3) i) Write a function which merges two ordered lists to produce an ordered list. Merging the empty list with an ordered list gives that list. To merge two non-empty lists, if the head of the first comes before the head of the second then join the head of the first onto the result of merging the tail of the first and the second. Otherwise, join the head of the second onto the result of merging the first onto the tail of the second. For example:

```
MERGE [1,4,7,9] [2,5,8] => ... =>
[1,2,4,5,7,8,9]
```

ii) Write a function which merges a list of ordered lists. For example:

```
LMERGE [[1,4,7],[2,5,8],[3,6,9]] => ... =>
[1,2,3,4,5,6,7,8,9]
```

7. COMPOSITE VALUES AND TREES

7.1. Introduction

In this chapter we are going to discuss the use of composite values to hold records of related values.

To begin with we will represent composite values as lists and process composite value sequences using linear list algorithms.

We will then introduce new notations to generalise list structure matching, to simplify list and composite value processing.

Finally, we will look at trees and consider the use of binary tree algorithms.

7.2. Composite values

So far, we have been looking at processing sequences of single values held in lists. However, for many applications, the data is a sequence of **composite values** where each consists of a number of related sub-values. These sub-values may in turn be composite so composite values may be nested.

For example, in processing a circulation list, we need to know each person's forename and surname. For example, in processing a stock control system, for each item in stock we need to know its name, the number in stock and the stock level at which it should be reordered. For example, in processing a telephone directory, we need to know each person's name, address and telephone number. Here, the name might in turn consist of a forename and surname.

Some languages provide special constructs for user defined composite values, for example the Pascal `RECORD`, the C `struct`ure and the ML tuple. These effectively add new types to the language.

Here, we are going to use lists to represent composite values. This is formally less rigorous than introducing a special construct but greatly simplifies presentation. We will look at the use of ML tuples in chapter 9.

For example, we might represent a name consisting of a string `<forename>` and a string `<surname>` as the list:

```
[<forename>,<surname>]
```

for example:

```
["Anna","Able"]
```

or

```
["Betty","Baker"].
```

For example, we might represent a stock item consisting of a string `<item name>`, an integer `<stock level>` and an integer `<reorder level>` as the list:

```
[<item name>,<stock level>,<reorder level>]
```

for example:

```
["VDU",25,10]
```

or

```
["modem",12,15].
```

For example, we might represent a telephone directory entry consisting of a composite value name as above, a string `<address>` and an integer `<number>` as a list:

```
[[<forename>,<surname>],<address>,<number>]
```

for example:

```
[["Anna","Able"],"Accounts",1212]
```

or

```
[["Betty","Baker"],"Boiler room",4242]
```

Now, a sequence of composite values will be represented by a list of lists, for example a circulation list:

```
[["Anna","Able"],
 ["Betty","Baker"],
 ["Clarice","Charlie"]]
```

or a stock list:

```
[["VDU",25,10],
 ["modem",12,15],
 ["printer,250,7]]
```

or a telephone directory:

```
[[["Anna","Able"],"Accounts",1212],
 [["Betty","Baker"],"Boiler room",4242],
 [["Clarice","Charlie"],"Customer orders",1234]]
```

7.3. Processing composite values sequences

We are using linear lists to represent composite value sequences so we will now look at the use of linear list algorithms to process them.

For example, suppose that given a circulation list we want to find someone’s forename from their surname. If the list is empty then return the empty list. If the list is not empty then if the surname matches that for the first name in the list then return the corresponding forename. Otherwise, try looking in the rest of the list.

```
rec NFIND S [] = []
 or NFIND S (H::T) =
```

```
  IF STRING_EQUAL S (HEAD (TAIL H))
  THEN HEAD H
  ELSE NFIND S T
```

For example:

```
NFIND "Charlie" [["Anna","Able"],
                 ["Betty","Baker"],
                 ["Clarice","Charlie"]] -> ... ->

NFIND "Charlie" [["Betty","Baker"],
                 ["Clarice","Charlie"]] -> ... ->

NFIND "Charlie" [["Clarice","Charlie"]] -> ... ->

"Clarice"
```

