Chapter 7
A Second Look At ML

Two Patterns You Already Know

- We have seen that ML functions take a single parameter:

```
fun f n = n*n;
```

- We have also seen how to specify functions with more than one input by using tuples:

```
fun f (a, b) = a*b;
```

- Both **`n`** and **`(a, b)`** are *patterns*. The **`n`** matches and binds to any argument, while **`(a,b)`** matches any tuple of two items and binds **`a`** and **`b`** to its components

Underscore as a Pattern

```
- fun f _ = "yes";
val f = fn : 'a -> string
- f 34.5;
val it = "yes" : string
- f [];
val it = "yes" : string
```

- The underscore can be used as a pattern
- It matches anything, but does not bind it to a variable
- Could have been defined as:
  `fun f x = "yes";`

3

Underscore as a Pattern

```
- fun f x = "yes";
val f = fn : 'a -> string
- x;
stdIn:42.1 Error: unbound variable or
constructor: x
- fun f x = x+1;
val f = fn : int -> int
```

- This would introduce an unused variable x. In ML, as in most languages, you should avoid introducing variables if you don't intend to use them.

4

Constants as Patterns

```
- fun f 0 = "yes";
Warning: match nonexhaustive
          0 => ...
val f = fn : int -> string
- f 0;
val it = "yes" : string
```

- Function **`f`** only works if its parameter is the integer constant 0.
- Any constant of an **equality type** can be used as a pattern, except for:
  `fun f 0.0 = "yes";`

Non-Exhaustive Match

- In that last example, the type of **`f`** was **`int -> string`**, but with a "match non-exhaustive" warning
- Meaning: **`f`** was defined using a pattern that didn't cover all the domain type (**`int`**)
- So you may get runtime errors like this:

```
- f 0;
val it = "yes" : string
- f 1;
uncaught exception nonexhaustive match failure
```

Lists of Patterns as Patterns

```
- fun f [a,_] = a;
Warning: match nonexhaustive
          a :: _ :: nil => ...
val f = fn : 'a list -> 'a
- f [#"f",#"g"];
val it = #"f" : char
```

- You can use a list of patterns as a pattern
- This example matches any list of length 2
- It treats **`a`** and **`_`** as sub-patterns, binding **`a`** to the first list element

7

Cons of Patterns as a Pattern

```
- fun f (x::xs) = x;
Warning: match nonexhaustive
          x :: xs => ...
val f = fn : 'a list -> 'a
- f [1,2,3];
val it = 1 : int
```

- You can use a cons of patterns as a pattern
- **`x::xs`** matches any non-empty list, and binds **`x`** to the head and **`xs`** to the tail
- Parentheses around **`x::xs`** are for precedence

8

ML Patterns So Far

- A variable is a pattern that matches anything, and binds to it
- A _ is a pattern that matches anything
- A constant (of an equality type) is a pattern that matches only that constant
- A tuple of patterns is a pattern that matches any tuple of the right size, whose contents match the sub-patterns
- A list of patterns is a pattern that matches any list of the right size, whose contents match the sub-patterns
- A cons (::) of patterns is a pattern that matches any non-empty list whose head and tail match the sub-patterns

Multiple Patterns for Functions

```
- fun f 0 = "zero"
= |   f 1 = "one";
Warning: match nonexhaustive
          0 => ...
          1 => ...
val f = fn : int -> string;
- f 1;
val it = "one" : string
```

- You can define a function by listing alternate patterns

Syntax for Function Definition

- A function definition contains one or more function bodies separated by the `'|'` token.

```
<fun-def>    ::= fun <fun-bodies> ;
<fun-bodies>  ::= <fun-body>
                | <fun-body> '|' <fun-bodies>
<fun-body>   ::= <fun-name> <pattern> = <expression>
```

- To list alternate patterns for a function
- You must repeat the function name (f) in each alternative

Overlapping Patterns

```
- fun f 0 = "zero"
= |   f _ = "non-zero";
val f = fn : int -> string;
- f 0;
val it = "zero" : string
- f 34;
val it = "non-zero" : string
```

- Patterns may overlap
- ML uses the first match for a given argument

Pattern-Matching Style

- These definitions are equivalent:

```
fun f 0 = "zero"
|   f _ = "non-zero";

fun f n =
  if n = 0 then "zero"
  else "non-zero";
```

