関数プログラミング ことはじめ

2015.11.13
岡山大学 プログラミング技法 講義資料

山本和彦

@kazu_yamamoto

IIJ INNOVATION INSTITUTE

関数プログラミングとは何ですか？

関数プログラミングとは
数学でいう「関数」を使って
プログラミングすることです。

「数学の関数」とは何ですか？

引数から値を計算して返す箱です。

$$f(x) = x + 1$$

数学の関数は、
引数が同じであれば
必ず同じ値を返します。

$$f(x) = x + 1$$
$$f(1) = 1 + 1 = 2$$
$$f(5) = 5 + 1 = 6$$

C言語にも関数がありますが、
これは「数学の関数」ですか？

いいえ。

「C言語の関数」は
「数学の関数」より
たくさんのことが
できてしまいます。

この講義では、数学の関数より
たくさんのことができる箱のことを
「手続き」と呼びます。

また、これ以降
単に「関数」と言うと
「数学の関数」のことです。

「C言語の関数」は「手続き」です。

手続きを使ったプログラミングのことを
「命令プログラミング」と呼びます。

ここまでのまとめ

関数プログラミング
=
数学の関数を使ったプログラミング

命令プログラミング
=
手続きを使ったプログラミング

C言語で普通にコードを書くと
命令プログラミングを
していることになります。

「手続き」は
機能を制限して利用すれば
「関数」としても使えます。

C 言語の手続きも
数学の関数として利用できるなら、
C言語で関数プログラミングを
することはできますか？

できなくはないですが
限られたことしかできません。

C 言語は関数プログラミングを
サポートするようには
作られてないからです。

数学の関数の具体例を
コードで示して下さい。

たとえばC言語で書いた

```
int f(int x) {
  return (x + 1);
}
```

は数学の関数です。

たとえばC言語の三角関数
`sin`
も数学の関数です。

しかしC言語の
`printf`
は数学の関数ではなく手続きです。

「文字を出力する」という
数学の関数では
できない仕事をするからです。

また、

```
int y = 0;

int f(int x) {
  y = y + 1;
  return (x + 1);
}
```

も数学の関数ではなく手続きです。

グローバル変数 y を書き換えています。
それは、数学の関数ではできないことです。

関数の「作用」(仕事)は値を返すことです。

値を返すこと以外の仕事は
「副作用」と呼ばれます。

この講義では
副作用を持つ箱を
手続きと呼んでいます。

副作用には
どんなものがありますか？

ディスプレイや
ネットワークへの入出力、
グローバル変数の書き換え
などが副作用の例です。

関数プログラミングでは
代入がないと聞いたことが
あるのですが本当ですか？

代入には
「初期化」
と
「再代入」
があります。

関数プログラミングでは
変数を「初期化」できますが
変数に「再代入」することはできません。

たとえば、初期化

```
int x = 1;
```

はできます。

しかし、再代入

```
x = x + 1;
```

はできません。

プログラミングを始めたとき

```
x = x + 1;
```

という文をみてギョッとしましたよね？

関数プログラミングとは、
「プログラムの変数」を「数学の変数」
として使うという意味もあります。

これまでのまとめ

関数プログラミングとは

プログラムの関数を数学の関数
として使ってプログラミングすること

プログラムの変数を数学の変数
として使ってプログラミングすること

再代入がないなら for 文さえ
書けないのではないですか？

はい、そうです。

繰り返しには、再帰を使います。

C言語では階乗のコードを
以下のように書けます。

```
int factorial(int n) {
  int r = 1;
  for (int i = 1; i <= n; i++) {
    r = r * i;
  }
  return r;
}
```

関数プログラミングでは
どのように書くのですか？

階乗の計算はこうですね。

factorial(*1*) = *1*
factorial(*2*) = *1* * *2*
factorial(*3*) = *1* * *2* * *3*
factorial(*4*) = *1* * *2* * *3* * *4*
factorial(*5*) = *1* * *2* * *3* * *4* * *5*

$factorial(1) = 1$
$factorial(2) = 1 * 2$
$factorial(3) = 1 * 2 * 3$
$factorial(4) = 1 * 2 * 3 * 4$
$factorial(5) = 1 * 2 * 3 * 4 * 5$