For example, given a stock list, suppose we want to find all the items which have the stock level below the reorder level. If the list is empty then return the empty list. If the list is not empty then if the first item's stock level is below the reorder level then add the first item to the result of checking the rest of the list. Otherwise, check the rest of the list:

```
rec SCHECK [] = []
 or SCHECK (H::T) =
  IF LESS (HEAD (TAIL H)) (HEAD (TAIL (TAIL H)))
  THEN H::(SCHECK T)
  ELSE SCHECK T
```

For example:

```
SCHECK [["VDU",25,12],
        ["modem",10,12],
        ["printer",125,10],
        ["mouse",7,12]] -> ... ->

SCHECK [["modem",10,12],
        ["printer",125,10],
        ["mouse",7,12]] -> ... ->

["modem",10,12]::(SCHECK [["printer",125,10],
                          ["mouse",7,12]]) -> ... ->

["modem",10,12]::(SCHECK [["mouse",7,12]]) -> ... ->

["modem",10,12]::["mouse",7,12]::(SCHECK []) -> ... ->

["modem",10,12]::["mouse",7,12]::[] ==

[["modem",10,12],
 ["mouse",7,12]]
```

For example, given a telephone directory, suppose we want to change someone's telephone number, knowing their surname. If the directory is empty then return the empty list. If the directory is not empty then if the required entry is the first then add a modified first entry to the rest of the entries. Otherwise, add the first entry to the result of looking for the required entry in the rest of the entries:

```
rec DCHANGE S N [] = []
 or DCHANGE S N (H::T) =
```

```
  IF STRING_EQUAL S (HEAD (TAIL (HEAD H)))
  THEN [(HEAD H),(HEAD TAIL H),N]::T
  ELSE H::(DCHANGE S N T)
```

For example:

```
DCHANGE "Charlie" 2424
 [[["Anna","Able"],"Accounts",1212],
  [["Betty","Baker"],"Boiler room",4242],
  [["Clarice","Charlie"],"Customer orders",1234]] => ... =>

[["Anna","Able"],"Accounts",1212]::
 (DCHANGE "Charlie" 2424
   [[["Betty","Baker"],"Boiler room",4242],
    [["Clarice","Charlie"],"Customer orders",1234]]) => ... =>

[["Anna","Able"],"Accounts",1212]::
 [["Betty","Baker"],"Boiler room",4242]::
  (DCHANGE "Charlie" 2424
    [[["Clarice","Charlie"],"Customer orders",1234]]) => ... =>

[["Anna","Able"],"Accounts",1212]::
 [["Betty","Baker"],"Boiler room",4242]::
  [["Clarice","Charlie"],"Customer orders",2424]::
   [] ==

[[["Anna","Able"],"Accounts",1212],
 [["Betty","Baker"],"Boiler room",4242],
 [["Clarice","Charlie"],"Customer orders",2424]]
```

7.4. Selector functions

Because composite values are being represented by lists, the above examples all depend on the nested use of the list selectors `HEAD` and `TAIL` to select sub-values from composite values. For composite values with many sub-values, this list selection becomes somewhat dense. Instead, we might define **selector** functions which are named to reflect the composite values that they operate on. These are particularly useful in LISP to simplify complex list expressions, because it lacks structure matching.

For example, for names we might define:

```
def FORENAME N = HEAD N

def SURNAME N = HEAD (TAIL N)
```

For example, for stock items we might define:

```
def ITEM N = HEAD N

def STOCK N = HEAD (TAIL N)

def REORDER N = HEAD (TAIL (TAIL N))
```

For example, for telephone directory entries we might define:

```
def NAME E = HEAD E

def EFORENAME E = FORENAME (NAME E)
```

```
def ESURNAME E = SURNAME (NAME E)

def ADDRESS E = HEAD (TAIL E)

def PHONE E = HEAD (TAIL (TAIL E))
```

These selector functions disguises the underlying representation and makes it easier to understand the functions that use them.

