限定継続を使った計算効果プログラミング

叢 悠悠（東京工業大学）

PPL サマースクール 2022

自己紹介

経歴：お茶大附属幼稚園 → お茶大大学院 → 東工大

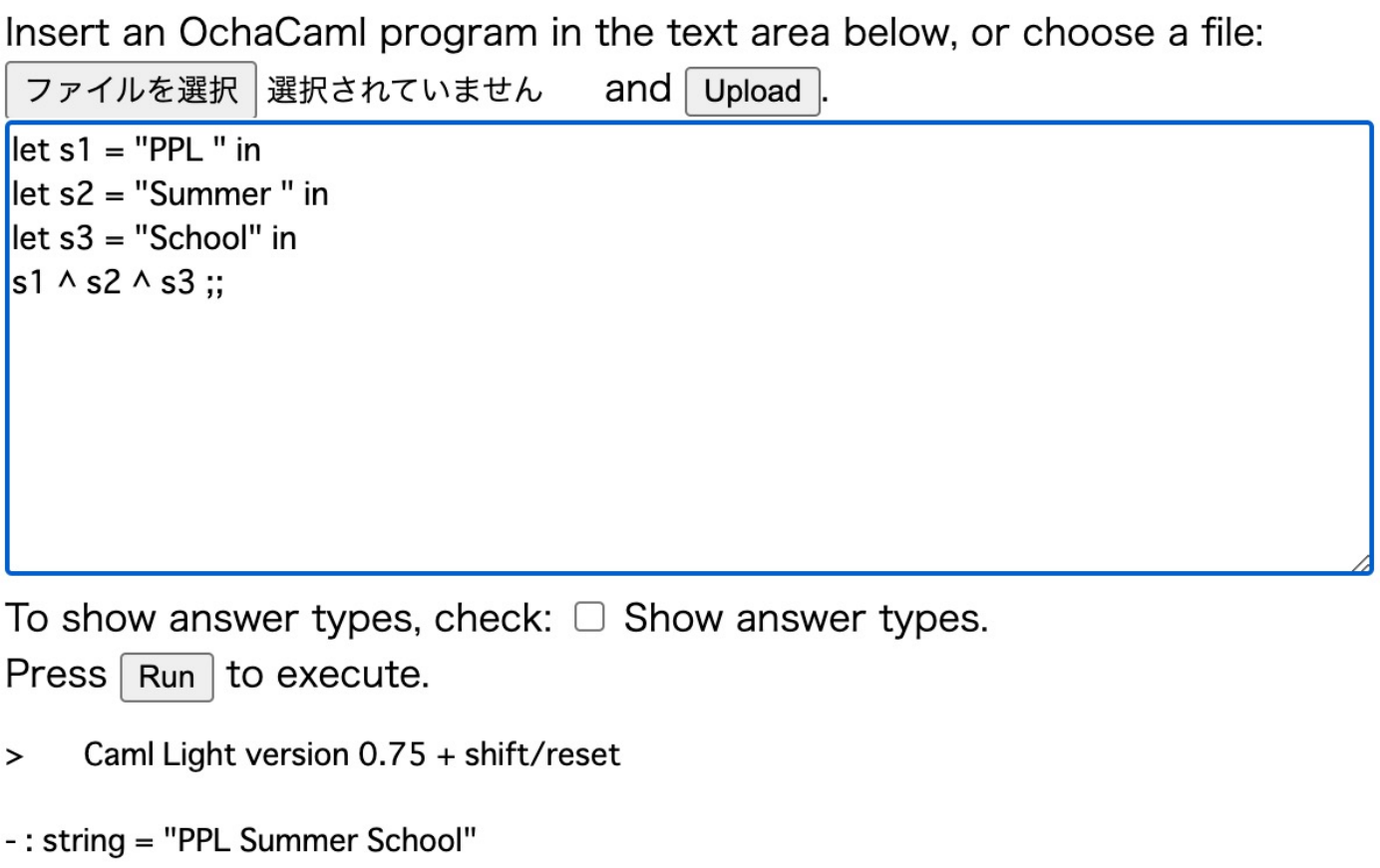
興味：継続、型システム、自然言語意味論、音楽理論、
プログラミング教育

目標と流れ

継続の基本的な考え方・使い方を習得する

- 継続と制御演算子 shift/reset
- shift/reset の応用例
- いろいろな制御機構

使用言語：OchaCaml



http://pllab.is.ocha.ac.jp/~asai/OchaCaml/demo/

継続 = 残りの計算

計算の中の継続

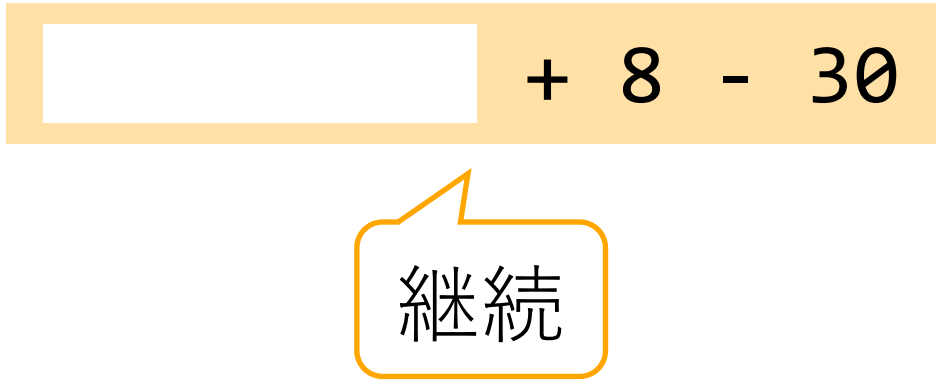
```
(20 * 22) + 8 - 30
```

計算の中の継続

```
(20 * 22) + 8 - 30
```

実行中の式

計算の中の継続



計算の中の継続

```
fun x -> x + 8 - 30
```

計算の中の継続

```
λx. x + 8 - 30
```

計算の中の継続

```
“JSSST ” ^ string_of_int 2022
```

実行中の式

注：^ は文字列をつなげる演算子、string_of_int は整数を文字列に変換する関数

計算の中の継続

```
“JSSST ” ^
```

継続

計算の中の継続

λx. “JSSST ” ^ x

継続

計算の中の継続

```
if y >= 0 then y else -y
```

実行中の式

計算の中の継続

```
if          then y else -y
```

継続

計算の中の継続

```
λx. if x then y else -y
```

継続

計算の中の継続

```
print_int (string_length (“Ocha” ^ “Caml”))
```

実行中の式

注：print_int は整数を出力する関数、string_length は文字列の長さを返す関数

計算の中の継続

```
print_int (string_length                    )
```

継続

計算の中の継続

```
λx. print_int (string_length x)
```

計算の中の継続

```
factorial 10
```

実行中の式

計算の中の継続

計算の中の継続

$\lambda x.\ x$

継続の操作 = 実行の制御

例：times

```
(* リストに含まれる整数の積を求める *)
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs -> x * times xs
```

times の振る舞い

```
  times [4; 0; 2]
= 4 * times [0; 2]
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

times の振る舞い（理想）

```
  times [4; 0; 2]
= 4 * times [0; 2]
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

0 発見

times の振る舞い（理想）

```
  times [4; 0; 2]
= 4 * times [0; 2]
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

スキップ

times の振る舞い（理想）

```
  times [4; 0; 2]
= 4 * times [0; 2]
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

最終結果

times の改良：条件文の導入

```
  times [4; 0; 2]
= 4 * times [0; 2]
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

```
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0
    then 0
    else x * times xs
```

times の改良：条件文の導入

```
  times [4; 0; 2]
= 4 * times [0; 2]
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

```
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0
    then 0
    else x * times xs
```

times の改良：条件文の導入

```
  times [4; 0; 2]
= 4 * times [0; 2]
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

```
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0
    then 0
    else x * times xs
```

やりたいこと：継続の破棄

```
  times [4; 0; 2]
= 4 * times [0; 2]
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

```
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0
    then ...
    else x * times xs
```

やりたいこと：継続の破棄

```
  times [4; 0; 2]
= 4 *
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

```
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0
    then ...
    else x * times xs
```

やりたいこと：継続の破棄

