情報システム科学実習II第4回
高階関数，多相性，多相的関数

担当: 山口 和紀・五十嵐 淳

2001年11月7日

keywords: **高階関数，多相性，多相的関数，型推論**

1 高階関数

OCaml を始めとする関数型言語では，関数を整数・文字列などのデータと同じように扱うことができる．すなわち，ある関数を，他の関数への引数として渡したり，組などのデータ構造に格納したりすることができる．このことを「OCaml では関数は**第一級の値**(*first-class value*)である」といい，また関数について操作を行うような関数を**高階関数**(*higher-order function*)と呼ぶ.

1.1 関数を引数とする関数

まず，前回の復習をかねて，$1^2+2^2+\cdots+n^2$ を計算する関数 `sqsum` と，$1^3+2^3+\cdots+n^3$ を計算する関数 `cbsum` を定義してみよう.

```
# let rec sqsum n =
#   if n = 0 then 0 else n * n + sqsum (n - 1)
# let rec cbsum n =
#   if n = 0 then 0 else n * n * n + cbsum (n - 1);;
val sqsum : int -> int = <fun>
val cbsum : int -> int = <fun>
```

よく観察するまでもなく，この二つの関数の定義は酷似していることがわかるだろう．プログラミングの重要な作業の一つは，類似の計算手順を関数によって共有することである．この二つの関数の共通部分を吸収するような関数を定義できないだろうか.

このふたつの関数の似ている点は，`n` が `0` ならば `0` を返すことと再帰呼出しの結果と，`n` に関する計算結果の和がとられていることで，違いは `n` に関する計算手順だけである．もともと計算手順を抽象化したものが関数であるので,「関数をパラメータとする関数」を定義すればよさそうである．それを，素直に表現したのが以下の定義である.

```
# let rec sigma (f, n) =
#   if n = 0 then 0 else f n + sigma (f, n-1);;
val sigma : (int -> int) * int -> int = <fun>
```

この関数 sigma は型が示しているように，整数上の関数 f と整数 n の組を受け取って，整数を返す．(型の読み方: 型構築子 * と -> は * の方が優先度が高い．)

これを使って，sqsum と cbsum は，

```
# let square x = x * x
# let sqsum n = sigma (square, n)
# let cbsum n =
#   let cube x = x * x * x in sigma (cube, n);;
val square : int -> int = <fun>
val sqsum : int -> int = <fun>
val cbsum : int -> int = <fun>
```

と定義することができる．高階関数に渡すだけの補助的な関数は，他で必要ない場合は，cbsum の例のように let-式で局所束縛を行うか，次節で説明する匿名関数を使うのがよいだろう．

```
# sqsum 5;;
- : int = 55
# cbsum 5;;
- : int = 225
```

1.2 匿名関数

さて，関数 sigma は，$\sum_{i=0}^{n} f(i)$ のような計算を異なる f について行いたい場合に便利である．しかし，これまでにみた関数は let を用いて定義するしかなく，実際の f ひとつひとつについて，新しい名前をつけて定義をしなければならない，というやや面倒な手順を踏まなければならない．

関数型言語では，名前のない関数，**匿名関数**(*anonymous function*) を扱う手段がたいてい用意されていて，OCaml もその例外ではない．OCaml では匿名関数は

fun ⟨パターン⟩ -> e

という形をとり，⟨パターン⟩で表される引数を受け取り式 e を計算する．この fun 式は関数が必要な場所どこにでも使用することができる．

```
# let cbsum n = sigma ((fun x -> x * x * x), n);;
val cbsum : int -> int = <fun>
# let sq5 = ((fun x -> x * x), 5) in
#   sigma sq5;;
- : int = 55
# (fun x -> x * x) 7;;
- : int = 49
```

二三番目の例のように，組の要素にもなり，また (あまり実用的な意味はないが) 直接適用することもできる．また，いずれの例でも fun のまわりの () が必要である (これがないと，“，n” が関数本体の一部と思われてしまい，x * x * x と n の組を返す関数として解釈されてしまう．一般的には，fun はできる限り先まで関数本体と思い込もうとするので，適宜 () を使ってどこまで関数本体か示してやる必要がある．)

実は，let による関数宣言

```
let f x = e
```

は

```
let f = fun x -> e
```

の略記法である．このことから，関数を構成すること (fun) と，それに名前をつけることは，必ずしも関連していない別の仕組みであることがいえる．

1.3 カリー化と関数を返す関数

OCaml の関数は全て 1 引数であるため，引数が二つ以上必要な関数を定義するには組を用いることをみてきた．ここでは,「関数を返す関数」を使って引数が複数ある関数を模倣できる方法をみる．このような「関数を返す関数」をその発見者 Haskell Curry の名をとって**カリー化関数**(*curried function*) と呼ぶ.

