

algebraic effect handlers
algebraic effects

を含むプログラムのステップ実行

PPL2020 Session 12: 代数的効果
お茶の水女子大学 浅井研究室 古川つきの

概要

algebraic effect handlers を含む言語のステッパを実装した

その過程で、

- algebraic effect handlers の big-step の CPS インタプリタを定義した
- インタプリタにプログラム変換を施すことでコンテキスト情報を導出した

もくじ

1. algebraic effect handlers
2. ステッパ
3. プログラム変換によらないステッパ関数の作り方
4. プログラム変換によるステッパ関数の作り方

もくじ

1. algebraic effect handlers
2. ステッパ
3. プログラム変換によらないステッパ関数の作り方
4. プログラム変換によるステッパ関数の作り方

algebraic effect handlers

- ・ プログラミング言語機能
- ・ 副作用を「オペレーション」の呼び出しで表現する
- ・ オペレーションを呼び出すと「エフェクト」が実行される
- ・ エフェクトは「ハンドラ」で定義する

algebraic effect handlers の構文

(型無し λ 計算 + algebraic effect handlers)

値 $v := x \mid \text{fun } x \rightarrow e$

式 $e := v \mid e \ e \mid \text{op } e \mid \text{with } h \ \text{handle } e$

ハンドラ $h := \{\text{return } x \rightarrow e;$
 $\quad \text{op}(x; k) \rightarrow e; \dots; \text{op}(x; k) \rightarrow e\}$

algebraic effect handlers の構文

(型無し λ 計算 + algebraic effect handlers)

値 $v := x \mid \text{fun } x \rightarrow e$

オペレーション呼び出し

ハンドラでハンドルして実行

式 $e := v \mid e \ e \mid \text{op } e \mid \text{with } h \text{ handle } e$

ハンドルした部分が正常終了した時に実行する式 (省略可)

ハンドラ $h := \{\text{return } x \rightarrow e;$
 $\text{op}(x; k) \rightarrow e; \dots; \text{op}(x; k) \rightarrow e\}$

エフェクト定義

algebraic effect handlers の意味

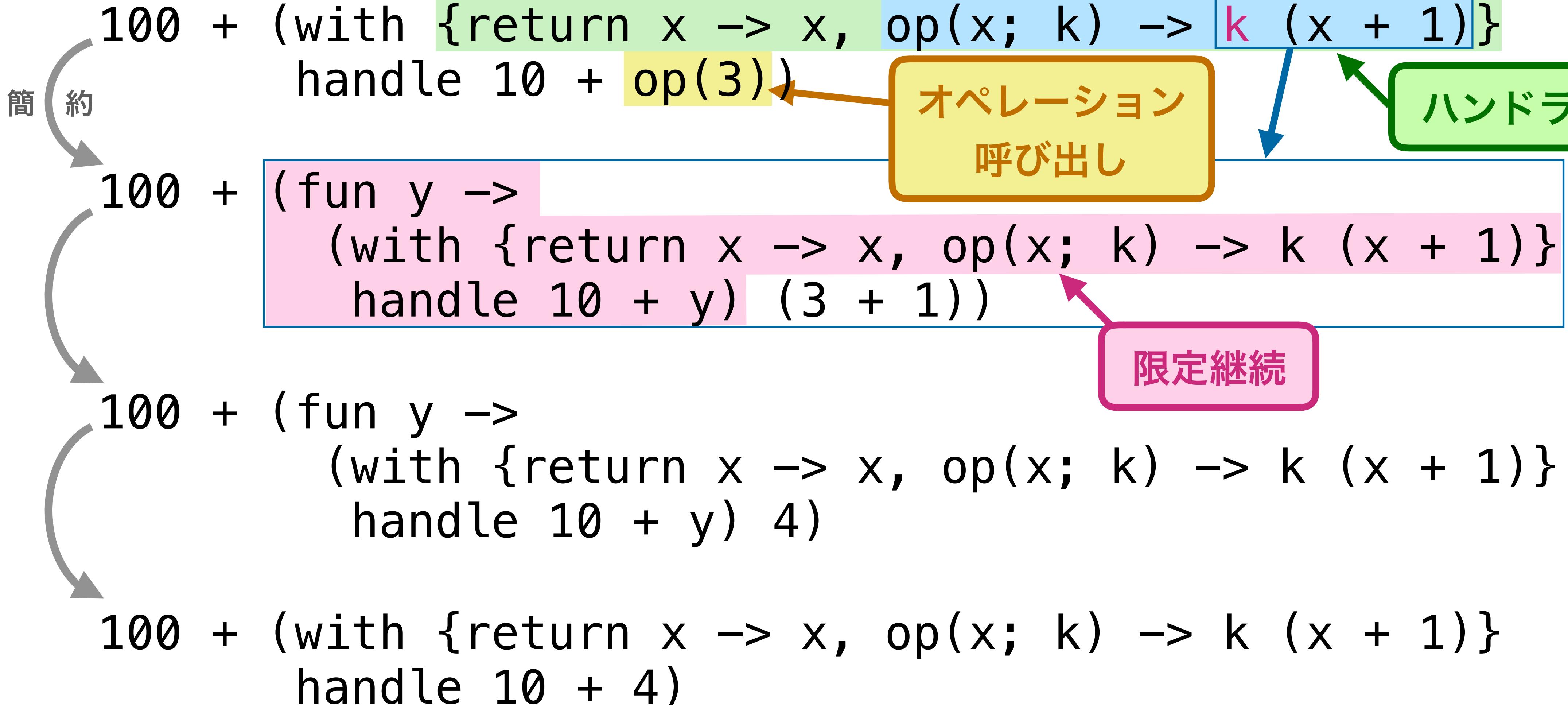
with {op (x; k) -> 式1} handle 式2

式2を実行中に (op 引数) が出てきたら、

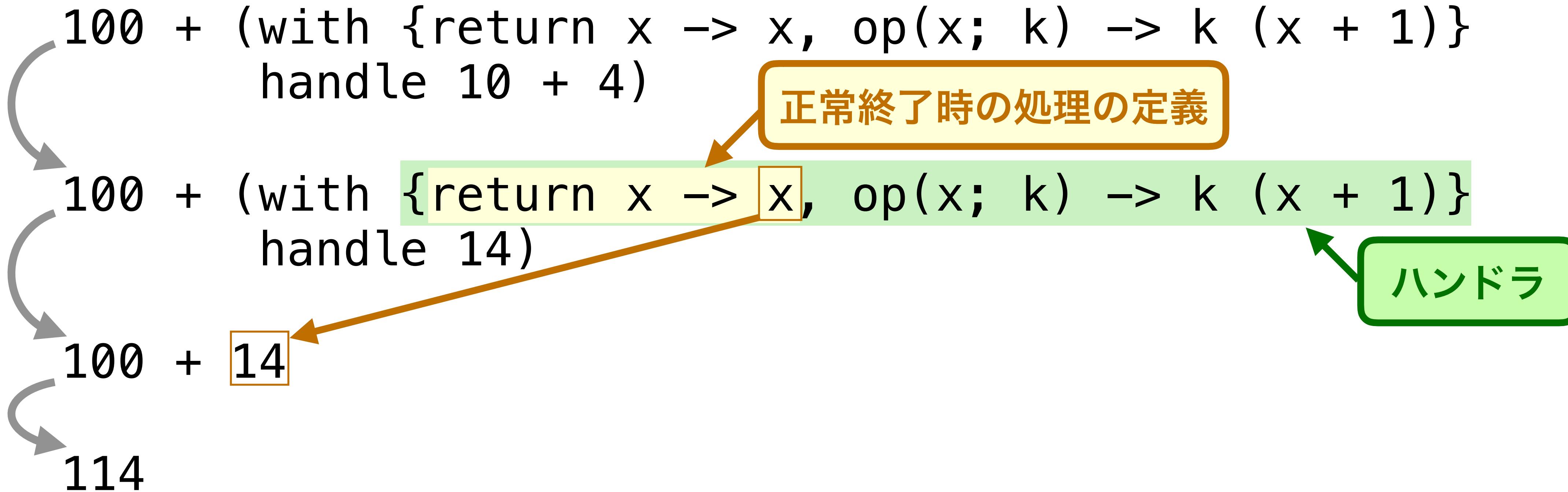
その時の with handle までの限定継続を k、
引数を x として式1の実行に移る

プログラム例

エフェクト定義



プログラム例



本研究のインタプリタ

型無し λ 計算 + algebraic effect handlers に対する
big-step インタプリタを定義した

既存のインタプリタ：

- small-step インタプリタ[1][2]
 - 繙続をフレームのリストとして保持する big-step インタプリタ[3][4]
- 高階の継続を直接使う big-step インタプリタを定義した

