函数式编程原理

Lecture 3

上节课内容回顾

- 数据类型：

  基础类型(int, real, bool…)

  元组(tuple types with *)

  函数(function types with ->)

  表(lists)
- 表达式：表达式求值的结果为一个值(if it terminates)

表(List)

- 相同类型元素的有限序列
- 元素可以重复出现，其顺序是有意义的

- 表中元素可以为任意类型，但需具有相同类型
- 表可以嵌套使用
- 表为多态类型

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

- 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)

声明(Declarations)

- 赋予某个对象一个名字，包括值、类型、签名、结构等

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：

```
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

声明的使用

声明将产生名字(变量)和值的绑定(结合)，
绑定具有静态作用域（ Bindings are *statically scoped*）

声明函数：

```
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;
```

变量pi的重新声明

area 1.0; 或者 area (1.0);

\> 3,14: real;

erea仍在3.14的scope中

声明的作用域

- Bindings have syntactically fixed *scope*

```
val x = 3.14;
... scope of x : 3.14
```

```
let
  val x = 3.14
in
   ... scope of x : 3.14
end;
... not in scope
```

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
```

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

局部声明：

```
let D in E 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
  - 模式中不是构造子的名字，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 //同一模式中，一个变量不能出现两次
- 常数 ：42, true, ~3
- 元组 ：(p1, ..., pk) //p1, ..., pk均为模式
- 表 ：nil, p1::p2, [p1, ..., pk]

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]　　　　d::L 和 [ ]

42 和 42 (value)　　　　42 和 0 (value)

变量x 和 任意值 v　　　　_ 和 任意值v

p1::p2 和 [ ]　　　　p1::p2 和 v1::v2

- A *pattern* can be *matched* against a *value*
- If the match *succeeds*, it produces *bindings*

  matching d::L against the value [2,4]
  succeeds with bindings ⟦d:2, L:[4]⟧

  matching d::L against the value [ ]
  fails

- Matching 42 against the value 42 *succeeds*
- Matching 42 against the value 0 *fails*
- Matching x against *any* value v *succeeds* with the binding x:v
- Matching _ against any value *succeeds*
- Matching p_1::p_2 against [ ] *fails*
- Matching p_1::p_2 against v_1::v_2 *fails* if matching p_1 against v_1 *fails*, or matching p_2 against v_2 *fails*

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 [2,4]

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

- eval : int list -> int
- This function definition uses *list patterns*
  - [ ] only matches empty list
  - d::L only matches non-empty list, binds d to head of list, L to tail
- eval [2,4] = 42

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->2, L->[4];

2. 代入表达式；

3.Binding：d->4, L->[ ];

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

模式匹配举例：decimal

```
decimal 42
   =>* if 42 < 10 then [42]
                  else (42 mod 10) :: decimal(42 div 10)

