Enumitem パッケージ

@monaqa

目次

1. はじめに

本ドキュメントは，enumitem パッケージ (v2.0.0) の仕様および使い方について述べたもので

す．Enumitem は v1.x から v2.0 にバージョンアップする際にほぼ全てのコマンドの仕様を変更しており，v1.x を使っている場合はこのドキュメントに書かれている内容では動きません．旧版のドキュメントを確認してください．

2. Enumitem パッケージの概要

Enumitem は，組版システム $\mathrm{SATYSF_I}$ において，豊富な箇条書きリストや番号付きのリストを出力するためのパッケージです．$\mathrm{SATYSF_I}$ には itemize というパッケージが標準で用意されていますが，enumitem パッケージでは itemize パッケージと比較してより豊富な機能を提供します（2020 年 5 月 23 日現在）．具体的には，enumitem パッケージを使うことで以下のような恩恵を受けられます．

- デフォルトで豊富なスタイルを選択できる
- 番号付き箇条書き環境をネストさせることができる[*1]
- 定義リストを作成できる
- ネストごとに箇条書きのスタイルを変更できる
- ユーザ自身がスタイルを容易に拡張できる

3. 基本的な使い方

Enumitem を使うには，当然ですがパッケージを読み込む必要があります．パッケージが正しくインストールされていれば，文書の冒頭に以下の 1 行を追加するだけで読み込むことができます．

```
@require: enumitem
```

Enumitem は標準パッケージと同様，`+listing` 及び `+enumerate` という箇条書きインターフェースを提供します．デフォルトでは以下のように箇条書きを書くことができます．

```
+listing{
  * hoge
  * fuga
    ** fuga1
      *** fuga11
```

1 2020 年 5 月現在，標準ではサポートされていません．

```
      *** fuga12
    ** fuga2
}
```

- hoge
- fuga
  - fuga1
    - fuga11
    - fuga12
  - fuga2

```
+enumerate{
  * hoge
  * fuga
    ** fuga1
      *** fuga11
      *** fuga12
    ** fuga2
}
```

1. hoge
2. fuga
   (a) fuga1
      (i) fuga11
      (ii) fuga12
   (b) fuga2

このように番号つき箇条書き環境のネストもサポートしており，ネストの深さによってインデックスの体裁を変えることができます.

また，文章の途中で箇条書きをはさむ場合は `\listing` や `\enumerate` コマンドを使う

ことで改段落を伴わずに箇条書きを挿入できます.

```
+p{
  寿限無，寿限無，五劫のすりきれ，
  海砂利水魚の
  \listing{
    * 水行末
    * 雲来末
    * 風来末
  }
  食う寝るところに住むところ ...
}
```

寿限無，寿限無，五劫のすりきれ，海砂利水魚の

- 水行末
- 雲来末
- 風来末

食う寝るところに住むところ ...

4. +listing における体裁の指定

ここまでの使用法は標準パッケージと大きく変わりませんでした. Enumitem パッケージが提供する +listing などのコマンドは，インデックス（）の体裁を指定されたものに変更する機能も持っています.

4.1. オプション引数を用いたインデックスの体裁指定

+listing 及び +enumerate は以下のように 1 つのオプション引数を受け付け，インデックスの体裁を指定することができます.

```
+listing?:(Enumitem.white-bullet){
  * hoge
    ** hoge1
    ** hoge2
```

```
  * fuga
  * piyo
}
```

- hoge
  - hoge1
  - hoge2
- fuga
- piyo

```
+enumerate?:(Enumitem.dot-arabic){
  * hoge
    ** hoge1
    ** hoge2
  * fuga
  * piyo
```

1. hoge
   1. hoge1
   2. hoge2
2. fuga
3. piyo

インデックスの体裁が上コードにおける `white-bullet` や `dot-arabic` というのは，`Enumitem` と呼ばれるモジュール内で定義されている関数であり，`idxfmt` 型を持ちます． モジュール内で定義されている `idxfmt` 型の関数は他にも多数存在し，それらをオプション引数に渡すことで箇条書きを模様替えできるという仕組みです．

