Đặc tả hình thức

GVHD:

- Nguyễn Công Hoan

Nhóm 5:

- Nguyễn Du Lịch
- Vũ Tuấn Hải
- Trần Trung Hiếu
- Nguyễn Duy Minh

Topic

5. Function definition
7. Patterns
8. Binding
9. Value (Constant) definition

5. Function

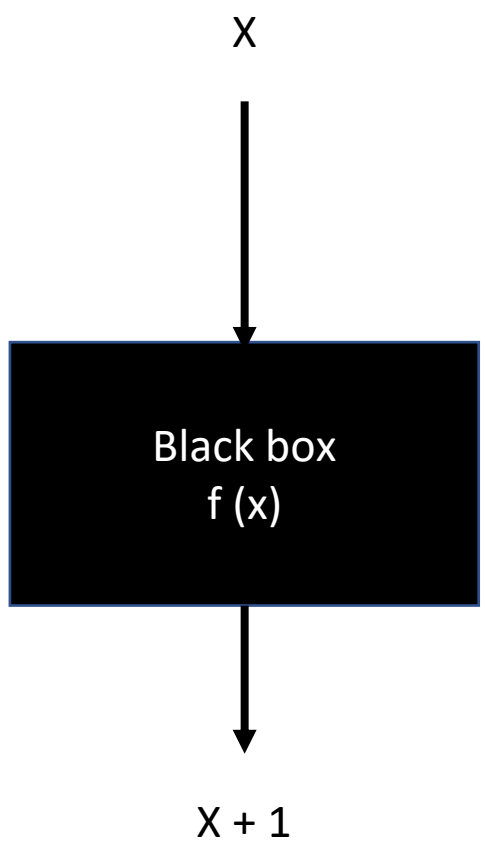
5.1. Definition
5.2. Polymorphic function
5.3. Higher Order function
5.4. Demo

5.1 Definition

A simple function has 3 actions (take argument, excute, return value) and 2 parts (type signature, body).

VDM:

extended explicit function definition =
identifier,
[type variable list],
parameter types,
identifier type pair list,‘==’,
function body,
[‘pre’, expression],
[‘post’, expression],
[‘measure’, measure body] ;



In math: $\lambda x.\,(x+1)$

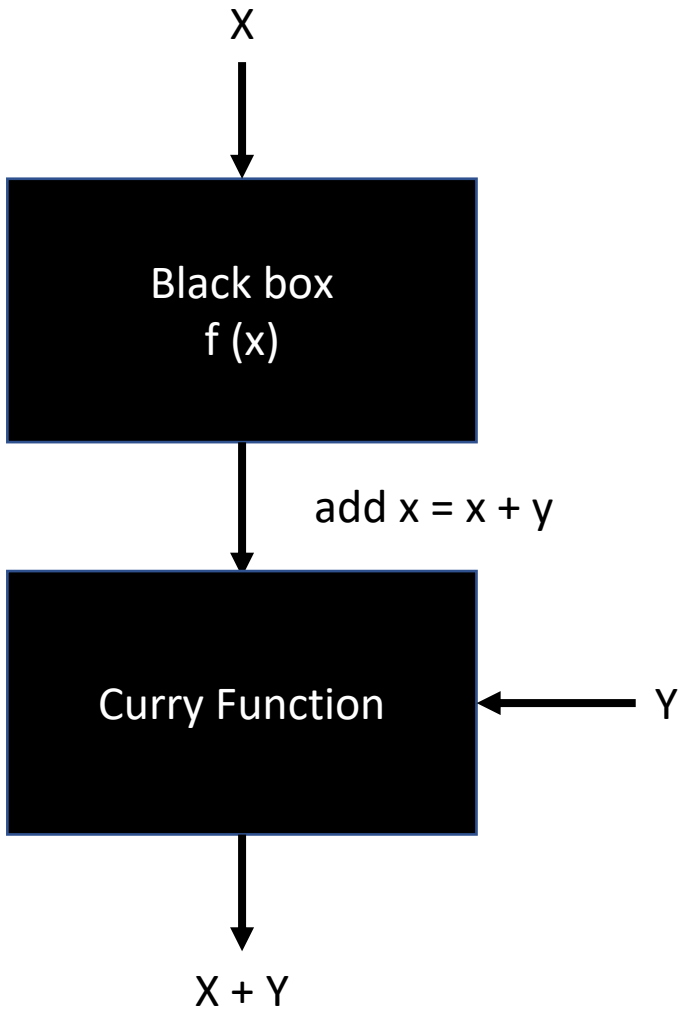
Haskell: add $x = x + 1$

5.1 Definition

A function can take only one argument and return one result.

