プログラム言語論
制御構造と型システム

亀山幸義

筑波大学コンピュータサイエンス専攻

筑波大学 情報科学類 講義, 2010 年 5 月 24 日

先週の課題 1

Fibonacci 関数

```
let rec fib n =
    if n = 0 then 1
    else if n = 1 then 1
    else (fib (n+(-2))) + (fib (n+(-1)))
in fib 5 ;;
```

Fibonacci 関数の別の定義

```
let rec fib2 n =
    if n = 0 then (0,1)
    else
      let x = fib2 (n+(-1)) in
        (snd x,  (fst x) + (snd x))
in snd (fib2 5) ;;
```


先週の課題 1

```
let rec fib n =
    if n = 0 then 1
    else if n = 1 then 1
    else (fib (n+(-2))) + (fib (n+(-1)))
in fib 5 ;;

fib 5 = (fib 3) + (fib 4) = 8
fib 4 = (fib 2) + (fib 3) = 5
fib 3 = (fib 1) + (fib 2) = 3
fib 2 = (fib 0) + (fib 1) = 2
fib 0 = fib 1 = 1
```


先週の課題 1

```
let rec fib2 n =
    if n = 0 then (0,1)
    else
      let x = fib2 (n+(-1)) in
        (snd x,  (fst x) + (snd x))
in snd (fib2 5) ;;

fib2 0 =(0, 1)
fib2 1 =(snd(fib2 0), fst(fib2 0)+snd(fib2 0)) =(1,1)
fib2 2 =(snd(fib2 1), fst(fib2 1)+snd(fib2 1)) =(1,2)
fib2 3 =(snd(fib2 2), fst(fib2 2)+snd(fib2 2)) =(2,3)
fib2 4 =(snd(fib2 3), fst(fib2 3)+snd(fib2 3)) =(3,5)
fib2 5 =(snd(fib2 4), fst(fib2 4)+snd(fib2 4)) =(5,8)

(fib2 n)=(fib n, fib (n+1))
```


先週の課題 1

fib と fib2 は実質的に同じ計算？

- YES! `(fib2 n)=(fib n, fib (n+1))`
- NO! `fib2` の方がずっと高速.
  - `(fib 10)` において `(fib 8)` の計算は 2 回.
  - `(fib2 10)` において `(fib2 8)` の計算は 1 回.
  - `(fib 10)` において `(fib 6)` の計算は 5 回.
  - `(fib2 10)` において `(fib2 6)` の計算は 1 回.
  - `(fib 10)` において `(fib 4)` の計算は 13 回.
  - `(fib2 10)` において `(fib2 4)` の計算は 1 回.

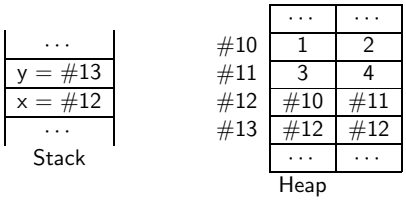
先週の課題 1

答. fib と fib2 はどちらも Fibonacci 関数を計算するが，前者が同じ計算を何度も繰返すために非常に効率が悪いのに対して，後者は同じ計算を繰返さず，引数 n に対して線形の時間 (O(n)) で計算が終了する.

先週の課題 2

問. heap の使われ方を説明せよ.

```
let x = ((1,2),(3,4)) in
let y = (x,x) in
let _ = show 1 in
   y ;;
```



先週の課題 2 補足

- stack は、(プログラム上の) ブロック構造に対応する。ブロック構造と lifetime が等しい変数 (局所変数) の値が stack に格納される。
- heap は、ブロック構造に対応しないデータを格納する。
- C 言語などでは、heap を獲得したり、獲得した heap 領域を返却したりするのはプログラマの責任である。
  - malloc(), free()
- 多くの関数型言語やオブジェクト指向言語では、heap を獲得してデータを置いたり、その領域が不要になった時に開放したりするのは言語処理系が行う。
  - ごみ集め (Garbage Collection)
  - もっとも C++ のように，ごみ集めをしないオブジェクト指向言語もある

先週の課題 3

問. MiniML 処理系のモード 0,1,2,3,4,5 はどういう計算を表すか.

```
let x=5 in
  (let f=(fun y->x+y) in
    let x=10 in
     (f 20) + x * 2)
  + x * 3 ;;
```

