

CSE216

Foundations of Computer Science

Instructor: Zhoulai Fu

State University of New York, Korea

Many slides taken from Cornell's CS3110. Thanks!
https://www.cs.cornell.edu/courses/cs3110/2014fa/lecture_notes.php

Plan

- Syntax Sugar (heading for higher-order functions)

Pattern matching in details

Match expressions

- **Syntax**

```
match e with p1 -> e1 | p2 -> e2 | ... | pn -> en
```

- **Evaluation:**

- Evaluate **e** to a value **v**
- If **p_i** is the first pattern to match **v**, then evaluate **e_i** to value **v_i** and return **v_i**
 - Note: pattern itself is **not** evaluated
- Pattern *matches* value if it “looks like” the value
 - Pattern **C_i (x₁, ..., x_n)** matches value **C_i (v₁, ..., v_n)**
 - *Wildcard* pattern **_** (i.e., underscore) matches any value

Matching can go deep

- Patterns can nest arbitrarily deep
 - (Just like expressions)
 - Easy-to-read, nested patterns can replace hard-to-read, nested **match** expressions
- Examples:
 - Pattern **$a :: b :: c :: d$** matches all lists with ≥ 3 elements
 - Pattern **$a :: b :: c :: []$** matches all lists with 3 elements
 - Pattern **$((a, b), (c, d)) :: e$** matches all non-empty lists of pairs of pairs

Exercise

- Write zip3 that zips 3 lists
 - Zip3 ["hello"; "world"] [1;2] [0.1;0.2] becomes [("hello", 1, 0.1); ("world", 2, 0.2)]
 - Zip3 ["hello"] [1;2] [0.1] should fail

example: zip 3 lists

```
let rec zip3 lists =  
  match lists with  
  ([], [], []) -> []  
  
  | (hd1::tl1, hd2::tl2, hd3::tl3) ->  
    (hd1, hd2, hd3)::zip3(tl1, tl2, tl3)  
  
  | _ -> raise (Failure "List length  
mismatch")
```

Exercise: unzip 3-element list

```
unzip3 : ('a * 'b * 'c) list -> 'a list * 'b list * 'c list
```

```
# unzip3 [(1,2,3);(4,5,6);(7,8,9);(10,11,12)] ;;  
- : int list * int list * int list =  
([1; 4; 7; 10], [2; 5; 8; 11], [3; 6; 9; 12])
```


Precise Definitions of Pattern Matching

Given a pattern p and a value v , decide

- Does pattern match value?
- If so, what variable bindings are introduced?

Let's give an evaluation rule for each kind of pattern...

Precise Definitions of Pattern Matching (2)

- If p is a variable x , the match succeeds and x is bound to v
- If p is $_$, the match succeeds and no bindings are introduced
- If p is a constant c , the match succeeds if v is c . No bindings are introduced.

Precise Definitions of Pattern Matching (3)

- If p is C , the match succeeds if v is C . No bindings are introduced.
- If p is $C \ p1$, the match succeeds if v is $C \ v1$ (i.e., the same constructor) and $p1$ matches $v1$. The bindings are the bindings from the sub-match.

Precise Definitions of Pattern Matching (4)

- If \mathbf{p} is $(\mathbf{p1}, \dots, \mathbf{pn})$ and \mathbf{v} is $(\mathbf{v1}, \dots, \mathbf{vn})$, the match succeeds if $\mathbf{p1}$ matches $\mathbf{v1}$, and ..., and \mathbf{pn} matches \mathbf{vn} . The bindings are the union of all bindings from the sub-matches.
 - The pattern $(\mathbf{x1}, \dots, \mathbf{xn})$ matches the tuple value $(\mathbf{v1}, \dots, \mathbf{vn})$
- If \mathbf{p} is $\{\mathbf{f1}=\mathbf{p1}; \dots; \mathbf{fn}=\mathbf{pn}\}$ and \mathbf{v} is $\{\mathbf{f1}=\mathbf{v1}; \dots; \mathbf{fn}=\mathbf{vn}\}$, the match succeeds if $\mathbf{p1}$ matches $\mathbf{v1}$, and ..., and \mathbf{pn} matches \mathbf{vn} . The bindings are the union of all bindings from the sub-matches.
 - (and fields can be reordered)
 - The pattern $\{\mathbf{f1}=\mathbf{x1}; \dots; \mathbf{fn}=\mathbf{xn}\}$ matches the record value $\{\mathbf{f1}=\mathbf{v1}; \dots; \mathbf{fn}=\mathbf{vn}\}$

Exercise

- Using pattern matching, write three functions, one for each of the following properties. Your functions should return true if the input list has the property and false otherwise.
- 1. the list's first element is "hello"
- 2. the list has exactly two or four elements; do not use the length function
- 3. the first two elements of the list are equal



“A language that doesn't affect the way you think about programming is not worth knowing.”