問)それぞれの式を一つ前の式を
使って表してください。

$factorial(1) = 1$
$factorial(2) = 1 * 2$
$factorial(3) = 1 * 2 * 3$
$factorial(4) = 1 * 2 * 3 * 4$
$factorial(5) = 1 * 2 * 3 * 4 * 5$

は、1つ前の式を使うと、
以下のようになります。

$factorial(1) = 1$
$factorial(2) = factorial(1) * 2$
$factorial(3) = factorial(2) * 3$
$factorial(4) = factorial(3) * 4$
$factorial(5) = factorial(4) * 5$

$factorial(1) = 1$
$factorial(2) = factorial(1) * 2$
$factorial(3) = factorial(2) * 3$
$factorial(4) = factorial(3) * 4$
$factorial(5) = factorial(4) * 5$

問) これを一般化してください。

$factorial(1) = 1$
$factorial(2) = factorial(1) * 2$
$factorial(3) = factorial(2) * 3$
$factorial(4) = factorial(3) * 4$
$factorial(5) = factorial(4) * 5$

を一般化すると以下のようになります。

$factorial(1) = 1$
$factorial(n) = factorial(n-1) * n$

$factorial(1) = 1$
$factorial(n) = factorial(n-1) * n$

をプログラムで書くとどうなりますか？

$factorial(1) = 1$
$factorial(n) = factorial(n-1) * n$

を疑似コードで書くと
以下のようになります。

```
int factorial(int n) =
  if (n == 1) then
    1
  else
    factorial(n - 1) * n;
```

この資料では、
関数プログラミングのために
山本が(適当に)作った「疑似コード」
を使います。

```
int factorial(int n) =
  if (n == 1) then
    1
  else
    factorial(n - 1) * n;
```

波括弧や return がないのは
どうしてですか？

命令プログラミングでは
複数の文を列挙します。

```
int main (void) {
  printf "Hello, ";
  printf "world!";
  printf "\n";
}
```

セミコロンは文の区切りで、
複数の文を一つの固まりにするために
波括弧が使われます。

関数プログラミングでは
文ではなく「式」を使って
プログラムを構成します。

```
int factorial(int n) =
  if (n == 1) then
    1
  else
    factorial(n - 1) * n;
```

関数定義の右側は「一つの式」
なので波括弧は必要ありません。

また関数は必ず値を返します。
return を書いていなくても
値を返すのだと理解してください。

```
int factorial(int n) =
  if (n == 1) then
    1
  else
    factorial(n - 1) * n;
```

これまでのまとめ

関数プログラミングとは
式でプログラムを書くことです。

命令プログラミングとは
文でプログラムを書くことです。

```
int factorial(int n) =
  if (n == 1) then
    1
  else
    factorial(n - 1) * n;
```

の n は何回も値が変わります。
再代入されているのではないですか？

いいえ。

あるスコープにおいて
n が初期化されると
そのスコープ内では n は不変です。

あなたの思っている n は、
別々の n と n なのです。

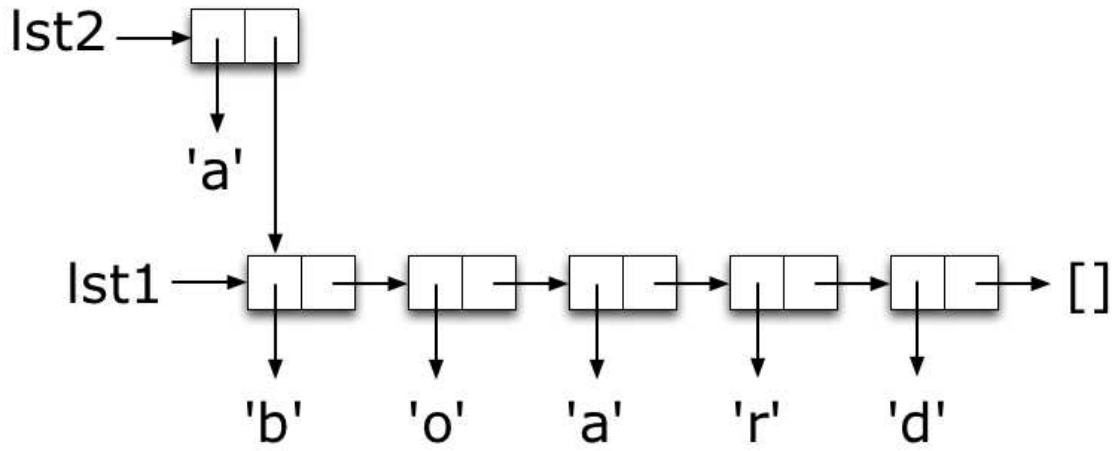
```
  factorial(3)
= if (3 == 1) then 1
              else factorial(3 - 1) * 3
= factorial(2) * 3
= (if (2 == 1) then 1
               else factorial(2 - 1) * 2)
  * 3
= factorial(1) * 2 * 3
= (if (1 == 1) then 1
               else factorial(1 - 1) * 1)
  * 2 * 3
= 1 * 2 * 3
```