   =>* (42 mod 10) :: decimal (42 div 10)
   =>* 2 :: decimal (42 div 10)
   =>* 2 :: decimal 4
   =>* 2 :: (if 4 < 10 then [4] else ...)
   =>* 2 :: [4]
   =>* [2,4]
```

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)

值绑定

- 【$x_1{:}v_1, \ldots, x_k{:}v_k$ 】：表示值绑定(value bindings)的集合
  $x, x_1, \ldots$ ：表示变量 (Variables)
  $v, v_1, \ldots$ ： 表示值 ((syntactic) Values)
  $e, e_1, \ldots$ ： 表示表达式 (Expressions)
  $t, t_1, \ldots$ ： 表示类型 (Types)
- 可终止状态(Termination):
  $e\downarrow$ 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**

表达式推导

- Reflexive

$$e \Rightarrow^* e$$

- Transitive

If

$$e_1 \Rightarrow^* e_2 \text{ and } e_2 \Rightarrow^* e_3$$

then

$$e_1 \Rightarrow^* e_3$$

表达式推导

- Given a collection of value bindings

  $$[\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]$$

  and an expression e

  we write

  $$[\![x_1{:}v_1, \ldots, x_k{:}v_k]\!] \, e$$

  for the expression obtained by substituting

  v_1 for x_1, ..., v_k for x_k in e

表达式推导

$[\![x{:}2]\!] (x + x)$ is $(2 + 2)$

$[\![x{:}2]\!] (\mathrm{fn}\ y \Rightarrow x + y)$ is $(\mathrm{fn}\ y \Rightarrow 2 + y)$

$[\![x{:}2]\!] (\mathrm{if}\ x>0\ \mathrm{then}\ 1\ \mathrm{else}\ f(x-1))$

is $(\mathrm{if}\ 2>0\ \mathrm{then}\ 1\ \mathrm{else}\ f(2-1))$

表达式推导

- **Arithmetic**

  If

  $e_1 \Rightarrow^* v_1$ and $e_2 \Rightarrow^* v_2$

  then

  $e_1 + e_2 \Rightarrow^* v_1 + e_2$

  $\Rightarrow^* v_1 + v_2$

- **Boolean**

  If $e \Rightarrow^*$ **true**, then

  if e **then** e_1 **else** $e_2 \Rightarrow^* e_1$

表达式推导

- **Functions**

  If

  $e_1 \Rightarrow^* (\text{fn } x \Rightarrow e)$ and $e_2 \Rightarrow^* v$

  then

  $e_1\ e_2 \Rightarrow^* (\text{fn } x \Rightarrow e)\ e_2$

  $\Rightarrow^* (\text{fn } x \Rightarrow e)\ v \Rightarrow^* [\![x:v]\!]e$

  a function call evaluates its argument

- **Declarations**

  In the scope of fun f(x) = e

  $f \Rightarrow^* (\text{fn } x \Rightarrow e)$

Patterns

- If matching p against v *succeeds* with bindings $[\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]$,

  $$(\text{fn } p \Rightarrow e)\ v \quad \Rightarrow^* \quad [\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]e$$

- If matching p_1 against v *fails*, and matching p_2 against v *succeeds* with bindings $[\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]$,

  $$(\text{fn } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2)\ v$$
  $$\Rightarrow^* \quad [\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]e_2$$

求值符号的使用

- e => e'　一次推导
- e =>* e'　有限次推导
- e =>$^+$ e'　至少一次推导

- 如：fun f(x:int):int = f x

  f 0 =>$^+$ (fn x => f x) 0
  　　=>* [x:0] f x
  　　=>* f 0

  因此：f 0 =>$^+$ f 0
  　　(f 0) ↑

- =>和=>*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 : int and all v:int
　if $e_1 + e_2$ =>* v then $e_2 + e_1$ =>* v

此时,我们关注计算结果多于计算过程

主要内容

- 代码说明 (Specifications)
- 程序证明 (proofs)
- 近似运行时间 (Asymptotic runtime)
- 递推分析 (Recurrence relations)

代码说明(Specifications)

- 函数定义前，用注释信息描述函数功能，形如(* comments*) ：
  - 函数名字和类型 (类型定义)
  - REQUIRES：参数说明 (明确参数范围)
  - ENSURES：函数在有效参数范围内的执行结果 (函数功能)

范例1：函数eval的说明

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

(* eval : int list -> int          *)

(* REQUIRES:                       *)
(*    every integer in L is a decimal digit              *)

(* ENSURES:                        *)
(*    eval(L) evaluates to a non-negative integer *)
```

范例2：函数decimal的说明

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

```
(* decimal : int -> int list    *)

(* REQUIRES: n >= 0             *)

(* ENSURES:                     *)
(*   decimal(n) evaluates to a list L of decimal digits, *)
(*      such that eval(L) = n *)
```

代码说明的作用

- 确保函数行为的正确性
- 确保在允许的参数范围内能得到正确的结果

- 如何证明函数能按说明的内容正确的执行?

——程序正确性证明

程序正确性证明

- 基于等式或推导的方式进行数学证明
- 程序结构作为指导：

程序语法	推导
if-then-else	布尔分析
case p of …	case分析
fun f(x) = …f…	归纳法

归纳法(Induction)

- 常见的几种归纳法：
  - 简单归纳法 (simple (mathematical) induction)
  - 完全归纳法 (complete (strong) induction)
  - 结构归纳法 (structural induction)
  - 良基归纳法 (well-founded induction)

简单归纳法(simple (mathematical) induction)

证明对所有非负整数n，P(n)都成立

基本情形(base case)：证明P(0)成立

推导过程(inductive step)：

假设对任意k(≥0)，P(k)成立，则P(k+1)也成立

用简单归纳法证明

```
fun f(x:int):int =
    if x=0 then 1 else f(x-1) + 1
(* REQUIRES x ≥0 *)
(* ENSURES f(x) = x+1 *)
```

试证明：对所有整数 x，当$x \geq 0$时, $f(x) = x+1$

用简单归纳法证明

```
fun f(x:int):int =
      if x=0 then 1 else f(x-1) + 1
(* REQUIRES x≥0         *)
(* ENSURES f(x) = x+1 *)
```

- To prove:

For all values x:int
such that $x \geq 0$, $f(x) = x+1$

用简单归纳法证明

- Let P(n) be f(n) = n+1
- Base case: we prove P(0), i.e. f(0) = 0+1

f 0 = (fn x => if x=0 then 1 else f(x-1)+1) 0
= ⟦x:0⟧(if x=0 then 1 else f(x-1)+1)
= **if** 0=0 **then** 1 **else** f(0-1) + 1
= **if true then** 1 **else** f(0-1) + 1
= 1

0+1 = 1

So f(0) = 0+1

用简单归纳法证明

- Let P(n) be f(n) = n+1
- Inductive step:
  let k≥0, assume P(k), prove P(k+1).
  Let v be the numeral for k+1.

