つくってあそぼ
ラムダ計算インタプリタ

2018/10/18 KMC例会講座 @utgwkk

自己紹介

- @utgwkk (うたがわきき)
- 工学部情報学科4回生
- プログラミング言語に関する研究
- Euro Truck Simulator 2

参考

[1] Urzyczyn et al. Lectures on the Curry-Howard Isomorphism, Studies in Logic and the Foundations of Mathematics, 149, Elsevier Science

[2] http://www.kb.ecei.tohoku.ac.jp/~sumii/class/keisanki-software-kougaku-2005/lambda.pdf

[3] http://ttic.uchicago.edu/~pl/classes/CMSC336-Winter08/lectures/lec4.pdf

おしながき

- **動機**
- λ計算について知ろう
- λ計算のインタプリタを実装しよう

動機

- λ計算の見た目はコンパクトでかつ表現力が強い
- しかし実装をしたことがない
- SOONインタプリタを実装してみよう
- ~~論文読むのがだるいときに手を動かしてた~~

この講座の目的

- 「みなさんが普段使っているプログラミング言語(のインタプリタ)もこのように実装されている」という実感を持ってもらう

- λ計算を実装したときのつまずきポイントを共有する

おしながき

- 動機
- **λ計算について知ろう**
- λ計算のインタプリタを実装しよう

(型無し) λ 計算

- 計算のモデル化のひとつ
- 関数をつくる(抽象)
- 関数を適用する
- これだけでなんとチューリング完全

λ項

- M ::= x | λx.M | M M
  - xは変数
  - 関数抽象は右結合 λx.λy.M = λx.(λy.M)
  - 関数適用は左結合 PQR = (PQ)R
  - 適当に括弧を加えてよい

さまざまなλ項

- $(\lambda x.x)y$

- $\lambda x.\lambda y.x$

- $\lambda x.\lambda y.\lambda z.xz(yz)$

束縛変数と自由変数

- 束縛変数
  - 関数抽象によって束縛されている変数
    - λx.λy.xyz の束縛変数はx,y
- 自由変数
  - 束縛変数ではないもの
    - λx.λy.xyz の自由変数はz

変数の置換

- M[x:=N]
  - 項Mに登場する自由変数xを
  - (差し支えない範囲で)項Nに置換する
- 差し支えない範囲ってなに？

変数の置換の例

- $x[x:=\lambda y.y] = \lambda y.y$
- $y[x:=\lambda z.z] = y \ (x \neq y)$
- $(x\ z\ (y\ x))[x:=\lambda w.w] = (\lambda w.w)\ z\ (y\ (\lambda w.w))$
- $\lambda x.x[x:=\lambda y.y] = \lambda x.x$
- $\lambda y.x[x:=\lambda z.z] = \lambda y.\lambda z.z \ (x \neq y)$

α変換

- λ x.M $=_\alpha$ λ y.M[x:=y]
- 差し支えない範囲で引数の変数名は本質ではない
- λ x.x と λ y.y を区別する必要はないですよね？

β簡約

- ざっくり言えば関数適用

- **(λx.M)** N -> M[x:=N]

β簡約を体感する

- def f(x): return x + x
- **f**(2) = 2 + 2 (= 4)
- // これはあくまで体感です

β簡約

(λx.λy.λz.xz(yz)) (λx.λy.x) (λx.λy.x)

->**(λy.λz.(λx.λy.x)z(yz))** (λx.λy.x)

->λz.**(λx.λy.x)**z((λx.λy.x)z)

->λz.(λy.z)(**(λx.λy.x)**z)

->λz.**(λy.z)**(λy.z)

->λz.z

正規形

- これ以上β簡約できない項
- 例
  - x
  - λ x.x
  - λ x.λ y.x

おしながき

- 動機
- ~~λ計算について知ろう~~
- **λ計算のインタプリタを実装しよう**

仕様

- 入力: λ 項の文字列表現
- 出力: β 簡約のステップと正規形(のようなもの)
  - (じつは必ずしも正規形に簡約されるわけではない)
  - (β 簡約してみて，項が変化しなかったら停止する)

実装言語

- 言語はOCamlを使う
- そこそこツールチェーンがある
- わたしがよく触っている
    - 使いこなせてはいないと思う……

簡約までの流れ

$(\lambda x.\lambda y.\lambda z.xz(yz))(\lambda x.\lambda y.x)(\lambda x.\lambda y.x)$

↓ 字句解析

トークン列 (LAMBDA ID DOT ...)

↓ 構文解析

構文木 (Fun (x, ...))