```
  times [4; 0; 2]
= 4 *
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

```
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0
    then drop    ; 0
    else x * times xs
```

times の改良：制御演算子の導入

```
  times [4; 0; 2]
= 4 *
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

```
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0
    then C (fun k -> 0)
    else x * times xs
```

Felleisen の C

times の改良：制御演算子の導入

```
  times [4; 0; 2]
= 4 *
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

```
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0
    then C (fun k -> 0)
    else x * times xs
```

継続

times の改良：制御演算子の導入

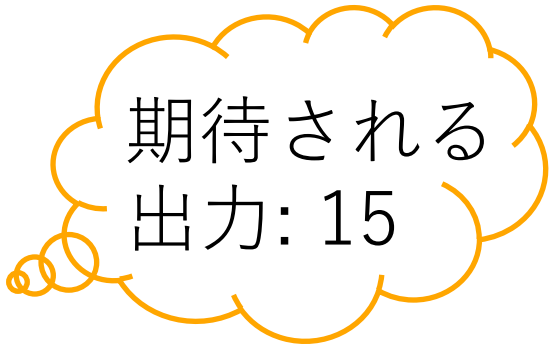
```
  times [4; 0; 2]
= 4 *
= 4 * 0 * times [2]
= 4 * 0 * 2 * times []
= 4 * 0 * 2 * 1
= 0
```

```
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0
    then C (fun k -> 0)
    else x * times xs
```

継続を使った計算

一般の継続の不便さ

```
times [4; 0; 2] + times [1; 3; 5]
```



一般の継続の不便さ

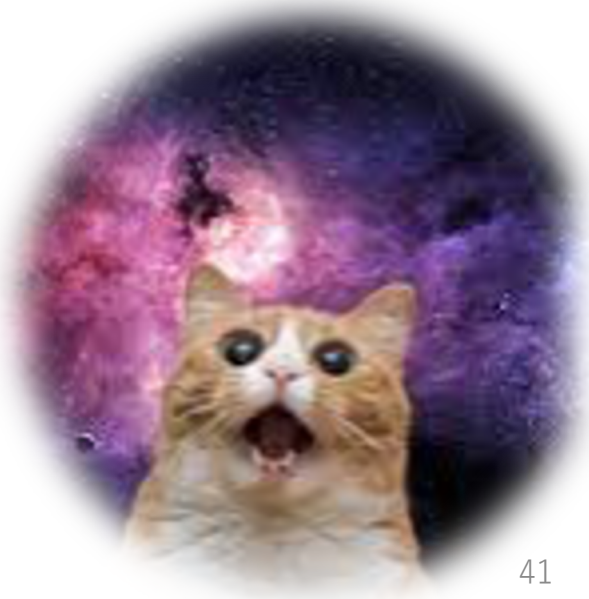
```
  times [4; 0; 2] + times [1; 3; 5]
= 4 * C (fun k -> 0) + times [1; 3; 5]
```

一般の継続の不便さ

```
  times [4; 0; 2] + times [1; 3; 5]
= 4 *                 + times [1; 3; 5]
```

一般の継続の不便さ

```
  times [4; 0; 2] + times [1; 3; 5]
= 4 * C (fun k -> 0) + times [1; 3; 5]
= 0
```



やりたいこと：継続の範囲の制限

```
  times [4; 0; 2] + times [1; 3; 5]
= < 4 * C (fun k -> 0) > + times [1; 3; 5]
= 15
```

限定継続 = 範囲が限定された継続

ステップ１：C から shift への置き換え

```
(* lst 中の整数の積を求める *)
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0 then C (fun k -> 0)
    else x * times xs
```

ステップ１：C から shift への置き換え

```
(* lst 中の整数の積を求める *)
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0 then shift (fun k -> 0)
    else x * times xs
```

ステップ１：C から shift への置き換え

```
(* lst 中の整数の積を求める *)
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0 then shift (fun k -> 0)
    else x * times xs
```

継続

ステップ１：C から shift への置き換え

```
(* lst 中の整数の積を求める *)
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0 then shift (fun k -> 0)
    else x * times xs
```

継続を使った計算

ステップ２：reset の導入

```
reset (fun () -> times [4; 0; 2]) +
reset (fun () -> times [1; 3; 5])
```

ステップ２：reset の導入

```
    reset (fun () -> times [4; 0; 2]) +
    reset (fun () -> times [1; 3; 5])
  = reset (fun () -> 4 * shift (fun k -> 0)) +
    reset (fun () -> times [1; 3; 5])
```

ステップ２：reset の導入

```
    reset (fun () -> times [4; 0; 2]) +
    reset (fun () -> times [1; 3; 5])
  = reset (fun () -> 4 *                ) +
    reset (fun () -> times [1; 3; 5])
```

ステップ２：reset の導入

```
    reset (fun () -> times [4; 0; 2]) +
    reset (fun () -> times [1; 3; 5])
  = reset (fun () -> 4 * shift (fun k -> 0)) +
    reset (fun () -> times [1; 3; 5])
  = 0 + 15
  = 15
```

計算の中の限定継続

```
reset (fun () -> (20 * 22) + 8) - 30
```

計算の中の限定継続

```
reset (fun () -> (20 * 22) + 8) - 30
```

計算の中の限定継続

```
reset (fun () ->          + 8) - 30
```

限定継続

計算の中の限定継続

```
if reset (fun () -> y >= 0) then y else -y
```

計算の中の限定継続

```
if reset (fun () ->          ) then y else -y
```

限定継続

計算の中の限定継続

```
print_int (
 reset (fun () -> string_length (“Ocha” ^ “Caml”)))
```

実行中の式

計算の中の限定継続

```
print_int (
 reset (fun () -> string_length                    ))
```

限定継続

継続と限定継続のまとめ

継続：　　… `C (fun k -> e)` …

限定継続：… `reset (fun () ->` … `shift (fun k -> e)` …) …

ワタシケイゾク
チョットワカル

継続の使い方

破棄

複製

中断

入れ替え

継続の破棄 = 例外の発生

```
(* lst 中の整数の積を求める *)
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0 then shift (fun k -> 0)
    else x * times xs ;;

reset (fun () -> times [4; 0; 2]) ;;
```

継続の破棄 = 例外の発生

```
(* lst 中の整数の積を求める *)
let rec times lst =
  match lst with
    [] -> 1
  | x :: xs ->
    if x = 0 then raise (Error 0)
    else x * times xs ;;

try times [4; 0; 2] with Error x -> x ;;
```

問題：sum

整数のリストを受け取り、すべての要素の和を返す関数 sum を定義せよ。ただし、リストに１つでも負の整数が含まれていた場合は 0 を返すものとする。

```
reset (fun () -> sum [1; 2; 3]) ;;
- : int = 6
reset (fun () -> sum [4; -5; 6]) ;;
- : int = 0
```

問題：sum

整数のリストを受け取り、すべての要素の和を返す関数 sum を定義せよ。ただし、リストに１つでも負の整数が含まれていた場合は 0 を返すものとする。

```
reset (fun () -> sum [1; 2; 3]) ;;
- : int = 6
reset (fun () -> sum [4; -5; 6]) ;;
- : int = 0
```

問題：sum

整数のリストを受け取り、すべての要素の和を返す関数 sum を定義せよ。ただし、リストに１つでも負の整数が含まれていた場合は 0 を返すものとする。

```
reset (fun () -> sum [1; 2; 3]) ;;
- : int = 6
reset (fun () -> sum [4; -5; 6]) ;;
- : int = 0
```

解答

```
let rec sum lst =
  match lst with
    [] -> 0
  | x :: xs ->
    if x < 0 then shift (fun k -> 0)
    else x + sum xs ;;
```

sum の振る舞い

```
< sum [4; -5; 6] >
```

sum の振る舞い

```
  < sum [4; -5; 6] >
= < 4 + sum [-5; 6] >
= < 4 + S k. 0 >
```

shift

sum の振る舞い

```
  < sum [4; -5; 6] >
= < 4 + sum [-5; 6] >
= < 4 +          >
```

sum の振る舞い

```
  < sum [4; -5; 6] >
= < 4 + sum [-5; 6] >
= < 4 + S k. 0 >
= 0
```

継続の使い方

破棄

複製

中断

入れ替え

例：either

```
let x = either 1 2 in
if x mod 2 = 0 then print_int x ;;

2- : unit = ()
```

