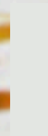
論理学入門 II

ラムダ計算と関数型言語
Part I, Part II



はじめに

- 小論は、1930-40年代に、Churchが作り上げた「ラムダ計算」の理論が、特に80年代の一連の関数型言語の誕生にどのような影響を与えたかを見ることを、一つの目標にしている。
- 新しいプログラム言語の誕生と発展の物語は、先行した技術の到達点と非常に複雑な時代の背景の中で語られるのが常である。いわゆる関数型言語と言われるものに限っても、むしろ、その「同一性」より「差異」の方が目立つほどで、それぞれの特徴は多岐に渡っている。
- 小論は、いくつかの関数型言語 LISP, Haskell, Ocaml, Coq 等を取り上げているのだが、基本的には それらの言語がChurchのラムダ計算から継承したもの、端的に言えば、AbstractionとApplication をどのように実装しているのかにフォーカスしている。

はじめに

- そうした試みは、様々の関数型言語の「紹介」としては、不十分で、トリビアルで、自明なことのように思われるかもしれない。一方で、「ラムダ計算と関数型言語」というコンテンツが、何故に、「論理学」というタイトルの元に収められているのか釈然としない人もいると思う。こうした疑問・不満は、確かに当たっていると思う。
- 振り返ってみると、基本的には「型を持つラムダ計算」の登場、すなわち「関数は型を持つ」という認識の登場が、それ以降のプログラム言語の発展の大きな転換点だったと筆者は考えている。いわゆる「関数型言語」の成立は、計算機科学の文脈での、こうした画期をなすものである。

はじめに

- 関数型言語の諸特徴は、現在のプログラム言語の多くに取り込まれている。IT技術者の多くは、すでに、それを日常的に利用している。ただ、残念ながら、関数型言語そのものに対する関心は、あまり広がっていないように思う。関数型言語について語ることは、意味があると考えている。
- 問題は、「型付きラムダ計算 = 関数は型を持つ」という視野の射程のさらにその先に、理論的認識が広がっているということである。筆者は、それは「従属型理論 Dependent Type Theory」であると考えている。そこで、初めて「計算」と「論理」の関係が明らかになる。
- その意味でも、今回の「論理学 II – ラムダ計算と関数型言語」は、中間的なものである。次回の「論理学 III – 計算と論理」で、きちんと話をしたいと考えている。

ラッセル
型の理論

ヒルベルト
プログラム

ゲーデル
不完全性定理

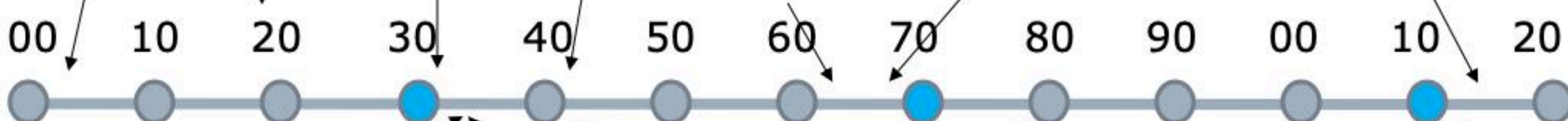
ゲーデル
連続体仮説
無矛盾性証明

コーエン
連続体仮説
独立性証明

グロタンティック
Scheme / Topos

ローヴェール
Topos理論

ヴォエヴォドスキー
Univalant
Foundation



チューリング
停止問題

チャーチ
ラムダ計算

チャーチ
型付きラムダ計算

マーティン・レフ
Dependent
Type Theory

ヴォエヴォドスキー
Homotopy
Type Theory

カリー＝ハワード対応

数学＝論理学での
重要な発見 と
計算機上での新しい
数学理論の実装

LISP
1958

Prolog
GHC

Haskel
1985
ML
OCaml

COQ
1984
Agda

サイモン LT

ロビンソン
Resolution

第五世代
Expert
System

UniMath

Agenda ラムダ計算と関数型言語

Part I 型のないラムダ計算

- 関数とは何か
- λ記法（ラムダ記法）
- Abstraction -- λ記法による関数の「抽象化」
- Application -- λ式への式の「適用」
- Substitution -- 変数への値の「代入」
- 型のないラムダ計算の形式化
- ちょっと変わったラムダ式

Agenda ラムダ計算と関数型言語

Part II 型付きラムダ計算

- □ ラムダ計算への型の導入
- □ 単純な変数 v は型を持つ
- □ 抽象化 λx. eは型を持つ
- □ 適用 e_1 e_2は型を持つ
- □ 型付きラムダでの計算
- □ 「適用」は型で制限される

- □ 型付きラムダ式と証明の解釈

Part I

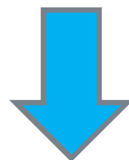
型のないラムダ計算

様々な「計算可能性」へのアプローチと その同値性の認識

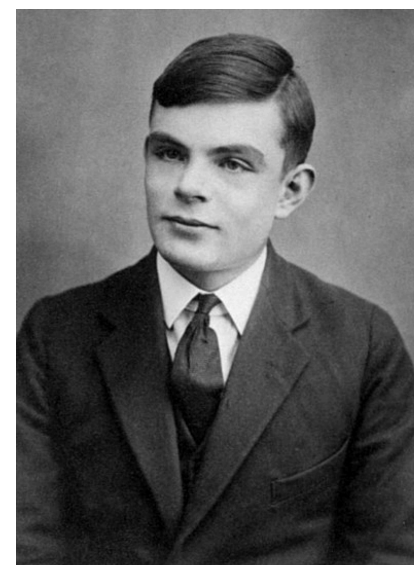
Kurt Gödel
1906-1978

帰納関数論

Alonzo Church
1903-1995



ラムダ計算



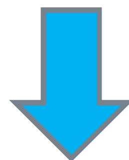
Alan Turing
1912-1954

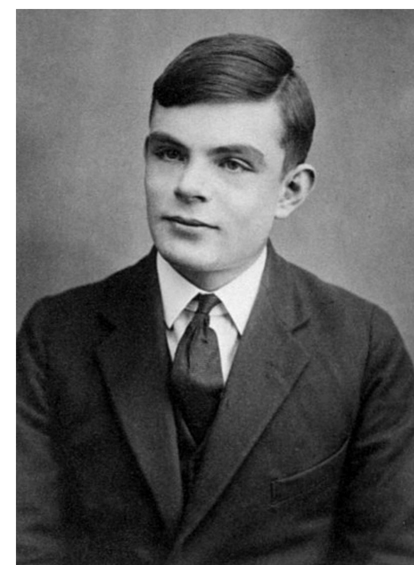
チューリングマシン

様々な「計算可能性」へのアプローチと
その同値性の認識

Kurt Gödel
1906-1978

Alonzo Church
1903-1995





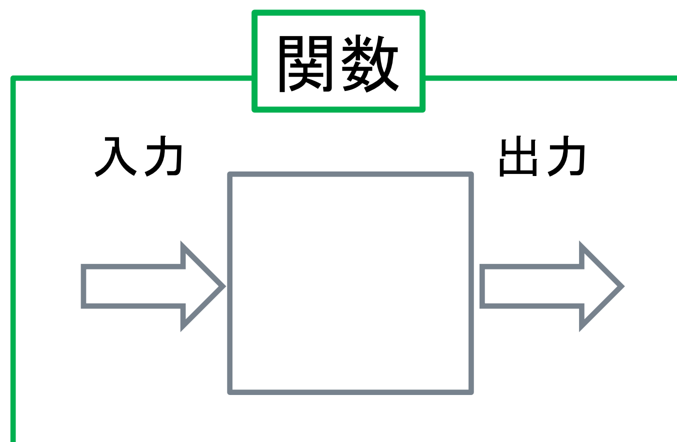
Alan Turing
1912-1954

帰納関数論　＝　ラムダ計算　＝　チューリングマシン

関数とは何か？

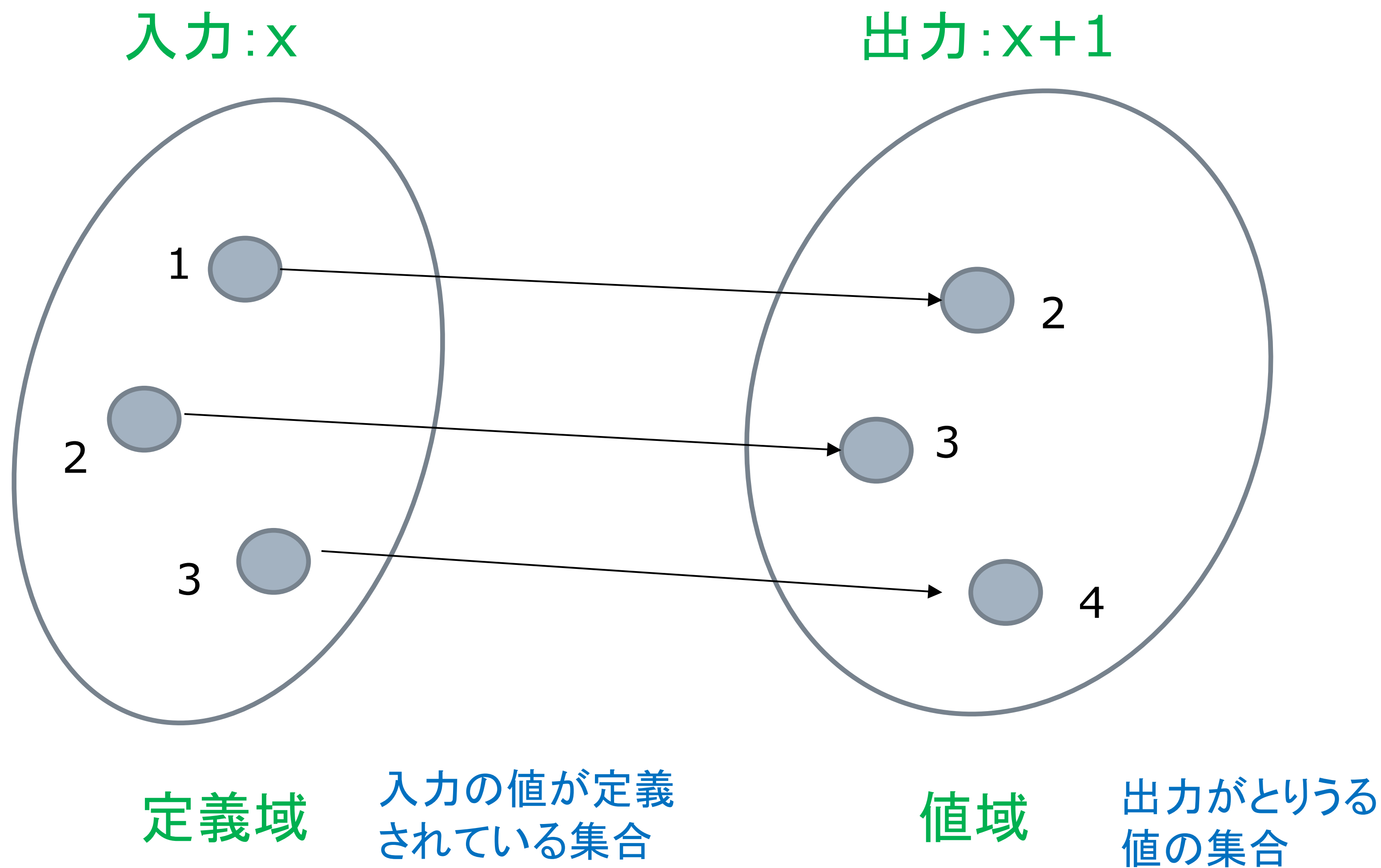
関数とは何か？

- 関数とは、ある入力に対して、ある出力を返すものだと考えよう。



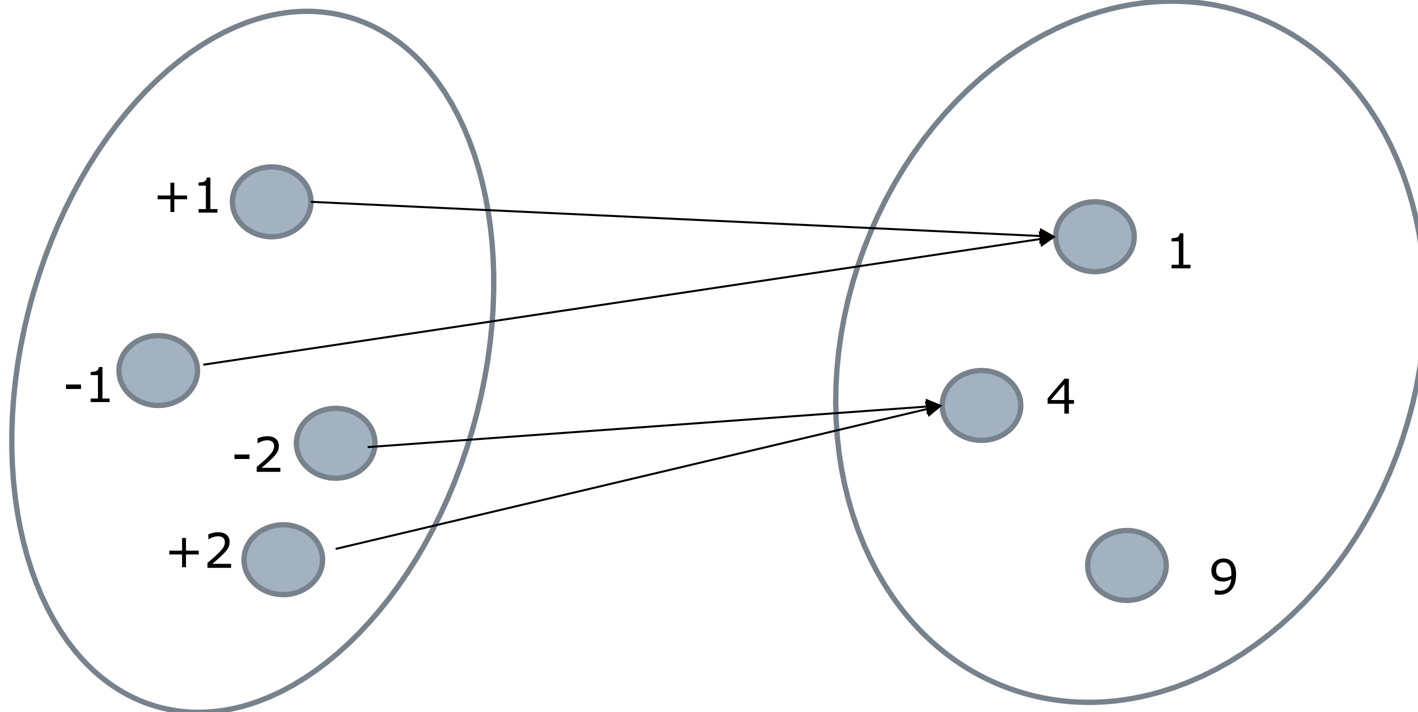
- 例えば、 x+1 は、変数xに1という値が入力として与えられれば、1+1=2を出力として返す関数である。
- 例えば、 x^2 は、変数xに2という値が入力として与えられれば、$2^2=4$を出力として返す関数である。