インタプリタ（言語定義）

```
(* λ計算 と algebraic effect handlers の言語 *)
(* 値 *)
type v = Var of string          (* x *)
        | Fun of string * e   (* fun x -> e *)
        | Cont of (k -> k)    (* 繙続 fun x => ... *)
(* ハンドラ *)
and h = {return : string * e;           (* handler {return x -> e, ...} *)
        ops : (string * string * string * e) list} (* op(x; k) -> e, ... *)
(* 式 *)
and e = Val of v                  (* v *)
        | App of e * e        (* e e *)
        | Op of string * e   (* op e *)
        | With of h * e      (* with h handle e *)
(* handle 内の継続 *)
and k = v -> a
(* handle 内の実行結果 *)
and a = Return of v              (* 値になった *)
        | OpCall of string * v * k (* オペレーションが呼び出された *)
```

インタプリタ（本体）

```
(* CPS インタプリタ *)
let rec eval (exp : e) (k : k) : a = match exp with
| Val (v) -> k v (* 繼続に値を渡す *)
| App (e1, e2) ->
  eval e2 (fun v2 ->
    eval e1 (fun v1 -> match v1 with
      | Fun (x, e) ->
        let reduct = subst e [(x, v2)] in (* e[v2/x] *)
        eval reduct k
      | Cont (cont_value) -> (cont_value k) v2
        (* 現在の継続と継続値が保持するメタ継続を合成して値を渡す *)
        | _ -> failwith "type error"))
    )
| Op (name, e) ->
  eval e (fun v -> OpCall (name, v, fun v -> k v))
| With (h, e) ->
  let a = eval e (fun v -> Return v) in (* 空の継続で handle 節内を実行 *)
  apply_handler k h a (* handle 節内の実行結果をハンドラで処理 *)
```

インタプリタ（ハンドラ処理）

```
(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) : a = match a with
| Return v -> (* handle 節内が値 v を返したとき *)
  (match h with {return = (x, e)} -> (* {return x -> e; ...} として *)
    let reduct = subst e [(x, v)] in (* e[v/x] に簡約される *)
    eval reduct k) (* e[v/x] を実行 *)
| OpCall (name, v, m) -> (* オペレーション呼び出しがあったとき *)
  (match search_op name h with
  | None -> (* ハンドラで定義されていない場合、 *)
    OpCall (name, v, (fun v ->
      let a' = m v in
      apply_handler k h a')))
  | Some (x, y, e) -> (* ハンドラで定義されている場合、 *)
    let cont_value =
      Cont (fun k'' -> fun v -> (* 適用時にその後の継続を受け取って合成 *)
        let a' = m v in
        apply_handler k'' h a') in
    let reduct = subst e [(x, v); (y, cont_value)] in
    eval reduct k)
```

定義したインタプリタ

- 初期継続を渡して実行を開始する `eval e (fun v -> Return v)`
- 環境を使わず代入に基づくインタプリタにした
 - 表示のために特定の式にマークを付ける等の目的
 - `handle` 節内の実行について CPS になっている

もくじ

1. algebraic effect handlers
2. ステッパ
3. プログラム変換によらないステッパ関数の作り方
4. プログラム変換によるステッパ関数の作り方

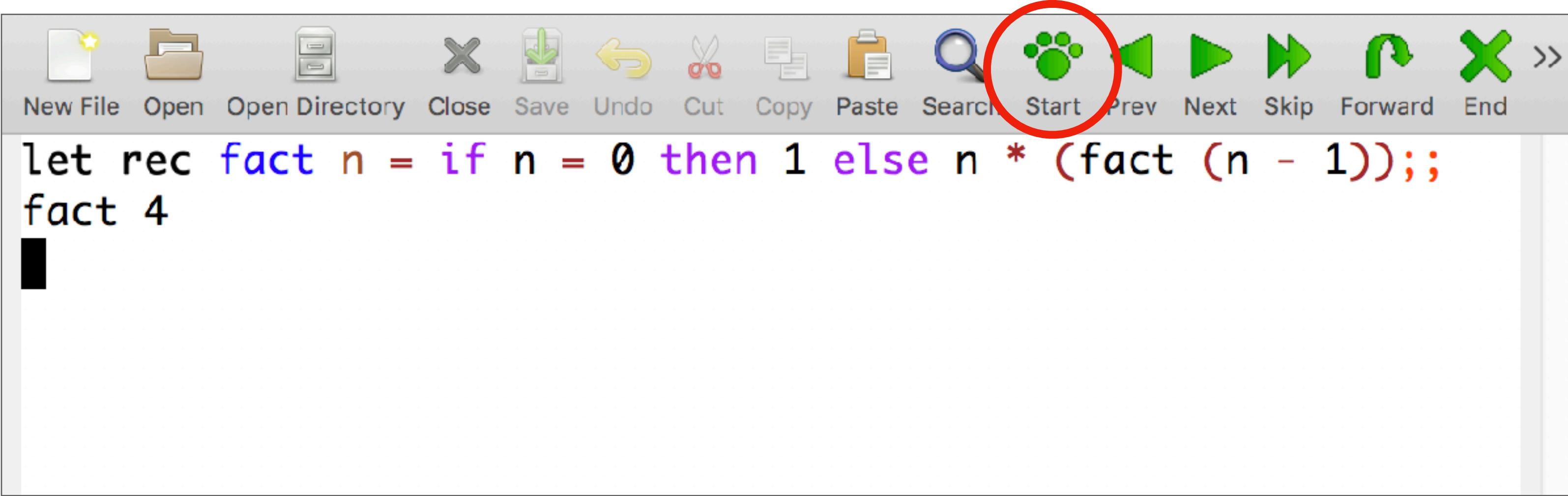
ステッパ

- ・ プログラムが簡約されていくのを観察するツール
- ・ プログラミング学習やデバッグのために使う
- ・ これまでに作られたステッパの対象：
 - ・ Racket の教育用の一部の構文[5]
 - ・ OCaml[6]
 - ・ 再帰を含む一部の構文[7]
 - ・ try-with を含む一部の構文[8]

A screenshot of a code editor interface. The menu bar at the top contains the following items: New File, Open, Open Directory, Close, Save, Undo, Cut, Copy, Paste, Search, Start, Prev, Next, Skip, Forward, End, and a double-right arrow icon. Below the menu bar, there is a code editor window containing the following F# code:

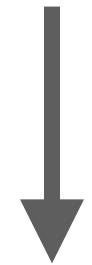
```
let rec fact n = if n = 0 then 1 else n * (fact (n - 1));;
fact 4
```

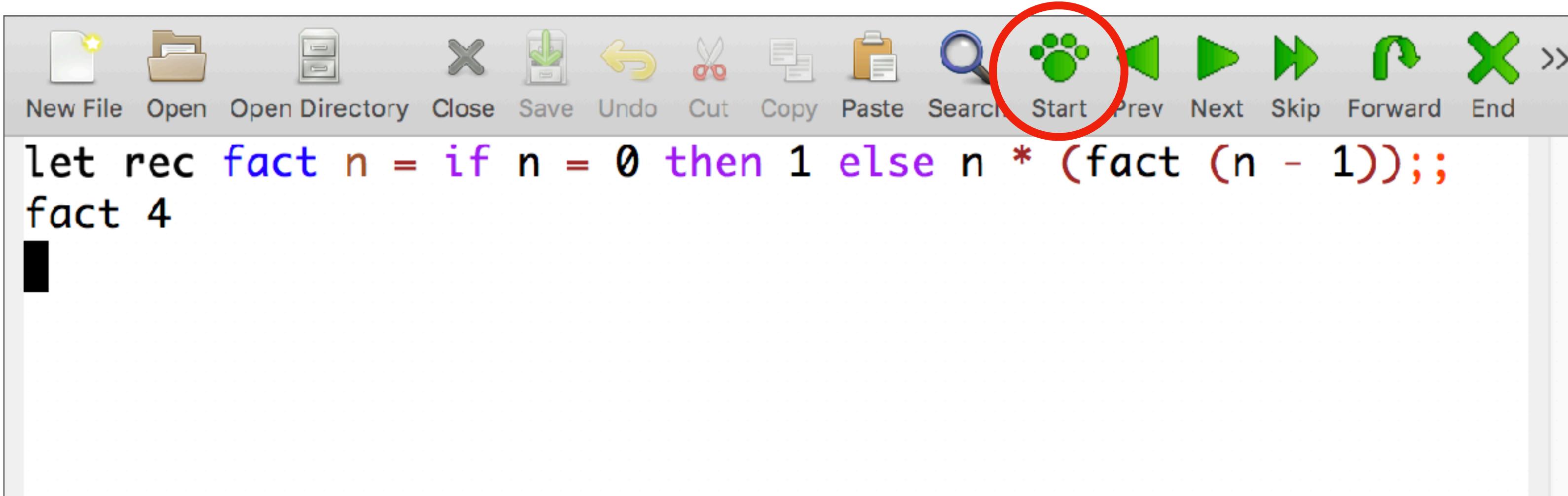
The code consists of two lines. The first line defines a recursive function named `fact` that takes an integer `n` and returns the factorial of `n`. It uses an `if` statement to handle the base case where `n` is 0, returning 1. Otherwise, it returns `n` multiplied by the result of calling `fact` with `n - 1`. The second line calls this function with the argument 4. A small black square cursor is located at the end of the second line.