例：either

```
let x = either 1 2 in
if x mod 2 = 0 then print_int x ;;

2- : unit = ()
```

非決定的

例：either

```
let x =  in
if x mod 2 = 0 then print_int x ;;

2- : unit = ()
```

例：either

```
let x =      1      in
if x mod 2 = 0 then print_int x ;;

2- : unit = ()
```

例：either

```
let x =      2      in
if x mod 2 = 0 then print_int x ;;

2- : unit = ()
```

shift/reset による either の実装

```
let either x y =
  shift (fun k -> k x; k y) ;;

reset (fun () ->
  let x = either 1 2 in
  if x mod 2 = 0 then print_int x) ;;
```

shift/reset による either の実装

```
let either x y =
  shift (fun k -> k x; k y) ;;

reset (fun () ->
  let x = either 1 2 in
  if x mod 2 = 0 then print_int x) ;;
```

shift/reset による either の実装

```
let either x y =
  shift (fun k -> k x; k y) ;;

reset (fun () ->
  let x = either 1 2 in
  if x mod 2 = 0 then print_int x) ;;
```

shift/reset による either の実装

```
let either x y =
  shift (fun k -> k x; k y) ;;

reset (fun () ->
  let x = either 1 2 in
  if x mod 2 = 0 then print_int x) ;;
```

either の振る舞い

```
  < let x = either 1 2 in
    if x mod 2 = 0 then print_int x >
= < let x = S k. k 1; k 2 in
    if x mod 2 = 0 then print_int x >
```

either の振る舞い

```
  < let x = either 1 2 in
    if x mod 2 = 0 then print_int x >
= < let x =                        in
    if x mod 2 = 0 then print_int x >
```

either の振る舞い

```
  < let x = either 1 2 in
    if x mod 2 = 0 then print_int x >
= < let x = S k. k 1; k 2 in
    if x mod 2 = 0 then print_int x >
= < if 1 mod 2 = 0 then print_int 1;
    if 2 mod 2 = 0 then print_int 2 >
= ()
```

問題：flip

自身を取り囲む計算を true と false で 1 回ずつ実行し、
その結果をペアとして返す関数 flip を定義せよ。

```
reset (fun () -> if flip () then 1 else 0) ;;
- : int * int = (1, 0)
```

解答

```
let flip () =
  shift (fun k -> (k true, k false)) ;;
```

flip の振る舞い

```
  < if flip () then 1 else 0 >

= < if (S k. (k true, k false)) then 1 else 0 >
```

flip の振る舞い

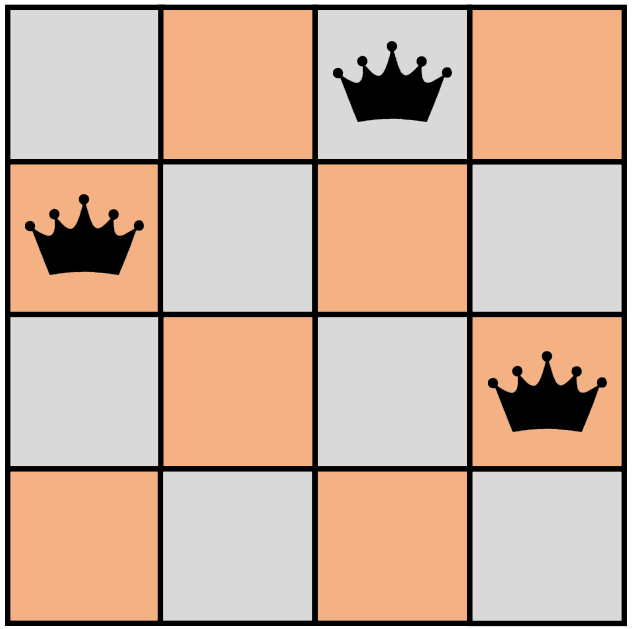
```
  < if flip () then 1 else 0 >

= < if                        then 1 else 0 >
```

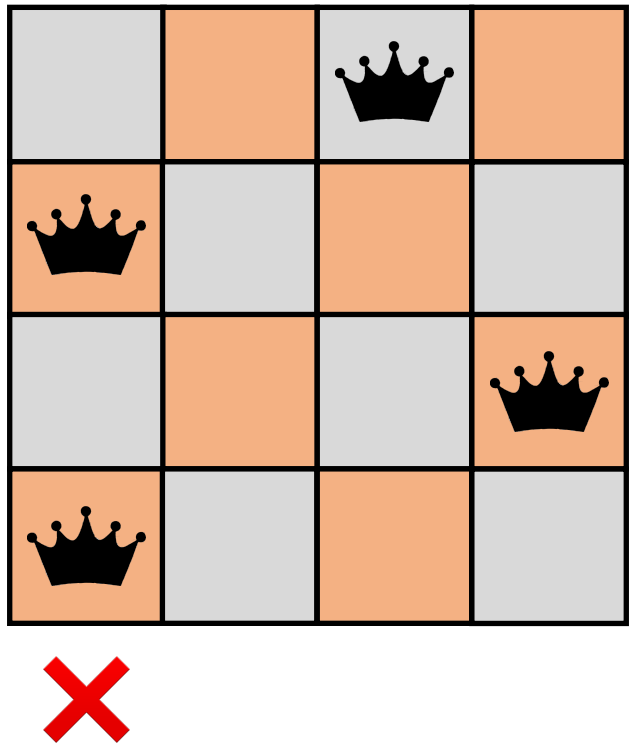
flip の振る舞い

```
  < if flip () then 1 else 0 >
= < if (S k. (k true, k false)) then 1 else 0 >
= (if true then 1 else 0, if false then 1 else 0)
= (1, 0)
```