For example, given a circulation list we might want to delete a name, knowing the surname. If the list is empty then return the empty list. If the list is not empty then if the surname is that of the first name then return the rest of the list. Otherwise, add the first name to the result of deleting the required name from the rest of the list:

```
rec NDELETE S [] = []
 or NDELETE S (H::T) =
  IF STRING_EQUAL S (SURNAME H)
  THEN T
  ELSE H::(NDELETE S T)
```

For example:

```
NDELETE "Charlie" [["Anna","Able"],
                   ["Betty","Baker"],
                   ["Clarice","Charlie"]] -> ... ->

["Anna","Able"]::
 (NDELETE "Charlie" [["Betty","Baker"],
                     ["Clarice","Charlie"]]) -> ... ->
["Anna","Able"]::
 ["Betty","Baker"]::
  (NDELETE "Charlie" [["Clarice","Charlie"]]) -> ... ->

["Anna","Able"]::
 ["Betty","Baker"]::
  [] ==

[["Anna","Able"],
 ["Betty","Baker"]]
```

For example, given a stock control list, we might want to increment the stock level, knowing the item name. If the list is empty then return the empty list. If the list is not empty then if the first item is the required one then increment its stock level and add the changed item to the rest of the list. Otherwise, add the first item to the result of searching the rest of the list:

```
rec SINCREMENT I V [] = []
 or SINCREMENT I V (H::T) =
  IF STRING_EQUAL I (ITEM H)
  THEN [(ITEM H),(STOCK H)+V,(REORDER H)]::T
  ELSE H::(SINCREMENT I V T)
```

For example:

```
SINCREMENT "modem" 10 [["VDU",25,12],
                       ["modem",10,12],
                       ["printer",125,10]] -> ... ->

["VDU",25,12]::
 (SINCREMENT "modem" 10 [["modem",10,12],
```

```
                          ["printer",125,10]]) -> ... ->

["VDU",25,12]::
 ["modem",20,12]::
  [["printer",125,10]] ==

[["VDU",25,12],
 ["modem",20,12]
 ["printer",125,10]]
```

For example, given a telephone directory, we might want to add a new entry in alphabetic surname order. If the directory is empty then make a new directory from the new entry. If the directory is not empty then if the new entry comes before the first entry then add it to the front of the directory. Otherwise, add the first entry to the result of adding the new entry to the rest of the directory:

```
rec DINSERT E [] = [E]
 or DINSERT E (H::T) =
  IF STRING_LESS (ESURNAME E) (ESURNAME H)
  THEN E::H::T
  ELSE H::(DINSERT E T)
```

For example:

```
DINSERT
 [["Chris","Catnip"],"Credit",3333]
 [[["Anna","Able"],"Accounts",1212],
  [["Betty","Baker"],"Boiler room",4242],
  [["Clarice","Charlie"],"Customer orders",2424]] -> ... ->

[["Anna","Able"],"Accounts",1212]::
 (DINSERT
   [["Chris","Catnip"],"Credit",3333]
   [[["Betty","Baker"],"Boiler room",4242],
    [["Clarice","Charlie"],"Customer orders",2424]]) -> ... ->

[["Anna","Able"],"Accounts",1212]::
 [["Betty","Baker"],"Boiler room",4242]::
 (DINSERT
   [["Chris","Catnip"],"Credit",3333]
   [[["Clarice","Charlie"],"Customer orders",2424]]) -> ... ->

[["Anna","Able"],"Accounts",1212]::
 [["Betty","Baker"],"Boiler room",4242]::
  [["Chris","Catnip"],"Credit",3333]::
   [[["Clarice","Charlie"],"Customer orders",2424]] ==

[[["Anna","Able"],"Accounts",1212],
 [["Betty","Baker"],"Boiler room",4242],
 [["Chris","Catnip"],"Credit",3333],
 [["Clarice","Charlie"],"Customer orders",2424]]
```

7.5. Generalised structure matching

In chapter 6 we introduced structure matching into function definitions. Objects are defined in terms of constant base cases and structured recursion cases. Thus, function definitions have base cases with constants instead of bound variables for matching against constant arguments, and recursion cases with structured bound variable for matching against structured arguments. In particular, for list processing we have used bound variable lists of the form:

```
[]
```

for matching against the empty list, and of the form:

```
(H::T)
```

so that `H` matches the head of a list argument and `T` matches the tail. We will now allow arbitrary bound variable lists for matching against arbitrary list arguments. The bound variable lists may contain implicit or explicit empty lists for matching against empty lists in arguments.

