

ML 演習 第 2 回

新井淳也、中村宇佑、前田俊行

2011/04/19

今日の内容

- 型多相性
 - Parametric Polymorphism
- ユーザ定義型
 - レコード型
 - バリエーション型
- 多相データ型
- 例外
- 副作用を利用したプログラミング

Type Polymorphism

型多相性

型多相性とは？

- 異なる型を持つ値・式などを
どうにかして
まとめて扱う仕組みのこと

型多相性が無いと悲惨な例: リストの先頭を取出す関数

- リストの要素の型によって別々に定義しなければならない
 - 整数値のリストの先頭を取出す関数
`val hd_int : int list -> int`
 - ブール値のリストの先頭を取出す関数
`val hd_bool : bool list -> bool`
 - 整数値とブール値の組のリストの先頭を取出す関数
`val hd_i_x_b :
 (int * bool) list -> int * bool`
 - etc.

しかし、中身は一緒にいいはず

- `let hd_int = function (x :: _) -> x ;;`
- `let hd_bool = function (x :: _) -> x ;;`
- `let hd_i_x_b = function (x :: _) -> x ;;`

完全に一致

- 全く同じ処理を繰り返し書くのは
無駄かつ誤りを生じやすい
 - どうやったら一つの関数で
どんな型のリストも扱えるようにできるか？

解決法: 型をパラメータにする

- $\text{hd}[\alpha] : \alpha \text{ list} \rightarrow \alpha$

型パラメータ

– $\text{hd}[\text{int}] : \text{int list} \rightarrow \text{int}$

– $\text{hd}[\text{bool}] : \text{bool list} \rightarrow \text{bool}$

– $\text{hd}[\text{int} * \text{bool}] :$

$(\text{int} * \text{bool}) \text{ list} \rightarrow (\text{int} * \text{bool})$

– ...

} どれも
hd[α] の
インスタンスに
なっている

- OCaml では型パラメータは書かなくてよい
 - 型推論システムが自動的に型パラメータを補う

Parametric Polymorphism

- 型をパラメータにすることで
型は異なるが本質的に同一なモノを
ひとまとめにする方法
 - Cf. オブジェクト指向の polymorphism
(\doteq subtyping polymorphism)
 - 例えば Javaの継承
 - Cf. overloading (ad-hoc polymorphism)
 - 例えば、OCaml の比較演算子、Haskell の type class

多相関数の例: 恒等関数

```
# let id x = x;;  
val id : 'a -> 'a = <fun>  
# id 1;;  
- : int = 1  
# id [1; 2; 3];;  
- : int list = [1; 2; 3]  
# id (fun x -> x + 1);;  
- : int -> int = <fun>
```

型パラメータは
'a や 'b で表わされる

多相関数の例: ペアの要素を取り出す関数

```
# let fst (x, _) = x;;
```

```
val fst : 'a * 'b -> 'a = <fun>
```

```
# let snd (_, x) = x;;
```

```
val snd : 'a * 'b -> 'b = <fun>
```

多相関数の例: 長さ n のリストを作る関数

```
# let rec make_list n v =  
    if n = 0 then []  
    else v :: make_list (n - 1) v;;  
val make_list : int -> 'a -> 'a list = <fun>  
  
# make_list 3 1;;  
- : int list = [1; 1; 1]  
# make_list 4 "a";;  
- : string list = ["a"; "a"; "a"; "a"]
```

多相関数の例

- 前回の課題で作った関数
 - `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
 - `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
 - `append : 'a list -> 'a list -> 'a list`
 - `filter : ('a -> bool) -> 'a list -> 'a list`
 - `split : ('a * 'b) list -> 'a list * 'b list`
 - `comb : 'a list -> int -> 'a list list`

型を明示的に指定することもできる

- 多相型を持つ式・変数に対し型を明示できる

```
# let id x = x;;
```

```
val id : 'a -> 'a = <fun>
```

```
# let id (x : int) = x;;
```

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

```
# let id x = (x : int);;
```

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

```
# id true;;
```

This expression has type bool but...