f(k+1) = **if** v=0 **then** 1 **else** f(v-1) + 1
= **if false then** 1 **else** f(v-1) + 1
= f(v-1) + 1
= f(k) + 1 since v=k+1
= (k + 1) + 1 by assumption P(k)

So P(k+1) follows from P(k)

用简单归纳法证明

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

(size = length of argument list, decreases by 1)

试证明：对所有值 L:int list，
存在一个整数 n，使eval L =>* n

?为什么不能用？

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

简单归纳法的适用范围

- 适用于涉及自然数的递归函数
  - 参数为非负整数
  - f(x)的递归调用形如f(y),且size(y)=size(x)-1

完全归纳法(complete (strong) induction)

证明对所有非负整数n，P(n)都成立

- 将P(k)简化为k个子问题： P(0), P(1), … , P(k-1)，且它们均成立时，可以利用{P(0), P(1), … , P(k-1)}推导出P(k)也成立
  - 如：P(0)成立

    P(1)可由P(0)推导出来

    P(2)可由P(0)， P(1)推导出来

    P(3)可由P(0)， P(1)， P(2)推导出来

    ……

    P(k)可由P(0), P(1), … , P(k-1)推导出来

完全归纳法的适用范围

- 适用于涉及自然数的递归函数
  - 参数为非负整数
  - f(x)的递归调用形如f(y),且size(y)<size(x)

用完全归纳法证明

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

(when n≥10,0≤n div 10 <n)

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

试证明：对所有值 n:int (n≥0),

eval (decimal n) = n

```
fun decimal (n:int) : int list =
   if n<10 then [n] else (n mod 10) :: decimal (n div 10);
fun eval ([ ]:int list) : int = 0
   | eval (d::L) = d + 10 * (eval L);
```

用完全归纳法证明

对所有值 n:int (n≥0)， eval (decimal n) = n

当0 <= n < 10时， decimal(n)=[n], 要证明eval (decimal n) = n

eval (decimal n) = eval([n])=n+10*eval([ ])=n+0=n

当0 > 10时， 有eval(decimal(m))=m,
要证明eval (decimal m+1) = m+1

设 n=m+1, x = m mod 10， y= m div 10
eval (decimal m+1) = eval((m+1 mod 10) :: decimal (m+1 div 10))
= x+1+10*eval (decimal (m+1 div 10))
=x+1 + 10* y = x + 10 * y +1 = m+1 = n

用完全归纳法证明

```
fun decimal (n:int) : int list =
   if n<10 then [n] else (n mod 10) :: decimal (n div 10);
fun eval ([ ]:int list) : int = 0
  | eval (d::L) = d + 10 * (eval L);
```

对所有值 n:int (n≥0)，eval (decimal n) = n

当0 <= n < 10时，decimal(n)=[n], 要证明eval (decimal n) = n

eval (decimal n) = eval([n])=n+10*eval([ ])=n+0=n

当0 > 10时，对于any 0<=m<n,
有eval(decimal(m))=m, 要证明eval (decimal n) = n

设x = n mod 10，y= n div 10
eval (decimal n) = eval((n mod 10) :: decimal (n div 10))
= eval(x:: decimal y)n+10*eval([ ])
=x + 10* decimal y = x + 10 * y = n

结构归纳法(structural induction)

完全归纳法在其他数据类型上的推广

- 基本情形： P([ ])
- 归纳步骤：1）对具有类型t的所有元素y和t list类型的数ys，都有P(ys)成立时，
  2）证明P(y::ys)成立,

换句话说，在∀ i < k, P(i)成立的条件下证明P(k)成立。

适用于涉及表和树的递归函数

近似运行时间

- 反映基于大批量数据的程序运行性能
  - 假设基本操作为常量执行时间(Assume basic ops take *constant* time)
  - 用O记号表示算法的时间性能(Give big-O classification)

- f(n)的近似运行时间为O(g(n))：
  - 存在整数N和c，满足

    ∀n≥N, f(n)的近似运行时间≤ c g(n)

为什么叫“近似”？

- 加法中的常数加不考虑(Additive constants don’t matter)

  $n^5+1000000$ is $O(n^5)$
- 乘法中的常数乘不考虑(Multiplicative constants don’t matter)

  $1000000n^5$ is $O(n^5)$
- g(n)尽可能精确(Be as accurate as you can)