- 動的束縛 vs 静的束縛
  - 動的束縛ならば、「65」が返る。
  - 静的束縛ならば、「60」が返る。

先週の課題 3

```
let f = fun x -> print x in
  let g = fun x -> ((x , x) , x) in
    g (f 5);;

let rec f x = f x in
  let g = fun x -> 0 in
    g (f 10);;
```

- 値呼びか、名前呼びか、必要呼びか。
  - 値呼び: 上のプログラムで「5」が 1 回プリント。
  - 名前呼び: 上のプログラムで「5」が 3 回プリント。
  - 必要呼び: 上のプログラムで「5」が 1 回プリント。
  - 値呼び: 下のプログラムが止まらない。
  - 名前呼び: 下のプログラムが「0」を返す。
  - 必要呼び: 下のプログラムが「0」を返す。

制御構造

- (先週) 末尾再帰
- (今週) 制御構造，例外，継続

制御構造

```
 10   IF (X .GT. 0.000001) GO TO 20
      X=-X
 11   Y=X*X-SIN(Y)/(X+1)
      IF (X .LT. 0.000001) GO TO 50
 20   IF (X*Y .LT. 0.000001) GO TO 30
      X=X-Y-Y
 30   X=X+Y
      ...
 50   CONTINUE
      X=A
      Y=B-A+C*C
      GO TO 11
```

スパゲッティ・コード (Mitchell, “Concepts in PL”, 2003 より)

構造化プログラミング

Dijkstra, “GO TO CONSIDERED HARMFUL” (1968)

- go to 文を多用したプログラムは理解しづらい.
- かわりに，より「構造的」な制御文を使うべき.
- if-then-else, while, for, case ...

さらに先の制御構造は?

例外 (exception) 機構

```
open List;;

let rec mult x =
  if x=[] then 1
  else (hd x) * (mult (tl x))
in mult [2; -3; 5]  ;;
```

リストの要素の積を返す関数.

例外

「リストの要素に負の数があったら，(積ではなく) その要素を返す」ことにしたい.

```
let rec mult2 x =
  if x=[] then 1
  else if (hd x) <=0 then (hd x)
  else (hd x) * (mult (tl x))
in mult [2; -3; 5]  ;;
```

これではうまくいかない.

例外

```
exception Negative of int ;; (例外の宣言)

try
  let rec mult3 x =
    if x=[] then 1
    else if (hd x) <= 0 then
        raise (Negative (hd x))
    else (hd x) * (mult3 (tl x))
  in
    mult3 [2; -3; 5]
with
  Negative n -> n
;;
```

raise (例外の発生) を実行すると，対応する try-with (例外の処理) までジャンプする.

例外機構

```
exception E1 of int;;
exception E2 of int;;
let f x =
   if x=0 then raise (E1 x)
   else x
in let g x =
   if x=1 then raise (E2 x)
   else f x
in let h x =
   try g x
   with E1 x -> x+10
in let i x =
   try h x
   with E2 x -> x+20
in i 0
```

- 2つ以上の関数呼び出しから一気に抜けることができる

例外機構は動的束縛

```
exception E3;;
try
  let f y = raise E3 in
  let g h =
    try h 1 with E3 -> 2
  in
    try g f with E3 -> 4
with E3 -> 6;;
```

- 答は「2」; raise と try-with の対応は動的に決まる.
- 例外処理部 (with 以下) は，スタックに積まれる.
- raise の処理の際に，スタックの active でない部分も見なければならない．(⇔静的束縛された変数の値を参照するときは，Access Link をたどり，active なスタックフレームのみを見に行く．)

いろいろな言語の例外

- C: setjmp(), longjmp() (あまり使われない)
- C++: try-catch, throw
- Java: try-catch-finally, throw
- ML: try-with (handle), raise

現代の (新しい) プログラム言語では，例外機構を持つことはほとんど必須.

継続 (continuation)

実行時の「残りの計算」を表す概念.

```
let rec fact n =
  if n=0 then 1
  else n * (fact (n-1))
in fact 10
let rec fact2 n k =
 if n=0 then k 1
 else
   fact2 (n-1) (fun x -> k (n*x))
in fact2 10 (fun x -> x)

fact2 10 (fun x -> x)
fact2 9 (fun x -> 10*x)
fact2 8 (fun x -> 10*9*x)
...
```


継続

「残りの計算」を，関数で表すことにより，末尾呼び出しの形に変形できた．(「継続渡し方式」(Continuation Passing Style) のプログラム)

- 継続は,「例外」などの制御構造を一般化したもの.
- 種々の制御構造を一般化して取り扱うのに便利.
- 高階関数があれば，継続をシミュレートできる.
- 直接，継続を扱う命令を持つ言語もある．call/cc (call-with-current-continuation; Scheme, SML/NJ, Ruby)
- call/cc などの実現の際には,「現在のスタックの保存」と「保存したスタックの活用」が必要になる.