- あまり使う機会はないかも

User-Defined Types

ユーザ定義型

独自データ型の定義

- レコード (record)
 - 複数の値を組にした値
 - C 言語では struct に相当
- バリエーション (variant)
 - 何種類かの値のうち一つをとる値
 - C 言語では enum, union, struct の組み合わせに相当
 - 操作の安全性が型検査により保証される

レコード型の定義

- 構文:

```
type 「型名」 = { 「フィールド名1」 : 「型1」;  
                「フィールド名2」 : 「型2」; ... }
```

- 例: 複素数を表す型 `complex` の定義

```
# type complex = { re : float; im : float };;  
type complex = { re : float; im : float };;
```


レコード型の値の生成を表す式

- 構文

{ 「フィールド名1」 = 「式1」; 「フィールド名2」 = 「式2」; ... }

- 例: complex 型を持つ値の作成

```
# let c1 = { re = 5.0; im = 3.0 };;  
val c1 : complex = {re = 5.; im = 3.}
```



フィールド名で
型が推論される

レコードのフィールドを使用する式

- 構文
「式」.「フィールド名」
- 例: 複素数の実数成分を取り出す

```
# c1.re;;  
- : float = 5.
```

レコードに対するパターンマッチング

```
# let scalar n { re = r; im = i } =  
    { re = n *. r; im = n *. i };;  
val scalar : float -> complex -> complex = <fun>
```

```
# scalar 2.0 { re = 5.0; im = 3.0 };;  
- : complex = {re = 10.; im = 6.}
```

– もちろん `match ... with ...` でも使える

バリエーション型の定義

- 構文

```
type 「型名」 = 「タグ1」 of 「型1」  
              | 「タグ2」 of 「型2」 | ...
```

- 例1: bool 型と同等の定義

```
# type mybool = True | False;;  
type mybool = True | False
```

値を取らない場合は
「of～」を省略する

タグの先頭は
必ず大文字

- 例2: シンプルなインタプリタの値の定義

```
# type value = Int of int  
              | Bool of bool;;  
type value = Int of int | Bool of bool
```

バリエーション型の値の生成を表す式

- 構文
「タグ」「式」

- 例1: mybool 型の値の生成

```
# True;;  
- : mybool = True  
  
# False;;  
- : mybool = False
```

値を取らない場合は
「式」を省略する

- 例2: シンプルなインタプリタの値の生成

```
# Int 156;;  
- : value = Int 156  
  
# Bool false;;  
- : value = Bool false
```

バリエーション型の値を使用する式: パターンマッチング

```
# let not = function
    | True -> False
    | False -> True;;
val not : mybool -> mybool = <fun>
# not True;;
- : mybool = False
# not False;;
- : mybool = True
# let print_value = function
    | Int i -> print_int i
    | Bool b -> print_string
                (if b then "true" else "false")
val print_value : value -> unit = <fun>
```

バリエーション型の再帰的な定義

- 例: 各ノードが整数値を持つ二分木
 - 二分木の型 `inttree` を定義

```
# type inttree =
```

```
  | Leaf
```

```
  | Node of int * inttree * inttree;;
```

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

- `inttree` 型の値を作る

```
# Leaf;;
```

```
- : inttree = Leaf
```

```
# Node (1, Leaf, Leaf);;
```

```
- : inttree = Node (1, Leaf, Leaf)
```

最初の縦棒「|」は省略してもよい

型の定義中にその型自身が含まれているものを再帰型という

バリエーションのパターンマッチング、再び

- 例: 木に含まれる全ての整数の和を求める

```
# let rec sum_tree = function
    | Leaf -> 0
    | Node (v, t1, t2) ->
        v + sum_tree t1 + sum_tree t2;;
val sum_tree : inttree -> int = <fun>
# sum_tree (Node(4, Node(5, Leaf, Leaf), Leaf));;
-: int = 9
```


相互再帰的な型の定義

- 例: 整数値とブール値が交互に現れるリスト

```
# type list =
```

```
  | Nil
```

```
  | Cons of int * list'
```

```
and list' =
```

```
  | Nil'
```

```
  | Cons' of bool * list;;
```

```
type list = Nil | Cons of int * list'
```

```
and list' = Nil' | Cons' of bool * list
```

and でつなくと
相互再帰的に
定義できる

Polymorphic Data Type

多相データ型

多相データ型の必要性

- 二分木が持つデータは整数とは限らない
 - type inttree = Leaf
| Node of int * inttree * inttree
 - type booltree = Leaf
| Node of bool * booltree * booltree
 - type ibtree = Leaf
| Node of (int * bool) * ibtree * ibtree
- データ構造としては同じなのでまとめたい
 - 多相関数と同じように.....

