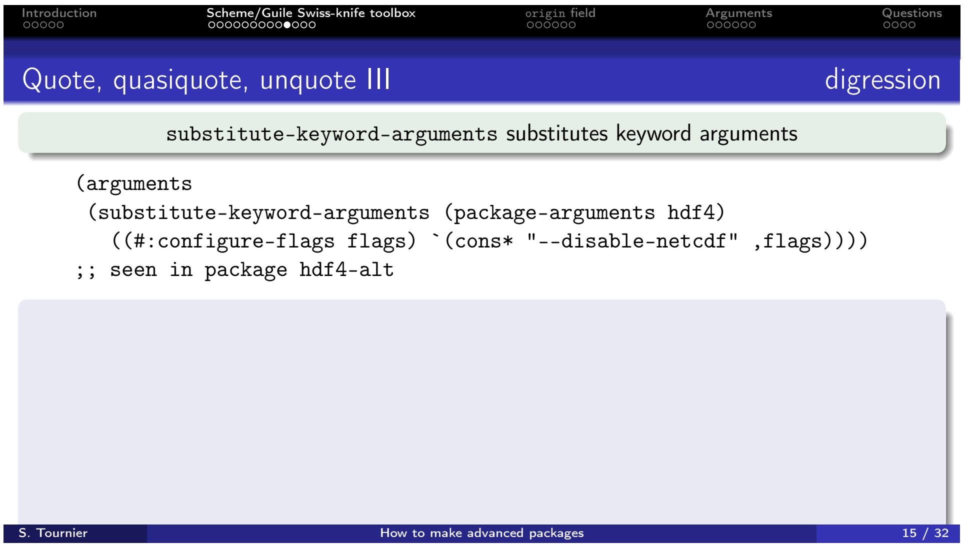


Quote, quasiquote, unquote III digression

substitute-keyword-arguments substitutes keyword arguments

```
(arguments
 (substitute-keyword-arguments (package-arguments hdf4)
   ((#:configure-flags flags) `(cons* "--disable-netcdf" ,flags))))
;; seen in package hdf4-alt
```

S. Tournier How to make advanced packages 15 / 32


Introduction Scheme/Guile Swiss-knife toolbox origin field Arguments Questions

Quote, quasiquote, unquote III digression

substitute-keyword-arguments substitutes keyword arguments

(arguments
 (substitute-keyword-arguments (package-arguments hdf4)
   ((#:configure-flags flags) `(cons* "--disable-netcdf" ,flags))))
;; seen in package hdf4-alt

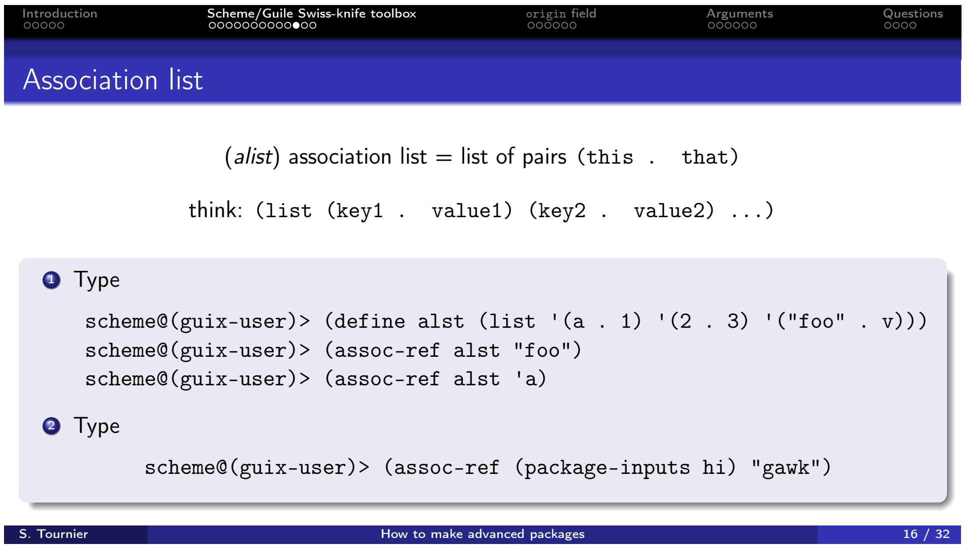
S. Tournier How to make advanced packages 15 / 32

the module `(guix build-system python)`.

It is possible to interactively explore these sequences of actions using `guix repl`. Here, we load the GNU build system `%standard-phases`, and we rebind that variable to `standard-phases` (without the percent `%`) and prefix it with `gnu`:

```
scheme@(guix-user)> (use-modules ((guix build gnu-build-system)
  #:select ((%standard-phases . standard-phases))  ;also rename
  #:prefix gnu:))
```

Similarly, let load the sequence of actions for the Python build system,



Introduction · **Scheme/Guile Swiss-knife toolbox** · origin field · Arguments · Questions

Association list

(*alist*) association list = list of pairs `(this . that)`

think: `(list (key1 . value1) (key2 . value2) ...)`

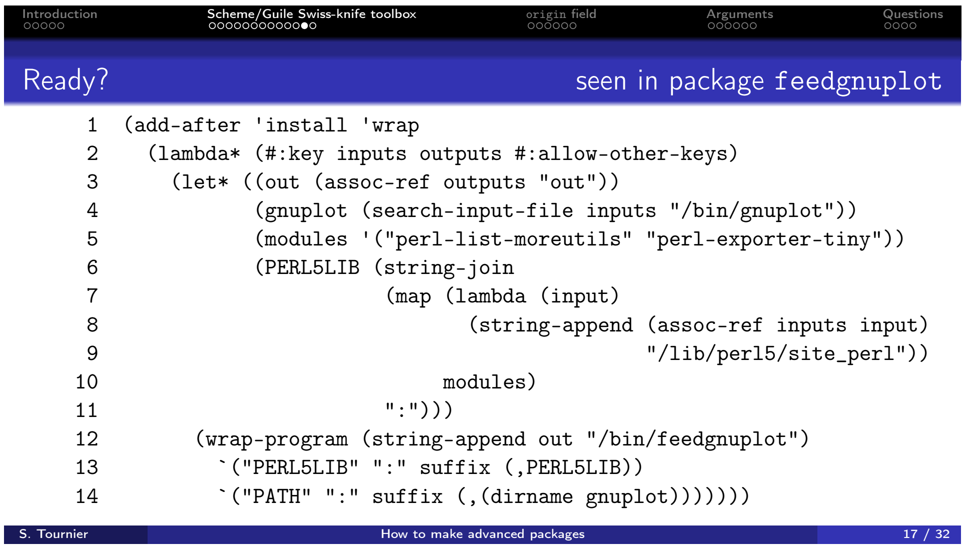
1. Type

```
scheme@(guix-user)> (define alst (list '(a . 1) '(2 . 3) '("foo" . v)))
scheme@(guix-user)> (assoc-ref alst "foo")
scheme@(guix-user)> (assoc-ref alst 'a)
```

2. Type

```
scheme@(guix-user)> (assoc-ref (package-inputs hi) "gawk")
```

S. Tournier · How to make advanced packages · 16 / 32



Introduction · **Scheme/Guile Swiss-knife toolbox** · origin field · Arguments · Questions