基本的なアイデアは「x と y を受け取り e を計算する関数」を「x を受け取ると,『y を受け取って e を計算する関数』を返す関数」として表現することである．具体的な例として，二つの文字列 s_1, s_2 から s_1s_2 のような連結をした文字列を返す関数を考えてみよう．これまでにみてきた，組を使った定義では,

```
# let concat (s1, s2) = s1 ^ s2 ;;
val concat : string * string -> string = <fun>
```

と定義され，型はまさに「文字列を二つ (組にして) 受け取り文字列を返す」ことを表している．使う場合も二つの文字列を同時に指定して concat ("abc", "def") のように呼び出す.

さて，この関数を カリー化関数として定義してみよう.

```
# let concat_curry s1 = fun s2 -> s1 ^ s2;;
val concat_curry : string -> string -> string = <fun>
```

concat_curry は fun 式を用いて,「s2 を受け取って (既に受け取り済の) s1 と連結するような関数」を返している．concat_curry の型は string -> (string -> string) と同じで，そのことを示している．この関数を呼び出すには，2 回の関数適用を経て,

```
# (concat_curry "abc") "def";;
- : string = "abcdef"
```

のように行う．(...) 内の関数適用で,「”abc” と与えられた引数を連結するような関数」が返ってきており，外側の関数適用 (...) "def" で文字列の連結が行われる．

カリー化関数は，組を用いた定義と違って，**部分適用**(*partial application*) と呼ばれる，一つ目の引数を固定したような関数を使うことが容易である．例えば，敬称 Mr. を名前 (文字列) に付加する関数を

```
# let add_title = concat_curry "Mr. ";;
val add_title : string -> string = <fun>
# add_title "Igarashi";;
- : string = "Mr. Igarashi"
```

と定義することができる．add_title は引数の置き換えモデルにしたがって，fun s2 -> "Mr. " ^ s2 という関数に束縛されていると考えることができる．

カリー化関数の型は，読み方によって,「二引数の関数の型」と読むことも,「関数を返す関数」と読むこともできる．

関数定義の文法拡張　上のカリー化関数の定義方法を，fun を入れ子にすることによって，三引数，四引数の関数の表現に拡張していくことも可能である．

実は，OCaml では，fun を入れ子にする代りに，let や fun でのパラメータパターンを複数個並べることによって，カリー化関数をより簡潔に定義することができる．先の例は,

```
# let concat_curry s1 s2 = s1 ^ s2;;
val concat_curry : string -> string -> string = <fun>
```

と定義することも,

```
# let concat_curry = fun s1 s2 -> s1 ^ s2;;
val concat_curry : string -> string -> string = <fun>
```

と定義することも可能である．一般的には,

fun ⟨パターン$_1$⟩ -> fun ⟨パターン$_2$⟩ -> ... fun ⟨パターン$_n$⟩ -> e

は

fun ⟨パターン$_1$⟩⟨パターン$_2$⟩ ...⟨パターン$_n$⟩ -> e

と同じである．(let についても同様にパターンを空白で区切って並べることができる．)

また，関数適用も (((f x) y) z) と書く代りに f x y z と，括弧を省略することができる．(別の言い方をすると，関数適用式は**左結合する**．) 関数型構築子は，既に見たように，右結合し，t1 -> t2 -> t3 -> t4 は t1 -> (t2 -> (t3 -> t4)) を意味する．

中置/前置演算子　これまで，なんの説明もなしに使ってきたが，+, ^ などの**中置演算子**(*infix operator*) は，内部ではカリー化された関数 (int->int->int などの型をもつ) として定義されている．さらに OCaml では新たな中置演算子を定義したり，(勧められないが) + などを再宣言することも可能ですらある．

中置演算子は (と) で囲むことによって，通常の関数として (前置記法) で使うことができる．

```
# (+);;
- : int -> int -> int = <fun>
# ( * ) 2 3;;
- : int = 6
```