解決法: 型をパラメータにする、再び

- $\text{tree}[\alpha] = \text{Leaf} \mid \text{Node of } \alpha * \text{tree}[\alpha] * \text{tree}[\alpha]$

型パラメータ

- $\text{tree}[\text{int}] = \text{Leaf}$
| Node of $\text{int} * \text{tree}[\text{int}] * \text{tree}[\text{int}]$
- $\text{tree}[\text{bool}] = \text{Leaf}$
| Node of $\text{bool} * \text{tree}[\text{bool}] * \text{tree}[\text{bool}]$
- ...

多相データ型の例: ノードが値を持つ二分木

```
# type 'a tree =  
    Leaf | Node of 'a * 'a tree * 'a tree;;  
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree  
# Node (5, Leaf, Leaf);;  
- : int tree = Node (5, Leaf, Leaf)  
# Node (true, Leaf, Leaf);;  
- : bool tree = Node (true, Leaf, Leaf)  
# Leaf;;  
- : 'a tree = Leaf
```

組込みの多相データ型の例

- オプション型

- `type 'a option = None | Some of 'a`

- リスト型

- `type 'a list = [] | (::) of 'a * 'a list`

- ちょっと構文が特殊

オプション型の例

- 整数の割り算

```
# let div x y =  
    if y = 0 then None  
    else Some (x / y);;
```

```
val div : int -> int -> int option = <fun>
```

```
# div 8 2;;
```

```
- : int option = Some 4
```

```
# div 8 0;;
```

```
- : int option = None
```

多相データ型と多相関数

- 二分木の高さを求める関数

```
# let rec height = function
  | Leaf -> 0
  | Node (_, t1, t2) ->
    1 + max (height t1) (height t2);;
val height : 'a tree -> int = <fun>
```


複数の型パラメータを持つ 多相データ型

タプルのように区切る

```
# type ('a, 'b) either = L of 'a | R of 'b;;  
type ('a, 'b) either = L of 'a | R of 'b  
# L 1;;  
- : (int, 'a) either = L 1
```

Exceptions

例外

例外処理とは?

- エラーが発生したときに現在の計算を中断してエラー処理用のコードにジャンプする機構
 - 開こうとしているファイルが見つからない
 - 配列の境界を越えてアクセスした
 - 0 による除算を行った
 - ユーザの入力がおかしい
 - etc.
- C++ や Java にも同様の機構がある

例外を発生させるには?

(* 例外発生を表す式: raise 「(例外を表す)式」 *)

```
# let div_e x y =  
  if y = 0  
  then raise Division_by_zero  
  else x / y;;
```

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

```
# div_e 8 2;;
```

```
- : int = 4
```

```
# div_e 8 0;;
```

```
Exception: Division_by_zero.
```

評価結果の値の代わりに
例外が表示される

例外を処理するには?

(* 例外処理を表す式: try 「式」 with 「パターン1」 -> 「式1」
| 「パターン2」 -> 「式2」
| ... *)

```
# let div x y =
```

```
  try
```

```
    Some (div_e x y) } この間で例外が発生したら.....
```

```
  with
```

```
    Division_by_zero -> None;;
```

これを評価

```
val div : int -> int -> int option = <fun>
```

```
# div 8 2;;
```

```
- : int option = Some 4
```

```
# div 8 0;;
```

```
- : int option = None
```

ユーザ定義例外

- 例外を定義する構文:
`exception 「タグ名」 of 「型」`

例外を定義

```
# exception My_exception;;  
exception My_exception
```

例外を普通の
式として評価

```
# My_exception;;  
- : exn = My_exception
```

例外を発生

```
# raise My_exception;;  
Exception: My_exception.
```

- じつは例外は特殊なバリエーションになっている

値を持つ例外

- 例外はバリエーションの一種なので値を持てる

```
# exception My_int_exception of int;;  
exception My_int_exception of int  
# let isprime n =  
    if n <= 0 then raise (My_int_exception n)  
    else ...;;  
val isprime : int -> bool = <fun>
```

複数の例外を処理するには?

```
# exception My_exception;;  
# exception My_int_exception of int;;  
# exception My_bool_exception of bool;;  
# let foo x =  
  try  
    bar x  
  with  
  | My_exception -> None  
  | My_int_exception i -> Some i  
  | My_bool_exception _ -> None;;
```

with の後の部分は
パターンマッチングに
なっている

try ... with 式の入れ子

```
try
  try
    raise My_exception
  with
    My_int_exception i -> Some i
with
  My_exception -> None
```

内側の with には
マッチしないのでスルー

外側の with では
マッチするので
ここで処理される

Programs with Side Effects

副作用を利用したプログラミング

副作用とは?