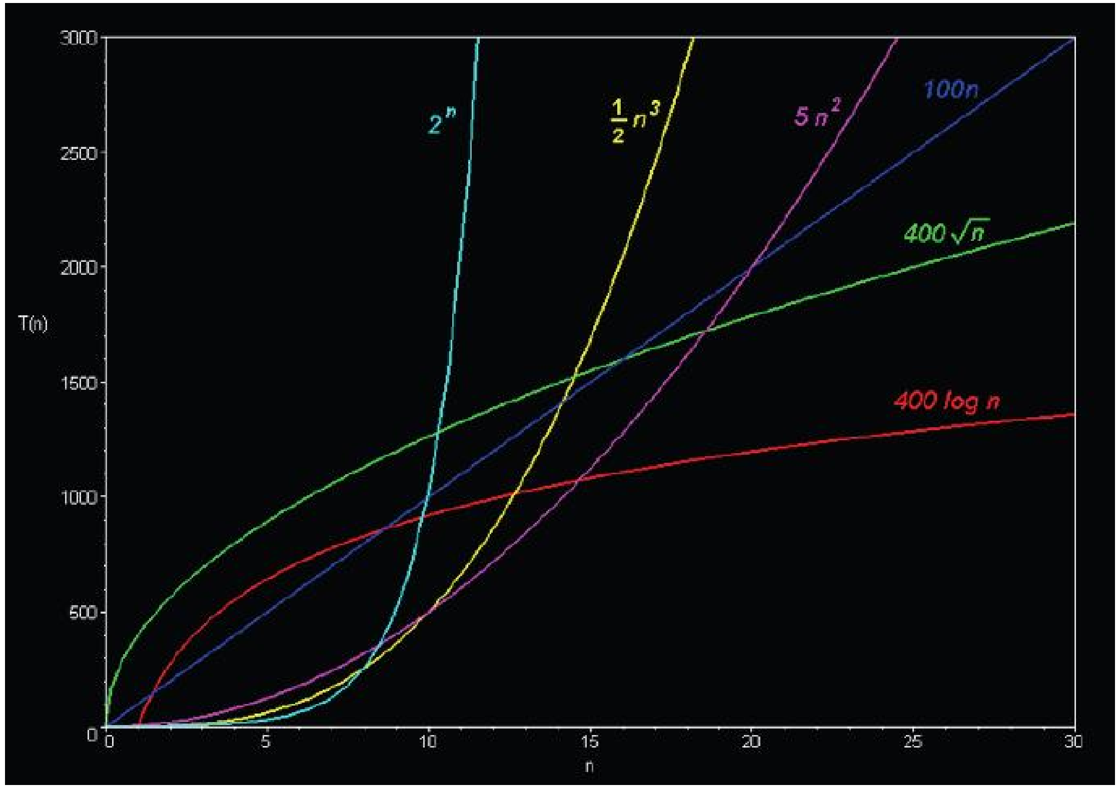
近似运行时间分析

- 求解步骤：
  1. 找出算法中的基本语句：算法中执行次数最多的那条语句就是基本语句，通常是最内层循环的循环体
  2. 计算基本语句的执行次数的数量级：忽略所有低次幂和最高次幂的系数，保证基本语句执行次数的函数中的最高次幂正确
  3. 用O记号表示算法的时间性能：将基本语句执行次数的数量级放入O记号中。

近似运行时间分析

```
for (i=1; i<=n; i++)
    x++;

for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      x++;
```



$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n\log n) < O(n^2) < O(n^3) < \ldots < O(2^n) < O(n!)$

对数时间 (logarithmic)　平方根时间 (square root)　线性时间 (linear)　Quadratic　Cubic　多项式时间(polynomial)　指数时间 (exponential)

递推分析(recurrences)

- 递归函数的定义给出了程序的递推关系，执行情况用*work*表示

  (A *recursive function definition* suggests a ***recurrence relation*** for *work*, or *runtime*)

  - W(n)表示参数规模为n的程序的执行情况work (W(n) = work on inputs of size n)

- W(n)的推导：
  - Base cases (P(0))：评估基本操作的执行 (Estimates the number of basic operations)
  - Inductive case (P(x))：
    - 用归纳法得到W(n)的表达式 (Try to find a *closed form* solution for W(n) using *induction*)
    - 对表达式进行简化，得到一个具有相同渐近属性的表达式(Find solution to a *simplified* recurrence with the same asymptotic properties)

注意：推导过程要规范(Appeal to table of standard recurrences)

递推分析实例

```
fun exp (n:int):int =
  if n=0 then 1 else 2 * exp (n-1);
```

M： (fn n => if n=0 then 1 else 2 * exp(n-1))

exp 4 $\Rightarrow^{(1)}$ M 4 $\Rightarrow^{(4)}$ 2 * (M 3)

$\Rightarrow^{(4)}$ 2 * (2 * (M 2))

$\Rightarrow^{(4)}$ 2 * (2 * (2 * (M 1)))

$\Rightarrow^{(4)}$ 2 * (2 * (2 * (2 * (M 0))))

$\Rightarrow^{(2)}$ 2 * (2 * (2 * (2 * 1)))

$\Rightarrow^{(4)}$ 16

M 4 => if 4=0 then ...
=> 2 * exp (4-1)
=> 2 * M (4-1)
=> 2 * M 3

由此可推出：

for all n≥0, exp n $\Rightarrow^{(5n+3)} 2^n$

近似运行时间为：O(n)

时间复杂度（big-O)