* の前後に空白が入っているのは，コメントの開始/終了と区別するためである．

中置演算子として使用可能な記号は，mod, *, =, or, & などのキーワードもしくは

- 1 文字目が =, <, >, @, ^, |, &, +, -, *, /, $, % のいずれかで，
- 2 文字目以降が !, $, %, *, +, -, ., /, :, <, =, >, ?, @, ^, |~ のいずれか

を満たす文字列である．

定義をするには (⟨ 中置演算子 ⟩) を普通の名前だと思って行う．

```
# let (^-^) x y = x * 2 + y * 3;;
val ( ^-^ ) : int -> int -> int = <fun>
# 9 ^-^ 6;;
- : int = 36
```

また，前置演算子といって，定義するときや単独で関数値として使うときは () が必要な，記号列からなる名前が用意されている．

```
# let ( !! ) x = x + 1;;
val ( !! ) : int -> int = <fun>
# !!;;
Characters 2-4:
Syntax error
# (!!);;
- : int -> int = <fun>
# !! 3;;
- : int = 4
```

前置演算子は -, -. もしくは

- 1 文字目が !, ?, ~ のいずれかで，
- 2 文字目以降が !, $, %, *, +, -, ., /, :, <, =, >, ?, @, ^, |~ のいずれか

からなる文字列である．

演算子同士の優先度は，名前から決まっている．表 1は優先度の高いものから並べたものである．コンストラクタなど，未出の概念，記号がでてきているがとりあえず無視しておいてよい．

表 1: 演算子の優先順位と結合

演算子	結合
前置演算子	–
関数適用	左
コンストラクタ適用	–
前置演算子としての `-`, `-.`	–
`**` で始まる名前	右
`*`, `/`, `%`, で始まる名前および `mod`	左
`+`, `-` で始まる名前	左
`::`	右
`@`, `^` で始まる名前	右
`=`, `<` など比較演算子, その他の中置演算子	左
`not`	–
`&`, `&&`	左
`or`, `||`	左
`,`	–
`<-`, `:=`	右
`if`	–
`;`	右
`let`, `match`, `fun`, `function`, `try`	–

1.4 Case Study: Newton-Raphson 法

高階関数の有効な例として，方程式の近似解を求める Newton-Raphson 法をプログラムしてみよう．Newton-Raphson 法は，微分可能な関数 f に対して，方程式 $f(x) = 0$ の解を求める方法であり，

$$g(x) = x - \frac{f(x)}{f'(x)}$$

の不動点 ($g(a) = a$ なる a) を求める．(もしくは漸化式 $x_n = x_{n-1} - f(x_{n-1})/f'(x_{n-1})$ の極限を求める．)

これを解くプログラムを考えてみよう．まず，微分をどう表現するかであるが，近似的に，とても小さい定数 dx に対して

$$g'(x) = \frac{g(x + dx) - g(x)}{dx}$$

としよう．微分を求める関数は，自然に次のような高階関数として定義できる．

```
# let deriv f =
#   let dx = 0.1e-10 in
#     fun x -> (f(x +. dx) -. f(x)) /. dx;;
val deriv : (float -> float) -> float -> float = <fun>
```

例えば $f(x) = x^3$ の 3 における微分係数は，

```
# deriv (fun x -> x *. x *. x) 3.0;;
- : float = 26.9999134161
```

と計算される．

次に不動点を求める関数を定義してみよう．この関数は，関数 f と初期値 x から，$f(x)$, $f(f(x)), \ldots$ を計算していき，$f^n(x) = f^{n-1}(x)$ となったときの $f^n(x)$ を返す．実数の計算には誤差が伴うので実際には，ある時点で $|f^{n-1}(x) - f^n(x)|$ がある閾値以下になったときに終了とする．

```
# let fixpoint f init =
#   (* computes a fixed-point of f, i.e., r such that f(r)=r *)
#   let threshold = 0.1e-10 in
#   let rec loop x =
#     let next = f x in
#     if abs_float (x -. next) < threshold then x
#     else loop next
#   in loop init;;
val fixpoint : (float -> float) -> float -> float = <fun>
```

さて，これを使って，Newton-Raphson 法で用いる不動点を求めるべき関数は元の関数から

```
# let newton_transform f = fun x -> x -. f(x) /. (deriv f x);;
val newton_transform : (float -> float) -> float -> float = <fun>
```

で計算できる.