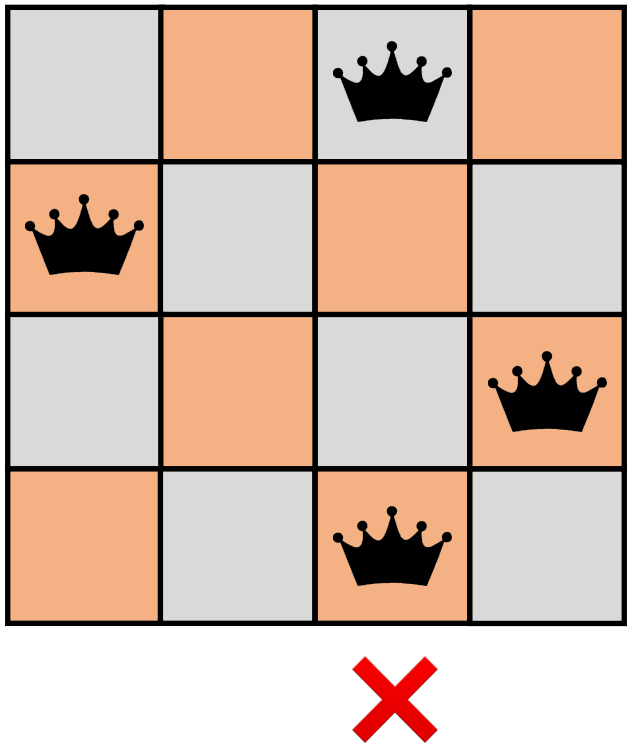
継続の複製の応用：バックトラック



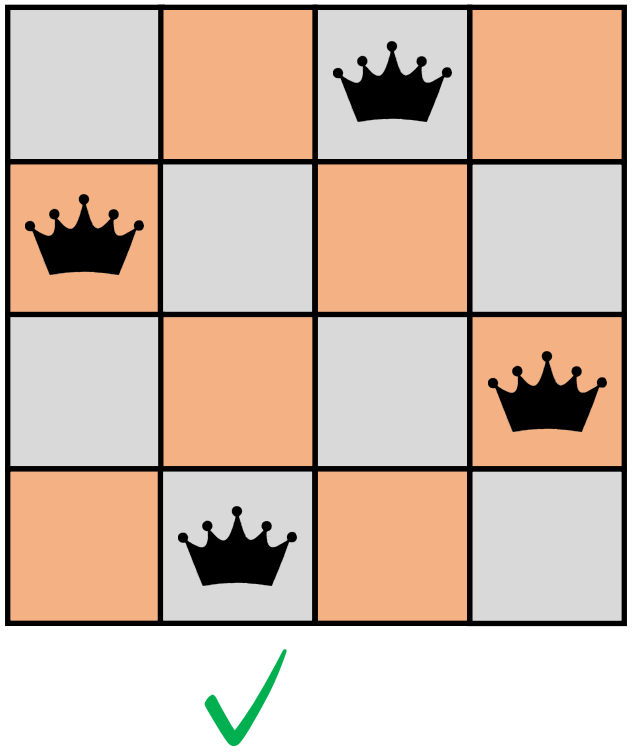
継続の複製の応用：バックトラック



継続の複製の応用：バックトラック

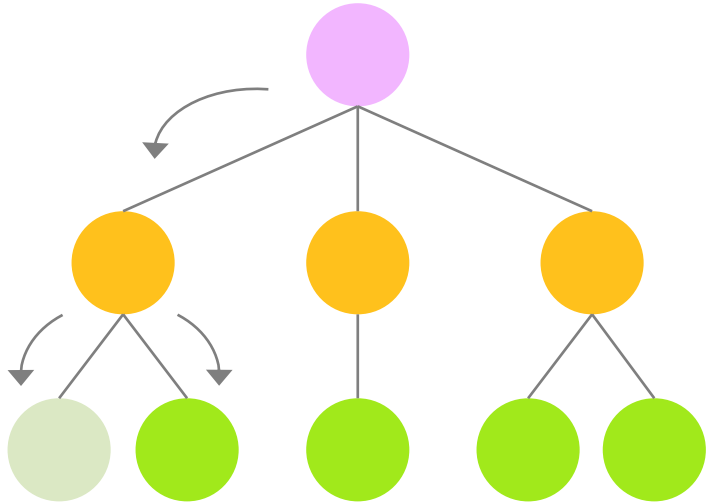


継続の複製の応用：バックトラック



探索ライブラリ [Kaneko and Asai '11]

- OchaCaml の `shift/reset` を使用
- 深さ / 幅優先、 α β 法など をサポート



継続の使い方

破棄

複製

中断

入れ替え

walk（その１）

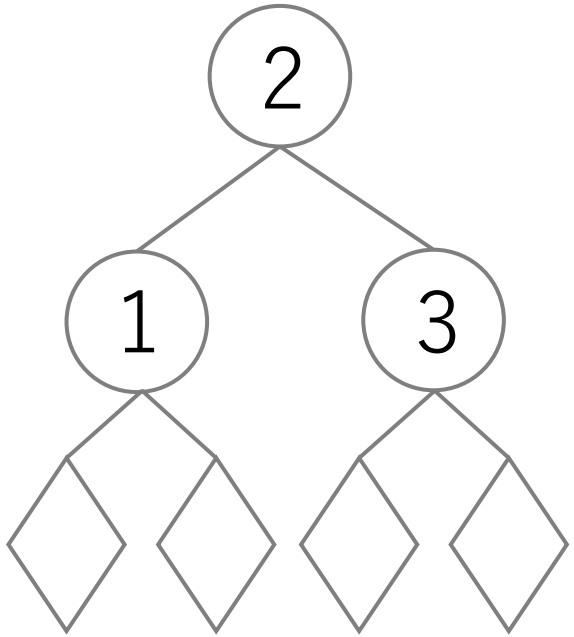
```
type tree_t =
  Leaf
| Node of tree_t * int * tree_t ;;

let rec walk tree = match tree with
    Leaf -> ()
  | Node (t1, n, t2) ->
     walk t1; print_int n; walk t2 ;;
```

walk の振る舞い

```
let tree1 =
  Node (Node (Leaf, 1, Leaf),
        2,
        Node (Leaf, 3, Leaf)) ;;

walk tree1 ;;
123- : unit = ()
```



tree1

walk（その２）

```
type tree_t =
  Leaf
| Node of tree_t * int * tree_t ;;

let rec walk tree = match tree with
    Leaf -> ()
  | Node (t1, n, t2) ->
     walk t1; print_int n; walk t2 ;;
```

walk（その２）

```
type tree_t =
  Leaf
| Node of tree_t * int * tree_t ;;

let rec walk tree = match tree with
    Leaf -> ()
  | Node (t1, n, t2) ->
     walk t1; yield n; walk t2 ;;
```

yield

```
type 'a result_t =
  Done
| Next of int * unit / 'a -> 'a result_t / 'a ;;

let yield n = shift (fun k -> Next (n, k)) ;;
```

yield

```
type 'a result_t =
  Done
| Next of int * unit / 'a -> 'a result_t / 'a ;;

let yield n = shift (fun k -> Next (n, k)) ;;
```

計算終了

yield

```
type 'a result_t =
  Done
| Next of int * unit / 'a -> 'a result_t / 'a ;;

let yield n = shift (fun k -> Next (n, k)) ;;
```

残りの計算あり

yield

```
type 'a result_t =
  Done
| Next of int * unit / 'a -> 'a result_t / 'a ;;

let yield n = shift (fun k -> Next (n, k)) ;;
```

継続

yield

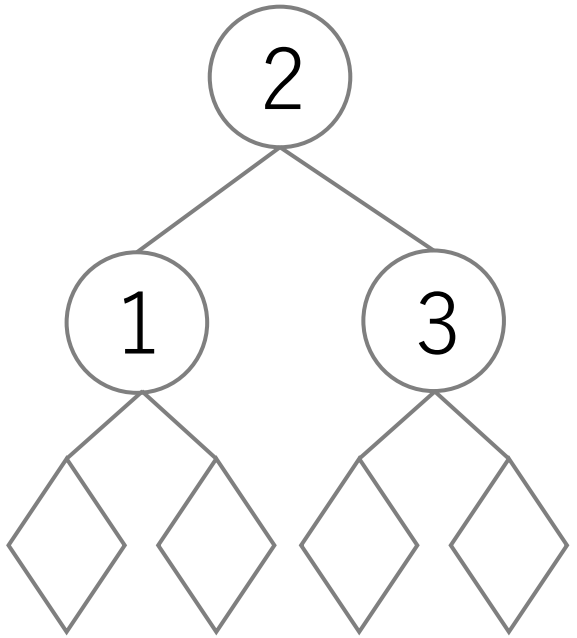
```
type 'a result_t =
  Done
| Next of int * unit / 'a -> 'a result_t / 'a ;;

let yield n = shift (fun k -> Next (n, k)) ;;
```

start

```
let start tree =
  reset (fun () -> walk tree; Done) ;;

start tree1 ;;
- : '_a result_t = Next (1, <fun>)
```



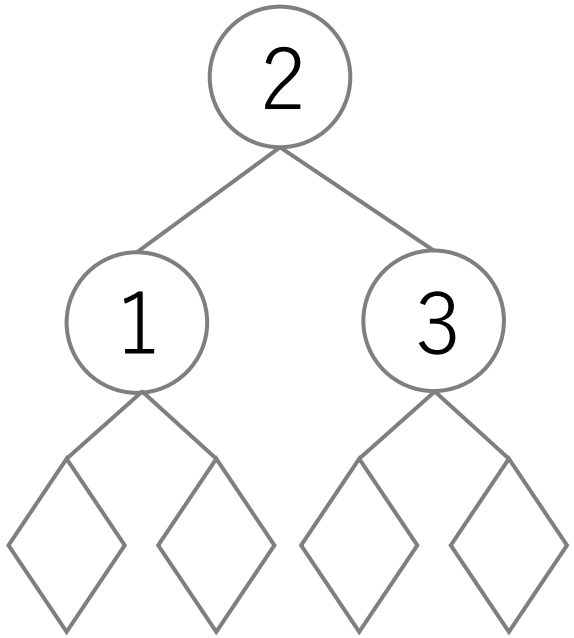
tree1

start

```
let start tree =
  reset (fun () -> walk tree; Done) ;;

start tree1 ;;
- : '_a result_t = Next (1, <fun>)
```