The function which has multiple arguments is a combination of functions (curry function).

→ Argument and result can be a function.



In math: $\lambda xy.\, xy = \lambda x.\, \lambda y.\, xy$

In Haskell: $add\ x\ y = x + y$

5.1 Definition: Example

map_inter: (map nat to nat) * (map nat to nat) -> map nat to nat
map_inter (m1, m2) ==

(dom m1 inter dom m2) <: m1

pre: (forall d in set dom m1 inter dom m2) & (m1(d) = m2(d))

-> Type signature: take 2 map as arguments, return intersection map

-> return map that can take d as key and return value.

m1: $(1, 4, 6, 3) \rightarrow (3,5,7,1)$, m2: (1, 6, 4, 2) → (3,7,9,9) (Ex: m1(1) = 3)

(dom m1) inter (dom m2) = (1,4,6,7) inter (1,6,4,2) = (1,6,4)

pre: d in (1,6,4), example d = 4, m1(4) = 5, m2(4) = 9 → fail

5.1 Definition: Measure clause

fac: nat +> nat

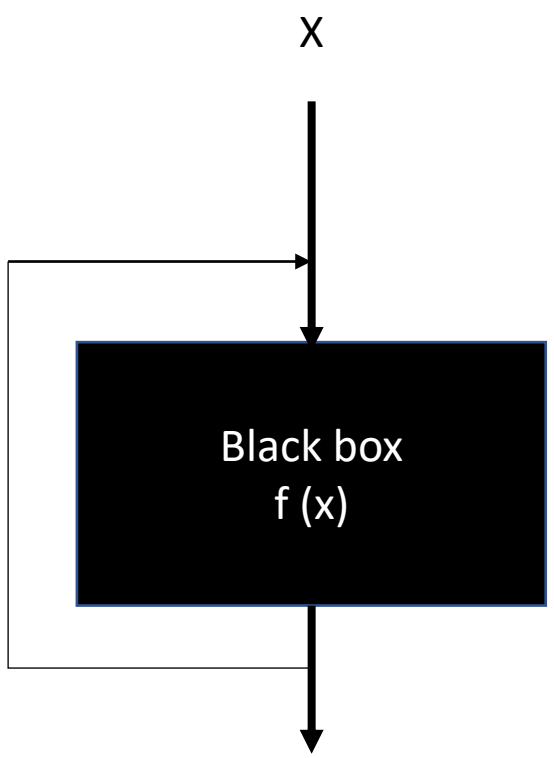
fac(n) ==

if n = 0 then 1 else n * fac(n-1)

measure n; // **or measure is not yet specified**

Boundary condition is n

Measure clause is not mandatory



In math: $(\lambda x.\, xx)(\lambda x.\, xx)$

$Y = \lambda f.\, (\lambda x.\, f(xx))\ (\lambda x.\, f(xx))$

5.1 Definition: Measure clause

```
dupe[@a]: seq of @a +> seq of @a
dupe(s) == cases s:
                  [] -> [],
                  [x] -> [x,x],
                  t^u -> dupe t ^ dupe u
                end;
measure len s;
----------
Boundary condition is len s = 0
```

5.1 Definition: Measure clause

In math: $(\lambda x.\, xx)(\lambda x.\, xx)$ (Ω function)
$Y = \lambda f.\, (\lambda x.\, f(xx))\ (\lambda x.\, f(xx))$ (Y combintor)
---
factorial = g = $\lambda r.\, \lambda n.\, (1\ if\ n = 0, else\ n\ *\ r\ (n\ -\ 1))$
Apply Y combinator to calculate 2!