- ある式の評価が参照透過でないとき
その式は副作用を持つ、という
 - ある式が参照透過であるとは、
大雑把に言って
その式の評価結果の値が常に等しくなること
- ※ 副作用はプログラムを読み難くするので
ここぞという時に限って使うとよい

Unit 型

- 「`()`」を唯一の値とする型

```
# ();;
```

```
- : unit = ()
```

- 用途:
 - 引数が不要な関数に渡すダミー引数
 - 意味のある戻り値を返さない関数や式が返すダミーの値
 - 特に式の副作用のみが重要な場合

Unit の使われ方の例

```
# print_string;;  
- : string -> unit = <fun>  
# print_string "Foo\n";;  
Foo  
- : unit = ()  
  
# read_line;;  
- : unit -> string = <fun>  
# let s = read_line ();;  
Bar  
val s : string = "Bar"
```

値を変更できるレコード

- レコード型を定義するとき、フィールド名の前に `mutable` を付けるとそのフィールドは更新可能になる

```
# type bank_account =  
    { name : string; mutable balance : int };;  
type bank_account =  
    { name : string; mutable balance : int; }
```

レコードのフィールドの更新

- 「<-」演算子を使う

(* 「式1」.「フィールド名」<- 「式2」 *)

```
# let a = { name = "maeda"; balance = 10000 };;
```

```
val a : bank_account =  
      {name = "maeda"; balance = 10000}
```

```
# a.balance;;
```

```
- : int = 10000
```

```
# a.balance <- 20000;;
```

```
- : unit = ()
```

```
# a.balance;;
```

```
- : int = 20000
```

代入を表す式の
型は unit

参照 (Reference)

- 中身を変更できる「入れ物」
 - C や Scheme の変数のように再代入できる
 - 以下のようなレコードとして組込みで定義されている
 - `type 'a ref = { mutable contents : 'a }`
 - 参照を操作するための関数や演算子も組込みで定義されている (次ページ)

参照のための組込み関数と演算子

```
# let r = ref 0;;
```

新しい参照を作る ref 関数

```
val r : int ref = {contents = 0}
```

```
# !r;;
```

参照の中身を取り出す「!」演算子

```
- : int = 0
```

```
# r := 1;;
```

参照に代入する「:=」演算子

```
- : unit = ()
```

```
# !r;;
```

```
- : int = 1
```

複数の式の逐次実行

- 複数の式を順番に評価する
- 最後に評価した式の結果を返す

- 構文:

「式1」 ; 「式2」 ; ... ; 「式n」

- 例:

```
# let t = (print_string "> "; read_line ());;
```

```
> foo
```

```
val t : string = "foo"
```

if 式の else 節の省略

else 節が () だけのときは省略可能

(* if 「式」 then 「式1」は
if 「式」 then 「式1」 else () と等しい *)

```
# let r = ref 0;;
```

```
val r : int ref = {contents = 0}
```

```
# if !r = 0 then r := 3;;
```

```
- : unit = ()
```

- 誤用例:

```
# r := if !r = 0 then 3;;
```

This expression has type int but is here used with type unit

条件式の評価結果が真のときだけ副作用のある計算を実行

型多相と副作用は相性が悪い

- (実際には型エラーになる) 例:

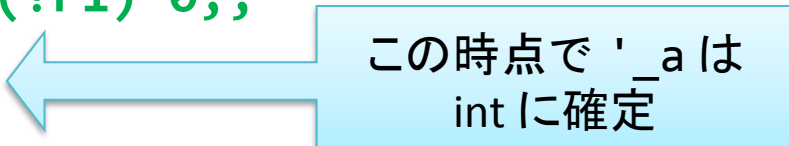
```
# let r1 = ref [];;  
val r1 : 'a list ref = { contents = [] }  
# let sum () = fold_right (+) (!r1) 0;;  
val sum : unit -> int = <fun>  
# r1 := [true];;  
- : unit = ()  
# sum ();;
```

- どこがおかしい??

