

新春最適化手法解説: *Tail Modulo Cons*

@第五回関数型プログラミング（仮）の会

びしょ〜じょ

こんにちは、びしょ〜じょです



株式会社カンム

Golangerしてます



すきなもの

代数的効果、継続、処理系



Nymphium



Nymphium



突然ですが

突然ですが

関数型プログラミングといえは…？

突然ですが

関数型プログラミングといえば…？

再帰でしょ

本日の内容

OCaml(≥ 4.14)で

OCaml(≥ 4.14)で 再帰関数に

本日の内容

OCaml(≥ 4.14)で

再帰関数に

[@tail_mod_cons]

と書くと

本日の内容

OCaml(≥ 4.14)で

再帰関数に

`[@tail_mod_cons]`

と書くと

速くなる 場合がある

本日の内容

Tail Modulo Cons
Frédéric Bour¹², Basile Clément¹, and Gabriel Scherer¹
¹ INRIA
² Tarides

再帰関数

[@tail_mod_cons]

と書くと

速くなる 場合がある

再帰

最も基本的な構造

```
let rec map f = function  
  | [] -> []  
  | x :: xs -> f x :: map f xs
```

再帰

最も基本的な構造

```
let rec map f = function  
  | [] -> []  
  | x :: xs -> f x :: map f xs
```

型、関数…

定義に自身を参照

末尾再帰

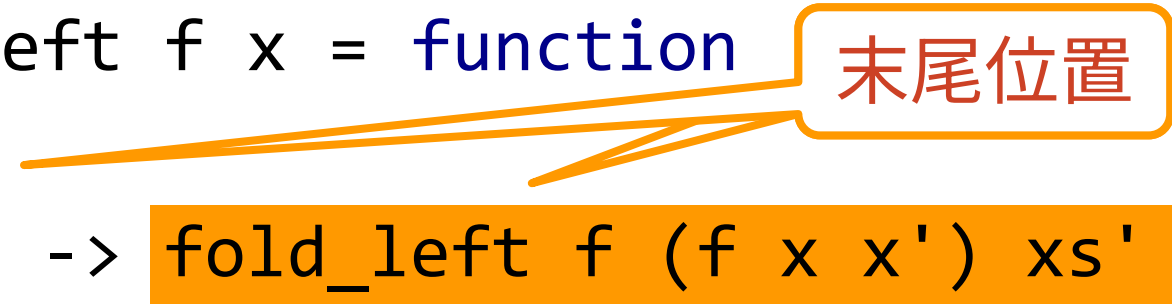
関数の末尾位置で自身を再帰呼び出し

```
let rec fold_left f x = function
| [] -> x
| x' :: xs' -> fold_left f (f x x') xs'
```

末尾再帰

関数の末尾位置で自身を再帰呼び出し

```
let rec fold_left f x = function
| [] -> x
| x' :: xs' -> fold_left f (f x x') xs'
```



末尾位置

末尾再帰

関数の末尾位置で自身を再帰呼び出し

```
let rec fold_left f x = function
| [] -> x
| x' :: xs' -> fold_left f (f x x') xs'
```

末尾位置

再帰呼び出し

末尾位置+再帰で

末尾再帰

末尾呼び出し最適化

末尾再帰にすると**末尾呼び出し最適化**ができる

- 再帰呼び出し(call)をjumpに置き換えられ
- コールスタックの操作が不要になりperformant
- そしてスタックオーバーフローもしない!

末尾呼び出し最適化

末尾再帰にすると**末尾呼び出し最適化**ができる

- 再帰呼び出し(call)をjumpに置き換えられ
- コールスタックの操作が不要になりperformant
- そしてスタックオーバーフローもしない!

ところで

mapって…末尾再帰じゃない

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

mapって…末尾再帰じゃない

```
let rec map f = function  
  | [] -> []  
  | x :: xs -> f x :: map f xs
```

A正規化すると…

```
let x' = f x in  
let xs' = map f xs' in  
x' :: xs'
```

後続処理が
あって
無理や…

mapを救いたい

どうする…？

mapを救いたい

どうする…?

- `map_rev`して`rev`
 - performantでないケース
- 継続渡し方式(CPS)
 - 手で書くのですか…?



mapを救いたい

どうする…?