$(Y\ g)\ 2\ =\ g\ (Y\ g)\ 2$
$= \left(\lambda n.\left(1\ if\ n = 0, else\ 2 * (Y\ g)(n-1)\right)\right) 2\ [\, r :=\ Y\ g]$
$= \left(1\ if\ 2 = 0, else\ 2 * (Y\ g)(2-1)\right)[\, n :=\ 2\ ]$
$= (2 * (Y\ g))\ 1\ = 2$

5.1 Definition: Measure clause

Implementation:

```
factorial::Integer -> Integer
factorial 0 = 1
factorial x = x * factorial (x - 1)
```

5.2 Polymorphic function

We can create generic functions that can be used on values of several different types (Denote @ for parameter)

Bag (multiset), example: {2,2,2,3,5} = {2:3, 3:1, 5:1}

num_bag[@elem] : @elem * (map @elem to nat1) +> nat

num_bag(e, m) == m(e)

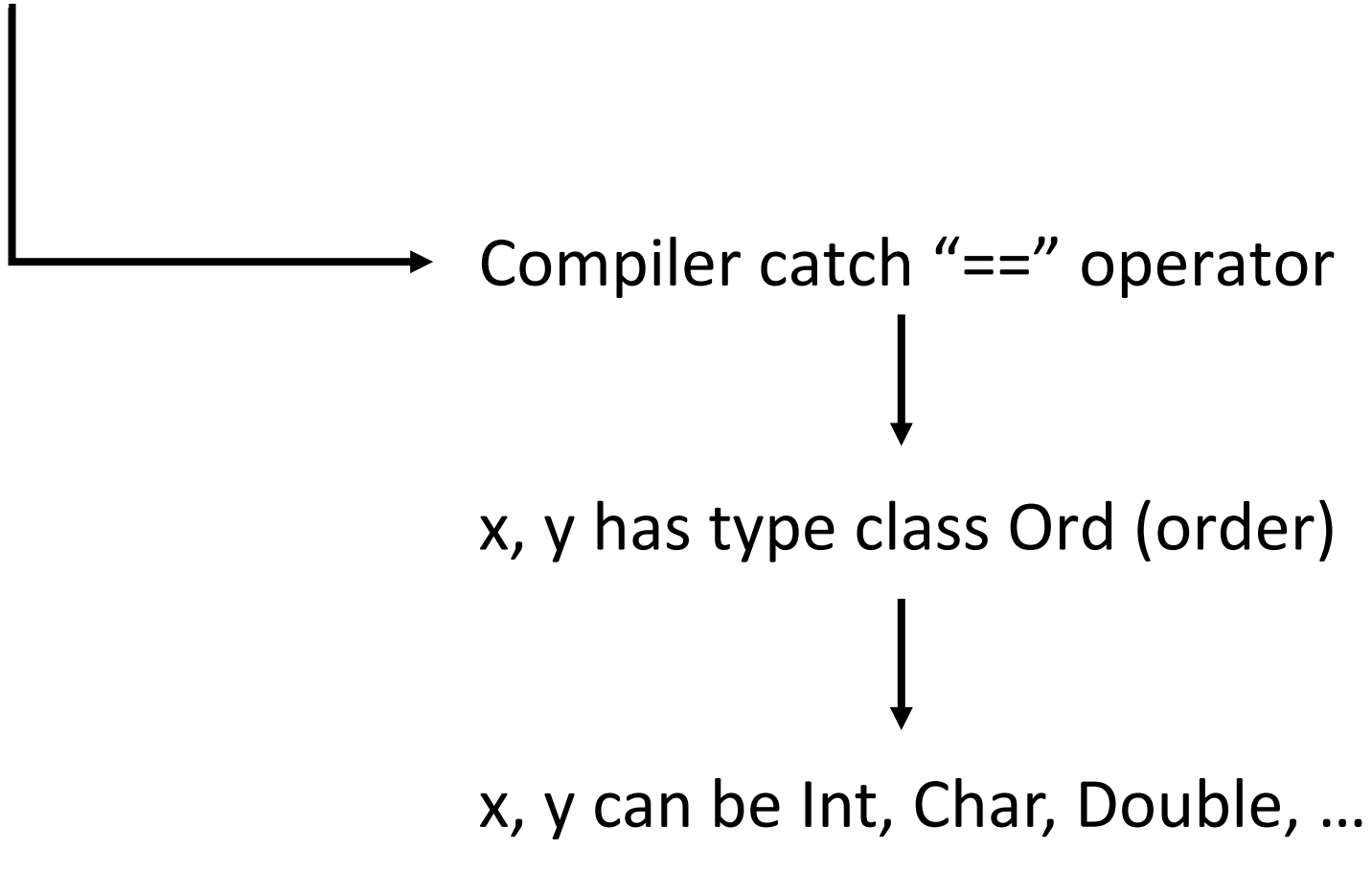
pre e in set dom m

plus_bag[@elem] : @elem * (map @elem to nat1) +> (map @elem to nat1)

plus_bag(e, m) == m ++ { e |-> num_bag[@elem](e, m) + 1 }.