- 时间复杂度也称渐近时间复杂度，表示为T(n)=O(f(n))，其中f(n)为算法中频度最大的语句频度。
    - 程序的执行时间依赖于具体的软硬件环境，不能用执行时间的长短来衡量算法的时间复杂度，而要通过基本语句执行次数的数量级来衡量。
    - 算法中语句的频度与问题规模有关，一般考虑问题规模趋向无穷大时，该程序时间复杂度的数量级。
    - 一般仅考虑在最坏情况下的时间复杂度，以保证算法的运行时间不会比它更长。

程序执行情况W(n)分析

```
fun exp (n:int):int =
  if n=0 then 1 else 2 * exp (n-1);
```

用$W_{exp}(n)$表示程序 exp(n)的执行时间

$$W_{exp}(0) = c_0$$

$$W_{exp}(n) = c_0 + n\, c_1 \quad (n>0)$$

For all $n \geq 0$, $W_{exp}(n) \leq c\, n$ → O(n)

程序执行时间随n值的增加线性增长

能否缩短程序运行时间、提高效率？

fastexp

```
fun square(x:int):int = x * x
fun fastexp (n:int):int =
   if n=0 then 1 else
   if n mod 2 = 0 then square(fastexp (n div 2))
                  else 2 * fastexp(n-1)
```

$W_{fastexp}(n)$如何推导？

```
fastexp 4 = square(fastexp 2)
          = square(square (fastexp 1))
          = square(square (2 * fastexp 0))
          = square(square (2 * 1))
          = square 4 =16
```

程序执行情况分析：Can it be faster?

- The definition of exp relies on the fact that

$$2^n = 2\,(2^{n-1})$$

- Everybody knows that

$$2^n = (2^{n \text{ div } 2})^2 \qquad \text{if n is even}$$

程序执行情况分析：Can it be faster?

```
fun square(x:int):int = x * x

fun fastexp (n:int):int =
   if n=0 then 1 else
   if n mod 2 = 0 then square(fastexp (n div 2))
                  else  2 * fastexp(n-1)
```

```
fastexp 4 = square(fastexp 2)
          = square(square (fastexp 1))
          = square(square (2 * fastexp 0))
          = square(square (2 * 1))
          = square 4 =16
```

程序执行情况分析：Can it be faster?

```
fun fastexp (n:int):int =
  if n=0 then 1 else
  if n mod 2 = 0 then square(fastexp (n div 2))
                 else  2 * fastexp(n-1)
```

- Let $W_{fastexp}(n)$ be the runtime for fastexp(n)

  $W_{fastexp}(0) = c_0$

  $W_{fastexp}(1) = c_1$

  $W_{fastexp}(n) = W_{fastexp}(n \text{ div } 2) + c_2$ for n>1, even

  $W_{fastexp}(n) = W_{fastexp}(n \text{ div } 2) + c_3$ for n>1, odd

  for some constants c_0, c_1, c_2, c_3

程序执行情况分析：Can it be faster?

- Let $T_{fastexp}(n)$ be given by

$$T_{fastexp}(0) = 1$$

$$T_{fastexp}(1) = 1$$

$$T_{fastexp}(n) = T_{fastexp}(n \text{ div } 2) + 1 \quad \text{for } n>1$$

$\boxed{T_{fastexp}(n) \leq c \log_2(n) \text{ for all large enough } n}$

整数的比较——compare

compare: int * int -> order

```
type order = LESS | EQUAL | GREATER;
fun compare(x:int, y:int):order =
        if x<y then LESS else
        if y<x then GREATER else EQUAL;
```

compare(x,y) = LESS	if x<y
compare(x,y) = EQUAL	if x=y
compare(x,y) = GREATER	if x>y

整数的比较

- $\leq$ 对int类型的数据进行线性排序(***linear ordering***)

  For all values a,b,c :int

  - If $a \leq b$ and $b \leq a$ then $a = b$ (反对称性，antisymmetry)
  - If $a \leq b$ and $b \leq c$ then $a \leq c$ (传递性，transitivity)
  - Either $a \leq b$ or $b \leq a$ (整体性，totality)

- $<$ 定义为

  For all values a,b:int

  - $a < b$ if and only if ($a \leq b$ and $a \neq b$)

  且满足

  For all values a,b:int

  - $a < b$ or $b < a$ or $a = b$ (三分法，trichotomy)

排序结果的判断——sorted

- sorted : int list -> bool
- 函数功能：线性表中的元素按照升序（允许相邻元素相同）的方式排列，则该整数表为有序表（增序）。A list of integers is <-sorted： if each item in the list is ≤ all items that occur later.

