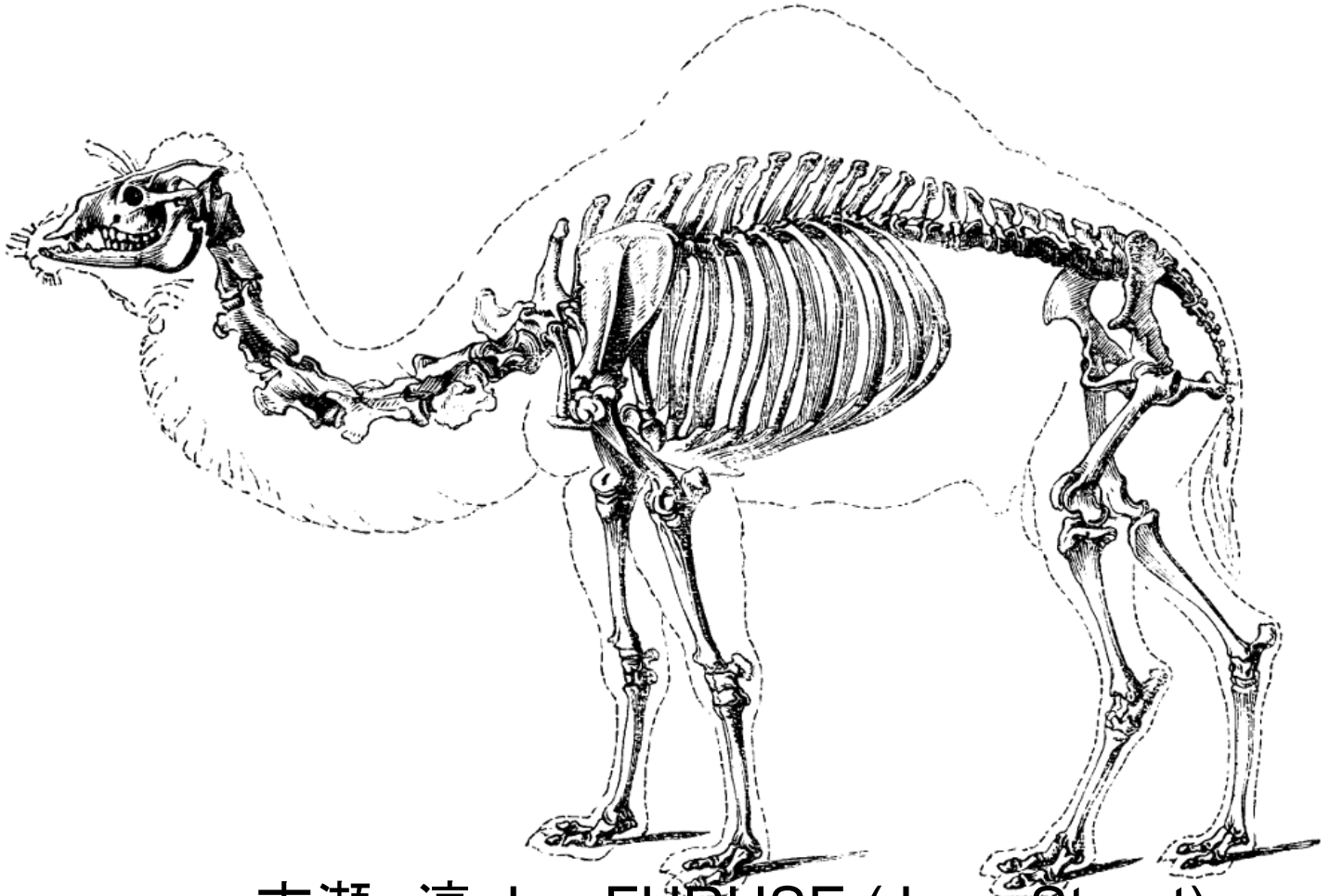
大駱駝解體變造概説

Ride your own OCaml: Quick OCaml Mod How-to

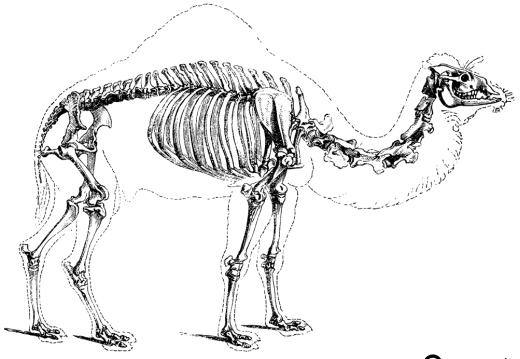


古瀬　淳 Jun FURUSE (Jane Street)

目的

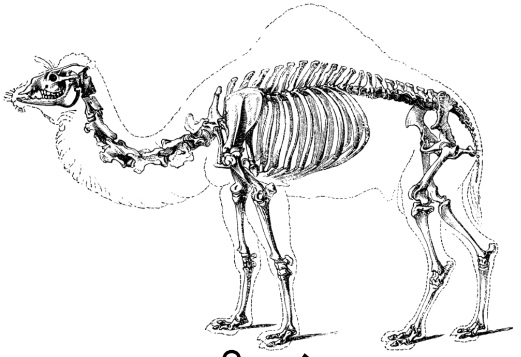
- OCaml コンパイラ内部を概観
- 実際の簡単な改造例を見てみる
- 改造すると面白いよ！

OCaml は OCaml で書く



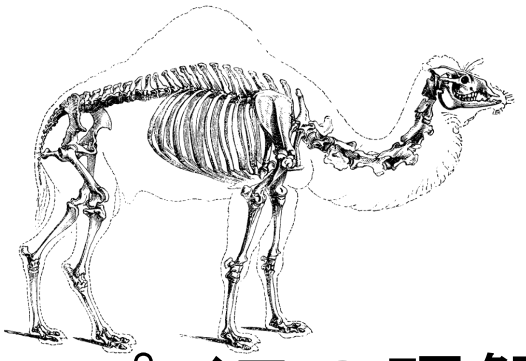
OCaml コンパイラ

Bootstrap



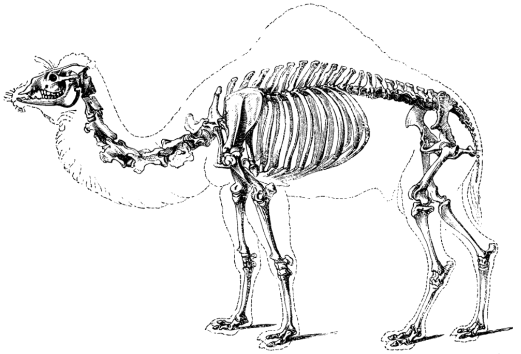
コンパイラソース
in OCaml

OCaml は OCaml で理解



コンパイラの理解

Bootstrap



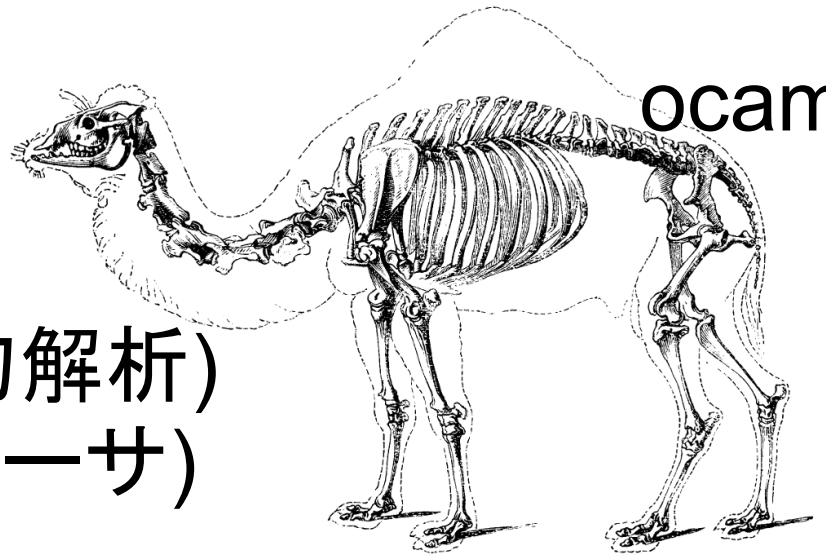
OCaml の理解

OCaml コンパイラシステム

ocamlc (バイトコードコンパイラ)

ocamlopt (ネイティブコードコンパイラ)

ocamlrun (ランタイム)



ocamllex (字句解析)
ocamlyacc (パーサ)

ocamldep (依存解析)

camlp4 (プリプロセッサ)

ocamlbuild (ビルドシステム)

