

題目 Scheme,Guile 勉強ノート¹⁾／ Guile 入門

作者 AlgoKajya

日付 2021.02.09

概要 **Scheme** の処理系は色々あります。筆者は GNU プロジェクトの一つである **Guile** を使うことにしました。Guile は「GNU Ubiquitous Intelligent Language for Extensions」の略だそうです。GNU Project の公式拡張言語だそうです。「Guile」はたぶん「ガイル」と読むのだと思います²⁾。このノートは、Guile の基本的な利用方法を説明しています。

筆者は **Debian Linux** を使用しています³⁾。そのため、すべての説明は Debian に基づいています。たぶん、Ubuntu などの Debian 派生のディストリビューションでも同じように作業できるだろうと思います。すべての作業はターミナル上で行います。Guile や Scheme に固有のコマンドを除いて、Linux の一般的なコマンドについては特段の説明は行いません。

参照資料に関して次のような略号を使用しています。

- GRM Guile Reference Manual 3.0.4 (および 2.2.6)⁴⁾
- R⁵RS Revised⁵ Report on the Algorithmic Language Scheme
- R⁶RS Revised⁶ Report on the Algorithmic Language Scheme
- R⁷RS Revised⁷ Report on the Algorithmic Language Scheme

¹⁾Lisp を学び直そうと思い立って、Scheme を勉強しています。処理系は Guile を使っています。Scheme や Guile について勉強したことをまとめて、勉強ノートを趣味として作っています。一応、読み物にしています。他の人達の参考になるかどうかは分かりませんが、参考になるのなら好き勝手に使ってください。ただし、内容は無保証です。使うにしても自己責任をお願いします。

²⁾英語力は高くないので信用しないで下さい。ただ、英語としての発音はたぶん「ガイル」だと思います。一方、GRM 9.1.2 に次のような記述があります。「Finally, "Guile" suggests "guy-ell", or "Guy L. Steele", who, together with Gerald Sussman, originally discovered Scheme.」もしかしたら「ガイエル」なのかも知れません。

³⁾勉強を始めた 2020.10.30 時点の安定版の最新バージョンは 10.6、通称 buster です。

⁴⁾勉強を始めた 2020.10.30 時点の Guile の最新版は 3.0.4 です。Debian の公式パッケージは 2.2.4 です。3.0.4 と 2.2.4 のマニュアルを見るようにしているのですが、このノートが扱っている範囲では両方の内容に違いはないように思います。ノート内のマニュアルの参照はすべて 3.0.4 のものです。

目 次

| | | |
|-------|-------------------------------|----|
| 第 1 章 | はじめに | 4 |
| 1.1 | Guile をインストールする | 4 |
| 1.2 | 動作確認 — REPL を動かしてみる — | 4 |
| 1.3 | 履歴機能を設定する | 5 |
| 第 2 章 | REPL — プログラムを実行する (その 1) — | 6 |
| 2.1 | サンプルプログラム | 6 |
| 2.2 | REPL を起動してプログラムをロードする | 6 |
| 2.3 | プログラムを変更・修正して再ロード | 7 |
| 2.4 | 自前の手続きを実行する | 8 |
| 2.5 | エラーが発生したとき | 8 |
| 2.6 | セッションが終わるまで, その束縛, 生きてます | 9 |
| 2.7 | その他 | 10 |
| 2.7.1 | キャッシュ上のバイトコード | 10 |
| 2.7.2 | 諸々 | 11 |
| 第 3 章 | スクリプト — プログラムを実行する (その 2) — | 13 |
| 3.1 | guile コマンドを使って実行する | 13 |
| 3.1.1 | -s スイッチを使ってプログラムを実行する | 13 |
| 3.1.2 | コマンドライン引数を指定する | 13 |
| 3.1.3 | -e スイッチを使ってエン트리ポイントを指定する | 14 |
| 3.1.4 | エン트리ポイントは仮引数が必須です | 16 |
| 3.2 | スクリプト (単独のプログラム) として実行する | 17 |
| 3.2.1 | スクリプトの作成と実行 | 17 |
| 3.2.2 | シェバンとブロックコメント | 18 |
| 3.2.3 | コマンドライン引数 | 19 |
| 3.3 | 複数のスイッチ (特に, エントリーポイント) を指定する | 20 |
| 3.3.1 | メタスイッチ | 20 |
| 3.3.2 | 2 行目の書き方はとても注意が必要です | 21 |
| 3.3.3 | Why メタスイッチ | 23 |
| 3.4 | その他 | 25 |
| 3.4.1 | 他のスイッチ | 25 |
| 3.4.2 | ソースファイルの文字コード | 26 |
| 第 4 章 | プログラムの全体像 | 28 |
| 4.1 | R ⁷ RS | 28 |
| 4.2 | Guile | 28 |
| 付 録 A | | 32 |
| A.1 | Scheme や Guile の情報源 | 32 |
| A.2 | REPL コマンド | 33 |
| A.3 | 旧版のマニュアルを入手する | 36 |
| A.3.1 | 一般的な方法 — Gnu のアーカイブを利用する | 36 |
| A.3.2 | Debian な方法 — ソースパッケージを利用する | 38 |

| | |
|---------------------------------|----|
| A.4 最新版の Guile をビルドする | 38 |
|---------------------------------|----|

更新履歴

第1章 はじめに

1.1 Guile をインストールする

筆者が勉強を始めた2020.10.30時点で、GuileのDebian用公式パッケージとしてguile-2.2 (Guile 2.2.4) が配布されています。そこで、次のコマンドを実行して guile-2.2 をインストールします。ドル記号 (\$) はターミナルのプロンプトを表します。

```
$ sudo apt install guile-2.2 guile-2.2-dev
```

guile-2.2-dev にはプログラム開発用のツールが入っています。それらを使うかどうか不明なのですが、一応、インストールしておきます。インストールはこれで完了です。

1.2 動作確認 — REPL を動かしてみる —

インストール後の確認作業として、Guile の **REPL** を動かしてみます。

■ REPL の起動

1.1

Guile の REPL を起動するには、ターミナル上で次のコマンドを実行します。

```
$ guile
```

これを実行すると、次のようなメッセージのあとにプロンプトが表示されてプログラムが入力できるようになります。scheme@(guile-user)> がプロンプトです。

```
GNU Guile 2.2.4
Copyright (C) 1995-2017 Free Software Foundation, Inc.

Guile comes with ABSOLUTELY NO WARRANTY; for details type ',show w'.
This program is free software, and you are welcome to redistribute it
under certain conditions; type ',show c' for details.

Enter ',help' for help.
scheme@(guile-user)>
```

試しに、簡単な式を入力して実行してみます。


```
scheme@(guile-user)> (+ 10 20)
$1 = 30
scheme@(guile-user)> (* 10 20)
$2 = 200
```

■ REPL の終了

1.2

以下のいずれかによって終了できます。

```
scheme@(guile-user)> (quit)
または
scheme@(guile-user)> Ctrl-D を押す
または
scheme@(guile-user)> ,quit
または
scheme@(guile-user)> ,q
```

または
`scheme@(guile-user)> (exit)` 

¶ Guile Reference Manual によると、次節で説明する履歴機能を設定した場合、`(quit)` または `Ctrl-D` によってセッションを終了したときには履歴を残すという説明があります (GRM 7.9.1)。さらに、`exit` は `quit` の alias であり、同じ機能を持つといった説明があります (GRM 7.2.7)。`,quit` および `,q` については同様の説明はありません (GRM 4.4.4.8)。

¶ `,quit` は REPL 固有のコマンドです。`,q` は `,quit` の省略形です。Guile の REPL 固有のコマンドは、Scheme の変数名や構文キーワードなどと確実に区別するため、カンマで始めることになっています (GRM 4.4.4)。

1.3 履歴機能を設定する

`~/.guile` に次のコードを保存すると、Guile の REPL で **履歴機能** が使えるようになります。このファイルがなければ新たに作ります。実行属性は不要です。

```
1 (use-modules (ice-9 readline))
2 (activate-readline)
```

この設定によって Guile を起動したときに履歴機能が有効になります。履歴機能は上下のカーソルキー (↑, ↓) で使えます。さらに、タブキーによる **補完機能** (定義済みの変数名を補完する機能) も使えるようになります。

¶ 履歴は `~/.guile_history` に保存されます。

¶ `~/.guile` は Guile の初期化ファイルです。Scheme のプログラムを保存しておきます。Guile を起動したときに、初期設定の一環として、これに保存してあるプログラムが実行されます。上で示した 1 行目は `(ice-9 readline)` というモジュールをロードして、2 行目はその中の `activate-readline` 手続きを実行します。

¶ タブキーによる補完機能は、定義済みの変数名を補完する機能であって、ファイル名やコマンド名を補完する機能ではありません。

¶ GRM 4.4.1, 4.4.2, 7.9.1

第2章 REPL — プログラムを実行する（その1） —

REPL の中で `load` 手続きを使ってプログラムを実行するといった方法を説明します。

2.1 サンプルプログラム

以後の説明のためにプログラムを2つ用意します。

■ `hello.scm`

2.3

一つ目はお約束の `hello world` プログラムです。適当なエディタを使って次のプログラムを作成します。セミコロンから始まる行はコメント行です。このファイル名を `hello.scm` とします。

```
1 ;; hello.scm
2 (display "*** Hello, world! ***")
3 (newline)
```

1 行目は `"*** Hello, world! ***"` という文字列を表示して、2 行目は改行しています。

■ `fib.scm`

2.4

2 つ目のプログラムは、これもお約束のフィボナッチ数を求めるプログラムです。フィボナッチ数 f_n ($n \geq 0$) は次の漸化式によって定義される整数です。

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2} \quad (n \geq 2)$$

以下のプログラムはこの漸化式を素直に実現しています。このファイル名を `fib.scm` とします。

```
1 ;; fib.scm
2 (define (fib n)
3   (if (<= n 1)
4       n
5       (+ (fib (- n 1)) (fib (- n 2)))))
```

■ ソースファイルの拡張子

2.5

Guile は、`‘.scm’` をソースファイルの標準の拡張子にしています。さらに、Emacs を使っているのならば、Emacs も拡張子が `.scm` のファイルを Scheme のソースファイルと判別し、キーワードハイライトや適切なインデントを行ってくれます。

¶ GRM 4.2.2, 6.18.7

2.2 REPL を起動してプログラムをロードする

まず、プログラムファイルを保存したディレクトリ上で REPL を起動します。

```
$ guile
GNU Guile 2.2.4
.....
.....起動メッセージ.....
.....
scheme@(guile-user)>
```