関数プログラミングとは
不変データプログラミングのことです。

さまざまな不変データがあります。

不変データの代表選手は
一方向リストです。

lst1の先頭に要素を追加しても
lst1は破壊されません。



なんとなく分かってきましたが、
関数プログラミングでは
できることが限られていませんか？

はい。

関数プログラミングでできる範囲は
命令プログラミングでできる範囲よりも
狭いのです。

ただし、あなたが想像するより
はるかに広い範囲の問題を扱えます。

ディスプレイに表示することが
副作用であるなら
関数プログラミングでは
ゲームは作れないのですか？

関数プログラミングだけでは
ゲームは作れません。

通常、関数プログラミングは
命令プログラミングと
組みわせて使います。

具体的には
関数プログラミングで書いた関数は
命令プログラミングの手続きから
呼び出して使います。

では逆に
関数プログラミングの関数から
命令プログラミングの手続きを
呼び出して使ってはいけないのですか？

副作用のない関数から
副作用のある手続きを呼び出すと
その関数には副作用があることになり
関数ではなく手続きになってしまいます。

せっかくの関数が台無しです。

そこまでして、
どうして関数プログラミングを
使うのですか？

関数プログラミングには
たくさんのメリットがあります。

たとえば
関数は副作用を持たないので
テストするのが簡単です。

```
test(factorial(5) == 120);
```

関数プログラミングでは
よくテストされた関数を使い
より大きな関数を作ります。

このため、大きな関数にも
バグが入り込みにくいのです。

また、静的型付き関数型言語を使うと
コンパイル時にたくさんの
エラーが発見できます。

静的型付き命令型言語で
コンパイルする時よりも
はるかに多くのエラーを
発見できます。

コンパイルが通れば
プログラムが正しく動くことも
たくさんあります。

これは静的型付き命令型言語では
体験できないことです。

プログラムの開発サイクルで
一番コストがかかるのは保守です。

静的型付き関数型言語では
プログラムのどこかを変更する場合
変更すべき部分のほとんどを
見つけてくれます。

つまり、静的型付き関数型言語では
他の手法に比べて
プログラムの保守が容易だと言えます。

もう少し関数プログラミングで
問題を解いてみましょう。

10,20,30,40,50
を格納する配列またはリストがあります。

0から数えて n 番目の要素に n を掛けて
足し合わせなさい。

つまり

10*0 + 20*1 + 30*2 + 40*3 + 50*4

という計算をします。

問) C 言語で書いてください。

```
int calc(int n, int arr[]) {
  ...
}
```

こんな感じになります。

```
int calc(int n, int arr[]) {
  int r = 0;
  for (int i = 0; i < n; i++) {
    r = r + arr[i] * i;
  }
  return r;
}
```

関数プログラミングでは
どうやって問題を解くのでしょうか？

関数プログラミングでは
map & reduce という考え方で
この問題を解けます。

関数プログラミングの
醍醐味を味わうために
少し準備をします。

階乗の疑似コードを
思い出してください。

```
int factorial(int n) =
  if (n == 1) then
    1
  else
    factorial(n - 1) * n;
```