- But the pattern-matching style is usually preferred in ML
- It often gives shorter and more legible functions

13

Pattern-Matching Example

- Original (from Chapter 5):

```
fun fact n =
  if n = 0 then 1 else n * fact(n-1);
```

- Rewritten using patterns:

```
fun fact 0 = 1
|   fact n = n * fact(n-1);
```

14

Pattern-Matching Example

- Original (from Chapter 5):

```
fun reverse L =
   if null L then nil
   else reverse(tl L) @ [hd L];
```

- Improved using patterns:

```
fun reverse nil = nil
|   reverse (first::rest) =
       reverse rest @ [first];
```

15

More Examples

- This structure occurs frequently in recursive functions that operate on lists: one alternative for the base case (**`nil`**) and one alternative for the recursive case (**`first::rest`**).
- Adding up all the elements of a list:

```
fun f nil = 0
|   f (first::rest) = first + f rest;
```

- Counting the true values in a list:

```
fun f nil = 0
|   f (true::rest) = 1 + f rest
|   f (false::rest) = f rest;
```

16

More Examples

- Making a new list of integers in which each is one greater than in the original list:

```
fun f nil = nil
|   f (first::rest) = first+1 :: f rest;
```

```
- fun f nil = nil
= |   f (first::rest) = first+1 :: f rest;
val f = fn : int list -> int list
- f [1,2,3];
val it = [2,3,4] : int list
```

17

More Examples

- Defining a function **`greater`** of type **`int * int list -> int list`** so that **`greater(e,L)`** is a list of all the integers in **`L`** that are greater than **`e`**:

```
fun greater (e,nil) = nil
|   greater (e,x::xs) =
      if x > e then x::greater (e,xs)
      else greater (e,xs);
```

18

A Restriction

- You can't use the same variable more than once in the same pattern
- This is not legal:

```
fun f (a,a) = … for pairs of equal elements
|   f (a,b) = … for pairs of unequal elements
```

- You must use this instead:

```
fun f (a,b) =
  if (a=b) then … for pairs of equal elements
  else … for pairs of unequal elements
```

19

Scope

Patterns Everywhere

```
- val (a,b) = (1,2.3);
val a = 1 : int
val b = 2.3 : real
- val a::b = [1,2,3,4,5];
Warning: binding not exhaustive
          a :: b = ...
val a = 1 : int
val b = [2,3,4,5] : int list
```

This does not cover all lists: empty list is not covered.

- Patterns are not just for function definition
- Here we see that you can use them in a **`val`** (**`val`** definitions)

20

Local Variable Definitions

- When you use **`val`** at the top level to define a variable, it is visible from that point forward
- There is a way to restrict the scope of definitions: the **`let`** expression

<let-exp> `::=` **`let`** *<definitions>* **`in`** *<expression>* **`end`**

21

Example with **`let`**

let只在本语句、表达式中起作用.

```
- let val x = 1 val y = 2 in x+y end;
val it = 3 : int;
- x;
Error: unbound variable or constructor: x
```

- The value of a **`let`** expression is the value of the expression in the **`in`** part
- Variables defined with **`val`** between the **`let`** and the **`in`** are visible only from the point of declaration up to the **`end`**

22

多行缩进.

Proper Indentation for **let**

```
let
  val x = 1
  val y = 2
in
  x+y
end;
```

- For readability, use multiple lines and indent **let** expressions like this
- Some ML programmers put a semicolon 分号 after each **val** declaration in a **let**

colon

X<=> to

Long Expressions with **let**

days to ms

```
fun days2ms days =
  let
    val hours = days * 24.0
    val minutes = hours * 60.0
    val seconds = minutes * 60.0
  in
    seconds * 1000.0
  end;
```

- The **let** expression allows you to break up long expressions and name the pieces
- This can make code more readable

什么b东西。

Patterns with `let`

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```

① 1,2,[3,4,5,6).
② 1,2,3,4[5,6].
① a=1 , b=2
② a=1, a=3
b=2, b=4

The **val** defines **x** and **y** by pattern matching.

