作业一

1.

- x::L 匹配成功，非空list
- _::_ 匹配成功，非空list
- x::(y::L) 匹配不成功，为至少有2个元素的list
- (x::y)::L 匹配不成功，为int list list 类型，且list非空，list中的子list非空
- [x, y] 匹配成功，匹配有两个元素的list

2.

- list of length 3 [x,y,z]
- lists of length 2 or 3 没有对应的模式，因为匹配的list长度应该固定
- Non-empty lists of pairs (x,y) :: L
- Pairs with both components being non-empty lists (x::L1 , y::L2)

3.

- 第4行：val x : int = 2;
- 第5行：val m = 12.4 : real； x : int = 2;
- 第6行：val x = 9001 : int;
- 第14行：val z: int = 24+x = 27

4.

```
fun zip ( (x::L1, y::L2) : string list * int list ) : (string * int) list = (x,y)::zip(L1,L2)
    |zip ( _ , [ ] ) = []
    |zip ( [ ] ,_ ) = [];

zip(["abc","def"],[1,2,3]);
```


```
fun zip ( (x::L1, y::L2) : string list * int list ) : (string * int) list = (x,y)::zip(L1,L2)
    |zip ( _ , [ ] ) = []
    |zip ( [ ] ,_ ) = [];

zip(["abc","def"],[1,2,3]);
```


5.

- (*f : int -> int*)

  fun f (3 : int) : int = 9

  f _ = 4;

  模式匹配出错格式，未加|

- (*circ : real -> real*)

  fun circ (r : real) : real = 2 *pi* r

2和pi相乘会出错，类型不匹配，改为2.0 *pi* r

- (*semicirc : real -> real*)

  fun semicirc : real = pie * r

  pie改成pi

- (*area : real -> real*) fun area (r : int) : real = pi *r* r

  结果类型不匹配 r : real

6.

```
fun fib n = if n<=2 then 1 else fib(n-1) + fib(n-2);
```

每计算一次fib(n)都需要递归计算fib(n-1)和fib(n-2),导致多次计算相同的子问题，如在计算 *F8*时：*F8 = F6 + F7 = F6 + (F5 + F6)*=...，在这个递归过程中，需要维持一个栈,用来保存迭代调用过程中产生的大量的中间结果，当数据量增大时，可能会出现堆栈溢出的情况。

由w(n) = w(n-1) + w(n-2) && w(0) = C => 时间复杂度近似于O(2^n^)

```
fun fibber (0: int) : int * int = (1, 1)
  | fibber (n: int) : int * int =
      let val (x: int, y: int) = fibber (n-1)
      in (y, x + y)
      end
```

自底向顶计算：F0->F1->F2->F3->F4->...相同的子问题只需要计算一次，减少了空间开销和冗余计算

由w(n) = C1+w(n-1) + w(n-2) && w(0) = C0 => 时间复杂度为O(n)

作业二

1.

证明：For all L : int list, msort(L) = a <- sorted permutation of L. fun msort [ ] = [ ] | msort [x] = [x] | msort L = let val (A, B) = split L in merge (msort A, msort B) end；

列表为空时排序为空，列表只有一个元素时，排序也就是包含该元素的列表，初始情况满足条件。 假设对于任意长度小于等于n的列表，msort(L)均能产生有序排列列表，只需证明函数对于长度为n+1的列表也成立即可。 由split L函数的定义可知，当列表L的长度，A和B列表的长度均小于等于n，因此msort A和msort B返回两个有序列表。再由merge函数的定义可知，merge函数最终也会产生一个有序列表，且该有序列表中的元素跟列表 L相同，因此该结果就是列表L的一个有序排列。因此函数对于长度等于n+1的列表也成立。 综上所述，对于所有的列表L，msort(L)生成L的一个有序排列，得证。

2.

定理：对所有树t和整数y，splitAt(y,t) = 二元组(t1,t2)，满足depth(t1)<=depth(t)且 depth(t2)<=depth(t)

设P(t)表示：对所有整数y，SplitAt(y, t) = 二元组(t1, t2)，满足t1中的每一项 ≤ y 且 t2中的每一项≥ y且 t1, t2由t中元素组成。证明：对所有有序树t，P(t)成立。

证明：

当t为空时P(t)成立。假设对于深度小于等于n的有序树来说P(t)成立，证明对深度为n+1的有序树来说P(t)也成立。

```
当x>y时，SplitAt(y, t) = let val (l1, r1) = SplitAt(y,t1) in (l1, Node(r1, x, t2)) end;
```

t1为有序树t的左子树，因此t1也是有序树，且t1中的所有元素小于x。由定理知t1的深度小于t的深度即n+1。再由归纳假设可知l1中所有元素小于等于y，r1中所有元素大于等于y，l1和r1由t1中元素组成。t2中所有元素大于x，自然也大于y。所以二元组(l1, Node(r1, x, t2))中l1所有元素小于等于y，Node(r1, x, t2)中包含r1, x, t2，其中的元素均大于等于y，且两个元组的元素均来自于t。因此P(t)对于深度为n+1的树T也成立。 当x<=y同理可证。 综上所述，对所有有序树t, P(t)成立，得证。

3.