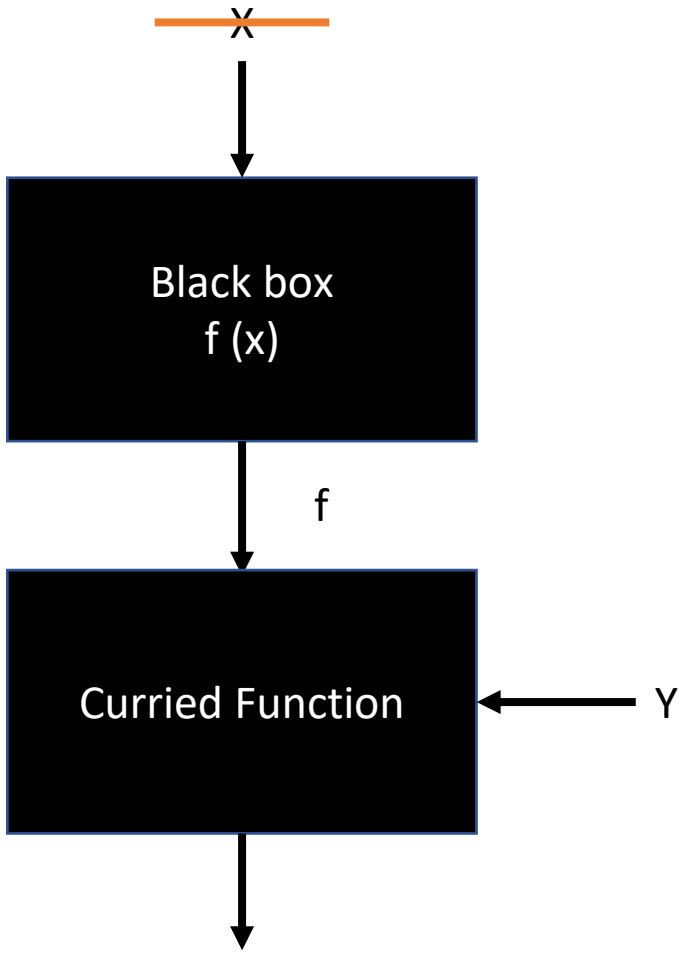
5.2 Polymorphic function: Example

isEqual x y = x == y



5.3 Higher Order function

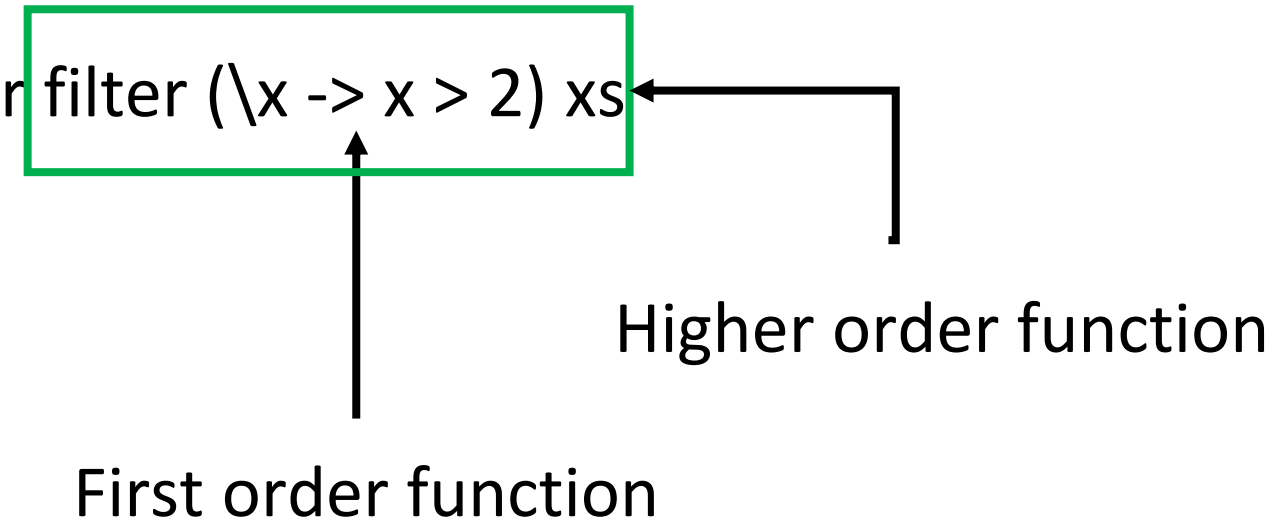
Functions are allowed to receive other functions as arguments.