関数の例:　　$f(x) = x + 1$



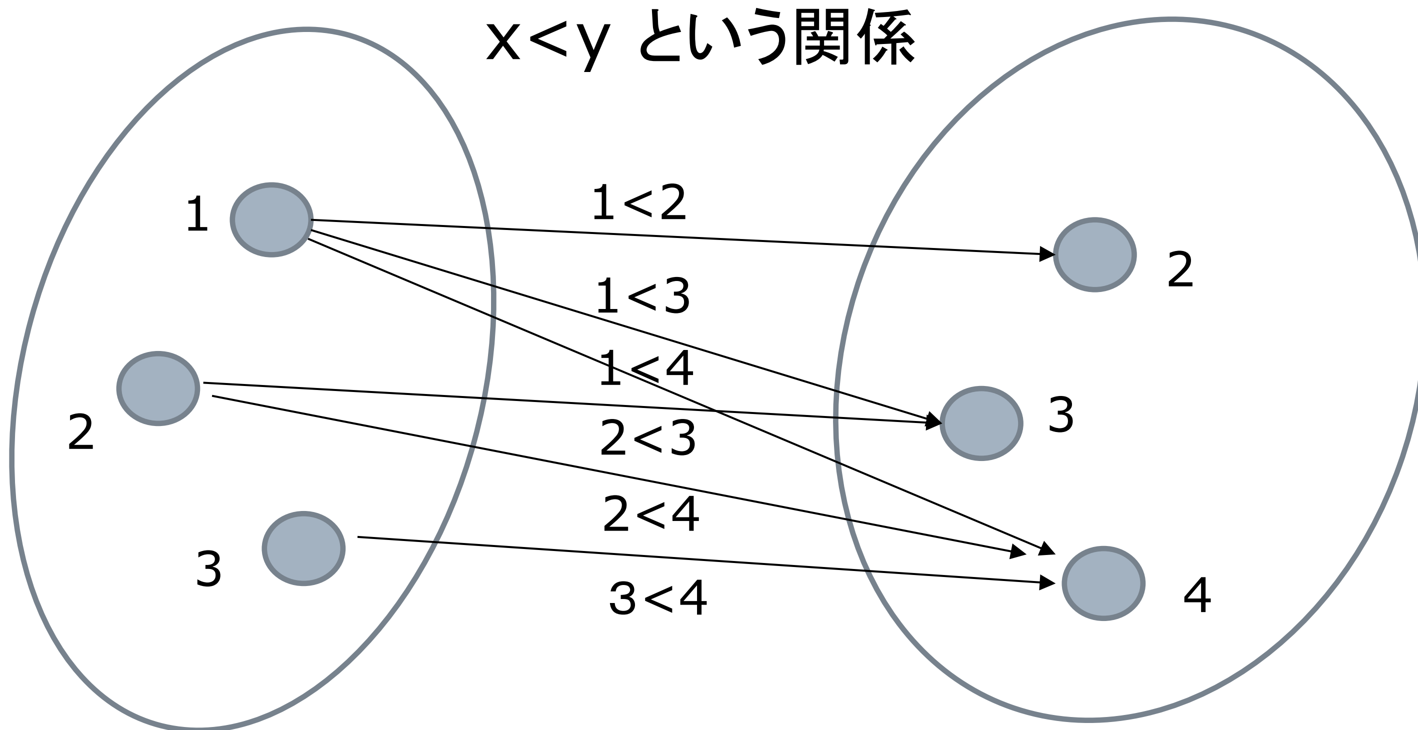
関数の例：　$f(x) = x^2$

二つの異なる値の入力が、同じ値の出力をすることがある。



関数でない例

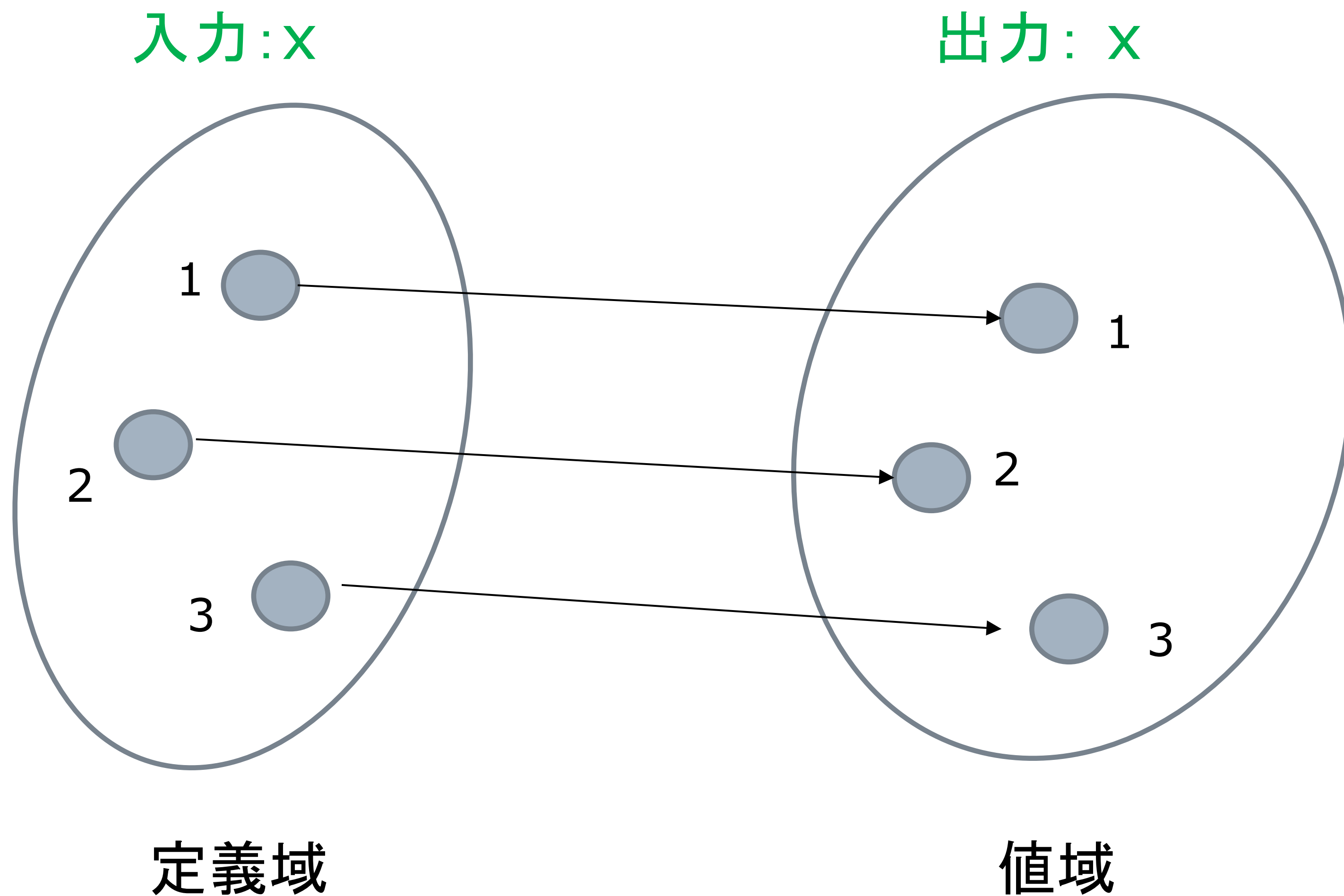
一つの入力の値は、一つの出力の値を持つ。二つ以上の値を出力をするのは関数ではない。



x<y という関係は、
関数ではない

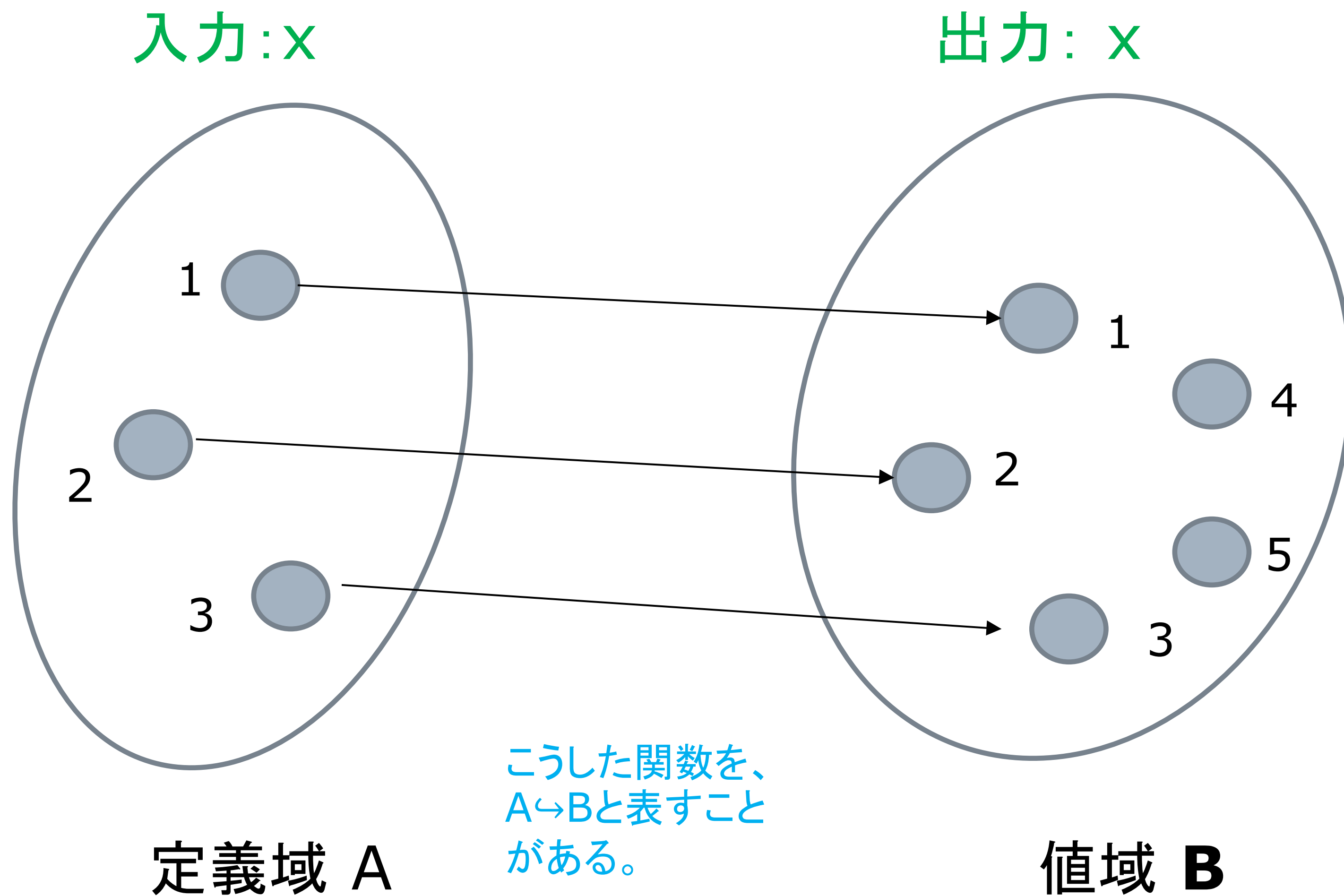
関数の例: $f(x) = x$

同じものを同じものにうつす自明な関数



関数の例:　$f(x) = x \quad A \subset B$

同じものを同じものにうつす関数



λ記法（ラムダ記法）

関数の表記としてのλ記法（ラムダ記法）

- x+1が、xの関数であることを、

  $\lambda x.(x+1)$

  と表す。
- この表現は、関数 x+1が、xに入力を受け取り(そのことが冒頭の、λxで表現されている)、そのxの値について、x+1を出力として返すことを意味する。
- 同様に、

  y+1が、yの関数であることを、

  $\lambda y.(y+1)$

  z+1が、zの関数であることを、

  $\lambda z.(z+1)$

  と表すことができる。

同じ関数？

- x+1 は、変数xに1という値が入力として与えられれば、1+1=2を出力として返す関数である。
- y+1 は、変数yに1という値が入力として与えられれば、1+1=2を出力として返す関数である。
- z+1 は、変数zに1という値が入力として与えられれば、1+1=2を出力として返す関数である。

同じ関数？

- x+1は、xの関数である。y+1は、yの関数であり、z+1はzの関数である。ただ、これらは、関数を表す式の形は違っているが、例えば、入力 1に対して、出力 2を返すという点では、同じ働きをする。
- 見かけは違っていても、同じ入力に対して同じ出力を返す関数は、同じ関数であると見なすことができる。

同じ関数であることを表現する

- x+1, y+1, z+1 が関数として同じものになりうることを x+1 = y+1 = z+1 と表現してはいけない。
  この式は、x = y = z という別のことを表現している。

- x+1, y+1, z+1 が関数として同じであることは、先のラムダ表記を使って、次のように表現できる。

  $$\lambda x.(x+1) = \lambda y.(y+1) = \lambda z.(z+1)$$

- これは、関数を表す式の変数を、一斉に、他の名前の変数に取り換えても、関数の働きは変わらないことを表している。
  (これを、α-Conversion という。)

二つの変数を含んだ関数の例

- $M = x^2+y$ としよう。このMを、
  xの関数としてみることを λx.M すなわちλx.(x^2+y)で表し、
  yの関数としてみることを λy.M すなわちλy.(x^2+y)で表わすことができる。
- これは、普通の関数の表記で考えると、
  λx.M は、 $f(x)=x^2+y$
  λy.M は、 $g(y)=x^2+y$
  と表記することに相当する。

二つの変数を含んだ関数

- $M = x^2+y$ は、また、xとyの二つの変数の関数とみることが出来る。

- これを $\lambda(x,y).M$ すなわち $\lambda(x,y).(x^2+y)$ と表そう。
  $\lambda(x,y).M$ は、　$h(x, y)=x^2+y$
  と表記することに相当する。
  $\lambda(x,y).(x^2+y)$ は、$\lambda xy.(x^2+y)$ と書いてもいい。

- 大事なことは、$\lambda xy.(x^2+y)$ は $\lambda x.(\lambda y.(x^2+y))$ と等しいことだ。（λによる変数の結合は右結合である。）
  $\lambda(x,y).(x^2+y) = \lambda xy.(x^2+y) = \lambda x.(\lambda y.(x^2+y))$

二つの変数を含んだ関数

- 繰り返す。
  大事なことは、それは $\lambda x.(\lambda y.(x^2+y))$と等しいことだ。
  $\lambda(x,y).(x^2+y) = \lambda xy.(x^2+y) = \lambda x.(\lambda y.(x^2+y))$

- 二変数の関数 $\lambda(x,y).(x^2+y)$は、yについての一変数の関数 $\lambda y.(x^2+y)$ を、あらためてxについての一変数の関数 $\lambda x.(\lambda y.(x^2+y))$と見直すことで得られる。

  （このように、多変数の関数を一変数の関数の組み合わせに還元することをCurry化という。後述。）

Abstraction

λ記法による関数の「抽象化」

抽象化としてのλ記法

- M = x^2+y としよう。先に見たように、
  λx.M は、　$f(x)=x^2+y$
  λy.M は、　$g(y)=x^2+y$
  λxy.M は、　$h(x, y)=x^2+y$
  と表記することに相当する。

- λ記法を使うことによって、上記の f, g, h のような関数の具体的な名前を使わなくても、関数のある特徴を抽象的に示すことが出来る。これをλによる抽象化と呼ぶ。

- 「λによる抽象化」によって、「無名の関数」が定義できる。

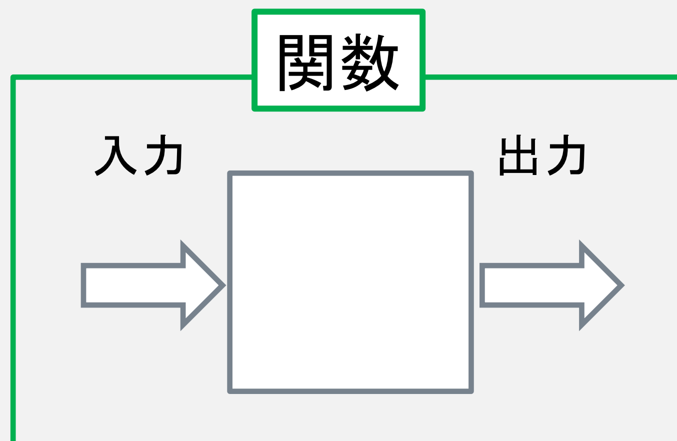
λによる抽象化の例

N $= x^2 + ax + b$ としよう。

- **λx.N** $= \lambda x.(x^2 + ax + b)$
  xの関数としてみた $x^2 + ax + b$
- **λa.N** $= \lambda a.(x^2 + ax + b)$
  aの関数としてみた $x^2 + ax + b$
- **λb.N** $= \lambda b.(x^2 + ax + b)$
  bの関数としてみた $x^2 + ax + b$
- **λxa.N** $= \lambda xa.(x^2 + ax + b)$
  xとaの関数としてみた $x^2 + ax + b$
- **λxab.N** $= \lambda xab.(x^2 + ax + b)$:
  xとaとbの関数としてみた $x^2 + ax + b$