最終的に，Newton-Raphson 法で $f(x) = 0$ の解を求める関数は，関数 f と近似解 $guess$ を受け取って

```
# let newton_method f guess = fixpoint (newton_transform f) guess;;
val newton_method : (float -> float) -> float -> float = <fun>
```

と定義できる.

```
# let square_root x = newton_method (fun y -> y *. y -. x) 1.0;;
val square_root : float -> float = <fun>
# square_root 5.0;;
- : float = 2.2360679775
```

1.5 練習問題

Exercise 4.1 実数上の関数 f に対して $\int_a^b f(x)dx$ を計算する関数 `integral f a b` を定義せよ．またこれを使って，$\int_0^\pi \sin x dx$ を計算せよ.

近似的な計算方法として，ここでは台形近似を説明するが他の方法でも良い．台形公式では $b-a$ を n 分割した区間の長さを δ として，台形の集まりとして 計算する．i 番目の区間の台形の面積は

$$\frac{(f(a+(i-1)\delta)+f(a+i\delta))\cdot\delta}{2}$$

として求められる.

Exercise 4.2 練習問題 3.8 の `pow` 関数をカリー化して定義せよ． 次に第一引数が指数になるよう (`pow n x`) に定義し，3 乗する関数 `cube` を部分適用で定義せよ．指数が第二引数であるように定義されている場合 (`pow x n`)，`cube` を `pow` から定義するには どうすればよいか?

Exercise 4.3 `fun` で再帰関数を定義することはできない，なぜか?

Exercise 4.4 以下の 3 つの型

- `int -> int -> int -> int`
- `(int -> int) -> int -> int`
- `(int -> int -> int) -> int`

の違いを説明せよ．また，各型に属する適当な関数を定義せよ.

2　多相性

いろいろな関数を書いていくと，引数の型に関わらず同じことをする関数が出現する場合がしばしば現れる．例えば，二つ組から 第一要素を取出す関数を考えてみよう．例えば，`int * int` に対するこのような関数は，

```
# let fstint ((x, y) : int * int) = x;;
val fstint : int * int -> int = <fun>
```

と書ける．明示的に型を宣言しているのは，意図的である．また，`(int * float) * string` のような組と文字列の組に対して同様な関数を定義すると

```
# let fst_ifs ((x, y) : (int * float) * string) = x;;
val fst_ifs : (int * float) * string -> int * float = <fun>
```

と書ける．さて，ここまでくると生じる疑問は，組の要素の組み合わせごとにいちいち別の第一要素を取り出す関数をかかなければいけないのだろうか?ということである．これらの関数は引数の型を除いて同じ格好をしている．それならば，共通部分はひとつの定義におさめ，差異をパラメータ化できないだろうか? パラメータ化するとしたら，パラメータは一体なんであろうか?

注意深く考えると，この共通の定義は引数の「型」，より正確には第一要素と第二要素の型 (`int` と `int`，または `int * float` と `string`) に関してパラメータ化することになることがわかるだろう．また，このパラメータとしては今までの関数とは異なり,「型を表現する変数」のようなものが必要なことがわかる．これを明示的に書き表したのが下の関数宣言である．

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

`'a` が `'b` が**型変数**(*type variable*) と呼ばれるものである．このような「型に関して抽象化された関数」を**多相的関数**(*polymorphic function*) と呼ぶ．この `fst` の型 `'a * 'b -> 'a` は「任意の型 `T1`, `T2` に対して，`T1 * T2 -> T1`」と読むことができる．(論理記号を使うならば ∀'a.∀'b.(`'a * 'b -> 'a`) と考えるのがより正確である．) 実は，OCaml の型推論機能はこのような多相的関数の宣言に際しても，明示的に型変数を導入する必要はない．

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

それどころか，型宣言を省略することによって，常にその定義のもっとも一般的な型 (**主要な型**(*principal type*) と呼ぶ) を求めることができる．`fst` の主要な型は `'a * 'b -> 'a` である．通常の「式に関して抽象化された関数」を適用する際に，実引数，つまりパラメータの実際の値を渡すのと同様，多相的関数には「型の実引数」を渡すと考えられるが，実際のプログラムでは，型引数を明示的に書き下す必要はない．(書き下すための文法は用意されていない．)

```
# fst (2, 3);;
- : int = 2
# fst ((4, 5.2), "foo");;
- : int * float = 4, 5.2
```

