

# 函数式编程原理

## Lecture 2

# ML (Meta-Language) 语言

- 通用的函数式编程语言，流行的有Standard ML和OCaml
- 非纯函数式编程语言，允许副作用和指令式编程
- ML的特点
  - 模式匹配 (Pattern Matching)
  - 异常处理 (Exception Handling)
  - 类型引用 (Type Inference)
  - 多态性 (Polymorphism)
  - 递归数据结构 (Recursive Data Structure)
- ML的交互会话：循环的执行“输入-执行-输出”

# 主要内容

- 类型和表达式 (Types and expressions)
- 声明和作用域 (Declarations and scope)
- 模式和匹配 (Patterns and matching)
- ML中的等式 (Equality in ML)
- 规则说明 (Specifications)

# ML标准类型

- 基础类型(basic types)    unit, int, real, bool, string
- 表(lists)    int list, (int -> int) list
- 元组(tuples)    int \* int, int \* int \* real
- 函数(functions)    int -> int, real -> int \* int

所有对象都要有类型，不一定显式说明，但必须能静态推导(在编译时该类型能被编译器根据上下文推算出来)。

```
fun sum [ ] = 0
| sum (x::L) = x + sum(L);
```

```
fun sum' ([ ], a) = a
| sum' (x::L, a) = sum' (L, x+a);
```

# ML基础类型

- 单元(unit)      只包含一个元素，用空的括号表示  
    ( ) : unit
- 整型(int)      负号用 “~” 表示
- 浮点型(real)
- 布尔型(bool) true, false
- 字符串型(string)      双引号间的字符序列

# 表(lists)

- 包含相同类型元素的有限序列
- 表中元素用 “,” 分隔，整个表用[ ]括起来
- 元素可以重复出现，其顺序有意义
- 表的类型取决于表中元素类型（可以为任意类型，但需具有相同的类型）
- 表可以嵌套
- 表的基本函数：

:: (追加元素), @ (连接表), null (空表测试), hd(返回表头元素),  
tl(返回非空表的表尾), length(返回表长)

- `nil = []` (空表)
- `[] : int list, [] : bool list, .....`
- `[1, 3, 2, 1, 21+21] : int list`
- `[true, false, true] : bool list`
- `[[1],[2, 3]] : (int list) list`
- `1::[2, 3] = [1, 2, 3]`
- `[1, 2]@[3, 4] = [1, 2, 3, 4]`

# 元组(tuples)

- 包含任意类型数据元素的**定长**序列
- 类型表达式：每个元素的类型用\*间隔并排列在一起

如： `int * int, int * int * real`

- 圆括号中用逗号分隔的数据元素，允许嵌套

如： `(“张三”, “男”, 19, 1.75)`

`[((“赵”, “子昂”), 21, 1.81), ((“张”, “文艺”), 20, 1.69)]`

`((string * string) * int * real) list`

# 记录(record)

- 类似C中的结构类型，可以包含不同类型的元素
- 每个元素都有一个名字
- 记录的值和类型的写法都用{ }括起来

如： {First\_name="赵", Last\_name="子昂"}

## • 元组、列表、记录的异同点？

- 符 号： ( ) vs. [ ] vs. { }
- 元素类型： 可以不同 vs. 必须相同 vs. 可以不同
- 长 度： 定长 vs. 变长 vs. 变长



# 函数(functions)

- 以一定的规则将定义域上的值映射到值域上去
- 类型由它的定义域类型和值域类型共同描述
- $\rightarrow$ 表示定义域到值域的映射

fn: <定义域类型>  $\rightarrow$  <值域类型>

如: fun add(x, y) = x + y;

**fn: int \* int  $\rightarrow$  int**

# ML标准函数

- 标准布尔函数：not, andalso, orelse

如：not true;    true andalso false;    true orelse false;

- 标准算数运算函数：~, +, -, \*, div, /

如：6 \* 7;        3.0 \* 2.0;

- 运算符重载(operator)：把同一运算符作用在不同类型上。

- 重载运算符的两边必须为同一类型。

- 整数到实数的转换：real

- 实数到整数的转换：floor(下取整), ceil(上取整), round(四舍五入), trunc(忽略小数)