型注釈は書かないことにします。

```
factorial n =
  if (n == 1) then
    1
  else
    factorial (n - 1) * n;
```

引数から丸括弧が消えたことにも
注意してください。

```
factorial n =
  if (n == 1) then
    1
  else
    factorial (n - 1) * n;
```

関数呼び出しのときも
丸括弧は不要だとします。

```
factorial 3;
→ 6
```

準備ができました。

今解きたい問題は

10*0 + 20*1 + 30*2 + 40*3 + 50*4

でした。

第一引数にリストの長さ、
第二引数にリストを取り
計算結果を返す
関数を定義します。

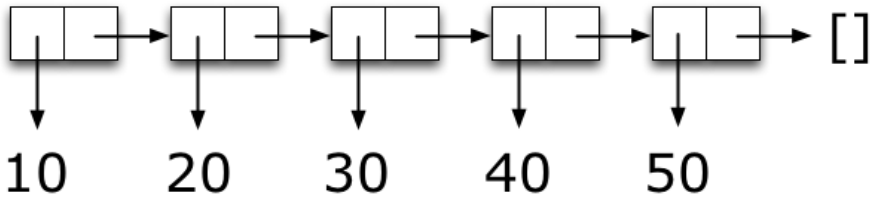
```
calc n lst = ...;
```

リストのリテラル(直接的な表現)
として以下を使います。

```
[10, 20, 30, 40, 50]
```

`[10, 20, 30, 40, 50]`

を図示するとこうなります。



リストの自動生成を使って
カウンタを用意します。

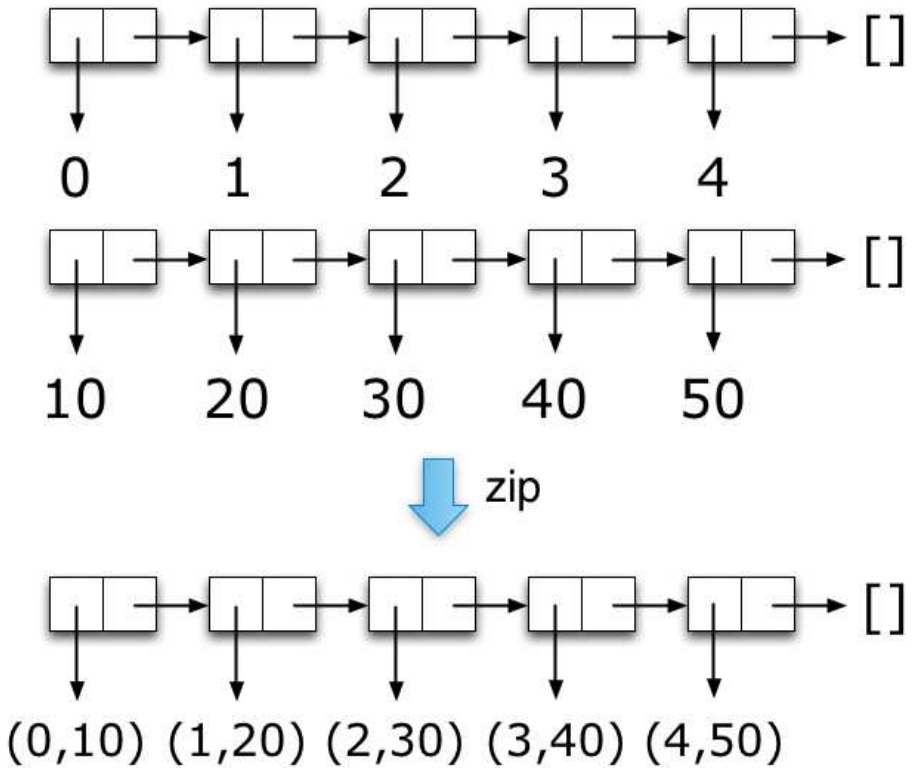
```
[0 .. 4];
→ [0,1,2,3,4]
```

後で 4 を n - 1 に置き換えます。

リストが2つあると扱いにくいので、
2つのリストを1つに
閉じ合わせます。

```
zip [0 .. 4] [10,20,30,40,50];
→ [(0,10),(1,20),(2,30),(3,40),(4,50)]
```