- map_revしてrev
 - performantでないケース
- 継続渡し方式(CPS)
 - 手で書くのですか…?
- ***Tail-Recursion
Modulo Cons***



Tail-Recursion Modulo Cons

Tail-Recursion Modulo Cons とは...

Tail-Recursion Modulo Cons

Tail-Recursion Modulo Cons とは...

末尾再帰

Tail-Recursion Modulo Cons

Tail-Recursion Modulo Cons とは...

末尾再帰

+

Destination Passing Style

(*DPS*)

Tail-Recursion Modulo Cons

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

Tail-Recursion Modulo Cons

```
let rec map f = function
| [] -> []
| x :: xs -> f x :: map f xs
```



```
let rec map f = function
| [] -> []
| x :: xs ->
  let y = f x in
  let d = y :: Hole in
  map_dps d f xs;
  d
```

```
and map_dps d f = function
| [] -> set_field d 1 []
| x :: xs ->
  let y = f x in
  let d' = y :: Hole in
  set_field d 1 d';
  map_dps d' f
```

Tail-Recursion Modulo Cons

```
let rec map f = function
| [] -> []
| x :: xs ->
  let y = f x in
  let d = y :: Hole in
  map_dps d f xs;
d
```

```
and map_dps d f = function
| [] -> set_field d 1 []
| x :: xs ->
  let y = f x in
  let d' = y :: Hole in
  set_field d 1 d';
  map_dps d' f
```

Tail-Recursion Modulo Cons

```
let rec map f = function
| [] -> []
| x :: xs ->
  let y = f x in
  let d = y :: Hole in
  map_dps d f xs;
  d
```

プレースホルダー

```
and map_dps d f = function
| [] -> set_field d 1 []
| x :: xs ->
  let y = f x in
  let d' = y :: Hole in
  set_field d 1 d';
  map_dps d' f
```

Tail-Recursion Modulo Cons

```
let rec map f = function
| [] -> []
| x :: xs ->
  let y = f x in
  let d = y :: Hole in
  map_dps d f xs;
d
```

プレースホルダー

```
and map_dps d f = function
| [] -> set_field d 1 []
| x :: xs ->
  let y = f x in
  let d' = y :: Hole in
  set_field d 1 d';
  map_dps d' f
```

データの
宛先を渡す
(*destination passing*)

Tail-Recursion Modulo Cons

```
let rec map f = function
| [] -> []
| x :: xs ->
  let y = f x in
  let d = y :: Hole in
  map_dps d f xs;
d
```

プレースホルダー

データの
宛先を渡す
(*destination passing*)

```
and map_dps d f = function
| [] -> set_field d 1 []
| x :: xs ->
  let y = f x in
  let d' = y :: Hole in
  set_field d 1 d';
  map_dps d' f
```

値を挿入

Tail-Recursion Modulo Cons

自身は再帰していない

```
let rec map f = function
| [] -> []
| x :: xs ->
  let y = f x in
  let d = y :: Hole in
  map_dps d f xs;
d
```

プレースホルダー

データの
宛先を渡す
(*destination passing*)

```
and map_dps d f = function
| [] -> set_field d 1 []
| x :: xs ->
  let y = f x in
  let d' = y :: Hole in
  set_field d 1 d';
  map_dps d' f
```

値を挿入

Tail-Recursion Modulo Cons

自身は再帰してない

末尾再帰してる!!

```
let rec map f = function
| [] -> []
| x :: xs ->
  let y = f x in
  let d = y :: Hole in
  map_dps d f xs;
d
```

プレースホルダー

データの
宛先を渡す
(*destination passing*)

```
and map_dps d f = function
| [] -> set_field d 1 []
| x :: xs ->
  let y = f x in
  let d' = y :: Hole in
  set_field d 1 d';
  map_dps d' f
```

値を挿入

[@tail_mod_cons]

Tail recursion modulo consは古典的な手法

[@tail_mod_cons]

Tail recursion modulo consは古典的な手法


- Lisp(1970～)、Prolog、etc.
- しかし手書き… 破壊的代入危ない 

[@tail_mod_cons]

Tail recursion modulo consは古典的な手法

- Lisp(1970～)、Prolog、etc.
- しかし手書き… 破壊的代入危ない 



定式化することで
プログラム変換として
コンパイラに導入 

Tail Modulo Cons

Frédéric Bour¹², Basile Clément¹, and Gabriel Scherer¹

¹ INRIA

² Tarides

[@tail_mod_cons]