O'Caml の解決案

- なんでもかんでも多相型にしない
- 参照には「未決定な単相型」を与える
 - いったん型が決まるともう他の型としては使えない

```
# let r1 = ref [];;  
val r1 : 'a list ref = { contents = [] }  
# let sum () = fold_right (+) (!r1) 0;;  
val sum : unit -> int = <fun>  
# r1;;  
- : int list ref = { contents = [] }  
# !r1 @ [true];;  
This expression has type bool but is here used with type int
```



この時点で 'a は
int に確定

問題は参照以外にも!

- この関数 f の型は $\text{unit} \rightarrow 'a \rightarrow 'a$ だが、式 $f ()$ の型は $'a \rightarrow 'a$ でよいか?

```
let f () =  
  let r = ref None in  
  fun x ->  
    let old = match !r with  
              None -> x | Some y -> y  
    in  
    r := Some x; old
```

OCaml の最終的な解決案

- Value restriction:
副作用がないと確実にわかる「値」
にのみ多相型を与える
 - 結局のところ、評価される式は副作用を持ちうるので
 - OK: 定数、fun 式、それらの タプル、
それらからなる変更不可データ構造
 - NG: 参照、let 式、関数適用 etc...
(fun () x -> x) ();;
- : 'a -> 'a = <fun>

Value restriction で注意すべきこと

- 部分適用が単相型になることがある

```
# let f = List.map (fun x -> (x, x));;  
(* 多相型になってほしいが、値ではないので  
   未決定単相型になってしまう *)
```

```
val f : '_a list -> ('_a * '_a) list = <fun>
```

- 解決策: η 展開 (仮引数を明示する)

```
# let f xs = List.map (fun x -> (x, x)) xs;;  
val f : 'a list -> ('a * 'a) list = <fun>
```


Value restriction の参考文献

- 最初に Value restriction を提案した論文
 - Andrew K. Wright, Matthias Felleisen.
A Syntactic Approach to Type Soundness.
 - Value restriction 以外の解決法との得失比較あり
- 提案された OCaml の拡張
 - Jacques Garrigue. Relaxing Value Restriction.
 - OCaml 3.07 で採用

アドバイス

- 論文の探し方
 - 基本的には Google、Google Scholar
 - タイトル、著者名、発表された会議や雑誌で検索
 - ACM Digital Library (<http://www.acm.org/> から)
 - ACM の学会で発表された論文ならここにもある
 - CiteSeer.IST (<http://citeseerx.ist.psu.edu>)
 - 論文データベース
 - Google で検索すると大抵この検索結果がかかる
 - 著者の Web サイト

第 2 回 課題

締切: 5/3 13:00 (日本標準時)

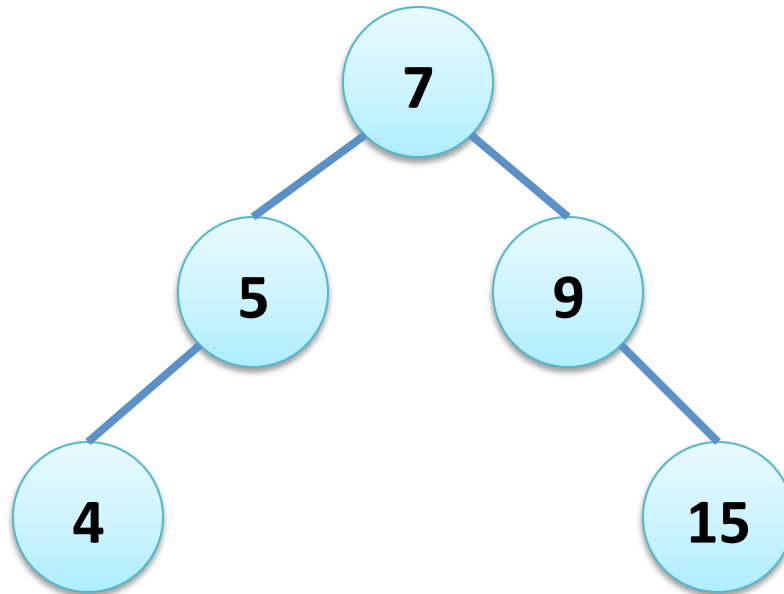
課題 1 (5点)