なお，実際のマークアップで変数名の前に `Enumitem.` を付けるのは多少面倒と感じられるかもしれません．このような場合，`Enumitem` モジュールを open することで名前空間を省略することができます．

```
@require: enumitem
open Enumitem
in
document(| 中略 |)'<
  +listing?:(white-bullet){
    * hoge
    * fuga
  }
>
```

このように書くと，Enumitem モジュール内で定義された関数の前に Enumitem. を付ける必要がなくなります． その代わり， 他のパッケージやユーザで定義した関数名と衝突しないようにするのは使用者の責任となります．Enumitem パッケージでは，+listing や +enumerate といったコマンドについては名前空間を省略できるようにしていますが，raw-arabic などの関数名は open しない限りモジュール名を省略できないようにしています．

また，箇条書きを書くときにいちいち同じオプション引数を付けるのは面倒だと感じられるかもしれません． その場合は最初に +listing のデフォルトの挙動を変更してもよいでしょう． プリアンブル部分に以下のように書くと， 以下のコードでは +listing を用いた場合は常に白丸がラベルとして使われるようになります．

```
let-block +listing item = '<
  +Enumitem.listing?:(Enumitem.white-bullet)(item);
>
```

余談ですが，+listing と +enumerate の違いはオプション引数を省略したときのデフォルトの体裁のみであり， オプション引数を設定すると内部での処理は全く同一となります．したがって，+listing?:(Enumitem.dot-arabic) のように +listing で番号付き箇条書きを書くことも，+enumerate?:(Enumitem.bullet) のように +enumerate で番号のない箇条書きを書くことも可能ではあります[*2]．

4.2. インデックスの体裁を定める変数名一覧

以下は Enumitem モジュール内で定義されている体裁指定用変数の一覧です．番号付きの箇

2 もちろん仕組み上そうなっているというだけであり，番号のない箇条書きには +listing を，番号付きには +enumerate を，などと使い分けたほうが可読性は向上するでしょう．

条書きを行いたい場合は，以下の変数を使用できます．

- アラビア数字系
  1 raw-arabic
  2. dot-arabic
  (3) paren-arabic
  [4] bracket-arabic
- ローマ数字系
  i raw-roman
  II raw-Roman
  iii. dot-roman
  IV. dot-Roman
  (v) paren-roman
  (VI) paren-Roman
  [vii] bracket-roman
  [VIII] bracket-Roman
- アルファベット系
  a raw-alph
  B raw-Alph
  c. dot-alph
  D. dot-Alph
  (e) paren-alph
  (F) paren-Alph
  [g] bracket-alph
  [H] bracket-Alph

また，番号の無い箇条書きには以下の変数を使用できます．

- bullet
- white-bullet

4.3. インデックスの体裁をユーザ定義する方法

ここまでの説明で，「dot-arabic 等は idxfmt 型の関数だ」と述べましたが，そもそも idxfmt 型とは何でしょうか．実は idxfmt 型とは int list -> context -> inline-boxes という型のシノニムであり，

- 箇条書きのインデックスのリスト (int list)
- 現在の本文のテキスト処理文脈 (context)

を入力として，箇条書きのインデックスを描画するインラインボックス列 (inline-boxes) を返す関数を表すものです．これは必ずしも予めモジュール内で用意された関数である必要はありません．つまり，idxfmt 型を持つ関数を自作して +listing コマンドのオプション引数に渡せば，自分の定義したインデックス表示を持つ箇条書きを作成することができます．

idxfmt の使い方を知るため，まず最も単純な idxfmt 型を持つ関数を設計してみましょう．以下のような関数を定義します．この関数を定義するためには，標準で提供されている list パッケージが必要です．

```
let example idxlst ctx =
  let idx-to-ib idx =
    (read-inline ctx (embed-string (arabic idx)))
      ++ (read-inline ctx {.})
  in
  idxlst |> List.map idx-to-ib
         |> List.fold-left (++) inline-nil
```

SATySFi や OCaml に慣れている方はすぐに読めるかもしれませんが，この関数は int list -> context -> inline-boxes 型を持ちます．まず，関数内部で定義された idx-to-ib は int -> inline-boxes 型を持つクロージャであり，この関数を用いて int list 型を持つ idxlst を inline-boxes list 型へと変換します．続いて得られたリストを左から結合することにより，最終的な inline-boxes 型の値を得ます．したがって，もし example の第 1 引数に [2; 3; 5; 7] というリストが与えられた場合，（第 2 引数のテキスト処理文脈により組み方は変わりますが），「2.3.5.7.」という数字の列を持ったインラインボックス列が出力となります．