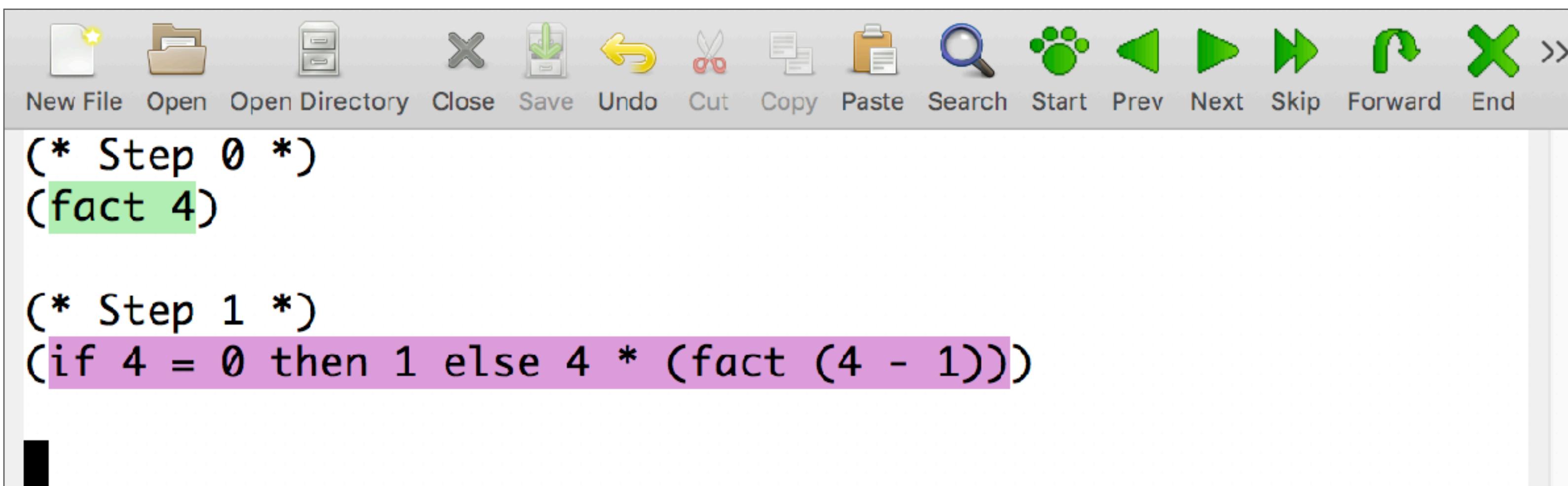
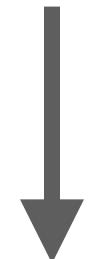
The screenshot shows a software interface with a toolbar at the top containing various icons for file operations like New File, Open, Save, and Paste, along with navigation buttons for Start, Prev, Next, Skip, Forward, and End. A red circle highlights the 'Start' button icon, which is a green paw print. Below the toolbar is a code editor window displaying the following F# code:

```
let rec fact n = if n = 0 then 1 else n * (fact (n - 1));;
fact 4
```



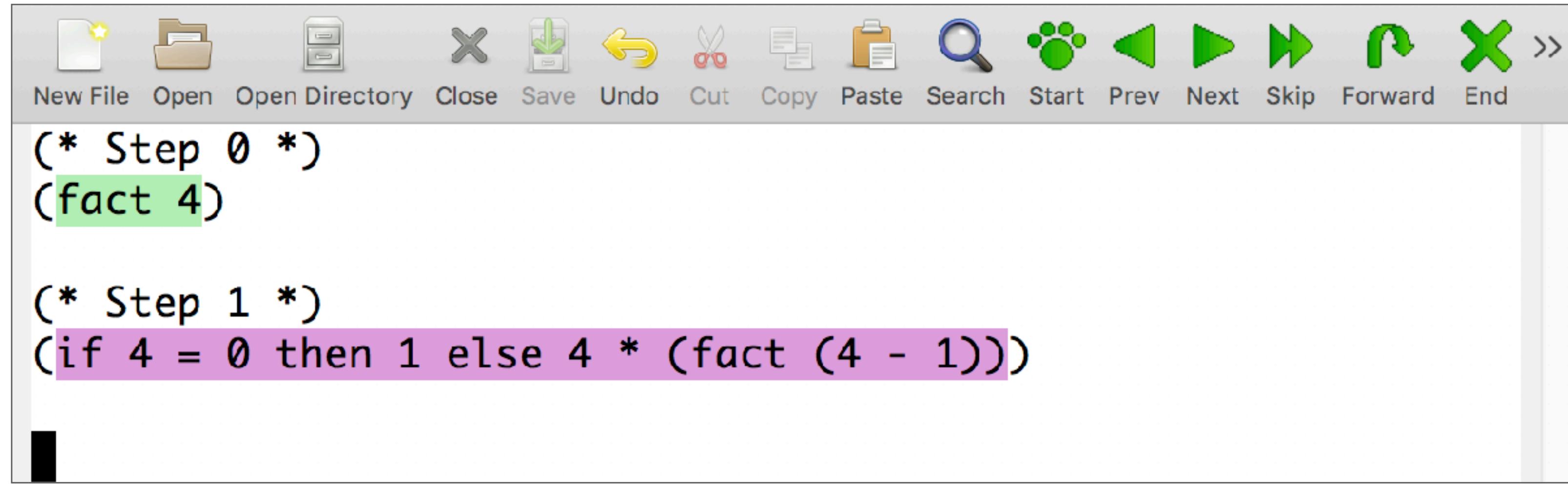


```
let rec fact n = if n = 0 then 1 else n * (fact (n - 1));;
fact 4
```



```
(* Step 0 *)
fact 4

(* Step 1 *)
(if 4 = 0 then 1 else 4 * (fact (4 - 1)))
```



The screenshot shows a software interface with a toolbar at the top and a code editor below it. The toolbar contains various icons for file operations like New File, Open, Save, Undo, Cut, Copy, Paste, Search, and navigation buttons for Start, Prev, Next, Skip, Forward, and End. The code editor displays two steps of a computation:

```
(* Step 0 *)
(fact 4)

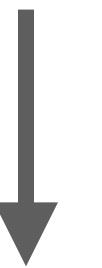
(* Step 1 *)
(if 4 = 0 then 1 else 4 * (fact (4 - 1)))
```

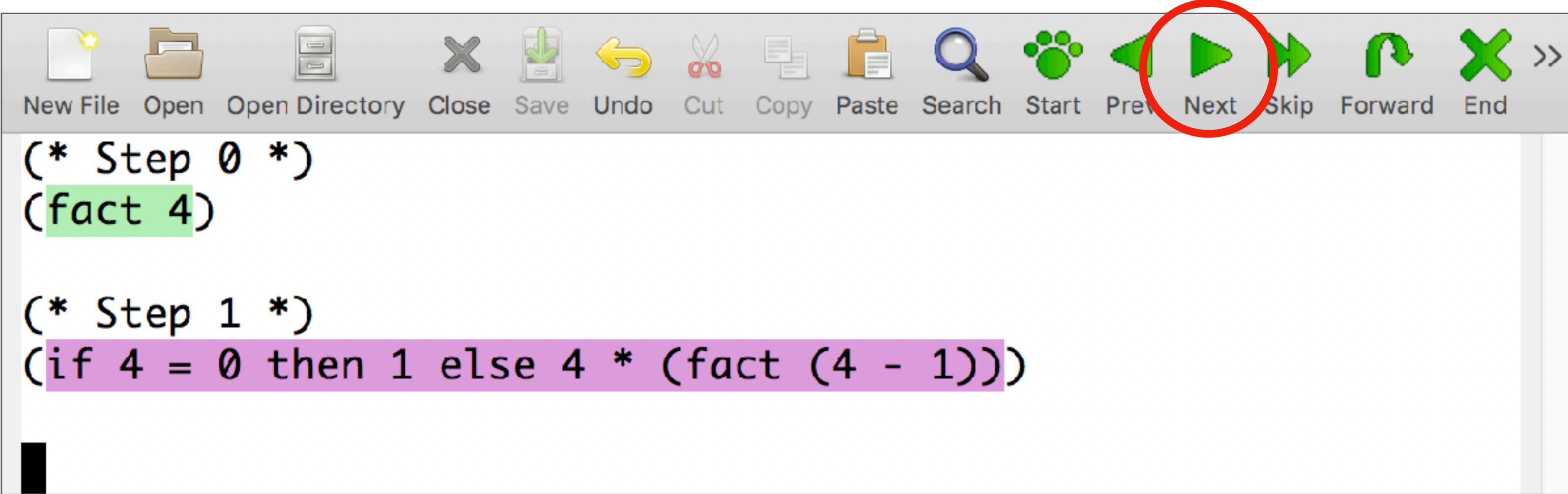
The first step, '(fact 4)', is highlighted with a green background. The second step, '(if 4 = 0 then 1 else 4 * (fact (4 - 1)))', is highlighted with a purple background. A vertical scroll bar is visible on the right side of the code editor.