  ——用于排序算法的测试

- 函数代码：

```
fun sorted [ ] = true
   | sorted [x] = true
   | sorted (x::y::L) =
      (compare(x,y) <> GREATER) andalso sorted(y::L);
```

<>: neq

For all L : int list,
sorted(L) = true if L is <-sorted
= false otherwise

插入排序

如何用递归程序实现？

- 基本思想：
  - 每次将一个待排数据按大小插入到已排序数据序列中的适当位置，直到数据全部插入完毕。
- 操作步骤：

1.从有序数列{}和无序数列$\{a_1,a_2，…，a_n\}$开始进行排序；

2.处理第i个元素(i=2,3，…，n)时，数列$\{a_1,a_2，…，a_{i-1}\}$是有序的，而数列$\{a_i,a_{i+1}，…，a_n\}$是无序的。用a_i与有序数列进行**比较**，找出合适的位置将a_i插入；

3.重复第二步，共进行n-i次**插入**处理，数列全部有序。

整数的插入——ins

ins : int * int list -> int list

(* REQUIRES L is a sorted list *)

(* ENSURES ins(x, L) = a sorted perm of x::L *)

```
fun ins (x, [ ]) = [x]
  | ins (x, y::L) = case compare(x, y) of
                GREATER => y::ins(x, L)
              |    _        => x::y::L
```

_ : less and equal

如何证明？

For all sorted integer lists L,
ins(x, L) = a sorted permutation of x::L.

用归纳法证明Ins函数的正确性

```
fun ins (x, [ ]) = [x]
  | ins (x, y::L) = case compare(x, y) of
                          GREATER => y::ins(x, L)
                        |    _        => x::y::L
```

For all sorted integer lists L,
  ins(x, L) = a sorted permutation of x::L.

- **根据L的长度进行归纳证明**

1. L长度为0时，证明ins(x, [ ]) 为有序表.
2. 假设对所有长度小于等于k的有序表A，ins(x, A) 为 x::A的有序表.
   证明：ins(x, L) 为x::L的有序表,其中L的长度为k+1且为有序表

proof outline

Theorem
For all sorted lists L,
ins(x, L) = a sorted permutation of x::L.

- **Proof**: By induction on length of L.
- **Base case**: When L has length 0, L must be [ ].
  [ ] is <-sorted. Show ins(x, [ ]) = a sorted perm of x::[ ].
- **Inductive case**: Let k>0 and assume IH:
  For all sorted lists A of length < k,
  ins(x, A) = a sorted perm of x::A.
  - Let L be a sorted list of length k.
    Pick y and R such that L=y::R. So length(R) < k.
  - R is a sorted list with length < k, and y ≤ all of R
  - By IH, ins(x, R) = a sorted perm of x::R
  - Show: ins(x, y::R) = a sorted perm of x::(y::R)

sketch

```
ins (x, y::R) =  case compare(x, y) of
                   GREATER => y::ins(x, R)
                 |    _    => x::y::R;
```

- R is sorted and y ≤ all of R.
- By IH, ins(x, R) = a sorted perm
  - If x>y we have ins(x, y::R) = y::ins(x,R)
    This list is *sorted* because...
    This list is a *perm* of x::y::R because...
  - If x≤y we have ins(x, y::R) = x::y::R
    This list is *sorted* because...
    This list is a *perm* of x::y::R because...
- In all cases, ins(x, y::R) = a sorted perm of x::y::L

插入排序——isort

```
fun ins (x, [ ]) = [x]
  | ins (x, y::L) = case compare(x, y) of
                      GREATER => y::ins(x, L)
                    |  _       => x::y::L
```

```
isort : int list -> int list

(* REQUIRES true                          *)
(* ENSURES isort(L) = a sorted perm of L    *)

fun isort [ ] = [ ]
  |  isort (x::L) = ins (x, isort L)
```

For all integer lists L,
isort L = a sorted permutation of L.

插入排序——isort

```
fun ins (x, [ ]) = [x]
  | ins (x, y::L) = case compare(x, y) of
                         GREATER => y::ins(x, L)
                       |    _    => x::y::L
```

isort : int list -> int list

(* REQUIRES true *)
(* ENSURES isort(L) = a sorted perm of L *)

```
fun isort [ ] = [ ]
  |  isort (x::L) = ins (x, isort L)
```

[5, 4, 3, 2, 1]怎么排的？

插入排序——isort

```
fun ins (x, [ ]) = [x]
  | ins (x, y::L) = case compare(x, y) of
                        GREATER => y::ins(x, L)
                      | _       => x::y::L
```

```
isort : int list -> int list
fun isort [ ] = [ ]
  |  isort (x::L) = ins (x, isort L)
```

```
isort (5::[4, …, 1])
= ins (5, isort [4, …, 1])
= ins (5, ins(4, isort [3, 2, 1]))
= ins (5, ins(4, ins(3, isort [2, 1])))
= ins (5, ins(4, ins(3, ins(2, isort[1]))))
= ins (5, ins(4, ins(3, ins(2, [1]))))
= …
```

[5, 4, 3, 2, 1]怎么排的？

插入排序——isort

- **Proof**: By induction on length of L.
- **Base case**: When L has length 0, L must be [ ].
  [ ] is <-sorted. Show ins(x, [ ]) = a sorted perm of x::[ ].
- **Inductive case**: Let k>0 and assume IH:

  For all sorted lists A of length < k,
  ins(x, A) = a sorted perm of x::A.