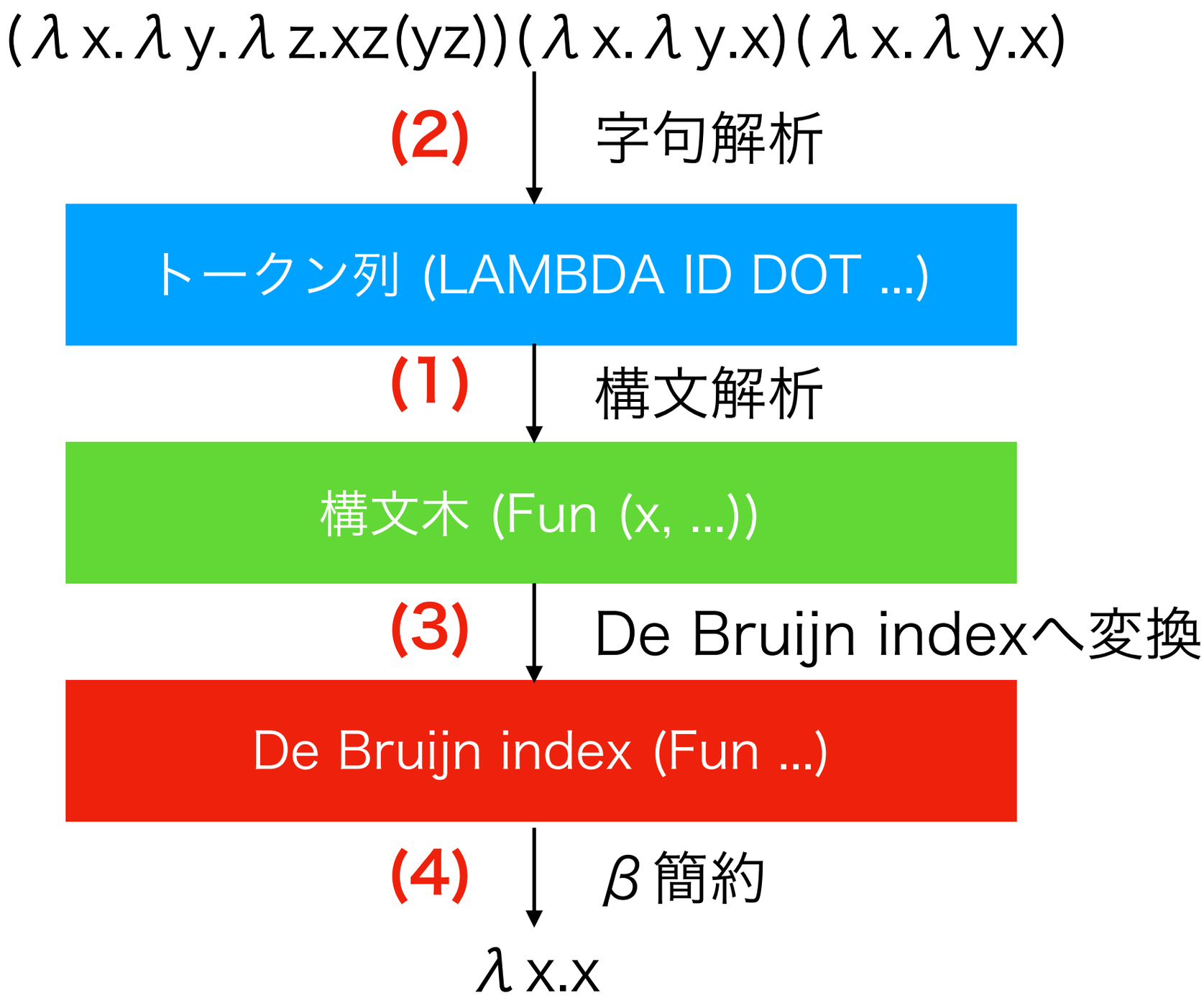
↓ De Bruijn indexへ変換

De Bruijn index (Fun ...)