zip は新しいリストを生成します。



(x,y) は「組」(タプル)と呼ばれます。

2つのデータを1つのデータとして
扱えます。

```
[(0,10),(1,20),(2,30),(3,40),(4,50)]
```

は、「整数と整数の組み」のリストです。

引数として整数と整数の組みを取り
掛け算して返す関数を定義します。

```
mul (i,x) = x * i;
```

ここで出てくる丸括弧は
組みの直接的な表現であることに
注意してください。

今手にしているのは、
整数と整数の組みのリスト

```
[(0,10),(1,20),(2,30),(3,40),(4,50)]
```

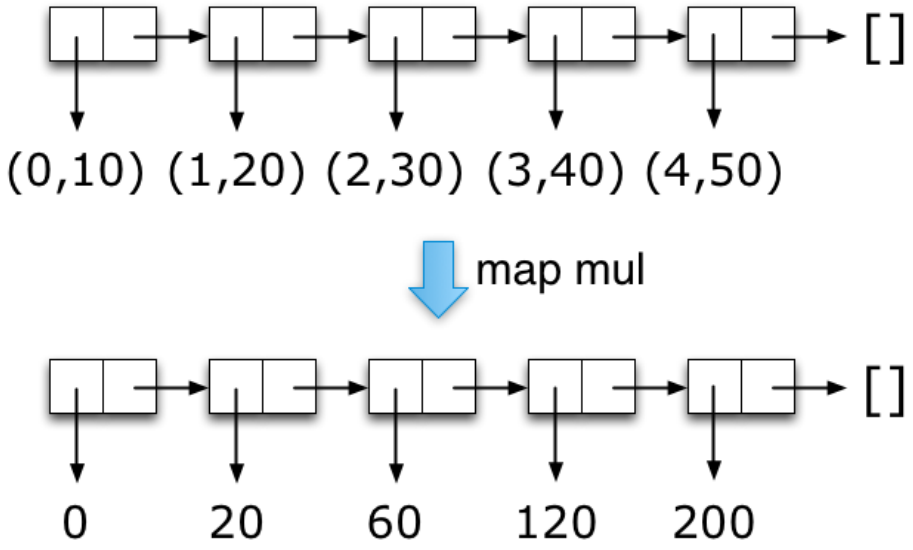
と関数 `mul` です。

各要素を `mul` に渡してみます。

各要素を mul に渡すには
map を使います。

```
map mul [(0,10),(1,20),(2,30),(3,40),(4,50)];
→ [0,20,60,120,200]
```

map も新しいリストを生成します。



リストの要素すべてを
足し合わせるには
畳み込み関数 reduce を使います。

```
reduce + 0 [0,20,60,120,200];
→ ((((0 + 0) + 20) + 60) + 120) + 200
→ 400
```

```
((((0 + 0) + 20) + 60) + 120) + 200
```

初期値 0 に対して
リストの要素を左から
+ で畳み込んでいます。

プログラムを完成させます。

```
calc n lst =
  reduce + 0 (map mul (zip [0 .. n-1] lst));
```

ここで |> という演算子を導入します。

```
(f |> g) x = g (f x)
```

この定義の意味は
分らなくて構いません。

```
calc n lst =
  reduce + 0 (map mul (zip [0 .. n-1] lst));
```