In math: $\lambda f(x)y.xy = \lambda f(x).\lambda y.xy$

5.3 Higher Order function

nat_filter : (nat -> bool) * seq of nat -> seq of nat
nat_filter (p,ns) == [n | n in seq ns & p(n)];
Example: filter (p > 2)
In math: $\{x \mid x \in XS \wedge x > 2\}$
In Haskell: $[\, x \mid x \leftarrow xs, x > 2]$ or filter (\x -> x > 2) xs



7. Patterns

7.1 Ngữ nghĩa

- Được sử dụng để kiểm tra việc khớp tới 1 giá trị có kiểu dữ liệu bất kỳ
    * Kiểm tra xem giá trị có khớp với 1 hoặc nhiều pattern
    * Binding các định danh trong pattern tới các giá trị tương ứng
- Cú pháp matching như sau:

```
cases expression:
    pattern_1 → value_1,
    pattern_2 → value_2,
    …
    others    → value_n
end;
```

7.2 Các loại pattern

1. Định danh: sử dụng định danh để khớp tới 1 giá trị cụ thể bất kỳ.

Đây là dạng đơn giản nhất của pattern. Ví dụ:

```
let top = GroupA(1)
in top.sc
```

2. Match value: khớp trực tiếp tới giá trị cụ thể không thông qua định danh.

- **Literal:** 1 giá trị thuộc kiểu dữ liệu cơ bản. Ví dụ: “Màu đỏ”, 7, True, …
- **Expresson:** là 1 biểu thức xác định. Khi sử dụng biểu thức này được đặt trong cặp dấu ()

Ví dụ:

```
HocSinh ::
        Ten : string
        Lop: string
       Diem: float
Hs_A = mk-HocSinh(“Nguyễn Văn A”, “10A1”, 9.5)
let lop10a2 = “10A2”
in   cases Hs_A.Lop:
       “10A1”      → “Học sinh lớp 10A1”,
       (lop10a2)  → “Học sinh lớp 10A2
     others          → “Học sinh không thuộc lớp nào”
     end;
```

3. Set enumeration pattern:

- Khớp trực tiếp tới giá trị của tập
- Tất cả phần tử trong tập hợp phải được khớp. Nếu số lượng phần tử khớp khác số lượng phần tử trong trong tập thì pattern không hợp lệ

* Pattern:

- {}: Khớp tập rỗng
- {pt1}: Khớp tới tập chứa 1 phần tử và lấy ra phần tử đó
- {pt1, pt2, … ptn}: Khớp tập chứa **n** phần tử và lấy ra **n** phần tử đó

* Ví dụ:

```
Group_HocSinh = [
    mk_HocSinh(“Nguyen Van A”, “10A1”, 9.5),
    mk_HocSinh(“Nguyen Van A”, “10A2”, 6) ]
let {hs1, hs2} = elems Group_HocSinh
in hs1.diem + hs2.diem
```

4. Set union pattern:

- Khớp 2 tập hợp con sao cho 2 tập hợp hợp nhau tạo thành tập hợp cần xét
- 2 tập hợp này luôn không rỗng và không giao nhau

*** Pattern:**

s1 **union** s2 : Set là tập hợp tạo từ 2 set **s1** và **s2**

*** Ví dụ:**

set2seq[@elem]: **set of** @elem → **seq of** @elem

set2seq(s) ==

```
cases s:
    {} → [],
    {x} → [x],
     s1 union s2 → set2seq(s1) ^ set2seq(s2)
end;
```

5. Sequence enumeration pattern:

- Khớp tới các phần tử của mảng

- Số lượng phần tử khớp phải bằng số lượng phần tử của mảng. Nếu không pattern không hợp lệ

*** Pattern:**

[]: Mảng rỗng

[x]: Lấy ra mảng chứa phần tử đầu tiên

[x_1, x_2, …, x_n]: Lấy ra **n** phần tử của mảng

6. Sequence concatenation pattern:

- Khớp 2 pattern tương ứng với 2 mảng con sao cho khi nối 2 mảng này thì được mảng cần xét

- Các mảng trong pattern đều không rỗng.