模式

- By using patterns in the declarations of a **let**, you can get easy "deconstruction"
- This example takes a list argument and returns a pair of lists, with half in each

25

Again, without Good Patterns

```
let
  val (x, y) = halve cs
```

- The recursive call to **halve** returns a pair of lists, and the **val** definition binds **x** to the first element of the pair and **y** to the second.
- The **let** expression could have been written as follows:

```
let
  val halved = halve cs
  val x = #1 halved
  val y = #2 halved
in
  (a::x, b::y)
end;
```

26

`halve` at Work

```
- fun halve nil = (nil, nil)
= |   halve [a] = ([a], nil)
= |   halve (a::b::cs) =
=       let
=         val (x, y) = halve cs
=       in
=         (a::x, b::y)
=       end;
val halve = fn : 'a list -> 'a list * 'a list
- halve [1];
val it = ([1],[]) : int list * int list
- halve [1,2];
val it = ([1],[2]) : int list * int list
- halve [1,2,3,4,5,6];
val it = ([1,3,5],[2,4,6]) : int list * int list
```


A Sort Example: Merge Sort 合并序列.

- The **`halve`** function divides a list into two nearly-equal parts
- This is the first step in a merge sort
- For practice, we will look at the rest

Example: Merge

```
fun merge (nil, ys) = ys
|   merge (xs, nil) = xs
|   merge (x::xs, y::ys) =
      if (x < y) then x :: merge(xs, y::ys)
      else y :: merge(x::xs, ys);
```