次に **load** 手続きを使ってプログラムファイルをロードします。load 手続きの一般的な形式は
 (load "ファイル名")
 です。

```
scheme@(guile-user)> (load "hello.scm")  
;;; .....
;;;      ..... (コンパイルに関するメッセージ) .....
;;; .....
*** Hello, world! ***
scheme@(guile-user)>
```

load 手続きを実行すると、プログラムファイルがコンパイルされて、そのメッセージが色々と表示されます。そのあとにプログラムの実行結果が表示されます。2つ目のプロンプトの直前にある「*** Hello, world! ***」という表示が実行結果です。コンパイルメッセージが混みいって、実行結果が分かりづらいのですが慣れるしかありません。

■ ファイル名の指定方法

2.6

ソースコードのファイル名は、ターミナル上で指定するのと同様に、相対パスや絶対パスで指定します。相対パスの起点はカレントディレクトリ（REPL を実行したディレクトリ）です。ただし、ホームディレクトリを表すティルダ ‘~’ は使えません（POSIX で定義されていないため）。

📌 上で述べているファイル名の指定方法は筆者の経験則であって、load 手続きの説明文には何も書かれていません。REPL の中で load 手続きを使っているかぎり、経験から言って上の通りだと思います。一方、**primitive-load** という手続きがあって、そちらのほうの説明文にはファイル名は上の通りであることが明記されています。

📌 GRM 6.18.6

2.3 プログラムを変更・修正して再ロード

プログラムを変更したり修正したときには load 手続きを使って再びロードします。例えば **hello.scm** を次のように変更してみます。

```
1 ;; hello.scm
2 (display "*** Hello, world! ***")
3 (newline)
4 (display "*** Hello, Guile! ***")
5 (newline)
```

これを再びロードすると、改めてコンパイルされて実行されます。

```
scheme@(guile-user)> (load "hello.scm")  
;;; .....
;;;      ..... (コンパイルに関するメッセージ) .....
;;; .....
*** Hello, world! ***
*** Hello, Guile! ***
scheme@(guile-user)>
```

REPL を使ってデバッグをしながらプログラムを作成するときには、修正と再ロードを繰り返します。

2.4 自前の手続きを実行する

もう一つの `fib.scm` をロードしてみます。

```
scheme@(guile-user)> (load "fib.scm")
;;; .....
;;;      ..... (コンパイルに関するメッセージ) .....
;;; .....
scheme@(guile-user)>
```

今度はコンパイルメッセージだけで実行結果らしきものは何も表示されません。 `define` 形式をロードすると `fib` 手続きの定義が現在のセッションに取り込まれるだけで数値的な計算は行いません。フィボナッチ数を計算するためには、自然数を実引数として `fib` 手続きを実行する必要があります。

```
scheme@(guile-user)> (fib 3)
$1 = 2
scheme@(guile-user)> (fib 6)
$2 = 8
scheme@(guile-user)> (fib 10)
$3 = 55
scheme@(guile-user)>
```

■ \$1, \$2, \$3 ...

2.7

‘`$n`’は計算結果に付けた一時的な変数名です。同じセッションの中でこの変数名を使って計算結果を利用することができます。例えば、次の式は上の操作に続けて `$1=2` と `$2=8` の積を求めています。

```
scheme@(guile-user)> (* $1 $2)
$4 = 16
scheme@(guile-user)>
```

これらの一時変数を **value history name** と呼びます。

¶ GRM 4.4.3

2.5 エラーが発生したとき

エラーが発生したとき、REPL はそれまでのセッションをいったん凍結して、エラーの状態を保持した新たなセッションを開始します。REPL を再帰的に実行してデバッグモードを開始するとも言えます。試しに、`hello.scm` を次のように変更します。2 番目の `display` をわざと `dipslay` にしています。

```
1 ;; hello.scm
2 (display "*** Hello, world! ***")
3 (newline)
4 (dipslay "*** Hello, Guile! ***")
5 (newline)
```

この `hello.scm` をロードしてみます。

```
scheme@(guile-user)> (load "hello.scm")
;;; .....
;;;      ..... (コンパイルに関するメッセージ) .....
scheme@(guile-user)>
```



```
;;; .....
*** Hello, world! ***
..... hello.scm:2:0: In procedure module-lookup: Unbound variable: dipslay

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

1行目と2行目には誤りはないので "*** Hello, world! ***" は表示されます。そのあとのところでエラーが発生しています。赤字がエラーメッセージです。「**dipslay** は未束縛（変数の値が未定義）だ」と怒っています。ただ、プログラムファイル名がフルパスで表示されるため、とても見にくいと思います。

それから、プロンプト（青字）に '**[1]**' が付いています。これはREPLがネスト（再帰）していることを示しています。数字はネスト（再帰）の深さを示しています。例えば、上の状態でさらにエラーが発生すると、この数字が '**[2]**' になります。試しに、誤りを含んだ `hello.scm` をもう一度ロードしてみてください。

デバッグモードのセッションはREPLを再帰的に実行しているだけなので、(quit) や Ctrl-D によって元々の（あるいは一つ前の）セッションに戻ることができます。ネストしているからと言って何も恐れることはありません。それに、ネストを放置していると、どんどんネストしていきます。エラーの原因が判明したり、プログラムを再実行するときには、ネストを解消したほうがよいでしょう。

❏ GRM 4.4.4.8, 4.4.5

2.6 セッションが終わるまで、その束縛、生きてます

REPL の load 手続きを使ってプログラムの動作を確認するとき、次の点に注意が必要です。

- 一度ロードして確立した束縛は、セッションが終わるまで有効です。

具体例を使って説明します。

次の2つのプログラムを考えましょう。1番目のファイル名を `sample-1.scm` とし、2番目を `sample-2.scm` とします。

```
1 ;; sample-1.scm
2 (define x 10)
3 (define y 20)
4 (display "x+y=") (display (+ x y)) (newline)
```

```
1 ;; sample-2.scm
2 (define y 99)
3 (display "x+y=") (display (+ x y)) (newline)
```

`sample-2.scm` は、**x** の定義をわざと外しています。x の値は不明ですから、当然、`(+ x y)` を実行するところでエラーが発生します。

これら2つのプログラムを同じセッションの中で実行してみます。まず `sample-1.scm` をロードし、続けて `sample-2.scm` をロードします。

```
$ guile❏
GNU Guile 2.2.4
..... 起動メッセージ .....
scheme@(guile-user)> (load "sample-1.scm")❏
```

```

..... コンパイルメッセージ .....
x+y=30
scheme@(guile-user)> (load "sample-2.scm")
..... コンパイルメッセージ .....
x+y=109
scheme@(guile-user)>

```

両方とも問題なく実行できています。sample-2.scm が実行できたのは、sample-1.scm をロードしたときに x の値が定義され、それを利用したからです。ちなみに、`,binding` というコマンドを使うと、各時点の束縛の一覧が表示されます。

```

scheme@(guile-user)> ,binding (先頭のカンマを忘れずに)
y                #<variable 562fda9a18b0 value: 99>
x                #<variable 562fda9a18e0 value: 10>
myload           #<variable 562fda6640a0 value: #<procedure myload (a)>>
scheme@(guile-user)>

```

これを見ると、x が 10 に束縛され、y が 99 に束縛されていることが分かります。

REPL の 1 つのセッションの中で色々なプログラムをロードしたとき、あるプログラムがうまく動いたからといって、そのプログラムが単独に正しく動作するとは限りません。確実に確認したいときには REPL を起動し直してプログラムを実行すべきでしょう。REPL を再起動して sample-2.scm を実行してみるとエラーがきちんと発生します。

```

$ guile
GNU Guile 2.2.4
..... 起動メッセージ .....
scheme@(guile-user)> (load "sample-2.scm")
x+y=In procedure module-lookup: Unbound variable: x
scheme@(guile-user)>

```

エラーメッセージがなんだか少しヘンですが、(display "x+y=") までは問題なく実行できて、そのあとの加算が実行できずにエラーメッセージを表示しているためです。

📌 プログラムを再ロードしたり、複数のプログラムをロードしたときに、ある変数名の定義（束縛）が重複した場合、最後に確立された束縛が有効です。上の例で言えば、sample-1.scm をロードしたときに変数 y はいったん 20 に束縛されますが、sample-2.scm をロードすると変数 y は 99 に束縛され直されます。binding コマンドは各時点での最新の束縛を表示しています。

📌 Guile の REPL 内で使用するコマンド（,binding など）は、Scheme の変数名や構文キーワードなどと確実に区別するため、カンマで始めることになっています（GRM 4.4.4）。

2.7 その他

2.7.1 キャッシュ上のバイトコード

Guile は、ホームディレクトリ上に **キャッシュディレクトリ**（`~/.cache/guile`）を作っていて、そこにコンパイル後の **バイトコード** を保存しています。コンパイルメッセージの中にバイトコードのファイル名が絶対パスで表示されます。

バイトコードファイルがソースファイルよりも古くならない限り、再コンパイルは行わずにキャッシュしたバイトコードを実行するようです。そのため、コンパイル済みのプログラムを再ロードしてみるとコンパイルメッセージは表示されません。

¶ GRM 4.2.2

2.7.2 諸々

■ -l スイッチ

2.8

guile コマンドの-l スイッチを使うと、REPL の起動と同時にプログラムファイルをロードできます。これは自前のソースライブラリをロードするときに役立ちます。例えば、mylibs.scm というファイルに自前の便利な手続きが色々入っていたとします。このとき、次のようにすれば REPL の起動と同時に mylibs.scm をロードできます。さらに、-l スイッチ（によるロード）は幾つでも指定できます。詳細は省略します。

```
$ guile -l mylibs.scm
```

■ デバッグモードを回避する

2.9

Guile の初期化ファイル ~/.guile に次の 2 行を追加するとデバッグモードが回避できます。

```
1 (use-modules (system repl common))
2 (repl-default-option-set! 'on-error 'report)
```

これを設定するとデバッグモードには入らずにエラーメッセージだけが表示されます。試しに、Guile を起動し直して先ほどのバグを含んだプログラムをロードしてみます。

```
$ guile
GNU Guile 2.2.4
.....
.....起動メッセージ.....
.....
scheme@(guile-user)> (load "hello.scm")
*** Hello, world! ***
In procedure module-lookup: Unbound variable: dipslay
scheme@(guile-user)>
```

エラーメッセージ（赤字）は表示されますが、デバッグモードを示す数字は表示されません。これはデバッグモードに入っていないことを示しています。

¶ 2 行目にある 'report' の代わりに 'backtrace' を指定すると、バックトレース（エラーが発生するところまでの実行過程）を含んだエラーメッセージが表示されます。それから、'debug' を指定するとデバッグモードが復活します。

¶ 一つ注意しておきたいのは、デバッグモードの回避を推奨しているわけではありません。ただ回避が可能だということを記録しているだけです。筆者はデバッグモードを残すべきだと思っています。

■ myload 手続き

2.10

せっかく Scheme (Guile) を勉強しているので、ファイル名の先頭にティルダ '~' が付いているとき、それをホームディレクトリの絶対パスに変換してロードする、といった手続きを作ってみます。

```
1 (define (myload filename)
2   (let* ((homedir (passwd:dir (getpwuid (getuid))))
3         (fname (if (char=? (string-ref filename 0) #\~)
```

```

4      (string-append homedir (substring filename 1))
5      filename)))
6  (primitive-load fname)))

```

`homedir` はホームディレクトリの絶対パスで, `fname` はティルダ (~) をホームディレクトリの絶対パスに置き換えたファイル名です.

この手続きを `~/.guile` に保存します. これによって REPL を実行したときにこの手続きの定義が自動的にロードされます. 以下, `hello.scm` (誤りのないもの) が `~/tmp` ディレクトリに保存されているとして, `myload` 手続きを試してみます.