Ready? seen in package `feedgnuplot`

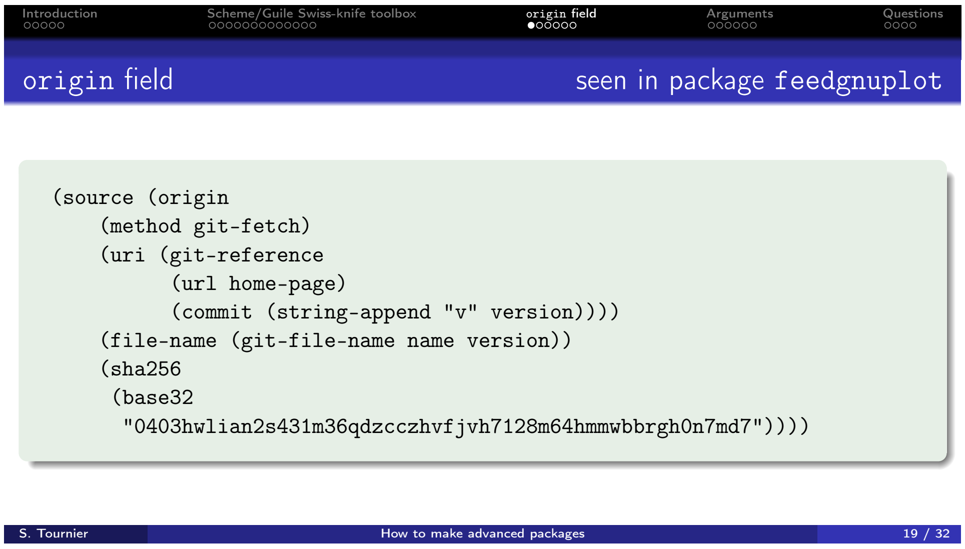
```
(add-after 'install 'wrap
  (lambda* (#:key inputs outputs #:allow-other-keys)
    (let* ((out (assoc-ref outputs "out"))
           (gnuplot (search-input-file inputs "/bin/gnuplot"))
           (modules '("perl-list-moreutils" "perl-exporter-tiny"))
           (PERL5LIB (string-join
                      (map (lambda (input)
                             (string-append (assoc-ref inputs input)
                                            "/lib/perl5/site_perl"))
                           modules)
                      ":")))
      (wrap-program (string-append out "/bin/feedgnuplot")
        `("PERL5LIB" ":" suffix (,PERL5LIB))
        `("PATH" ":" suffix (,(dirname gnuplot)))))))
```

S. Tournier · How to make advanced packages · 17 / 32

```
scheme@(guix-user)> (use-modules ((guix build python-build-system)
  #:select ((%standard-phases . standard-phases))
  #:prefix python:))
```

And using `lset-difference` from the module `(srfi srfi-1)`, it is straightforward to list the items that are part of the GNU build system but not part of the Python build system.

```
scheme@(guix-user)> ,use(srfi srfi-1)
scheme@(guix-user)> ,pp (lset-difference eq?
                          gnu:standard-phases python:standard-phases)
```

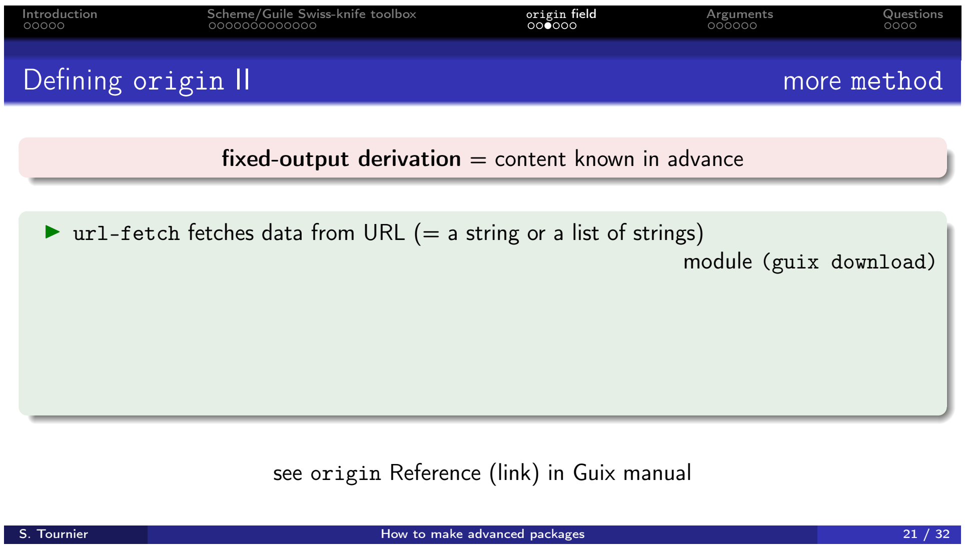


Introduction · Scheme/Guile Swiss-knife toolbox · origin field · Arguments · Questions

origin field — seen in package feedgnuplot

```
(source (origin
    (method git-fetch)
    (uri (git-reference
          (url home-page)
          (commit (string-append "v" version))))
    (file-name (git-file-name name version))
    (sha256
     (base32
      "0403hwlian2s431m36qdzcczhvfjvh7128m64hmmwbbrgh0n7md7"))))
```

S. Tournier — How to make advanced packages — 19 / 32



Introduction · Scheme/Guile Swiss-knife toolbox · origin field · Arguments · Questions

Defining origin II — more method

fixed-output derivation = content known in advance

- `url-fetch` fetches data from URL (= a string or a list of strings)

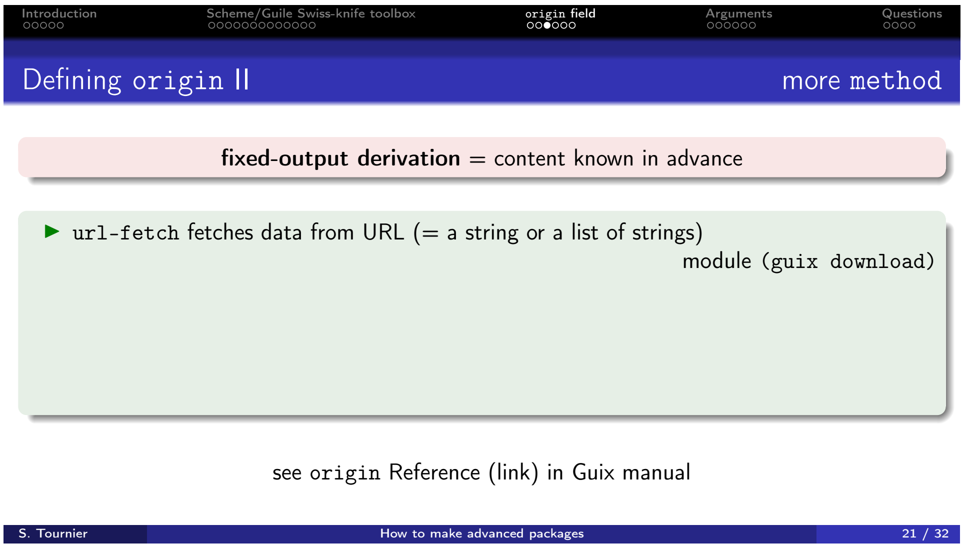
  module `(guix download)`

see `origin` Reference (link) in Guix manual

S. Tournier — How to make advanced packages — 21 / 32

Based on that, we are able to compare the sequence of actions (`%standard-phases`) for two build systems. And we could have a simple script for automating:

```
$ ./examples/scripts/compare-bs-phases.scm gnu python

Compare %standard-phases of "gnu" and "python":
  22 phases in "gnu"
  28 phases in "python"
  20 phases in common
```