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs
```

`@tail_mod_cons`

```
let @tail_mod_cons  
  rec map f = function  
    | [] -> []  
    | x :: xs -> f x :: map f xs
```

アノテーション
つけるだけで

[@tail_mod_cons]

```
let[@tail_mod_cons]  
  rec map f = function  
    | [] -> []  
    | x :: xs -> f x :: map f xs
```

アノテーション
つけるだけで



プログラム変換!!

(コンパイラの間言言語)

```
let rec map f = function  
  | [] -> []  
  | x :: xs ->  
    let y = f x in  
    let d = y :: Hole in  
    map_dps d f xs;  
  d
```

```
and map_dps d f = function  
  | [] -> set_field d 1 []  
  | x :: xs ->  
    let y = f x in  
    let d' = y :: Hole in  
    set_field d 1 d';  
  map_dps d' f
```


Tail Modulo Cons Context

小さな対象言語

Exprs $\ni e, d ::=$	FunctionNames $\ni f$
$n \in \mathbb{N}$	Patterns $\ni p ::= x \mid K(p_i)^i$
$f e$	
let $x = e$ in e'	Stmt $\ni s ::=$ let rec $(f_i x = e_i)^i$
$K(e_i)^i$	
match e with $(p_i \rightarrow e'_i)^i$	
$d.e \leftarrow e'$	

Tail Modulo Cons Context

小さな対象言語

Exprs $\ni e, d ::=$ x, y

$| n \in \mathbb{N}$

$| f e$

$| \text{let } x = e \text{ in } e'$

$| K(e_i)^i$

$| \text{match } e \text{ with } (p_i \rightarrow e'_i)^i$

$| d.e \leftarrow e'$

FunctionNames $\ni f$

Patterns $\ni p ::= x \mid K(p_i)^i$

Stmt $\ni s ::= \text{let } \text{rec} (f_i x = e_i)^i$

関数定義は
トップレベルのみ

- let式
- 関数適用
- 代数的データ型
- パターンマッチング
- 代入

Tail Modulo Cons Context

$\text{ConstrCtx} \ni C [\Box] ::= \Box$
 $\quad \mid K ((e_i)^i, C [\Box], (e_j)^j)$

$\text{TailCtx} \ni T ::=$
 $\quad \mid \Box$
 $\quad \mid e; T$
 $\quad \mid \text{let } x = e \text{ in } T$
 $\quad \mid \text{match } e \text{ with } (p_j \rightarrow T_j)^j$

$\text{TailModConsCtx} \ni U ::=$
 $\quad \mid \Box$
 $\quad \mid e; U$
 $\quad \mid \text{let } x = e \text{ in } U$
 $\quad \mid \text{match } e \text{ with } (p_j \rightarrow U)^j$
 $\quad \mid K ((e_i)^i, U, (e_j)^j)$

Tail Modulo Cons Context

$\text{ConstrCtx} \ni C[\Box] ::= \Box$
 $\quad | K((e_i)^i, C[\Box], (e_j)^j)$

$\text{TailCtx} \ni T ::=$
 $\quad | \Box$
 $\quad | e; T$
 $\quad | \text{let } x = e \text{ in } T$
 $\quad | \text{match } e \text{ with } (p_j \rightarrow T_j)^j$

$\text{TailModConsCtx} \ni U ::=$

$\quad | \Box$
 $\quad | e; U$
 $\quad | \text{let } x = e \text{ in } U$
 $\quad | \text{match } e \text{ with } (p_j \rightarrow U)^j$
 $\quad | K((e_i)^i, U, (e_j)^j)$

コンストラクタ
に潜っていく
("modulo")

Tail Modulo Cons Context

$\text{ConstrCtx} \ni C[\Box] ::= \Box$
 $\quad | K((e_i)^i, C[\Box], (e_j)^j)$

$\text{TailCtx} \ni T ::=$
 $\quad | \Box$
 $\quad | e; T$
 $\quad | \text{let } x = e \text{ in } T$
 $\quad | \text{match } e \text{ with } (p_j \rightarrow T_j)^j$

$\text{TailModConsCtx} \ni U ::=$

コンストラクタ
に潜っていく
("modulo")

$\quad | \Box$
 $\quad | e; U$
 $\quad | \text{let } x = e \text{ in } U$
 $\quad | \text{match } e \text{ with } (p_j \rightarrow U)^j$
 $\quad | K((e_i)^i, U, (e_j)^j)$

- 例:
- $f \ x :: \text{map } f \ xs$
 - $U = f \ x :: \Box$
 - $\Box = \text{map } f \ xs$

TMC变换

$$\frac{}{d.n \leftarrow \square \rightsquigarrow_{dps} \square [d.n \leftarrow \square]} \quad \frac{d.n \leftarrow U \rightsquigarrow_{dps} T[d_i.n_i \leftarrow \square]^i}{d.n \leftarrow \text{let } x = e \text{ in } U \rightsquigarrow_{dps} \text{let } x = e \text{ in } T[d_i.n_i \leftarrow \square]^i}$$