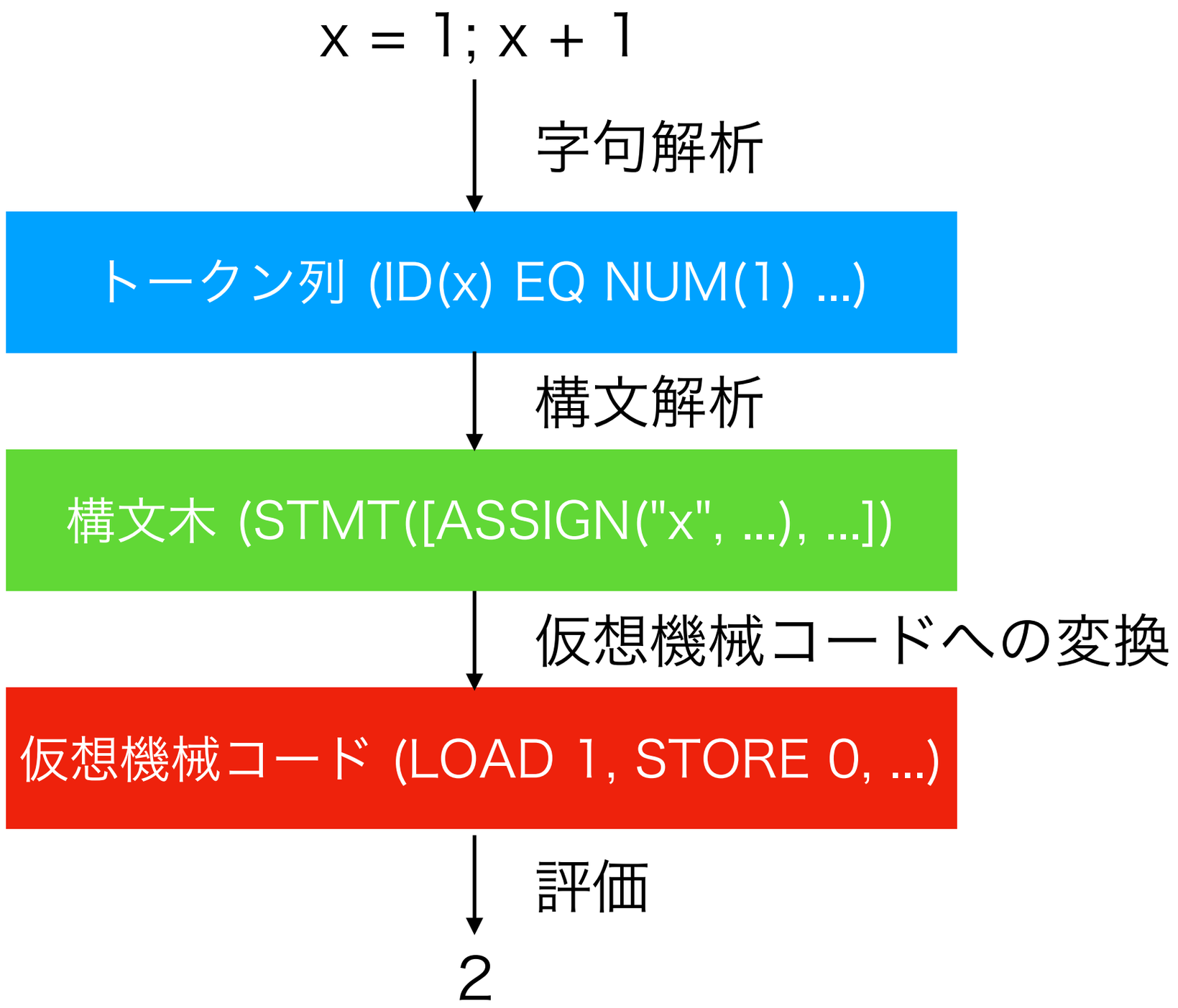
↓ β 簡約

$\lambda x.x$

実装の流れ


(λx.λy.λz.xz(yz))(λx.λy.x)(λx.λy.x)
(2)
字句解析
トークン列 (LAMBDA ID DOT ...)
(1)
構文解析
構文木 (Fun (x, ...))
(3)
De Bruijn indexへ変換
De Bruijn index (Fun ...)
(4)
β簡約
λx.x