継続



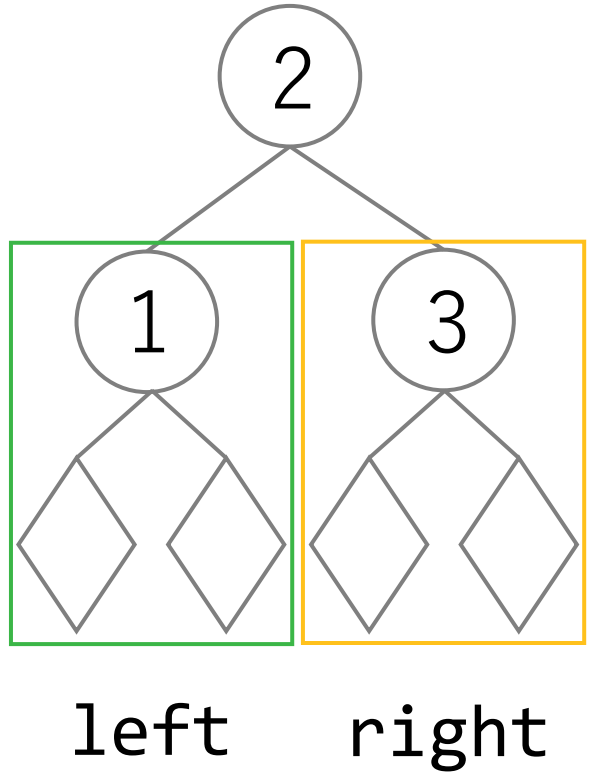
tree1

print_nodes

```
let print_nodes tree =
  let rec loop r =
    match r with
      Done -> ()
    | Next (n, k) ->
      print_int n; loop (k ()) in
  loop (start tree) ;;
```

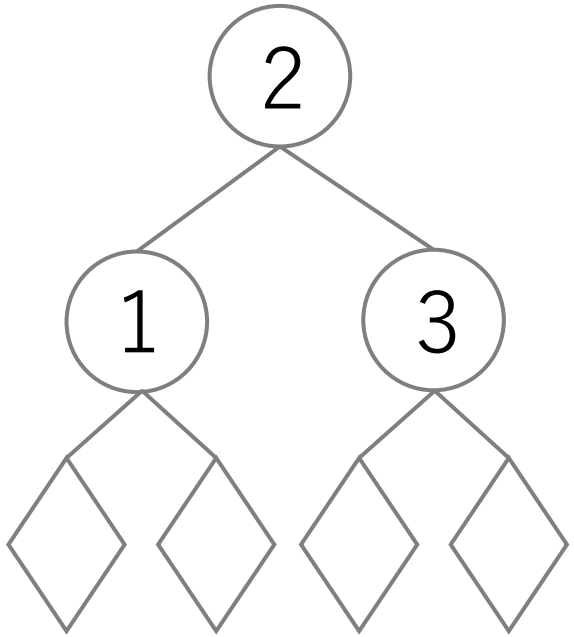
print_nodes の振る舞い

```
  print_nodes tree1
= loop < walk left; yield 2;
         walk right; Done >
```



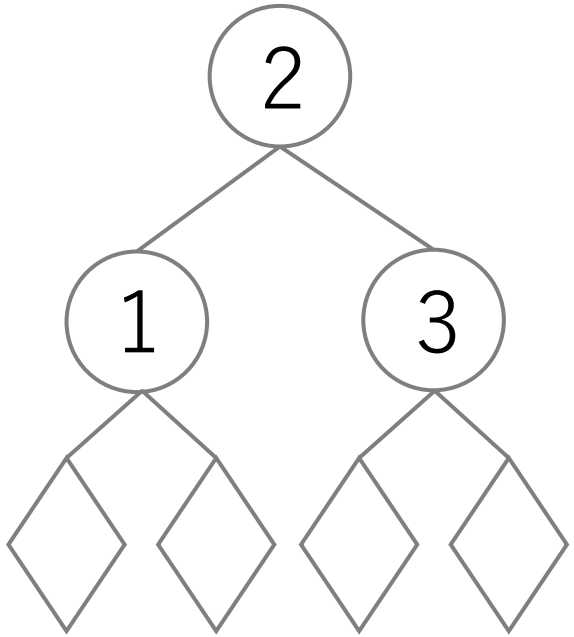
print_nodes の振る舞い

```
  print_nodes tree1
= loop < walk left; yield 2;
         walk right; Done >
= loop < S k. Next (1, k); yield 2;
         walk right; Done >
```



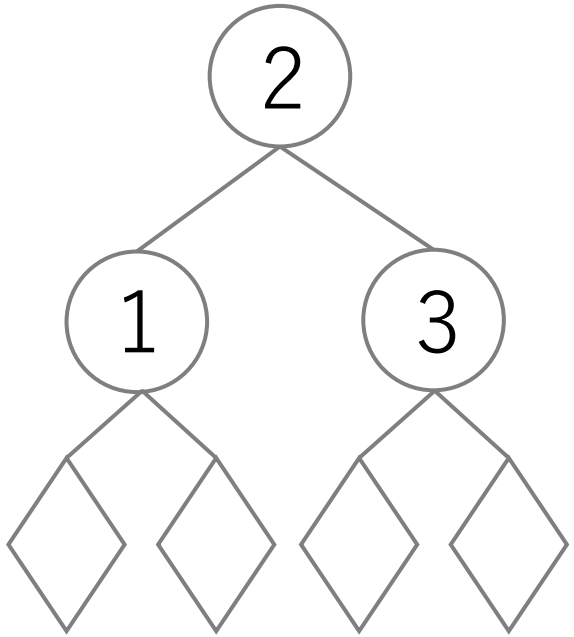
print_nodes の振る舞い

```
  print_nodes tree1
= loop < walk left; yield 2;
         walk right; Done >
= loop <                    ; yield 2;
         walk right; Done >
```



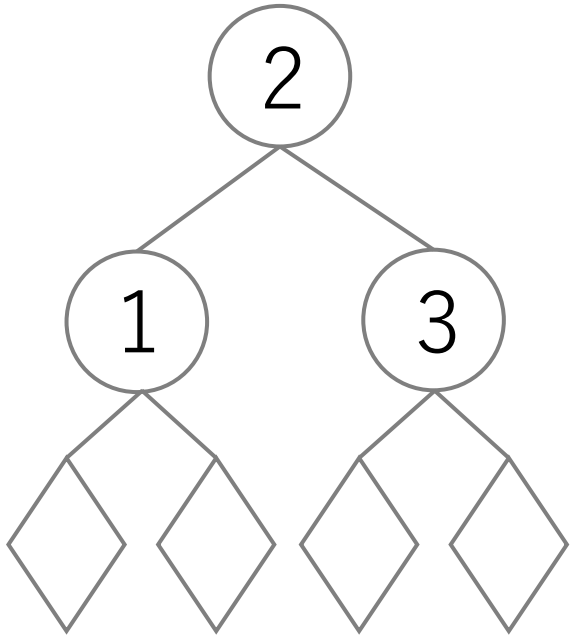
print_nodes の振る舞い

```
  print_nodes tree1
= loop < walk left; yield 2;
         walk right; Done >
= loop < S k. Next (1, k); yield 2;
         walk right; Done >
= loop (Next (1, k))
```



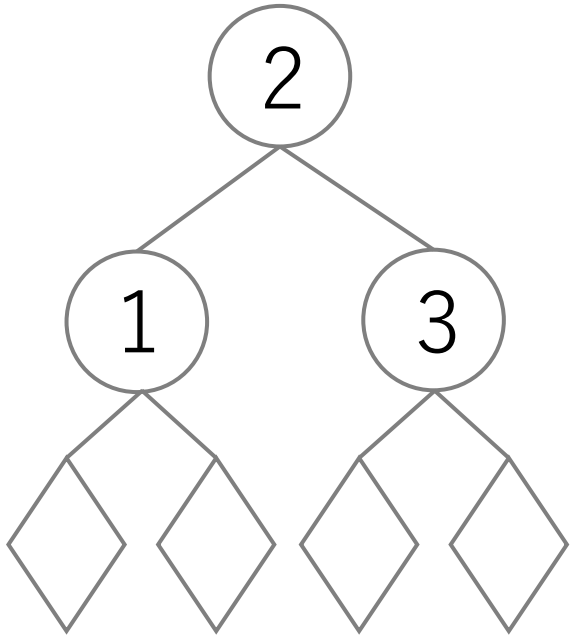
print_nodes の振る舞い

```
  print_nodes tree1
= loop < walk left; yield 2;
         walk right; Done >
= loop < S k. Next (1, k); yield 2;
         walk right; Done >
= loop (Next (1, k))
= print_int 1; loop (k ())
```