は |> を使うと、以下のようになります。

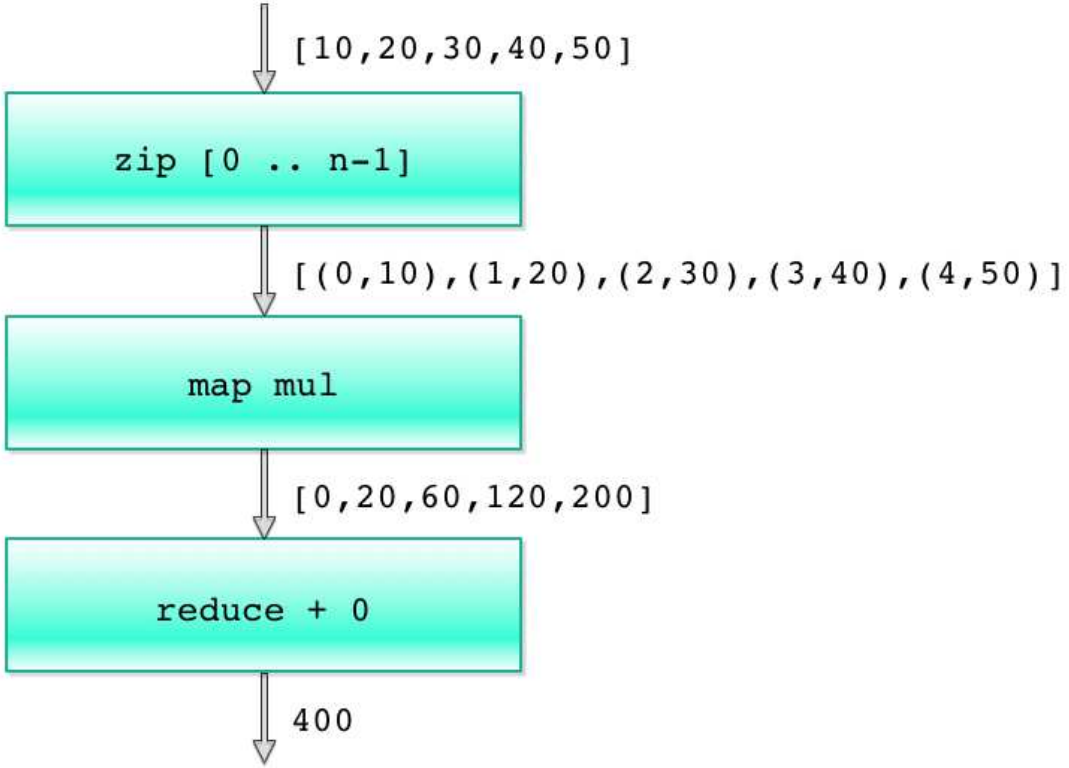
```
calc n = zip [0 .. n-1]
       |> map mul
       |> reduce + 0
        ;
```

```
calc n = zip [0 .. n-1]
       |> map mul
       |> reduce + 0
        ;
```

はシェルでのパイププログラミングの
ように見えませんか？

関数プログラミングとは
パイププログラミングのような
プログラミングのことです。

図示してみましょう。


[10,20,30,40,50]
zip [0 .. n-1]
[(0,10),(1,20),(2,30),(3,40),(4,50)]
map mul
[0,20,60,120,200]
reduce + 0
400

UNIX の哲学

"Do one thing and do it well"

パイプの作者 Doug McIlroy

関数プログラミングでは、
一つのことをうまくやれるように
関数を作ります。

関数プログラミングでは
バグがないと確信できる関数を
組み合わせて
新しい関数を作ります。

関数プログラミングとは
部品プログラミングです。

```
calc n = zip [0 .. n-1]
       |> map mul
       |> reduce + 0
        ;
```

zip の引数は2つのはずです。

zip [0 .. n-1] は、
引数が足りなくないですか？

関数型言語には
「部分適用」という機能があり
引数の一部だけを
関数に渡せます。

ここでまた準備をします。
新しい型注釈を導入します。

新しい型注釈は
本文とは別の行に書きます。

```
factorial : int -> int;
factorial n =
  if (n == 1) then
    1
  else
    factorial (n - 1) * n;
```

一番右側が返り値の型
それ以外は引数の型です。

```
factorial : int -> int;
```

引数が複数ある例を示します。

```
count : char -> string -> int;
```

第一引数の型が文字、
第二引数の型が文字列、
返り値の型が整数
という意味です。

```
count : char -> string -> int;
```

は、指定された文字が
文字列の中に何個あるか
数える関数だとします。

```
count 'a' "banana";
→ 3
```

準備ができました。

部分適用の動作原理を考えましょう。

count の型注釈に
丸括弧を補ってみます。

```
count : char -> (string -> int);
```

```
count : char -> (string -> int);
```