*** Pattern:**

- ^ [x] **^ -** : Mảng có ít nhất 3 phần tử

[x] ^ -: Mảng có ít nhất 2 phần tử và lấy ra phần tử đầu tiên

- ^ [x]: Mảng có ít nhất 2 phần tử và lấy ra phần tử cuối cùng

* Ví dụ: Tính tổng các phần tử của 1 mảng

SumOfSeq: seq of nat → nat

SumOfSeq(s) ==

cases s:

  [] → 0,

  [x] → x,

  [x] ^ seq → x + **SumOfSeq**(seq)

end;

7. Map enumeration pattern:

- Được sử dụng để khớp tới giá trị của map

- Gía trị khớp có thể nằm trong tập nguồn hoặc tập đích hoặc cả 2

*** Pattern**

{ định_danh **|->** giá_trị_đích } = { giá_trị_nguồn **|->** giá_tri_đích }

{ giá_trị_nguồn **|->** định_danh } = { giá_trị_nguồn **|->** giá_tri_đích }

{ định danh_1 **|->** định_danh_2 } = { giá_trị_nguồn **|->** giá_tri_đích }

*** Ví dụ:**

```
let {a |-> b} = {1 |-> 2} in mk_(a,b) = mk_(1,2)
```

7. Maplet pattern list:

- Được sử dụng để khớp với các ánh xạ bên trong map

- Toàn bộ các phần tử đều phải khớp. Nếu không thì pattern không hợp lệ

*** Pattern:**

{**|->** }: Khớp tới map rỗng

{- **|->** -}: Khớp tới map có 1 phần tử

{ mep_1, mep_2, …, mep_n } = { ánh_xạ_1, ánh_xạ_2, … ánh_xạ_n }

*** Ví dụ:**

```
let {1 |-> a,a |-> b,b |-> c} = {1 |-> 4,2 |-> 3,4 |-> 2} in
c = 3
```

8. Map union list:

- Khớp dựa trên 2 pattern tương ứng với 2 map con của map cần xét.

- 2 map con này luôn không rỗng và không giao nhau.

* Pattern

m1 **munion** m2: map là hội của m1 và m2 và map có ít nhất 2 ánh xạ

* Ví du:

```
public map2seq[@T1, @T2] :
    map @T1 to @T2 -> seq of (map @T1 to @T2)
map2seq(m) ==
  cases m:
    ({|->})      -> [],
    {- |-> -}    -> [m],
    m1 munion m2 ->
        map2seq[@T1, @T2] (m1) ^  map2seq[@T1, @T2] (m2)
 end;
```

7.1 Object Pattern

CÚ PHÁP:

object-pattern = ‘**obj_** ’, *identifier*, ‘**(**’, [*field-pattern-list*], ‘**)**’ ;

field-pattern-list = *field-pattern*, { ‘**,**’, *field-pattern* } ;

field-pattern = *identifier*, ‘**|->**’, *pattern* ;

7.1 Object Pattern

NGỮ NGHĨA:

- Một mẫu đối tượng **(object pattern)** khớp với các đối tượng tham chiếu **(object references)**;
- Một **đối tượng** được khớp (matched) với **định danh của một lớp** (class identified), được thể hiện bằng **tên lớp** với tiền tố **obj_**;

  Ví dụ: **obj_Student**
- Các **biến thể hiện** (instance variables) có tên trong **mẫu đối tượng** được khớp với đối tượng đó từ trái sang phải**;**

  Ví dụ: **obj_Student(name |-> “John”, age |-> 20)**

7.1 Object Pattern

Lưu ý:

- Các **biến thực thể private** chỉ có thể được so khớp bên trong các hoạt động của cùng một lớp**.**
- **Giá trị của biến thể hiện tham chiếu** không đảm bảo không thay đổi trong khi so khớp (matching).

7.1 Object Pattern

MỘT SỐ VÍ DỤ

Ví dụ 1. Khớp tất cả các thể hiện của lớp Student:

```
cases person:
    obj_Student() -> <STUDENT>,
    others -> <NOT_STUDENT>
end
```

Trong đó:

object pattern = obj_Student();

field pattern list = Ø

7.1 Object Pattern

MỘT SỐ VÍ DỤ

Ví dụ 2. Khớp thể hiện nào đó của lớp Student mà có giá trị biến thể hiện name là John:

```
cases person:
   obj_Student(name |-> "John") -> <JOHN>,
   others -> <NOT_A_STUDENT_OR_NOT_JOHN>
end
```

Trong đó:

object pattern = obj_Student(name |-> “John”);

field pattern list = name |-> “John”