- 函数的类型为: fn: int * string list -> string list
- 函数的类型为: fn: ('a *int ->int)* 'a list -> int
- (fn x => (fn y=>x))的类型为'a->'b->'a，后面使用"Hello. World!"作为参数，最终表达式的类型为'b->string。

4.

```
(* PrefixSum: int list -> int list *)
fun prefixsum [] = []
    |prefixsum [x] = [x]
    |prefixsum (x::y::L) = x::prefixsum( (x+y)::L );
```

```
(* fastPrefixSum: int list -> int list *)
fun fastprefixsum(Z:int list):int list =
    let
        fun helper(A:int list , B:int list):int list =
            case (A,B) of
                ([],_) => B
                |(x::L,[]) => helper(L,[x])
                |(x::L,_) =>
                    let
                        val len = length B
                        val y::R = List.drop(B,len -1)
                    in
                        helper(L,B@[x+y])
                    end
    in
        helper(Z,[])
    end;

val exp1 = [1,2,3,4,5];
prefixsum exp1;
fastprefixsum exp1;
```

5.

```
(* treecompare: tree * tree -> order *)
datatype tree = Empty | Node of tree *int *tree;
fun treecompare (Empty :tree, _ :tree):order = GREATER
    |treecompare(_,Empty) = LESS
    |treecompare(Node(_,x,_),Node(_,y,_)) = Int.compare(x,y);
```

```
(* SwapDown: tree -> tree *)
fun SwapDown(Empty:tree):tree = Empty
    |SwapDown(Node(Empty,x,Empty)) = Node(Empty,x,Empty)
    |SwapDown(Node(t1,x,t2))=
        case treecompare(t1,t2) of
            LESS =>
                let
                    val Node(l,y,r) = t1
                in
                    case Int.compare(x,y) of
                        GREATER => Node(SwapDown(Node(l,x,r)),y,t2)
                        | _ => Node(t1,x,t2)
                end
            | _ =>
                let
```

```
                val Node(l,y,r) = t2
            in
                case Int.compare(x,y) of
                    GREATER => Node(t1,y,SwapDown(Node(l,x,r)))
                    | _ => Node(t1,x,t2)
            end;
```

```
(* heapify : tree -> tree *)
fun heapify(Empty:tree):tree = Empty
    |heapify(Node(t1,x,t2)) =
        SwapDown(Node(heapify(t1),x,heapify(t2)));

val exp2 = Node( Node( Node(Empty,1,Empty), 3, Node(Empty,2,Empty)), 7,
                 Node( Node(Empty,4,Empty), 6, Node(Empty,5,Empty))  );

heapify(exp2);
```

- 分析SwapDown 和heapify两个函数的work和span。

  SwapDown通过比较根节点和两个孩子的相对大小，取最小的作为新的根节点，后递归调用SwapDown维护子树。work是 O(H)（H是树的深度），span也是O(H)。

  heapify函数先对左右子树递归调用heapify函数得出最小堆，然后调用SwapDown函数将原 树变成最小堆。最坏情况下：W(H+1) = 2 * W(H) + O(H)，work是O(H^2^)。 S(H+1) = S(H) + O(H)，则span也是O(H^2^)。

作业三

2.

```
(* toInt: int -> int list -> int *)
fun toInt (b:int) =
    let fun baseToInt([]:int list):int = 0
        |baseToInt(L) =
            let
                val len = length L
                val L1 = List.take(L,len -1)
                val y::L2 = List.drop(L,len -1)
            in
                y + b * baseToInt(L1)
            end
    in baseToInt end;
```

```
(* toBase: int -> int -> int list  *)
fun toBase (b:int) =
    let fun intToBase(0:int):int list = [0]
        |intToBase(x) =
            let
                fun helper(n:int, L:int list):int list =
                case n of
                    0 => L
                    |_ => helper(n div b , (n mod b)::L )
            in
                helper(x,[])
            end
    in intToBase end;
```

```
(* convert: int * int -> int list -> int list *)
fun convert (b1:int, b2:int) =
    let
        fun B1ToB2( L:int list ):int list = toBase b2 ( toInt b1 L )
    in
        B1ToB2
    end;
```

```
val exp1 = toInt 3 (convert(2, 3) [1,0,0,1,1]);
val exp2 = toInt 2 [1,0,0,1,1];
```