は、第一引数が文字で
`string -> int` を返す
関数だと解釈できます。

`string -> int`

とは何か考えてください。

```
string -> int
```

は、第一引数の型が文字列で
返り値の型が整数である
関数の型です。

```
count : char -> (string -> int);
```

つまり count は、
引数として文字を取り
関数を返す関数です。

count に例えば 'a' を部分適用した関数 count_a を作ってみましょう。

```
count_a : string -> int;
count_a = count 'a';

count_a "banana";
→ 3
```

このように部分適用とは
一部の引数を固定して
新たに関数を作ることです。

部分適用によって
汎用の部品から
専用の部品を
作り出せます。

今は、zip [0 .. n-1] が何かを
考えているのでした。

```
calc n = zip [0 .. n-1]
       |> map mul
       |> reduce + 0
        ;
```

zip の型は、以下の通りです。

```
zip : [a] -> [b] -> [(a,b)];
```

a や b は型変数と呼ばれます。

型注釈に型変数を持つ関数は、
引数の型に依存せずに仕事をします。

実際に利用するときには、
型変数が何かの型に固定されます。

たとえば、
a が `int` に固定されたり
b が `char` に固定されたりします。

`[int]` と書くと
要素の型が `int` である
リストの型という意味だとします。

[a] は、要素が何かの型を持つ
リストの型だという意味です。

リストの中の要素は、
すべて同じ型を持っていないと
いけません。

`[1,2,3,4]` は OK ですが、

`[1,'a',3.1,"hello"]` は NG です。

`zip [0 .. n-1]`
の型を考えてみましょう。

```
zip [0 .. n-1]
```

と

```
zip : [a] -> [b] -> [(a,b)];
```

を見比べると
a は int だと分ります。

よって以下のようになります。

```
zip [0 .. n - 1] : [b] -> [(int,b)];
```

引数の数が減ることにも
注意しましょう。

```
zip : [a] -> [b] -> [(a,b)];
```

次に `map mul` の型を考えましょう。

map の型は以下の通りです。

```
map : (a -> b) -> [a] -> [b];
```

mul の型は以下の通りです。

```
mul : (int,int) -> int;
mul (i,x) = x * i;
```

(int,int) は整数と整数の組み
という意味です。

```
mul : (int,int) -> int;
map : (a -> b) -> [a] -> [b];
```

から map mul の型が
以下のように推論できます。

```
map mul : [(int,int)] -> [int];
```

同様に

```
+ : int -> int -> int;
reduce : (b -> a -> b)
          -> b -> [a] -> b;
```

から、reduce + 0 の型は
以下のように推論できます。

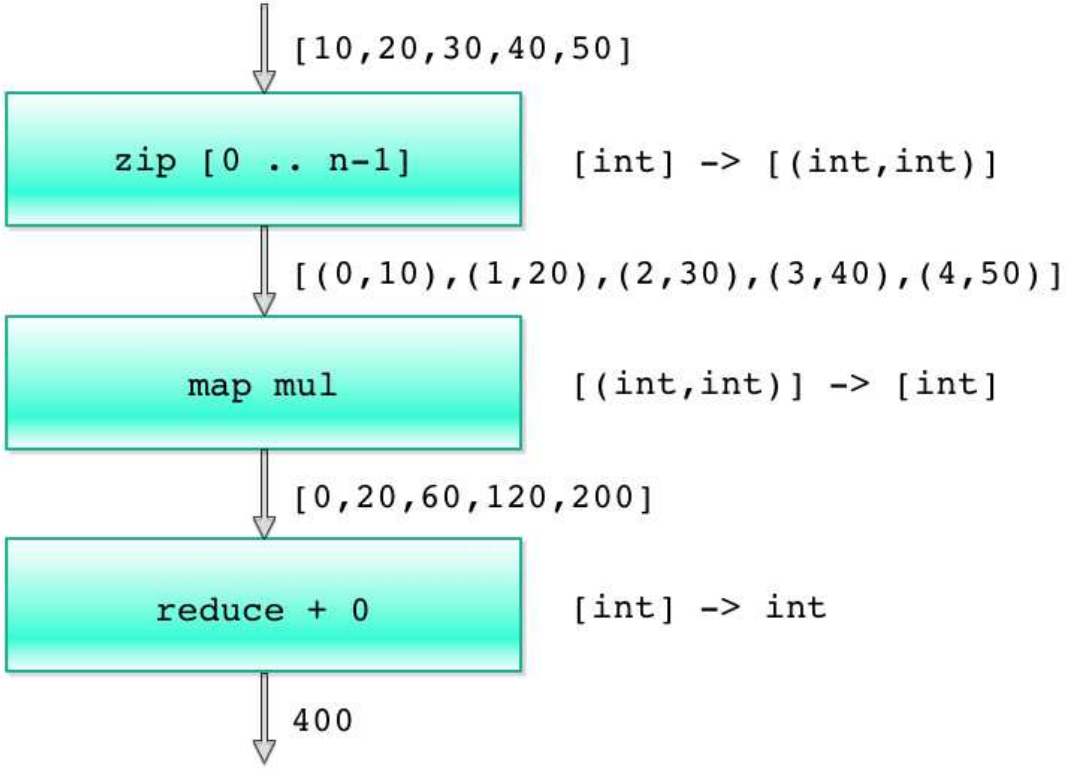
```
reduce + 0 : [int] -> int;
```

```
zip [0 .. n - 1] : [b] -> [(int,b)];
map mul : [(int,int)] -> [int];
reduce + 0 : [int] -> int;
```