```

$ guile
GNU Guile 2.2.4
..... 起動メッセージ .....
scheme@(guile-user)> (myload "~/tmp/hello.scm")
*** Hello, world! ***
*** Hello, Guile! ***
scheme@(guile-user)>

```

うまく動きました.

第3章 スクリプト — プログラムを実行する（その2） —

プログラムファイルをスクリプト（単体のプログラム）として実行する方法を説明します。

3.1 guile コマンドを使って実行する

¶ GRM4.2.1

3.1.1 -s スイッチを使ってプログラムを実行する

guile コマンドの **-s スイッチ** を利用することによって、REPL に入ることなく Scheme のプログラムが実行できます。

```
$ guile -s <script> █
```

ここで、<script> は Scheme プログラムのファイルを表しています。

■ 例

3.11

以下の Scheme プログラムを `greetings.scm` というファイルに保存したとします。

```
1 ;; greetings.scm
2 (display "Hello, Scheme!\n")
3 (display "Hello, Guile!\n")
```

このプログラムを実行してみます。

```
$ guile -s greetings.scm █
..... (コンパイルに関するメッセージ) .....
Hello, Scheme!
Hello, Guile!
```

¶ ファイル名がハイフン (-) で始まるのでない限り、'-s' を省略することもできます。

```
$ guile <script> █
```

でも、このノートでは省略しないことにします。

3.1.2 コマンドライン引数を指定する

上で示した形式に **コマンドライン引数** を指定することもできます。

```
$ guile -s <script> <arg> ...
```

ここで、<arg> は記号列を表します。記号列は幾つでも指定できます。各記号列は空白によって区切ります。これを実行すると、<script> を実行する際に、

```
'("<script>" "<arg>" ... )
```

といった文字列からなるリストが構成され、<script> 自身へ実引数として渡されます。この実引数は **command-line** 手続きを使って取得できます。

■ 例

3.12

以下の Scheme プログラムを `arglist.scm` というファイルに保存したとします。

```
1 ;; arglist.scm
2 (write (command-line))
3 (newline)
```

このプログラムは、コマンドライン引数から構成された

```
'("<script>" "<arg>" ... )
```

というリストを `command-line` 手続きによって取得して、そのリストを表示します。

```
$ guile -s arglist.scm AAA BBB CCC
.....
..... (コンパイルに関するメッセージ) .....
.....
("arglist.scm" "AAA" "BBB" "CCC")
```

最後の行が実行結果です。

📌 スクリプトに渡されるコマンドライン引数のリストの先頭は、スクリプトのファイル名です。

3.1.3 -e スイッチを使ってエントリーポイントを指定する

`-s` スイッチと一緒に **-e スイッチ** を利用することによって、プログラム内の **エントリーポイント** (実行を開始する手続き) を指定することができます。

```
$ guile -e <proc> -s <script>
または
$ guile -e <proc> -s <script> <arg> ...
```

ここで、`<proc>` はプログラム内の手続き名を表します。2 番目の形式はコマンドライン引数 (`<arg> ...`) を指定する場合を示しています。

コマンドライン引数 (`<arg> ...`) を指定しなかったときには、`<script>` を実行する際に、

```
'("<script>")
```

といったリストが構成され、そのリストが `<proc>` へ実引数として渡されて、`<proc>` から `<script>` の実行が始まります¹⁾。

一方、コマンドライン引数 (`<arg> ...`) を指定したときには、

```
'("<script>" "<arg>" ... )
```

といったリストが構成され、そのリストが `<proc>` へ実引数として渡されて、`<proc>` から `<script>` の実行が始まります²⁾。

■ 例

3.13

次のプログラムを `greetings.scm` に保存したとします。なお、仮引数の `args` をまったく使っていないので奇妙に感じるかも知れません。上で述べたように、これはエントリーポイントの手続きには必ず実引数が渡されるためです。この点についてはのちほど詳しく説明します。

```
1 ;; greetings.scm
2 (define (hello-scheme args)
3   (display "Hello, Scheme!") (newline))
```

¹⁾ この説明は不正確です。エントリーポイントに関する動作のより正確な説明は 4.2 節を見て下さい。

²⁾ 同上。4.2 節を見て下さい。

```

4 |
5 | (define (hello-guile args)
6 |   (display "Hello, Guile!") (newline))

```

それぞれの手続きをエントリーポイントとして実行してみます。

```

$ guile -e hello-scheme -s greetings.scm
.....
..... (コンパイルに関するメッセージ) .....
.....

Hello, Scheme!
$ guile -e hello-guile -s greetings.scm
Hello, Guile!

```

1 つ目は `hello-scheme` 手続きを実行し、2 つ目は `hello-guile` 手続きを実行しています。なお、1 つの目のときにコンパイルが済んでいるので、2 つ目のときにはコンパイルされません。

■ 例

3.14

コマンドライン引数を利用したプログラム例を示します。ファイル名は `greetings.scm` とします。

```

1 | ;; greetings.scm
2 | (define (hello-everybody args)
3 |   (write args) (newline)
4 |   (for-each hello-somebody (cdr args)))
5 |
6 | (define (hello-somebody arg)
7 |   (display (string-append "Hello," arg "!"))
8 |   (newline))

```

```

$ guile -e hello-everybody -s greetings.scm Alice Bob Carol
.....
..... (コンパイルに関するメッセージ) .....
.....

("greetings.scm" "Alice" "Bob" "Carol")
Hello,Alice!
Hello,Bob!
Hello,Carol!

```

`hello-everybody` は、ファイル名を含めたコマンドライン引数からなるリスト

```
("greetings.scm" "Alice" "Bob" "Carol")
```

を受け取って、そのリストを表示したあと、先頭要素（ファイル名）を除く残りの各文字列（`"Alice"`, `"Bob"`, `"Carol"`）に対して `hello-somebody` を適用しています。

■ エントリーポイントの `<proc>` へ実引数として渡されたリストは、プログラムの中のあらゆる場所で `command-line` 手続きを使って取得することもできます。例えば、次のプログラムを `arglist.scm` に保存したとします。

```

1 | ;; arglist.scm
2 | (define (get-arguments args)
3 |   (display "arg:") (write args) (newline)
4 |   (display "cmd:") (write (command-line)) (newline))

```

この手続きは実引数の `args` と `command-line` 手続きによって取得したコマンドライン引数の両方を表示します。

```
$ guile -e get-arguments -s arglist.scm AAA BBB CCC DDD
.....
..... (コンパイルに関するメッセージ) .....
.....
arg:("arglist.scm" "AAA" "BBB" "CCC" "DDD")
cmd:("arglist.scm" "AAA" "BBB" "CCC" "DDD")
```

`get-arguments` の引数 `args` にもコマンドライン引数が渡されるので、両方とも同じリストが表示されます。

3.1.4 エントリーポイントは仮引数が必須です

コマンドライン引数 (`<arg> ...`) を何も指定しないときでも、

```
'("<script>")
```

といったリスト（ファイル名だけからなるリスト）が、エントリーポイントの手続きに実引数として渡されます。従って、

- エントリーポイントの手続きは必ず仮引数を用意しなければなりません。

そうしないとエラーが発生します。

例えば、`greetings.scm` の `hello-scheme` 手続きを次のように変更して実行してみます。

```
1 ;; greetings.scm
2 (define (hello-scheme)
3   (display "Hello, Scheme!") (newline))
```

```
$ guile -e hello-scheme -s greetings.scm
.....
..... (コンパイルに関するメッセージ) .....
.....
Backtrace:
      3 (apply-smob/1 #<catch-closure 560b53f7b160>)
In ice-9/boot-9.scm:
  705:2 2 (call-with-prompt _ _ #<procedure default-prompt-...handle>)
In ice-9/eval.scm:
  619:8 1 (_ (#(#<directory (guile-user) 560b5400f140>)))
In /home/user/tmp/greetings.scm:
    2:0 0 (hello-scheme ("greetings.scm"))

..... greetings.scm:2:0: In procedure hello-scheme:
Wrong number of arguments to #<procedure hello-scheme ()>
```

赤字がエラーメッセージです。「引数の個数が間違ってる」と怒ってます。Backtrace 以下のメッセージは `hello-scheme` が呼び出されるまでの過程を示しています。メッセージ全体はなんだか意味不明ですが、青字のところから

```
(hello-scheme '("greetings.scm"))
```

を呼び出そうとしたことが何となく分かります。つまり、`hello-scheme` にはファイル名だけからなるリストを実引数として渡さなければならなかったのですが、それが出来なかったのでエラーが発生したのです。

3.2 スクリプト（単独のプログラム）として実行する

¶ GRM 4.3

3.2.1 スクリプトの作成と実行

Scheme のプログラムファイルの先頭に次の 2 行を追加することによって、プログラムファイルをスクリプト（単独で動作するプログラム）として実行することができます。

```
1 #!/usr/bin/guile -s
2 !#
```

ただし、スクリプトを実行するためには、プログラムファイルに実行属性を付与する必要があります。具体例を使って説明します。

■ hello.scm — サンプルプログラム —

3.15

次のプログラムを `hello.scm` というファイルに保存したとします。

```
1 ;; hello.scm
2 (define (hello-somebody arg)
3   (display (string-append "Hello," arg "!"))
4   (newline))
5
6 (define (hello-everybody args)
7   (for-each hello-somebody args))
8
9 (hello-somebody "Alice")
10 (hello-everybody '("Bob" "Carol" "Dana"))
```

`hello-somebody` は文字列 `arg` を受け取って、その文字列の前後に `"Hello,"` と `!"` を連結して表示します。 `hello-everybody` は文字列からなるリスト `args` を受け取って、リスト内の各文字列に対して `hello-somebody` を適用します。最後の 2 行はそれぞれの手続きを適当に実行しています。

■ スクリプトの作成

3.16

`hello.scm` の先頭に次の 2 行（赤字部分）を追加します。