For example, we can use structure matching to redefine the circulation list selector functions:

```
def FORENAME [F,S] = F

def SURNAME [F,S] = S
```

Here the bound variable list:

```
[F,S] == F::S::NIL
```

matches the argument list:

```
[<forename>,<surname>] == <forename>::<surname>::NIL
```

so:

```
F == <forename>
S == <surname>
```

We can also pick up the forename and surname from the first entry in a list of names by structuring matching with the bound variable list:

```
([F,S]::T)
```

so `[F,S]` matches the head of the list and `T` matches the tail. For example we might count how often a given forename occurs in a circulation list. If the list is empty the the count is 0. If the list is not empty then if the forename matches that for the first entry then add 1 to the count for the rest of the list. Otherwise, return the count for the rest of the list:

```
rec NCOUNT N [] = 0
 or NCOUNT N ([F,S]::T) =
  IF EQUAL N F
  THEN 1 + (NCOUNT N T)
  ELSE (NCOUNT N T)
```

For example, we can redefine the stock control selector functions as:

```
def ITEM [I,S,R] = I

def STOCK [I,S,R] = S

def REORDER [I,S,R] = R
```

Here, the bound variable list:

```
[I,S,R] == I::S::R::NIL
```

matches the argument:

```
[<item name>,<stock level>,<reorder level>] ==
 <itemname>::<stock level>::<reorder level>::NIL
```

so:

```
I == <item name>
S == <stock level>
R == <reorder level>
```

We can use the bound variable list:

```
([I,S,R]::T)
```

to match against a stock control list so that `[I,S,R]` matches the first item and `T` matches the rest of the list. For example, we might find all the items that need to be reordered. If the list is empty then return the empty list.If the list is not empty, if the first item needs to be reordered then add it to those to be reordered in the rest of the list. Otherwise return those to be reordered in the rest of the list:

```
rec REORD [] = 0
 or REORD ([I,S,R]::T) =
  IF LESS S R
  THEN [I,S,R]::(REORD T)
  ELSE REORD T
```

For example, we can redefine the telephone directory selector functions as:

```
def ENAME [N,A,P] = N

def EFORENAME [[F,S],A,P] = F

def ESURNAME [[F,S],A,P] = S

def ADDRESS [N,A,P] = A

def PHONE [N,A,P] = P
```

Here, the bound variable list:

```
[N,A,P] == N::A::P::NIL
```

matches the argument list:

```
[<name>,<address>,<number>] == <name>::<address>::<number>::NIL
```

so:

```
N == <name>
A == <address>
P == <number>
```

Similarly, the bound variable list:

```
[[F,S],A,P] == (F::S::NIL)::A::P::NIL
```

matches the argument list:

```
[[<forename>,<surname>],<address>,<phone>] ==
 (<forename>::<surname>::NIL)::<address>::<phone>::NIL
```

so:

```
F == <forename>
S == <surname>
```

A bound variable list of the form:

```
[N,A,P]::T
```

can be used to match a directory so `[N,A,P]` matches the first entry and `T` matches the rest of the directory. For example, we might sort the directory in telephone number order using the insertion sort from chapter 7:

```
rec DINSERT R [] = [R]
 or DINSERT [N1,A1,P1] ([N2,A2,P2]::T) =
  IF LESS P1 P2
  THEN [N1,A1,P1]::[N2,A2,P2]::T
  ELSE [N2,A2,P2]::(DINSERT [N1,A1,P1] T)

rec DSORT [] = []
 or DSORT (H::T) = DINSERT H (DSORT T)
```

7.6. Local definitions

It is often useful to introduce new name/value associations for use within an expression. Such associations are said to be **local** to the expression and are introduced by **local definitions**. Two forms of local definition are used in functional programming and they are both equivalent to function application.