The screenshot shows a software interface with a toolbar at the top containing various icons for file operations like New File, Open, Save, and Paste, as well as navigation buttons like Prev, Next, Skip, Forward, and End. The 'Next' button is highlighted with a red circle. Below the toolbar is a code editor window displaying the following code:

```
(* Step 0 *)
(fact 4)

(* Step 1 *)
(if 4 = 0 then 1 else 4 * (fact (4 - 1)))
```

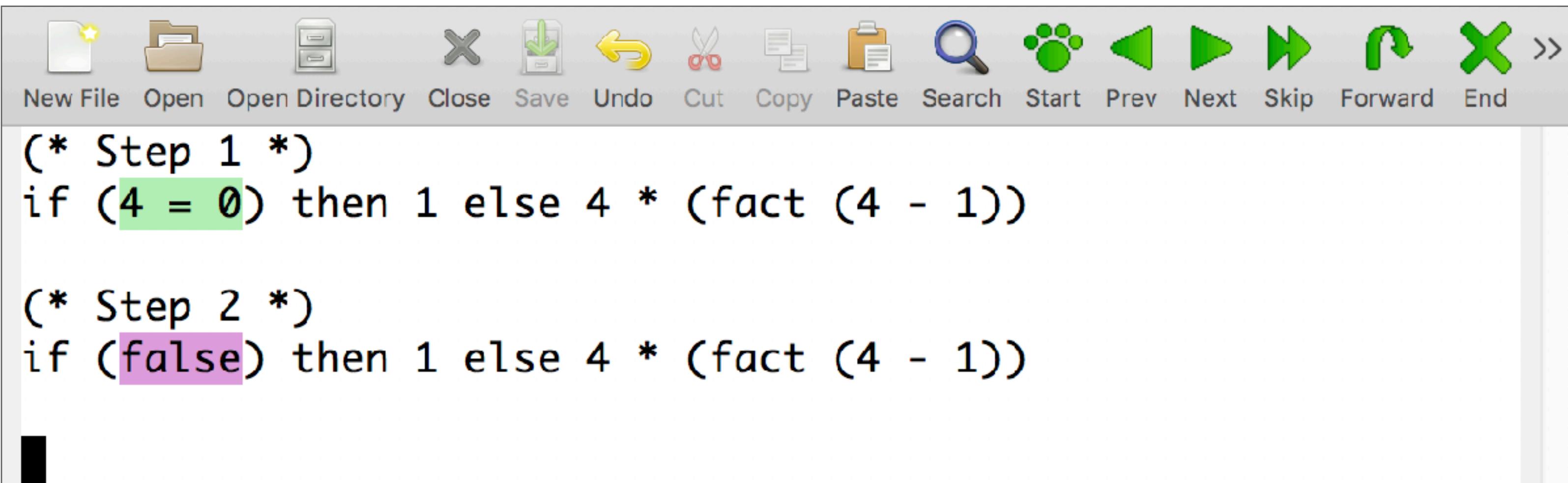
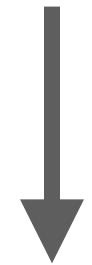




Software interface screenshot showing a toolbar with various icons (New File, Open, Open Directory, Close, Save, Undo, Cut, Copy, Paste, Search, Start, Prev, Next, Skip, Forward, End) and a code editor window. The 'Next' button in the toolbar is circled in red.

```
(* Step 0 *)
(fact 4)

(* Step 1 *)
(if 4 = 0 then 1 else 4 * (fact (4 - 1)))
```

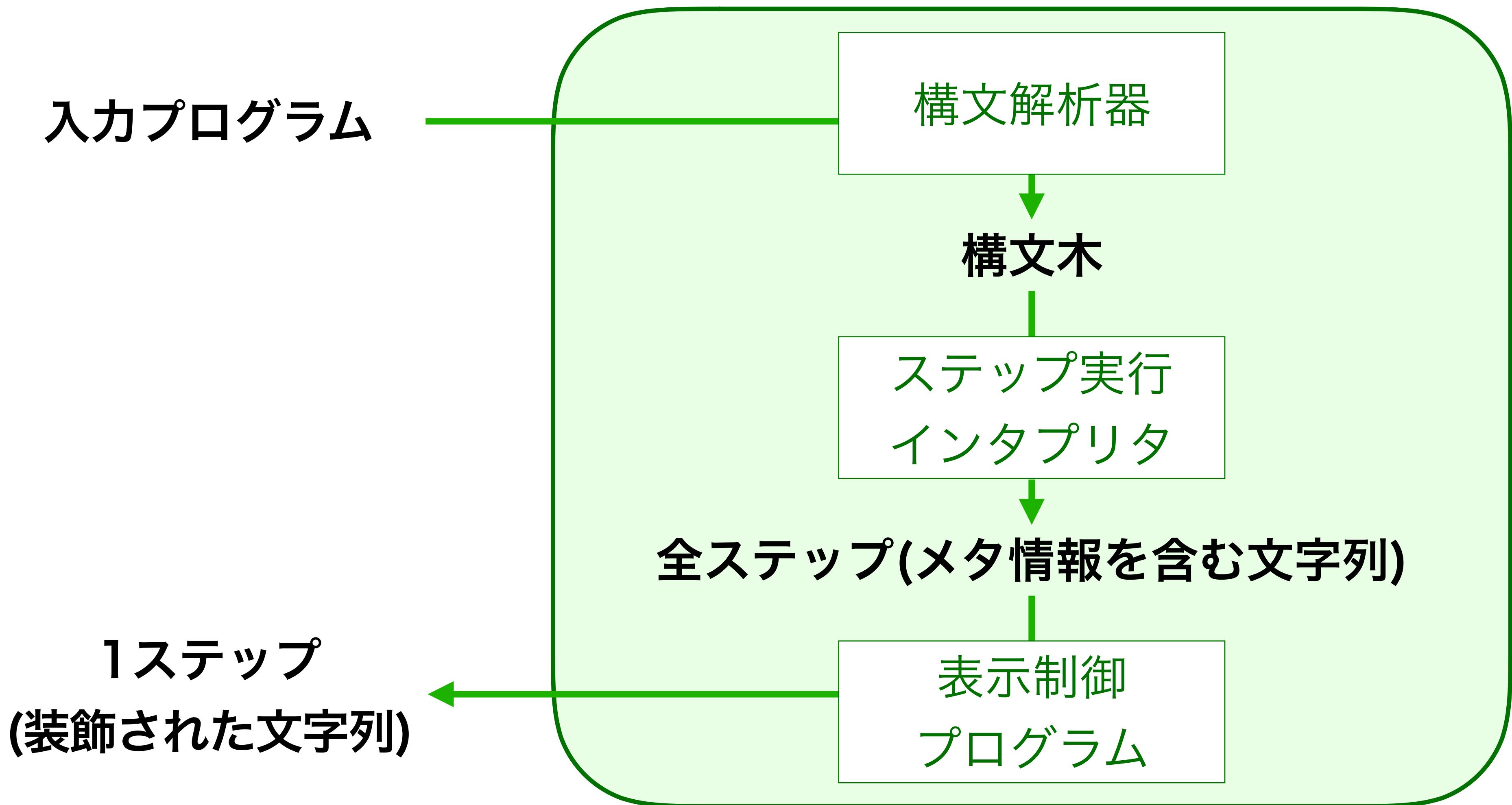


Software interface screenshot showing the same toolbar and code editor window. The code has changed to reflect the next step in the execution.