まとめ

- 構造化プログラミングとプログラムの制御構造
- 例外と継続

興味がある人向けの宿題 (必須ではないが，良いレポートには加点する)

- Scheme の call/cc や Ruby の yield などの制御機構を調べ，それぞれの特徴を述べなさい.
- 例外機構があると，プログラムを書きやすくなったり，理解しやすくなったり，また，プログラムの効率が良くなったりすることがある．これらのメリットを表すため，例外機構を使ったプログラムと，使わないで同じ内容を実現したプログラムとを比較せよ．(具体的なプログラム例を，C++, Java, ML などで記述せよ．)

short quiz (2限用)

問題. GO TO 文 (無制限のジャンプ命令) は，なぜ，有害なのだろうか.

MiniML 再考

- MiniML 言語では、int や bool の定数のほかに「関数」をあらわすデータがある。このような関数の型は、どうなるであろうか？
- MiniML 言語のプログラム中には、int や bool といった型名を書かないが、そんなことで大丈夫だろうか？
- OCaml では (100+"abc") などの式は，コンパイル時にエラーとなる．これはどういう仕組みか?

実行時 vs コンパイル時

式 (1+"abc") が「いつ」エラーになるか。

- MiniC や MiniML では、実行時に (動的に) エラーになる。
- C や OCaml では、コンパイル時に (静的に) エラーになる。

エラーは静的に見つかる方がよい。

- 早い段階でエラーが見つかる。
- (実行に時間がかかる場合)、速く見つかる。

型システム Type System

式 (1+"abc") の整合性に関するエラーを、静的に発見したい。
一般に、(f(g(f(1),2)) + h(100,200)) のような式のエラーを、この式を実行せずに発見したい。

- 式の値を見るのではなく、式に「型」(type) を付け、その型のみを追うことによって整合性を検査する。
- 式の種類ごとに、どのような型が付くかを決めたものを「型システム」という。
- 現代のプログラム言語の多くが、強力な型システムを持っている。(例外は Prolog, Lisp/Scheme などごく一部)
- 型の整合性を検査するだけで、非常に多くのバグを発見することができる。

型検査

- C 言語では、変数の型は、その変数が使われるより前に宣言されている。
- さらに、他で定義された関数についても、その引数や返す値の型は事前に宣言する。
- ⇒型の整合性の検査は、変数や定数などのアトミックな式からはじめて、より大きな式の型が整合しているか検査する、という形式で行われる。

Java 言語なども同様。

型推論

- ML 言語では、(通常は) 変数や関数の型を宣言しない。
- さらに、他で定義された関数についても、その引数や返す値の型は宣言しない。
- ⇒型の整合性の検査は、「変数の型を推論する」という過程を含むため、それほど単純ではない。
- ML 言語では、「与えられたプログラム (らしき式) に対して、最も一般的な型を推論する」という型推論アルゴリズムが非常に重要。

```
(fun x→ x+1) :  int → int
(fun f→(fun x→ f(f(x+1)))) :  (int→ int)→(int→
int)
(fun x→ x) :  'a → 'a
(fun f → (fun x→ f (f x))) :  ('a → 'a) → ('a →
'a)
```

型推論の詳細は、「計算論理」の過去の講義資料等を参照のこと。

更に...

- 静的に (強く) 型付けされたプログラム言語: C, ML など。
- 動的に (弱く) 型付けされたプログラム言語: Lisp, Scheme, Ruby など。

Lisp, Scheme では、コンパイル時には型検査等は行われないので、(*lambda*(*x*)(+ 1 "*abc*")) という関数を書いても、(コンパイル時には) エラーにならない。この関数に引数を与えて実行したときに、はじめてエラーになる。ただし、型の概念がまったくないわけではない。(整数と文字列は実行時には区別される。)
Ruby でも、型エラーに相当するエラーも実行時に発生する。
考え方の相違: 信頼性 vs プログラムの書きやすさ・柔軟性

short quiz

次の MiniML プログラムの型を推論しなさい。

- `fun f -> (fun x -> f (f (f x)))`
- `fun y -> (fun x -> if x then y else y + 1)`

型システムの設計

現代のプログラム言語の設計において、型システムをどう設計するかは極めて重要。

型 (Type) とは?