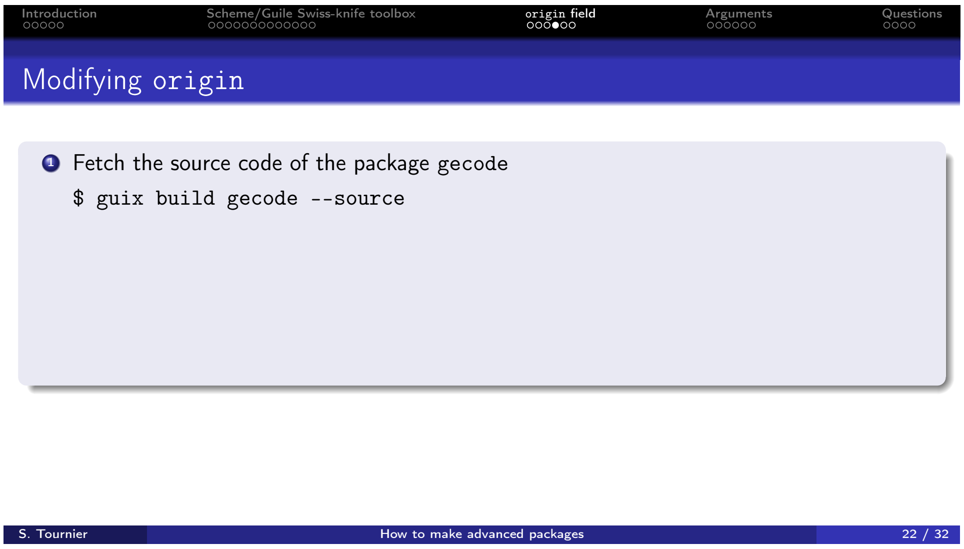
Defining origin II — more method

fixed-output derivation = content known in advance

- url-fetch fetches data from URL (= a string or a list of strings)

  module (guix download)

see origin Reference (link) in Guix manual

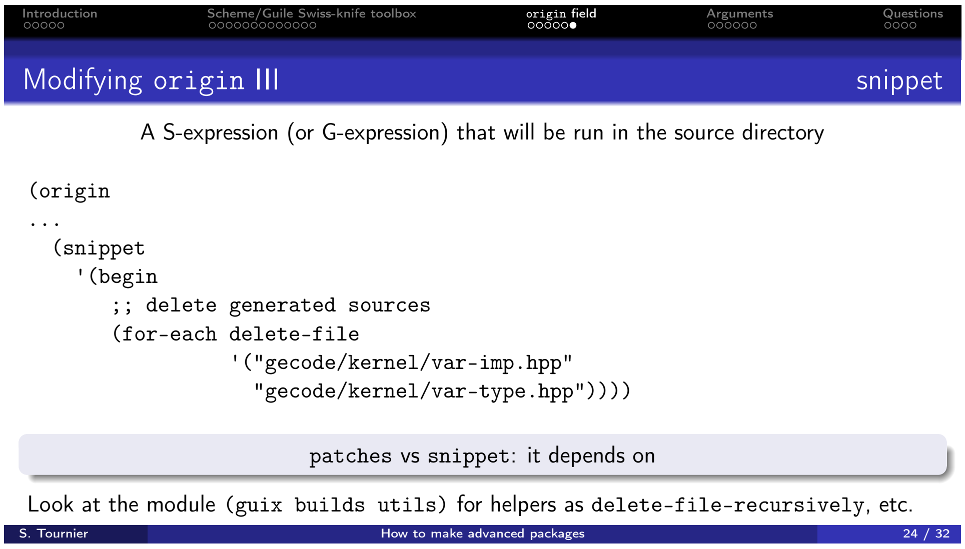


Modifying origin

1. Fetch the source code of the package gecode

   `$ guix build gecode --source`

```
Only in "gnu":
 + bootstrap
 + configure

Only in "python":
 + ensure-no-mtimes-pre-1980
 + enable-bytecode-determinism
 + ensure-no-cythonized-files
 + add-install-to-pythonpath
 + add-install-to-path
```


Introduction
Scheme/Guile Swiss-knife toolbox
origin field
Arguments
Questions

Modifying `origin` III — `snippet`

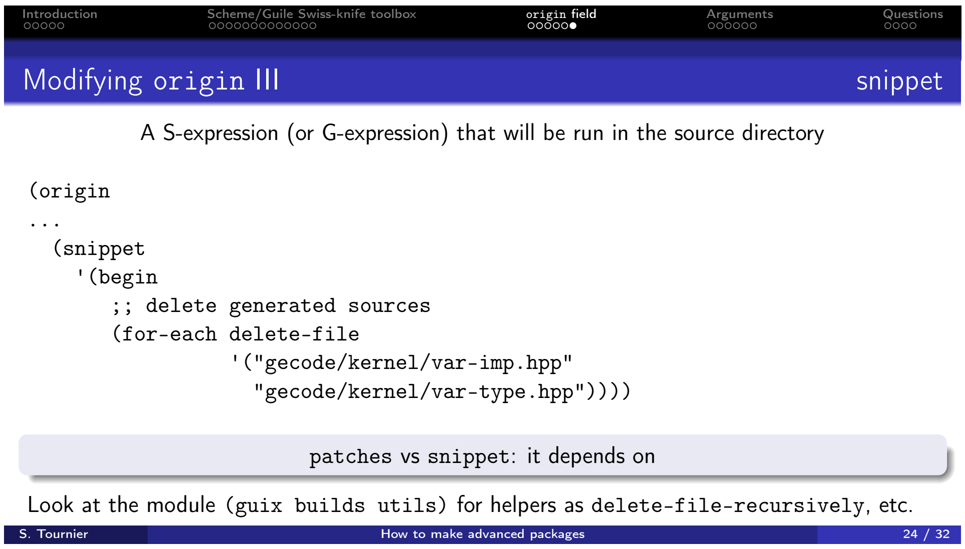
A S-expression (or G-expression) that will be run in the source directory

```
(origin
...
  (snippet
    '(begin
       ;; delete generated sources
       (for-each delete-file
                 '("gecode/kernel/var-imp.hpp"
                   "gecode/kernel/var-type.hpp"))))
```

`patches` vs `snippet`: it depends on

Look at the module `(guix builds utils)` for helpers as `delete-file-recursively`, etc.