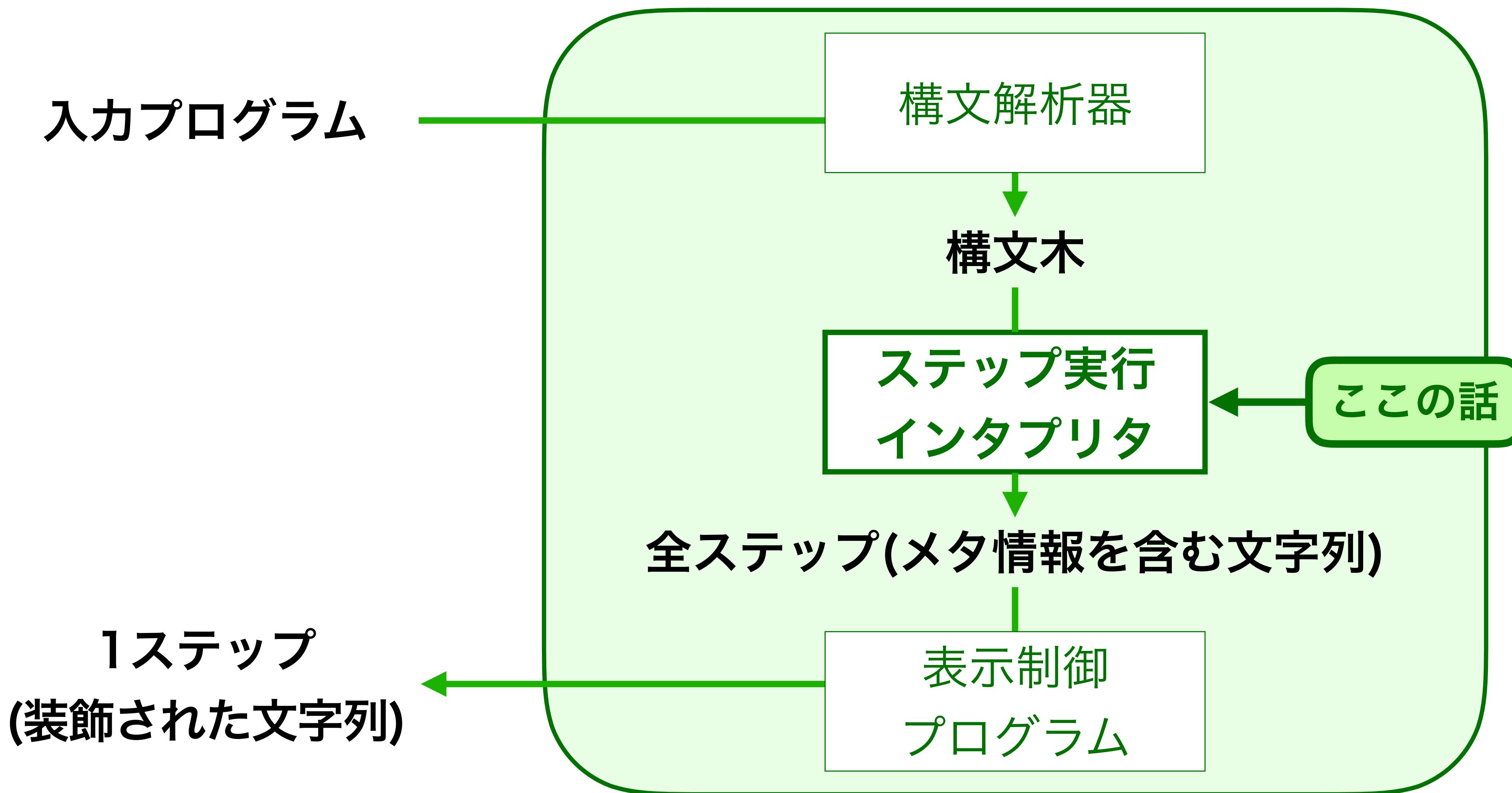
```
(* Step 1 *)
if (4 = 0) then 1 else 4 * (fact (4 - 1))

(* Step 2 *)
if (false) then 1 else 4 * (fact (4 - 1))
```

ステッパの構成



ステッパの構成



ステップ実行インタプリタ

構文木を受け取ってステップ文字列を出力する関数

→ インタプリタに、
簡約時に簡約前後のプログラムを出力する機能を足したもの

以後「ステッパ関数」

ステップ実行インタプリタ

構文木を受け取ってステップ文字列を出力する関数

→ インタプリタに、

簡約時に簡約前後のプログラムを出力する機能を足したもの

以後「ステッパ関数」

常にプログラム全体を出力する（重要）

もくじ

1. algebraic effect handlers
2. ステッパ
3. プログラム変換によらないステッパ関数の作り方
4. プログラム変換によるステッパ関数の作り方

プログラム変換によらない実装

ステッパはインタプリタに副作用を足したものなので、
インタプリタを書き換えることで実装できる

方法1：big-step インタプリタの簡約を行う部分に出力作用を追加する
→ 大域的な制御（一部のステップを飛ばす等）が簡単に実装できる

方法2：small-step インタプリタの1ステップ実行ごとに出力する
→ 単純に実装できる

プログラム変換によらない実装

ステッパはインタプリタに副作用を足したものなので、
インタプリタを書き換えることで実装できる

方法 1 : big-step インタプリタの簡約を行う部分に出力作用を追加する
→ 大域的な制御（一部のステップを飛ばす等）が簡単に実装できる

方法 2 : small-step インタプリタの 1 ステップ実行ごとに出力する
→ 単純に実装できる

ステップ関数の実装における課題

部分式を再帰的に実行すると、

その部分式以外の情報が参照できなくなる

インタプリタ関数(例：λ計算)

```
(* 型無しλ計算の定義 *)
type v = Var of string
        | Fun of string * e
and   e = Val of v
        | App of e * e
(* インタプリタ *)
let rec eval exp =
  match exp with
  | Val (v) -> v
  | App (e1, e2) ->
    let v2 = eval e2 in
    let v1 = eval e1 in
    let reduct = match v1 with
      | Fun (x, e) -> subst e [(x, v2)] in
    eval reduct
```

インタプリタ関数(例：λ計算)

```
let rec eval exp =
  match exp with
  | Val (v) -> v
  | App (e1, e2) ->
    let v2 = eval e2 in
    let v1 = eval e1 in
    let reduct = match v1 with
      | Fun (x, e) -> subst e [(x, v2)] in
    eval reduct
```

```
type v = Var of string
       | Fun of string * e
and e  = Val of v
       | App of e * e
```

これに式 `(fun x -> x) ((fun y -> y) 1)`
を渡すと、

インタプリタ関数(例：λ計算)

```
let rec eval `(`fun x -> x) ((`fun y -> y) 1)` =
  match `(`fun x -> x) ((`fun y -> y) 1)` with
    | Val (v) -> v
    | App (e1, e2) ->
        let v2 = eval `(`fun y -> y) 1` in
        let v1 = eval `(`fun x -> x)` in
        let reduct = match v1 with
          | Fun (x, e) -> subst e [(x, v2)] in
        eval reduct
```

```
type v = Var of string
      | Fun of string * e
and e = Val of v
      | App of e * e
```

このようになる。

インタプリタ関数(例：λ計算)

```
let rec eval `(`fun x -> x) ((`fun y -> y) 1)` =
  match `(`fun x -> x) ((`fun y -> y) 1)` with
  | Val (v) -> v
  | App (e1, e2) ->
    let v2 = eval `(`fun y -> y) 1` in
    let v1 = eval `(`fun x -> x)` in
    let reduct = match v1 with
      | Fun (x, e) -> subst e [(x, v2)] in
    eval reduct
```

```
type v = Var of string
       | Fun of string * e
and e = Val of v
       | App of e * e
```

まず、right-to-left のインタプリタなので引数部分
`(`fun y -> y) 1` を再帰的に実行する。

インタプリタ関数(例：λ計算)

```
let rec eval `(`fun x -> x) ((`fun y -> y) 1)` =
  match `(`fun x -> x) ((`fun y -> y) 1)` with
  | Val (v) -> v
  | App (e1, e2) ->
    let v2 = eval `(`fun y -> y) 1` in
    let v1 = eval `(`fun x -> x)` in
    let reduct = match v1 with
      | Fun (x, e) -> subst e [(x, v2)] in
    eval reduct
```

```
type v = Var of string
       | Fun of string * e
and e = Val of v
       | App of e * e
```