- 二分木 ('a tree) を受け取り
以下に挙げた探索順それぞれについて
全要素を並べたリストを生成する関数を定義せよ
 - 行きがけ順 (preorder)
 - 通りがけ順 (inorder)
 - 帰りがけ順 (postorder)
- 二分木の定義は以下の通りでよい

```
type 'a tree = Leaf
           | Node of 'a * 'a tree * 'a tree
```

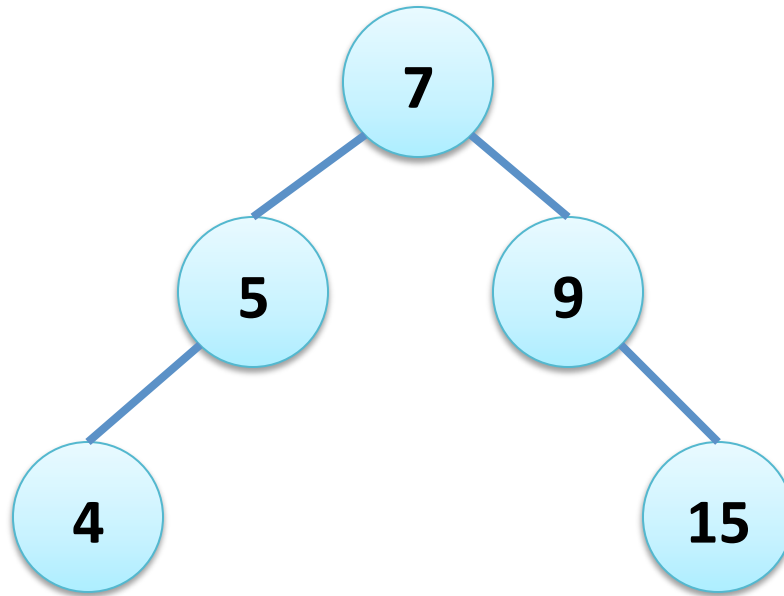
課題 1 (例)

```
# preorder (Node(7,  
                Node(5, Node(4, Leaf, Leaf), Leaf),  
                Node(9, Leaf, Node(15, Leaf, Leaf))))  
- : int list = [7; 5; 4; 9; 15]
```



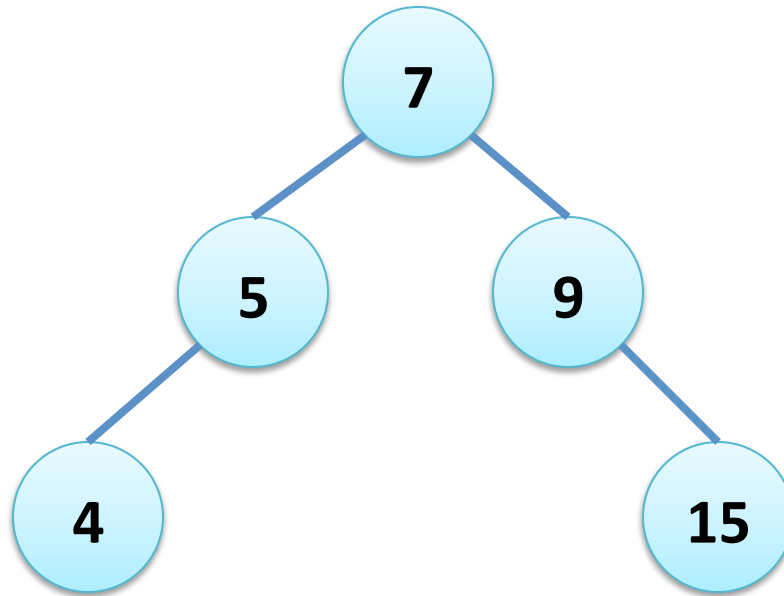
課題 1 (例)

```
# inorder (Node(7,  
              Node(5, Node(4, Leaf, Leaf), Leaf),  
              Node(9, Leaf, Node(15, Leaf, Leaf))))  
- : int list = [4; 5; 7; 9; 15]
```



課題 1 (例)

```
# postorder (Node(7,  
                Node(5, Node(4, Leaf, Leaf), Leaf),  
                Node(9, Leaf, Node(15, Leaf, Leaf))))  
- : int list = [4; 5; 15; 9; 7]
```



課題 2 (5点)