$$\frac{\forall j, \quad d.n \leftarrow U_j \rightsquigarrow_{dps} T_j[d_{i_j}.n_{i_j} \leftarrow \square]^{i_j}}{d.n \leftarrow \text{match } e \text{ with } (p_j \rightarrow U_j)^j \rightsquigarrow_{dps} \text{match } e \text{ with } (p_j \rightarrow T_j[d_{i_j}.n_{i_j} \leftarrow \square]^{i_j})^j}$$

$$\frac{n' = |I| + 1 \quad d'.n' \leftarrow U \rightsquigarrow_{dps} T[d'_l.n'_l \leftarrow \square]^l}{d.n \leftarrow K((e_i)^{i \in I}, U, (e_j)^j) \rightsquigarrow_{dps} \quad \begin{array}{l} \text{let } d' = K((e_i)^{i \in I}, \text{Hole}, (e_j)^j) \text{ in} \\ d.n \leftarrow d'; \\ T[d'_l.n'_l \leftarrow \square]^l \end{array}}$$

$$\frac{}{\square \rightsquigarrow_{direct} \square}$$

$$\frac{U \rightsquigarrow_{direct} E}{\text{let } x = e \text{ in } U \rightsquigarrow_{direct} (\text{let } x = e \text{ in } E)}$$

$$\frac{\forall j, \quad U_j \rightsquigarrow_{direct} E_j}{\text{match } e \text{ with } (p_j \rightarrow U_j)^j \rightsquigarrow_{direct} (\text{match } e \text{ with } (p_j \rightarrow E_j)^j)}$$

$$\frac{n = |I| + 1 \quad d.n \leftarrow U \rightsquigarrow_{dps} T[d_l.n_l \leftarrow \square]^l}{K((e_i)^{i \in I}, U, (e_j)^j) \rightsquigarrow_{direct} \quad \begin{array}{l} \text{let } d = K((e_i)^{i \in I}, \text{Hole}, (e_j)^j) \text{ in} \\ T[d_l.n_l \leftarrow \square]^l; \\ d \end{array}}$$

TMC変換

$$\begin{array}{c} \frac{}{d.n \leftarrow \square \rightsquigarrow_{dps} \square [d.n \leftarrow \square]} \quad \frac{d.n \leftarrow U \rightsquigarrow_{dps} T[d_i.n_i \leftarrow \square]^i}{d.n \leftarrow \text{let } x = e \text{ in } U \rightsquigarrow_{dps} \text{let } x = e \text{ in } T[d_i.n_i \leftarrow \square]^i} \\[10pt] \frac{\forall j, \quad d.n \leftarrow U_j \rightsquigarrow_{dps} T_j[d_{i_j}.n_{i_j} \leftarrow \square]^{i_j}}{d.n \leftarrow \text{match } e \text{ with } (p_j \rightarrow U_j)^j \rightsquigarrow_{dps} \text{match } e \text{ with } (p_j \rightarrow T_j[d_{i_j}.n_{i_j} \leftarrow \square]^{i_j})^j} \\[10pt] \frac{n' = |I| + 1 \quad d'.n' \leftarrow U \rightsquigarrow_{dps} T[d'_l.n'_l \leftarrow \square]^l}{\begin{array}{c} d.n \leftarrow K((e_i)^{i \in I}, U, (e_j)^j) \rightsquigarrow_{dps} \quad \text{let } d' = K((e_i)^{i \in I}, \text{Hole}, (e_j)^j) \text{ in} \\ d.n \leftarrow d'; \quad T[d'_l.n'_l \leftarrow \square]^l \end{array}} \quad \frac{}{\square \rightsquigarrow_{direct} \square} \quad \frac{U \rightsquigarrow_{direct} E}{\text{let } x = e \text{ in } U \rightsquigarrow_{direct} (\text{let } x = e \text{ in } E)} \\[10pt] \frac{\forall j, \quad U_j \rightsquigarrow_{direct} E_j}{\text{match } e \text{ with } (p_j \rightarrow U_j)^j \rightsquigarrow_{direct} (\text{match } e \text{ with } (p_j \rightarrow E_j)^j)} \\[10pt] \frac{n = |I| + 1 \quad d.n \leftarrow U \rightsquigarrow_{dps} T[d_l.n_l \leftarrow \square]^l}{\begin{array}{c} K((e_i)^{i \in I}, U, (e_j)^j) \rightsquigarrow_{direct} \quad \text{let } d = K((e_i)^{i \in I}, \text{Hole}, (e_j)^j) \text{ in} \\ d \quad T[d_l.n_l \leftarrow \square]^l; \end{array}}$$

え～、つまり？
ということですか？
3行で頼む

TMC变换

TMC変換

1. 対象の関数からDPS versionの関数を作る

`map f xs`  `map_dps d idx f xs`

ここで、

`set_field d idx (map f xs) = map_dps d idx f xs`

TMC変換

1. 対象の関数からDPS versionの関数を作る

`map f xs`  `map_dps d idx f xs`

ここで、

`set_field d idx (map f xs) = map_dps d idx f xs`

2. 2つの変換

- $\rightsquigarrow_{direct}$ でmapの再帰をmap_dpsの呼び出しに
- \rightsquigarrow_{dps} でmapに基づいたmap_dpsの生成

TMC变换