ふりかえり。関数とは何か？

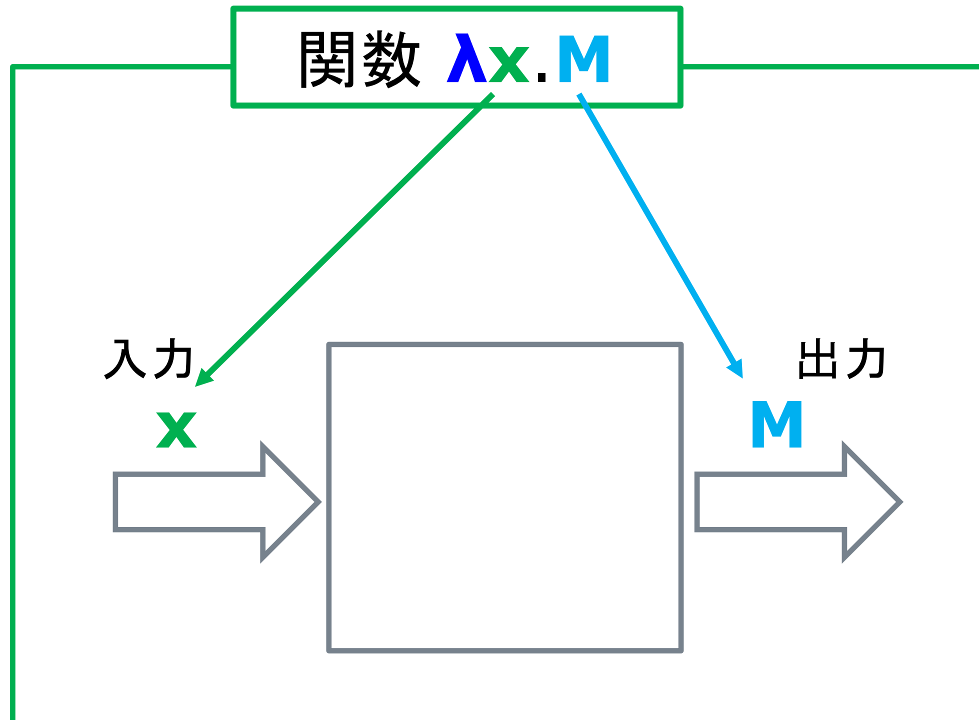
- 関数とは、ある入力に対して、ある出力を返すものだと考えてきた。



- 例えば、 x+1 は、変数xに1という値が入力として与えられれば、1+1=2を出力として返す関数である。
- 例えば、 x^2 は、変数xに2という値が入力として与えられれば、$2^2=4$を出力として返す関数である。

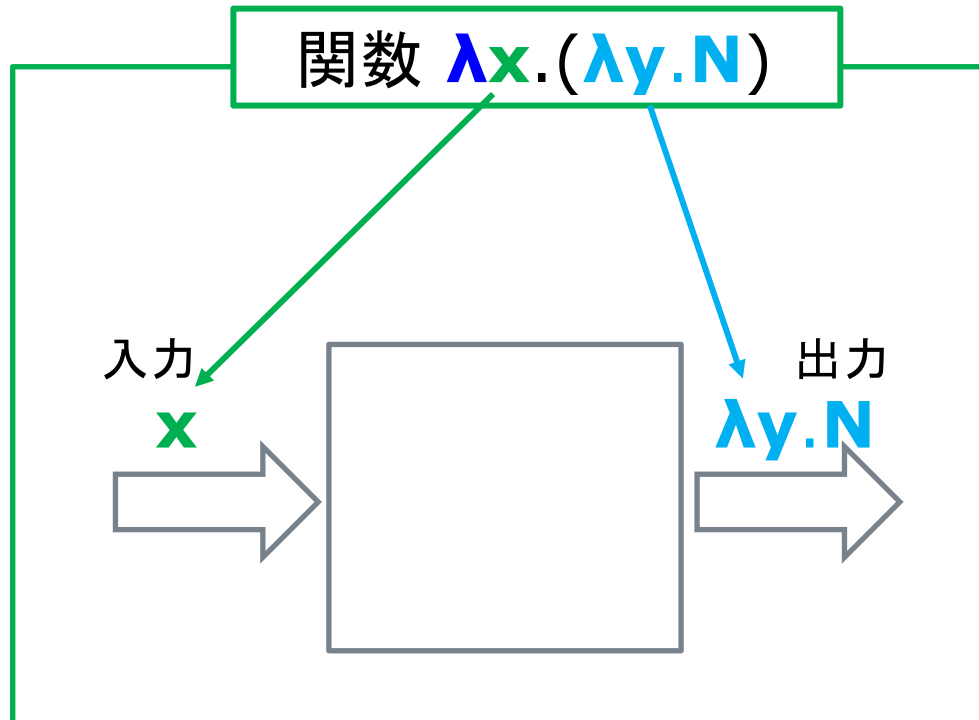
抽象化 **λx.M** があらわす関数

入力となる変数を**x**, 出力となる式(xとかx+1とかx^2とか)を**M**で表すと、λによる抽象化 **λx.M** は、次の関数を表す。



抽象化 λx.(λy.N) があらわす関数

- 式Mは、単純な式だけとは限らない。ラムダを含む式（ラムダ式）も式である。次の例では、M = λy.N **である。**



この関数の出力は、yを変数とする関数である。

Application
λ式への式の「適用」

λ式へ式を「適用」することで、
関数の値（出力）を計算する

関数が返す値の計算

- $L = x^2+3$ としよう。
  x=1の時のLの値は $1^2+3 = 1+3 = 4$
  x=2の時のLの値は $2^2+3 = 4+3 = 7$
  x=3の時のLの値は $3^2+3 = 9+3 = 12$
- Lを、xの関数とみた時、$\lambda x.L = \lambda x.(x^2+3)$ である。
  先の計算を次のように表す。
  $\lambda x.(x^2+3)$ **1** $= 4$
  $\lambda x.(x^2+3)$ **2** $= 7$
  $\lambda x.(x^2+3)$ **3** $= 12$
- これは、$f(x)= x^2+3$ として、
  f(1), f(2), f(3) を計算するのと同じである。

関数を値として返す関数

- $M = x^2+y$ としよう。
  このMを、xの関数とみた時、x=2の時のMの値は 4+y
  このMを、yの関数とみた時、y=1の時のMの値は x^2+1
- λで抽象化されたλx.M, λy.Mが、例えば、x=2, y=1の時に返す値は、それぞれ、4+y, x^2+1といった関数である。すなわち、λx.M, λy.Mは、関数を値として返す関数であることに注意しよう。
- λxy.Mの時、二つの変数 x, yに値を与えると、この関数は、はじめて具体的な値をとる。例えば、x=2, y=1の時のMの値は、5である。

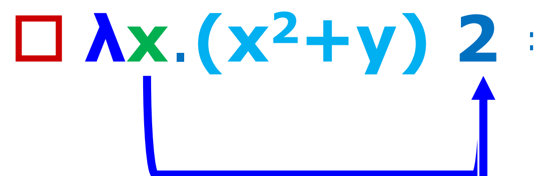
λ式への値（式）の適用

- λ式 Mに、ある値（式） Nを適用して関数の値を計算することを、λ式 Mの後ろに、値（式） Nを並べて、M Nのように表す。

- MNより、M(N) の方がわかりやすいなら、それでもいい。
  ただ、ラムダ計算では、MNのスタイルの方が一般的である。

λ式への値(式)の適用の例

- 例えば、**λx.(x²+y) 2** **の計算は**、次のようになる。



この関数の入力はxで、
その値は2である

λ式への値(式)の適用の例

- 例えば、$\lambda x.(x^2+y)\ 2$ の計算は、次のようになる。

- $\lambda x.(x^2+y)\ 2 = \lambda x.(x^2+y)\ 2 =$

この関数の入力はxで、
その値は2である

式に x=2 を代入して
値を計算する

λ式への値（式）の適用の例

- 例えば、$\lambda x.(x^2+y)\ 2$ の計算は、次のようになる。

- $\lambda x.(x^2+y)\ 2 = \lambda x.(x^2+y)\ 2 = 2^2+y = 4+y$

この関数の入力はxで、
その値は2である

式に x=2 を代入して
値を計算する

λ式への値(式)の適用の例

- 例えば、$\lambda y.(x^2+y)\ 1$ の計算は、次のようになる。

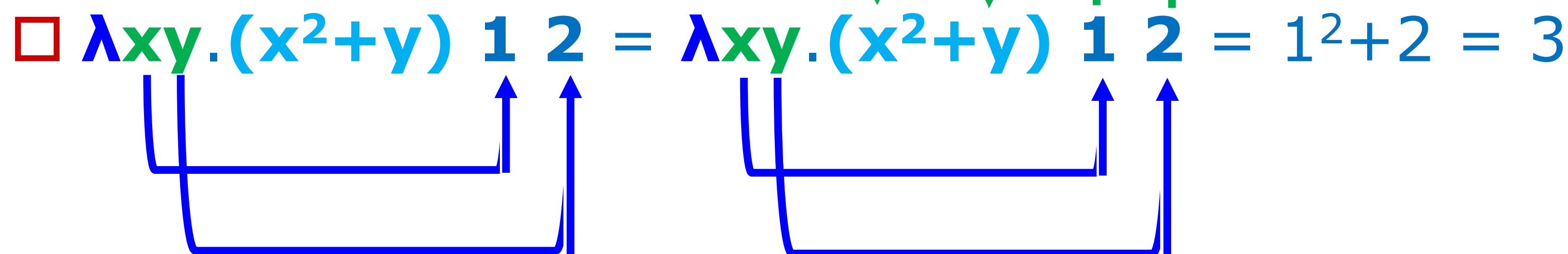
- $\lambda y.(x^2+y)\ 1 = \lambda y.(x^2+y)\ 1 = x^2+1$

この関数の入力はyで、
その値は1である

式に y=1 を代入して
値を計算する

λ式への値（式）の適用の例

- 例えば、**$\lambda xy.(x^2+y)$ 1 2** の計算は、次のようになる。



この関数の入力はxとyで、
xの値は1, yの値は2である

式に x=1,y=2 を代入して
値を計算する

Substitution

変数への値の代入

変数への値の代入

- 関数 f(x) が、x = aの時にとる値は、f(x)の中に現れる x に、x = aの値を代入することで得ることが出来る。これは、通常、f(a)と表わされる。
- 名前のないλ式 Mでの変数 xへの値Nの代入による関数の値の計算を、次のように表現する。

  M [x:=N]

- もっとも単純なλ式である、x自体が変数である時、代入は次のようになる。

  x [x:=a] = a

- 変数が違っていれば、代入は行われない。

  x [y:=a] = x

適用と代入

- 適用と代入の関係を見ておこう。
- λ式 λx.Mへの、aの適用 (λx.M) aとは、代入 M[x:= a]を計算することである。

$$(\lambda x.M)\ a = M[x:=a]$$

適用と代入の例

- 先の例での値の適用の直観的な説明は、代入を用いると、次のように考えることが出来る。(次のスライドも参照)

$(\lambda x.M)\ a = M[x:=a]$である。

$M = x^2+y$ とする。

$\lambda x.M\ 2 = \lambda x.(x^2+y)\ 2$
$= (x^2+y)\ [x:=2] = 2^2+y = 4+y$
$\lambda y.M\ 1 = \lambda y.(x^2+y)\ 1$
$= (x^2+y)\ [y:=1]=x^2+1$
$\lambda xy.M\ 1\ 2 = \lambda xy.\ (x^2+y)\ 1\ 2$
$= (x^2+y)\ [x:=1][y:=2] = 1^2+2 = 3$

適用と代入の例

- $\lambda x.(x^2+y)\ 2$ の計算は、次のようになる。

- $\lambda x.(x^2+y)\ 2 = (x^2+y)\ [x:=2] = 2^2+y = 4+y$

この関数の入力はxで、
その値は2である

式に x=2 を代入して
値を計算する

適用と代入の例

- $\lambda y.(x^2+y)\ 1$ の計算は、次のようになる。

- $\lambda y.(x^2+y)\ 1 = (x^2+y)\ [y:=1] = x^2+1$

この関数の入力はyで、
その値は1である

式に y=1 を代入して
値を計算する

適用と代入の例

- $\lambda xy.(x^2+y)\ 1\ 2$ の計算は、次のようになる。

- $\lambda xy.(x^2+y)\ 1\ 2 = (x^2+y)\ [x:=1][y:=2] = 1^2+2$
  $= 3$

この関数の入力はxとyで、
xの値は1, yの値は2である

式に x=1,y=2 を代入して
値を計算する

$\lambda xy.(x^2+y)\ 1\ 2 = (\lambda x.\ \lambda y.(x^2+y)\ 1)\ 2$
$(x^2+y)\ [x:=1][y:=2] = ((x^2+y)\ [x:=1])[y:=2]$
f x y = (f x) y で、関数の適用は、左結合である。

型のないラムダ計算の形式化

λ式の形式的定義

- これまでは関数とその引数、引数に具体的な値が与えられた時の関数の値をベースに、λ式とその値の計算を説明してきたが、ここでは、λ式の形式的な定義を与えよう。（正確なものではなく、簡略化したものであることに留意）

1. 変数 x,...はλ式である。(variable)
2. Mがλ式で、xが変数であるなら、
   λxによる抽象化 **λx.M** はλ式である。(abstraction)
3. M, Nがλ式であるなら、
   MへのNの適用 **M N** は、λ式である。(application)

λ式の形式的定義

先の定義と、次の定義は
<name> を「変数」
<expression>を「λ式」
と読み替えれば、同じものである。

$$
\begin{aligned}
<\text{expression}> \quad &:= \quad <\text{name}> \mid <\text{function}> \mid <\text{application}> \\
<\text{function}> \quad &:= \quad \lambda <\text{name}>.<\text{expression}> \\
<\text{application}> \quad &:= \quad <\text{expression}><\text{expression}>
\end{aligned}
$$

自由変数と束縛変数

- 次のような変数を「自由変数」という
  1. 変数x の自由変数は、xだけである
  2. λx. M の自由変数は、x 以外のMの自由変数である
  3. 適用 M N の自由変数は、M の自由変数と N の自由変数を合わせたものである。
- 自由変数以外の変数を「束縛変数」という。

- 例えば、$\lambda x.\,x$ には自由変数はなく、xが束縛変数である
- $\lambda x.\,yx$ の束縛変数はx で、唯一の自由変数はy である

λ計算の形式的ルール

- 次の三つのルールを利用して、λ式を(基本的には単純なものに)変換することをλ計算という。

1. **α-conversion:**
   抽象化に用いる変数の名前は、自由に変更出来る。
   例えば、 $\lambda x.(x^2+1) \Rightarrow \lambda y.(y^2+1) \Rightarrow \lambda z.(z^2+1)$
2. **β-reduction:**
   代入による計算ルール
   (λx.M) a => M[x:=a]
3. **η-conversion:**
   λx.(f x) => f
   xで抽象化された(f x)は、fに等しいということ。

β-reductionと代入ルール

- 実際のλ計算で、大きな役割を果たすのはβ-reductionである。その元になっている代入ルールを少し詳しく見ておこう。

1. $x[x := N] \equiv N$
2. $y[x := N] \equiv y$, if $x \neq y$
3. $(M1\ M2)[x := N] \equiv (M1[x := N])\ (M2[x := N])$
4. $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$,
   if $x \neq y$, provided $y \notin FV(N)$
   xとyが異なり、yがNの自由変数でないなら、(λx.M)への x:=Nの代入は、Mへのx:=Nの代入である

カッコの扱い（カッコの省略ルール）

- 一番外側のカッコは省略できる。
  (MN) => MN
  (λx. M) => λx. M
- 適用は「左-結合的」である。
  (M(NL)) => MNL
- 適用は抽象化より優先する。
  λx. (MN) => λx. MN
- 連続する抽象化は、一つのλに「右-結合的」にまとめられる。
  (λx.(λy. M)) => λx.(λy. M) => λxy. M

ちょっと変わったラムダ式

ちょっと変わったλ式1－ Ω

- λ式 $\Omega := (\lambda x.xx)\ (\lambda x.xx)$ を考えよう。
  これを計算してみる。

ちょっと変わったλ式1－ Ω

- λ式　Ω ：＝　*(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)　第二項の束縛変数の名前をyに変更
  ＝ (λx.xx) (λy.yy)　　α-conversion

ちょっと変わったλ式1－ Ω

- λ式　Ω := *(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λ**x**.**xx**) **(λy.yy)**　適用の代入形　β-reduction
  = **xx**[**x**:=**λy.yy**]

ちょっと変わったλ式1－ Ω

- λ式　Ω := *(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λx.xx) (λy.yy)
  = **xx**[**x**:=**λy.yy**]　代入の実行
  = (λy.yy) (λy.yy)

ちょっと変わったλ式1－ Ω

- λ式　Ω := *(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λx.xx) (λy.yy)
  = xx[x:=λy.yy]
  = (λy.yy) (λy.yy)　束縛変数の名前変更
  = (λx.xx) (λx.xx)　α-conversion