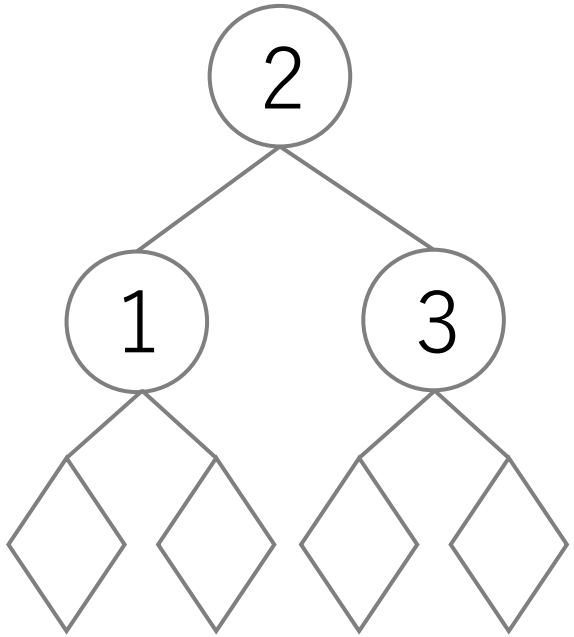
print_nodes の振る舞い

```
  print_nodes tree1
= loop < walk left; yield 2;
         walk right; Done >
= loop < S k. Next (1, k); yield 2;
         walk right; Done >
= loop (Next (1, k))
= print_int 1; loop (k ())
= loop < yield 2; walk right; Done >
```



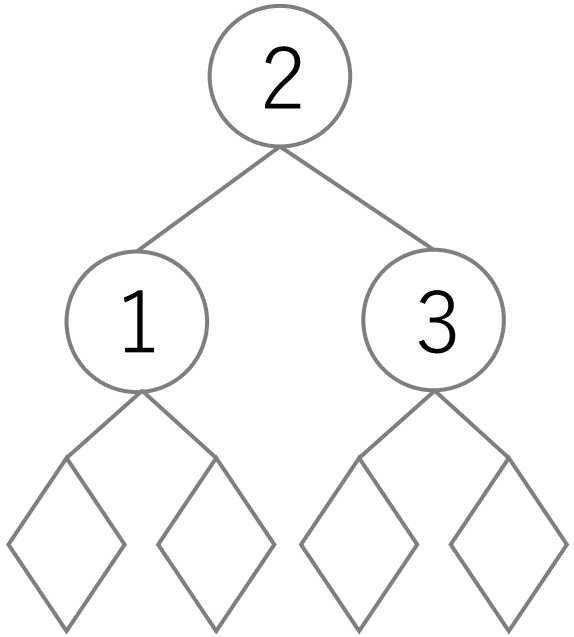
print_nodes の振る舞い

```
  print_nodes tree1
= loop < walk left; yield 2;
         walk right; Done >
= loop < S k. Next (1, k); yield 2;
         walk right; Done >
= loop (Next (1, k))
= print_int 1; loop (k ())
= loop <          ; walk right; Done >
```



print_nodes の振る舞い

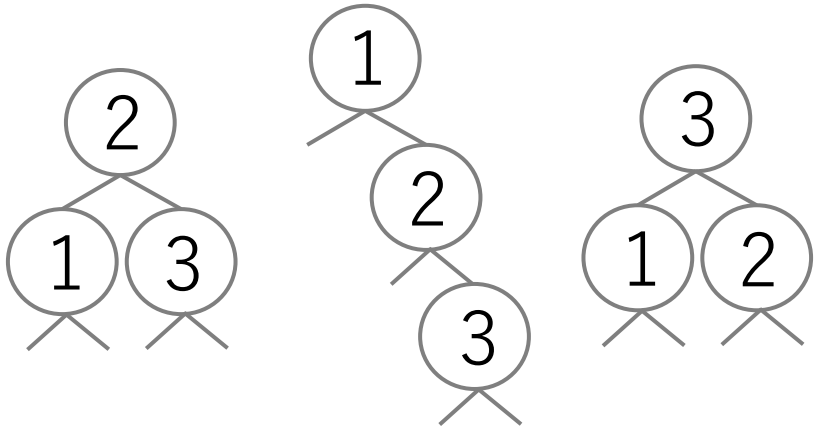
```
  print_nodes tree1
= loop < walk left; yield 2;
         walk right; Done >
= loop < S k. Next (1, k); yield 2;
         walk right; Done >
= loop (Next (1, k))
= print_int 1; loop (k ())
= loop < yield 2; walk right; Done >
= ()
```



問題：same_fringe

２つの木を受け取り、数字の並びが同じかどうかを判定する関数 same_fringe を定義せよ。ただし、走査は左から深さ優先で行う。

```
same_fringe tree1 tree2 ;;
- : bool = true
same_fringe tree1 tree3 ;;
- : bool = false
```



tree1　tree2　tree3

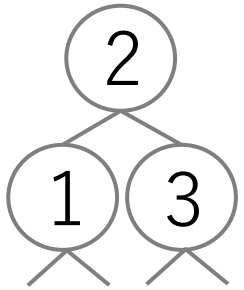
解答

```
let same_fringe tree1 tree2 =
  let rec loop r1 r2 = match (r1, r2) with
      (Done, Done) -> true
    | (Next (n1, k1), Next (n2, k2)) ->
       if n1 = n2 then loop (k1 ()) (k2 ())
       else false
    | _ -> false in
  loop (start tree1) (start tree2) ;;
```

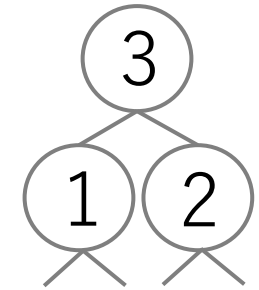
same_fringe の振る舞い

```
  same_fringe tree1 tree3

= loop (start tree1) (start tree3)
```



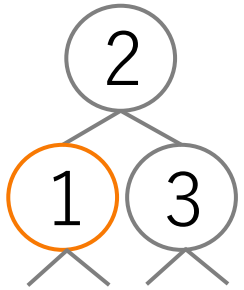
tree1



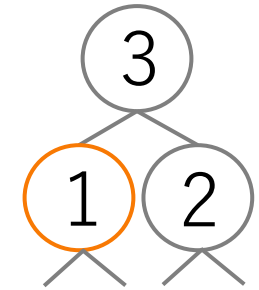
tree3

same_fringe の振る舞い

```
  same_fringe tree1 tree3
= loop (start tree1) (start tree3)
= loop (Next (1, k1)) (Next (1, k3))
```



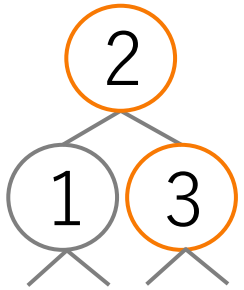
tree1



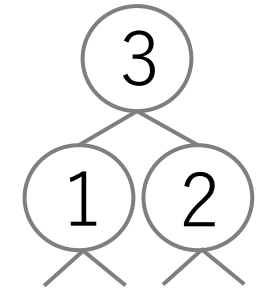
tree3

same_fringe の振る舞い

```
  same_fringe tree1 tree3
= loop (start tree1) (start tree3)
= loop (Next (1, k1)) (Next (1, k3))
```



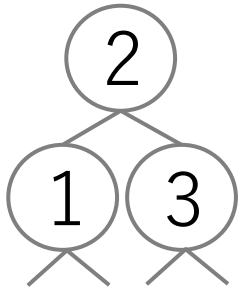
tree1



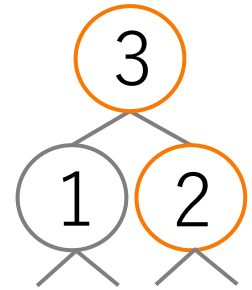
tree3

same_fringe の振る舞い

```
  same_fringe tree1 tree3
= loop (start tree1) (start tree3)
= loop (Next (1, k1)) (Next (1, k3))
```