- Merges two sorted lists 合并两个排序之序列
- Note: default type for **<** is **`int`**

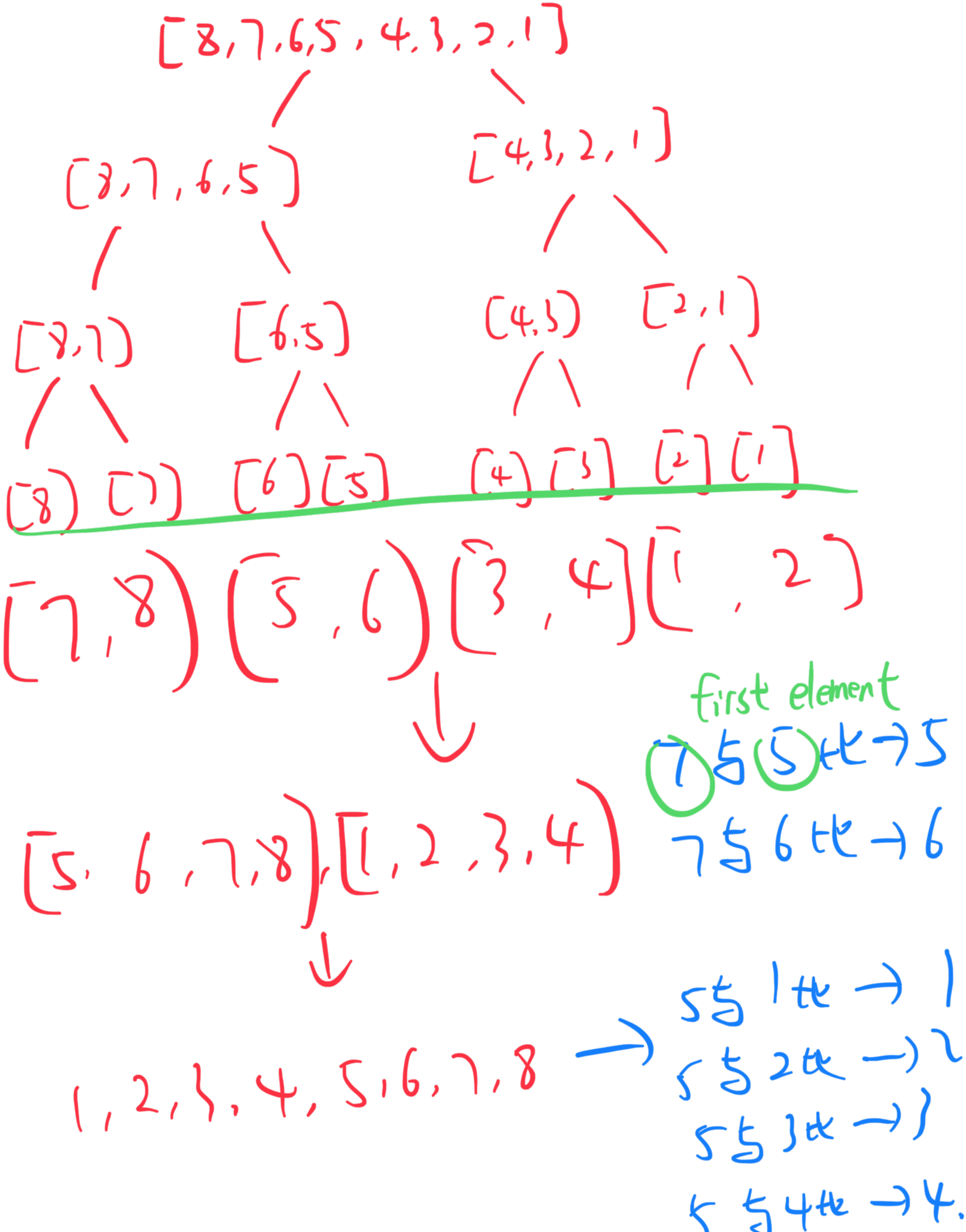
两个排好序之列表达合并的排序

29

Merge at Work

```
- fun merge (nil, ys) = ys
= |   merge (xs, nil) = xs
= |   merge (x::xs, y::ys) =
=       if (x < y) then x :: merge(xs, y::ys)
=       else y :: merge(x::xs, ys);
val merge = fn : int list * int list -> int list
- merge ([2],[1,3]);
val it = [1,2,3] : int list
- merge ([1,3,4,7,8],[2,3,5,6,10]);
val it = [1,2,3,3,4,5,6,7,8,10] : int list
```

30


[8,7,6,5,4,3,2,1]
[8,7,6,5]
[4,3,2,1]
[8,7]
[6,5]
[4,3]
[2,1]
[8] [7] [6] [5] [4] [3] [2] [1]
[7,8] [5,6] [3,4] [1,2]
[5,6,7,8] [1,2,3,4]
1,2,3,4,5,6,7,8
first element
7与5比→5
7与6比→6
5与1比→1
5与2比→2
5与3比→3
5与4比→4.

Example: Merge Sort

把一个列表交替地分成两半，分别进行排序，再进行组合。重复以上操作

```
fun mergeSort nil = nil
|   mergeSort [a] = [a]
|   mergeSort theList =
      let
        val (x, y) = halve theList
      in
        merge(mergeSort x, mergeSort y)
      end;
```

halve + merge

- To merge sort a list of more than one element, halve the list into two halves, recursively sort the halves, then merge the two sorted halves.
- Type is **`int list -> int list`**, because of type already found for **`merge`**

31

Merge Sort at Work

```
- fun mergeSort nil = nil
= |   mergeSort [a] = [a]
= |   mergeSort theList =
=       let
=         val (x, y) = halve theList
=       in
=         merge(mergeSort x, mergeSort y)
=       end;
val mergeSort = fn : int list -> int list
- mergeSort [4,3,2,1];
val it = [1,2,3,4] : int list
- mergeSort [4,2,3,1,5,3,6];
val it = [1,2,3,3,4,5,6] : int list
```

32

Nested Function Definitions

嵌套函数定义

- You can define local functions, just like local variables, using a **`let`**
- You should do it for helper functions that you don't think will be useful by themselves
- We can hide **`halve`** and **`merge`** from the rest of the program this way
- Another potential advantage: inner function can refer to variables from outer one (as we will see in Chapter 12)

33

halve + merge 完整体

Nested Function Definitions

```
(* Sort a list of integers. *)
fun mergeSort nil = nil
|   mergeSort [e] = [e]
|   mergeSort theList =
      let
        (* From the given list make a pair of lists
         * (x,y), where half the elements of the
         * original are in x and half are in y. *)
        fun halve nil = (nil, nil)
        |   halve [a] = ([a], nil)
        |   halve (a::b::cs) =
              let
                val (x, y) = halve cs
              in
                (a::x, b::y)
              end;
```

continued…

34

Nested Function Definitions

```
  (* Merge two sorted lists of integers into
   * a single sorted list. *)
  fun merge (nil, ys) = ys
  |   merge (xs, nil) = xs
  |   merge (x::xs, y::ys) =
        if (x < y) then x :: merge(xs, y::ys)
        else y :: merge(x::xs, ys);

  val (x, y) = halve theList
in
  merge(mergeSort x, mergeSort y)
end;
```

35

Commenting

- Everything between `(*` and `*)` in ML is a comment
- You should (at least) comment every function definition, as in any language
  - what parameters does it expect
  - what function does it compute
  - how does it do it (if non-obvious)
  - etc.

36

Questions?