```
let rec map f = function
| [] -> []
| x :: xs ->
    f x :: map f xs
```

TMC変換

dps版を作成
bodyをコピー

```
let rec map f = function
| [] -> []
| x :: xs ->
  f x :: map f xs
```

```
and map_dps d i f = function
| [] -> []
| x :: xs ->
  f x :: map f xs
```

TMC変換

dps版を作成
bodyをコピー

```
let rec map f = function
| [] -> []
| x :: xs ->
    let d = f x :: Hole in
    map_dps d 1 f;
    d
```

```
and map_dps d i f = function
| [] -> []
| x :: xs ->
    f x :: map f xs
```

コンストラクタに潜る

```
f x :: map f xs
```

dps版を呼ぶ

TMC変換

変換の終端
フィールドにセット

```
let rec map f = function
| [] -> []
| x :: xs ->
    let d = f x :: Hole in
    map_dps d 1 f;
    d
```

dps版を作成
bodyをコピー

```
and map_dps d i f = function
| [] -> set_field d i []
| x :: xs ->
    f x :: map f xs
```

コンストラクタに潜る

```
f x :: map f xs
```

dps版を呼ぶ

TMC変換

変換の終端
フィールドにセット

```
let rec map f = function
| [] -> []
| x :: xs ->
    let d = f x :: Hole in
    map_dps d 1 f;
d
```

コンストラクタに潜る