S. Tournier
How to make advanced packages
24 / 32


Modifying origin III snippet

A S-expression (or G-expression) that will be run in the source directory

(origin
...
  (snippet
    '(begin
       ;; delete generated sources
       (for-each delete-file
                 '("gecode/kernel/var-imp.hpp"
                   "gecode/kernel/var-type.hpp"))))

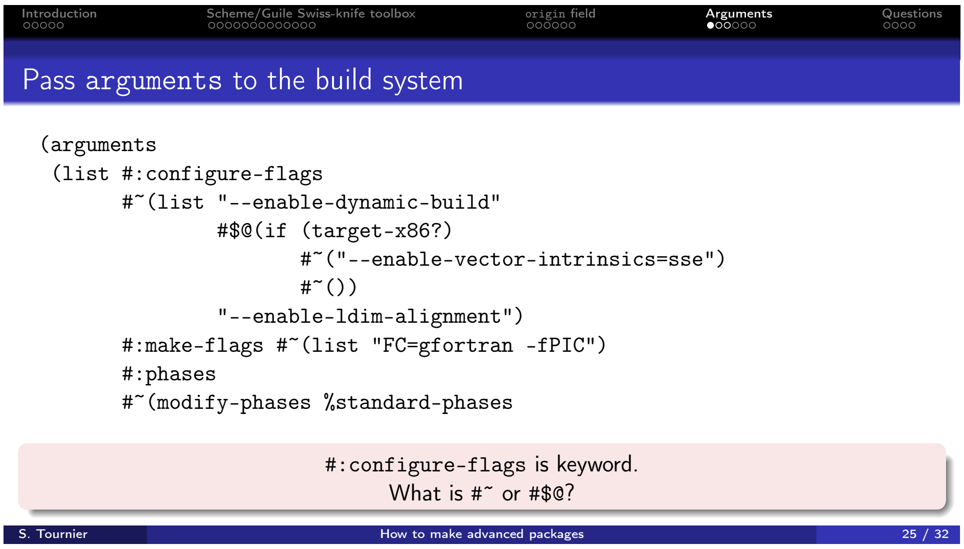
patches vs snippet: it depends on

Look at the module (guix builds utils) for helpers as delete-file-recursively, etc.

```
+ wrap
+ sanity-check
+ rename-pth-file
```

The manual provides some details about the meaning of various phases, see Build Systems.

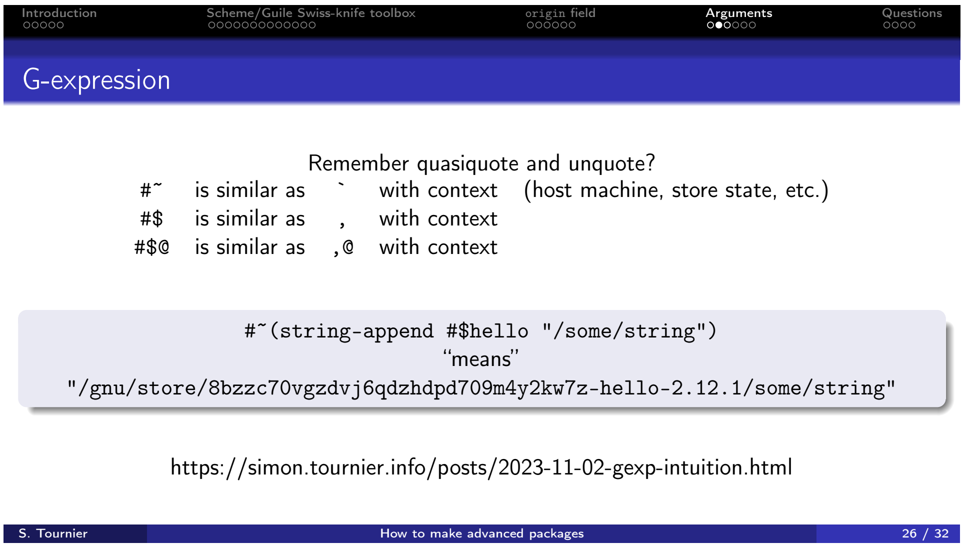
5 Questions



Pass arguments to the build system

```
(arguments
 (list #:configure-flags
       #~(list "--enable-dynamic-build"
               #$@(if (target-x86?)
                      #~("--enable-vector-intrinsics=sse")
                      #~())
               "--enable-ldim-alignment")
       #:make-flags #~(list "FC=gfortran -fPIC")
       #:phases
       #~(modify-phases %standard-phases
```

`#:configure-flags` is keyword.
What is `#~` or `#$@`?



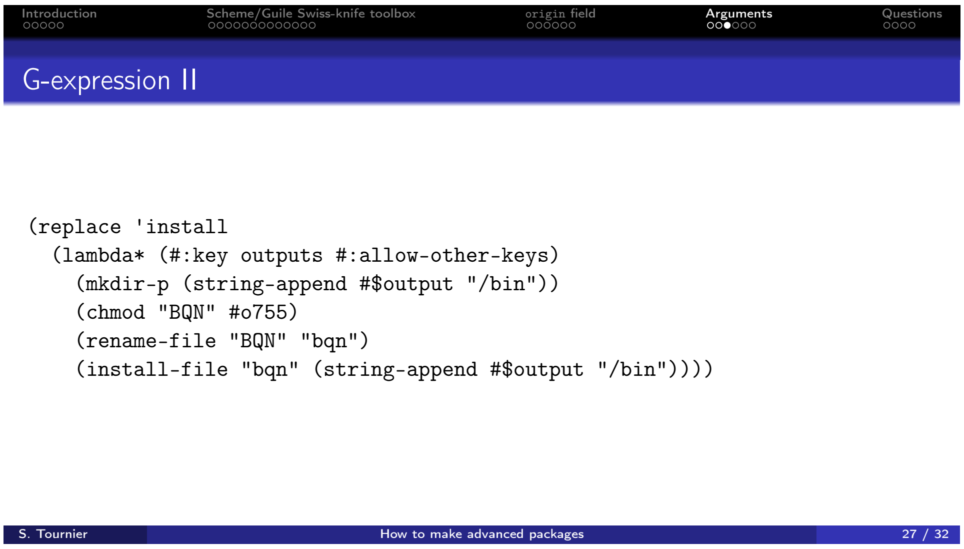
G-expression

Remember quasiquote and unquote?

`#~`	is similar as	`` ` ``	with context	(host machine, store state, etc.)
`#$`	is similar as	`,`	with context	
`#$@`	is similar as	`,@`	with context	

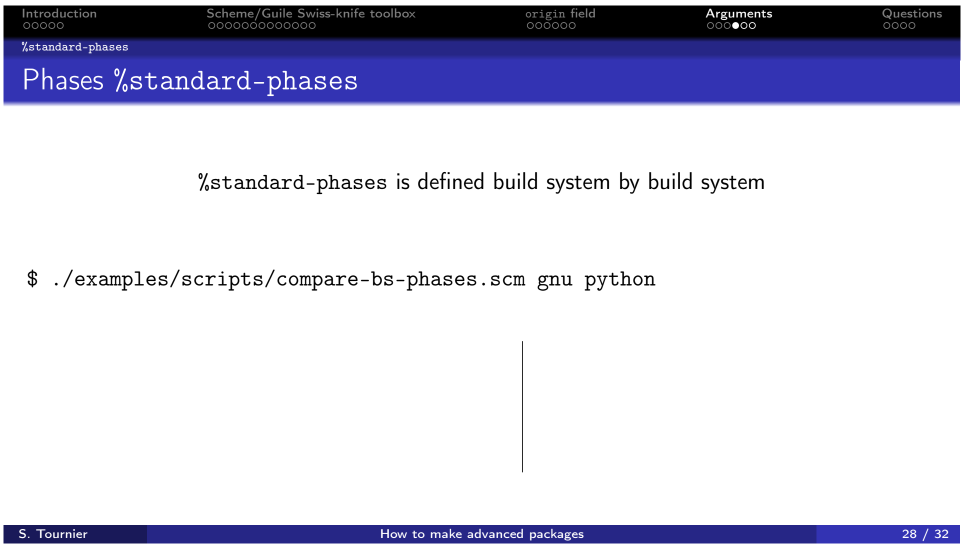
`#~(string-append #$hello "/some/string")`
"means"
`"/gnu/store/8bzzc70vgzdvj6qdzhdpd709m4y2kw7z-hello-2.12.1/some/string"`