cf. インタプリタ



※どれくらいデータ構造を変換するかは設計による

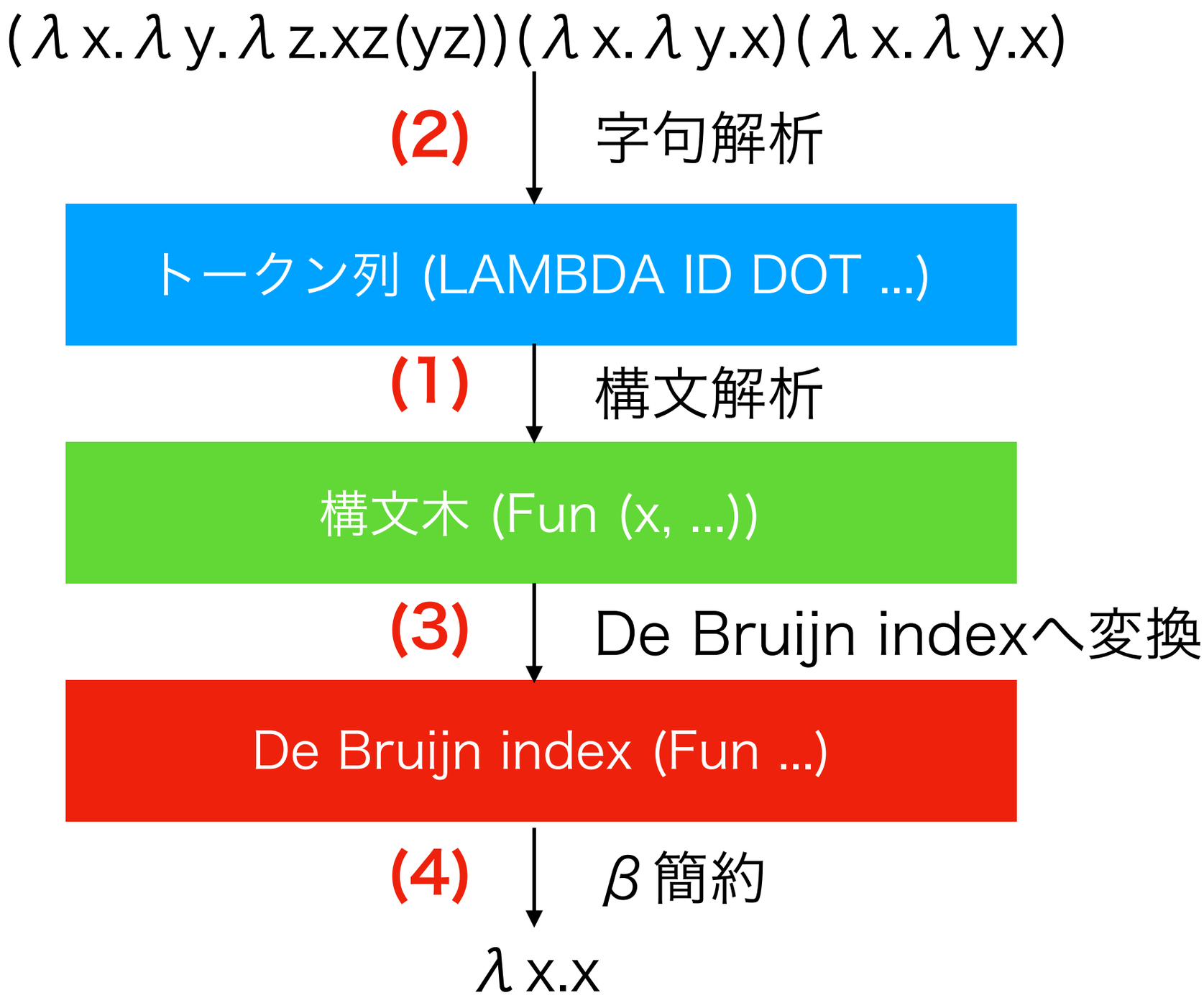
λ項のデータ型を定義しよう

- M ::= x | λx.M | M M
  - 忠実に落としこむ

λ項のデータ型を定義しよう

```
type id = string

type term =
    | Var of id
    | Fun of id * term
    | App of term * term
```

実装の流れ


(λx.λy.λz.xz(yz))(λx.λy.x)(λx.λy.x)
(2)
字句解析
トークン列 (LAMBDA ID DOT ...)
(1)
構文解析
構文木 (Fun (x, ...))
(3)
De Bruijn indexへ変換
De Bruijn index (Fun ...)
(4)
β簡約
λx.x

構文解析

- λx.x のような記号列に意味を持たせる作業
  - λx.x -> Fun (x, Var x)
- (厳密にはこの記号列をそのまま扱うわけではない)

構文解析器を実装しよう

- menhirを使うよ
- yaccのOCaml版のすごいやつ
  - ocamlyaccというのもある
- 文法をBNF風のもので記述

トークンを定義する

- M ::= x | λx.M | M M
  - グッと睨んで必要そうなトークンを定義しよう
    - ID (変数)
    - LAMBDA, DOT (λ , .)
    - LPAREN, RPAREN (括弧)
    - EOL (λ項の終端)

トークンを定義する

```
%{
  open Syntax
%}

%token <Syntax.id> ID
%token LAMBDA DOT
%token LPAREN RPAREN
%token EOL
```