```
f x :: map f xs
```

dps版を呼ぶ

dps版を作成
bodyをコピー

```
and map_dps d i f = function
| [] -> set_field d i []
| x :: xs ->
    let d' = f x :: Hole in
    set_field d i d';
    map_dps d' 1 f
```

コンストラクタに潜る

1つ前のフィールドにセット
末尾再帰

- OCamlの標準ライブラリでも利用が進んでいる

List: replace rev-based tail recursion with TRMC #11402

Merged

nojl

Make List.{map,mapi,map2} TRMC #11362

Merged

nojb merged 6 commits into `ocaml:trunk` from `nojb:list_trmc` on Jul 6, 2022

- containers、baseなどでも普及
- 最近の関連研究オモロイ(紹介論文は2021)
 - *Tail Recursion Modulo Context*
 - consの一般化、諸問題にtackle
 - *Destination-passing style programming: a Haskell implementation*
 - DPS+線形型、結局手で書きたいパターンも安全に




まとめ

- Tail recursion modulo consという技法
 - 非末尾再帰→末尾再帰
 - 手で書くと面倒だし破壊的代入がある
- OCamlなら[@tail_mod_cons]でOK!
- つまり

OCamlを書こう!!!!!!

- 公式も読むと良いですよ
 - https://v2.ocaml.org/manual/tail_mod_cons.html

Reference

-  Bour, Frédéric, Basile Clément, and Gabriel Scherer. "Tail modulo cons." *arXiv preprint arXiv:2102.09823* (2021).
-  Bagrel, Thomas. "Destination-passing style programming: a Haskell implementation." *arXiv preprint arXiv:2312.11257* (2023).
-  Leijen, Daan, and Anton Lorenzen. "Tail Recursion Modulo Context: An Equational Approach." *Proceedings of the ACM on Programming Languages* 7, no. POPL (2023): 1152-1181.

TMC変換

$$\begin{array}{c}
 \frac{}{\Box \rightsquigarrow_{direct} \Box} \qquad \frac{U \rightsquigarrow_{direct} E}{\text{let } x = e \text{ in } U \rightsquigarrow_{direct} (\text{let } x = e \text{ in } E)} \\
 \\
 \frac{\forall j, U_j \rightsquigarrow_{direct} E_j}{\text{match } e \text{ with } (p_j \rightarrow U_j)^j \rightsquigarrow_{direct} (\text{match } e \text{ with } (p_j \rightarrow E_j)^j)} \\
 \\
 \frac{n = |I| + 1 \quad d.n \leftarrow U \rightsquigarrow_{dps} T[d_l.n_l \leftarrow \Box]^l}{\text{let } d = K((e_i)^{i \in I}, \text{Hole}, (e_j)^j) \text{ in } K((e_i)^{i \in I}, U, (e_j)^j) \rightsquigarrow_{direct} \text{let } d = K((e_i)^{i \in I}, \text{Hole}, (e_j)^j) \text{ in } T[d_l.n_l \leftarrow \Box]^l; d}
 \end{array}$$

equationでdpsの呼び出しに変換

$$\begin{array}{c}
 \frac{}{d.n \leftarrow \Box \rightsquigarrow_{dps} \Box[d.n \leftarrow \Box]} \qquad \frac{d.n \leftarrow U \rightsquigarrow_{dps} T[d_i.n_i \leftarrow \Box]^i}{d.n \leftarrow \text{let } x = e \text{ in } U \rightsquigarrow_{dps} \text{let } x = e \text{ in } T[d_i.n_i \leftarrow \Box]^i} \\
 \\
 \frac{\forall j, d.n \leftarrow U_j \rightsquigarrow_{dps} T_j[d_{i_j}.n_{i_j} \leftarrow \Box]^{i_j}}{d.n \leftarrow \text{match } e \text{ with } (p_j \rightarrow U_j)^j \rightsquigarrow_{dps} \text{match } e \text{ with } (p_j \rightarrow T_j[d_{i_j}.n_{i_j} \leftarrow \Box]^{i_j})^j} \\
 \\
 \frac{n' = |I| + 1 \quad d'.n' \leftarrow U \rightsquigarrow_{dps} T[d'_l.n'_l \leftarrow \Box]^l}{\text{let } d' = K((e_i)^{i \in I}, \text{Hole}, (e_j)^j) \text{ in } d.n \leftarrow d'; \text{let } d' = K((e_i)^{i \in I}, \text{Hole}, (e_j)^j) \text{ in } T[d'_l.n'_l \leftarrow \Box]^l}
 \end{array}$$

equationで再帰呼び出しに変換

末尾呼び出し最適化

Q. なぜ**末尾再帰**は**performant**なんですか？

A. **末尾呼び出し最適化**ができるから

Q. 末尾呼び出し最適化とはなんですか？

A. 再帰呼び出し(**call**)を**jump**にする(続)

末尾呼び出し最適化

再帰呼び出し(call)をjumpにする

```
fold_left:
```

```
.....
```

```
call fold_left
```

後続の処理がない!

```
fold_left:
```

```
.....
```

```
jmp fold_left
```

エントリの先頭まで*jmp*すれば
よくないですか?