(`fun y -> y) 1 の実行に移ると、

インタプリタ関数(例：λ計算)

```
let rec eval `(fun y -> y) 1` =
  match `(fun y -> y) 1` with
    | Val (v) -> v
    | App (e1, e2) ->
        let v2 = eval `1` in
        let v1 = eval `fun y -> y` in
        let reduct = match v1 with
          | Fun (x, e) -> subst e [(x, v2)] in
        eval reduct
```

```
type v = Var of string
      | Fun of string * e
and e = Val of v
      | App of e * e
```

このようになる。

インタプリタ関数(例：λ計算)

```
let rec eval `(fun y -> y) 1` =  
  match `(fun y -> y) 1` with  
  | Val (v) -> v  
  | App (e1, e2) ->  
    let v2 = eval `1` in  
    let v1 = eval `fun y -> y` in  
    let reduct = match v1 with  
      | Fun (x, e) -> subst e [(x, v2)] in  
    eval reduct
```

```
type v = Var of string  
       | Fun of string * e  
and e = Val of v  
       | App of e * e
```

1と**fun y -> y**は値なので実行するとそのままの値が返される。

インタプリタ関数(例：λ計算)

```
let rec eval `(fun y -> y) 1` =
  match `(fun y -> y) 1` with
    | Val (v) -> v
    | App (e1, e2) ->
        let v2 = eval `1` in
        let v1 = eval `fun y -> y` in
        let reduct = match `fun y -> y` with
          | Fun (x, e) -> subst `y` [(`y`, `1`)] in
        eval reduct
```

```
type v = Var of string
      | Fun of string * e
and e = Val of v
      | App of e * e
```

1と**fun y -> y**は値なので実行するとそのままの値が返される。

インタプリタ関数(例：λ計算)

```
let rec eval `(fun y -> y) 1` =  
  match `(fun y -> y) 1` with  
  | Val (v) -> v  
  | App (e1, e2) ->  
    let v2 = eval `1` in  
    let v1 = eval `fun y -> y` in  
    let reduct = match `fun y -> y` with  
      | Fun (x, e) -> subst `y` [(`y`, `1`)] in  
    eval reduct
```

↑(式 y の中の変数 y に値 1 を代入した式)

```
type v = Var of string  
       | Fun of string * e  
and e = Val of v  
       | App of e * e
```

この代入が (**fun** y -> y) 1 の簡約なので、
ステッパにする為にはここでステップを出力したい。

インタプリタ関数(例：λ計算)

```
let rec eval `(fun y -> y) 1` =
  match `(fun y -> y) 1` with
    | Val (v) -> v
    | App (e1, e2) ->
        let v2 = eval `1` in
        let v1 = eval `fun y -> y` in
        let reduct = match `fun y -> y` with
          | Fun (x, e) -> subst `y` [(`y`, `1`)] in
        eval reduct
```

この代入が (**fun** y -> y) 1 の簡約なので、
ステッパにする為にはここでステップを出力したい。
出力したいもの：

```
((fun x -> x) ((fun y -> y) 1))
((fun x -> x) 1)
```

```
type v = Var of string
      | Fun of string * e
and e = Val of v
      | App of e * e
```

インタプリタ関数(例：λ計算)

```
let rec eval `(`fun y -> y) 1` =
  match `(`fun y -> y) 1` with
    | Val (v) -> v
    | App (e1, e2) ->
        let v2 = eval `1` in
        let v1 = eval `fun y -> y` in
        let reduct = match `fun y -> y` with
          | Fun (x, e) -> subst `y` [(`y`, `1`)] in
        eval reduct
```

しかしコンテキスト $(\text{fun } x \rightarrow x) [.]$ の情報が
見えていないため $(\text{fun } x \rightarrow x)$ は出力できない。
出力したいもの：

$((\text{fun } x \rightarrow x) ((\text{fun } y \rightarrow y) 1))$
 $((\text{fun } x \rightarrow x) 1)$

```
type v = Var of string
      | Fun of string * e
and e = Val of v
      | App of e * e
```

ステップ関数の実装における課題

部分式を再帰的に実行すると、

その部分式以外の情報が参照できなくなる

→ 実行中の部分式のコンテキストを参照できるようにしたい

既存の実装方法

- Racket の教育用の一部の構文のステッパ[5]
Racket の機能 (continuation-mark) を使って dynamic に記録
- OCaml のステッパ[6]
small-step のインタプリタを書く
- 再帰関数を含む OCaml の一部のステッパ[7]
- try-with を含む OCaml の一部のステッパ[8]
コンテキストの情報を持つ引数を追加する

既存の実装方法

- Racket の教育用の一部の構文のステッパ[5]
Racket の機能 (continuation-mark) を使って dynamic に記録
Racket を使わないので実装したい
- OCaml のステッパ[6]
small-step のインタプリタを書く
big-step で実装したい
- 再帰関数を含む OCaml の一部のステッパ[7]
- try-with を含む OCaml の一部のステッパ[8]
コンテキストの情報を持つ引数を追加する
この方法をとる

先行研究[8]の方法（λ計算）

```
(* インタプリタ *)
let rec eval exp =
  match exp with
  | Val (v) -> v
  | App (e1, e2) ->
    let v2 = eval e2 in
    let v1 = eval e1 in
    let reduct = match v1 with
      | Fun (x, e) -> subst e [(x, v2)] in
    eval reduct
```

```
type v = Var of string
      | Fun of string * e
and e = Val of v
      | App of e * e
```

先行研究[8]の方法（λ計算）

```
(* ステッパー *)
let rec eval exp ctxt =
  match exp with
    | Val (v) -> v
    | App (e1, e2) ->
        let v2 = eval e2 ((CAppR e1) :: ctxt) in
        let v1 = eval e1 ((CAppL v2) :: ctxt) in
        let reduct = match v1 with
          | Fun (x, e) -> subst e [(x, v2)] in
            eval reduct ctxt
(* 実行を始める関数 *)
let stepper exp = eval exp []
```

```
type v = Var of string
      | Fun of string * e
and e  = Val of v
      | App of e * e
```

```
(* コンテキストの型 *)
type frame_t = CAppR of e (* e [.] *)
              | CAppL of v (* [.] v *)
type ctxt_t = frame_t list
```

先行研究[8]の方法（λ計算）

```
(* ステッパー *)
let rec eval exp ctxt =
  match exp with
    | Val (v) -> v
    | App (e1, e2) ->
        let v2 = eval e2 ((CAppR e1) :: ctxt) in
        let v1 = eval e1 ((CAppL v2) :: ctxt) in
        let reduct = match v1 with
          | Fun (x, e) -> subst e [(x, v2)] in
            eval reduct ctxt
(* 実行を始める関数 *)
let stepper exp = eval exp []
```

```
ctxt[e1 [.] ]  
 ctxt[.] v2  
空のコンテキスト
```

```
type v = Var of string
      | Fun of string * e
and e  = Val of v
      | App of e * e
```

```
(* コンテキストの型 *)
type frame_t = CAppR of e (* e [.] *)
              | CAppL of v (* [.] v *)
type ctxt_t = frame_t list
```

先行研究[8]の方法

- ・ インタプリタに引数を追加して、コンテキストの情報を渡す
- ・ コンテキストを表すデータ型を考えて、
インタプリタの再帰呼び出し時に適切に拡張したコンテキストを渡す

先行研究[8]の方法

- ・ インタプリタに引数を追加して、コンテキストの情報を渡す
- ・ コンテキストを表すデータ型を考えて、
インタプリタの再帰呼び出し時に適切に拡張したコンテキストを渡す

どのような型にするべきか自明でない

本研究の方法

- ・ インタプリタに引数を追加して、コンテキストの情報を渡す
- ・ コンテキストを表すデータ型を考えて、
インタプリタの再帰呼び出し時に適切に拡張したコンテキストを渡す
- ・ インタプリタにプログラム変換を施すことでコンテキストを抽出する

型が機械的に定まる

もくじ

1. algebraic effect handlers
2. ステッパ
3. プログラム変換によらないステッパ関数の作り方
4. プログラム変換によるステッパ関数の作り方

プログラム変換

プログラムを別のプログラムにすること

今回使ったのは

- **CPS変換**
 - 引数を1つ増やして、継続を表す関数を渡すようになる
 - プログラムの挙動を変えない
- **非関数化**
 - 関数を関数でないデータに変換する
 - プログラムの挙動を変えない