ちょっと変わったλ式1－ Ω

- λ式　Ω := *(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λx.xx) (λy.yy)
  = xx[x:=λy.yy]
  = (λy.yy) (λy.yy)
  = (λx.xx) (λx.xx)
  = Ω

ちょっと変わったλ式1－ Ω

- λ式　Ω := *(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λx.xx) (λy.yy)
  = xx[x:=λy.yy]
  = (λy.yy) (λy.yy)
  = (λx.xx) (λx.xx)
  = Ω

となって、同じλ式Ωが現れる。
これは、この計算が終わらないことを意味する。

ちょっと変わったλ式2 — Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

ちょっと変わったλ式2 ― Y combinator

- **Y** := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y **g**　　**Y** への **g**の適用
  = λg.(λx.g (x x)) (λx.g (x x)) **g**

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λ**g**.(λx.**g** (x x)) (λx.**g** (x x)) g 束縛変数名の変更
  = λ**f**.(λx.**f** (x x)) (λx.**f** (x x)) g　　α-conversion

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λ**f**. (λx.f (x x)) (λx.f (x x)) **g**　　gの適用
  = (λx.f (x x)) (λx.f (x x)) [**f**:=**g**]
  = (λx.**g** (x x)) (λx.g (x x))

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf. (λx.f (x x)) (λx.f (x x)) g
  = (λx.f (x x)) (λx.f (x x)) [f:=g]
  = (λ**x**.g (**x** **x**)) (λx.g (x x))　束縛変数名の変更
  = (λ**y**.g (**y** **y**)) (λx.g (x x))　α-conversion

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf. (λx.f (x x)) (λx.f (x x)) g
  = (λx.f (x x)) (λx.f (x x)) [f:=g]
  = (λx.g (x x)) (λx.g (x x))
  = (λ**y**. g (y y)) (λx.g (x x))　　適用
  = g (y y)[**y**:=λx.g (x x)]　　　β-reduction
  = g ((λx.g (x x)) (λx.g (x x)))
  = g (Y g)

ちょっと変わったλ式2 ― Y combinator

- Y := λg. (λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf. (λx.f (x x)) (λx.f (x x)) g
  = (λx.f (x x)) (λx.f (x x)) [f:=g]
  = (λx.g (x x)) (λx.g (x x))
  = (λy. g (y y)) (λx.g (x x))
  = g (y y)[y:=λx.g (x x)]
  = g ((λx.g (x x)) (λx.g (x x))) この青字の式をHとする
  = g (H)　Y g = λg.H g= H　η-conversion

ちょっと変わったλ式2 ― Y combinator

- Y := λg. (λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf. (λx.f (x x)) (λx.f (x x)) g
  = (λx.f (x x)) (λx.f (x x)) [f:=g]
  = (λx.g (x x)) (λx.g (x x))
  = (λy. g (y y)) (λx.g (x x))
  = g (y y)[y:=λx.g (x x)]
  = g ((λx.g (x x)) (λx.g (x x)))
  = g (H)
  = g (Y g)

ちょっと変わったλ式2 ― Y combinator

- Y := λg. (λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf. (λx.f (x x)) (λx.f (x x)) g
  = (λx.f (x x)) (λx.f (x x)) [f:=g]
  = (λx.g (x x)) (λx.g (x x))
  = (λy. g (y y)) (λx.g (x x))
  = g (y y)[y:=λx.g (x x)]
  = g ((λx.g (x x)) (λx.g (x x)))
  = g (H)
  = g (Y g)

すなわち、Y g = g (Y g)
となって、Y gは、gの不動点
を与える。

Part II

型付きラムダ計算

ラムダ計算への型の導入

ラムダ計算への型の導入

- 変数 x は型 τ を持つ。そのことを $x : \tau$ と表す。
- あるλ式 e が型 τ を持つことを、$e : \tau$ と表す。
- 型σから型τへの関数は、型 $\sigma \rightarrow \tau$ を持つ。
- アルファ変換、ベータ還元、エータ変換のα, β, ηの変換ルールは同じものとする。

ふりかえり:
λ式の形式的定義

- これまでは関数とその引数、引数に具体的な値が与えられた時の関数の値をベースに、λ式とその値の計算を説明してきたが、ここでは、λ式の形式的な定義を与えよう。(正確なものではなく、簡略化したものであることに留意)

1. 変数 x,...はλ式である。(variable)
2. Mがλ式で、xが変数であるなら、
   λxによる抽象化 **λx.M** はλ式である。(abstraction)
3. M, Nがλ式であるなら、
   MへのNの適用 **M N** は、λ式である。(application)

型付きλ式の形式的定義

型付きλ式は、形式的には、次のように定義される。

1. 変数(variable)
   単純な変数 v は型を持ち、型付きλ式である。

   $$v : \tau$$

2. 抽象化(abstraction)
   型付きλ式 e : τ を型を持つ変数 x : σ で抽象化した λx.eは型付きλ式である。その型は、 σ → τ である。

   $$(\lambda x_\sigma . e_\tau) : (\ \sigma \to \tau\ )$$

   (λx:σ . e:τ) : (σ → τ)
   と書いてもいい

3. 適用(application)
   型付きλ式 e_1 を型付きλ式e_2 に適用した $(e_1\ e_2)$ は、型付きλ式である。その型は、次のようになる。

   $e_1 : (\ \sigma \to \tau\ )$で、$e_2 : \sigma$ なら、$(e_1\ e_2) : \tau$

単純な変数 v は型を持つ。

v ： T

項 ： 型

単純な変数 v は型を持つ。

v : T

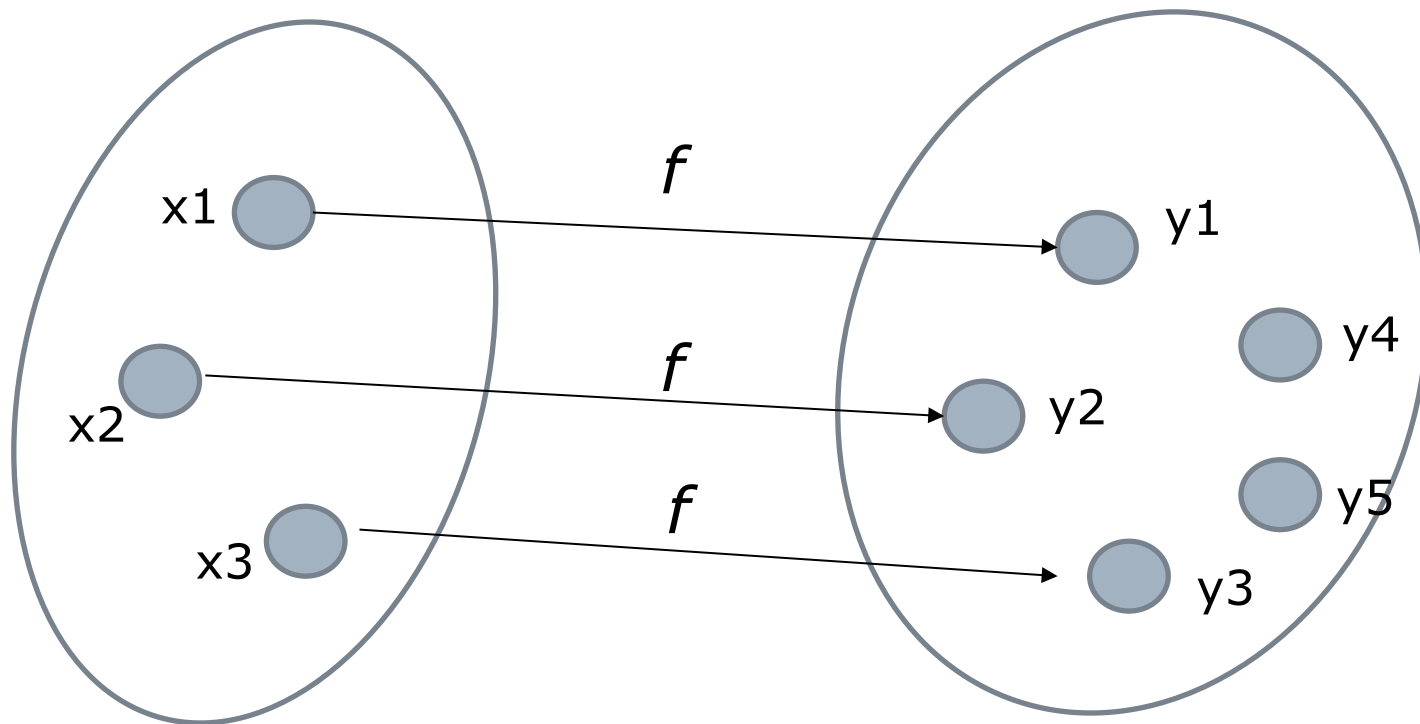
項 : 型

項 v は、型 T を持つ。
型 T は、項 v を含む。

関数 y=f(x) の定義域と値域

変数 x は定義域の集合S={x1,x2,x3}の要素に値を取り、
変数 y は値域の集合T={y1,y2,y3,y4,y5}の要素に値をとる。

入力変数:x ∈ S　　出力変数: y ∈ T

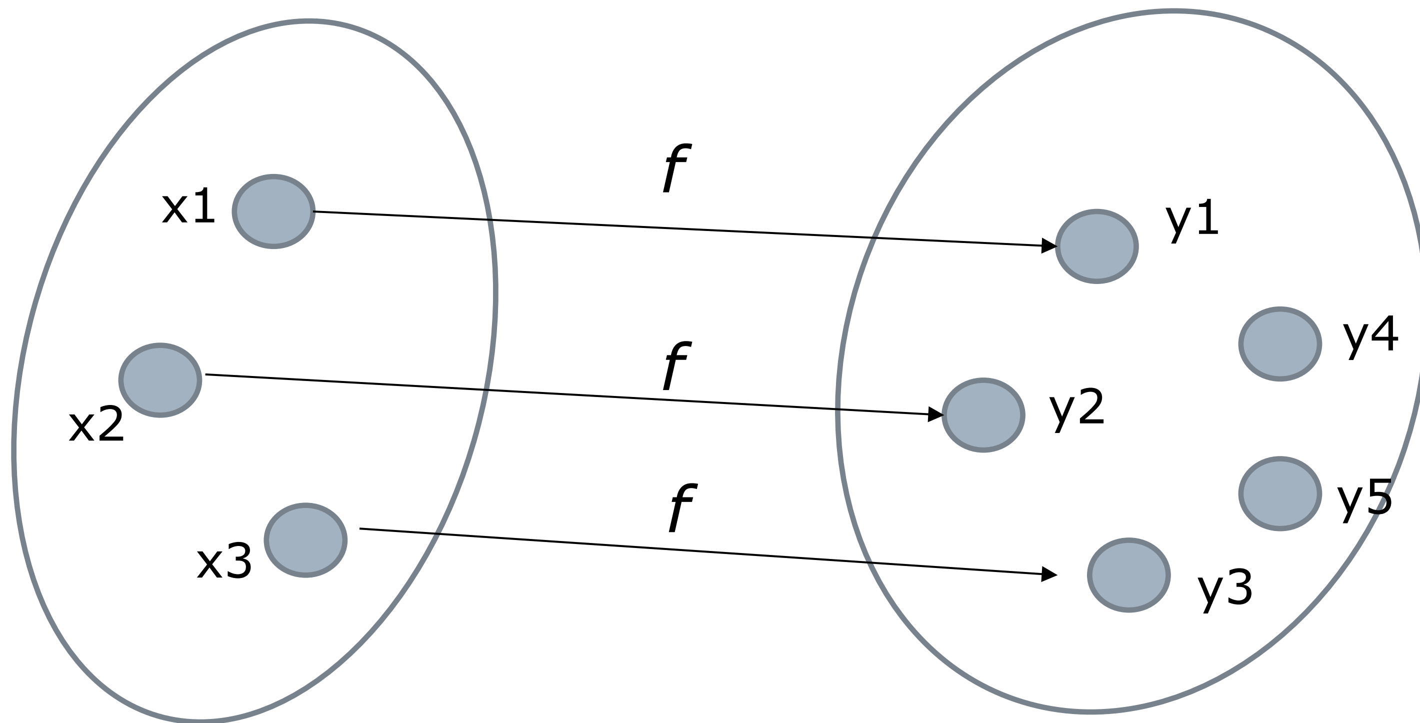


定義域 S　　値域 **T**

集合から型へ

入力変数 x は型 σ の項 x1,x2,x3 に値を取り、
出力変数 y は型 τ の項 y1,y2,y3,y4,y5に値をとる。

入力変数 x : σ　　　　出力変数 y : τ



型 σ　　　　型 τ

集合と型の対応　'∈'を ':'に置き換えれば

集合と型は、ほぼ、同じものと思っていい

要素 ∈ 集合　　　　項 : 型

集合 S
- $x1 \in S$
- $x2 \in S$
- $x3 \in S$

型 σ
- $x1 : \sigma$
- $x2 : \sigma$
- $x3 : \sigma$

集合 T
- $y1 \in T$
- $y2 \in T$
- $y3 \in T$
- $y4 \in T$
- $y5 \in T$

型 τ
- $y1 : \tau$
- $y2 : \tau$
- $y3 : \tau$
- $y4 : \tau$
- $y5 : \tau$

抽象化 λx. eは型を持つ

x : σ で e : τ なら、

$(\lambda x_\sigma.\ e_\tau) : (\sigma \rightarrow \tau)$

x : σ　e : τ
を表す

抽象化 λx. eは型を持つ

x : σ で e : τ なら、

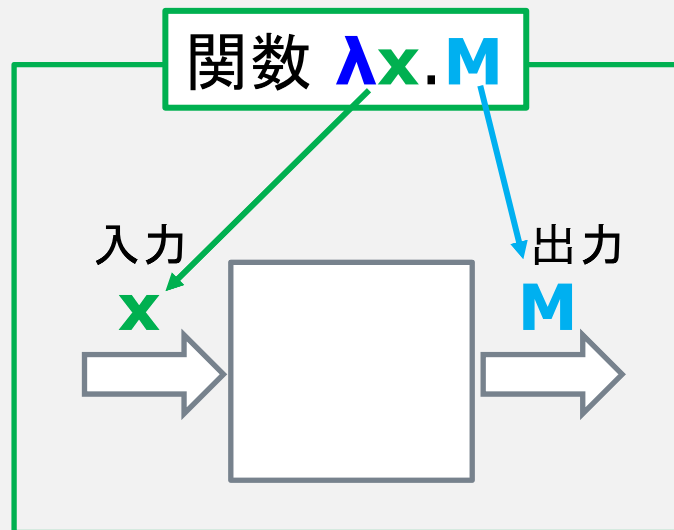
$$(\lambda x_\sigma.\ e_\tau) : (\sigma \rightarrow \tau)$$