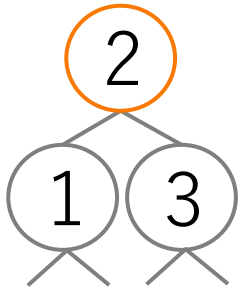
tree1



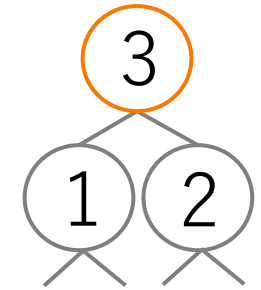
tree3

same_fringe の振る舞い

```
  same_fringe tree1 tree3
= loop (start tree1) (start tree3)
= loop (Next (1, k1)) (Next (1, k3))
= loop (Next (2, k1)) (Next (3, k3))
```



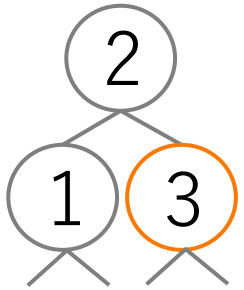
tree1



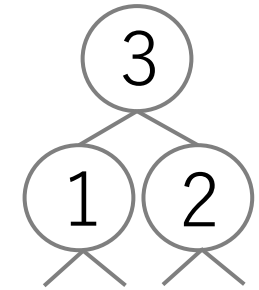
tree3

same_fringe の振る舞い

```
  same_fringe tree1 tree3
= loop (start tree1) (start tree3)
= loop (Next (1, k1)) (Next (1, k3))
= loop (Next (2, k1)) (Next (3, k3))
```



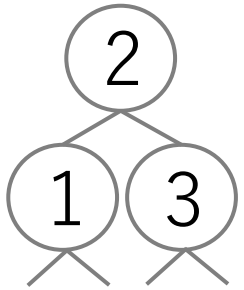
tree1



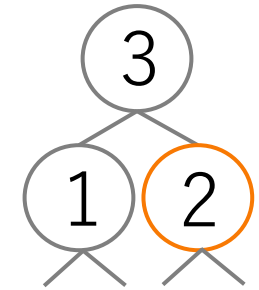
tree3

same_fringe の振る舞い

```
  same_fringe tree1 tree3
= loop (start tree1) (start tree3)
= loop (Next (1, k1)) (Next (1, k3))
= loop (Next (2, k1)) (Next (3, k3))
```



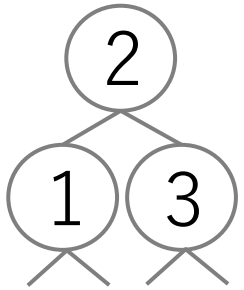
tree1



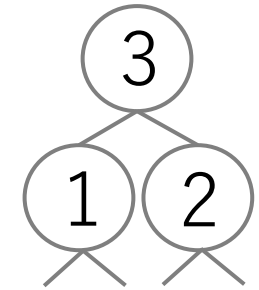
tree3

same_fringe の振る舞い

```
  same_fringe tree1 tree3
= loop (start tree1) (start tree3)
= loop (Next (1, k1)) (Next (1, k3))
= loop (Next (2, k1)) (Next (3, k3))
= false
```

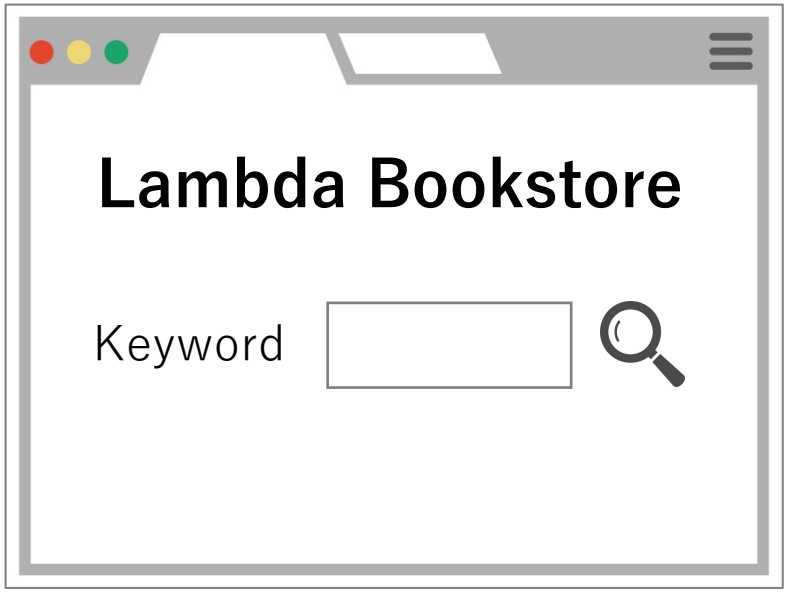


tree1



tree3

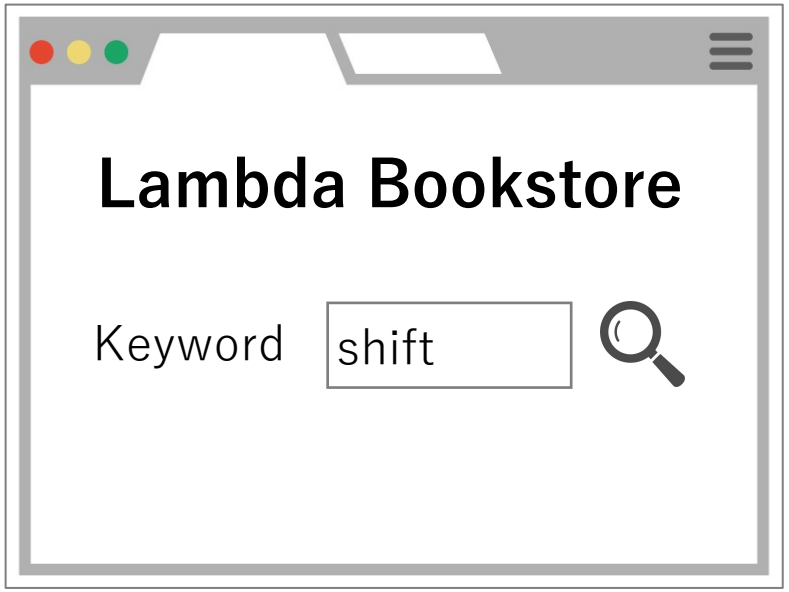
継続の中断の応用：Web プログラミング



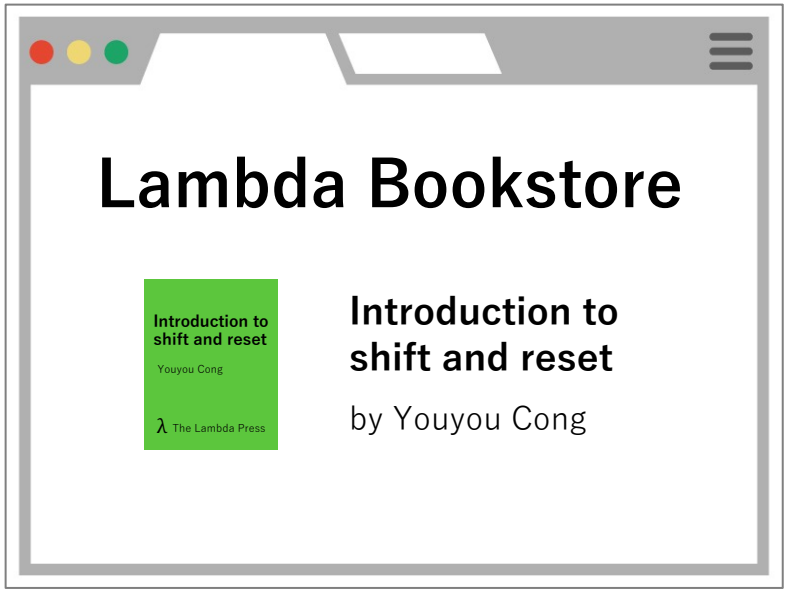
continuation name = k42

* 架空の書店です。

継続の中断の応用：Web プログラミング



継続の中断の応用：Web プログラミング



http://lambdabookstore/resume?continuation=k42&keyword=shift

* 架空の本です。

web-server [Krishnamurthi et al. ‘07]