7.1 Object Pattern

MỘT SỐ VÍ DỤ

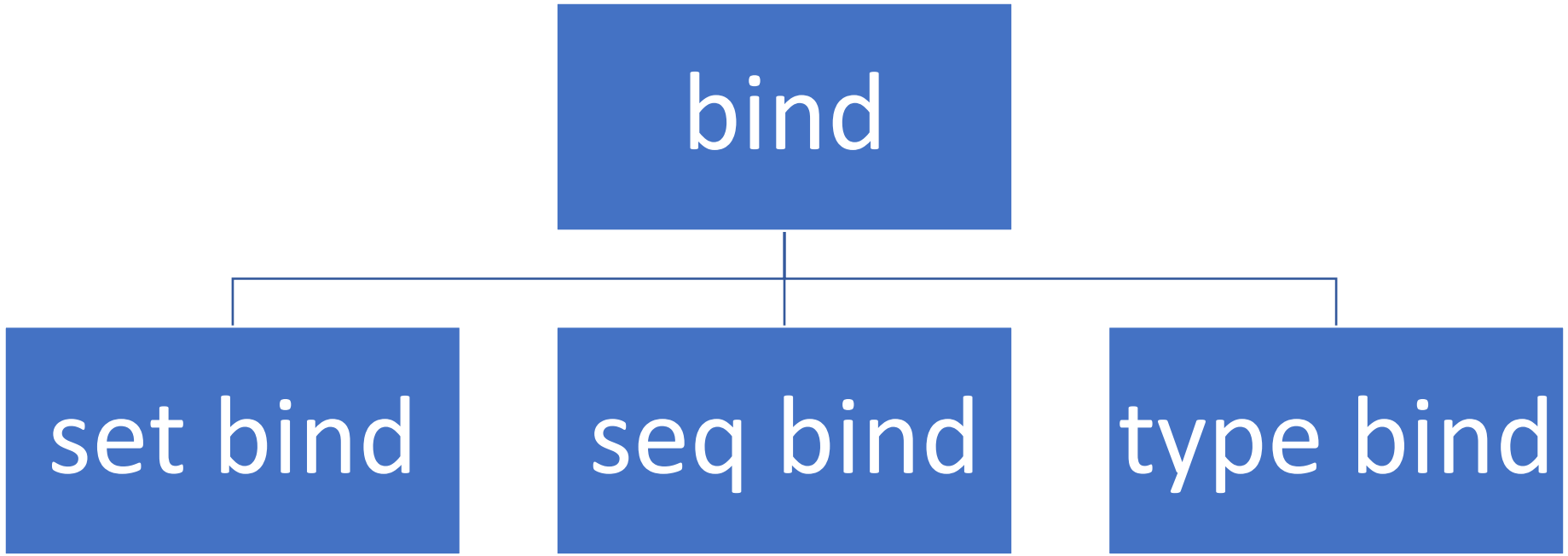
Ví dụ 3: Object pattern có thể chứa các object pattern khác.

```
cases person:
   obj_Student
      (name |-> n, dept |-> obj_Department(name->dname))
         -> n^"@"^dname,
   obj_Professor(name |-> n) -> "Prof. "^n,
   others -> ""
end
```

8. Binding

8. Binding

Bind: một *bind* dùng để nối một pattern (mẫu) với một value (giá trị)



8. Binding

Bind: một *bind* dùng để nối một pattern (mẫu) với một value (giá trị)

- Set bind: value sẽ được chọn từ một set defined bằng một set expression của bind
- Syntax: pattern, ‘in set’, expression;
- Ví dụ:

    let a in set {1,2,3,4,5} | a>5

8. Binding

Bind: một *bind* dùng để nối một pattern (mẫu) với một value (giá trị)

- Seq bind: value sẽ được chọn từ một sequence defined bằng một sequence expression của bind
- Syntax: pattern, ‘in seq’, expression;
- Ví dụ:

$$forall\ nt\ in\ seq\ \mathbb{N} \mid nguyên{-}tố(nt)$$

8. Binding

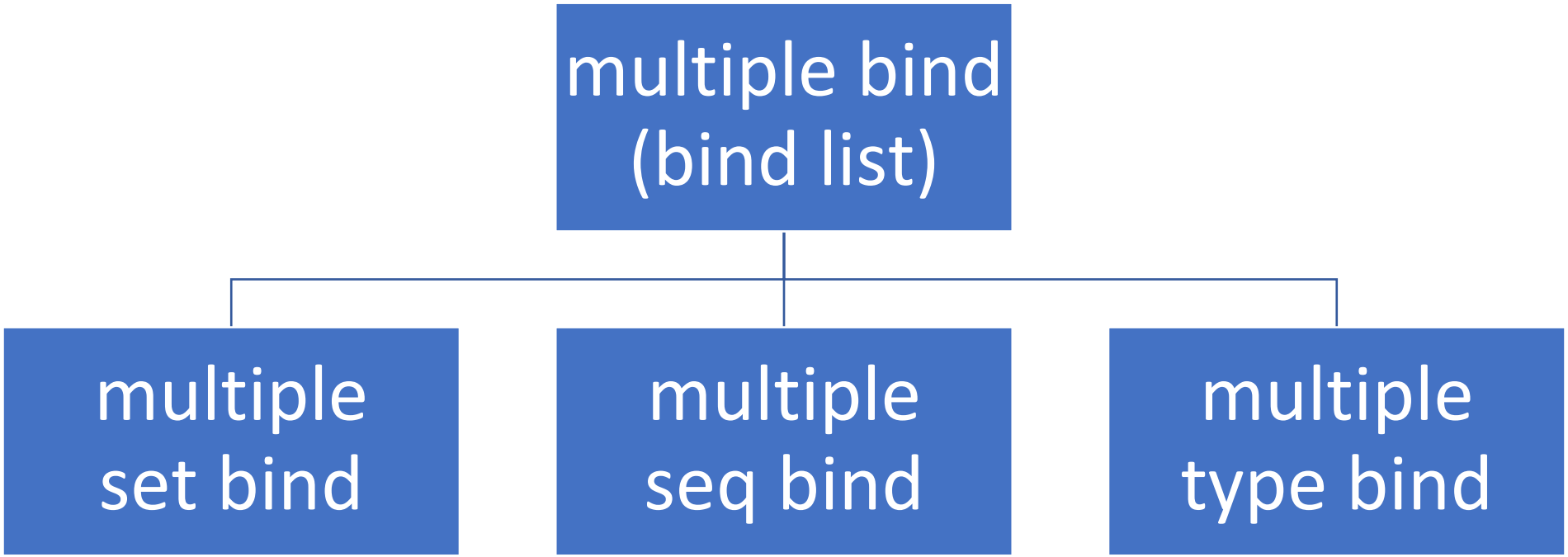
Bind: một *bind* dùng để nối một pattern (mẫu) với một value (giá trị)

- Type bind: value sẽ được chọn từ một type defined bằng một type expression của bind
- Syntax: pattern, ‘:’ , type;
- Ví dụ:

$$n: \mathbb{N}, \mathbb{N}-set$$

8. Binding

Bind list: *bind list* giống hệt *bind* trừ việc sẽ có nhiều pattern chung set, sequence hoặc type



8. Binding

Bind list: *bind list* giống hệt *bind* trừ việc sẽ có nhiều pattern chung set, sequence hoặc type

- Multiple set bind: values sẽ được chọn từ một set defined bằng một set expression của bind
- Syntax: pattern list, ‘in set’, expression;
- Ví dụ:

    let a, b in set {1,2,3,4,5} | a>5

8. Binding

Bind list: *bind list* giống hệt *bind* trừ việc sẽ có nhiều pattern chung set, sequence hoặc type

- Multiple seq bind: values sẽ được chọn từ một sequence defined bằng một sequence expression của bind
- Syntax: pattern list, ‘in seq’, expression;
- Ví dụ:

$$forall\ nt_1, nt_2\ in\ seq\ \mathbb{N} \mid nguyên{-}tố(nt_1) \wedge nguyên{-}tố(nt_2)$$

8. Binding

Bind list: *bind list* giống hệt *bind* trừ việc sẽ có nhiều pattern chung set, sequence hoặc type

- Multiple type bind: values sẽ được chọn từ một type defined bằng một type expression của bind
- Syntax: pattern list, ‘:’ , type;
- Ví dụ:

$$n, m: \mathbb{N}, \mathbb{N}-set$$

9. Value (Constant) definition

9. Value (Constant) definition

- Value (constant) dùng để định nghĩa của một giá trị tương tự như bao ngôn ngữ lập trình truyền thống khác
- Syntax: '**values**', [access value definition], {';', access value definition},[';']]
  Trong đó: access value definition = [access], value definition;
  value definition = pattern, [':', type], '=', expression
- Ví dụ:

```
values
    public id = -1
    public name = s : string = getname(id)
```