```
1 #!/usr/bin/guile -s
2 !#
3 ;; hello.scm
4 (define (hello-somebody arg)
5   (display (string-append "Hello," arg "!"))
6   (newline))
7
8 (define (hello-everybody args)
9   (for-each hello-somebody args))
10
11 (hello-somebody "Alice")
12 (hello-everybody '("Bob" "Carol" "Dana"))
```

■ スクリプトの実行

3.17

次に、ファイルに実行属性を付けます。

```
$ chmod 755 hello.scm
      または
$ chmod +x hello.scm
```

これでスクリプトは完成です。実行してみます。

```
$ ./hello.scm
.....
..... (コンパイルに関するメッセージ) .....
.....
Hello,Alice!
Hello,Bob!
Hello,Carol!
Hello,Dana!
```

3.2.2 シェバンとブロックコメント

■ 先頭 2 行の意味

3.18

先頭に追加した第 1 行目の最初の 2 文字 **#!** は **シェバン** (shebang) と呼ばれています。そのため第 1 行目を **シェバン行** と呼んだりします。シェバン行は Linux によって処理されます。Linux は実行可能なテキストファイルの先頭の 2 文字がシェバンだったとき、そのうしろにはコマンドが記述されているものと解釈し、そのコマンドを実行します。上の例の場合、先頭行の

```
#!/usr/bin/guile
```

に対して、Linux は **guile** コマンド (**/usr/bin/guile**) を実行します。その際、コマンドのうしろに続く **-s** スイッチとプログラムファイル自体の名前 (**hello.scm**) をコマンドライン引数にして実行します。つまり、Linux は **hello.scm** のシェバン行である

```
#!/usr/bin/guile -s
```

を読み取ると、このシェバン行を

```
/usr/bin/guile -s hello.scm
```

というコマンドに変えて実行します。

前節で説明したように、このように実行された **guile** は、**hello.scm** の先頭から Scheme のプログラムとして処理していきます。先頭行 (シェバン行) は Linux によって処理されましたが、**guile** によってもう一度処理されることになります。その際、**guile** は **#!** から **!#** までを **ブロックコメント** として処理します。つまり、これらによって囲まれた部分を読み飛ばします。追加した 2 行目に **!#** があるのは、**guile** にとってのブロックコメントを閉じるためです。

■ ブロックコメントに関する注意

3.19

上のブロックコメントに関して、GRM 4.3.1 に次のような記述があります。

- The second line of the script should contain only the characters '!'#' ...

2 行目には **!#** 以外は記述しないほうがよいでしょう。

■ シェバン行があっても邪魔にはなりません

3.20

上で説明したように、**guile** は **#!** から **!#** までをブロックコメントとして読み飛ばします。従って、シェバン行を追加したスクリプトも普通のプログラムファイルとして実行できます。

```
$ guile -s hello.scm
Hello,Alice!
```

```
Hello,Bob!
Hello,Carol!
Hello,Dana!
```

さらに、REPL の中で `load` 手続きを使っても実行できます。

```
$ guile
GNU Guile 2.2.4
.....起動メッセージ.....
scheme@(guile-user)> (load "hello.scm")
Hello,Alice!
Hello,Bob!
Hello,Carol!
Hello,Dana!
$1 = (#<unspecified> #<unspecified> #<unspecified>)
scheme@(guile-user)>
```

ちなみに、`$1` の値は、`hello-everybody` 手続きの本体である `for-each` 手続きの呼び出しの返値を示しています。 `for-each` 手続きは、第 1 引数で指定された手続き (`hello-somebody`) を第 2 引数で指定されたリスト (`args`) の各要素に先頭要素から順に適用したときの副作用（上の例では各要素を表示すること）を目的としています。適用結果を求めることは目的としていません。そのため、各要素への適用結果は `#<unspecified>` にすると定められています。詳細は R⁷RS 6.10 や GRM 6.6.9.8 を参照して下さい。

3.2.3 コマンドライン引数

スクリプトにコマンドライン引数を指定して実行することもできます。コマンドライン引数は `command-line` 手続きによって取得します。例えば、`hello.scm` を次のように変更してみます。

```
1  #!/usr/bin/guile -s
2  !#
3  ;; hello.scm
4  (define (hello-somebody arg)
5    (display (string-append "Hello, " arg "!\n")))
6
7  (define (hello-everybody args)
8    (for-each hello-somebody args))
9
10 (let ((args (command-line)))
11   (write args)
12   (newline)
13   (hello-everybody (cdr args)))
```

このプログラム (`let` 以下の処理) は、`command-line` 手続きを使ってコマンドライン引数を取得し、そのコマンドライン引数を表示したあと、先頭（ファイル名）を除く残りのリストに対して `hello-everybody` を適用します。

```
$ ./hello.scm Alice Bob Carol Dana
..... コンパイルメッセージ .....
("./hello-proc.scm" "Alice" "Bob" "Carol" "Dana")
Hello, Alice!
Hello, Bob!
```

```
Hello, Carol!
Hello, Dana!
```

3.3 複数のスイッチ（特に、エントリーポイント）を指定する

¶ GRM 4.3.2

3.3.1 メタスイッチ

シェバン行のところで2つ以上のスイッチを指定するときには、次の形式を使用します。

```
1  #!/usr/bin/guile \
2  ..... スイッチ .....
3  !#
4          .....
5  ..... (Scheme のプログラム) .....
6          .....
```

例えば、エントリーポイントを指定したいときには次の形式を使用します。

```
1  #!/usr/bin/guile \
2  -e <proc> -s
3  !#
```

ここで、<proc>はエントリーポイントの手続き名を表しています。

¶ シェバン行の最後にあるバックスラッシュ（\）のことを **メタスイッチ**（meta switch）と呼びます。

■ 例

3.21

前節の hello.scm を次のように変更したとします。

```
1  #!/usr/bin/guile \
2  -e main -s
3  !#
4  ;; hello.scm
5  (define (main args)
6    (for-each hello-somebody (cdr args)))
7
8  (define (hello-somebody arg)
9    (display (string-append "Hello, " arg "!\n")))
```

このスクリプトは main 手続きをエントリーポイントにしています。main 手続きは、コマンドライン引数（文字列からなるリスト）を実引数として受け取って、先頭の文字列（ファイル名）を除く残りの文字列に hello-somebody を適用します。

```
$ ./hello.scm Alice Bob Carol Dana
.....
..... (コンパイルに関するメッセージ) .....
.....
```

```
Hello, Alice!
Hello, Bob!
Hello, Carol!
Hello, Dana!
```

3.3.2 2行目の書き方はとても注意が必要です

2行目の書き方には次のような掟があります (GRM 4.3.2)。これらを破るとエラーが発生します。

- 行の先頭に空白やタブ文字を入れてはいけません。
- スイッチの各構成要素 (上の例では ‘-e’, ‘main’, ‘-s’ のこと) は空白 1 文字 (きっかり 1 文字) で区切らなければなりません。
- 最後の構成要素 (上の例では ‘-s’ のこと) の直後で改行しなければなりません。(空白 1 文字入れて改行しても大丈夫ですが、2 文字入れるとエラーが発生します。)

GRM 4.3.2 によると、2 行目に指定したスイッチや引数の区切りは空白 1 文字 (きっかり 1 文字) にしているのだそうです。そのため、先頭に空白があったり空白 2 文字があったりすると、空の文字列 "" (以下、「空文字列」) が引数として指定されたと判断し、実行時のコマンドライン引数に加えるそうです。先頭に空白があるときには、その空白の直前に空文字列が指定されていると判断し、空白 2 文字があるときには、それらの空白の間に空文字列が指定されていると判断するようです。guile コマンドを実行するときに、この空文字列がエラーを起こします。

🔔 以上の処理は (Linux ではなく) Guile 自体が行っているのですが、なんでそんなことになっているのか理由は不明です。空白を読み飛ばせばいいだけじゃないか、手抜きじゃないかと疑いたくなりますが、何か大人の事情があるのでしょうか。いずれにしても、とても分かりにくいエラーなので注意が必要です。

■ 例

3.22

具体例を使って確認してみましょう。2 行目の先頭に空白があった場合を試してみます。以下の 2 行目の先頭にある `_` は空白があることを表しています。ファイル名を `secondline.scm` とします。

```
1  #!/usr/bin/guile \
2  _ -e main -s
3  !#
4  ;; secondline.scm
5  (define (main args)
6    (write args) (newline))
```

これを実行すると次のようなエラーが発生します。

```
$ ./secondline.scm
;;; note: auto-compilation is enabled, set GUILE_AUTO_COMPILE=0
;;;      or pass the --no-auto-compile argument to disable.
;;; compiling /home/user/tmp/
;;; WARNING: compilation of /home/user/tmp/ failed:
;;; In procedure fport_read: ディレクトリです
Backtrace:
      0 (primitive-load "/home/user/tmp/")

ERROR: In procedure primitive-load:
```

In procedure fport_read: ディレクトリです

最後から2行目にエラーメッセージがありますが、そのエラーの正体は赤字が示しています。guile コマンドは空文字列 "" がスクリプトファイル名を表していると解釈していて、そのファイルをロードしようとしています。もう少し詳しく言うと、次のコマンドと同じ処理をしようとしています。

```
$ guile "" -e main -s secondline.scm
```

ファイル名が tmp ディレクトリになっているのは、./secondline.scm を tmp ディレクトリ上で実行しているためです。つまり、カレントディレクトリである tmp ディレクトリ上の "" という名前のファイル（名無しのファイル）をロードしようとして失敗しています。

■ 例

3.23

もう1つ試してみましょう。今度は、-e スイッチのうしろに空白が2文字あります。

```
1 #!/usr/bin/guile \
2 -e  _ main -s
3 !#
4 ;; secondline.scm
5 (define (main args)
6   (write args) (newline))
```

これを実行すると次のようなエラーが発生します。

```
$ ./secondline.scm
;;; Stat of /home/user/tmp/main failed:
;;; In procedure stat: ..... (省略) .....
Backtrace:
      0 (primitive-load "/home/user/tmp/main")

ERROR: In procedure primitive-load:
In procedure open-file: ..... (省略) .....
```

この場合、次のコマンドと同じ処理をしようとしています。

```
$ guile -e "" main -s secondline.scm
```

つまり、-e スイッチの直後にある空文字列 "" をエン트리ポイントと見なし、main を tmp ディレクトリ上のスクリプトファイルの名前と見なしています。そのファイルをロードしようとして失敗しています。

■ 例

3.24

くどいようですが、もっとも醜惡な具体例を試しておきましょう。この例では、-s スイッチのあとに空白が2文字あります。

```
1 #!/usr/bin/guile \
2 -e main -s  _
3 !#
4 ;; secondline.scm
5 (define (main args)
6   (write args) (newline))
```

これを実行すると次のようなエラーが発生します。

```
$ ./secondline.scm
;;; note: auto-compilation is enabled, set GUILE_AUTO_COMPILE=0
```