https://simon.tournier.info/posts/2023-11-02-gexp-intuition.html



G-expression II

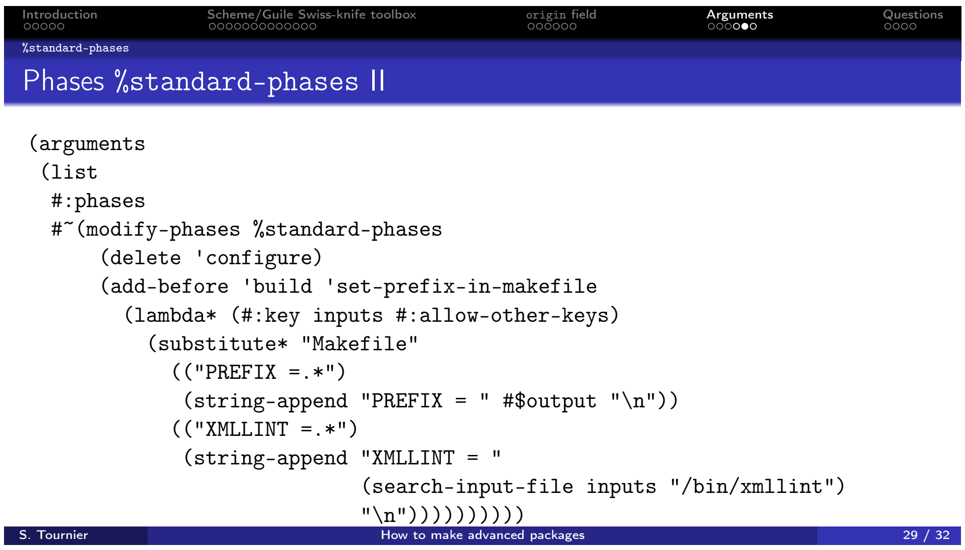
```
(replace 'install
  (lambda* (#:key outputs #:allow-other-keys)
    (mkdir-p (string-append #$output "/bin"))
    (chmod "BQN" #o755)
    (rename-file "BQN" "bqn")
    (install-file "bqn" (string-append #$output "/bin"))))
```



Phases %standard-phases

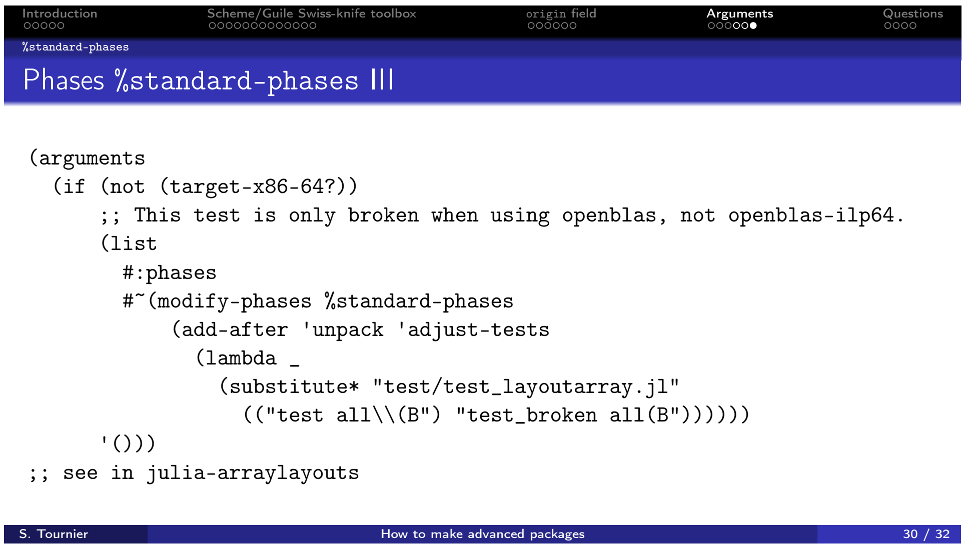
%standard-phases is defined build system by build system

```
$ ./examples/scripts/compare-bs-phases.scm gnu python
```



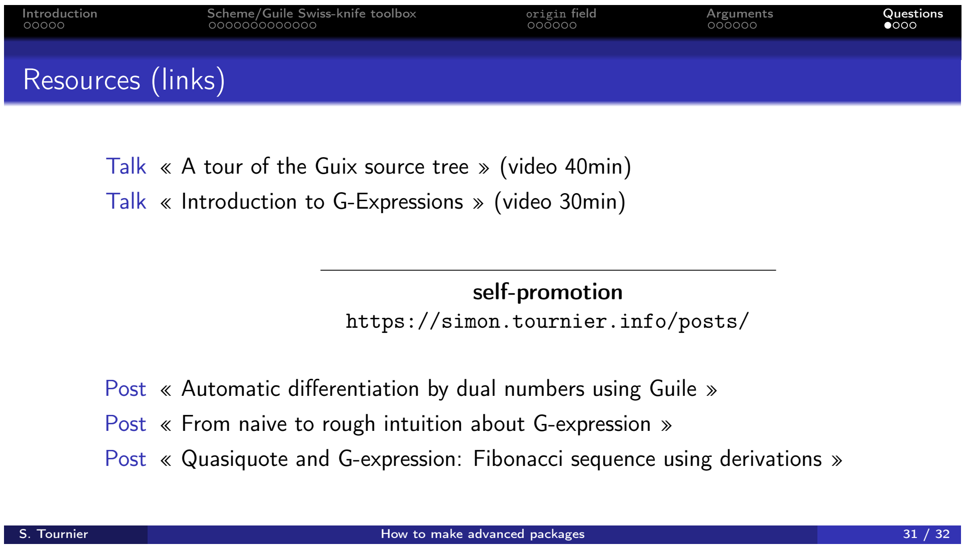
Phases %standard-phases II

```
(arguments
 (list
  #:phases
  #~(modify-phases %standard-phases
      (delete 'configure)
      (add-before 'build 'set-prefix-in-makefile
        (lambda* (#:key inputs #:allow-other-keys)
          (substitute* "Makefile"
            (("PREFIX =.*")
             (string-append "PREFIX = " #$output "\n"))
            (("XMLLINT =.*")
             (string-append "XMLLINT = "
                            (search-input-file inputs "/bin/xmllint")
                            "\n"))))))))
```



Phases %standard-phases III

```
(arguments
  (if (not (target-x86-64?))
      ;; This test is only broken when using openblas, not openblas-ilp64.
      (list
        #:phases
        #~(modify-phases %standard-phases
            (add-after 'unpack 'adjust-tests
              (lambda _
                (substitute* "test/test_layoutarray.jl"
                  (("test all\\(B") "test_broken all(B"))))))
      '()))
;; see in julia-arraylayouts
```