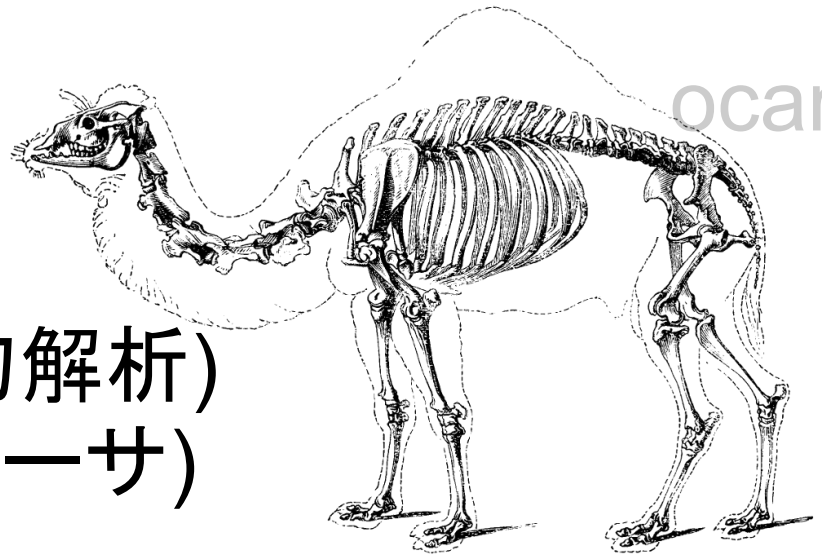
ocamldoc (ドキュメントシステム)

OCaml コンパイラシステム

ocamlc (バイトコードコンパイラ)

ocamlopt (ネイティブコードコンパイラ)

ocamlrun (ランタイム)



ocamllex (字句解析)
ocamlyacc (パーサ)

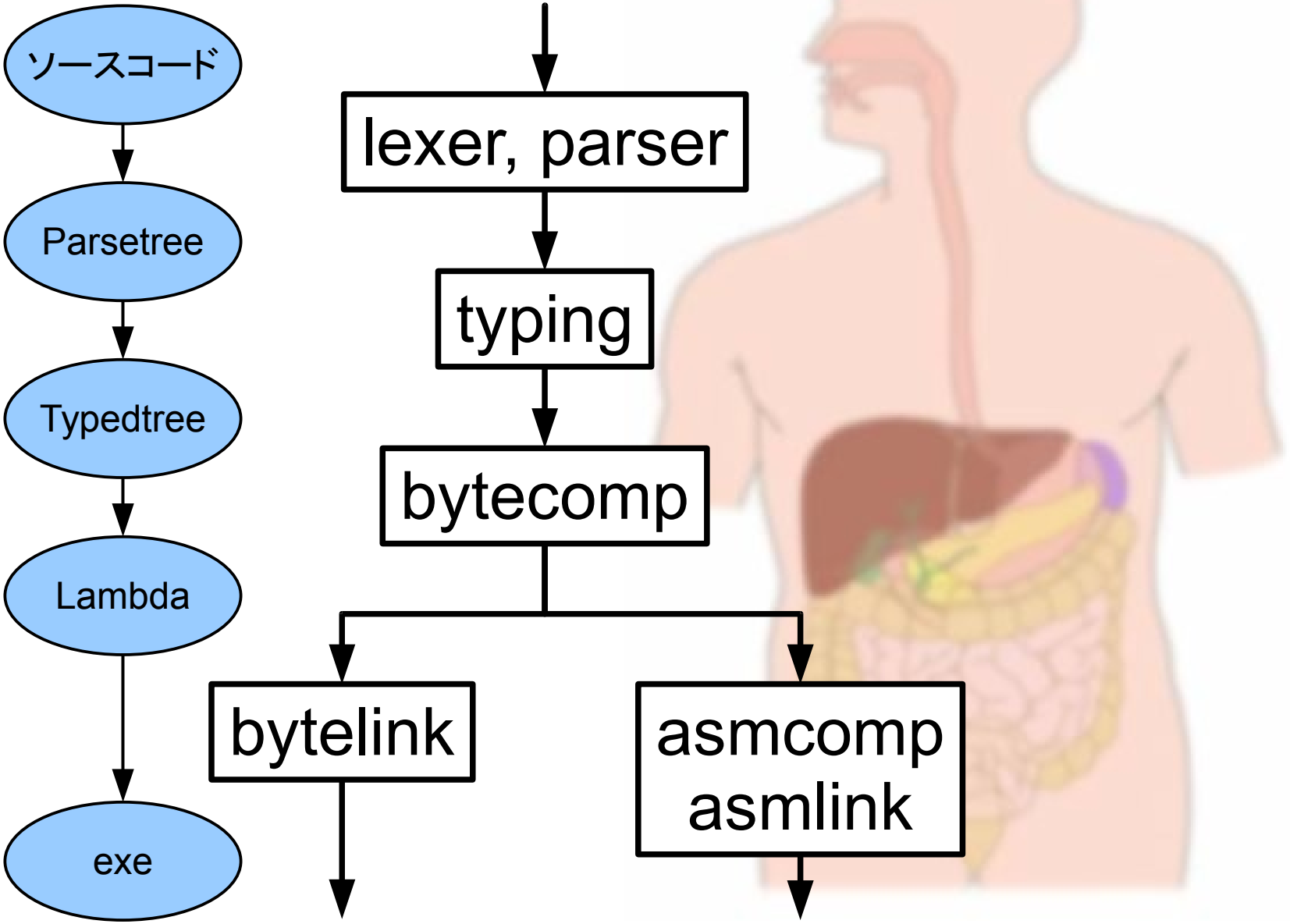
ocamldep (依存解析)

ocamlbuild (ビルドシステム) camlp4 (プリプロセッサ)

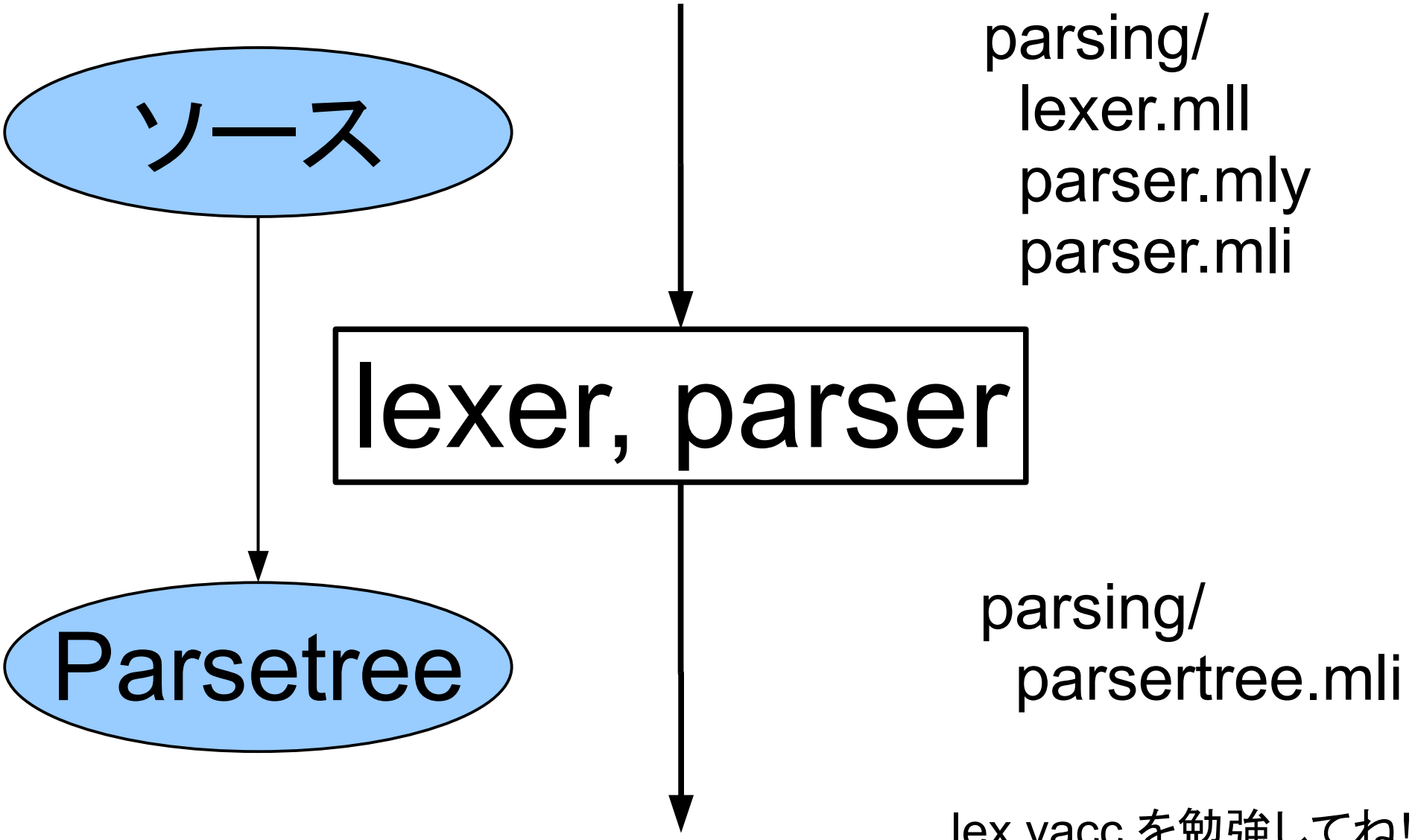
ocamldoc (ドキュメントシステム)