  - Let L be a sorted list of length k.
    Pick y and R such that L=y::R. So length(R) < k.
  - R is a sorted list with length < k, and y ≤ all of R
  - By IH, ins(x, R) = a sorted perm of x::R
  - Show: ins(x, y::R) = a sorted perm of x::(y::R)

插入排序——isort

```
ins (x, y::R) =  case compare(x, y) of
                   GREATER => y::ins(x, R)
                 |    _    => x::y::R;
```

- R is sorted and y ≤ all of R.
- By IH, ins(x, R) = a sorted perm of x::R
    - If x>y we have ins(x, y::R) = y::ins(x,R)
      This list is *sorted* because...
      This list is a *perm* of x::y::R because...
    - If x≤y we have ins(x, y::R) = x::y::R
      This list is *sorted* because...
      This list is a *perm* of x::y::R because...
- In all cases, ins(x, y::R) = a sorted perm of x::y::L

proof outline

For all L: int list,
isort L = a <-sorted permutation of L.

- **Proof**: By induction on length of L.
- **Base case**: for L = [ ].
  Show that isort [ ] = a sorted perm of [ ].
- **Inductive case**: for L = y::R.
  IH: isort R = a sorted perm of R.
  Show: isort(y::R) = a sorted perm of y::R.

Use the *proven* spec for ins!

另一个插入排序——isort’

- isort’ : int list -> int list

```
fun isort' [ ] = [ ]
  |   isort' [x] = [x]
  |   isort' (x::L) = ins (x, isort' L);
```

isort and isort’ are extensionally equivalent.

For all L : int list, isort L = isort’ L.

归并排序

- 基本思想：采用分治法（*Divide and Conquer*）将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。

split : int list -> int list * int list

- 操作步骤：
  1. 将n个元素**分成**两个含n/2元素的子序列
  2. 将两个子序列递归排序
  3. **合并**两个已排序好的序列

merge : int list * int list -> int list

善于使用帮助(helper)函数

We'll use helper functions to do the splitting and merging

split : int list -> int list * int list

merge : int list * int list -> int list

善于使用帮助(helper)函数

- 如何实现更加复杂的函数功能？
  - 状态的记录和改变
  - 更复杂的递归
  - Helper functions

善于使用帮助(helper)函数

- 满足调用函数的功能需求
- 扩展应用到其他函数中，实现更广泛的功能

merge : int list * int list -> int list

在归并排序中：

For all sorted lists A and B,
merge(A, B)= a sorted permutation of A@B

通常情况下：

For all integer lists A and B,
merge(A, B)= a permutation of A@B

表的分割——split

```
split : int list -> int list * int list

(* REQUIRES true                                          *)
(* ENSURES split(L) = a pair of lists (A, B)              *)
(* such that length(A) and length(B) differ by at most 1, *)
(* and A@B is a permutation of L.                         *)

fun split [ ]  = ([ ], [ ])
  | split [x] = ([x], [ ])
  | split (x::y::L) =
       let val (A, B) =split L
       in (x::A, y::B)
       end
```

能否去掉？

For all L:int list,
split(L) = a pair of lists (A, B) such that
length(A) ≈ length(B) and
A@B is a permutation of L.

用归纳法证明split函数的正确性

根据L的长度用完全归纳法进行证明

1. L = [ ], [x]

   ①split [ ] = a pair (A, B) such that length(A)≈length(B) & A@B is a perm of [ ].

   ②split [x] = a pair (A, B) such that length(A)≈length(B) & A@B is a perm of [x].

2. 假设split(R) = a pair (A’, B’) such that length(A’)≈length(B’) & A’@B’ is a perm of R.

   证明: split(L) = a pair (A, B) such that length(A)≈length(B) & A@B is a perm of x::y::R.

   (L=x::y::R)

表的分割——split

[5, 4, 3, 2, 1]怎么split的？

split : int list -> int list * int list

```
fun split [ ]  = ([ ],  [ ])
    | split [x] = ([x], [ ])
    | split (x::y::L) =
          let val (A, B) =split L
          in (x::A, y::B)
          end
```

split (5::4::[3, 2, 1])
= split
(5::4::split(3::2::split[1]))
= split (5::4::split(3::[1], 2::[ ]))
= split (5::3, 1], 4::[2])
= [5,3,1], [4,2]

表的合并——merge

merge : int list * int list -> int list

```
(* REQUIRES A and B are <-sorted lists                *)
(* ENSURES merge(A, B) = a <-sorted perm of A@B        *)

fun merge (A, [ ]) = A
  | merge ([ ], B) = B
  | merge (x::A, y::B) = case compare(x, y) of
                  LESS => x :: merge(A, y::B)
                | EQUAL => x::y::merge(A, B)
                | GREATER => y :: merge(x::A, B)
```

能否写成:
merge([], []) = []?