「型」は「データの集合」の一種ではあるが、データの集合がすべて型になるとは限らない。

- コンピュータ (ハードウェア) で扱うことのできるデータの種類のこと。
- 同じ演算たちが適用できるデータの集まり。

型システム: どのようなプログラムにどのような型がつくか、定めるための体系。通常は、プログラムの構文の定義と一緒に与えられる。

具体的に‥

基本型 (atomic type)

- 例: `int`, `bool`, `string`
- 基本型の要素、基本型に対する演算などはプログラム言語ごとにあらかじめ与えられる。

複合的な型: 既にある型と型構成子 (type constructor) を使って構成。

- 例
  - 配列型: `int` の配列型、`string` の配列型
  - ポインタ型: 「`int` のポインタ型」
  - 構造体型、共用型、関数型 etc.
- 複合的な型の要素に対する演算は、型構成子ごとに与えられる。
  - 例. ポインタ型のデータを作る。`p=&x;`
  - 例. ポインタ型のデータを使う。`x=*p;`

再帰的に適用可能な型構成子が 1 つでもあると、型は無限個存在する。

型構成子の例

- C 言語: 構造体 (strucut)、共用 (union)、ポインタ、(関数) など。
- ML 言語: 直積、レコード (record)、バリアント (variant), 参照、関数、リスト、再帰的な型など。

再帰的な型の例 (OCaml での 2 分木):

```
type binary_tree =
  Leaf of int
| Node of binary_tree * binary_tree

Node(Leaf(1), Node(Leaf(2),Leaf(3)))
```

cf. BNF での構文定義: $e ::= 0 \mid 1 \mid (e + e)$.

型があることの利点(メリット)

Benjamin Pierce, "Types and Programming Languages", MIT Press から引用 (一部)。

- Documentation
- Error Detection
- Efficiency
- Abstraction, Language Safety
- etc.

Types for Documentation-1

```
goo (s) { return *s; }
foo (p, f()) { return *(f(&p)); }
main () {
  x = 10;
  printf ("%d\n", foo(&x,&goo));
}

fun f -> (fun x -> f (f (f (x+1)))) ;;
fun y -> (fun x -> if x then y else y + 1) ;;
```


Types for Documentation-2

```
int * goo (int **s) { return *s; }
int foo (int *p, int *f(int **)) {
  return *(f(&p)); }
int main () {
  int x = 10;
  printf ("%d\n", foo(&x,&goo));
}

fun f -> (fun x -> f (f (f (x+1))))
  : (int -> int) -> (int -> int)  ;;
fun y -> (fun x -> if x then y else y + 1)
  : int -> (bool -> int) ;;
```


Types for Error Detection-1

```
int goo (int x) { return x+1;}
int foo (int x, int *p, int f(int)) {
  return (x + f(*p));
}
int main () {
  int x = 10; int y = 20;
  printf ("%d\n", foo(&x,x,&goo));
  printf ("%d\n", foo(x,&x,&foo));
  printf ("%d\n", foo(&x,&goo));
}
```


Types for Error Detection-2

```
# fun y -> (fun x -> if x then x else y + 1) ;;
  Characters 37-42:
  fun y -> (fun x -> if x then x else y + 1) ;;
                                       ^^^^^
This expression has type int but is here used with type bool
```

ソフトウェア作りの鉄則

- エラーは早い段階 (上流) で見つければ見つけるほど、良い (修正のための手間が少ない)。
- 実行時 (動的) に見つけるよりも、コンパイル時 (静的) に見つける方がよい。
- たまたまその部分を実行したときに見つかるよりも、実行するかどうかにかかわらず見つけた方がよい。

Types for Efficiency-1

MiniC 言語の (OCaml で書いた) インタープリタ (簡略版):

```
| Plus (e1,e2) ->
   let v1 = (eval e1 env) in
   let v2 = (eval e2 env) in
     match (v1,v2) with
       | (IntValue(i1), IntValue(i2)) -> IntValue (i1+i2)
       | _ -> failwith "non-int value in arithmetic"
```

($e1 + e2$) という式の評価:

- $e1$ を計算して、結果を $v1$ とする。
- $e2$ を計算して、結果を $v2$ とする。
- $v1$, $v2$ がともに整数値であれば、加算をおこない、その結果を整数値として返す。

実行時に、データの型をチェックしている。
このようなチェックをしない場合、たとえば、「単なる整数値を、ポインタとして使う (メモリ領域上のとんでもない場所を指してしまうかもしれない)」という Segmentation Fault を起こす危険

Types for Efficiency-2

C 言語や ML 言語の処理系: MiniC, MiniML とは異なり、コンパイル時に型検査、型推論を行なう。
($e1 + e2$) という式の評価:

- $e1$ を計算して、結果を $v1$ とする。
- $e2$ を計算して、結果を $v2$ とする。
- $v1$, $v2$ はともに整数値であることはチェック済みなので、 そのまま加算をおこない、その結果を返す。

コンパイル時に、整数型であることをチェックしている。
補足: C 言語の + は、int 型だけでなく、float 型にも使われる (overloading) ので、コンパイル時に、int の加算か、float の加算かも判定する。また、必要ならば、int 型のデータを float 型のデータに変換するコードを挿入する。

Types for Abstraction-1

抽象化のための型とは?

型があることの利点 (再掲)

Benjamin Pierce, "Types and Programming Languages", MIT Press から引用 (一部)。

- Documentation (文書⇒理解しやすさ)
- Error Detection (エラーの早期発見)
- Efficiency (実行効率の良さ)
- Abstraction (抽象化), Language Safety(言語の安全性)
- etc.

静的な型システムの利点

Type Soundness (型に関する健全性):

- コンパイル時に型が整合していれば、(型検査あるいは型推論に合格すれば)、実行時には型に関するエラーを起こさない。

この性質が満たされるプログラム言語では、実行時に型のチェックをする必要がない。
この性質自体は、「証明」する必要がある。(現実のプログラム言語では、Standard ML などごく一部の言語に対してしか証明されていない。)

型検査

- 与えられたプログラム p に含まれる変数や関数の型が全て宣言されているとき (型が既知のとき)、p の型が整合してているかどうかを検査すること。
- 型検査の答えは YES or NO である。(判定問題)
- C 言語や Java 言語は、基本的には、すべての変数、関数 (あるいはクラス) の型が宣言されているため、コンパイル時に型検査を行う。

型推論

- 与えられたプログラム p に含まれる変数や関数の型がわからないものもあるとき (型が未知のとき)、p に型がつくかどうかを検査し、また、型がつく場合はその型が何であるかを推論すること。
- 型推論の答えは (YES と p の型) or NO である。(正確には、YES の場合は、p に含まれる変数等の型も推論される。)
- ML 言語は、変数の型で宣言されていないものがあるため、コンパイル時に型推論を行う。

型検査と型推論:

- 一般に、型検査問題より、型推論問題の方が困難。
- ML 言語では、どのような「プログラムらしきもの」を与えられても、型推論できるかどうかが有限時間で判定できる。
- 型システムを複雑にしていくと、型推論や型検査が有限時間内に判定できなくなることがある。

動的な型システム

静的な型システムを持たない言語でも、動的な型システムを持つことが多い。

- 型の整合性という概念はある。(整数と文字列を加えるとエラーなど)
- ただし、コンパイル時に全ての整合性をチェックするのではなく、実行時に (も) 型検査を行なう。
- 単純な効率やプログラムの理解のしやすさの観点からは、静的型システムに比べて不利。
- 静的な型システムで記述できないような、柔軟なプログラミングができる可能性がある。(実行結果によって、式の型が変化するプログラム。) cf. Ruby

多相型 (Polymorphism)

```
void swap (int *p, int *q) {
  int *r;
  r = p; p = q; q = r;
}
```

swap 関数は (int *) 型だけでなく、どんな型でも使える。

```
void swap (T *p, T *q) {
  T *r;
  r = p; p = q; q = r;
}
for any T.

# let swap (x,y) = (y,x) ;;
  val swap : 'a * 'b -> 'b * 'a = <fun>
```

多相型=「任意の型」を含む型。

型システムの設計

- プログラム言語の設計において、型システムの設計は極めて重要。
- プログラム言語の型システムの設計は、自由度が大きい。(個性があらわれる。)

型推論について

```
let rec fact x =
   if x = 0 then 1
   else x * (fact (x - 1))

val fact : int -> int = <fun>
```