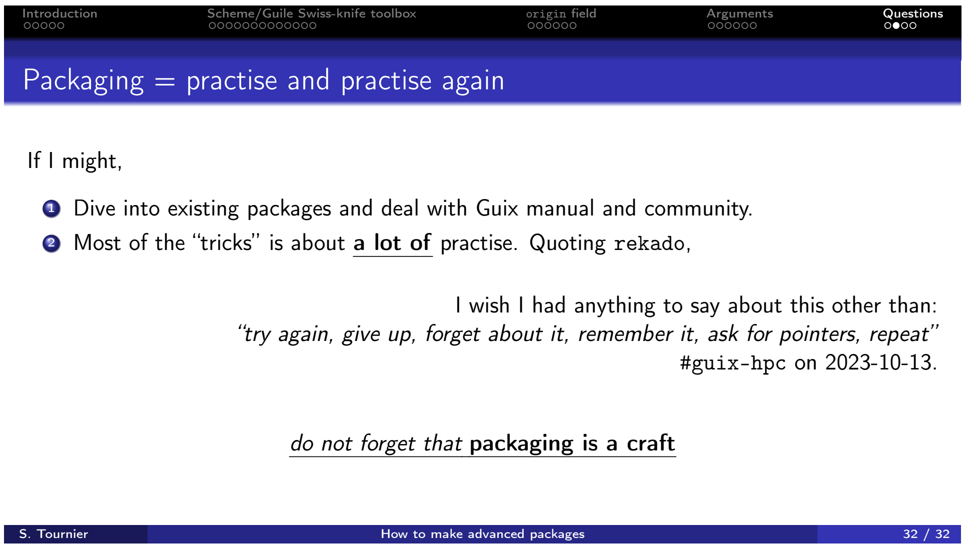


Resources (links)

Talk « A tour of the Guix source tree » (video 40min)
Talk « Introduction to G-Expressions » (video 30min)

self-promotion
`https://simon.tournier.info/posts/`

Post « Automatic differentiation by dual numbers using Guile »
Post « From naive to rough intuition about G-expression »
Post « Quasiquote and G-expression: Fibonacci sequence using derivations »



Packaging = practise and practise again

If I might,

1. Dive into existing packages and deal with Guix manual and community.
2. Most of the "tricks" is about **a lot of** practise. Quoting `rekado`,

I wish I had anything to say about this other than:
"try again, give up, forget about it, remember it, ask for pointers, repeat"
`#guix-hpc` on 2023-10-13.

do not forget that **packaging is a craft**

A First intuition about G-expression

G-expressions make it easy to write to files Scheme code that refers to store items, or to write Scheme code to build derivations. This section is an attempt to build an intuition about them. Please take this section as a first introduction containing some approximations for clarity.

The vehicle for the journey is `guix repl`. Let start by binding the variable `hi` to the string `"path/to/hi"` and let append the string `"/bin/bye"`.

```
scheme@(guix-user)> (define hi "path/to/hi")
scheme@(guix-user)> (begin (string-append hi "/bin/bye"))
$1 = "path/to/hi/bin/bye"
```

Now consider that we do not want to fully evaluate the resulting string but we would like to construct an intermediate expression, which will be then evaluated. The quasiquote allows to protect the evaluation and unquote to escape this protection.

Scheme is able to control what is evaluated when constructing an expression and what is evaluated when computing this expression. It is about quasiquote and unquote; they are so familiar that they have their own syntactic sugar: backtick (`) and comma (,). For instance,

```
scheme@(guix-user)> `(begin (string-append ,hi "/bin/bye"))
$2 = (begin (string-append "path/to/hi" "/bin/bye"))
```

Now, this expression can be evaluated later. Let say later is now:

```
scheme@(guix-user)> (eval $2 (interaction-environment))
$3 = "path/to/hi/bin/bye"
```

Here the term $2 refers to the interactive previous result. Please note that the expression is evaluated in the context of the `(interaction-environment)`. So far, so good.

What if the variable refers to a package? Let check it. First, we import the packages from the module `base` and the variable `hello` corresponds to the package hello.

```
scheme@(guix-user)> (use-modules (gnu packages base))
scheme@(guix-user)> hello
$4 = #<package hello@2.12.1 gnu/packages/base.scm:90 7f0d0bba8dc0>
```

Let tweak the previous expression and replace the variable `hi` by the variable `hello`.

```
scheme@(guix-user)> `(begin (string-append ,hello "/bin/bye"))
$5 = (begin (string-append #<package hello@2.12.1 gnu/packages/base.scm:90 ...>
                            "/bin/bye"))
```

Interresting isn't it? Then, as previously, we are going to evaluate it.

```
scheme@(guix-user)> (eval $5 (interaction-environment))
ice-9/boot-9.scm:1685:16: In procedure raise-exception:
In procedure string-append: Wrong type (expecting string): #<package hello@2.12.1

Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guix-user) [1]> ,q
```

Bang! An error. Well, it was expected, no? The variable `hello` does not refer to a string so it does not make sense to append with one other string. **G-expressions make it easy to write to files Scheme code that refers to store items**. Let try that. Instead of quasiquote, let introduce gexp. And instead of unquote, let introduce ungexp. And similarly, they have their syntatic sugar:

quasiquote	(backtick)	`	`#~`	gexp
unquote	(comma)	,	`#$`	ungexp

Before continuing, we need to load the module providing all this fun.

```
scheme@(guix-user)> (use-modules (guix gexp))
```

We are ready for replacing the two symbols:

```
scheme@(guix-user)> #~(begin (string-append #$hello "/bin/bye"))
$6 = #<gexp (begin (string-append #<gexp-input #<package hello@2.12.1
                                   "/bin/bye")) 7f0d0a9b6780>
```

Nice, it returns a G-expression type (`gexp`).

Warning: The following procedure is an helper for easing the explanations of this section. *Please skip it elsewhere.* It does not matter what it means or how to invoke it and, to my knowledge, it is not required for writing packages.

```
scheme@(guix-user)> (define gexp->sexp (@@ (guix gexp) gexp->sexp))
```

Once typed, consider it transforms from the G-expression type (`gexp`) to the regular S-expression type (`sexp`). And let be back to the explanations.

```
scheme@(guix-user)> (gexp->sexp $6 "x86_64-linux" #f)
$7 = #<procedure 7f0d0ac011e0 at guix/gexp.scm:1387:2 (state)>
```

Wait, the result is a procedure. And it seems that a `state` is required. What does it mean? Somehow, it captures the context for the evaluation. How to evaluate? The interactive `guix repl` provides an handy `,run-in-store` command – Guix specific REPL command; here the comma (`,`) is not related to unquote but to the command switch. Let run this procedure in the context of the Guix local store.

```
scheme@(guix-user)> ,run-in-store (gexp->sexp $7 "x86_64-linux" #f)
$8 = (begin
  (string-append
    "/gnu/store/8bzzc70vgzdvj6qdzhdpd709m4y2kw7z-hello-2.12.1"
    "/bin/bye"))
```

Bingo! We get a S-expression where the G-expression had been replaced by the value. Now we would be able to evaluate this S-expression and append the two strings.

G-expressions make it easy to manipulate Scheme code refering to store items.

Variation

In the very first example, the whole expression is quasiquoted and only the variable `hi` is unquoted. Why not unquote all the append? It would mean that the string append will be happened at construction time and not at evaluation time.

```
scheme@(guix-user)> `(begin ,(string-append hi "/bin/bye"))
$9 = (begin "path/to/hi/bin/bye")
scheme@(guix-user)> (eval $9 (interaction-environment))
$10 = "path/to/hi/bin/bye"
```

Similarly as previously, let just replace the symbols:

```
scheme@(guix-user)> #~(begin #$(string-append hello "/bin/bye"))
```

Again, as previously, it raises an error and it is expected. It does not make sense to append the type package with the type string.

```
ice-9/boot-9.scm:1685:16: In procedure raise-exception:
In procedure string-append: Wrong type (expecting string): #<package hello@2.12.1

Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guix-user) [1]> ,q
```

What is the solution? The solution is to not use the procedure `string-append` and instead rely on the procedure `file-append` which is designed for these kind of situations.

```
scheme@(guix-user)> #~(begin #$(file-append hello "/bin/bye"))
$11 = #<gexp (begin #<gexp-input #<file-append #<package hello@2.12.1
                                             "/bin/bye">:out>) 7f0d09225900>
scheme@(guix-user)> (gexp->sexp $11 "x86_64-linux" #f)
$12 = #<procedure 7f0d09393450 at guix/gexp.scm:1387:2 (state)>
scheme@(guix-user)> ,run-in-store $12
$13 = (begin
  "/gnu/store/8bzzc70vgzdvj6qdzhdpd709m4y2kw7z-hello-2.12.1/bin/bye")
```

B Quasiquote and G-expressions

From Guix: Quasiquote and G-expressions blog post.