では，この example を +listing に指定するとどうなるでしょうか．実際に組んでみると以下のようになります．

```
+listing?:(example){
  * あああ
  * いいい
  * ううう
    ** あああああ
    ** いいいいい
    ** ううううう
  * えええ
    ** あああああ
      *** あああああああ
      *** いいいいいい
      *** ううううううう
    ** いいいいい
  * おおお
}
```

1.あああ
2.いいい
3.ううう
    1.3.あああああ
    2.3.いいいいい
    3.3.ううううう
4.えええ
    1.4.あああああ
        1.1.4.あああああああ
        2.1.4.いいいいいい
        3.1.4.ううううううう
    2.4.いいいいい
5.おおお

結果を見れば規則性が分かると思いますが，`+listing` コマンドは `idxfmt` 型の関数が与えられたとき，`idxfmt` に「現在の箇条書きの項目が何番目か」という情報を格納したリス

トと現在のテキスト処理文脈の 2 つを渡し，出てきた出力をそのまま箇条書きのラベルとします．なお，「現在の箇条書きが何番目か」を表したリストでは，「n 番目に深いリストの番号」が「末尾から数えて n 番目の要素」に格納されています．つまり，リストの先頭にあるものが「最も深いネストの番号」を表し，末尾にあるものが「最も浅いネストの番号」を表します．これは少し直感に反するかもしれませんが，逆にするより実装が楽になることが多いためこのような仕様にしました．

この仕様さえ理解すれば，好きな箇条書き環境を定義できます．たとえば，「問 1.」のようなインデックスは以下のようにして実装できます．

```
let label-toi idxlst ctx =
  let idx = match idxlst with
    | [] -> 1
    | idx :: _ -> idx
  in
  let it-num = embed-string (arabic idx) in
  read-inline ctx {問 #it-num;.\ }
```

`let idx =` から始まる行で `idxlst` の一番最初の要素を取り出し，それをインラインテキストに変換しています．実際には enumitem パッケージの内部で `label-toi` の引数に空リストが与えられることはないはずであり，それを踏まえればもう少し簡潔に書くこともできます[*3]．

```
let label-toi (idx :: _) ctx =
  let it-num = embed-string (arabic idx) in
  read-inline ctx {問 #it-num;.\ }
```

話が少しそれました．実際に `label-toi` を使ってみましょう．

```
+enumerate?:(label-toi){
  * hoge
  * fuga
    ** fuga1
      *** fuga11
```

3 ただし，ユーザが故意に与えた，enumitem パッケージの不具合があったなどの理由で `label-toi` の引数に空リストが与えられてしまった場合は実行時エラーとなるため，その点には注意が必要です．

```
      *** fuga12
    ** fuga2
}
```

問 1. hoge

問 2. fuga

　問 1. fuga1

　　問 1. fuga11

　　問 2. fuga12

　問 2. fuga2

このように，ネストがある場合でも機能していることが分かります．

なお，第 1 引数を「使わない」だけで，番号のない箇条書きの体裁も作成できます．

```
let label-japanese-ichi _ ctx =
  read-inline ctx {一} ++ inline-skip 10pt
```

番号つき箇条書きよりも更にシンプルになりました．このように定義された箇条書きは以下のように用いることができます．

```
+listing?:(label-japanese-ichi){
  * 廣ク會議ヲ興シ萬機公論ニ決スベシ
  * 上下心ヲ一ニシテ盛ニ經綸ヲ行フベシ
  * 官武一途庶民ニ至ル迄各其志ヲ遂ケ人心ヲシテ倦マサラシメン事ヲ要ス
  * 舊來ノ陋習ヲ破リ天地ノ公道ニ基クベシ
  * 智識ヲ世界ニ求メ大ニ皇基ヲ振起スベシ
}
```