OCaml システムはどうやって 関数 fact が `int -> int` であることを知ったのか？

型推論の例

```
let rec fact x =
   if x = 0 then 1
   else x * (fact (x - 1))
```

- fact の型は未知の型 α とする。
- x の型は未知の型 β とする。
- 式 x=0 から β=int とわかる。また、式 x=0 の型は bool である。
- 式 (x-1) の型は int である。(x は int 型であることは既知であり、それとこの式は整合している。)
- 式 fact(x-1) において、$x-1$ の型は int であるので、α = `int->`γ の形であることがわかる。ここで γ は新しい未知の型である。

型推論の例

```
let rec fact x =
   if x = 0 then 1
   else x * (fact (x - 1))
```

- `x:int` と `fact:int->`γ であり、`fact(x-1)` の型は γ である。
- `x*fact(x-1)` の型は `int` であり、また、γ = `int` でなければいけない。
- `if..then..else` 全体の型は `int` である。
- 左辺の `fact x` において、`x` は `int` 型であり、`fact` は `int->int` 型であるので、左辺 `int` 型となり、右辺と一致する。
- 以上から、この式は型が整合し、`fact:int->int` であることがわかった。

型推論の別の例

```
let double f x = f (f (x + 1))

val double : (int -> int) -> int -> int = <fun>
```

- `f` : γ とする。
- `x` : δ とする。
- `x+1` から、δ =`int` とわかる。
- `f(x+1)` から、γ =`int->`ϵ の形。
- `f(f(x+1))` から、ϵ =`int`。
- よって、`double=(int->int)->(int->int)`

型推論

(型) $\alpha ::=$ `int` | `bool` | α`->`α | $\cdots$

括弧の省略: `int->bool->string` は、`int->(bool->string)` のこと (右結合)。
型推論の途中の段階では、「未知の型」を表す型変数を使う。
型推論アルゴリズム

- 与えられた式の、各部分式の型が整合するためには、どの型とどの型が一致しなければいけないかを導く。(型に関する制約たちが得られる。)
- 「未知の型」を、どのような具体的な型にすると、これらの制約を全て満足できるか、計算する。(制約の解消)
- 全ての制約を満足できるとき、そのうち最も一般的な型を返す。
- 全ての制約を満足できないとき、「型エラー」である。

プログラム言語における型推論

- 強く型付けされたプログラム言語では、コンパイル時に (静的に) 型検査または型推論を行う。(例: C, Java, ML, Haskell)
- 強く型付けされていないプログラム言語では、(型システムがあれば) 実行時に (動的に) 型検査を行う。(例: Lisp, Scheme, Ruby, Perl. PHP)

強く型付けされたプログラム言語では、

- 束縛変数の型を明示的に記述する言語では、型検査を行う。(例: C, Java)
- 束縛変数の型を明示的に記述しなくてもよい (することもある) 言語では、型推論を行う。(例: ML, Haskell)

型システムの設計 (データ構造、データ抽象化の提供) は、現代のプログラム言語の設計の中心的な課題。

静的な型システム

保証すべき性質 (型システムの健全性):

- コンパイル時に (静的に) 型検査または型推論を行ったら、実行時には、型の不整合によるエラー (実行時の型エラー) は起こさない。

実行時の型エラーが起きるプログラム言語では。。。

- 実行時に、`3+"abc"` のような計算があり得る。
- 何もチェックしないと、Segmentation Fault に類似したエラー (または、意味をなさない計算結果が得られる)。
- 足し算をする毎に、「足されるものが両方とも `int` 型かどうか」をチェックするとしたら (実行時の型検査)、効率が悪くなる。

Quiz

次のプログラム の型を推論しなさい。

```
let f1 x y = x
let f2 f x = f (f (f x))
let f3 f g x = (f x) (g x)
```


Quiz の答

```
let f1 x y = x
```

- f1 : α -> (β -> α)
- x : α
- y : β

型変数の名前は α, β でなくてもよい。(ただし、2 つの型変数は異なるものでなければいけない。)
OCaml 処理系では、α, β 等は'a, 'b 等と記述される。

Quiz の答

```
let f2 f x = f (f (f x))
```

- f2 : (α -> α) -> (α -> α)
- f : α -> α
- x : α

Quiz の答

```
let f3 f g x = (f x) (g x)
```

- f3 : (α -> (β -> γ)) -> ((α -> β) -> (α -> γ))
- f : α -> (β -> γ)
- g : α -> β
- x : α