–Alan J. Perlis (Turing Laureate)

1. If expressions are just matches

- **if** expressions exist only in the *surface syntax* of the language
- Early pass in compiler can actually replace **if** expression with **match** expression, then compile the **match** expression instead

```
if e0 then e1 else e2
```

becomes...

```
match e0 with true -> e1 | false -> e2
```

because...

```
type bool = false | true
```

2. Option is a built-in datatype

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

- **None** and **Some** are constructors
- **'a** means “any type”

```
let string_of_intopt(x:int option) =  
  match x with  
    None      -> ""  
  | Some(i)   -> string_of_int(i)
```


3. List is another builtin datatype

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

```
let rec append (xs: 'a list)(ys: 'a list) =
```

```
    match xs with
```

```
    [] -> ys
```

```
  | x::xs' -> x :: (append xs' ys)
```

- Ocaml uses [], ::, @ instead

4. Let expressions are pattern matches

- The syntax on the LHS of = in a let expression is really a pattern

```
let p = e
```

- (Variables are just one kind of pattern)

- Implies it's possible to do this (e.g.):

```
let [x1;x2] = lst
```

5. Function arguments are patterns

A function argument can also be a pattern

- Match against the argument in a function call

```
let f p = e
```

Examples:

```
let sum_triple (x, y, z) =  
  x + y + z
```

```
let sum_stooges {larry=x; moe=y; curly=z} =  
  x + y + z
```

Back to 'a (some textbooks call it alpha)

Length of a list:

```
let rec len (xs: int list) =  
  match xs with  
    [] -> 0  
  | _::xs' -> 1 + len xs'
```

```
let rec len (xs: string list) =  
  match xs with  
    [] -> 0  
  | _::xs' -> 1 + len xs'
```

No algorithmic difference! Would be silly to have to write function for every kind of list type...

Type variables ‘a to the rescue

Use *type variable* to stand in place of an arbitrary type:

```
let rec len (xs: 'a list) =  
  match xs with  
    [] -> 0  
  | _::xs' -> 1 + len xs'
```

- Just like we use *variables* to stand in place of arbitrary values
- Creates a *polymorphic* function (“poly”=many, “morph”=form)

Somewhat related to Java generics and C++ templates

BTW, Java generics looks like this

```
public class Box<T> {  
    private T content;  
  
    public void addContent(T content) {  
        this.content = content;  
    }  
  
    public T getContent() {  
        return content;  
    }  
  
    public static void main(String[] args) {  
        Box<String> box1 = new Box<>();  
        box1.addContent("Hello, world!");  
        System.out.println(box1.getContent()); // Output: Hello, world!  
  
        Box<Integer> box2 = new Box<>();  
        box2.addContent(42);  
        System.out.println(box2.getContent()); // Output: 42  
    }  
}
```

Extended datatype syntax

```
type 'a t = C1 [of t1] | ...  
| Cn [of tn]
```

- $t_1..t_n$ are now allowed to mention t and $'a$

Exercise

- Define a Polymorphic Binary Tree Type in OCaml, called **binary_tree**
 - support storing elements of any type.
 - Use constructor Empty to create an empty binary tree
 - Use constructor Node to create a node in the tree that carries data

```
let mytree = Node (2, Node (1, Empty, Empty), Node (3, Empty, Empty))
```

- Define a function “height” to get a tree’s height; “mytree” above has height 2.

Exercise

- Write a function *drop* : *int* -> 'a list -> 'a list such that *drop n lst* returns all but the first *n* elements of *lst*. If *lst* has fewer than *n* elements, return the empty list. Here, *n* can be any integer including negative number.