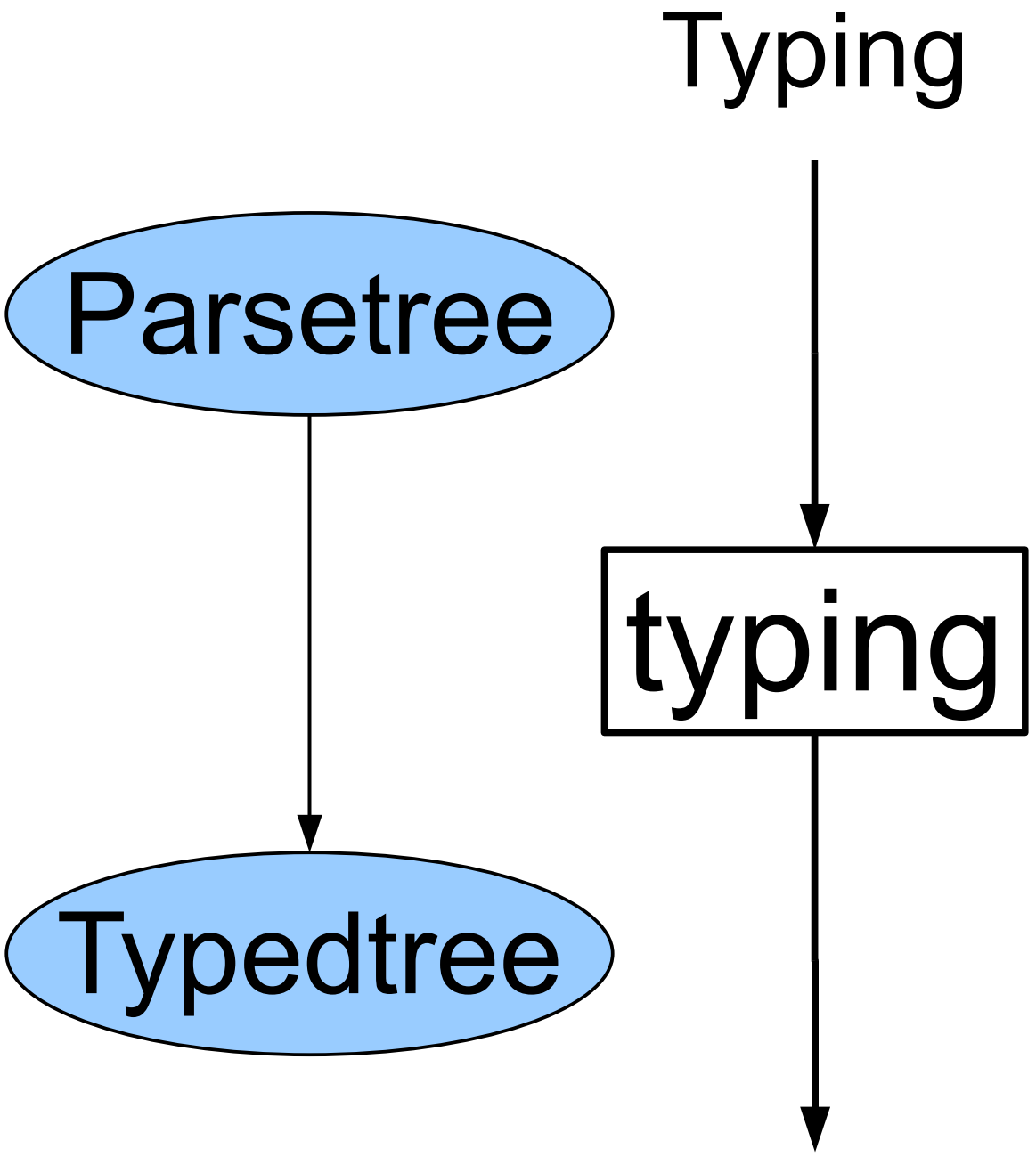
全部知る必要は(あまり)無い

OCaml コンパイラ内部



Lexer/Parser

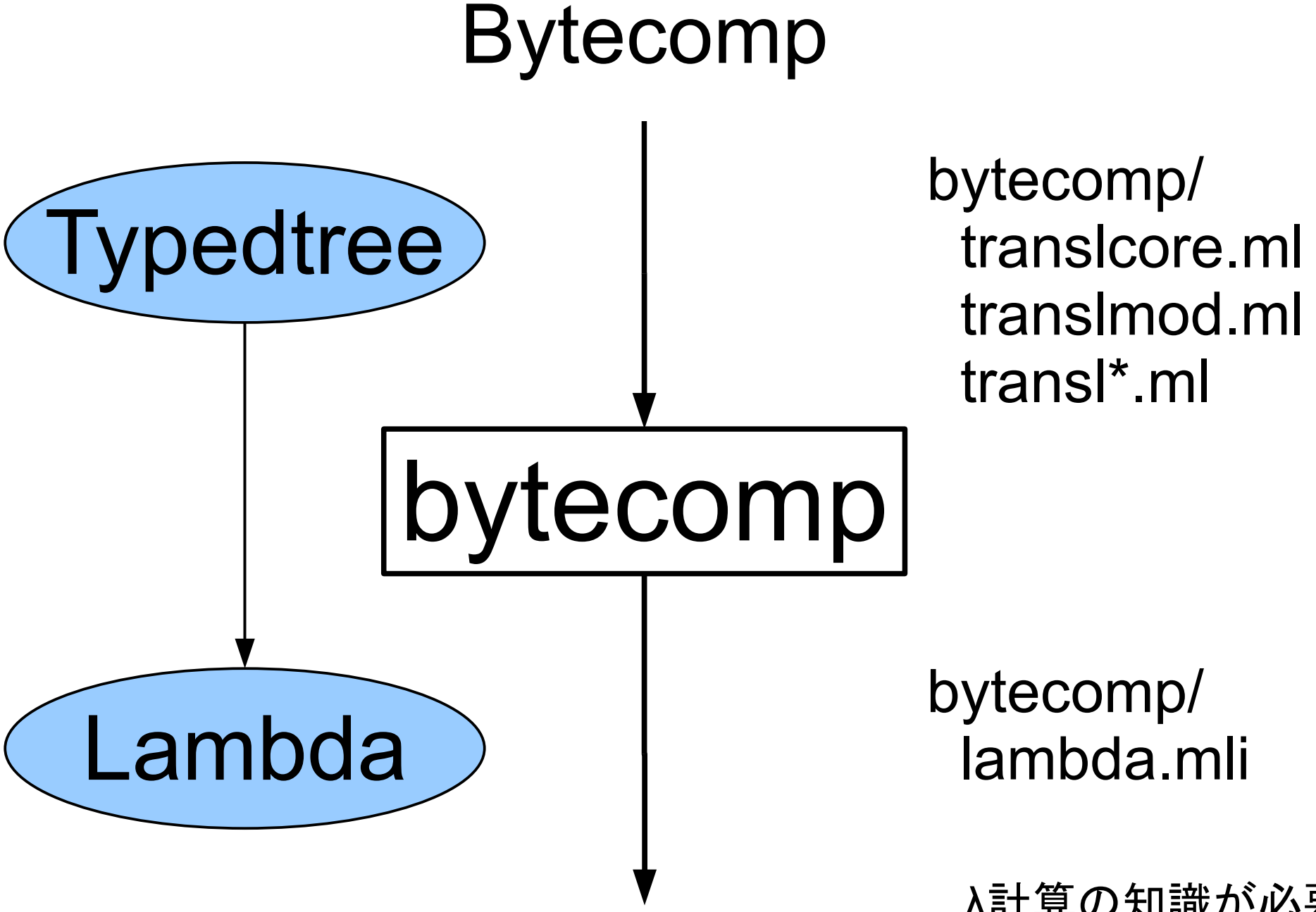




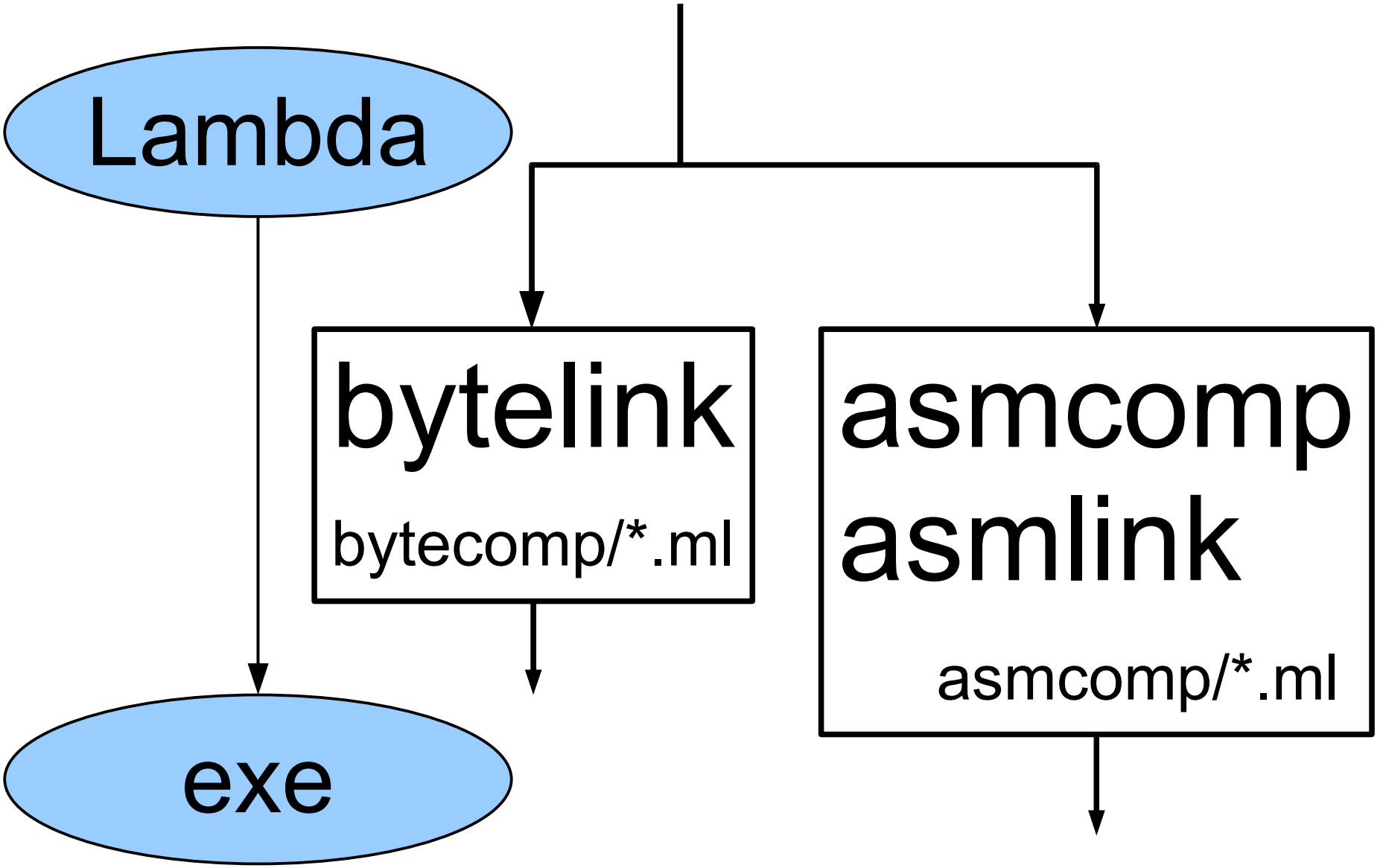
typing/
  typecore.ml
  typemod.ml
  type.ml

typing/
  typedtree.mli

さすがにML型システム
の知識がいる…


Bytecomp
Typedtree
Lambda
bytecomp
bytecomp/
translcore.ml
translmod.ml
transl*.ml
bytecomp/
lambda.mli
λ計算の知識が必要
ocamlc -dlambda

Bytecomp 以降は知らなくても大丈夫



じゃあ改造しよう！

(+) と (+.) がウザイよね!!

(+) は int 専用

```
# 1 + 1;;
- : int = 2
```

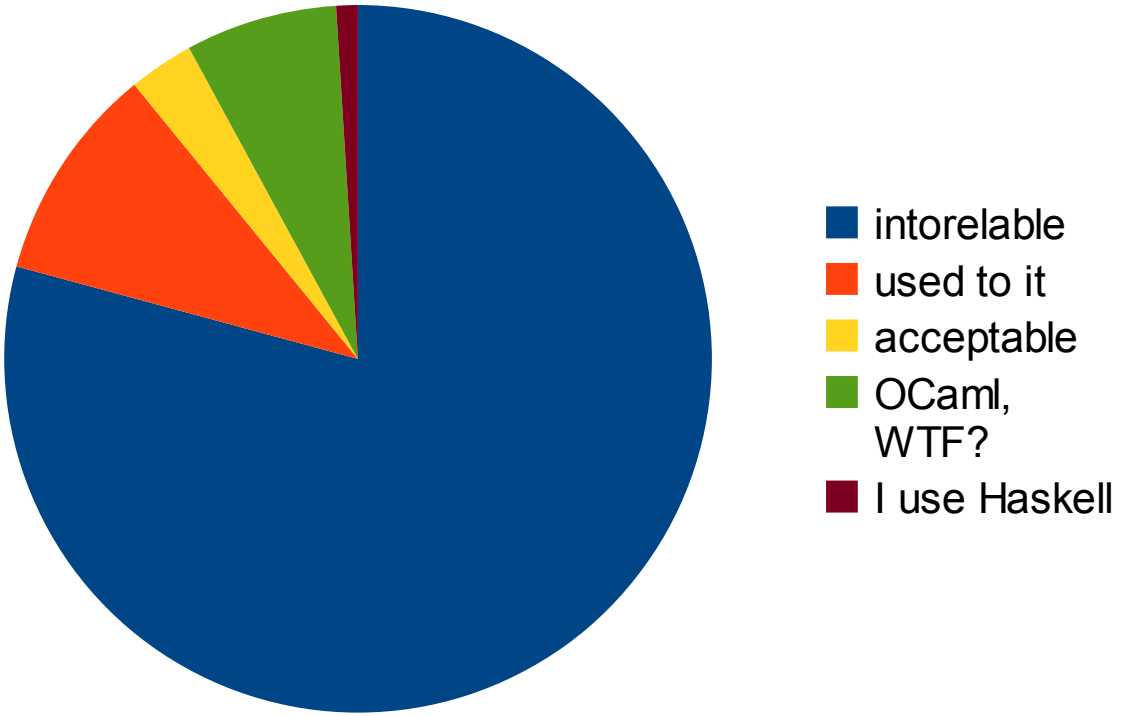
(+.) は float 専用

```
# 1.2 +. 3.4;;
- : float = 4.6
```

すぐ間違える

```
# 1.2 + 3.4;;
```

```
Characters 0-3:
Error: This expression has type float
but an expression was expected of type int
```



+/+. user report from an undetermined source

(+) と (+.) がウザイよね!!

+ 一本にしたい

SML '97 方式

(+) を int と float 両用に

(+) が int → int → int なら Pervasives.(+) に

```
# 1 + 2;;      ⇒  Pervasives.(+)  1  2
- : int = 3
```

(+) が float → float → float なら Pervasives.(+.)

```
# 1.2 + 3.4;;   ⇒ Pervasives.(+.)  1.2  3.4
- : float = 4.6
```

多分とっても嬉しい \(^o^)/

SML '97 方式

改造開始!!

