関数型言語とラムダ計算

JAIST D1 宇田 拓馬

1. 関数型言語

- Common Lisp, Scheme, Clojure, Erlang
- Haskell, F#, SML, OCaml, Scala, Idris, PureScript

特徴

- 関数が第一級オブジェクト
- 式に参照透過性がある
- 文ではなく式が中心

```
int n = 3 + 2; // 代入文
if (n === 2) { ... } // if文
```

```
> 3 + 2 // 式
> n == 2 // 式
```

1.1. 関数が第一級オブジェクト

第一級オブジェクト

→ 値として扱うことができる

値の生成, 変数への格納, 引数・戻り値への受け渡し, リテラル表記の存在

```c
int foo() {
  int arr[3] = {0, 1, 2};

  return arr; // C言語では配列は第一級オブジェクトではないため戻り値にできない
}
```

```haskell
succ :: Int -> Int
succ x = x + 1;

log (succ 3) -- // -> 4
```

1.2. 参照透過性

参照透過性をもつ式

→ 値に置き換えることができる

- 関数が参照透過である
  → 同じ引数に対していつも同じ値を返す
- 値が参照透過である
  → 破壊的変更がなされない

入出力はどう扱う？

→ モナド, 一意型

```
int foo(int n) {
  int m;
  scanf("%d", m); // 入力が同じでも出力が同じとは限らない
  return n + m;
}
```

1.3. 式が中心

式を組み合わせて処理を構成する

```
int twice(int *xs) {
  for (int i = 0; i < sizeof(xs) / sizeof(int); i++) {
    xs[i] = xs[i] * xs[i];
  }
}
```

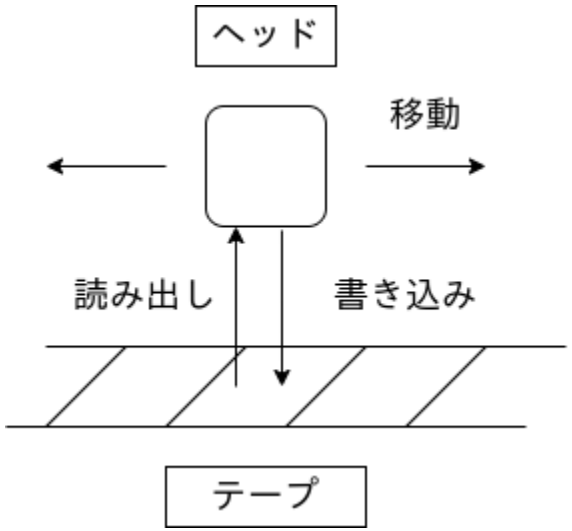
```
twice :: List Int -> List Int
twice xs = map (n -> n * 2) xs

map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

2. ラムダ計算とチューリングマシン

コンピュータ上で表現できる処理を「計算」と呼ぶ。

- チューリングマシン
  → 手続き型言語の基礎
- ラムダ計算
  → 関数型言語の基礎
- 計算可能関数
  チャーチ＝チューリングのテーゼ

2.1. チューリングマシンと手続き型言語




```
int main() {
  int n = 3;
  n = n + n;

  return 0;
}
```

2.2. ラムダ計算

変数, 関数, 関数適用で処理を組み立てる

1. 変数はラムダ式
2. xを変数, Mをラムダ式としたとき, ラムダ抽象`λx. M`はラムダ式
3. M, Nをラムダ式としたとき, 適用`(M N)`はラムダ式

```
0 := λf x. x -- xにfを0回適用
1 := λf x. f x -- xにfを1回適用
2 := λf x. f (f x)

SUCC := λn f x. f (n f x)
PLUS = λm n. m SUCC n
```

```
SUCC 2
-> (λn f x. f (n f x)) 2
-> λf x. f (2 f x)
-> λf x. f ((λf x. f (f x)) f x)
-> λf x. f (f (f x))
-> 3
```

3. まとめ

- 関数型言語の特徴: 関数が第一級, 参照透過性, 式中心
- 手続き型言語→チューリングマシン
- 関数型言語→ラムダ計算