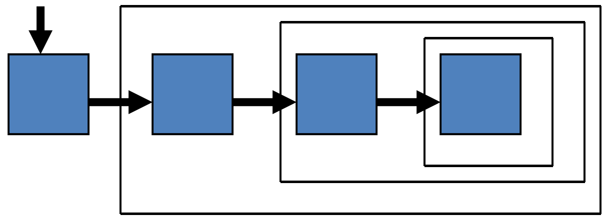
Recursion: Lab 2 examples

- **An entity <u>partially defined</u>** in terms of itself is recursively defined
- **A function <u>which conditionally calls</u>** itself is recursively defined

- Ex: Linked List



- **List ::= Head Tail | ¤**

  Head ::= element

  Tail ::= List

- **List is either empty or non-empty**

Recursion: add to a sequence

S ::= **head tail** | **empty**; **head** ::= element; **tail** ::= **S**

```
static listref be_add_val(listref L, int v)
{
   return is_empty(L)                ? create_e(v)
    :     v < get_value(head(L)) ? cons(create_e(v), L)
    :     cons(head(L), be_add_val(tail(L),v));
}
```

➔ reconstructor (cons) & deconstructors (head & tail)

Sequence pattern

1. **Empty** case
2. Non-empty, non-recursive (**head**)
3. Non-empty, recursive (**tail**)

This “forces” a certain abstract programming style

Recursion: add to a BST

- Haskell

```
bAdd v [ ]                    = v: [ ]              // empty
bAdd v [x:xs]
  | v < x                     = v : [x:xs]          // head
  | otherwise                 = x : bAdd v xs       // tail
```

Recursion: add to a BST

```
static treeref b_add(treeref T, int v)
{
  return is_empty(T)            ? create_node(v)
    : v < get_value(node(T)) ?
                                  cons(b_add(LC(T), v), node(T), RC(T))
    : v > get_value(node(T)) ?
                                 cons(LC(T), node(T), b_add(RC(T), v))
    :                                       T;
}
```

re-constructor **cons**

de-constructors **LC, node, RC**

BST pattern

1. **Empty** case
2. Non-empty case, recursive **LC**
3. Non-empty case, recursive **RC**
4. Non-empty case, non recursive **node**

This “forces” a certain abstract programming style

Recursion: remove from a BST

```
 static treeref b_rem(treeref T, int v)
{
   return is_empty(T)          ? T
   : v <  get_value(node(T)) ?
                                   cons(b_rem(LC(T), v), node(T), RC(T))
   : v >  get_value(node(T)) ?
                                   cons(LC(T), node(T), b_rem(RC(T), v))
   :                                       removeAtRoot(T);
}
```

BT ::= LC node RC | empty; node ::= element; LC, RC ::= BT

Recursion: remove from a BST

```
static  treeref removeAtRoot(treeref T)         // 00 01 10 11
{
  return  is_empty(LC(T))        ? RC(T)                 // 00 01
       :  is_empty(RC(T))        ? LC(T)                 // 10
       :  HDiff(T) > 0           ? LCmaxAsRoot(T)        // 11
       :                           RCminAsRoot(T);
  }
```

A slightly different pattern!

NB: LCmaxAsRoot & RCminAsRoot will call cons & b_rem!

Recursion: remove

- E.g. tree ((1 3 4) 5 (6 7 8)) – remove 5

- b_rem ➔ removeAtRoot (2 children) ➔

➔ RCminAsRoot (6)

➔ cons((1, 3, 4), 6, b_rem((6, 7, 8), 6))

➔ cons(b_rem((6), 6), 7, 8)

➔ removeAtRoot LC=¤, RC=¤

➔ cons(*, 7, 8)

➔ cons((1, 3, 4), 6, (*, 7, 8)) --- result

Abstraction…

- **… hides the "mechanics" of the implementation**
- **… provides a "thinking tool" for "abstract programming"**
- **…thinking in terms of references and values**
- **… requires practise**
- **… in the end is a more efficient method of programming**