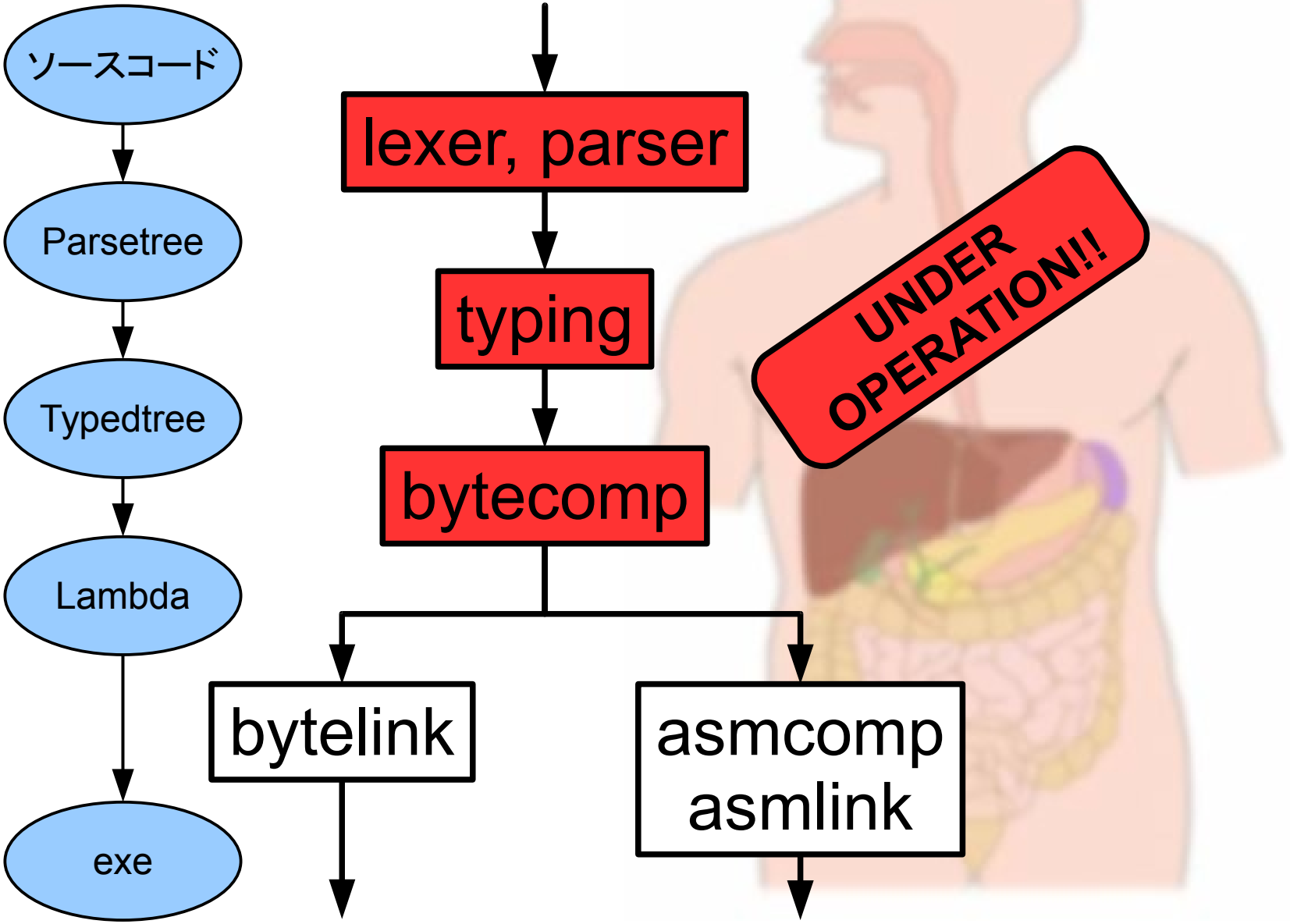
改造開始!!

保護者の方へ

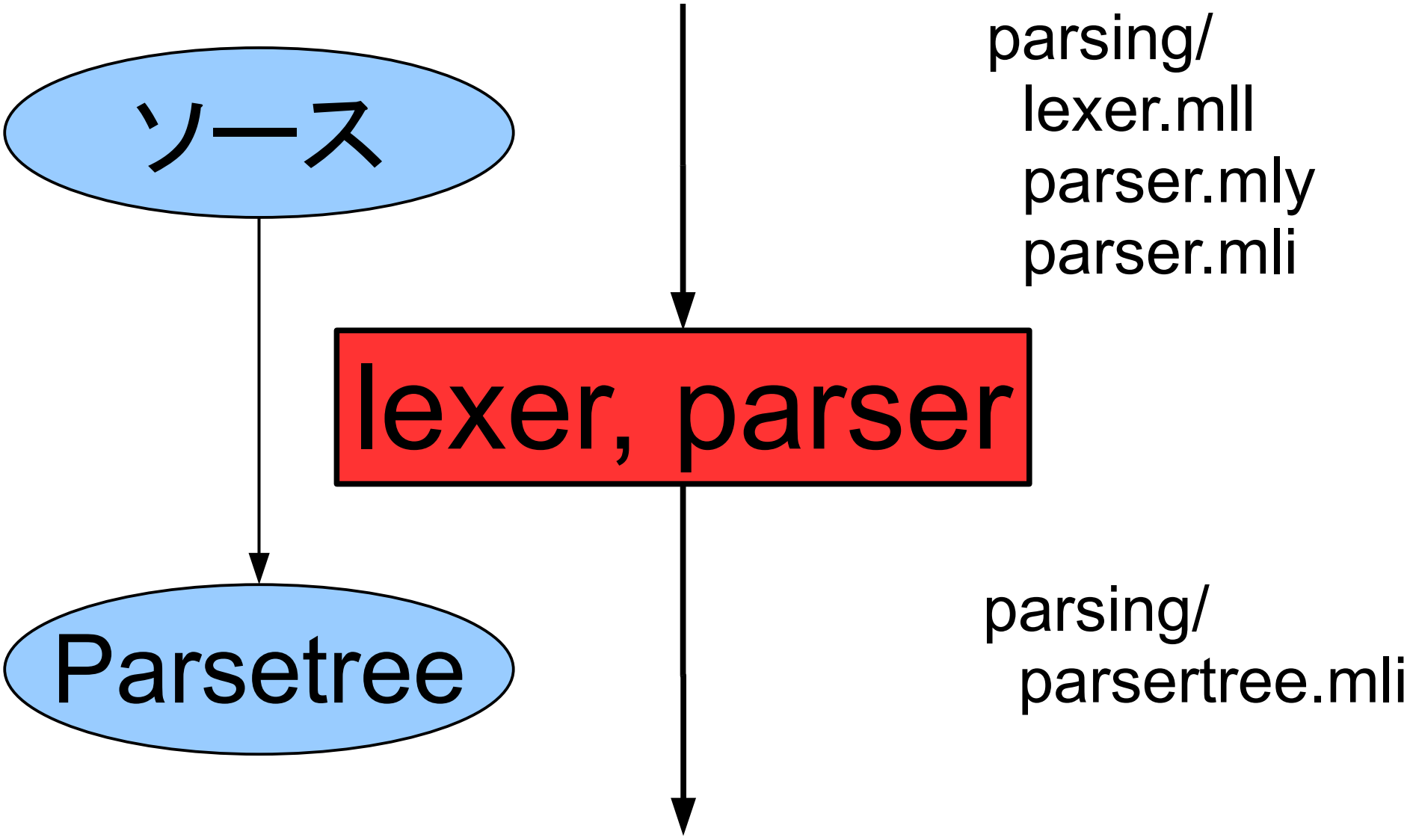
改造はとても危険な行為です。
お子様の自己責任で行なわせましょう

解説では全ての改造点は取り上げません
お子様に空気を読ませましょう

OCaml コンパイラ内部



Lexer/Parser



Parser: + を特別にする

parsing/parsetree.mli : expression_desc を拡張

```
type expression_desc =
    Pexp_ident of Longident.t
| Pexp_constant of constant
...
| Pexp_super_plus
```

Parser: + を特別にする

parsing/parser.mly に Pexp_super_plus の生成コードを追加

```
let mkoperator name pos =
  if name = "+" then
    { pexp_desc = Pexp_super_plus;
      pexp_loc = rhs_loc pos }
  else
    { pexp_desc = Pexp_ident(Lident name);
      pexp_loc = rhs_loc pos }
```

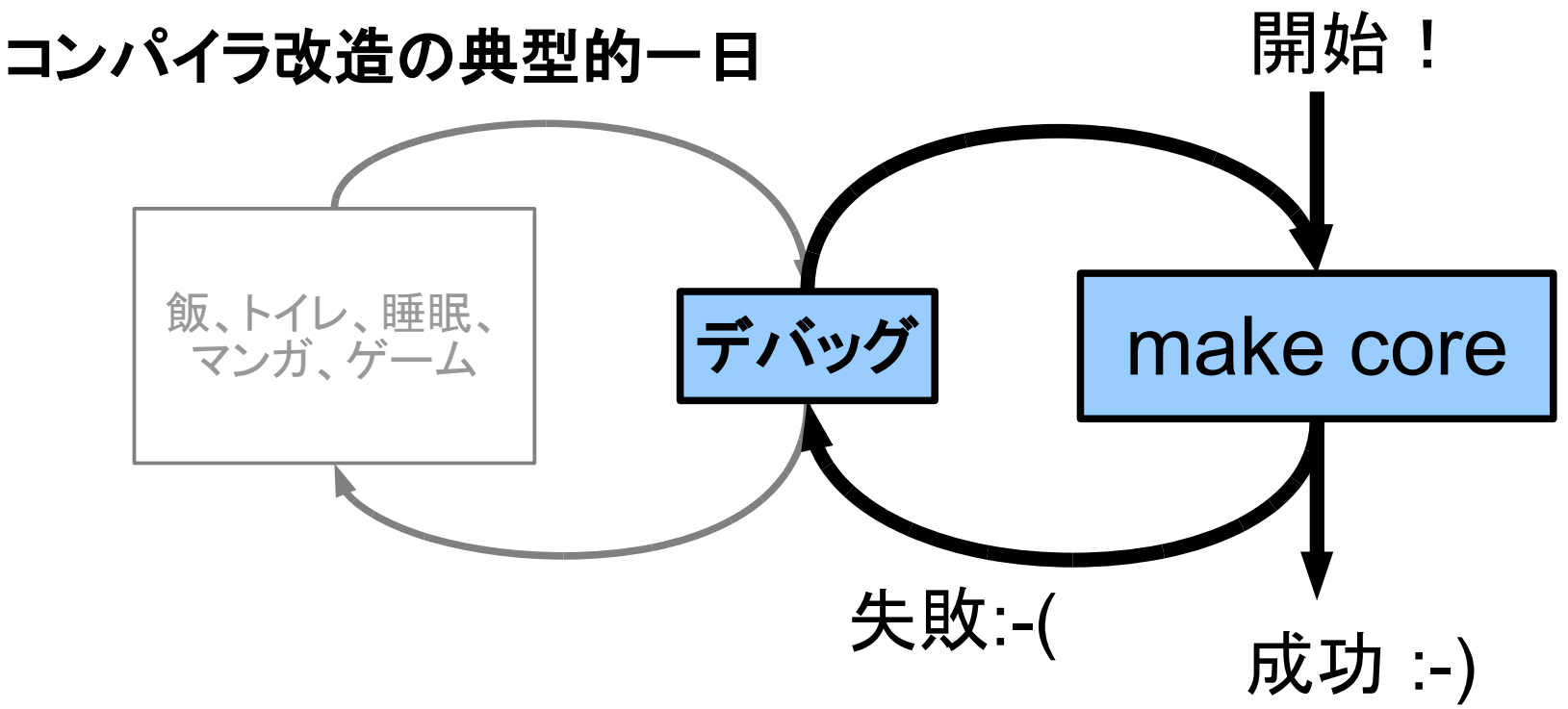
1 + 2 と書くと Pexp_Ident (Lident “+”) ではなく Pexp_super_plus に。