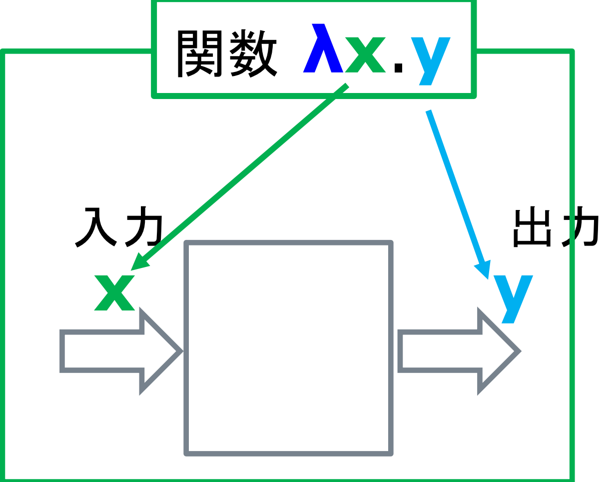
項　　　　　　　型

項 ラムダ式 $\lambda x_\sigma.\ e_\tau$ は型 $\sigma \rightarrow \tau$ を持つ

ふりかえり　抽象化 λx.M があらわす関数

- 入力となる変数をx, 出力となる式(xとかx+1とかx^2とか)をM で表すと、λによる抽象化 λx.M は、次の関数を表す。





λx.y

入力 x : σ

x1
x2
x3

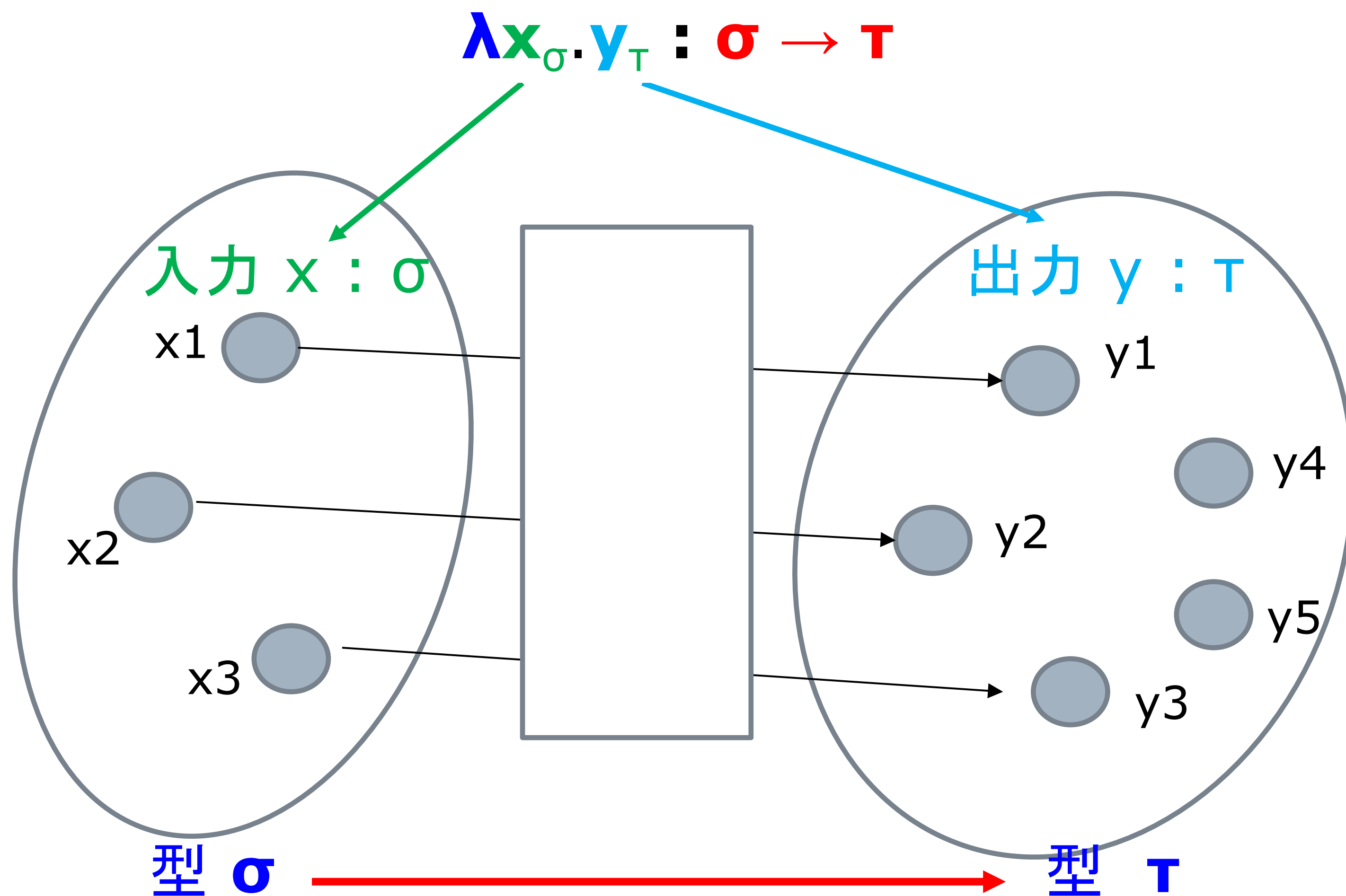
出力 y : τ

y1
y4
y2
y5
y3

型 σ

型 τ

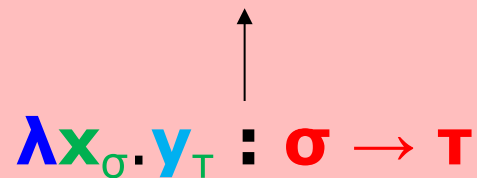
型σから型τへの関数は、
型 $\sigma \to \tau$ を持つ。



適用 e_1 e_2は型を持つ

e_1 : ($\sigma \to \tau$)で、e_2 : σ なら、

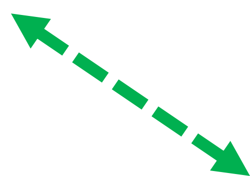
$(e_1\ e_2)$: τ



この例は、次のように考えるとわかりやすい

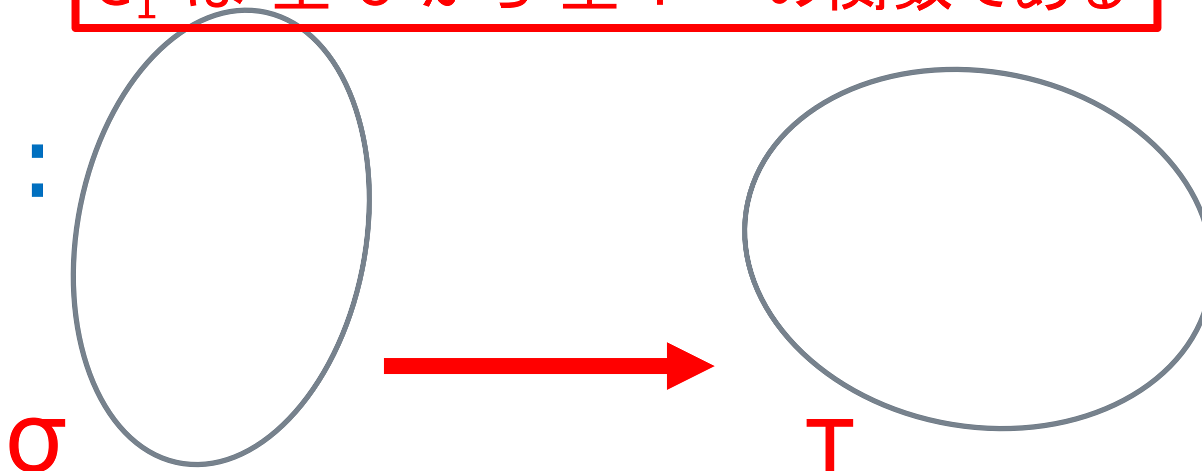
「適用」は型を持つ

$e_1 : (\sigma \rightarrow \tau)$で、



e_1 は 型 σ から 型 τ への関数である

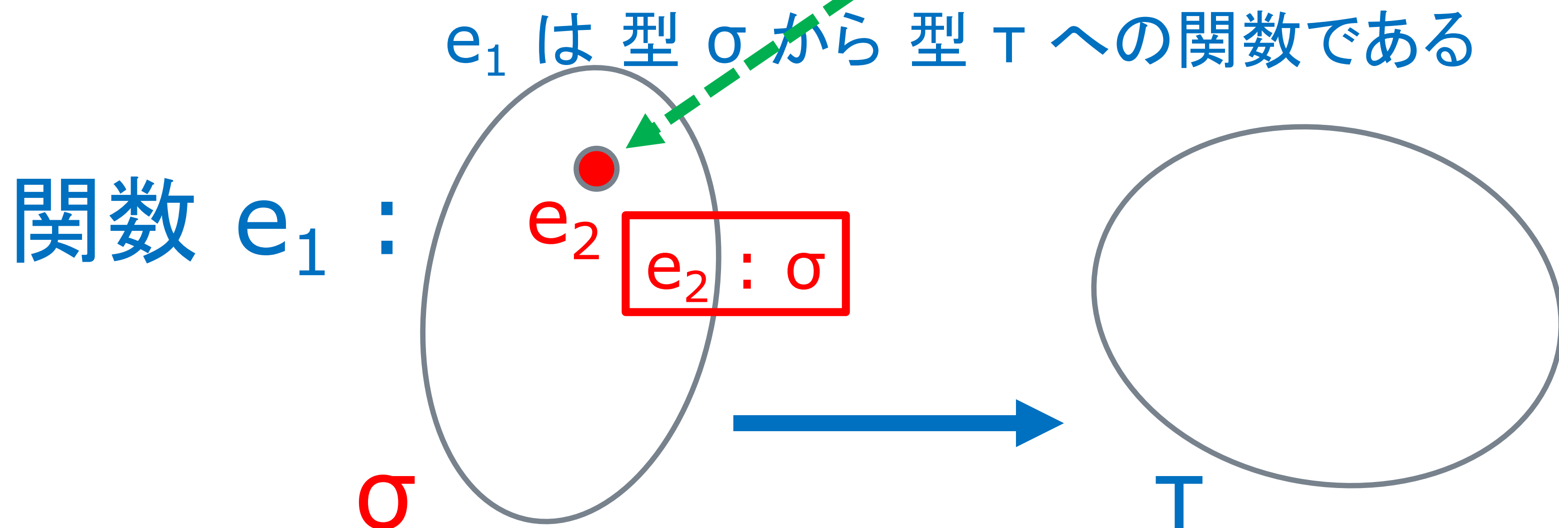
関数 e_1 :



この例は、次のようにに考えるとわかりやすい

「適用」は型を持つ

e_1 : (σ → τ)で、e_2 : σ なら、

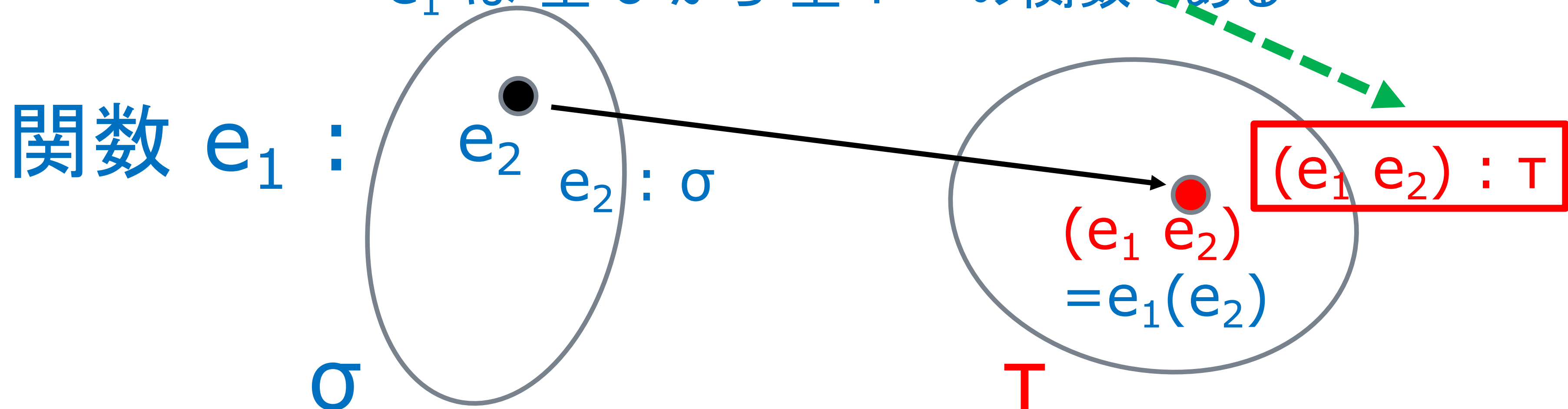


この例は、次のように考えるとわかりやすい

「適用」は型を持つ

$e_1 : (\sigma \to \tau)$で、$e_2 : \sigma$ なら、

$(e_1\ e_2) : \tau$



型付きラムダでの計算

型のないラムダ計算と 型を持つラムダ計算の違い

- 先に見た型のないλ計算で成り立つ性質の大部分は、型を持つλ計算でも成り立つ。
- ただ、両者のあいだには違いもある。
  型のないλ計算では、任意のλ式に対して任意のλ式の適用を許していたが、型を持つλ計算では、λ式の適用に型による制限が入っている。

型のないラムダ計算と
型を持つラムダ計算の違い

- 先に見た型のないλ計算で成り立つ性質の大部分は、型を持つλ計算でも成り立つ。
- ただ、両者のあいだには違いもある。
  型のないλ計算では、任意のλ式に対して任意のλ式の適用を許していたが、型を持つλ計算では、λ式の適用に型による制限が入っている。
- 型を持つラムダ計算では、同じ型を持つλ式同士の
  $S_\sigma T_\sigma$ という適用は許されない。

型のないラムダ計算と型を持つラムダ計算の違い

- 先に見た型のないλ計算で成り立つ性質の大部分は、型を持つλ計算でも成り立つ。
- ただ、両者のあいだには違いもある。
  型のないλ計算では、任意のλ式に対して任意のλ式の適用を許していたが、型を持つλ計算では、λ式の適用に型による制限が入っている。
- 型を持つラムダ計算では、同じ型を持つλ式同士の$S_{\sigma}T_{\sigma}$という適用は許されない。
- 許されるのは、$S_{\sigma\rightarrow\tau}T_{\sigma}$という型を持つλ式どうしの適用のみである。

Sは$\sigma\rightarrow\tau$型、Tはσ型

型のないラムダ計算と
型を持つラムダ計算の違い

- 先に見た型のないλ計算で成り立つ性質の大部分は、型を持つλ計算でも成り立つ。
- ただ、両者のあいだには違いもある。
  型のないλ計算では、任意のλ式に対して任意のλ式の適用を許していたが、型を持つλ計算では、λ式の適用に型による制限が入っている。
- 型を持つラムダ計算では、同じ型を持つλ式同士の $S_{\sigma}T_{\sigma}$という適用は許されない。
- 許されるのは、$S_{\sigma\rightarrow\tau}$ T_{σ}という型を持つλ式どうしの適用のみである。
- 先の Ω := *(λx.xx) (λx.xx)* は、同じ型を持つλ式同士の適用なので型を持つラムダ計算では許されないλ式である。

「適用」は型で制限される。可能なのは、

S : (σ → τ)で、T : σ の場合で、
(S T) : τ の場合のみ。

これは、次のようにも表現できる

$S_{\sigma \to \tau} T_{\sigma} : \tau$

単純な型を持つラムダ計算の特徴

- 単純な型を持つラムダ計算は、こうした点では、型を持たないラムダ計算より表現力が弱いにもかかわらず、次のような重要な特徴を持つ。

- 単純な型を持つラムダ計算では、変換による計算は、必ず停止する。

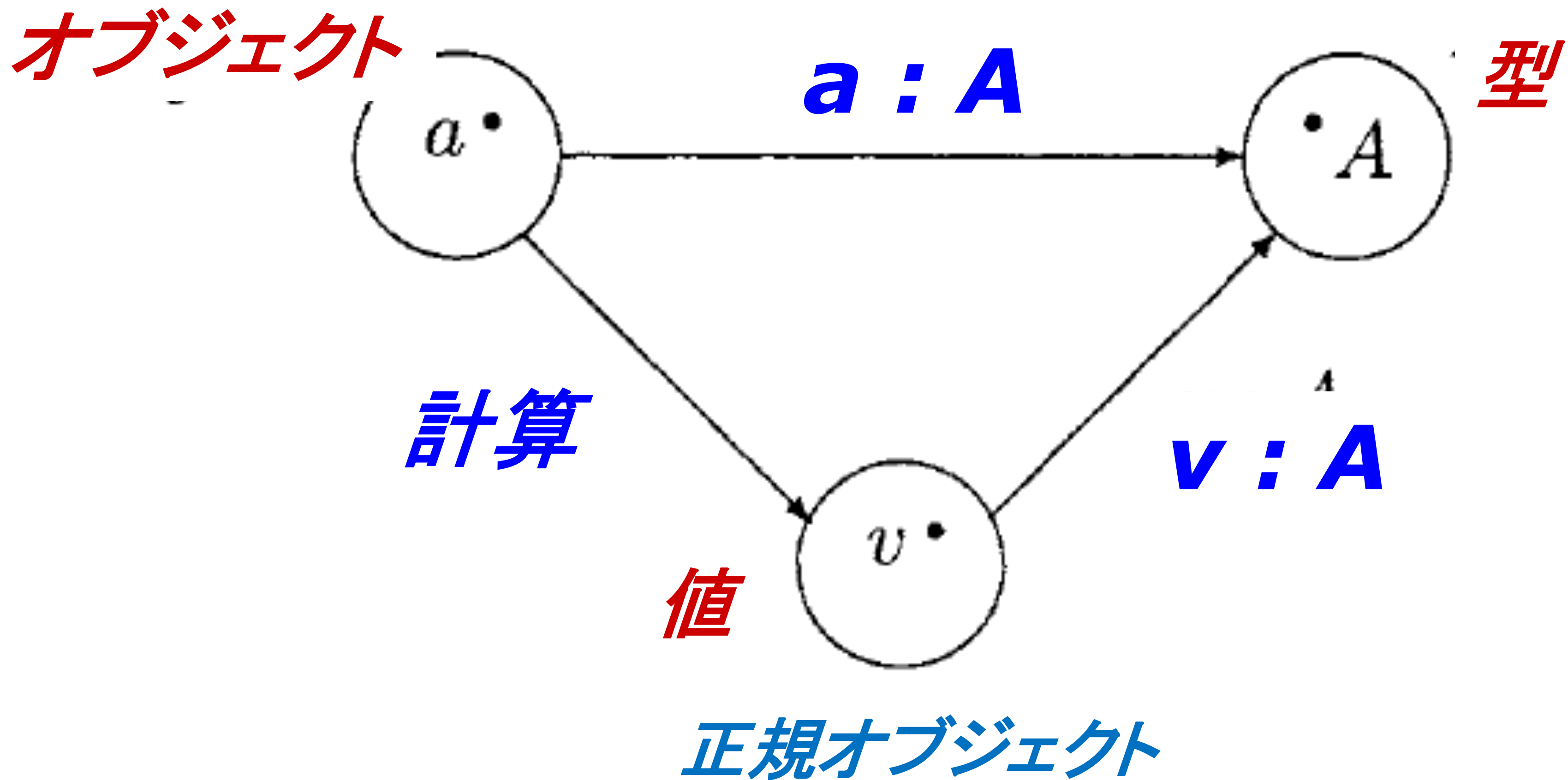
型を持つラムダ計算での計算

- あるオブジェクト aがある型 Aを持つという判断を、 ***a : A*** と表わそう。
- 型を持つラムダ計算の理論では、計算は、次の形をしている。
- 関数 ***λ x:A. b[x]*** を、型Aに属するオブジェクトaを適用して ***b[a]***を得る。
- 計算の下で、全てのオブジェクトはユニークな値を持つ。また、計算で等しいオブジェクトは、同一の値を持つ。