概念的には，最初の例では 'a, 'b に int が，次の例では 'a に int * float，'b に string が渡っていると考えることができる．別の見方をすると，fst という式が，二つの違った型の式 int * int -> int と (int * float) * string -> int * float として使われていると見ることができる．このように一つの式が複数の型を持つことを言語に**多相性**(*polymorphism*) がある，という．また，特に，関数の型情報の一部をパラメータ化することによって発生する多相性を**パラメータ多相**(*parametric polymorphism*) と呼ぶ．パラメータ多相の関数は型変数に何が渡されようともその振舞いは同じであるという特徴がある．多相性は，ひとつの定義の (再) 利用性を高めるのに非常に役立つ．

いくつか多相性のある関数の例をみてみよう．以下の id は**恒等関数**(*identity function*) とよばれ，与えられた引数をそのまま返す関数である．また，関数適用関数 apply は関数とその引数を引数と受け取って，関数適用を行う．

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
```

apply の型では，('a -> 'b) が f の型を，'a が x の型を示す．型変数 'a が，ふたつの引数 f と x の間の制約，つまり f は x に適用できなければならないことを表現していることに注意したい．

```
# apply abs (-5);;
- : int = 5
# apply abs "baz";;
Characters 6-9:
This expression has type int -> int but is here used with type string -> 'a
```

関数の多相性はどこに由来するものであろうか? fst の定義をよく見ると，本質的に，(x, y) というパターンの要請する引数に関する制約は,「二つ組であること」だけで，各要素が整数であろうが，文字列であろうが，構わないはずである．また， 各要素 x, y に対して何の操作も行われていないため，それが唯一の制約である．また，apply の定義からは f に対する要請は「x に適用できる関数であること」だけである．このように，パラメータ多相は，関数が引数の部分的な構造 (組である，関数であること) のみで計算が行われることに起因している．また型変数は，関数自身は操作しない部分の構造を抽象化していると考えることができる．つまり 'a * 'b -> 'a という型を見れば，その関数が引数の第一要素も第二要素も使わないことがわかるのである．

> **様々な多相性** その他の多相性としては，多くの言語に見られる + という一つの記号が整数同士もしくは実数同士の足し算どちらでも使える，といった**アドホックな多相性**(*ad-hoc polymorphism*) と呼ばれるもの，オブジェクト指向言語で見られる親子クラス関係な

ど，型上に定義された二項関係によって，式が複数の型を持ちうる**部分型多相**(*subtyping polymorphism*) といったものがある．部分型多相については，オブジェクト指向を扱う部分で詳しく見ていくことになる．

2.1 let 多相と値多相

OCaml では多相性を持てるのは，let(宣言もしくは文) で導入された変数のみである．関数のパラメータを多相的に使うことはできない．具体的には，下の例で x を (id が来るとわかっていても) 異なる型の値への適用は許されない．

```
# (fun x -> (x 1, x 2.0)) id;;
Characters 19-22:
This expression has type float but is here used with type int
```

このエラーメッセージは，x 1 を型推論した時点で x の引数の型は int として決定されてしまったのに，float に対して適用している，という意味である．

また，任意の let 式でよいわけではなく，定義されるものが「値」でなければいけない，という制限がある．値として扱われるものは，関数の宣言，定数，など計算を必要としない式である．逆に許されないものは，関数適用などの値に評価されるまでに計算を伴う式である．以下の例は f を x に二度適用する double 関数である．

```
# let double f x = f (f x);;
val double : ('a -> 'a) -> 'a -> 'a = <fun>
# double (fun x -> x + 1) 3;;
- : int = 5
# double (fun s -> "<" ^ s ^ ">") "abc";;
- : string = "<<abc>>"
```

さらに，この関数を組み合わせると，同じ関数を 4 度適用する関数として用いることができる．

```
# double double (fun x -> x + 1) 3;;
- : int = 7
# double double (fun s -> "<" ^ s ^ ">") "abc";;
- : string = "<<<<abc>>>>"
```

ところがこれを let でそのまま宣言しても，右辺が関数適用という値ではない式なので，多相的につかうことはできない．