make core

コンパイラコアをコンパイル

make world は全システムコンパイルする。遅い。

make core はコアだけ。早い。



make core

$ make core

…
File "parsing/printast.ml", line 203, characters 2-3450:
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Pexp_super_plus
File "parsing/printast.ml", line 1, characters 0-1:
Error: Error-enabled warnings (1 occurrences)

未実装のところは埋めていく、で、
$ make core

make core

Typing 直前まで何とかたどり着く

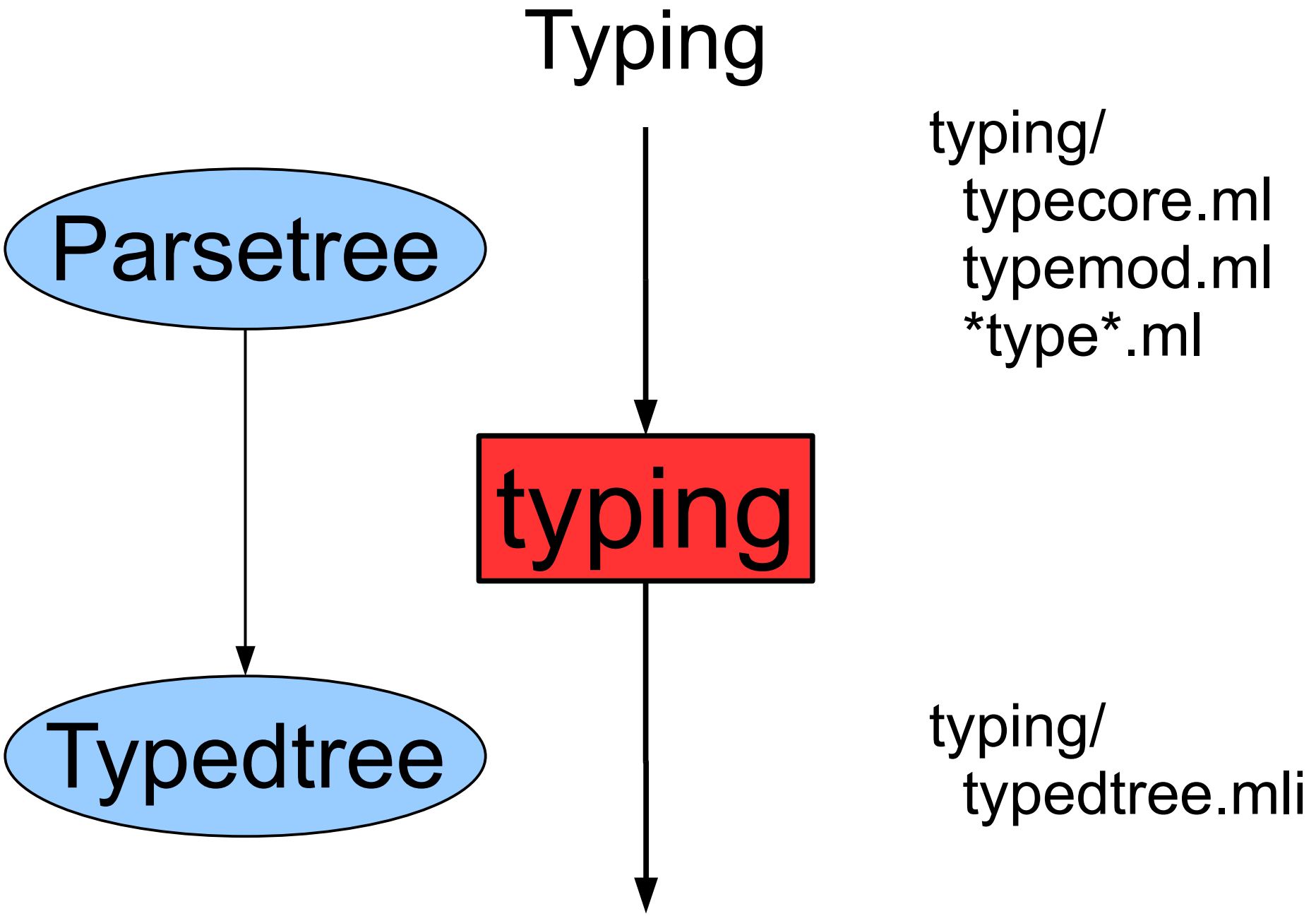
```
$ make core
   File "typing/typecore.ml", line 1018, characters 2-
   23911:
   Warning P: this pattern-matching is not exhaustive.
   Here is an example of a value that is not matched:
   Pexp_super_plus
   File "typing/typecore.ml", line 1, characters 0-1:
   Error: Error-enabled warnings (1 occurrences)
```

make core

$ make core

…
File "parsing/printast.ml", line 203, characters 2-3450:
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Pexp_super_plus
File "parsing/printast.ml", line 1, characters 0-1:
Error: Error-enabled warnings (1 occurrences)

未実装のところは埋めていく、で、
$ make core


Typing
Parsetree
typing
Typedtree
typing/
typecore.ml
typemod.ml
type.ml
typing/
typedtree.mli

Typedtree を拡張

Parsetree.Pexp_super_plus に対応する物を
Typedtree に追加

```
type expression_desc =
    Texp_ident of Path.t * value_description
  | Texp_constant of constant
  ...
  | Texp_super_plus
```

Typing rule を拡張

Pexp_super_plus を Texp_super_plus にするルールを　typecore.ml, type_expr 関数に追加

```
let rec type_exp env sexp =
    match sexp.pexp_desc with
    | Pexp_ident lid →
        …
    | Pexp_super_plus →
      ???
```

Pexp_super_plus の型付け

1 + 2 だったら int → int → int

1.2 + 3.4 だったら float → float → float

じゃあ、それ以外は？

$$(\text{fun } x \rightarrow x + x)\ e$$

e の型が決まらないと + の型が決められない。
「**後で考える**」の型が必要

「後で考える」の型: 型変数

ML型システムのお約束: 未定の型は変数

fun x → x + x　　+ は 'a → 'a → 'a

(fun x → x + x) 2　　実は 'a は int でした。
+ は int → int → int に

(fun x → x + x) 1.2　　実は 'a は float だった。
+ は float → float → float

実は 'a は 〇〇でした = 単一化(unification)

型: types.mli

```
type type_expr =
    { mutable desc: type_desc;
      mutable level: int; }

and type_desc =
      Tvar
    | Tarrow of type_expr * type_expr
    | Ttuple of type_expr list
    | Tconstr of Path.t * type_expr list
    | Tlink of type_expr
```