型を持つラムダ計算での
正規オブジェクトとその値

- ある型に属するオブジェクトは、型の計算ルールによって、値が求まるなら、正規オブジェクトと呼ばれる。
- 例えば、1 + 1 は、自然数の型に属するが正規ではない。2 は、正規のオブジェクトである。
- ある型 A の正規オブジェクト v は、それ以上計算ルールの適用が出来ず、それ自身を値として持つ。
この時、***v : A*** と書く。

型、オブジェクト、値



型付きラムダ式と証明の解釈

型付きラムダ式と命題 a → bの証明

- Kolmogorovは、命題 a → bの証明に、「命題 aの証明を命題 bの証明に変換する方法を構築すること」という解釈を与えた。（彼の立場は「構成主義」と呼ばれる）
- このことは、命題a → bの証明を、命題aの証明から命題bの証明への関数と見ることが出来るということを意味する。
- そして、この関数は、型付きラムダ式として、 a → bという型を持つだろう。

- このように、ある命題の証明が、もっと基本的な他の証明からどのように構成されるか考えてみよう。

命題の証明は、何から構成されるか？

$\bot$	なし
$A \land B$	Aの証明とBの証明の両方
$A \lor B$	Aの証明、あるいは、Bの証明
$A \to B$	Aの証明からBの証明を導く方法
$(\forall x)B(x)$	任意のaに対してB(a)の証明を与える方法
$(\exists x)B(x)$	あるaに対するB(a)の証明

命題の証明は、何から構成されるか？
形式的に

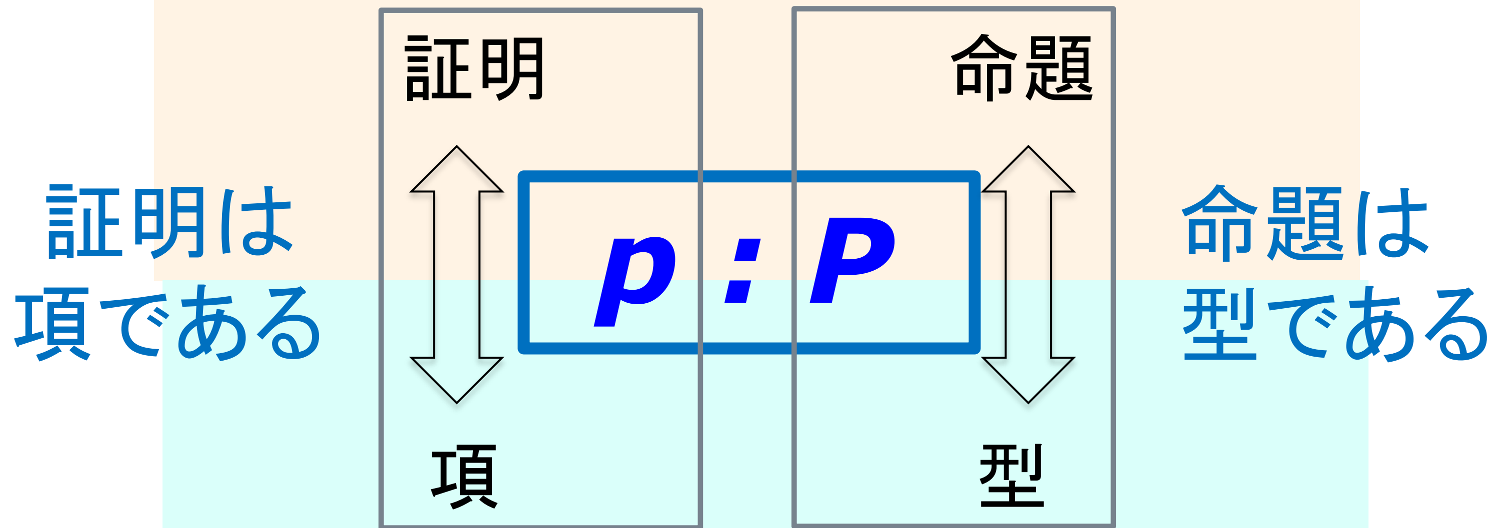
- $\bot$ none
- $A \wedge B$ Aの証明であるaと、Bの証明であるbのペア**(a,b)**
- $A \vee B$ Aの証明である**i(a)**、あるいは、Bの証明である**j(b)**
- $A \rightarrow B$ Aの証明であるaに対して、Bの証明b(a)を与える関数**(λx)b(x)**
- $(\forall x)B(x)$ 任意のaに対してBの証明b(a)を与える関数**(λx)b(x)**
- $(\exists x)B(x)$ あるaと、B(a)の証明であるbのペア**(a,b)**

証明の解釈と型概念の拡大

- このことは、命題をその型と同一視出来ることを示唆するものだ。しかし、残念ながら、「型付きラムダ計算」で定義される「型」は、関数の型でしかないのだ。一般の命題への型概念の拡大は、「従属型の理論」の登場によって行われることになる。

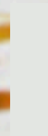
- そこでは、ある命題の型はその命題の証明の集まりを表現し、そうした証明の一つは、その命題が対応する型の、一つの項と見なすことが出来る。
- こうした関係を「Curry-Howard対応」と呼ぶ。
  それについては、今回のセミナーでは、触れることができなかった。

Curry-Howard対応
「型」と「命題」、「項」と「証明」との「双対性」



論理学入門 II

ラムダ計算と関数型言語
Part III, Part IV



Agenda ラムダ計算と関数型言語

Part III ラムダ計算とLISP,Haskell,OCaml

- LISP
- 関数型プログラミングの発展
- Haskell
- Ocaml

Part IV Coqでの関数型プログラミング

- Coqでの関数定義
- ListをCoqでプログラムする

Part III
ラムダ計算と
LISP,Haskell,OCaml

LISP

"Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I"

John McCarthy, 1960

http://www-formal.stanford.edu/jmc/recursive.pdf

LISPの誕生

- LISPは、John McCarthyによって、チャーチの型のないラムダ計算の、ある意味、忠実な実装として 1958年に生まれた。McCarthyによる、最初の実装の基本的な文献は、次の二つである。

- “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”
John McCarthy, 1960
http://www-formal.stanford.edu/jmc/recursive.pdf
- “LISP-1.5-Programmers-Manual.pdf”
John McCarthy et al, 1962
https://github.com/informatimago/lisp-1-5/blob/master/LISP-1.5-Programmers-Manual.pdf

LISP
S式とリスト

S式 (Symbolic Expression)

- LISPで、最も基本的なコンセプトは「S式」である。S式は、次のように帰納的に定義される。

1. Atom (名前を示す文字列) は、S式である。
2. 二つのS式 A, Bから構成されるペアは、S式である。これを(A . B)と表す。

- S式の例

```
ATOM
(A . B)
(A . (B . C))
((A1 . A2) . B)
((U . V) . (X . Y))
((U . V) . (X . (Y . Z)))
```

S式に関係する関数 cons, car, cdr

- **cons** 二つのS式から新しいS式を作る

cons[A;B]=(A . B)
cons[(A . B);C]=((A . B) . C)
cons[cons[A;B];C]=((A . B) . C)

- **car** S式の左の要素を取り出す
- **cdr** S式の左の要素を取り出す

car[(A . B)]=A
car[(A . (B1 . B2))]=A
car[((A1 . A2) . B)]=(A1 . A2)
car[A] is undefined

cdr[(A . B)]=B
cdr[(A . (B1 . B2))]=(B1 . B2)
cdr[((A1 . A2) . B)]=B
cdr[A] is undefined

関数 cons, car, cdr と
関数 eq, atom のコードサンプル

```
car[cdr[(A . (B1 . B2))]]=B1
car[cdr[(A . B)]] is undefined
car[cons[A;B]]=A
```

```
car[cons[x;y]]=x
cdr[cons[x;y]]=y
cons[car[x];cdr[x]]=x
```

```
eq[A;A]=T
eq[A;B]=F
eq[A;(A . B)] is undefined
eq[(A . B);(A . B)] is undefined
```

```
atom[EXTRALONGSTRINGOFLETTERS]=T
atom[(U . V)]=F
atom[car[(U . V)]]=T
```

List Notation

```
(A B C)=(A . (B . (C . NIL)))
((A B) C)=((A . (B . NIL)) . (C . NIL))
(A B (C D))=(A . (B . ((C . (D . NIL)) . NIL)))
(A)=(A . NIL)
((A))=((A . NIL) . NIL)
(A (B . C))=(A . ((B . C) . NIL))
```

```
car[(A B C)]=A
cdr[(A B C)]=(B C)
cons[A; (B C)]=(A B C)
car[((A B) C)]=(A B)
cdr[(A)]=NIL
car[cdr[(A B C)]]=B
```

```
cadr[(A B C)]=car[cdr[(A B C)]=B
caddr[(A B C)]=C
cadadr[(A (B C) D)]=C
```

型のないラムダ計算とLISP

チャーチのラムダ記法と
マッカーシーのラムダ記法とS式

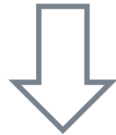
- チャーチのラムダ記法とマッカーシーのラムダ記法には、若干の違いがあるが、本質的には同じものである。
  LISPでは、さらにそれがS式に変形される。

- ラムダによる抽象化
  $\lambda(x, y).(y^2 + x)$ チャーチ流
  $\lambda[[x; y]\ (y^2 + x)]$ マッカーシー流
  ((LAMBDA (X Y) (PLUS (SQUARE Y) X)) S式

- ラムダ式への値の適用
  $\lambda(x, y).(y^2 + x)\ 3\ 4 = 19$ チャーチ流
  $\lambda[[x; y]\ (y^2 + x)][3;4] = 19$ マッカーシー流
  (((LAMBDA (X Y) (PLUS (SQUARE Y) X)) (3 4)) ➔19

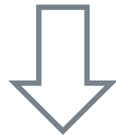
ラムダ式への値の適用
関数 apply と eval

apply[(LAMBDA (X Y) (CONS X Y)); (A B);NIL]



apply は、変数に値をバインドして、a-list を作り、
関数と a-list を eval に渡す。

eval[(CONS X Y); ((X . A) (Y . B))]



eval は、変数の値を評価して、関数に渡す

cons[A;B] = (A . B)

ふりかえり　λ計算の形式的ルール

- 次の三つのルールを利用して、λ式を(基本的には単純なものに)変換することをλ計算という。

1. **α-conversion:**
   抽象化に用いる変数の名前は、自由に変更出来る。
   例えば、 $\lambda x.(x^2+1) \Rightarrow \lambda y.(y^2+1) \Rightarrow \lambda z.(z^2+1)$
2. **β-reduction:**
   代入による計算ルール
   (λx.M) a => M[x:=a]
3. **η-conversion:**
   λx.(f x) => f
   xで抽象化されたf(x)は、fに等しいということ。

LISPでのλ計算の形式的ルールの表現

1. **α-conversion:**
   抽象化に用いる変数の名前は、自由に変更出来る。
   例えば、$\lambda x.(x^2+1) \Rightarrow \lambda y.(y^2+1) \Rightarrow \lambda z.(z^2+1)$

   (LAMBDA <vars> <body>)
   = (LAMBDA <newvars>
   Subst[<newvars> <vars> <body>])

2. **β-reduction:**
   代入による計算ルール
   $(\lambda x.M)\ a \Rightarrow M[x:=a]$

   ((LAMBDA <vars> <body>) <args>)
   = **Subst**[<args> <vars> <body>]

条件式とlabel

条件式

- マッカーシーは、$[p_1 \to e_1; \cdot\cdot\cdot ; p_n \to e_n]$ という形の「条件式」を導入する。この式は、p_iが真の時、e_iを返す。
  p_i に、「常に真」を表す記号 T を使うことができる。

- 条件式の例
  [atom[x] → x ; T → ff[car [x]]]
- そのS式
  (**COND** ((ATOM X) X)
  ((QUOTE T) (FF (CAR X))))

label

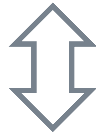
- マッカーシーは、ラムダによる抽象化で得られる無名の関数に名前をつけるために、labelという記法を導入する。これは、再帰的に関数を定義するのに必要である。

- label の例
  label[ff; λ[[x]; [atom[x] → x ; T → ff[car [x]]]]]
- そのS式

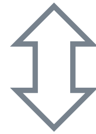
```
(LABEL FF (LAMBDA (X)
   (COND  ((ATOM X) X)
          ((QUOTE T) (FF (CAR X))))))
```

labelと **η-conversion**:

label[ff;λ[[x];[atom[x]→x; T→ff[car[x]]]]]



ff=λ[[x];[atom[x]→x; T→ff[car[x]]]]



η-conversion:
λx.(f x) => f

ff[x]=[atom[x]→x; T→ff[car[x]]].