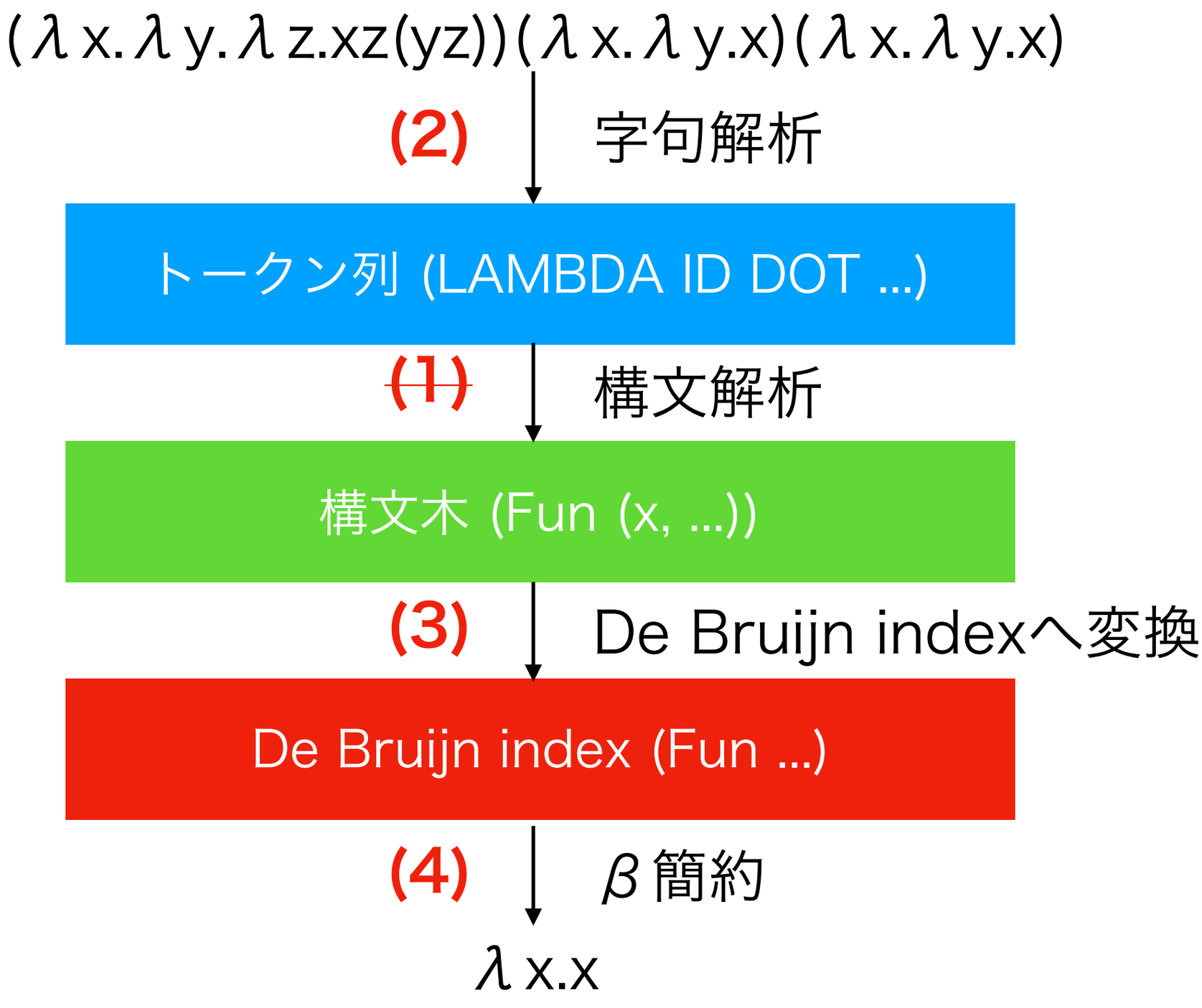
文法を定義する

- M ::= x | λx.M | M M
  - グッと睨んで文法を書く
    - <main> ::= <Expr> EOL
    - <Expr> ::= LAMBDA ID DOT <Expr> | <AppExpr>
    - <AppExpr> ::= <AppExpr> <AExpr> | <AExpr>
    - <AExpr> ::= ID | LPAREN <Expr> RPAREN

文法を定義する

```
%start main
%type <Syntax.term> main
%%

main:
  Expr EOL { $1 }

Expr:
| LAMBDA i=ID DOT e=Expr { Fun (i, e) }
| AppExpr { $1 }

AppExpr:
| e1=AppExpr e2=AExpr { App (e1, e2) }
| AExpr { $1 }

AExpr:
| i=ID { Var i }
| LPAREN e=Expr RPAREN { e }
```

実装の流れ


(λx.λy.λz.xz(yz))(λx.λy.x)(λx.λy.x)
(2)
字句解析
トークン列 (LAMBDA ID DOT ...)
(1)
構文解析
構文木 (Fun (x, ...))
(3)
De Bruijn indexへ変換
De Bruijn index (Fun ...)
(4)
β簡約
λx.x

字句解析

- そもそもλx.xのような文字列のままだと扱いづらい
  - 空白文字を無視するとか
  - コメントもここで取り除かれる
- LAMBDA ID(x) DOT ID(x) ぐらいに明確な列だとよい