```

;;;      or pass the --no-auto-compile argument to disable.
;;; compiling /home/user/tmp/
;;; WARNING: compilation of /home/user/tmp/ failed:
;;; In procedure fport_read: ディレクトリです
Backtrace:
      0 (primitive-load "/home/user/tmp/")

ERROR: In procedure primitive-load:
In procedure fport_read: ディレクトリです

```

この場合、次のコマンドと同じ処理をしようとしています。

```
$ guile -e main -s "" secondline.scm
```

1 番目の具体例と同様に、カレントディレクトリである `tmp` ディレクトリ上の `""` という名前のファイル（名無しのファイル）をロードしようとして失敗しています。

🔴 `-s` スイッチのうしろに空白 2 文字がある場合は気が付きにくいので、特に注意が必要だと思います。筆者は何回も引っかかっています。

🔴 上の例で示した空文字列 `""` や `"main"` は `-e` スイッチの引数ではなく、しかも `-s` スイッチよりも前に指定されているか、`-s` スイッチの直後に指定されています。guile は、このような文字列をスクリプトファイル名と見なすようです。これは、スクリプトファイルを指定するとき `-s` スイッチを省略してもよいことに起因しています。さらに、スクリプトファイル名と見なした文字列のうしろにある文字列はコマンドライン引数と見なすようです。例えば、次のスクリプト（明らかに書き方がオカシイ）はエラーが起りません。

```

1 #!/usr/bin/guile \
2 -e main secondline.scm -s
3 !#
4 ;; secondline.scm
5 (define (main args)
6   (write args) (newline))

```

```

$ ./secondline.scm
.....
..... (コンパイルに関するメッセージ) .....
.....
("secondline.scm" "-s" "./secondline.scm")

```

この場合、次のコマンドと同じ処理をしようとしています。

```
$ guile -e main secondline.scm -s ./secondline.scm
```

guile は、1 番目の `secondline.scm` をスクリプトファイル名と見なし、そのうしろに続く `-s` と `./secondline.scm` をコマンドライン引数と見なしています。secondline.scm は実在するスクリプトファイルなのでエラーが起きないのです。

3.3.3 Why メタスイッチ

■ 論理的な推測が通用するわけではありません

3.25

前節のシェバン行の説明から推測して、スイッチが複数あった場合もシェバン行を次のように記述すればよさそうな感じがします。

```

1  #!/usr/bin/guile -e main -s
2  !#
3  (define (main args)
4    (for-each hello-somebody (cdr args)))
5
6  (define (hello-somebody arg)
7    (display (string-append "Hello, " arg "!\\n")))

```

例えば、先ほどと同様に

```
$ ./hello.scm Alice Bob Carol Dana
```

を実行すると、前節の説明から推測して

```
$ /usr/bin/guile -e main -s hello.scm Alice Bob Carol Dana
```

というコマンドが実行されるだろうと期待します。でも、次のように「認識不可能なスイッチ（スイッチ不明）」といったエラーが発生します。このエラーが発生する理由を GRM 4.3.2 は詳しく説明しています。

```

$ ./hello.scm Alice Bob Carol Dana
error: unrecognized switch -e main -s
.....
..... (Guile の使い方に関するメッセージ) .....
.....

```

このエラーは Linux が準拠している POSIX に由来しています。残念なことに、POSIX はシェバン行に指定できるスイッチを一つしか許していないそうです。より正確に言うならば、スイッチとして指定された記号列全体を単一のスイッチとして処理するそうです。例えば、

```
#!/usr/bin/guile -e main -s
```

というシェバン行の場合、`-e main -s` を 3 つの要素からなるスイッチではなくて、単一のスイッチとして処理します。そこで、POSIX に準拠する Linux は、

```
/usr/bin/guile '-e main -s' hello.scm ...
```

というコマンドを生成して実行します³⁾。そこで次に、guile コマンドが `-e main -s` を単一のスイッチとして処理しようとしします。でも、そんな奇妙なスイッチはありません。そのため、スイッチ不明といったエラーを報告してコマンドを終了します。上のエラーメッセージは guile コマンドが表示したものです。そのあとに続く Guile の使い方に関するメッセージは `guile --help` を実行したときのものと同じです。

オイラは悪くないんだよ、POSIX がヘンなんだ、だからメタスイッチを用意したんだよ ... ということのように。世間はなかなかすっきりいきません。

■ メタスイッチを指定したときの処理

3.26

メタスイッチ (`\`) は POSIX による制約を克服するためのものです。メタスイッチに関わる処理を説明するため、再び次のスクリプト (`hello.scm`) を考えてみます。

```

1  #!/usr/bin/guile \
2  -e main -s
3  !#
4  ;; hello.scm
5  (define (main args)
6    (for-each hello-somebody (cdr args)))

```

³⁾`-e main -s` という書き方は、あくまで単一のスイッチとして処理されることを強調しているだけであって、実際の書き方を示しているものではありません。


```

7 |
8 | (define (hello-somebody arg)
9 |   (display (string-append "Hello, " arg "!\n")))

```

前と同様に、このスクリプトを次のように実行したとします。

```
$ ./hello.scm Alice Bob Carol Dana
```

まず POSIX 準拠の Linux はシェバン行を次のようなコマンドに変えて実行します。

```
$ /usr/bin/guile \ hello.scm Alice Bob Carol Dana
```

これによって起動された guile は、メタスイッチ (\) を 2 行目にある 3 つの要素 ‘-e’, ‘main’, ‘-s’ に置き換えて処理を進めます。つまり、次のコマンドを実行した場合と同じ処理を行います。

```
$ /usr/bin/guile -e main -s hello.scm Alice Bob Carol Dana
```

これによって main をエントリーポイントとしてスクリプトが実行されるのです。

先に述べた 2 行目に関する制限は、guile によるメタスイッチの処理に由来しています。やっぱり手抜きじゃないでしょうか ... いやいや大人の事情があるのです、たぶん。

3.4 その他

3.4.1 他のスイッチ

¶ GRM 4.2.1

■ -l スイッチ, -ds スイッチ

3.27

-l スイッチ を使うと複数のソースファイルを 1 つにまとめて実行できます。-l スイッチのあとにロードしたいファイルを指定します。

- 以下は prg-1.scm, prg-2.scm, prg-3.scm の順にプログラムをロードします。

```
$ guile -l prg-1.scm -l prg-2.scm -s prg-3.scm
```

-l スイッチは幾つでも指定できます。

- -s スイッチは (POSIX の規定により) 最後に指定しなければいけません。そのため、-s で指定したファイルを途中で差し込みたいときには **-ds スイッチ** を使います。

```
$ guile -l prg-1.scm -ds -l prg-2.scm -s prg-3.scm
```

これは prg-1.scm, prg-3.scm, prg-2.scm の順にプログラムをロードします。

- エントリーポイントを指定することもできます。

```
$ guile -e main -l prg-1.scm -l prg-2.scm -s prg-3.scm
```

これは prg-1.scm, prg-2.scm, prg-3.scm の順にプログラムをロードして、ロードが完了したらエントリーポイントの手続きを実行します。ただし、エントリーポイントの手続きの定義が重複しているときには最後にロードされた定義が有効です。

- スクリプトの先頭にも上のスイッチを指定できます。例えば、prg-3.scm の先頭部分を次のように指定したとします。

```

1  #!/usr/bin/guile \
2  -e main -l prg-1.scm -l prg-2.scm -s
3  !#
4      ..... プログラム .....

```

この prg-3.scm を実行したとき、つまり

```
$ ./prg-3.scm
```

を実行したとき、それは

```
$ guile -e main -l prg-1.scm -l prg-2.scm -s prg-3.scm
```

を実行したのと同じになります。

📌 規模の大きな Guile スクリプトを複数のファイルに分割して構成するときには、-l スイッチや-ds スイッチよりもモジュールを使うべきなのだろうと思います。

3.4.2 ソースファイルの文字コード

ソースファイルの文字コードを UTF-8 にしてよいのかどうかは気になることです。

📌 GRM 4.3.1 に次の記述があります。

- If this source code file is not ASCII or ISO-8859-1 encoded, a coding declaration such as `coding: utf-8` should appear in a comment somewhere in the first five lines of the file: see Section 6.18.8 [Character Encoding of Source Files], page 395.

GRM 6.18.8 の出だしは次の通りです。

- Scheme source code files are usually encoded in ASCII or UTF-8, but the built-in reader can interpret other character encodings as well. When Guile loads Scheme source code, it uses the `file-encoding` procedure (described below) to try to guess the encoding of the file. **In the absence of any hints, UTF-8 is assumed.** One way to provide a hint about the encoding of a source file is to place a coding declaration in the top 500 characters of the file.

GRM 4.3.1 が言ってることと食い違ってますが、**こちらを信じることにします**。

📌 用心のために文字コードを指定するときには、その指定方法は次の通りです。

- A coding declaration has the form `coding: XXXXXX`, where XXXXXX is the name of a character encoding in which the source code file has been encoded. The coding declaration must appear in a scheme comment. It can either be a semicolon-initiated comment, or the first block `#!` comment in the file.

具体的には、シェバン行を指定するのであれば、Guile ブロックコメントの中に次のように指定すればよいと思います。

```

1  #!/usr/bin/guile -s
2  coding: utf-8
3  !#
4      ..... ソースコード .....

```

```

1 #!/usr/bin/guile \
2 -e main -s
3 coding: utf-8
4 !#
5     ..... ソースコード .....

```

シェバン行を指定しないのであれば、ファイルの冒頭で次のように指定すればよいのだと思います。

```

1 ;; coding: utf-8
2     ..... ソースコード .....

```

でも、GRM 6.18.8 を信じるのであれば、UTF-8 のソースファイルを素直に処理するはずです。例えば、次のプログラム（ファイル名は `utf-8.scm`）は問題なく動作しています。

```

1 #!/usr/bin/guile \
2 -e main -s
3 !#
4 (define (main args)
5   (let ((x "文字コードは UTF-8 だけど、 ")
6         (y "問題なく動くはずです. "))
7     (display (string-append x y)) (newline) ))

```

```

$ ./utf-8.scm
      ..... コンパイルメッセージ .....
文字コードは UTF-8 だけど、問題なく動くはずです.

```

第4章 プログラムの全体像

4.1 R⁷RS

R⁷RS 5.1 は、プログラムの全体像を次のように述べています。

- Scheme のプログラムは、1 つ以上の **インポート宣言** に続けて、**式** (`<expression>`) または **定義** (`<definition>`) の列からなる

従って、Scheme のプログラムは模式的に次のような形式になります。