- 标准字符串函数：

- 把两个字符串合并成一个：^

- 返回字符串的长度：size

# 值(Values)

- 每个类型都有一个值的集合

For each type  $t$  there is a set of *values*

- 一个类型的表达式求值结果为该类型的一个值(或出错)

An expression of type  $t$  *evaluates to a value of type  $t$*  (or fails to terminate)

- **TYPE SAFETY**

- **严格类型检查, 具有well-typed特性**

- **int**                      ... *integers*

- **real**                     ... *real numbers*

- **int list**                ... *lists of integers*

- **int -> int**             ... *Functions from integers to integers*

***functions are values***

# 表达式求值

| expression      | Value: type                        |
|-----------------|------------------------------------|
| $(3+4)*6$       | 42: int                            |
| $(3.0+4.0)*6.0$ | 42.0: real                         |
| $(42, 5)$       | $(42, 5): \text{int} * \text{int}$ |

Standard ML of New Jersey [...]

- 225 + 193;

val it = 418 : int

利用数学推导:

$225 + 193 \Rightarrow 418$

# 函数求值

函数：以一定的规则将定义域上的值映射到值域上

原型：  $\text{fn} : \langle \text{定义域类型} \rangle \rightarrow \langle \text{值域类型} \rangle$

| expression                       | Value: type         |
|----------------------------------|---------------------|
| <b>fn</b> (x:int):int => x+1     | fn - : int -> int   |
| <b>fn</b> (x:real):real => x+1.0 | fn - : real -> real |
| Math.sin                         | fn - : real -> real |

**fn** (x:int, y:int) : int\*int => (x div y, x mod y)

type **int\*int** -> **int\*int**

(**fn** (x:int, y:int) : int\*int => (x div y, x mod y)) (42, 5)

type **int\*int**

evaluates to the value (8, 2)

# 声明(Declarations)

- 赋予某个对象一个名字，包括值、类型、签名、结构和函子

函数的声明：

fun <函数名> (<形式参数>) : <结果类型> = <函数体>

例： fun divmod(x:int, y:int) : int\*int = (x div y, x mod y)

值的声明：

val pi = 3.1415;

val (q:int, r:int) = divmod(42, 5);

采用静态绑定方式——重新声明不会损坏系统、库或程序

# 声明、类型和值

- 任意一个类型的表达式都可以进行求值操作  
An expression of type  $t$  can be *evaluated*
- 任意一个类型表达式求值的结果为该类型的一个值  
If it terminates, we get a *value* of type  $t$
- ML提供重新声明功能  
ML reports type and value
  - `val it = 3 : int`
  - `val it = fn - : int -> int`
- 声明将产生名字(变量)和值的绑定(结合)  
Declarations produce *bindings*
- 绑定具有静态作用域  
Bindings are *statically scoped*

# 声明的使用

声明函数：

check : int \* int -> bool

```
fun check(x:int, y:int):bool =  
let  
  val (q:int, r:int)= divmod(x, y)  
in  
  (x = q*y + r)  
end
```

局部声明

```
val pi : real = 3.14;  
fun square(r:real) : real = r * r;  
fun area(r:real) : real = pi * square(r);  
  
val pi : real = 3.14159;  
fun area(r:real) : real = pi * square(r);
```

全局声明



# 声明的作用域

函数的两种定义方法：

**fun** circ(r:real):real = 2.0 \* pi \* r;

**fun** circ(r:real):real =  
**let**  
    **val** pi2:real = 2.0 \* pi  
**in**  
    pi2 \* r  
**end**

局部声明：  
    let D in E end

**local**  
    **val** pi2:real = 2.0 \* pi  
**in**  
    **fun** circ(r:real):real = pi2 \* r  
**end**

隐藏声明(一般很少使用):  
    local D1 in D2 end

# 模式(Patterns)

- 只包含变量、构造子(数值、字符、元组、表等)和通配符的表达式
  - 模式不是构造子的名字，是变量
  - 模式中的变量必须彼此不同
  - 构造子必须和变量区分开来
- 通配符: `_`
- 变量 : `x`
- 常数 : `42, true, ~3` // 实数和函数没有常数模式
- 元组 : `(p1, ..., pk)` // `p1, ..., pk`均为模式
- 表 : `nil, p1::p2, [p1, ..., pk]`

`(a,b)=(11,"hello")`

# ML =

- “=” 用于类型的等式判断，称为等式类型(“*equality types*”)