LISPプログラミング

階乗関数 FACT の再帰的定義

階乗の定義

$$n! = [n=0 \to 1;\ T \to n \cdot [n-1]!]$$

```
(DEFINE FACT
(LAMBDA (N) (IF (= N 0) 1
      (* N (FACT (- N 1))))))
```

Recursive programs

```
(DEFINE FACT
(LAMBDA (N) (IF (= N 0) 1
      (* N (FACT (- N 1))))))
```

```
(FACT 3)
=> (IF (- 3 0) 1 (* 3 (FACT (- 3 1))))
=> (* 3 (FACT (- 3 1)))
=> (* 3 (FACT 2))
=> (* 3 (IF (= 2 0) 1 (* 2 (FACT (- 2 1)))))
=> (* 3 (* 2 (FACT (- 2 1))))
=> (* 3 (* 2 (FACT 1)))
=> (* 3 (* 2 (IF (= 1 0) 1 (* 1 (FACT (- 1 1))))))
=> (* 3 (* 2 (* 1 (FACT (- 1 1)))))
=> (* 3 (* 2 (* 1 (FACT 0))))
=> (* 3 (* 2 (* 1 (IF (= 0 0) 1 (* 0 (FACT (- 0 1)))))))
=> (* 3 (* 2 (* 1 1)))
=> (* 3 (* 2 1))
=> (* 3 2)
=> 6
```

階乗関数 FACT の繰り返しによる定義

```
(DEFINE FACT
   (LAMBDA (N)
       (LABELS ((FACT1
                 (LAMBDA (M ANS)
                     (IF (= M 0) ANS
                         (FACT1 (- M 1) (* M ANS))))))
        (FACT1 N 1))))
```

Iteration

```
(DEFINE FACT
  (LAMBDA (N)
    (LABELS ((FACT1
              (LAMBDA (M ANS)
                (IF (= M 0) ANS
                    (FACT1 (- M 1) (* M ANS))))))
          (FACT1 N 1))))
```

```
(FACT 3)
⇒ (FACT1 3 1)
⇒ (IF (= 3 0) 1 (FACT1 (- 3 1) (* 3 1)))
⇒ (FACT1 (- 3 1) (* 3 1))
⇒ (FACT1 2 (* 3 1))
⇒ (FACT1 2 3)
⇒ (IF (= 2 0) 3 (FACT1 (- 2 1) (* 2 3)))
⇒ (FACT1 (- 2 1) (* 2 3))
⇒ (FACT1 1 (* 2 3))
⇒ (FACT1 1 6)
⇒ (IF (= 1 0) 6 (FACT1 (- 1 1) (* 1 6)))
⇒ (FACT1 (- 1 1) (* 1 6))
⇒ (FACT1 0 (* 1 6))
⇒ (FACT1 0 6)
⇒ (IF (= 0 0) 6 (FACT1 (- 0 1) (* 0 6)))
⇒ 6
```

evalquote / apply
関数への値の適用

関数への値の適用

関数 evalquote / apply

f: λ[[x;y];cons[car[x];y]]

fn: (LAMBDA (X Y) (CONS (CAR X) Y))

arg_1: (A B)

arg_2: (C D)

args: ((A B) (C D))

evalquote[(LAMBDA (X Y) (CONS (CAR X) Y)); ((A B) (C D))] =

λ[[x;y];cons[car[x];y]][(A B);(C D)]=

(A C D)

evalquote[fn;x] = apply[fn;x;NIL]

apply 関数への値の適用

```
apply[fn;x;a] =
      [atom[fn] → [eq[fn;CAR] → caar[x];
                   eq[fn;CDR] → cdar[x];
                   eq[fn;CONS] → cons[car[x];cadr[x]];
                   eq[fn;ATOM] → atom[car[x]];
                   eq[fn;EQ] → eq[car[x];cadr[x]];
                   T → apply[eval[fn;a];x;a]];
      eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
      eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
                                                    caddr[fn]];a]]]
```

```
pairlis[x;y;a] = [null[x]→a;T→cons[cons[car[x];car[y]];
                   pairlis[cdr[x];cdr[y];a]]]

pairlis[(A B C);(U V W);((D . X) (E . Y))] =
                             ((A . U) (B . V) (C . W) (D . X) (E . Y))


assoc[x;a] = [equal[caar[a];x]→car[a];T→assoc[x;cdr[a]]]

assoc[B;((A . (M N)), (B . (CAR X)), (C . (QUOTE M)), (C . (CDR X)))]
= (B . (CAR X))
```

eval

```
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
        atom[car[e]]→
                [eq[car[e],QUOTE] → cadr[e];
                eq[car[e];COND] → evcon[cdr[e];a];
                T → apply[car[e];evlis[cdr[e];a];a]];
      T → apply[car[e];evlis[cdr[e];a];a]]

evcon[c;a] = [eval[caar[c];a] → eval[cadar[c];a];
                T → evcon[cdr[c];a]]

evlis[m;a] = [null[m] → NIL;
               T → cons[eval[car[m];a];evlis[cdr[m];a]]]
```

関数型プログラミングの発展

A History of Haskell: Being Lazy With Class

Paul Hudak et al. 2007

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf

1975～1980

LISPの進化（”Lambda Paper”）

Steele and Sussman

The Original 'Lambda Papers' by Guy Steele and Gerald Sussman

- Gerald Jay Sussman and Guy Lewis Steele, Jr.. "Scheme: An Interpreter for Extended Lambda Calculus". MIT AI Lab. AI Lab Memo AIM-349. December 1975. Available online: ps pdf.
- Guy Lewis Steele, Jr. and Gerald Jay Sussman. "Lambda: The Ultimate Imperative". MIT AI Lab. AI Lab Memo AIM-353. March 1976. Available online: ps pdf.
- Guy Lewis Steele, Jr.. "Lambda: The Ultimate Declarative". MIT AI Lab. AI Lab Memo AIM-379. November 1976. Available online: ps pdf.
- Guy Lewis Steele, Jr.. "Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO". MIT AI Lab. AI Lab Memo AIM-443. October 1977. Available online: ps pdf.
- Guy Lewis Steele, Jr. and Gerald Jay Sussman. "The Art of the Interpreter of, the Modularity Complex (Parts Zero, One, and Two)". MIT AI Lab. AI Lab Memo AIM-453. May 1978. Available online: ps pdf.
- Guy Lewis Steele, Jr.. "RABBIT: A Compiler for SCHEME". Masters Thesis. MIT AI Lab. AI Lab Technical Report AITR-474. May 1978. Available online: ps pdf.
- Guy Lewis Steele, Jr. and Gerald Jay Sussman. "Design of LISP-based Processors, or SCHEME: A Dielectric LISP, or Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode". MIT AI Lab. AI Lab Memo AIM-514. March 1979. Available online: ps pdf.
- Guy Lewis Steele, Jr.. "Compiler Optimization Based on Viewing LAMBDA as RENAME + GOTO". *AI: An MIT Perspective*. 1980.
- Guy Lewis Steele, Jr.. "Debunking the "Expensive Procedure Call" Myth, or Procedure Call Implementations Considered Harmful, or LAMBDA, the Ultimate GOTO". *ACM Conference Proceedings*. 1977. Available online: ACM Digital Library.
- Guy Lewis Steele, Jr. and Gerald Jay Sussman. "Design of a Lisp-based Processor". *CACM*. 23. 11. November 1980. Available online: ACM Digital Library.

https://web.archive.org/web/20160510140804/http://library.readscheme.org/page1.html

1978 John Backus チューリング賞受賞講演

“Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs”

- Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak … their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

https://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf

1978 Robin Milner **ML**

"A Theory of Type Polymorphism in Programming"

- The aim of this work is largely a practical one. A widely employed style of programming, paticularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures which work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists).

https://homepages.inf.ed.ac.uk/wadler/papers/papers-we-love/milner-type-polymorphism.pdf

"A proposal for Standard ML" 1982

https://dl.acm.org/doi/10.1145/800055.802035

1981 FPCA 最初のカンファレンス

Functional Programming Languages and Computer Architecture (FPCA)

1981 D.A.Turner **lazy evaluation**

"The semantic elegance of applicative languages"

- Very briefly it can be summarised as (non-strict, higher order) recursion equations + set abstraction.

https://dl.acm.org/doi/10.1145/800223.806766

"**Miranda**: a non-strict functional language with polymorphic types" **1985**

https://dl.acm.org/doi/10.5555/5280.5281

non-von Neumann hardware architectures

- Id project at MIT (Arvind and Nikhil, 1987)
- Rediflow project at Utah (Keller et al., 1979)
- SK combinator machine SKIM at Cambridge (Stoye et al., 1984)
- Manchester dataflow machine (Watson and Gurd, 1982)
- ALICE parallel reduction machine at Imperial (Darlington and Reeve, 1981)
- Burroughs NORMA combinator machine (Scheevel, 1986)
- DDM dataflow machine at Utah (Davis, 1977).

Scheme, ML の標準化の進行

- The Scheme community had major loci in MIT, Indiana, and Yale, and had just issued its ‘revised revised’ report (Rees and Clinger, 1986) (subsequent revisions would lead to the ‘revised5 ’ report (Kelsey et al., 1998)).
- Robin Milner had issued a ‘proposal for Standard ML’ (Milner, 1984) (which would later evolve into the definitive Definition of Standard ML (Milner and Tofte, 1990; Milner et al., 1997)), and Appel and MacQueen had released a new high-quality compiler for it (Appel and MacQueen, 1987).

1987 Caml 最初の実装

- “Caml” was originally an acronym for *Categorical Abstract Machine Language*. It was a pun on CAM, the Categorical Abstract Machine, and ML, the family of programming languages to which Caml belongs.

A History of OCaml
https://ocaml.org/learn/history.html

1987 Haskell

- By 1987, the situation was akin to a supercooled solution—all that was needed was a random event to precipitate crystallisation. That event happened in the fall of ’87, when Peyton Jones stopped at Yale to see Hudak on his way to the 1987 Functional Programming and Computer Architecture Conference (FPCA) in Portland, Oregon.
- After discussing the situation, Peyton Jones and Hudak decided to initiate a meeting during FPCA, to garner interest in designing a new, common functional language. Wadler also stopped at Yale on the way to FPCA, and also endorsed the idea of a meeting.

1996 Objective Caml

- first released in 1996 and renamed to OCaml in 2011. Objective Caml was the first language to combine the full power of object-oriented programming with ML-style static typing and type inference.
- It supports many advanced OO programming idioms (type-parametric classes, binary methods, mytype specialization) in a statically type-safe way, while these idioms cause unsoundness or require run-time type checks in other OO languages such as C++ and Java.

A History of OCaml
https://ocaml.org/learn/history.html

Haskell

A History of Haskell: Being Lazy With Class

Paul Hudak et al. 2007

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf

Haskell
Goals, principles, and processes

1. Haskell is lazy
2. Haskell is pure
3. Haskell has type classes
4. Haskell has no formal semantics
5. Haskell is a committee language
6. Haskell is a big language

型付きラムダ計算とHaskell

チャーチのラムダ記法とHaskellのラムダ記法

- チャーチのラムダ記法とHaskellのラムダ記法は、若干の違いがあるが(`λ’が` \’に、ラムダ変数と関数本体を区切るドット`.’が`->’に変わっている)、本質的には同じものである。Haskellでは、`.’ は関数の結合に用いられる。

- ラムダによる抽象化
  $\lambda x.(\lambda y.(y^2 + x))$　　チャーチ流
  `\x -> \y -> (y * y + x)`　　Haskell流

- ラムダ式への値の適用
  $\lambda x.(\lambda y.(y^2 + x))\ 3\ 4 = 19$　　チャーチ流
  `(\x -> \y -> (y * y + x)) 3 4`　　Haskell流

チャーチのラムダ記法とHaskellのラムダ記法

- ラムダによる抽象化

  $\lambda(x,y).(y^2 + x)$ チャーチ流

  `\(x,y) -> (y * y + x)` Haskell流

- ラムダ式への値の適用

  $\lambda(x,y).(y^2 + x))(3,4) = 19$ チャーチ流

  `(\(x,y) -> (y * y + x))(3,4) = 19` Haskell流

Curry化

二つの引数をとる関数の二つの定義の例

- hyp :: **(Float, Float) -> Float**

  hyp (x,y) = sqrt (x*x + y*y)

  hyp (3,4) = sqrt (9 + 16) = sqrt 25 = …

- hyp :: **Float -> Float -> Float**

  hyp x y = sqrt (x*x + y*y)

  hyp 3 4 = (¥x -> ¥y -> sqrt(x*x+y*y) 3 4
  = (¥x -> (¥y -> sqrt(x*x+y*y) 3) 4
  = (¥y -> sqrt(3*3 + y*y) 4
  = sqrt(3*3 + 4*4) = sqrt 25 = …

ラムダの束縛は、右結合

f x y = (f x) y

f x y z = (f x) y z =((f x) y) z

関数の適用は、左結合

Curry化

- n個の引数をもつ関数f があるとする。n個の引数の型を t_1, t_2, t_3, …, t_n とし関数f の返り値の型をt_{n+1}とすると、関数f の型は、次のようになる。

f :: t_1-> t_2-> t_3-> …-> t_n->t_{n+1}

n個の引数の型　　返り値の型

- **例**　関数の型**:: Float -> Float -> Float**
  は、Float型の引数を二つ取り、返り値の型がFloat である関数の型を表している。

Type Class

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

「型 a は、型クラス Eq に属する。もしも、(==)と(/=)という名の、与えられた型をもつ関数が、その上で定義されるなら。」

Type Class

```haskell
elem :: Eq a => a -> [a] -> Bool
elem y []     = False
elem y (x:xs) = (x == y) || elem y xs
```

「関数 elem は、Eq a というコンテキストのもとで、
　型 a -> [a] -> Bool を持つ。(Eq a => ....)」

Haskellでは、'**=>**' は、**class constrain** と呼ばれることがある。

次の出力を説明せよ

```
*Main Lib> :t \(x, y)-> (x*x +y)
\(x, y)-> (x*x +y) :: Num a => (a, a) -> a
*Main Lib> ( \(x, y)-> (x*x +y)) (3,4)
13
*Main Lib> :t \x -> \y -> (y * y + x)
\x -> \y -> (y * y + x) :: Num a => a -> a -> a
*Main Lib> (\x -> \y -> (y * y + x)) 3 4
19
*Main Lib> :t (\x -> \y -> (y * y + x)) 3
(\x -> \y -> (y * y + x)) 3 :: Num a => a -> a
*Main Lib> :t \(x,y) -> (y * y + x)
\(x,y) -> (y * y + x) :: Num a => (a, a) -> a
*Main Lib> (\(x,y) -> (y * y + x))(3,4)
19
*Main Lib>
```

Data型
List Comprehension

データ型

```
data Vector = Vector Float Float
```

ここで、**Vector** は、データ型の名前で、**Vector** は、この型のコンストラクタである。

データ型

- data Maybe a = Nothing | Just a

  ```
  mapMaybe :: (a->b) -> Maybe a -> Maybe b
  mapMaybe f (Just x) = Just (f x)
  mapMaybe f Nothing = Nothing
  ```

- data Tree a = Leaf a | Branch (Tree a) (Tree a)

  ```
  size :: Tree a -> Int size (Leaf x) = 1
  size (Branch t u) = size t + size u + 1
  ```

List comprehensionsの例

- [x*x | x <- xs]
  リストxsの要素の二乗からなるリストを返す。

- [f | f <- [1..n], n ‘mod‘ f == 0]
  nの因数のリストを返す。

- concatMap :: (a -> [b]) -> [a] -> [b]
  concatMap f xs = [y | x <- xs, y <- f x]
  リストxsの各要素に関数fを適用して、その結果のリストを返す。

Haskellプログラミング

https://github.com/ghc/ghc/blob/master/libraries/base/GHC/List.hs

Declaration style と Expression style

filter :: (a -> Bool) -> [a] -> [a]

- -- Declaration style

```
filter p [] = []
filter p (x:xs) | p x = x : rest
                | otherwise = rest
                  where rest = filter p xs
```

- -- Expression style

```
filter = ¥p -> ¥xs ->
         case xs of
         [] -> []
         (x:xs) -> let rest = filter p xs
                   in if (p x) then x : rest
                      else rest
```

- The declaration style attempts, so far as possible, to define a function by multiple equations, each of which uses pattern matching and/or guards to identify the cases it covers.
- In contrast, in the expression style a function is built up by composing expressions together to make bigger expressions. Each style is characterised by a set of syntactic constructs:

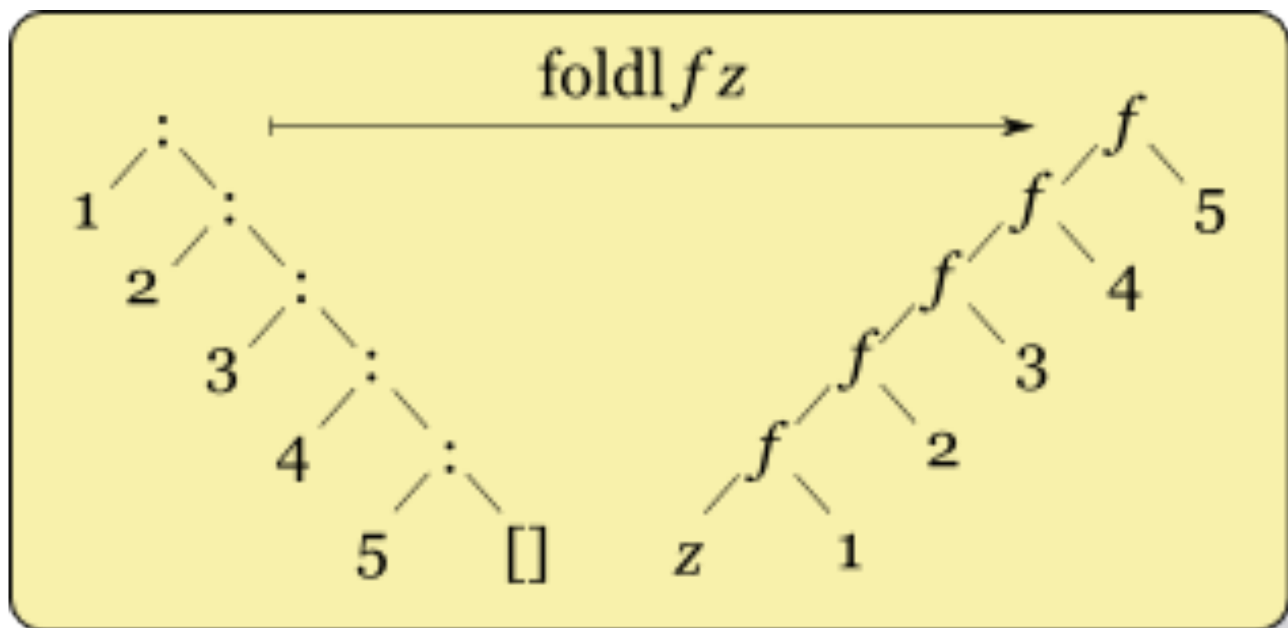
Declaration style	Expression-style
`where` clause	`let` expression
Function arguments on left hand side	Lambda abstraction
Pattern matching in function definitions	`case` expression
Guards on function definitions	`if` expression