```
1  <import_declaration>
2  .....
3  <import_declaration>
4
5  <expression> または <definition>
6  .....
7  <expression> または <definition>
```

インポート宣言、式、定義は先頭から順に逐次的に実行されます。インポート宣言や定義はグローバルな環境に対して新たな束縛を生成したり、既存の束縛を変更したりします。一方、プログラムのもっとも外側に書かれた式は、(グローバルな環境に対して) 既存の束縛を変更することはあっても、新たな束縛を生成することはありません。プログラムの初期環境は空です。そのため、少なくとも 1 つのインポート宣言が必要です。

¶ R⁷RS 7.1.6 の構文規則を見ると、上の `<expression>` の代わりに `<command>` が使われています。でも、`<command>` は `<expression>` だけなので (R⁷RS 7.1.3)、上の模式図と同じものになります。

¶ R⁶RS 8.1 も、上と同じように、インポート宣言のあとに定義や式を並べる形式を示しています。

4.2 Guile

■ インポート宣言が見当たらないんですけど ...

4.28

R⁷RS 5.1 は「少なくとも 1 つのインポート宣言が必要である」と述べています。でも、Guile Reference Manual のサンプルコードを見ても、インポート宣言は見当たりません。その代わりに、しばしば `use-modules` を使ってモジュールをロードしています。さらに加えて言うと、モジュールの話はマニュアル全体を通して頻繁に出てきます。しかし、インポート宣言に関する説明は GRM 6.20.6 以外に見つけることができません。そこでは R⁶RS で定義されているライブラリー形式が Guile でもオプション的な選択肢として利用できることを述べるだけにとどめています¹⁾。さらに加えて言うと、ネット上に散見されるプログラムを見てもモジュールをロードしたあとに定義と式の列が続くという構成になっています。

以上から推測して、Guile では、**インポート宣言の代わりにモジュールをロードする**ということだと思います。

■ Guile プログラムの構成

4.29

以上のような事情をもとに、Guile のプログラムは次の形式をしていると考えてよいのだと思います。

¹⁾Guile は R⁶RS を基準にしているようなので、仕様書の話をするときのバージョンは R⁶RS になっています。

```

1  #!/usr/bin/guile \
2  -e <proc> -s
3  !#
4
5  <use_modules>
6  .....
7  <use_modules>
8
9  (define (<proc> args)
10     .....
11     ..... メインの処理 .....
12     ..... )
13
14 <expression> または <definition>
15 .....
16 <expression> または <definition>

```

¶ <proc>はエントリーポイントの手続き名を表しています。

■ プログラムの動作

4.30

上のプログラムは次のように変更したプログラムと等価です。赤字が変更したところです。エントリーポイント（<proc>手続き）の指定を外し、その代わりに<proc>手続きの呼び出しをプログラムの最後に行っています。

```

1  #!/usr/bin/guile -s
2  !#
3
4  <use_modules>
5  .....
6  <use_modules>
7
8  (define (<proc> args)
9     .....
10     ..... メインの処理 .....
11     ..... )
12
13 <expression> または <definition>
14 .....
15 <expression> または <definition>
16
17 (<proc> (command-line))

```

つまり、モジュールのロードや式や定義の評価を先頭から順に行ったあとに、エントリーポイントの手続き（<proc>）を、コマンドライン引数を実引数として**最後に実行します**。この点は、GRM 4.2.1の-e オプションの説明文中にあります。なお、以下の *function* は手続きのことです。

- -e *function*

Make *function* the *entry point* of the script. 省略

A -e switch can appear anywhere in the argument list, but **Guile always invokes the *function* as the last action it performs.** 省略

■ 例

4.31

上で述べたことを確認してみます。例えば、次のプログラムは正常に動作するでしょうか？

```

1  #!/usr/bin/guile \
2  -e main -s
3  !#
4
5  (define (main args)
6    (display x) (newline))
7
8  (define x "I am X!")

```

エントリーポイントの `main` から評価が始まると素朴に考えたとき、つまり、`main` 手続きから実行が開始されると考えたとき、`main` の本体を実行した時点で `x` は未束縛なのでエラーが出るはずです。でも、これは問題なく動きます。`x` の定義が終わったあとに `main` が呼び出されるからです。ファイル名を `prg.scm` として実行してみます。

```

$ ./prg.scm
I am X!

```

さらに、上のプログラムを次のように変更すると、エントリーポイントの実行が「the last action」であることがよく分かります。

```

1  #!/usr/bin/guile \
2  -e main -s
3  !#
4
5  (define (main args)
6    (display x) (newline) )
7
8  (define x "I am X!")
9
10 (display " *** THIS IS THE LAST LINE OF THIS PROGRAM. ***\n")

```

```

$ ./prg.scm
*** THIS IS THE LAST LINE OF THIS PROGRAM. ***
I am X!

```

この実行結果は、最終行の `display` 手続きが実行されたあと、エントリーポイント (`main`) が呼び出されていることを示しています。

■ 例

4.32

モジュールを利用した例を示します。

```

1  #!/usr/bin/guile \
2  -e main -s
3  !#
4
5  (use-modules (srfi srfi-1)
6              (srfi srfi-11))
7
8  (define (main args)
9    (let ((lst (cdr args)))

```

```

10 (display "Before sorting: ")
11 (display lst)
12 (newline)
13 (display "After  sorting: ")
14 (display (qsort (map string->number lst)))
15 (newline))
16
17 (define (qsort lst)
18   (if (null? lst)
19       lst
20       (let ((pivot (car lst))
21             (lst (cdr lst)))
22         (let-values
23           (((left right) (partition (lambda (x) (< x pivot)) lst)))
24           (append (qsort left) (cons pivot (qsort right)))))))

```

このプログラムは、コマンドライン引数に指定された数値列をクイックソートを使って小さい順にソートします。ファイル名を `qsort.scm` とします。

```

$ ./qsort.scm 9 3 8 4 6 7 5 2 1
..... コンパイルメッセージ .....
Before sorting: (9 3 8 4 6 7 5 2 1)
After  sorting: (1 2 3 4 5 6 7 8 9)

```

上のプログラムは、`(srfi srfi-1)` と `(srfi sfri-11)` の2つのモジュールをロードしています。ちなみに、Guile のモジュールは、シンボルのリストによって表されます。

`(srfi srfi-1)` はリストを処理するための色々と便利な手続きを提供するモジュールです。上のプログラムでは `partition` 手続きを使うためにロードしています。 `partition` 手続きは、

```
(partition pred lst)
```

といったように、述語 `pred` とリスト `lst` を受け取って、`lst` の要素を `pred` を真とするものと偽とするものの2つのリストに分割して、それら2つのリストを多値として返します。上のプログラムでは、`partition` を使ってクイックソートの分割処理を実現しています。

`(srfi sfri-11)` は `let-values` 形式を使うため(だけ)のモジュールです。 `partition` 手続きは2つのリストを多値として返してくるので、それを受け取るために `let-values` を使っています。

`partition` と `let-values` と自前の `main` と `qsort` 以外の手続きや構文形式は、Guile の **コア言語** (core language) が提供しています。例えば、`string->number` (数詞を数値に変換する手続き) や `map` (第1引数の手続きを第2引数のリストの各要素に適用する手続き) は、いずれも特殊な機能を果たしているように感じるのですが、Guile のコア言語が提供しています。そのため、これらについては特定のモジュールをロードする必要はありません。

📌 モジュールを表すシンボルのリストは、モジュールファイルを検索するときの基点となるディレクトリ上のサブディレクトリ名とファイル名を表しています。例えば、モジュール名の `(srfi srfi-1)` は、基点となるディレクトリ (`/usr/share/guile/2.2`) 上の `srfi` というサブディレクトリにある `srfi-1.scm` (または `srfi-1.go`) というファイルを示しています。

ここで、`srfi-1.go` はコンパイル済みのモジュールファイルです。Debian の公式版の Guile では、コンパイル済みのモジュールファイルはなく、すべてソースファイルのままです。

付 録 A

A.1 Scheme や Guile の情報源

- R7RS (原版), <https://small.r7rs.org/attachment/r7rs.pdf>
R7RS (日本語版), <http://milkpot.sakura.ne.jp/scheme/r7rs.pdf>
- R6RS (原版) <http://www.r6rs.org/>
R6RS (日本語版) 下記の「プログラミング言語 Scheme の解説」を参照して下さい.
- R5RS (原版), <https://schemers.org/Documents/Standards/R5RS/r5rs.pdf>
R5RS (日本語版), 下記の「プログラミング言語 Scheme の解説」を参照して下さい.
- プログラミング言語 Scheme の解説
<http://www7b.biglobe.ne.jp/~saia/scheme.html>
- Guile 公式サイト, <https://www.gnu.org/software/guile/>
GNU Guile 3.0.4 Reference Manual (略号 GRM),
<https://www.gnu.org/software/guile/manual/>
GNU Guile 2.2.6 Reference Manual (略号 GRM),
<https://www.gnu.org/software/guile/docs/docs-2.2/guile-ref/>
筆者がこれまで読んでいる限りでは (つまり, 筆者程度のレベルでは), 3.0.4 と 2.2.6 は違いがありません. そのため, 略号は 1 つにしています.
- The Adventures of a Pythonista in Schemeland
<http://www.phyast.pitt.edu/~micheles/scheme/>
- Going from Python to Guile Scheme
<http://www.draketo.de/proj/py2guile/> Basic questions and answers for GNU Guile
(from the perspective of a Pythonista)
<https://www.draketo.de/proj/guile%2Dbasics/>
- Gauche ユーザリファレンス (略号 Gauche)
<https://practical-scheme.net/gauche/man/gauche-refj/index.html>
オンラインマニュアルが充実しています. 何より日本語です.
- Racket Documentation (略号 Racket)
<https://docs.racket-lang.org/>
オンラインマニュアルが充実しています.
- Schemers.org: an improper list of Scheme resources
<https://schemers.org/>
- R. Kent Dybvig: The Scheme Programming Language Fourth Edition, 2009. (略号 TSPL4)
<https://www.scheme.com/tspl4/>
- Dorai Sitaram: Teach Yourself Scheme in Fixnum Days
<https://ds26gte.github.io/tyscheme/>
(日本語版) Nobuo Yamashita: 独習 Scheme 三週間
<https://www.sampou.org/scheme/t-y-scheme/t-y-scheme.html>

- Michael Gran (edited): The Guile 100 Programs Project
<http://www.lonelycactus.com/guile100/>
- Shriram Krishnamurthi: Programming Languages: Application, 2007-04-26.
<http://cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/>
- Hal Abelson, Jerry Sussman and Julie Sussman: Structure and Interpretation of Computer Programs, Second Edition, 1993. (略号 SICP)
<https://mitpress.mit.edu/sites/default/files/sicp/index.html>
(日本語版・和田英一訳) <https://sicp.iiijlab.net/fulltext/>
(日本語版・真鍋宏史訳) <http://vocrf.net/docs-ja/jsicp.pdf>
(同上) <https://github.com/hiroshi-manabe/sicp-pdf/blob/japanese/jsicp.pdf>
- Matthias Felleisen, Robert Bruce Findler, Matthew Flatt and Shriram Krishnamurthi: How to Design Programs — An Introduction to Computing and Programming (Second Edition) <https://htdp.org/>
(First Edition) <https://htdp.org/2003-09-26/Book/>
- POSIX パス名
https://pubs.opengroup.org/onlinepubs/009604499/basedefs/xbd_chap04.html#tag_04_11

A.2 REPL コマンド

REPL はたくさんのコマンドを用意しています (GRM 4.4.4)。そのうちの役立ちそうなものを説明します。ただし、筆者自身も慣れていないので、どのように役立つかは不明です。

■ サンプルプログラム

A.33

実行例を示すときには次の 2 つの手続きを使います。いずれもフィボナッチ数を求めるための手続きです。fib 手続きはフィボナッチ数の漸化式を素直に計算します。fib-iter は末尾再帰（繰り返し）を使って添字の小さいほうから計算します。ファイル名は fib.scm とします。