Well, the best explanation is "show me the code", isn't it?

let start `guix repl` and explore...

Before introducing *G-expressions* we need to have a clear idea about classic Scheme concepts `quasiquote` (` ) and `unquote` ( , ); as well as `quote` ('). In Lisp-family language, "*quoted*" data remains *unevaluated.*

```
scheme@(guix-user)> (define x 42)
scheme@(guix-user)> x
$1 = 42
scheme@(guix-user)> (quote x)
$2 = x
```

This `quote` is so frequent that it has syntactic sugar ('). Somehow, `'(x y z)` is short for `(list 'x 'y 'z)`.

```
scheme@(guix-user)> '(x y z)
$3 = (x y z)
scheme@(guix-user)> (list? '(x y z))
$4 = #t
scheme@(guix-user)> (list 'x 'y 'z)
$5 = (x y z)
scheme@(guix-user)> (equal? '(x y z) (list 'x 'y 'z))
$6 = #t
```

Everything on the list is a value – e.g., `'x` or `'y` or `'z`; namely there are symbols. In short, a symbol looks like a variable name except that it starts with `quote` (') and it plays a role similar as strings; somehow symbols are a great way to represent "*symbolic*" information as data.

That's said, we would like to be able to escape back inside of a quoted list and evaluate something. Thanks to `quasiquote` (` ) and `unquote` (,), it is possible.

```
scheme@(guix-user)> (define y 24)
scheme@(guix-user)> `(,x y z)
$7 = (42 y z)
```

Here, the `unquoted` expression is evaluated during the construction of the list, while the other remaining unevaluated. Another example:

```
scheme@(guix-user)> (define something 'cool)
scheme@(guix-user)> (define (tell-me) `(Guix is ,something))
scheme@(guix-user)> (tell-me)
```

```
$8 = (Guix is cool)
scheme@(guix-user)> (define something 'awesome)
scheme@(guix-user)> (tell-me)
$9 = (Guix is awesome)
```

And `unquote` evaluates everything, including procedures if any.

```
scheme@(guix-user)> `(Again ,(tell-me))
$10 = (Again (Guix is awesome))
scheme@(guix-user)> `(1 ,(+ 2 3) 4)
$11 = (1 5 4)
```

For instance, let build[1] the Fibonacci sequence.

```
scheme@(guix-user)> (define (fibo n)
                      (if (or (= 0 n) (= 1 n))
                          `(begin ,n)
                          (let ((f_1 (fibo (- n 1)))
                                (f_2 (fibo (- n 2))))
                            `(begin (+ ,f_1 ,f_2)))))
scheme@(guix-user)> ,pp (fibo 7)
$12 = (begin
  (+ (begin
       (+ (begin
            (+ (begin
                 (+ (begin
                      (+ (begin (+ (begin 1) (begin 0))) (begin 1)))
                    (begin (+ (begin 1) (begin 0)))))
               (begin
                 (+ (begin (+ (begin 1) (begin 0))) (begin 1)))))
          (begin
            (+ (begin
                 (+ (begin (+ (begin 1) (begin 0))) (begin 1)))
               (begin (+ (begin 1) (begin 0)))))))
     (begin
       (+ (begin
            (+ (begin
                 (+ (begin (+ (begin 1) (begin 0))) (begin 1)))
               (begin (+ (begin 1) (begin 0)))))
          (begin
            (+ (begin (+ (begin 1) (begin 0))) (begin 1)))))))
```

Here, nothing is evaluated. All is data and symbolically manipulated. The evaluation (computation) is then done with `eval`,

[1]It would be possible to make the code more symmetric and implement the Fibonacci sequence using `quasiquote` / `unquote` and files. However, it makes everything far more complicated. Somehow, that's why G-expressions had been introduced, after all! ;-)

```
scheme@(guix-user)> (eval (fibo 7) (interaction-environment))
$13 = 13
```

So far, so good. **What about G-expressions?** Quoting dedicated section of Guix manual:

> To describe a derivation and its build actions, one typically needs to embed build code inside host code. It boils down to manipulating build code as data, and the homoiconicity of Scheme — code has a direct representation as data — comes in handy for that. But we need more than the normal `quasiquote` mechanism in Scheme to construct build expressions.

Somehow, G-expressions simplifies the machinery for staging code. Compared to classic expression, it introduces both: context – e.g., the set of inputs associated with the expression – and the ability to serialize high-level objects – i.e., to replace a reference to a package object with its `/gnu/store/` file name.

G-expressions consist of syntactic forms: `gexp`, `ungexp` – or simply: `#~` and `#$` – which are comparable to `quasiquote` and `unquote`, respectively. Other said, it allows to control the context of the evaluation.

Let make it concrete with the Fibonacci example. We need the modules `(guix gexp)`, `(guix derivations)` and `(guix store)`.

```
scheme@(guix-user)> (use-modules (guix gexp)
                                 (guix derivations)
                                 (guix store))
```

blank

Now, instead of manipulating `quasiquote` (` ) and `unquote` ( `,` ), we are going to manipulate `gexp` ( `#~` ) and `ungexp` ( `#$`). Let start with three helpers:

```
(define (number->name n)
  (string-append "Fibonacci-of-"
                 (number->string n)))

(define (number->gexp n)
  #~(begin
      (use-modules (ice-9 format))
      (call-with-output-file #$output
        (lambda (port)
          (format port "~d" #$n)))))

(define (store-item->number path)
  #~(begin
      (use-modules ((ice-9 textual-ports) #:select (get-string-all)))
      (string->number
       (call-with-input-file #$path
         get-string-all))))
```

Nothing special to say about `number->name`. What do the others do? `number->gexp` takes an interger number and returns a G-expression, such that, after evaluation, it will write this number to some file. What makes the machinery convenient is that `#$output` will be replaced – evaluated with adequate context – by a string containing the output reference to its `/gnu/store/` file name. Can you guess what `store-item->number` does?

The core: computing the Fibonacci sequence using G-expressions,

```
(define (fibonacci n)
  (if (or (= 0 n) (= 1 n))
      (gexp->derivation
       (number->name n)
       (number->gexp n))
      (let* ((store (open-connection))
             (drv-1 (run-with-store store
                      (fibonacci (- n 1))))
             (drv-2 (run-with-store store
                      (fibonacci (- n 2))))
             (f_1 (store-item->number drv-1))
             (f_2 (store-item->number drv-2)))
        (gexp->derivation
         (number->name n)
         (number->gexp #~(+ #$f_1 #$f_2))))))
```

First, `gexp->derivation` returns a derivation with the name (`number->name`) that runs the `gexp` (`number->gexp`); somehow this `gexp` stores something to compute. Second, `run-with-store` runs a `gexp` in the context of the `store` – see The Store monad.