ML only permits use of = on types with an exact equality test

Such types are called *equality types*

- 等式类型包括整数、布尔值结合元组、表等构造子生成的类型

Equality types include all types built from **int**, **bool** using tuple and list constructors

如：

- int list
- int \* bool \* int
- (int \* bool \* int) list

# 模式匹配

- 模式与值进行匹配
  - 如果匹配成功，将产生一个绑定(bindings)
  - 如果匹配不成功，声明就会失败(抛出异常)

- 判断下列模式匹配的结果：

$d::L$  和  $[2,4]$

42 和 42 (value)

变量  $x$  和 任意值  $v$

$p1::p2$  和  $[]$

$d::L$  和  $[]$

42 和 0 (value)

$\_$  和 任意值  $v$

$p1::p2$  和  $v1::v2$

# 约定

- **【 $x_1:v_1, \dots, x_k:v_k$ 】** : 表示值绑定(value bindings)的集合
  - $x, x_1, \dots$  : 表示变量 (Variables)
  - $v, v_1, \dots$  : 表示值 ((syntactic) **V**alues)
  - $e, e_1, \dots$  : 表示表达式 (**E**xpressions)
  - $t, t_1, \dots$  : 表示类型 (**T**ypes)
- 可终止状态(Termination):  
$$e \downarrow \quad \text{when } \exists v. e \Rightarrow^* v$$
- 不可终止状态(Non-termination):  
$$e \uparrow$$
- **TYPE SAFETY**
- **严格类型检查, well-typed特性**

# 求值符号的使用

- $e \Rightarrow e'$  一次推导
- $e \Rightarrow^* e'$  有限次推导
- $e \Rightarrow^+ e'$  至少一次推导
- 如: `fun f(x:int):int = f x`

$f\ 0 \Rightarrow^+ (fn\ x \Rightarrow f\ x)\ 0$   
 $\Rightarrow^* [x:0]\ f\ x$   
 $\Rightarrow^* f\ 0$

因此:  $f\ 0 \Rightarrow^+ f\ 0$   
 $(f\ 0) \uparrow$

- $\Rightarrow$ 和 $\Rightarrow^*$ 可以精确反映程序行为
- 某些时候,计算顺序可以忽略,如

For all  $e_1, e_2 : \text{int}$  and all  $v:\text{int}$   
if  $e_1 + e_2 \Rightarrow^* v$  then  $e_2 + e_1 \Rightarrow^* v$

此时,我们关注计算结果多于计算过程

# 模式匹配举例：eval

```
fun eval ([ ]:int list):int = 0  
  | eval (d::L) = d + 10 * (eval L);
```

- 函数eval的类型定义？
- 该函数定义使用了表模式，参数[ ]、d::L匹配的结果是什么？
- 模式eval[2,4]匹配的值是多少？给出匹配/推导过程。

# 模式匹配举例: eval

```
fun eval ([ ]:int list):int = 0  
  | eval (d::L) = d + 10 * (eval L);
```

```
eval [2,4] =>* [d:2, L:[4]] (d + 10 * (eval L))  
           =>* 2 + 10 * (eval [4])  
           =>* 2 + 10 * (4 + 10 * (eval [ ]))  
           =>* 2 + 10 * (4 + 10 * 0)  
           =>* 2 + 10 * 4  
           =>* 42
```

传值调用(call-by-value):

1. Binding:  $d \rightarrow 2, L \rightarrow [4]$ ;
2. 代入表达式;
3. Binding:  $d \rightarrow 4, L \rightarrow [ ]$ ;
4. 代入表达式;
5. 计算;
6. 计算。



# 模式匹配举例： decimal

```
fun decimal (n:int) : int list =  
  if n<10 then [n]  
    else (n mod 10) :: decimal (n div 10);
```

- 函数decimal的类型定义？
- 模式匹配： decimal 42 = ?                  decimal 0 = ? ， 给出匹配过程

# 规则说明

- 部分操作的内建规则：

- \* 结合性强于 ->
- \* 无结合规则
- -> 为右结合

- 以下几种类型定义的表述是否相同？

int \* int -> real      vs.      (int \* int) -> real

int -> int -> int      vs.      int -> (int -> int)

int \* int \* int      vs.      (int \* int) \* int      vs.      int \* (int \* int)