- Racket の Web プログラミング用領域特化言語
- Racket の制御演算子を使用
- 実用例：学会運営システム

継続の使い方

破棄

複製

中断

入れ替え

例：リストの反転

```
let reverse lst =
  let rec reverse' lst acc =
    match lst with
      [] -> acc
    | x :: xs -> reverse' xs (x :: acc)
  in reverse' lst [] ;;
```

shift/reset によるリストの反転の実装

```
let reverse lst =
  let rec reverse' lst =
    match lst with
      [] -> []
    | x :: xs ->
      shift (fun k -> x :: k (reverse' xs))
  in reset (fun () -> reverse' lst) ;;
```

shift/reset によるリストの反転の実装

```
let reverse lst =
  let rec reverse' lst =
    match lst with
      [] -> []
    | x :: xs ->
      shift (fun k -> x :: k (reverse' xs))
  in reset (fun () -> reverse' lst) ;;
```

shift/reset によるリストの反転の実装

```
let reverse lst =
  let rec reverse' lst =
    match lst with
      [] -> []
    | x :: xs ->
      shift (fun k -> x :: k (reverse' xs))
  in reset (fun () -> reverse' lst) ;;
```

非末尾呼び出し

reverse の振る舞い

```
reverse [1; 2; 3]
```

reverse の振る舞い

```
  reverse [1; 2; 3]
= < S k1. 1 :: k1 (reverse' [2; 3]) >
```

reverse の振る舞い

```
  reverse [1; 2; 3]
= <                                  >
```

reverse の振る舞い

```
  reverse [1; 2; 3]
= < S k1. 1 :: k1 (reverse' [2; 3]) >
= < 1 :: k1 (S k2. 2 :: k2 (reverse' [3])) >
```

reverse の振る舞い

```
  reverse [1; 2; 3]
= < S k1. 1 :: k1 (reverse' [2; 3]) >
= < 1 :: k1                         >
```

reverse の振る舞い

```
  reverse [1; 2; 3]
= < S k1. 1 :: k1 (reverse' [2; 3]) >
= < 1 :: k1 (S k2. 2 :: k2 (reverse' [3])) >
= < 2 :: k2 (S k3. 3 :: k3 (reverse' [])) >
```

reverse の振る舞い

```
  reverse [1; 2; 3]
= < S k1. 1 :: k1 (reverse' [2; 3]) >
= < 1 :: k1 (S k2. 2 :: k2 (reverse' [3])) >
= < 2 :: k2                                  >
```

reverse の振る舞い

```
  reverse [1; 2; 3]
= < S k1. 1 :: k1 (reverse' [2; 3]) >
= < 1 :: k1 (S k2. 2 :: k2 (reverse' [3])) >
= < 2 :: k2 (S k3. 3 :: k3 (reverse' [])) >
= < 3 :: k3 [] >
```

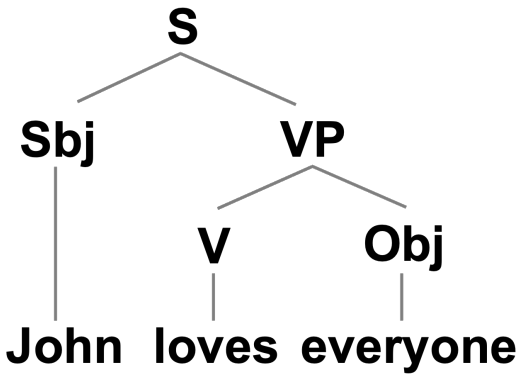
reverse の振る舞い

```
  reverse [1; 2; 3]
= < S k1. 1 :: k1 (reverse' [2; 3]) >
= < 1 :: k1 (S k2. 2 :: k2 (reverse' [3])) >
= < 1 :: k1 (S k2. 2 :: k2 (reverse' [])) >
= < 3 :: k3 [] >
= < 3 :: < 2 :: < 1 :: < [] > > > >
= [3; 2; 1]
```

問題：everyone

自然言語の文の意味を文字列として表現したい。
“everyone” の意味を表す項を定義せよ。

```
let john = “john” ;;
let love obj sbj =
  “love(” ^ sbj ^ “, ” ^ obj ^ “)” ;;
let everyone x = … ;;
```




```
reset (fun () -> love (everyone “x”) john) ;;
- : string = “forall x. love(john, x)”
```

ヒント

```
let everyone x =
  shift (fun k -> ... k x) ;;
```

< love (everyone “x”) john > の場合、

```
k      = love                  john
k “x” = love(john, x)
```

解答

```
let everyone x =
  shift (fun k -> “forall ” ^ x ^ “. ” ^ k x) ;;
```

everyone の振る舞い

```
  < love (everyone “x”) john >
= < love (S k. “forall ” ^ “x” ^ “. ” ^ k “x”) john >
```

everyone の振る舞い

< love (everyone “x”) john >

= < love [redacted] john >

everyone の振る舞い

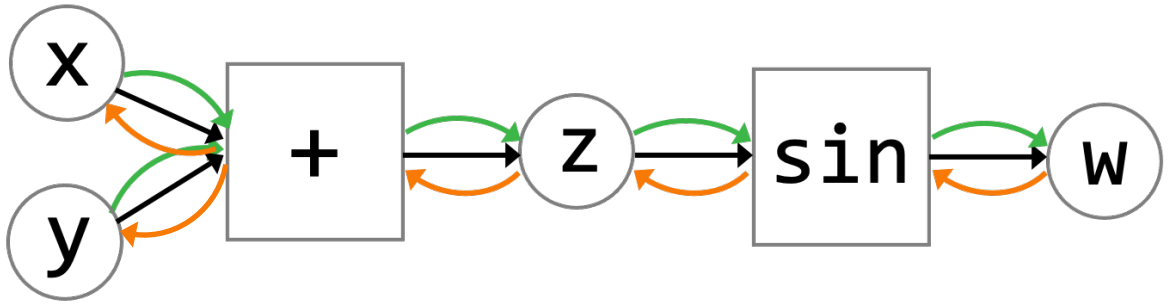
```
  < love (everyone “x”) john >
= < love (S k. “forall ” ^ “x” ^ “. ” ^ k “x”) john >
= “forall ” ^ x ^ “. ” ^ “love(john, x)”
= “forall x. love(john, x)”
```

順序の入れ替えの応用：自動微分

$\frac{d}{dx}$`sin(x + y) = ?`

```
let z = x + y in
let w = sin z in
w
```

1. 順伝播 (値を計算)



2. 逆伝播 (微分を計算)

Lantern [Wang et al. ‘19]

- Scalaの機械学習フレームワーク
- `shift/reset` を使用
- SqueezeNet, ResNet, TreeLSTM などの例で既存フレームワークと同等の性能

継続の使い方のまとめ

破棄
(例外)

複製
(探索)

中断
(Web)

入れ替え
(自動微分)

継続
完全に理解した

さまざまな制御演算子

継続

- C
- call/cc

限定継続

- shift/reset
- control/prompt
- shift0/reset0
- control0/prompt0

制御演算子を提供する言語

代数的効果とハンドラ [Plotkin & Pretnar '13]

- 例外機構の一般化
- ハンドラ内で継続の操作が可能

```
exception MyErr of unit

try 2 * MyErr () with
MyErr () -> 42
```

```
effect MyErr : unit -> int

handle (2 * MyErr ()) with
MyErr () k -> 42
```

代数的効果とハンドラ [Plotkin & Pretnar '13]

- 例外機構の一般化
- ハンドラ内で継続の操作が可能

```
exception MyErr of unit

try 2 * MyErr () with
MyErr () -> 42
```

```
effect MyErr : unit -> int

handle (2 * MyErr ()) with
MyErr () k -> 42
```

制御演算子 vs. 代数的効果

```
let MyErr () =
  shift (fun k -> 42)

reset (fun () ->
  2 * MyErr ())
```

例外を起こす側が
解釈を規定

```
effect MyErr : unit -> int

handle (2 * MyErr ())
with MyErr () k -> 42
```

例外を処理する側が
解釈を規定

まとめ

- 継続 = 残りの計算
- 実行の制御に有用
- 制御演算子や代数的効果・ハンドラで操作可能

```
reset (fun () ->
  “Thank ” ^
  shift (fun k ->
    k “you” ^ “!”)
```

自習用リンク集

- shift/reset プログラミング入門（浅井 & Kiselyov）
  http://pllab.is.ocha.ac.jp/~asai/cw2011tutorial/main-j.pdf
- An Introduction to Algebraic Effects and Handlers (Pretnar)
  https://www.eff-lang.org/handlers-tutorial.pdf
- Effects Bibliography
  https://github.com/yallop/effects-bibliography