一　廣ク會議ヲ興シ萬機公論ニ決スベシ

一　上下心ヲ一ニシテ盛ニ經綸ヲ行フベシ

一　官武一途庶民ニ至ル迄各其志ヲ遂ケ人心ヲシテ倦マサラシメン事ヲ要ス

一　舊來ノ陋習ヲ破リ天地ノ公道ニ基クベシ

一　智識ヲ世界ニ求メ大ニ皇基ヲ振起スベシ

4.4. 便利な関数

ネストの深さに応じて箇条書きのラベルを変えたい，という要求は珍しくありません．インデックスの体裁を自身で定義すれば当然そういった箇条書きも作成できます．しかし，既存の体裁を使いまわしたいのであれば，Enumitem.change-by-depth 関数を用いるのがより簡単です．

```
+listing?:(
  change-by-depth [bracket-Alph; white-bullet; paren-arabic]
){
  * hoge
  * fuga
    ** fuga1
      *** fuga11
      *** fuga12
    ** fuga2
}
```

[A] hoge

[B] fuga

- fuga1
  (1) fuga11
  (2) fuga12
- fuga2

5. +genlisting を用いたスタイルの指定

+listing で指定できるのはインデックスの体裁のみでしたが，より一般化されたコマンドである +genlisting を用いると，本文の体裁を指定することもできます．

実は +listing や +enumerate コマンドは +genlisting を用いて定義されています．

お世辞にも簡潔なインターフェースとは言えませんが，複雑なぶん，柔軟な設定が可能です．たとえば

- 本文のフォントサイズや色といった体裁
- 箇条書きの項目間の幅
- 1 つの項目内で行を折り返すときのインデント幅
- ネストするときに追加されるインデントの幅

といった項目は，全てこの関数 1 つで調整することができます．`+genlisting` コマンドの第 2 引数に渡すだけなので，パッケージを書き換える必要もありません．

6. 動的なフラグを用いたラベル操作

6.1. `+xgenlisting` コマンド

たとえば，以下のような To-Do リストを作成したくなったとします．

☐ ミルクを買う．

☐ The SATySFIbook を読む．

☑ SATySFI を完全に理解する．

☐ 課題を解く．

ここで， ラベルを作成する関数は既に用意されているとします． たとえば以下の関数 `square-label is-checked ctx` は，第 1 引数 `is-checked` が true のときチェック済みの，false のときチェック済みでないチェックボックスを描画する関数です．

```
let square-label is-checked ctx =
  let fs value = (get-font-size ctx) *' value in
  let fsize = fs 1.0 in
  let gr-square (x, y) =
    stroke 0.5pt Color.black
      (Gr.rectangle (0pt, 0pt) (fs 0.5, fs 0.5 ))
      |> shift-graphics (x, y +' fs 0.15)
  in
  let gr-mark-done (x, y) =
    stroke 0.5pt Color.black (
      Gr.poly-line (fs (0.0 -. 0.1), fs 0.45)
        [(fs 0.15, fs 0.15); (fs 0.75, fs 0.85)])
```

```
        |> shift-graphics (x, y +' fs 0.15)
  in
  let gr point =
    if is-checked then
      [gr-square point; gr-mark-done point]
    else
      [gr-square point]
  in
  inline-skip 10pt ++ (inline-graphics fsize fsize 0pt gr)
```

今までに紹介された +genlisting を用いても，上のように「3 番目のみチェックが付いたリスト」を実現することはできます．以下のようにパターンマッチや if 文などを用いて，チェックを付けたい場所のみ場合分けして処理すればよいのです．

```
+genlisting(fun depth idx ctx -> (
  let checked = match idx with
    | 3 -> true
    | _ -> false
  in
  square-label checked ctx
))(default-item){
  * ミルクを買う．
  * The \SATySFi;book を読む．
  * \SATySFi; を完全に理解する．
  * 課題を解く．
}
```

しかしこれはあまり直観的ではなく，また編集もしやすいとはいえません．「ミルクを買う」にチェックを付けたいとき，「ミルクを買う」というテキストから離れた場所を編集しなければならないのは手間ですし，「どこにチェックが付いているのか」をひと目で判断することができません．また，今回はアイテムの数が 4 つだったため数えるのも楽でしたが，もっと長いリストの途中にチェックをつけるためにいちいち数えなければならないのは手間です．このように，体裁指定用変数を用いたレイアウトの指定は最初から最後まで一貫した規則を持つ箇条書きの生成には適しているものの，例外があったり，動的にレイアウトを変更したかったりするケースにはあまり適していません．