```

1 ;; fib.scm
2 (define (fib n)
3   (if (<= n 1)
4       n
5       (+ (fib (- n 1)) (fib (- n 2)))))
6
7 (define (fib-iter n)
8   (let fib-tail ((k 1) (current 1) (previous 0))
9     (if (>= k n)
10        current
11        (fib-tail (+ k 1) (+ current previous) current))))

```

REPL を起動し、上のプログラムをロードしている状態で具体例を示します。

```

$ guile
GNU Guile 2.2.4
..... 起動メッセージ .....
scheme@(guile-user)> (load "fib.scm")
..... 起動メッセージ .....
scheme@(guile-user)>

```

■ コマンド

A.34

各項目名の左側はコマンド名を示し、右側は省略名（または別名）を示します。

- `,help` `,h` コマンドのヘルプを表示します。
- `,import` `,use` 現在のセッションにロードされているモジュールを一覧表示します。引数にモジュール名を指定すると (`,import <module>` `,use <module>`)、そのモジュールを現在のセッションにロードできます。
- `,binding` `,b` 現在の束縛を一覧表示します。10 ページ参照。
- `,time <exp>` `,t <exp>` `<exp>` は式です。その式を評価したときの実行時間を表示します。

```
scheme@(guile-user)> ,time (fib 40)
$1 = 102334155
;; 6.840974s real time, 6.840933s run time. 0.000000s spent in GC.
scheme@(guile-user)> ,time (fib-iter 40)
$2 = 102334155
;; 0.006476s real time, 0.006461s run time. 0.000000s spent in GC.
scheme@(guile-user)>
```

多重再帰と末尾再帰では計算時間が劇的に違うことが分かります。real time は経過時間、run time は CPU の計算時間を示します。これらは Linux の time コマンドを実行したときに表示される時間と同じです。ただし、run time は system time と user time を合わせたものです。時間に関する詳細は Linux の time コマンド関連の説明を参照して下さい。

- `,profile <exp>` `,pr <exp>` `<exp>` を実行したときのプロファイルを表示します。

```
scheme@(guile-user)> ,profile (fib 30)
%      cumulative      self
time  seconds      seconds  procedure
100.00      1.15      0.06      ..... fib.scm:9:0:fib
---
Sample count: 3
Total time: 0.056528541 seconds (0.0 seconds in GC)
```

これによって表示されるすべての時間は実行時間であり、経過時間ではありません。

self 欄は手続き自身が消費した実行時間を示します。Guile のマニュアルは何も述べていませんが、おそらく手続き呼び出し 1 回当たりの時間だと思います。

%time 欄は、それぞれの手続きによって消費された時間比率を示します。この時間にはそこから呼び出した手続きの時間は含みません。これは self 時間を total 時間で割った値です。

cumulative 欄は self 欄の累計です。ただし、再帰的な手続きについては、すべての呼び出しの計算時間が累計されます。従って、再帰的な手続きについては、self（呼び出し 1 回当たりの時間）よりもずっと大きな時間になります。

Total time 欄は self 欄の総計だと思います。procedure 欄は手続きの名前を示し、GC はガベージコレクタが消費した時間を示します。

実行時間が短すぎるとプロファイルは表示されません。

```
scheme@(guile-user)> ,profile (fib-iter 30)
No samples recorded.
```

```

scheme@(guile-user)> ,profile (fib-iter 100000)
%      cumulative      self
time   seconds      seconds  procedure
54.55    0.48        0.26    ..... fib.scm:14:0:fib-iter
45.45    0.22        0.22    %after-gc-thunk
0.00     0.22        0.00    anon #x7f37e8981100
---
Sample count: 22
Total time: 0.475632156 seconds (0.302463575 seconds in GC)

```

プロファイルの各欄について GRM 7.20 に説明があります。でも、あまり詳しくはありません。プロファイルの詳細は `gnu profiler` などの説明を参照して下さい。概念的にはほぼ同じだと思います。

- `,trace <exp>` ,`tr <exp>` 手続き呼び出しの様子を模式図的に表示します。以下は `(fib 5)` と `(fib-iter 5)` を実行したときの手続き呼び出しの様子を示しています。名前付き `let` で定義した手続き (`fib-tail`) の呼び出しは表示されないようです。そのため、`fib-iter` 手続きのトレースは `fib-iter` 自身だけになっています。

```

scheme@(guile-user)> ,trace (fib 5)
trace: (fib 5)
trace: | (fib 4)
trace: | | (fib 3)
trace: | | | (fib 2)
trace: | | | | (fib 1)
trace: | | | | 1
trace: | | | | (fib 0)
trace: | | | | 0
trace: | | | 1
trace: | | | (fib 1)
trace: | | | 1
trace: | | 2
trace: | | (fib 2)
trace: | | | (fib 1)
trace: | | | 1
trace: | | | (fib 0)
trace: | | | 0
trace: | | 1
trace: | 3
trace: | (fib 3)
trace: | | (fib 2)
trace: | | | (fib 1)
trace: | | | 1
trace: | | | (fib 0)
trace: | | | 0
trace: | | 1
trace: | | (fib 1)
trace: | | 1
trace: | 2
trace: 5
scheme@(guile-user)> ,trace (fib-iter 5)
trace: (fib-iter 5)
trace: 5

```

```
scheme@(guile-user)>
```

`fib-tail` の定義を `define` 形式を使った内部定義に変更してみてもトレースは表示されませんでした。内部定義の手続きはトレースしないようです。

`fib-iter` 手続きのトレースがとても味気ないので、次のような `fib-iter2` 手続きを実行してみます。

```
1 (define (fib-iter2 n)
2   (fib-tail2 n 1 1 0))
3
4 (define (fib-tail2 n k current previous)
5   (if (>= k n)
6       current
7       (fib-tail2 n (+ k 1) (+ current previous) current)))
```

```
scheme@(guile-user)> ,trace (fib-iter2 5)
trace: (fib-iter2 5)
trace: (fib-tail2 5 1 1 0)
trace: (fib-tail2 5 2 1 1)
trace: (fib-tail2 5 3 2 1)
trace: (fib-tail2 5 4 3 2)
trace: (fib-tail2 5 5 5 3)
trace: 5
scheme@(guile-user)>
```

`fib-tail2` は末尾再帰を利用しているので、再帰呼び出しがいつまで行われずに手続き呼び出しが繰り返されているだけなのが分かります。

他にもデバッグ用のコマンドなどがあるのですが、利用方法がよく分からないので割愛します。

A.3 旧版のマニュアルを入手する

Guile の公式ページは、

- 最新版のマニュアルを各種形式で配布していて、
- 旧版については、マイナーバージョンが最新のマニュアルだけが web page として閲覧できる

といった状況です。例えば、Guile 2.2 については、2.2.6 のマニュアルだけが web page として閲覧できて、2.2.4 のマニュアルはありません。そこで、Guile のソースコードに含まれるドキュメントからマニュアルの PDF ファイルを作成する方法を説明します。

¶ Debian 用の Guile パッケージ (`guile-2.2`) をインストールすれば、Textinfo を使ったマニュアルが利用できます。残念ながら、筆者は Texinfo に慣れていないので、PDF ファイルを作成して利用しています。

A.3.1 一般的な方法 — Gnu のアーカイブを利用する

■ ソースコードを入手して署名を確認する

Guile のソースコードを Gnu のミラーサイト

• <http://ftp.jaist.ac.jp/pub/GNU/guile/>
から入手します。Guile の旧版から最新版までのソースコードがありますが、この説明では Guile 2.2.4 の

- ソースコード `guile-2.2.4.tar.gz` と
- 署名ファイル `guile-2.2.3.tar.gz.sig`

を適当な作業用ディレクトリにダウンロードします。

```
$ wget http://ftp.jaist.ac.jp/pub/GNU/guile/guile-2.2.4.tar.gz
$ wget http://ftp.jaist.ac.jp/pub/GNU/guile/guile-2.2.4.tar.gz.sig
```

次にソースコード (tar ボール) の署名確認をします。

```
$ gpg --verify guile-2.2.4.tar.gz.sig
gpg: 署名されたデータが'guile-2.2.4.tar.gz'にあると想定します
gpg: 2018 年 07 月 02 日 17 時 01 分 51 秒 JST に施された署名
gpg: RSA 鍵 464EC3558A84FDC69DB40CFBB090D319199AEBB5 を使用
gpg: 署名を検査できません: No public key
```

tar ボール作成者の公開鍵がインポートされていないので、検証できずに失敗しています。そこで、作成者の公開鍵をインポートします。ちなみに、上の作業は作成者の公開鍵を知るために必要です。

```
$ gpg --keyserver keys.gnupg.net --recv-keys 464EC3558A84FDC69DB40CFBB090D319199AEBB5
gpg: 鍵 B090D319199AEBB5: "Breen Glue <breen@gnu.org>" 新しい署名を 15 個
gpg: 処理数の合計: 1
gpg: 新しい署名: 15
```

一応、インポートが成功したことを確認します。

```
$ gpg --list-keys
/home/algo/.gnupg/pubring.kbx
-----
pub   rsa4096 2014-08-11 [SC] [期限切れ: 2020-12-21]
      464EC3558A84FDC69DB40CFBB090D319199AEBB5
uid           [期限切れ] Breen Glue <breen@gnu.org>
uid           [期限切れ] Breen Glue <breen@hoge.org>
uid           [期限切れ] Breen Glue (Nippon) <breen.glue@nippon.com>
```

再び、ソースコード (tar ボール) の署名確認をします。