The procedure `fibonacci` takes an integer number and constructs some G-expressions controlling the context of evaluation. Let run it!

```
scheme@(guix-user)> ,run-in-store (fibonacci 7)
$14 = #<derivation /gnu/store/db97xy9d5icaa64n2n9l7q2v66npmm6c-Fibonacci-of-7.drv
    => /gnu/store/8b5g05g4z5r9f3ash53ppb5m1r7kksfj-Fibonacci-of-7 7f7ca8ed3640>
```

Awesome! We get back a derivation. Note the handy `,run-in-store` command – Guix specific REPL command – which hides the plumbing of `run-with-store`. From the REPL, it is possible to explore this derivation, although the pretty-printer is not handy. Well, this derivation reads:

```
Derive
([("out","/gnu/store/8b5g05g4z5r9f3ash53ppb5m1r7kksfj-Fibonacci-of-7","","")]
 ,[("/gnu/store/0wkxvd2ll0gff37wghamb12dz4x50n14-Fibonacci-of-6.drv",["out"])
   ,("/gnu/store/9r95y1j1rg4q7vb528lh51w0cz3c5hvi-Fibonacci-of-5.drv",["out"])
   ,("/gnu/store/zraigp7miin3vzr5dcbr4i9rvds0i07r-guile-3.0.9.drv",["out"])]
 ,["/gnu/store/z0jspla9advx77ihbc7nfjvnky2gfvjz-Fibonacci-of-7-builder"]
 ,"x86_64-linux"
 ,"/gnu/store/g8p09w6r78hhkl2rv1747pcp9zbk6fxv-guile-3.0.9/bin/guile"
 , ["--no-auto-compile"
 ,  "/gnu/store/z0jspla9advx77ihbc7nfjvnky2gfvjz-Fibonacci-of-7-builder"]
 ,[("out","/gnu/store/8b5g05g4z5r9f3ash53ppb5m1r7kksfj-Fibonacci-of-7")])
```

Figure 1: the derivation Fibonacci-of-7.drv

And the 7th term depends on the 6th and 5th ($F_7 = F_6 + F_5$); the expected recursive sequence. The interesting part is the builder.

blank

```
(begin
  (use-modules (ice-9 format))
  (call-with-output-file
      ((@ (guile) getenv) "out")
    (lambda (port)
      (format port "~d"
              (+
               (begin
                 (use-modules ((ice-9 textual-ports)
                   #:select
                   (get-string-all)))
                 (string->number
                  (call-with-input-file
                    "/gnu/store/rhjmlgaz4f1niwhrnm2nsfdj2g6dya6h-Fibonacci-of-6"
                   get-string-all)))
               (begin
                 (use-modules
                  ((ice-9 textual-ports) #:select (get-string-all)))
                 (string->number
                  (call-with-input-file
                    "/gnu/store/xcp9b4j0nskk6lk5jxlpv3926j19vpw0-Fibonacci-of-5"
                   get-string-all)))))))
```

Figure 2: the builder of Fibonacci-of-7.drv

All had been correctly replaced. Somehow, that builder script is similar as the output of the previous `fibo` procedure, and instead of `eval`, now the computation will be done by Guix daemon evaluating this builder script.

Let make that computation!

```
scheme@(guix-user)> ,build $14
$15 = "/gnu/store/8b5g05g4z5r9f3ash53ppb5m1r7kksfj-Fibonacci-of-7"
```

And guess what? This file contains the value 13. Yeah!

As you can see, all the previous values of the Fibonacci sequence are also computed via derivations and the result stored as files. Guix daemon starts to construct the derivation `Fibonacci-of-0.drv`, then `Fibonacci-of-1.drv`, and compute them by writing 0 then 1 inside the output store item files `Fibonacci-of-0` and `Fibonacci-of-1`. Then Guix daemon evaluates the builder script of the derivation `Fibonacci-of-2.drv`, i.e., it reads the values from two previous store item files, adds them and writes the result inside the store item file `Fibonacci-of-2`. The Guix daemon repeats until 7. Other said, if we want to compute the value for the 8th Fibonacci number, all the previous computations are cached in the Guix store; the Guix store acts as a good memoization mean. Check it with:

```
$ guix gc --list-dead | grep Fibonacci-of
```

C Do not shoot yourself in the foot

Closing remark about G-expressions, let mention that "*With great power there must also come great responsibility*". Once defined a package, Guix daemon, for concretely building it, will first construct a derivation that mainly lists two components:

- All the other derivations as dependencies,
- How to build the output artefact with a Scheme builder script.

And doing so, it will keep track of the information about the derivations that the G-expressions refer to.

Example unrelated to packaging, see Fibonacci sequence as implemented in section B. For instance, Figure 1 (p.24) shows the derivation and Figure 2 (p.25) the associated script builder for computing the 7th Fibonacci number. As we can see, because the builder depends on some store items, namely `Fibonacci-of-5` and `Fibonacci-of-6`, their derivations are listed in `Fibonacci-of-7.drv`. And we have not explicitely mentioned this dependency relationship. That's the G-expression machinery which does the job.

That's great! Where is the potential shoot? Let create a package named `bye` which just adds only one phase compared to the base package `hello`,

blank

```
(define-module (appendix)
  #:use-module (guix packages)
  #:use-module (gnu packages base)
  #:use-module (guix gexp)
  #:use-module (gnu packages emacs))

(define-public bye
  (package
    (inherit hello)
    (name "bye")
    (arguments
     (list
      #:phases
      #~(modify-phases %standard-phases
          (add-after 'install 'do-something-with-emacs
            (lambda _
              (invoke #$(file-append emacs-minimal
                                     "/bin/emacs") "--version"))))))))
```

Please note there is no package `inputs`, i.e., the package `emacs-minimal` is listed nowhere. When we build this package, we will see:

```
$ guix build -L example bye --no-grafts
...
phase ‘install’ succeeded after 0.2 seconds
starting phase ‘do-something-with-emacs’
GNU Emacs 29.1
Copyright (C) 2023 Free Software Foundation, Inc.
GNU Emacs comes with ABSOLUTELY NO WARRANTY.
You may redistribute copies of GNU Emacs
under the terms of the GNU General Public License.
For more information about these matters, see the file named COPYING.
phase ‘do-something-with-emacs’ succeeded after 0.0 seconds
...
```

Wow, the package `emacs-minimal` is not listed as inputs and it is used, thanks to G-expressions and its machinery that captured it. No magic, all is transparent: it appears in the derivation `bye.drv`,

```
$ cat $(guix build -L example bye --no-grafts --derivations) \
   | sed ’s/),/\n/g’  # transform the one single line to several
   | grep emacs
("/gnu/store/1nagdikm5rwdf8ilp5r607112wrnrlhi-emacs-minimal-29.1.drv",["out"]
```

Thanks to Guix great transparency, we only wind on, no shoot.

What about the builder script? If you open it, you will see,

```
              #:phases
              (modify-phases %standard-phases
                (add-after
                    (quote install)
                    (quote do-something-with-emacs)
                  (lambda _
                    (invoke "/gnu/store/9y222z4lgfyddi2k65ycc1nx6cal73ic-emacs-minim
```

Nice, G-expression rocks! However, this item is not listed under `%build-inputs` passed to the `gnu-build` system.

Concretely, what does it mean? Because it is a *abuse* of G-expression machinery, it escapes some Guix features as Package-Transformation-Options or Invoking `guix-refresh`. To make a long story short, most package manipulations operate on a level that cannot access to such plumbing level.

Hope that what G-expression means and how to use them is clearer. My conclusion is: *Don't fear the G-expression.*