Consider:

```
λ<name>.<body> <argument>
```

This requires the replacement of all free occurences of `<name>` in `<body>` with `<argument>` before `<body>` is evaluated.

Alternatively, `<name>` and `<argument>` might be thought of as being associated throughout the evaluation of `<body>`. This might be written in a bottom up style as:

```
let <name> = <argument>
in <body>
```

or in a top down style as:

```
<body>
where <name> = <argument>
```

We will use the bottom up `let` form of local definition on the grounds that things should be defined before they are used.

7.7. Matching composite value results

The use of bound variable lists greatly simplifies the construction of functions which return composite value results represented as lists.

For example, suppose we have a list of forename/surname pairs and we wish to split it into separate lists of forenames and surnames. We could scan the list to pick up the forenames and again to pick up the surnames. Alternatively, we can pick them both up at once.

To split an empty list, return an empty forename list and an empty surname list: Otherwise, split the tail and put the forename from the head pair onto the forename list from the tail and the surname from the head pair onto the surname list from the tail:

```
rec SPLIT [] = []::[]
 or SPLIT ([F,S]::L) =
  let (FLIST::SLIST) = SPLIT L
  in  ((F::FLIST)::(S::SLIST))
```

Note that at each stage `SPLIT` is called recursively on the tail to return a list of lists. This is then separated into the lists `FLIST` and `SLIST`, the items from the head pair are added and a new list of lists is returned.

For example:

```
SPLIT [["Allan","Ape"],["Betty","Bat"],["Colin","Cat"]] => ... =>

let (FLIST::SLIST) = SPLIT [["Betty","Bat"],["Colin","Cat"]]
in (("Allan"::FLIST)::("Ape"::SLIST))
```

The first recursive call to `SPLIT` involves:

```
 SPLIT [["Betty","Bat","Colin","Cat"] => ... =>

 let (FLIST::SLIST) = SPLIT [["Colin","Cat"]]
 in (("Betty"::FLIST)::("Bat"::SLIST))
```

The second recursive call to `SPLIT` involves:

```
  SPLIT [["Colin","Cat"]]  => ... =>

  let (FLIST::SLIST) = SPLIT []
  in (("Colin"::FLIST)::("Cat"::SLIST))
```

The third recursive call to `SPLIT` is the last:

```
   SPLIT [] => ... =>

   []::[]
```

so the recursive calls start to return:

```
  let (FLIST::SLIST) = []::[]
  in (("Colin"::FLIST)::("Cat"::SLIST)) => ... =>

  (("Colin"::[])::("Cat"::[])) ==

  (["Colin"]::["Cat"]) => ... =>

 let (FLIST::SLIST) = (["Colin"]::["Cat"])
 in (("Betty"::FLIST)::("Bat"::SLIST)) => ... =>

 (("Betty"::["Colin"])::("Bat"::["Cat"])) ==

 (["Betty","Colin"]::["Bat","Cat"]) => ... =>
```

```
let (FLIST::SLIST) = (["Betty","Colin"]::["Bat","Cat"])
in (("Allan"::FLIST)::("Ape"::SLIST)) => ... =>

(("Allan"::["Betty","Colin"])::("Ape"::["Bat","Cat")]) ==

(["Allan","Betty","Colin"]::["Ape","Bat","Cat"])
```

We can simplify this further by making the local variables `FLIST` and `SLIST` additional bound variables:

```
rec SPLIT [[]] L = L
 or SPLIT ([F,S]::L) (FLIST::SLIST) = SPLIT L ((F::FLIST)::(S::SLIST))
```

Now, on the recursive call, the variables `FLIST` and `SLIST` will pick up the lists `(F::FLIST)` and `(S::SLIST)` from the previous call. Initially, `FLIST` and `LIST` are both empty. For example:

```
SPLIT [["Diane","Duck"],["Eric","Eagle"],["Fran","Fox"]] ([]::[]) => ... =>

SPLIT [["Eric","Eagle"],["Fran","Fox"]] (["Diane"]::["Duck"]) => ... =>

SPLIT [["Fran","Fox"]] (["Eric","Diane"]::["Eagle","Duck"]) => ... =>

SPLIT [[]] (["Fran","Eric","Diane"]::["Fox","Eagle","Duck"]) => ... =>

(["Fran","Eric","Diane"]::["Fox","Eagle","Duck"])
```

Note that we have picked up the list components in reverse order because we've added the heads of the argument lists into the new lists before processing the tails of the argument lists.