如何证明？

用归纳法证明Merge函数的正确性

For all <-sorted lists A and B,
merge(A, B) = a <-sorted permutation of A@B.

- **Method**: *strong* induction on length(A)*length(B).
- **Base cases**: (A, [ ]) and ([ ], B).
  (i) Show: if A is <-sorted, merge(A,[ ]) = a <-sorted perm of A@[ ].
  (ii) Show: if B is <-sorted, merge([ ],B) = a <-sorted perm of [ ]@B.
- **Inductive case**: (x::A, y::B).
  Induction Hypothesis: for all *smaller* (A′, B′), if A′ & B′ are <-sorted, merge(A′, B′) = a <-sorted perm of A′@B′.
  Show: if x::A and y::B are <-sorted,
  merge(x::A, y::B) = a <-sorted perm of (x::A)@(y::B).

Merge函数的特点

```
fun merge (A, [ ])  = A
 |   merge ([ ], B) = B
 |  merge (x::A, y::B) = case compare(x, y) of
                                   LESS   => x :: merge(A, y::B)
                                |   EQUAL  => x::y::merge(A, B)
                                | GREATER  => y :: merge(x::A, B)
```

Does clause order matter? **NO**

Patterns are { Exhaustive
Overlap of first two clauses is harmless

Each yields merge([ ], [ ]) = [ ]

Could use *nested* **if-then-else** instead of **case**.

But we need a 3-way branch, so **case** is *better style*.

开始使用帮助(helper)函数

- We *defined* split and merge
- We *proved* they meet their specs
- Now let's use them to implement the **mergesort** algorithm...

归并排序—— mergesort

msort : int list -> int list

(* REQUIRES true *)
(* ENSURES msort(L) = a <-sorted perm of L *)

```
fun msort [ ] = [ ]
   | msort [x] = [x]
   | msort L =  let
                    val (A, B) = split L
                in
                    merge (msort A, msort B)
                end
```

能否去掉？

msort [1] =
=> merge(msort[1], msort[])
⇒merge(merge(msort[1], msort[]),
…))
⇒ 不断拆分split[x]

归并排序—— mergesort

msort : int list -> int list

```
(* REQUIRES  true                              *)
(* ENSURES   msort(L) = a <-sorted perm of L   *)

fun msort [ ] = [ ]
  |  msort [x] = [x]
  |  msort L =  let
                  val (A, B) = split L
                  val A′ = msort A
                  val B′ = msort B
                in
                  merge (A′, B′)
                end
```

an alternative version

归并排序

```
fun split [ ]  = ([ ],  [ ])
    | split [x] = ([x], [ ])
    | split (x::y::L) =
         let val (A, B) =split L

         in (x::A, y::B)
         end

fun merge (A, [ ]) = A
| merge ([ ], B) = B
| merge (x::A, y::B) = case compare(x, y) of
                   LESS => x :: merge(A, y::B)
                 | EQUAL => x::y::merge(A, B)
                 | GREATER => y :: merge(x::A, B)

fun msort [ ] = [ ]
   | msort [x] = [x]
   | msort L =  let  val (A, B) = split L
                in   merge (msort A, msort B)
                end
```

ML编程原则(principles)

- 每个函数都对应一个功能描述说明 (Every function needs a spec)
- 需要验证程序符合功能描述说明 (Every spec needs a proof)
- 用归纳法进行递归函数的正确性验证 (Recursive functions need inductive proofs)
  - 选取合适的归纳法 (Learn to pick an appropriate method...)
  - 设计恰当的帮助函数 (Choose helper functions wisely)

*msort*的证明非常简单，源于函数***split and merge***的使用

功能说明的作用 (the joy of specs)

- 就是注释，函数用来干嘛的？什么参数？什么类型？结果是什么？
- 函数的证明有时依赖于某个被调用函数的证明结果(符合spec要求)

The **proof for msort relied only on the *specification proven for split*** (and the specification proven for merge)

- 被调用函数可以由具有相同功能说明的其他函数替换，而且证明过程仍然有效

In the definition of msort we can *replace* split by *any function that satisfies this specification, and the proof will still be valid,* for the new version of msort

函数替换举例

```
fun split' [ ]  = ([ ],  [ ])
    | split' [x] = ([ ], [x])
    | split' (x::y::L) =
         let val (A, B) =split' L
         in (x::A, y::B)
         end
```

```
fun msort' [ ] = [ ]
    | msor't [x] = [x]
    | msort' L =  let
                     val (A, B) = split' L
                      in
                      merge (msort' A, msort' B)
                       end
```

尽管split和split'函数不相同，但他们都满足整数表分割功能，在正确性证明过程中没有区别，所以函数msort和mosrt'都是正确的。