```
$ gpg --verify guile-2.2.4.tar.gz.sig
gpg: 署名されたデータが'guile-2.2.4.tar.gz'にあると想定します
gpg: 2018 年 07 月 02 日 17 時 01 分 51 秒 JST に施された署名
gpg: RSA 鍵 464EC3558A84FDC69DB40CFBB090D319199AEBB5 を使用
gpg: "Breen Glue <breen@gnu.org>"からの正しい署名 [期限切れ]
gpg: 別名"Breen Glue <breen@hoge.org>" [期限切れ]
gpg: 別名"Breen Glue (Nippon) <breen.glue@nippon.com>" [期限切れ]
gpg: 注意: この鍵は期限切れです!
```

成功なのか失敗なのか、なんだかよく分からないのですが、ネット上の幾つかのサイトを見ると

- gpg: "Breen Glue <breen@gnu.org>"からの**正しい署名** [期限切れ]

と表示されれば成功と判断してよいようです。

🔑 個人的な情報を記述するのは気が引けるので、上の実行例では、個人名、Email アドレス、RSA 鍵はデタラメなものに変更しています。

まず tar ボールを展開します。

```
$ tar -zxvf guile-2.2.4.tar.gz
```

ドキュメントファイル (*.texi) が格納されているディレクトリに移動します。

```
$ cd guile-2.2.4/doc/ref
```

マニュアルの PDF ファイルを作成します。

```
$ texi2pdf guile.texi
```

以上の作業によって guile.pdf というファイルが作成されます。適当な PDF ビューアを使って内容を確認して下さい。

A.3.2 Debian な方法 — ソースパッケージを利用する

Debian のソースパッケージを利用することもできます。下記のコマンドを適当な作業ディレクトリで実行して下さい。

```
(1) 必要なツールをインストールする
$ sudo apt install build-essential
$ sudo apt install flex
$ sudo apt install devscripts
(2) ソースパッケージをインストールする
$ apt source guile-2.2
(3) ソースコードのディレクトリに移動して configure する。
$ cd guile-2.2-2.2.4+1/
$ ./autogen.sh
$ ./configure
(4) ドキュメントディレクトリに移動して PDF ファイルを生成する。
$ cd doc/ref/
$ make      (注) 下記参照
$ make      (注) 下記参照
$ texi2pdf guile.texi
    または
    make guile.pdf
```

❗ make は vesion.texi と effective-version.texi を生成するためのものです。1 度目の make は effective-version.texi の生成で失敗します。でも、2 度目の make で成功します。

❗ これで PDF ファイルを作成すると、「5.8.2 Autoconf Mactros」の内容が削減されていて、マニュアル全体として 2 ページ分短くなります。理由は不明です。ちなみに、Debian の公式パッケージと一緒にインストールされた Texinfo のマニュアルを見ると削減されていません。このことから、バイナリーパッケージをビルドしないとダメなのかも知れません（なんだかヘンな感じですけど）。この節の内容は単なる記録です。

A.4 最新版の Guile をビルドする

Guile のサイトを見ると、Guile 2.2.4 は 2018.7 にリリースされた安定版のようです。筆者が勉強を始めたのが 2020.10.30 なので、そんなに古くはないと思います。でも、勉強開始時点の最新版は Guile 3.0.4 です。メジャーバージョンが違くと最新版が使いたくなります。

Debian の不安定版 (sid) のパッケージとして guile-3.0 (Guile 3.0.4) が配布されています。でも、他のパッケージとの依存関係のためにインストールできません。そこで、最新版のソースコードを持ってきてビルドしてみました。

ビルドなんてほとんど経験がないのでけっこう大変でした。ただ、大変だった理由ははっきりしていて、

- **開発用パッケージをインストールしとけ**

ってことです。考えて見れば当たり前のことです。作業内容を正確に解説する力量はないのですが、おおよその作業内容を記録します。

■ ソースコードの入手と展開

Guile のダウンロードサイトからソースコードを入手して、適当な作業用ディレクトリに保存します。筆者が勉強を始めた 2020.10.30 時点の最新版は guile-3.0.4.tar.gz です。

- サイト <https://www.gnu.org/software/guile/download/>
- ファイル guile-3.0.4.tar.gz

作業用ディレクトリ上で tar ボールを展開して、展開後のディレクトリに移動します。

```
$ tar -zxvf guile-3.0.4.tar.gz
.....
$ cd guile-3.0.4
```

■ configure

guile-3.0.4にある configure スクリプトを実行します。

```
$ ./configure | tee -a guile.install.log
```

configure を実行するときは、tee コマンドを使ってログを取ります。筆者はログを取らずに漫然と作業を始めてしまったので後悔しています（そのために詳しいことが書けません）。2,3 回失敗したのちにログを取るようになりました。

ビルドの環境が整っていないと、「ライブラリが存在しない」とか「バージョンが古い」といった、たいていはウソのエラーメッセージを表示して configure は停止します。これは開発用パッケージ (-dev の付いたパッケージ) がインストールされていないことが原因です。

例えば、「GNU MP が存在しないかまたはバージョンが古い！4.1 以上をよこせ！」なんてエラーメッセージが表示されたりします。これは GNU MP (多倍長精度演算ライブラリ) が存在しないわけでもバージョンが古いわけでもありません。なぜなら、libgmp はインストールされてるし、バージョンは 6.1.2 だからです。エラーは、GNU MP の開発用パッケージ libgmp-dev がインストールされていないことが原因です。そこで、それをインストールします。

```
$ sudo apt install libgmp-dev
```

インストールしたあと、再び configure を実行します。そうすると、今度は GNU MP に対するエラーメッセージは表示されません。でも、他のライブラリのところで同様のエラーメッセージが表示されます。

こういったライブラリは幾つかあり、おそらく次のパッケージをインストールする必要があります。ただ、ログを取らずに作業を始めてしまったので少しあやふやです。

- libgmp-dev
- libltdl-dev
- libunistring-dev
- libgc-dev


```
- libffi-dev
- libreadline-dev
- pkg-config
```

他にも必要なパッケージがあるかも知れません。

不足しているものがあるとき、`configure` はエラーメッセージを表示して停止します。それを手掛かりに必要なパッケージ（たいていは開発用パッケージ）の名前を調べて、インストールして、`configure` を再度実行する、といった作業を繰り返します。

それから、`libreadline-dev` については `WARNING` だけを表示して `configure` はとりあえず成功します。この点はログを取って確認しました。 `WARNING` といえども解消したほうがよいでしょう。 `libreadline-dev` をインストールすると解消します。 `configure` が完了したらログファイルを見て、おかしいメッセージがないか確認したほうがよいでしょう。

■ ビルドとインストール

`configure` が完了したら `make` して Guile をビルドします。少し時間がかかります。

```
make
```

ビルドした Guile をインストールします。

```
sudo make install
```

Guile のコマンド群（`guile` や `guild` など）は `/usr/local/bin` にインストールされ、その他の諸々のファイルは `/usr/local/share/guile` に保存されます。

以上で完了です。大変だったのは `configure` だけです。そのあとの `make` やインストールは何もトラブルはありませんでした。

¶ 上で述べた作業によって次のコマンドが `/usr/local/bin` にインストールされます。

- `guile` Guile のインタプリタ
- `guild` 仮想マシンコードへのコンパイラ
- `guile-tools` `guild` へのシンボリックリンク
- `guile-config`
- `guile-snarf`

`guile-config` と `guile-snarf` の機能や用途は今の筆者には分かりません。

¶ Debian Linux の標準の設定では、`PATH` 変数（パス環境変数）には `/usr/bin` よりも前に `/usr/local/bin` が設定されています。そのため、上のインストール作業を行ったあとに `guile` コマンドを実行すると Guile 3.0.4（つまり、`/usr/local/bin/guile`）が起動します。

■ alias の設定と動作確認

筆者は、試しに最新版をビルドしてみたものの、あとで述べる「気になる点」のために Debian 用の公式版である Guile 2.2.4 を使おうと思います。そこで、次のような alias を設定しています。ただし、`guile` と `guild` だけ設定します（他のコマンドの用途がいまのところ不明なので）。

```
alias guile='/usr/bin/guile'
alias guild='/usr/bin/guild'
alias guile3='/usr/local/bin/guile'
alias guild3='/usr/local/bin/guild'
```

ターミナルを再起動して `guile3` を動かしてみます。

```
$ guile3
```



```
GNU Guile 3.0.4
Copyright (C) 1995-2020 Free Software Foundation, Inc.

Guile comes with ABSOLUTELY NO WARRANTY; for details type ',show w'.
This program is free software, and you are welcome to redistribute it
under certain conditions; type ',show c' for details.

Enter ',help' for help.
scheme@(guile-user)> (+ 10 20) ☐
$1 = 30
scheme@(guile-user)> (* 10 20) ☐
$2 = 200
scheme@(guile-user)> (quit) ☐
```

ちなみに、guile3を実行したときにも ~/.guile が実行されます。

■ 気になる点

configure が大変だったので、ほんのりと達成感があります。でも、喜んでばかりはいられません。気になる点を記録しておきます。

- Guile 3.0.4 によって必須とされていないライブラリが筆者の Debian にはインストールされていなくて、make の際にそのライブラリがリンクされずに、そのライブラリの機能が使えないかも知れない。これはあり得ることです。残念ながら経験値が不足していてログを見てもそういった状況を特定することができません。実際に使ってみて不具合が発生したときに対応するしかありません。
- make する際に最適化を行っているわけではないので、実行速度が遅いかも知れない。これもあり得ることです。実際、パッケージ版の Guile 2.2.4 と自前の Guile 3.0.4 の実行時間を簡単に比較してみました。残念ながら Guile 3.0.4 のほうがちょっぴり遅いといった結果になりました。最適化の方法がよく分からないので、いまのところ仕方ありません。

更新履歴

- 2020.10.30 勉強始め.
- 2021.02.09 Github に保存.

索引

!# 18
#! 18
,b 34
,binding 10, 34
,h 34
,help 34
,import 34
,pr 34
,profile 34
,q 4
,quit 4
,t 34
,time 34
,tr 35
,trace 35
,use 34
-ds スイッチ 25
-e スイッチ 14
-l スイッチ 11, 25
-s スイッチ 13
~/.cache/guile 10
~/.guile 5, 11, 41
~/.guile.history 5
.scm 6
\ 20

activate-readline 5

coding: XXXXXX 26
command-line 13, 15, 19

Debian Linux 1

exit 5

Guile 1, 36, 38
guile 6

(ice-9 readline) 5

let-values 31
load 7

partition 31
POSIX 24
primitive-load 7

quit 4

REPL 4

(srfi srfi-1) 31
(srfi srfi-11) 31

value history name 8

インポート宣言 28

エントリーポイント 14, 16

拡張子 6

キャッシュ 10
キャッシュディレクトリ 10

コア言語 31
コマンドライン引数 13-16, 19

シェバン 18
シェバン行 18
式 28

定義 28
デバッグモード 8

バイトコード 10

ブロックコメント 18

補完機能 5

メタスイッチ 20

履歴機能 5