変換の内容

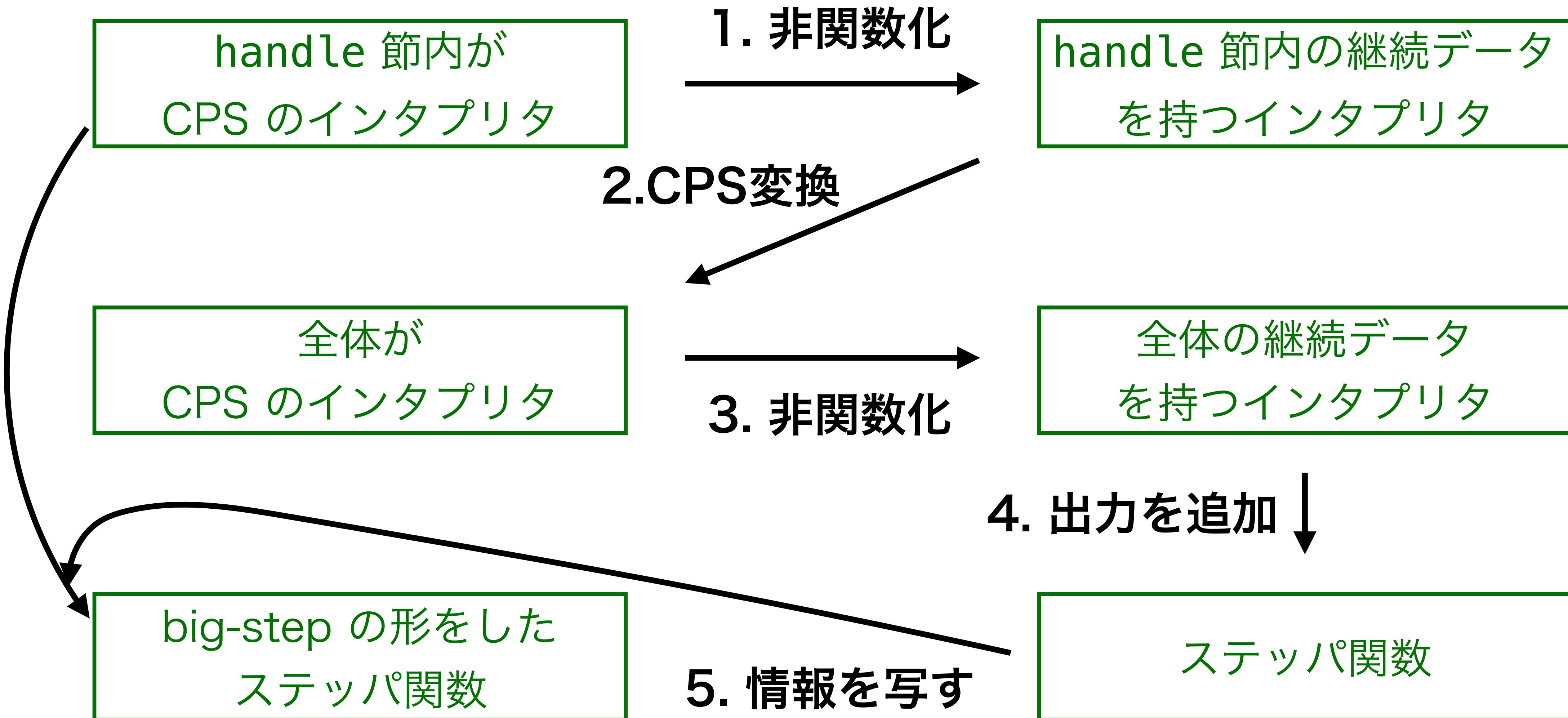
handle 節内を実行する部分が CPS のインタプリタに、

1. 非関数化
2. CPS 変換
3. 非関数化
4. 出力命令の挿入

をして、

5. 得られたコンテキスト情報と出力命令を元のインタプリタの対応する箇所に書き加える

変換の内容



CPS変換とは

継続 渡し 形式
CPS : Continuation Passing Style にすること

- ・ 関数呼び出しの後にその値に対してする計算を関数として持つようになる
- ・ 継続を実行するタイミングや対象が記述できるようになる
- ・ この研究においては、インタプリタの継続を明示的に扱える関数に変換するためには用いる

CPS変換とは

継続 渡し 形式
CPS : Continuation Passing Style にすること

```
let rec product lst = match lst with
| [] -> 1
| first :: rest -> first * product rest
```

```
let rec loop lst k = match lst with
| [] -> k 1
| first :: rest -> loop rest (fun n -> first * n)
let product lst = loop lst (fun n -> n)
```

CPS変換とは

継続 渡し 形式
CPS : Continuation Passing Style にすること

```
let rec product lst = match lst with
| [] -> 1
| first :: rest -> first * product rest
```

```
let rec loop lst k = match lst with
| [] -> k 1
| first :: rest -> loop rest (fun n -> first * n)
let product lst = loop lst (fun n -> n)
```

非関数化とは

- 関数を関数でないデータで表す
- その関数を呼び出している部分を、別の関数(apply)の呼び出しに変える
- この研究においては、継続を関数にしたものを、**関数(継続)の種類や関数内部で使うデータを判別できるものにするために用いる**

非関数化とは

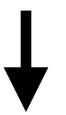
```
let rec loop lst k = match lst with
| [] -> k 1
| first :: rest -> loop rest (fun n -> first * n)
let product lst = loop lst (fun n -> n)
```



```
type f = Times of int | Return
let rec loop lst k = match lst with
| [] -> apply k 1
| first :: rest -> loop rest (Times (first))
and product lst = loop lst Return
and apply k n = match k with
| Times (first) -> first * n
| Return -> n
```

非関数化とは

```
let rec loop lst k = match lst with
| [] -> k 1
| first :: rest -> loop rest (fun n -> first * n)
let product lst = loop lst (fun n -> n)
```



```
type f = Times of int | Return
let rec loop lst k = match lst with
| [] -> apply k 1
| first :: rest -> loop rest (Times (first))
and product lst = loop lst Return
and apply k n = match k with
| Times (first) -> first * n
| Return -> n
```

非関数化とは

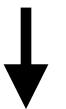
```
let rec loop lst k = match lst with
| [] -> k 1
| first :: rest -> loop rest (fun n -> first * n)
let product lst = loop lst (fun n -> n)
```



```
type f = Times of int | Return
let rec loop lst k = match lst with
| [] -> apply k 1
| first :: rest -> loop rest (Times (first))
and product lst = loop lst Return
and apply k n = match k with
| Times (first) -> first * n
| Return -> n
```

非関数化とは

```
let rec loop lst k = match lst with
| [] -> k 1
| first :: rest -> loop rest (fun n -> first * n)
let product lst = loop lst (fun n -> n)
```



```
type f = Times of int | Return
let rec loop lst k = match lst with
| [] -> apply k 1
| first :: rest -> loop rest (Times (first))
and product lst = loop lst Return
and apply k n = match k with
| Times (first) -> first * n
| Return -> n
```

変換前後のプログラム（入計算）

```
let rec eval exp =
  match exp with
  | Val (v) -> v
  | App (e1, e2) ->
    let v2 = eval e2 in ...
```

```
let rec eval exp k =
  match exp with
  | Val (v) -> k v
  | App (e1, e2) ->
    eval e2 (fun v2 -> ...)
```

```
let rec eval exp k =
  match exp with
  | Val (v) -> apply k v
  | App (e1, e2) ->
    eval e2 (Arg (e1) :: k)
and apply k v = match k with
  | Arg (e1) :: rest -> ...
```

CPS変換

非関数化

変換前後のプログラム（入計算）

```
let rec eval exp =
  match exp with
  | Val (v) -> v
  | App (e1, e2) ->
    let v2 = eval e2 in ...
```

```
let rec eval exp k =
  match exp with
  | Val (v) -> k v
  | App (e1, e2) ->
    eval e2 (fun v2 -> ...)
```