```
# let fourtimes = double double;;
val fourtimes : ('_a -> '_a) -> '_a -> '_a = <fun>
```

アンダースコアのついた型変数 '_a は,「一度だけ」任意の型に置換できる型変数であり，一度決まってしまうと二度と別の型として使うことはできない．

```
# fourtimes (fun x -> x + 1) 3;;
- : int = 7
# fourtimes;;
- : (int -> int) -> int -> int = <fun>
# fourtimes (fun s -> "<" ^ s ^ ">") "abc";;
Characters 35-40:
This expression has type string but is here used with type int
```

fourtimes の型変数が一度目の適用で int に固定されてしまうことに注意.

この制限を**値多相**(*value polymorphism*) と呼ぶ．値多相に制限しなければならない理由は副作用と深く関係があり，ここでは説明しない．(次々回くらいに説明する．) fourtimes のような定義に多相性を持たせるためには,

```
# let fourtimes' f = double double f
#   (* equivalent to "let fourtimes' = fun f -> double double f" *) ;;
val fourtimes' : ('a -> 'a) -> 'a -> 'a = <fun>
```

のように，明示的にパラメータを導入して，関数式として (つまり fun を使って) 定義してやればよい.

```
# fourtimes' (fun x -> x + 1) 3;;
- : int = 7
# fourtimes' (fun s -> "<" ^ s ^ ">") "abc";;
- : string = "<<<<abc>>>>"
```

2.2 多相型と型推論

さて，これまで OCaml では型推論の機能があり，プログラマは関数の引数の型を特に明示的に宣言しなくてもよいことには触れたが，それがどのような仕組みで実現されているかについては詳しく述べなかった．ここでは，簡単に型推論の仕組みを見る.

まず簡単な例から考えてみよう．fun x -> x + 1 という式はもちろん int -> int 型を持つわけだがこれがなぜか考えてみると，ひとつの考え方として

- + は int -> int -> int 型だから両辺に int が来なければならない.
- x の型は与えられていないが，+ の左側にあるから，x は int でなければならないはずだ．1 も同様に int でなければならないが，これは OK だ.
- x が int であると仮定すれば，x + 1 は int になる.
- x + 1 を int でなければならない x で抽象化するから，全体の型は int -> int である.

という，推論が成り立つ．OCaml はまさにこのように型推論を行っている．より具体的には，式をボトムアップに見ていく．定数 1 などはその型が自明に与えられ，変数に関しては

とりあえず，未知の型を表す型変数を割り当てる．関数適用や if 式など，部分式から構成される個所で，部分式の型に関する制約 (「x の型 'a は int でなければならない」など) を構成し，それを解く．もしも制約が解けない場合は，その式は型があっていないと結論づけられる．制約が解けない例は

```
fun x -> if x then 1 else x
```

を考えるとわかる．if 式の型規則は if 直後の式は bool に等しく，then 節，else 節の型も等しい，というものであるから，x に割り当てられた型変数を 'a とすると，int = 'a = bool という制約が得られる．これは明らかに解くことができないので，この式は型エラーとなる．

制約を解いても型変数が残る場合がある．この式が値で let で宣言されるものであれば，この型変数は型パラメータとして (暗黙の内に) 抽象化される．例えば，先ほどの apply の宣言は，f, x の未知の型をそれぞれ 'a, 'b とすると f x という式から，f x の型を 'c として，'a = 'b -> 'c という制約と fun f x -> f x 全体の型 'a -> 'b -> 'c が導かれる．これを書き換えて，('b -> 'c) -> 'b -> 'c となる．

型宣言省略のススメ ところで，型推論は型宣言を省ける一方，型宣言そのものはプログラムを読む上でも有用なものである．どのような場合に宣言をすべき/しないべきかは，やや趣味の問題である部分もあるが，高階関数/匿名関数を使うようになると，型が文脈から自明な場合が多く現れる．たとえば，先ほどの Newton-Raphson 法で，

```
let square_root x = newton_method (fun y -> y *. y -. x) 1.0;;
```

のように高階関数 newton_method に，匿名関数を渡している．newton_method の型から匿名関数のパラメータ y の型が float であることはすぐわかる．ここで，わざわざ fun (y : float) -> ... と書かなければならないとしたら結構面倒である．

また，主要な型を求める一番確実な方法は，型宣言をしないことである．先ほどの fst を

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

と宣言しても，型チェックを通過するが，組の要素が同じ型を持たなくてはならないという，一般性を欠く定義になってしまう．