- スタックのデータ構造を表現する多相型を定義して次の操作を実装せよ

```
type 'a stack = { mutable s : 'a list }
val new_stack : unit -> 'a stack
    (* ↑新しい stack を作成する *)
val push : 'a stack -> 'a -> unit
    (* ↑要素を push する *)
val pop : 'a stack -> 'a
    (* ↑要素を pop する。
       stack が空なら Empty_stack 例外を投げる *)
```


課題 2 (例)

```
# let s = new_stack ();;
val s : '_a stack = {s = []}
# push s 1;;
- unit = ()
# push s 2;;
- unit = ()
# pop s;;
- : int = 2
# pop s;;
- : int = 1
# pop s;;
Exception : Empty_stack.
```

課題 3 (5点)

- 課題 2 の実装で、

```
let s = new_stack ()
```

に対するインタプリタの応答が

```
val s : '_a stack = {s = []}
```

となっている。これについて以下の問いに答えよ。

- a. 'a stack と '_a stack の違いを説明せよ
- b. 仮に型が '_a stack ではなくて 'a stack だったとしたら
どのような問題が生じるか説明せよ

課題 4 (10点)

- キューのデータ構造を表現する多相型を定義して次の操作を実装せよ

– ただし

- 各操作はキューの長さによらず $O(1)$ で終了するようにすること
- 副作用を用いてもよい

```
type 'a queue = ???
```

```
val new_queue : unit -> 'a queue
```

(* ↑新しい queue を作成する *)

```
val enqueue : 'a queue -> 'a -> unit
```

(* ↑要素を追加する *)

```
val dequeue : 'a queue -> 'a
```

(* ↑要素を取り出す。

queue が空なら Empty_queue 例外を投げる *)

課題 5 (10点)

- 関数 f を受け取って f を再帰的に無限回合成する関数を返す関数 fix を書け

- f は「関数を受け取って関数を返す関数」と仮定してよい

- 以下のようなイメージ

- $fix\ f \equiv f\ (f\ (f\ (f\ (f\ (f\ \dots))))))$

- 使用例

- ```
let fib = fix (fun g n ->
 if n <= 2 then 1
 else g (n - 1) + g (n - 2));;

val fib : int -> int = <fun>
fib 10;;
- : int = 55
```