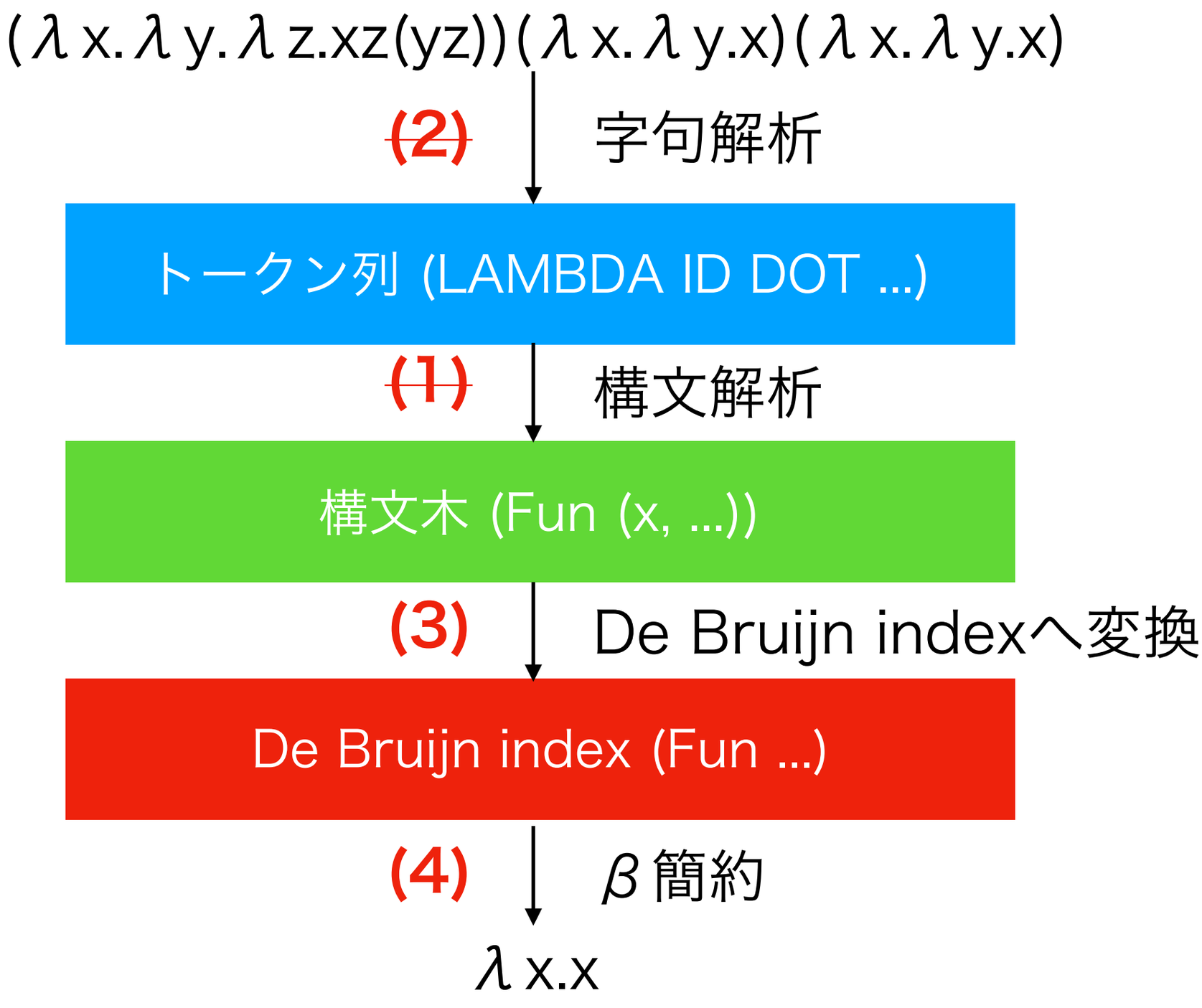
字句解析器を実装しよう

- ocamllexを使うよ
- lexのOCaml版
- ルールを正規表現で記述

字句解析ルールを定義する

```
{
  open Parser
}

rule main = parse
  [' ' '\t']+ { main lexbuf }
| ['\n'] { EOL }
| ['a'-'z'] { ID (Lexing.lexeme lexbuf) }
| '\\' { LAMBDA }
| '.' { DOT }
| '(' { LPAREN }
| ')' { RPAREN }
| eof { exit 0 }
```

実装の流れ


(λx.λy.λz.xz(yz))(λx.λy.x)(λx.λy.x)
(2)
字句解析
トークン列 (LAMBDA ID DOT ...)
(1)
構文解析
構文木 (Fun (x, ...))
(3)
De Bruijn indexへ変換
De Bruijn index (Fun ...)
(4)
β簡約
λx.x

De Bruijn indexに変換しよう

- ……ってなに？

α変換(思い出す)

- 「束縛変数の名前は本質ではない」

- いっぽう構文木には束縛変数の名前がある

- ➡SOON適切に変数の名前のことを考えないと
  簡約結果がめちゃくちゃになる

- ➡SOON束縛変数の名前に束縛されたくない！！！！

β簡約(まちがい)

(λx.λy.λz.xz(yz)) (λx.λz.x) (λx.λy.x)

->**(λy.λz.(λx.λz.x)z(yz))** (λx.λy.x)

->λz.**(λx.λz.x)**z((λx.λy.x)z)

->λz.(λz.z)(**(λx.λy.x)**z)

->λz.**(λz.z)**(λy.z)

->λz.λy.z

De Bruijn indexに変換しよう

- 関数抽象による束縛からのネストの深さで変数が一意に定まる
- ここでは0-indexとする
  - λx.λy.x -> λ.λ.1
  - λx.λy.λz.xz(yz) -> λ.λ.λ.2 0 (1 0)
- ただし自由変数は放っておく