型変数

t → t

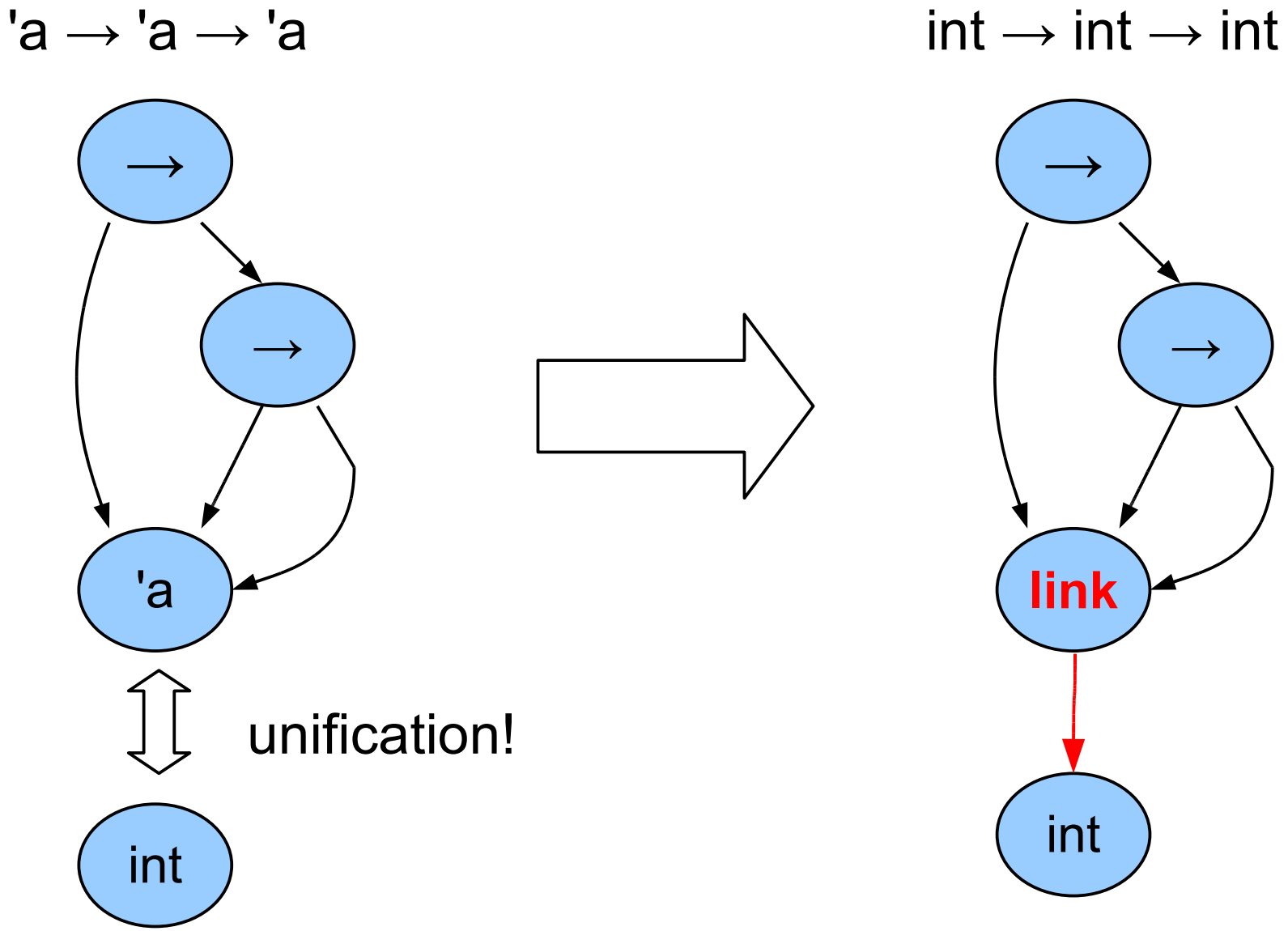
t * t

int, float

リンク

一部省略

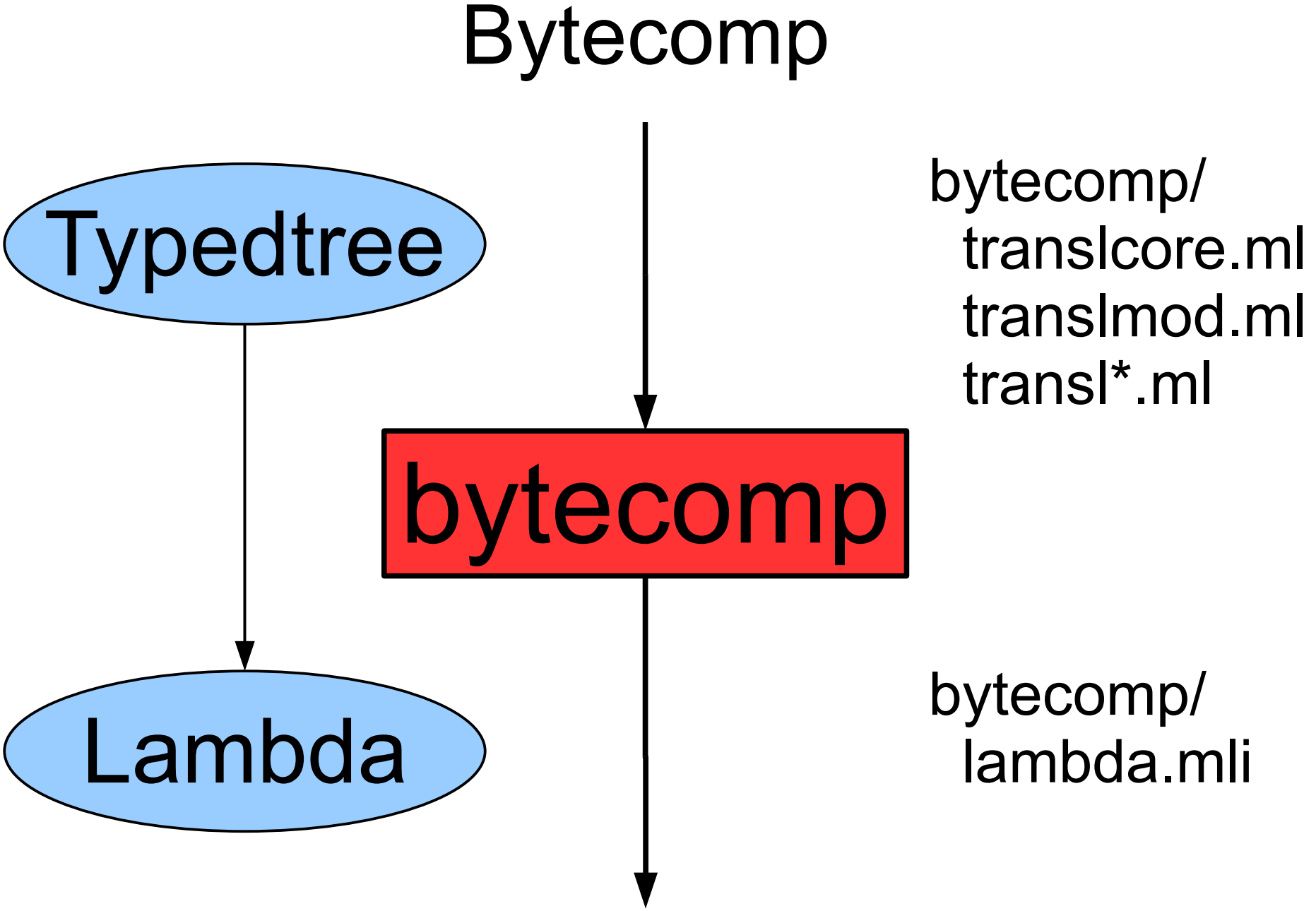
型: Tlink による unification



Pexp_super_plus の型付

```
let rec type_exp env sexp =  match sexp.pexp_desc with
      …
    | Pexp_super_plus →
      (* 'a を作る *)
      let tvar = Btype.newty2 Btype.lowest_level Tvar in
      let typ = (* 'a → 'a → 'a を作る *)
        Ctype.newty (Tarrow (tvar,
                             Ctype.newty (Tarrow(tvar, tvar)))
      in
      { exp_desc = Texp_super_plus;
        exp_type = typ; }                    (* type は typ *)
```

一部省略



好き勝手やっても最後に Lambda をちゃんと作ればつじつまが合う。

Texp_super_plus をコンパイルする

translcore.ml : transl_exp を改造

Typedtree.expression → Lambda.lambda

Texp_super_plus のケースが未実装なので実装する:
型が:

Int → int → int なら %addint にコンパイル

float → float → float なら %addfloat にコンパイル

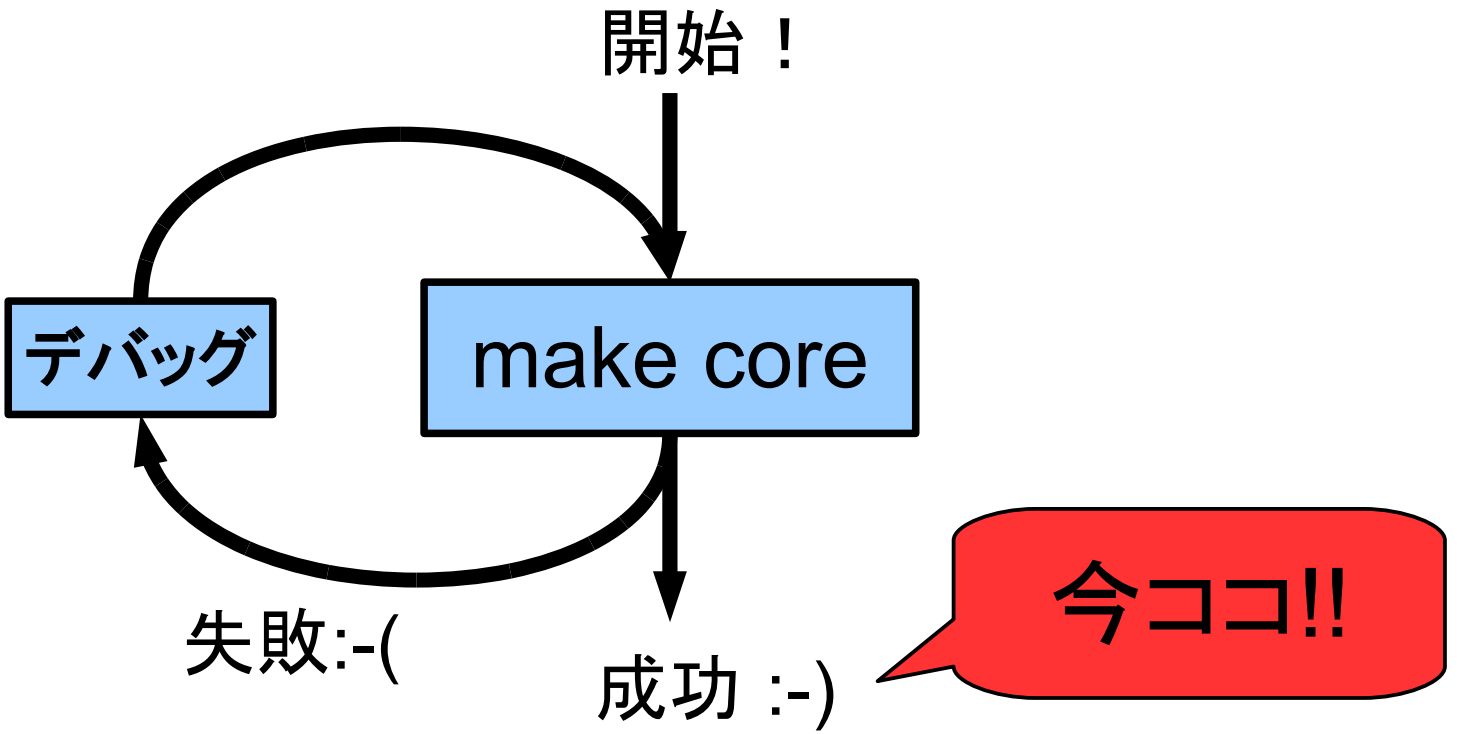
それ以外はゴメン、エラーだわ。

Lambda.lambda にコンパイル

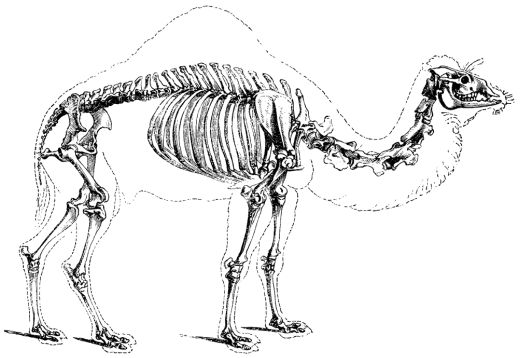
```
let rec transl_exp e = match e.exp_desc with
    | Texp_ident(path, {val_kind = Val_prim p}) → ...
    | Texp_super_plus →
      let ty = match e.exp_type.desc with
        | Tarrow (_, ty, _, _) →  ty     (* ty → ty → ty  *)
        | _ →  assert false in
      let prim = match ty.desc with    (* ty での場合分け *)
        | Tconstr (p, [], _) when Predef.path_int = p →
          { Primitive.prim_name = "%addint"; }
        | Tconstr (p, [], _) when Predef.path_float = p →
          { Primitive.prim_name = "%addfloat"; }
        | _ →  ERROR にする
      in
       transl_primitive prim
```

一部省略

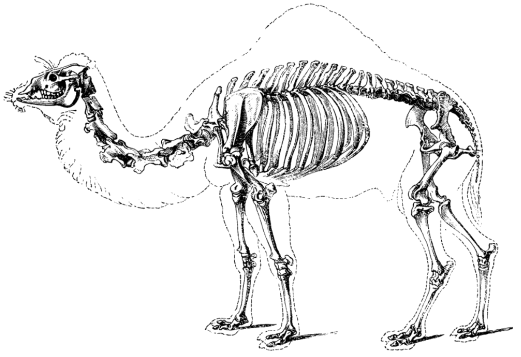
make core 成功!!



make coreboot



Bootstrap



make coreboot

- Bootstrap に失敗したら make restore
  - よく Makefile を確認したほうがよい
- Bootstrap に成功したら make ocaml
  - ocaml toplevel
- テストしよう!!

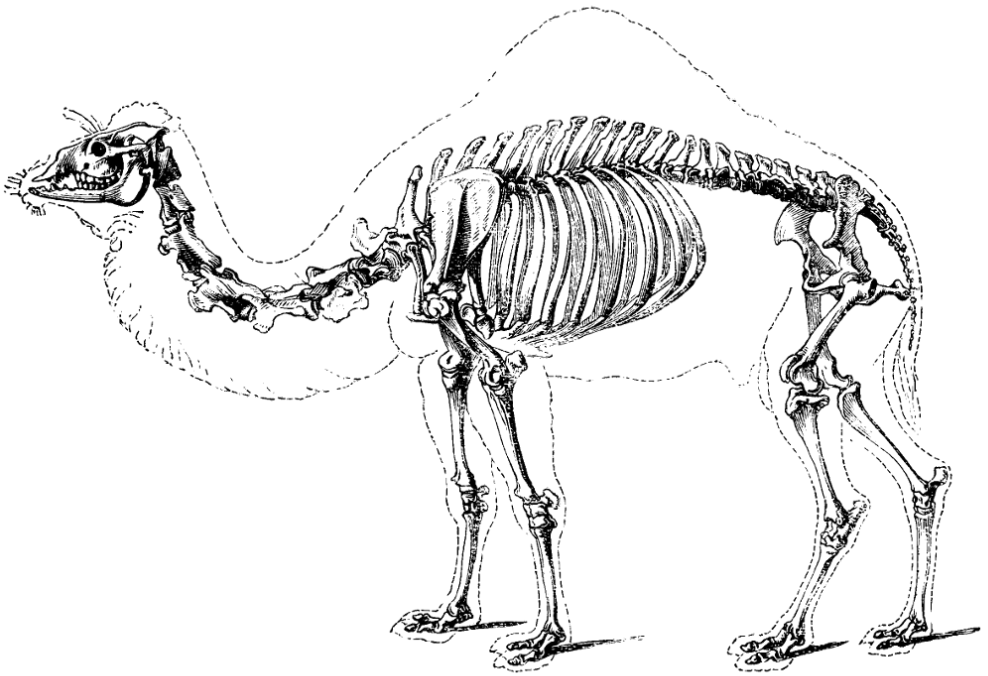
```
# 1 + 1;;
- : int = 2
# 1.2 + 3.4;;
- : float = 4.6
```

make world opt opt.opt

- ライブラリ等をコンパイルしていく過程で改造の問題点を見つかるかも
- ただし、オブジェクトはあまり使っていないので、テスト不足になりがち
  - その上、オブジェクト回りは型が面倒なのでエラーが起きやすい
    - 私は LablGTK をコンパイルしてテストしてみる

できたー!!

- Q ライセンスだから、公開するがいいさ。



触れなかったこと

- let double x = x + x をどうするか
  - double 1 も double 1.2 も動くようにする:
    - let double x = x + x in
      double 1, double 1.2
    - ホントの overloading これはむずい
  - (let double x = x + x in double 1,
    let double x = x + x in doube 1.2);;
    - これはできるけど、型変数のレベルをうまく扱ってやらないといけない。ちょっと面倒くさい。

触れなかったこと

- Polymorphism と型変数のレベル

```
type type_expr =
  { mutable desc: type_desc;
    mutable level: int; }
```

  - Level = let polymorphism 入れ子レベル
  - 型変数のレベル = 型変数が出現する一番外側の let 入れ子レベル
  - 理解してないと polymorphic な所が改造できない

チャレンジ

- もっと一般的に好きな物を overload する。
- ホントの overloading を作る:
  double : { Num 'a } => 'a -> 'a -> 'a
- OCaml に正式採用してもらう。
  - これが一番むずい。