DIFFERENCES

- Eliminattion of Nested Expressions
  - Expression → variable reference
  - Primitive operation
  - The condition part of the `if` expression
  - Function application: both the functional and arguments parts
  - Tuple constructor
  - Array constructor/modifier
- Boolean constants: false/true → 0/1

  AST

  KNF: only have int and float/no boolean
- Only two types of comparison: $=$ and $\leq$

```
(* Type annotations are stripped and "Ti#" => Ti *)
Let(Ti5, Let(Ti3, Let (Ti1, Int 1, Let (Ti2, Int 2, Add(Ti1, Ti2))),
         Let (Ti4, Int 3, Add(Ti3, Ti4))))

(* or in OCaml *)
let t5 =
  let t3 =
    let t1 = 1 in
    let t2 = 2 in
    t1 + t2 in
let t4 = 3 in t3 + t4
```

bind numbers to var

expression

$\mathcal{K} : \texttt{Syntax.t} \to \texttt{KNormal.t}$

$$
\begin{aligned}
\mathcal{K}(c) &= c \\
\mathcal{K}(\texttt{not}(e)) &= \mathcal{K}(\texttt{if}\ e\ \texttt{then false else true}) \\
\mathcal{K}(e_1 = e_2) &= \mathcal{K}(\texttt{if}\ e_1 = e_2\ \texttt{then true else false}) \\
\mathcal{K}(e_1 \leq e_2) &= \mathcal{K}(\texttt{if}\ e_1 \leq e_2\ \texttt{then true else false}) \\
\mathcal{K}(op(e_1, \ldots, e_n)) &= \texttt{let}\ x_1 = \mathcal{K}(e_1)\ \texttt{in}\ \ldots\ \texttt{let}\ x_n = \mathcal{K}(e_n)\ \texttt{in}\ op(x_1, \ldots, x_n)
\end{aligned}
$$

op が論理演算・比較以外の場合

when op is not a logical nor comparison operator

$$
\begin{aligned}
\mathcal{K}(\texttt{if not}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= \mathcal{K}(\texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2) \\
\mathcal{K}(\texttt{if}\ e_1 = e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4) &= \texttt{let}\ x = \mathcal{K}(e_1)\ \texttt{in let}\ y = \mathcal{K}(e_2)\ \texttt{in} \\
&\ \texttt{if}\ x = y\ \texttt{then}\ \mathcal{K}(e_3)\ \texttt{else}\ \mathcal{K}(e_4) \\
\mathcal{K}(\texttt{if}\ e_1 \leq e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4) &= \texttt{let}\ x = \mathcal{K}(e_1)\ \texttt{in let}\ y = \mathcal{K}(e_2)\ \texttt{in} \\
&\ \texttt{if}\ x \leq y\ \texttt{then}\ \mathcal{K}(e_3)\ \texttt{else}\ \mathcal{K}(e_4) \\
\mathcal{K}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= \mathcal{K}(\texttt{if}\ e_1 = \texttt{false}\ \texttt{then}\ e_3\ \texttt{else}\ e_2)
\end{aligned}
$$

e_1 が論理演算・比較以外の場合

when e1 is not a logical/ comparison operator

compare with false(0) is faster than compare with true(1)

$\alpha : \texttt{Id.t M.t} \rightarrow \texttt{KNormal.t} \rightarrow \texttt{KNormal.t}$

mapping
identifier->identifier mapping

$$
\begin{array}{lcl}
\alpha_\varepsilon(c) & = & c \\
\alpha_\varepsilon(op(x_1,\ldots,x_n)) & = & op(\varepsilon(x_1),\ldots,\varepsilon(x_n)) \\
\alpha_\varepsilon(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } \varepsilon(x) = \varepsilon(y) \texttt{ then } \alpha_\varepsilon(e_1) \texttt{ else } \alpha_\varepsilon(e_2) \\
\alpha_\varepsilon(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } \varepsilon(x) \leq \varepsilon(y) \texttt{ then } \alpha_\varepsilon(e_1) \texttt{ else } \alpha_\varepsilon(e_2) \\
\alpha_\varepsilon(\texttt{let } x = e_1 \texttt{ in } e_2) & = & \texttt{let } x' = \alpha_\varepsilon(e_1) \texttt{ in } \alpha\boxed{_{\varepsilon,x\mapsto x'}(e_2)} \\
\alpha_\varepsilon(x) & = & \varepsilon(x) \\
\alpha_\varepsilon(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & \texttt{let rec } x'\ y'_1\ \ldots\ y'_n = \alpha_{\varepsilon,x\mapsto x',y_1\mapsto y'_1,\ldots,y_n\mapsto y'_n}(e_1) \texttt{ in} \\
 & & \alpha_{\varepsilon,x\mapsto x'}(e_2) \\
\alpha_\varepsilon(x\ y_1\ \ldots\ y_n) & = & \varepsilon(x)\ \varepsilon(y_1)\ \ldots\ \varepsilon(y_n) \\
\alpha_\varepsilon((x_1,\ldots,x_n)) & = & (\varepsilon(x_1),\ldots,\varepsilon(x_n)) \\
\alpha_\varepsilon(\texttt{let } (x_1,\ldots,x_n) = y \texttt{ in } e) & = & \texttt{let } (x'_1,\ldots,x'_n) = \varepsilon(y) \texttt{ in } \alpha_{\varepsilon,x_1\mapsto x'_1,\ldots,x_n\mapsto x'_n}(e) \\
\alpha_\varepsilon(x.(y)) & = & \varepsilon(x).(\varepsilon(y)) \\
\alpha_\varepsilon(x.(y) \leftarrow z) & = & \varepsilon(x).(\varepsilon(y)) \leftarrow \varepsilon(z)
\end{array}
$$

x can occur in e2.
when alpha convert e2
all the x occurs in e2
should change to x'

図 6: α 変換。ε は α 変換前の変数を受け取って、α 変換後の変数を返す写像。右辺に出現していて左辺に出現していない変数（x' など）は、すべて fresh とする。

α conversion: ε is a mapping that takes a variable name and gives its α-converted name.
A name that occurs only in RHS should be considers new/fresh name.

alpha
conversion

```
let rec g env = function (* alpha conversion *)
  | ...
  | FAdd(x, y) -> FAdd(find x env, find y env) renaming
  | Let((x, t), e1, e2) ->
      let x' = Id.genid x in generate new var x' to represent x
      Let((x', t), g env e1, g (M.add x x' env) e2)extended env for x in e1 applied to alpha conversion of e2
  | Var(x) -> Var(find x env)
  | LetRec({ name = (x, t); args = yts; body = e1 }, e2) ->
      let env = M.add x (Id.genid x) env in env extended by x with the same name
      let ys = List.map fst yts in
      let env' = M.add_list2 ys (List.map Id.genid ys) env in
      LetRec({ name = (find x env, t);
               args = List.map (fun (y, t) -> (find y env', t)) yts;
               body = g env' e1 },
             g env e2)
```

Alpha conversion tries to disambiguiate the use of variable