- ただし、参照は使ってもよいが `let rec` は使わないこと

# 課題 6 (10点)

- Bool 値と int 値に対する簡単な計算を表す式  $E$  の文法を以下のように定義する

$V \rightarrow (\text{bool 値}) \quad V \rightarrow (\text{int 値})$

$E \rightarrow V$

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$

$E \rightarrow E \&\& E$

$E \rightarrow E || E$

$E \rightarrow E = E$

$E \rightarrow \text{if } E \text{ then } E \text{ else } E$

次ページに続く

# 課題 6 (続き)

- また値  $V$  に対応するバリエント型 `value` を以下のように定義する

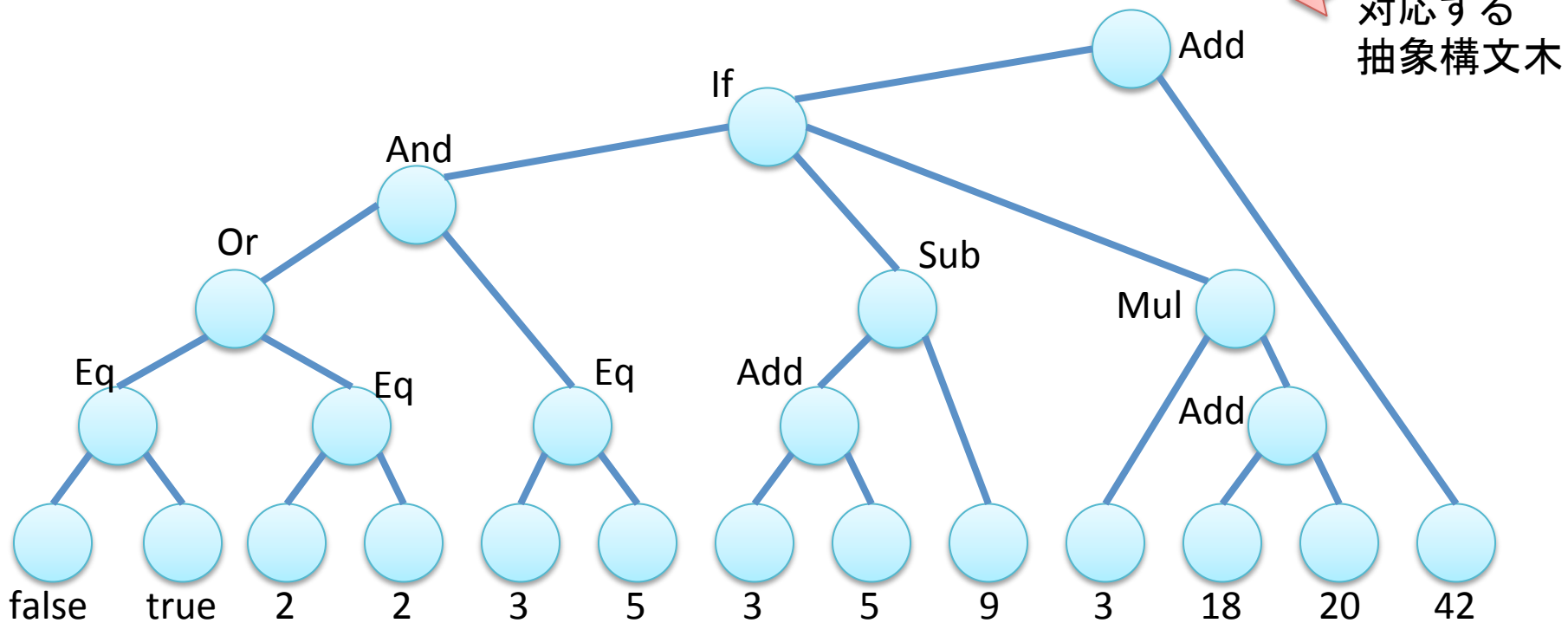
```
type value =
 | Bool_value of bool
 | Int_value of int
```

- このとき式  $E$  の抽象構文木に対応するバリエント型 `expr` を定義せよ
  - 以下の未完成の定義を完成させる形でよい

```
type expr =
 | Const of value
 | Add of expr * expr
 | ...
```

# 課題 6 (式の抽象構文木の例)

```
(if ((false = true) || (2 = 2)) && (3 = 5)
 then (3 + 5) - 9 else 3 * (18 + 20)) + 42
```



# 課題 7 (15点)

- 課題 6 で定義したバリエーション型 `expr` で与えられた式を評価し結果の値を返す関数  
`eval : expr -> value`  
を実装せよ
  - Bool 値と int 値を足し算するなど、評価時エラーが発生した場合はエラーが発生した式を値に持つ例外 `Eval_error` を投げるようにせよ



# 課題 8 (20点)

- 型 `false_t`, `not_t`, `and_t`, `or_t` が以下のように定義されているとする

```
type false_t = B of false_t
type 'a not_t = 'a -> false_t
type ('a, 'b) and_t = 'a * 'b
type ('a, 'b) or_t = L of 'a | R of 'b
```
- このとき、次ページの 1~7 の型についてそれぞれの型を持つ式を `let rec`、再帰型、例外、副作用を用いずに定義できるか示せ
  - 定義できる場合はその定義を、できない場合はその理由を示せばよい
- `let rec` と再帰型を用いてもよいとするとどうか?

# 課題 8 (続き)

1. ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
2. ('a, ('b, 'c) and\_t) or\_t ->  
((('a, 'b) or\_t, ('a, 'c) or\_t) and\_t)
3. ((('a, 'b) or\_t, ('a, 'c) or\_t) and\_t ->  
(('a, ('b, 'c) and\_t) or\_t)
4. ('a, 'a not\_t) or\_t
5. ('a, 'a not\_t) and\_t
6. 'a -> 'a not\_t not\_t
7. 'a not\_t not\_t -> 'a

# 課題 9 (20点)

- キューのデータ構造を表現する多相型を定義して次の操作を実装せよ
  - ただし
    - 各操作はキューの長さによらず  $O(1)$  で終了するようにすること
    - 副作用は用いないこと

```
type 'a queue = ???
```

```
val new_queue : unit -> 'a queue
```

```
(* ↑新しい queue を作成する *)
```

```
val enqueue : 'a queue -> 'a -> 'a queue
```

```
(* ↑要素を追加する *)
```

```
val dequeue : 'a queue -> 'a * 'a queue
```

```
(* ↑要素を取り出す。
```

```
queue が空なら Empty_queue 例外を投げる *)
```