```
# fst (true, 3.2);;
Characters 6-15:
This expression has type bool * float but is here used with type bool * bool
```

もちろん，後でみるようにモジュールを使って分割コンパイルをしようとすると，型宣言を明示的にしなければならなくなるのだが，ひとまず型推論の力，型宣言をしなくてよいうれしさをできるだけ味わってみよう．

2.3 Case Study: コンビネータ

「計算」という概念のモデルとして基本的なものにチューリング機械がある．チューリング機械は，CPU がメモリの読み書きをしながら計算を進めて行くことを単純化したものであ

る．これに対して，関数型プログラミングの計算をモデル化したものに λ-計算，コンビネータ理論といったものがある．λ-計算の体系は関数適用におけるパラメータの引数の置換をモデル化したもので，複雑な計算も関数抽象式 (OCaml の `fun` 式) と関数適用の組み合わせのみで表すことができる．コンビネータ理論は，λ-計算の理論と密接に関わっていて，コンビネータと呼ばれるいくつかの高階関数の組合わせで，複雑な計算を表現するものである．ここでは簡単にコンビネータについてみていきたい．

まずは，関数合成を行うコンビネータ `o` は OCaml で，

```
# let o f g x = f (g x);;
val o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

と表現される．例えば，二つの関数の組合わせからなる関数

```
fun x -> -. (sqrt x)
```

は，

```
# o ( -. ) sqrt;;
- : float -> float -> float = <fun>
```

という式で表現できる．コンビネータのポイントは，明示的にパラメータを導入することなく，単純な関数を組み合わせてより複雑な関数を構成できるところにある．

もっとも単純なコンビネータは先に見た `id` である．(コンビネータ理論では I コンビネータと呼ばれる．) `id f` は `f` と同じ関数を表現する．また I コンビネータは関数合成と組み合わせても何も起らない．

```
# o id (sqrt) 3.0;;
- : float = 1.73205080757
```

K コンビネータは定数関数を構成するためのコンビネータであり，以下の関数で表現される．

```
# let k x y = x;;
val k : 'a -> 'b -> 'a = <fun>
```

`k x` は何に適用しても `x` を返す関数になる．

```
# let const17 = k 17 in const17 4.0;;
- : int = 17
```

次の S コンビネータは関数合成を一般化したものである．

```
# let s x y z = x z (y z);;
val s : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

OCaml で `fun` 式と関数適用の組合わせ「のみ」で表現できる関数 (`id`, `double` などで，λ-計算の表現できる関数のクラスと同じである) は S と K の組合わせのみで表現できることが知られている．例えば I コンビネータは $S\ K\ K$ として表せる．

```
# s k k 5;;
- : int = 5
```

コンビネータ，λ-計算はチューリング機械と同程度の表現力がある計算モデルであることが知られている．

2.4 練習問題

Exercise 4.5 以下の関数 curry は，与えられた関数をカリー化する関数である．curry の型の意味と，使い方の例，を説明し，この逆，つまり (2 引数の) カリー化関数を受け取り，二つ組を受け取る関数に変換する uncurry 関数を定義せよ．

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

Exercise 4.6 以下の関数 repeat は double, fourtimes などを一般化したもので，f を n 回，x に適用する関数である．

```
# let rec repeat f n x =
#   if n > 0 then repeat f (n - 1) (f x) else x;;
val repeat : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

これを使って，フィボナッチ数を計算する関数 fib を定義する．... を埋めよ．

```
let fib n =
  let (fibn, _) = ...
  in fibn;;
```

Exercise 4.7 次の関数 funny がどのような働きをするか説明せよ．

```
# (* f <@> g denotes the composition of f and g *)
# let ( <@> ) f g x = (f (g x)) ;;
val ( <@> ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let rec funny f n =
#   if n = 0 then id
#   else if n mod 2 = 0 then funny (f <@> f) (n / 2)
#   else funny (f <@> f) (n / 2) <@> f;;
val funny : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Exercise 4.8 S K K が恒等関数として働く理由を S K K 1 が評価される計算ステップを示すことで，説明せよ．

Exercise 4.9 double double f x が f (f (f x))) として働く理由を前問と同様にして説明せよ．