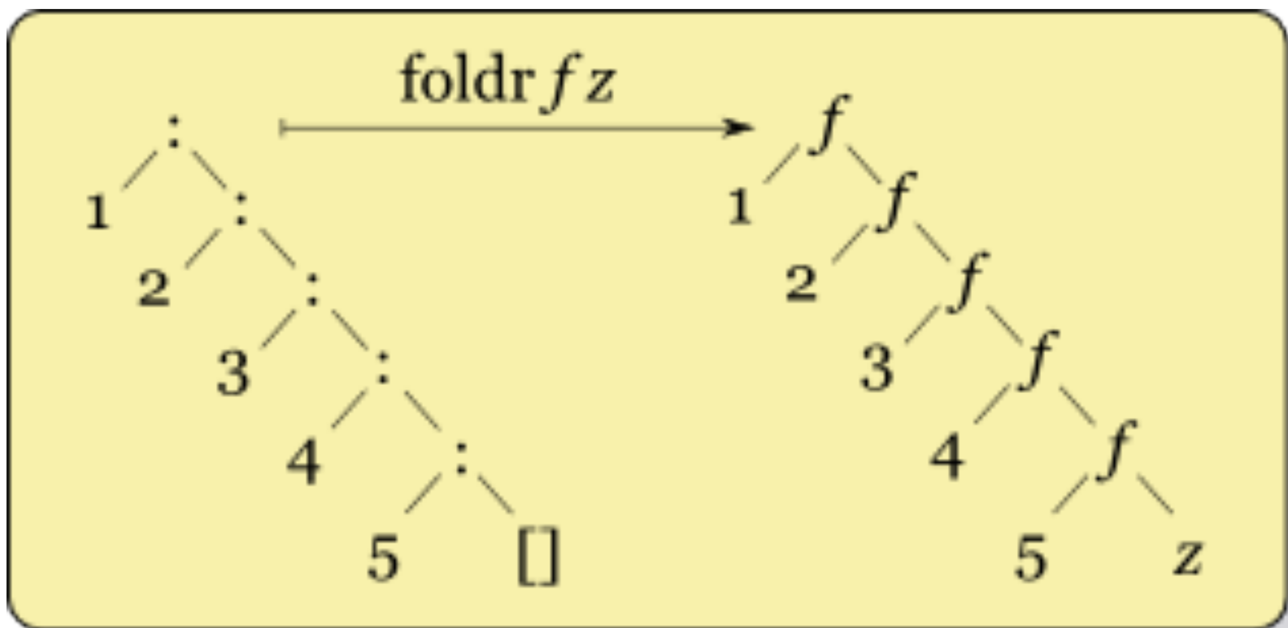
```
head                    :: [a] -> a
head (x:_)              =  x
head []                 =  badHead

tail                    :: [a] -> [a]
tail (_:xs)             =  xs
tail []                 =  errorEmptyList "tail"

length                  :: [a] -> Int
length xs               = lenAcc xs 0

lenAcc          :: [a] -> Int -> Int
lenAcc []    n = n
lenAcc (_:ys) n = lenAcc ys (n+1)

filter :: (a -> Bool) -> [a] -> [a]
filter _pred []    = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      = filter pred xs
```

```
-- > foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...)
`f` xn
--
-- The list must be finite.
--
-- >>> foldl (+) 0 [1..4]
-- 10
-- >>> foldl (+) 42 []
-- 42
-- >>> foldl (-) 100 [1..4]
-- 90
-- >>> foldl (¥reversedString nextChar -> nextChar :
reversedString) "foo" ['a', 'b', 'c', 'd']
-- "dcbafoo"
foldl :: forall a b. (b -> a -> b) -> b -> [a] -> b
{-# INLINE foldl #-}
foldl k z0 xs =
  foldr (¥(v::a) (fn::b->b) -> oneShot (¥(z::b) -> fn (k z v)))
(id :: b -> b) xs z0
  -- See Note [Left folds via right fold]
```

```haskell
-- | The 'sum' function computes the sum of a finite list
of numbers.

sum                      :: (Num a) => [a] -> a
sum                      =  foldl (+) 0

-- | The 'product' function computes the product of a
finite list of numbers.

product                  :: (Num a) => [a] -> a
product                  =  foldl (*) 1
```

```
-- | 'foldr1' is a variant of 'foldr' that has no starting value argument,
-- and thus must be applied to non-empty lists.

foldr1                  :: (a -> a -> a) -> [a] -> a
foldr1 f = go
  where go [x]          =  x
        go (x:xs)       =  f x (go xs)
        go []           =  errorEmptyList "foldr1"
```





https://en.wikipedia.org/wiki/Fold_(higher-order_function)

OCaml

The OCaml system release 4.10
Documentation and user's manual
Xavier Leroy, 2020

http://caml.inria.fr/pub/docs/manual-ocaml/

型付きラムダ計算とOCaml

チャーチのラムダ記法とOCamlのラムダ記法

- チャーチのラムダ記法とOCamlのラムダ記法は、若干の違いがあるが(｀λ'が｀fun'に、ラムダ変数と関数本体を区切るドット｀.'が｀->'に変わっている)、本質的には同じものである。

- ラムダによる抽象化

$\lambda x.(\lambda y.(y^2 + x))$ チャーチ流

```
fun x -> fun y -> (y * y + x)      OCaml流
\x -> \y -> (y * y + x)            Haskell流
```

- ラムダ式への値の適用

$\lambda x.(\lambda y.(y^2 + x))\ 3\ 4 = 19$ チャーチ流

```
(fun x -> fun y -> (y * y + x)) 3 4      OCaml流
(\x -> \y -> (y * y + x)) 3 4            Haskell流
```

チャーチのラムダ記法とOCamlのラムダ記法

- ラムダによる抽象化
  - $\lambda(x,y).(y^2 + x)$ チャーチ流
  - `fun (x,y) -> (y * y + x)` OCaml流
  - `\(x,y) -> (y * y + x)` Haskell流
  - `((LAMBDA (X Y) (PLUS (SQUARE Y) X))` LISP S式

- ラムダ式への値の適用
  - $\lambda(x,y).(y^2 + x))\ (3,4) = 19$ チャーチ流
  - `(fun (x,y) -> (y * y + x))(3,4) = 19` OCaml流
  - `(\(x,y) -> (y * y + x))(3,4) = 19` Haskell流
  - `(((LAMBDA (X Y) (PLUS (SQUARE Y) X)) (3 4))` LISP S式

Ocamlプログラミング

https://github.com/ocaml/ocaml/blob/trunk/stdlib/list.ml

関数を変数にアサインする
関数に名前をつける

```
(* function can be assigned to variable *)
let f = fun n -> n + 1;;
f 2;;     (* ⇒ 3 *)

(* define named function *)
let f x = x + 1;;
f 3;;     (* ⇒ 4 *)
(* named function of 2 parameters *)
let f x y = x + y;;
f 3 4;;   (* ⇒ 7 *)
```

再帰的関数

```
(* example of factorial function *)
let rec f n = if n == 1 then 1 else n * f (n-1);;
f 3     (* ⇒ 6 *)

(* example of fibonacci sequence. *)
(* nth term is sum of previous 2 terms. Starting with
0 and 1. *)
let rec f n = match n with 0 -> 0
    | 1 -> 1
    | x -> f (n-1) + f (n-2);;
f 9;;
```

```
(* List operations *)

let rec length_aux len = function
    [] -> len
  | _::l -> length_aux (len + 1) l

let length l = length_aux 0 l

let cons a l = a::l

let hd = function
    [] -> failwith "hd"
  | a::_ -> a

let tl = function
    [] -> failwith "tl"
  | _::l -> l
```

```
let rec flatten = function
    [] -> []
  | l::r -> l @ flatten r

let concat = flatten

let rec map f = function
    [] -> []
  | a::l -> let r = f a in r :: map f l


let rec fold_left f accu l =
  match l with
    [] -> accu
  | a::l -> fold_left f (f accu a) l

let rec fold_right f l accu =
  match l with
    [] -> accu
  | a::l -> f a (fold_right f l accu)
```

Part IV

Coqでの関数型プログラミング

型付きラムダ計算とCoq

チャーチのラムダ記法とCoqのラムダ記法

- チャーチのラムダ記法とCoqのラムダ記法は、若干の違いがあるが(`λ’が`fun’に、ラムダ変数と関数本体を区切るドット`.’が`=>’に変わっている)、本質的には同じものである。

- ラムダによる抽象化

λx.(λy.(y^2 + x))	チャーチ流
fun x => fun y => (y * y + x)	Coq流
fun x -> fun y -> (y * y + x)	OCaml流
\x -> \y -> (y * y + x)	Haskell流

- ラムダ式への値の適用

λx.(λy.(y^2 + x)) 3 4 = 19	チャーチ流
(fun x -> fun y -> (y * y + x)) 3 4	Coq流
(fun x -> fun y -> (y * y + x)) 3 4	OCaml流
(\x -> \y -> (y * y + x)) 3 4	Haskell流

Coqのラムダ記法

```
Check fun x : nat => x + 3 .
```

```
fun x : nat => x + 3
    : nat -> nat
```

```
Check fun x:nat => fun y:nat => y*y + x .
```

```
fun x y : nat => y * y + x
    : nat -> nat -> nat
```

```
Compute (fun x:nat => fun y:nat => y*y + x) 3 4 .
```

```
    = 19
    : nat
```

関数型言語のラムダ記法

	ラムダ記法
Church	$\lambda x.\lambda y.\ (y^2 + x)$
LISP	(LAMBDA (X) ((LAMBDA (Y) (PLUS (SQUARE Y) X)))
Haskell	\x -> \y -> (y*y + x)
OCaml	fun x -> fun y -> (y*y + x)
Coq	fun x => fun y => (y*y + x)

CoqでのType Class

- 型クラス　EqDec の宣言

```
Class EqDec (A : Type) :=
  { eqb : A -> A -> bool ;
    eqb_leibniz : forall x y, eqb x y = true -> x = y }.

Definition neqb {A} {eqa : EqDec A} (x y : A) := negb (eqb x y).
```

- 型クラスEqDec の実装

```
Instance unit_EqDec : EqDec unit :=
  { eqb x y := true ;
    eqb_leibniz x y H :=
      match x, y return x = y with
      | tt, tt => eq_refl tt
      end }.
```

Coqでの関数定義

型のチェックと定義

```
Coq < Check 2 + 3.
2 + 3 : nat

Coq < Check 2.
2 : nat

Coq < Check (2 + 3) + 3.
(2 + 3) + 3 : nat

Coq < Definition three := 3.
three is defined
```

型のチェックと定義

```
Coq < Definition add3 (x : nat) := x + 3.
add3 is defined

Coq < Check add3.
add3 : nat -> nat

Coq < Check add3 + 3.
Error the term "add3" has type "nat -> nat"
while it is expected to have type "nat”

Coq < Check add3 2.
add3 2 : nat
```

値の計算

```
Coq < Compute add3 2.
= 5 : nat

Coq < Definition s3 (x y z : nat) := x + y + z.
s3 is defined

Coq < Check s3.
s3 : nat -> nat -> nat -> nat

Coq < Compute s3 1 2 3
=6 : nat
```

関数の定義と引数

```
Coq < Definition rep2 (f : nat -> nat)(x:nat) := f (f x).
rep2 is defined

Coq < Check rep2.
rep2 : (nat -> nat) -> nat -> nat

Coq < Definition rep2on3 (f : nat -> nat) := rep2 f 3.
rep2on3 is defined

Coq < Check rep2on3.
rep2on3 : (nat -> nat) -> nat
```

無名関数の定義

```
Coq < Check fun (x : nat) => x + 3.
fun x : nat => x + 3 : nat -> nat

Coq < Compute (fun x:nat => x+3) 4.
     = 7 : nat

Coq < Check rep2on3 (fun (x : nat) => x + 3).
rep2on3 (fun x : nat => x + 3) : nat

Coq < Compute rep2on3 (fun (x : nat) => x + 3) .
     = 9 : nat
```

帰納的データ型の定義の例

bool型の定義（enumerated type）

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

自然数型の定義 (recursive type)

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

帰納的データ型の定義の例

List型の定義 (recursive type)

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A -> list A -> list A.
```

二分木型の定義 (recursive type)

```
Inductive natBinTree : Set :=
| Leaf : natBinTree
| Node (n:nat)(t1 t2 : natBinTree).
```

パターン・マッチングによる定義

bool値の否定の定義

```
Definition negb b :=
match b with
| true => false
| false => true
end.
```

パターン・マッチングによる定義

```
Definition tail (A : Type) (l:list A) :=
match l with
| x::tl => tl
| nil => nil
end.

Definition isempty (A : Type) (l : list A) :=
match l with
| nil => true
| _ :: _ => false
end.
```

パターン・マッチングによる定義

```
Definition has_two_elts (A : Type) (l : list A) :=
match l with
| _ :: _ :: nil => true
| _ => false
end.

Definition andb b1 b2 :=
match b1, b2 with
| true, true => true
| _, _ => false
end.
```

リカーシブな定義

```
Fixpoint plus n m :=
match n with
| O => m
| S n' => S (plus n' m)
end.

Notation : n + m for plus n m
```

1+1 = S(O+1)=S(1)=S(S(O))

リカーシブな定義

```
Fixpoint minus n m := match n, m with
| S n', S m' => minus n' m'
| _, _ => n
end.

Notation : n - m for minus n m
```

3-2=2-1=1-0=1
2-3=1-2=0-1=0

リカーシブな定義

```
Fixpoint mult (n m :nat) : nat :=
match n with
| 0 => 0
| S p => m + mult p m
end.

Notation : n * m for mult n m
```

3*2=2+2*2=2+2+2*1=2+2+2+2*0

ListをCoqでプログラムする

list型の定義

listの基本的な定義は、次のものである。

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A -> list A -> list A.
```

ここで、listは、A : Type なる任意の型に対して
定義されていることに注意しよう。(Polymorphism)

constructor consに対して次の記法を導入する。

```
infix “::” := cons (at level 60, ....)
```

Infix "::" := cons (at level 60, right associativity) : list_scope.

listについての基本的な関数　length

lengthは、listの長さを返す。

```
Definition length (A : Type) : list A -> nat :=
  fix length l :=
  match l with
   | nil => O
   | _ :: l' => S (length l')
  end.
```

再帰的な関数の定義には、**fixpoint** を使うが、
定義の本体に再帰的な定義が現れる場合は、
fix を使う。

listについての基本的な関数　app

app(++)は、二つのlistを結合(append)する。

```
Definition app (A : Type) : list A -> list A ->
list A :=
  fix app l m :=
  match l with
   | nil => m
   | a :: l1 => a :: app l1 m
  end.
```

```
Infix "++" := app (right associativity, ....
```

listについての基本的な関数　map

mapは、listの各要素に関数を適用する。

```
Fixpoint map A B (f : A -> B)(l : list A)
: list B :=
match l with
| nil => nil
| a::l' => f a :: map f l'
end.
```

```
Compute map (fun n => n * n)(1::2::3::4::5::nil).
1::4::9::16::25::nil : list nat
```

listについての基本的な関数 reverse

reverseは、listの要素を逆順にする。

```
Fixpoint naive_reverse (A:Type)(l: list A) :
list A :=
match l with
| nil => nil
| a::l' => naive_reverse l' ++ (a::nil)
end.
```

```
Compute naive_reverse (1::2::3::4::5::6::nil).
= 6 :: 5 :: 4 :: 3 :: 2 :: 1 :: nil    : list nat
```