Enumitem パッケージではこのようなケースに対応するため，本文中でパラメータに値をセットし，その値をラベルに反映させる方法を提供しています．たとえば先程の To-Do リストは，プリアンブルにて以下のように定義された +todo-list 及び \done コマンドで作成したものです．

```
let done = EnumitemBase.make-param false
let-inline \done = {\set-item(done)(true);}
let-block +todo-list item = '<
  +xgenlisting(
    fun depth idx ctx -> square-label (Param.get done) ctx
  )(default-item)(item);
>
```

この +todo-list 及び \done コマンドを使うと，先程の箇条書きは以下のようにシンプルに書くことができます．

```
+todo-list{
  * ミルクを買う．
  * The \SATySFi;book を読む．
  * \done; \SATySFi; を完全に理解する．
  * 課題を解く．
}
```

□ ミルクを買う．

□ The SATYSFIbook を読む．

☑ SATYSFI を完全に理解する．

□ 課題を解く．

このように，本文に \done; というコマンドが付いているアイテムに限って，チェックボックスにチェックマークが付くようになります．\done の位置は 3 番目のアイテムの本文中であれば，どこにあってもかまいません．

何が起きたのか，もうすこし詳細に説明します．ポイントは以下の 4 点です．

- 箇条書きのアイテム内で一時的な値を保持するための器（パラメータ）を定義することができる．

- パラメータには，本文中で `\set-item(param)(value);` と打つことで一時的に値を設定することができる．
- `+xgenlisting` を使うと，第 1 引数でラベルのスタイルを指定する際に，`Param.get` 関数でパラメータの値を受け取り， その値に応じてラベルの出力を変更することができる．
- `\set-item` で変更した値はそのアイテムが終わると破棄され，デフォルト値にもどる．

今回定義した `done` パラメータは bool 型を保持する値でしたが，実際には `int`，`inline-text`，`int list -> context -> inline-boxes` 型など様々な型を持つパラメータを作成することができます． したがって， 上記の例のようにフラグとして使うだけではなく， パラメータの値に応じてインデックスの値を変更したり，ラベルそのもののレイアウトを変えたり，と多彩なカスタマイズが可能となります．

6.2. +xgenlisting の注意点

`+xgenlisting` は便利なコマンドですが，`+genlisting` にはない欠点が存在します． それは「本文に副作用のあるコマンドを入れると（おそらく）不具合が起きる」ということです．たとえば，内部でカウンタをインクリメントさせる `\footnote` を `+genlisting` の本文で用いると，（実装にもよるとは思いますが）脚注の数字が 2 つインクリメントされる可能性があります．

理由は実装上の事情にあります． 実は，「本文中にあるコマンドを読み取ってラベルに反映する」という行為は少し不自然なのです． なぜなら本来，ラベルを組み終わってはじめて後続する本文のテキスト幅が分かり，本文を組むことができるようになるからです． ラベルを組まないと本文が組めない，しかし本文を読まないとラベルを組めない，というのが本実装を困難にする点でした．

そこで，`+xgenlisting` ではその問題を解決するため，「ラベルを組む前に `read-inline` で本文を読み，読み終わったものは一度破棄する」という方式で実装を行いました．本文中のコマンドは `read-inline` で読まれることにより評価され，`\set-item` がある場合はパラメータが一時的な値へとセットされます． その後であれば既にユーザ指定が分かっているためラベルを組むことができ，ラベルを組んでから再度改めて本文を組むことができます．

そしてこの方法だと （お気付きの通り）， 本文は `read-inline` によって 2 度評価されます． 副作用のないコマンドであれば何度評価されても結果は変わりませんから問題ありませんが， 副作用のあるコマンド， 特に 1 度だけ呼ばれることを想定しているコマンドを `+xgenlisting` の中に入れると，このことによって挙動がおかしくなるのです．

7. 定義リストの作成

工事中

8. おわりに

バグ報告・ PR などお待ちしています.