De Bruijn indexの強み

- 変数の名前を構文木から取り除くことができる

- 簡約の過程でのα変換を考えなくてよい

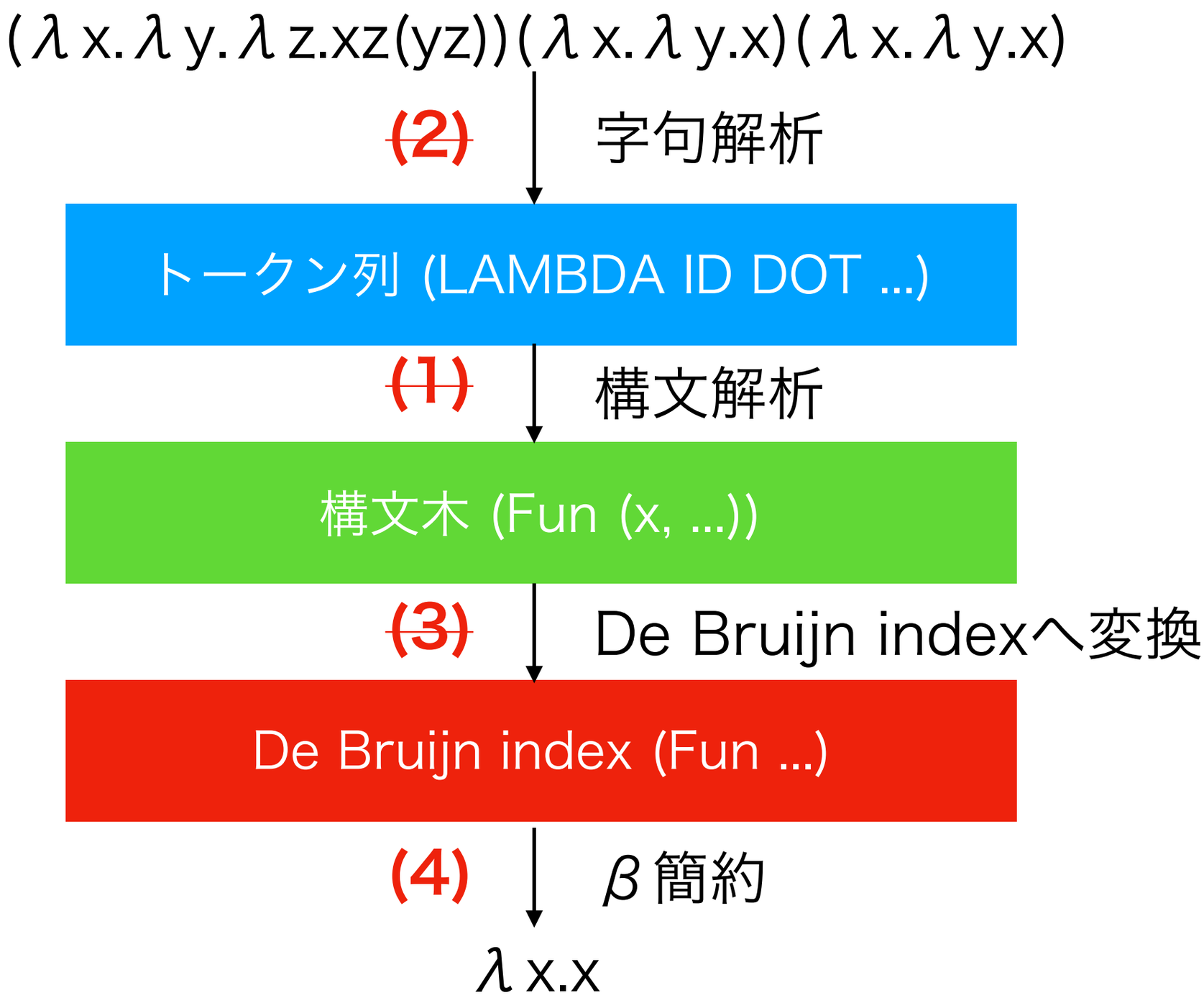
De Bruijn indexへの変換

- D(Γ, N, x) = N - Γ(x) (Γ(x)が定義されている)
- D(Γ, N, x) = x (Γ(x)が定義されない，自由変数である)
- D(Γ, N, λx.M) = λ.(D(Γ{x:N+1}, N+1, M))
- D(Γ, N, PQ) = (D(Γ, N, P))(D(Γ, N, Q))
- D_{main}(M) = D({}, 0, M)

De Bruijn indexへの変換

- 例
  - $D_{main}(\lambda\, x.\lambda\, y.x)$
  - $= D(\{\},0,\lambda\, x.\lambda\, y.x)$
  - $= \lambda\,.(D(\{x:1\},1,\lambda\, y.x))$
  - $= \lambda\,.\lambda\,.(D(\{x:1,y:2\},2,x))$
  - $= \lambda\,.\lambda\,.(2\text{-}1) = \lambda\,.\lambda\,.1$