から、b は int だと
推論できます。

(本当は |> の型も必要です)

図示してみましょう。


[10,20,30,40,50]
zip [0 .. n-1]
[int] -> [(int,int)]
[(0,10),(1,20),(2,30),(3,40),(4,50)]
map mul
[(int,int)] -> [int]
[0,20,60,120,200]
reduce + 0
[int] -> int
400

1つ前の返り値の型と
次の引数の型が
同じであることが分かります。

factorial の型を推論してみましょう。

```
factorial n =
  if (n == 1) then
    1
  else
    factorial (n - 1) * n;
```

引数は1つですから、
? -> ?
の形をしているはずです。

```
factorial : ? -> ?;
factorial n =
  if (n == 1) then
    1
  else
    factorial (n - 1) * n;
```

n == 1 から
第一引数は int だと
分ります。

```
factorial : int -> ?;
factorial n =
  if (n == 1) then
    1
  else
    factorial (n - 1) * n;
```

1 を返しているので、
返り値の型は int だと分ります。

```
factorial : int -> int;
factorial n =
  if (n == 1) then
    1
  else
    factorial (n - 1) * n;
```

これまで `factorial` が
どのように作られているか
から型を推論しました。

factorial は再帰で定義されているので
使われてもいます。

```
factorial : int -> int;
factorial n =
  if (n == 1) then
    1
  else
    factorial (n - 1) * n;
```

```
factorial (n - 1) * n;
```

の部分では、型は合っているでしょうか？

```
* : int -> int -> int;
```
ですから factorial の返り値の型は
int のはずです。

```
n - 1 : int;
```
ですから、factorial の引数の型は
int のはずです。

つまり、こうです。
```
factorial : int -> int;
```

どう組み立てられているかから

```
factorial : int -> int;
```

どう使われているかから

```
factorial : int -> int;
```

のように双方向の型推論をして
型に間違いがないことが確かめらました。

プログラムを式で組み立てると
あらゆる部分式で
内側から組み立てた型と
外側から利用する型とで
双方向の型検査を実施できます。

これが
静的型付き関数型言語を使うと
たくさんエラーを発見できる理由です。

型注釈は必ずしも書く必要がありません。

コンパイラーが推論するので、
推論結果を自動的に挿入できます。

難しい関数を書く場合は
先に型注釈を書いて
型のレベルで設計することもあります。

その場合、
型が実装を導いてくれる
ことがたびたびあります。

そういう体験をすれば
関数プログラミングの
真髄に触れたことになります。

頑張ってください。

関数プログラミングに
興味を持った人は
以下の記事を読むとよいでしょう。

「入門 関数プログラミング」

http://gihyo.jp/dev/feature/01/functional-prog