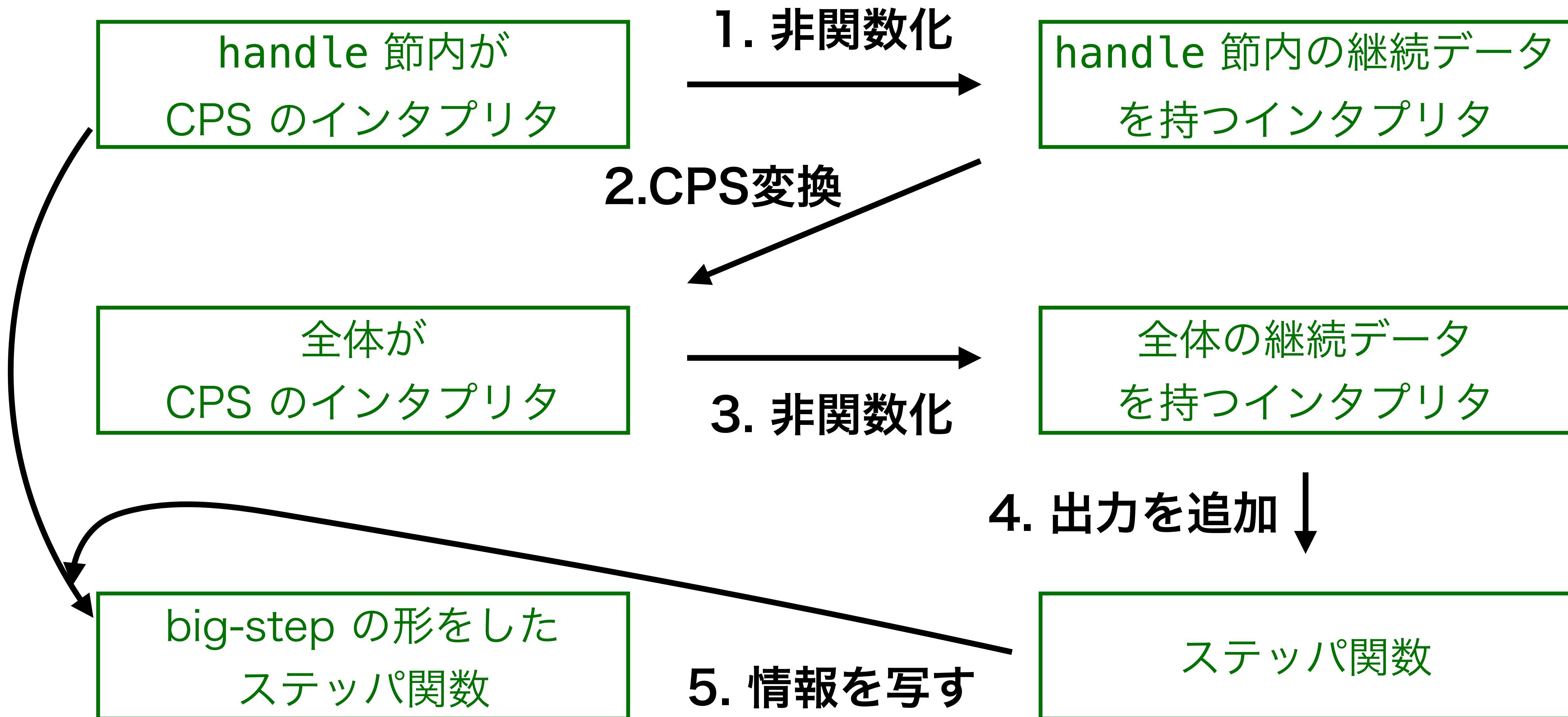
```
let rec eval exp k =
  match exp with
  | Val (v) -> apply k v
  | App (e1, e2) ->
    eval e2 (Arg (e1) :: k)
and apply k v = match k with
  | Arg (e1) :: rest -> ...
```

CPS変換

非関数化

実行部分から外れる式の
情報を足しながら
再帰しているので c に
外の式の情報が入っている

変換の内容



変換の内容

handle 節内が
CPS のインタプリタ

1. 非関数化

handle 節内の継続データ
を持つインタプリタ

2.CPS変換

全体が
CPS のインタプリタ

3. 非関数化

全体の継続データ
を持つインタプリタ

big-step インタプリタを CPS 変換して非関数化
(抽象機械を導出できる変換[9])

変換の内容

handle 節内が
CPS のインタプリタ

1. 非関数化

handle 節内

handle 節内の継続データ
を持つインタプリタ

2.CPS変換

全体

全体が
CPS のインタプリタ

3. 非関数化

全体の継続データ
を持つインタプリタ

big-step インタプリタを CPS 変換して非関数化

(抽象機械を導出できる変換[9])

を handle 節内に 1 回、全体に 1 回施す

(handle 節内はもともと CPS なので CPS 変換は不要)

得られたコンテキストの型

(* handle 内のコンテキスト *)

```
and c = FId
  | FApp2 of e * c
  | FApp1 of c * v
  | FOp of string * c
```

(* [.] *)
(* [e [.]] *)
(* [[.]] v *)
(* [op[.]] *)

(* 全体のコンテキスト *)

```
and c2 = GId
  | GHandle of h * c * c2
```

(* [.] *)
(* [[[with h handle [.]]]] *)

→ Hillerström ら[10, 11] の algebraic effect handlers が入った体系の抽象機械において継続を表すデータと同じ構造が導出された

ステップ例

```
0: 100 + (with {return x -> x, op(x; k) -> k (x + 1)}
               handle 10 + op(3))
```

```
1: 100 + (fun y ->
               (with {return x -> x, op(x; k) -> k (x + 1)}
                     handle 10 + y) (3 + 1))
```

```
1: 100 + (fun y ->
               (with {return x -> x, op(x; k) -> k (x + 1)}
                     handle 10 + y) (3 + 1))
```

```
2: 100 + (fun y ->
               (with {return x -> x, op(x; k) -> k (x + 1)}
                     handle 10 + y) 4)
```

ステップ例

```
2: 100 + (fun y ->
  (with {return x -> x, op(x; k) -> k (x + 1)}
    handle 10 + y) 4)

3: 100 + (with {return x -> x, op(x; k) -> k (x + 1)}
  handle 10 + 4)

3: 100 + (with {return x -> x, op(x; k) -> k (x + 1)}
  handle 10 + 4)

4: 100 + (with {return x -> x, op(x; k) -> k (x + 1)}
  handle 14)
```

ステップ例

```
4: 100 + (with {return x -> x, op(x; k) -> k (x + 1)}  
           handle 14)
```

```
5: 100 + 14
```

```
5: 100 + 14
```

```
6: 114
```

他の言語の場合

- 型無しλ計算の CPS インタプリタ
非関数化
- 型無しλ計算の DS インタプリタ
CPS変換→**非関数化**
- shift/reset の CPS インタプリタ
非関数化→**CPS変換**→**非関数化**
(内側) (全体) (全体)
- try-with、shift/reset の DS インタプリタ
CPS変換→**非関数化**→**CPS変換**→**非関数化**
(内側) (内側) (全体) (全体)

限定継続を操作しない言語では
全体に対して
CPS変換と非関数化をする

限定継続を操作する言語では
継続を限定する範囲の内側と
全体に対してそれぞれ
CPS変換と非関数化をする

まとめと今後の課題

- algebraic effect handlers に対する
big-step の CPS インタプリタを定義した
- インタプリタに CPS 変換と非関数化を施すことで、コンテキストを明示的に保持するインタプリタを得られ、ステッパを実装できた
- 課題 1 得られたコンテキスト情報を元のインタプリタに書き加える工程も機械的なプログラム変換で行う
- 課題 2 algebraic effect handlers を含む Multicore OCaml に対するステッパツールを実装する

参考文献

- [1] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP'13), pp. 145–158, 2013.
- [2] Matija Pretnar. An introduction to algebraic effect handlers and handlers, invited tutorial paper. Electronic Notes in Theoretical Computer Science, Vol. 319, pp. 19–35, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).
- [3] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In Proceedings of the 1st International Workshop on Type-Driven Development (TyDe'16), pp. 15–27, 2016.
- [4] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In Dale Miller, editor, 2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017), Leibniz International Proceedings in Informatics (LIPIcs), pp. 18:1–18:19, September 2017.
- [5] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In European symposium on programming, pp. 320–334. Springer, 2001.

参考文献

- [6] John Whitington and Tom Ridge. Direct interpretation of functional programs for debugging. In Sam Lindley and Gabriel Scherer, editors, Proceedings ML Family / OCaml Users and Developers work- shops, Oxford, UK, 7th September 2017, Vol. 294 of Electronic Proceedings in Theoretical Computer Science, pp. 41–73, 2019.
- [7] Youyou Cong and Kenichi Asai Implementing a stepper using delimited continuations. Software Science 39, pp. 42–54, 2016.
- [8] Tsukino Furukawa, Youyou Cong, and Kenichi Asai. Stepping OCaml. In Proceedings Seventh International Workshop on Trends in Functional Programming in Education, Chalmers University, Gothenburg, Sweden, 14th June 2018, Vol. 295 of Electronic Proceedings in Theoretical Computer Science, pp. 17–34, 2019.
- [9] Olivier Danvy. Defunctionalized interpreters for programming languages. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP’08), pp. 131–142, 2008.
- [10] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In Proceedings of the 1st International Workshop on Type-Driven Development (TyDe’16), pp. 15–27, 2016.
- [11] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In Dale Miller, editor, 2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017), Leibniz International Proceedings in Informatics (LIPIcs), pp. 18:1–18:19, September 2017.