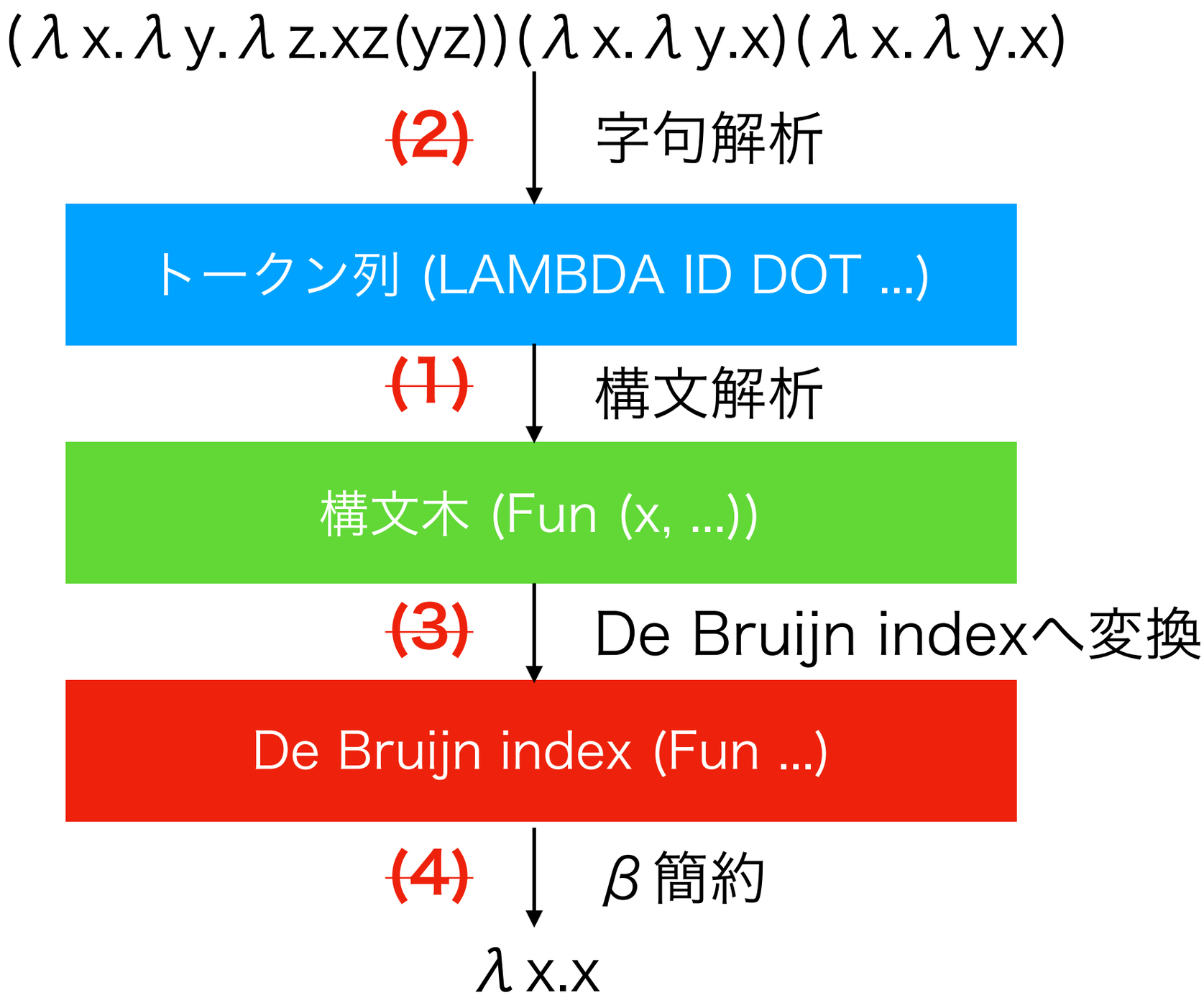
実装の流れ


(λx.λy.λz.xz(yz))(λx.λy.x)(λx.λy.x)
(2)
字句解析
トークン列 (LAMBDA ID DOT ...)
(1)
構文解析
構文木 (Fun (x, ...))
(3)
De Bruijn indexへ変換
De Bruijn index (Fun ...)
(4)
β簡約
λx.x

De Bruijn indexの β 簡約

- 機械的なindexの操作と置換だけで β 簡約が実行される
- (なんかすごい式)
- 資料[3]を参照

実装の流れ


(λx.λy.λz.xz(yz))(λx.λy.x)(λx.λy.x)
(2)
字句解析
トークン列 (LAMBDA ID DOT ...)
(1)
構文解析
構文木 (Fun (x, ...))
(3)
De Bruijn indexへ変換
De Bruijn index (Fun ...)
(4)
β簡約
λx.x

実装したもの

- https://github.com/utgwkk/lambda-chama

デモ

- (\x.\y.\z.xz(yz))(\x.\y.x)(\x.\y.x)
- (\x.\y.\z.xz(yz))(\x.\z.x)(\x.\y.x) (変数名を変えただけ)
- \f.(\x.f(xx))(\x.f(xx)) (停止しない)

今後の課題

- 型を付ける
  - 実装はしたが講座では省略
- 自由変数の扱いをうまくやりたい

まとめ

- λ計算についてなんとなく分かってきた

- λ計算のインタプリタを実装した