The bound variables `FLIST` and `SLIST` are known as **accumulation variables** because they are used to accumulate partial results.

7.8. List inefficiency

Linear lists correspond well to problems involving flat sequences of data items but are relatively inefficient to access and manipulate. This is because accessing an item always involves skipping past the preceding items. In long lists this becomes extremely time consuming. For a list with N items, if we assume that each item is just as likely to be accessed as any other item then:

to access the 1st item, skip 0 items;
to access the 2nd item, skip 1 item;
...
to access the N-1th item, skip N-2 items;
to access the Nth item, skip N-1 items

Thus, on average it is necessary to skip:

(1+...(N-2)+(N-1))/N = (N*N/2)/N = N/2

items. For example, to find one item in a list of 1000 items it is necessary to skip 500 items on average.

Sorting using the insertion technique above is far worse. For a worst case sort with N items in complete reverse order then:

to place the 1st item, skip 0 items;
to place the 2nd item, skip 1 item;
...
to place the N-1th item, skip N-2 items;

to place the Nth item, skip N-1 items

Thus, in total it is necessary to skip:

1+...(N-2)+(N-1) = N*N/2

items. For example, for a worst case sort of 1000 items it is necessary to skip 500000 items.

Remember, for searching and sorting in a linear list, each skip involves a comparison between a list item and a required or new item. If the items are strings then comparison is character by character so the number of comparisons can get pretty big for relatively short lists.

Note that we have been considering naive linear list algorithms. For particular problems, if there is a known ordering on a sequence of values then it may be possible to represent the sequence as an ordered list of ordered sub-sequences. For example, a sequence of strings might be represented as list of ordered sub-lists, with a sub-list for each letter of the alphabet.

7.9. Trees

Trees are general purpose nested structures which enable far faster access to ordered sequences than linear lists. Here we are going to look at how trees may be modelled using lists. To begin with, we will introduce the standard tree terminology.

A tree is a nested data structure consisting of a hierarchy of **nodes.** Each node holds one data item and has **branches** to **sub-trees** which are in turn composed of nodes. The first node in a tree is called the **root.** A node with empty branches is called a **leaf.** Often, a tree has the same number of branches in each node. If there are N branches then the tree is said to be **N-ary**.

If there is an ordering relationship on the tree then each sub-tree consists of nodes whose items have a common relationship to the original node's item. Note that ordering implies that the node items are all the same type. Ordered linear sequences can be held in a tree structure which enables far faster access and update.

We are now going to look specifically at **binary** trees. A binary tree node has two branches, called the **left** and **right** branches, to binary sub-trees. Formally, the empty tree, which we will denote as `EMPTY`, is a binary tree:

```
EMPTY is a binary tree
```

and a tree consisting of a node with an item and two sub-trees is a binary tree if the sub-trees are binary trees:

```
NODE ITEM L R is a binary tree
if L is a binary tree and R is a binary tree
```

We will model a binary tree using lists. We will represent `EMPTY` as `NIL`:

```
def EMPTY = NIL

def ISEMPTY = ISNIL
```

and a node as a list of the item and the left and right branches:

```
def NODE ITEM L R = [ITEM,L,R]
```

The item and the sub-trees may be selected from nodes:

```
ITEM (NODE I L R) = I
LEFT (NODE I L R) = L
RIGHT (NODE I L R) = R
```

but no selection may be made from empty trees:

```
ITEM EMPTY = TREE_ERROR
LEFT EMPTY = TREE_ERROR
RIGHT EMPTY = TREE_ERROR
```

Note that we cannot use these equations directly as functions as we have not introduced trees as a new type into our notation. Instead, we will model tree functions with list functions using `LIST_ERROR` for `TREE_ERROR`:

```
def TREE_ERROR = LIST_ERROR

def ITEM EMPTY = TREE_ERROR
 or ITEM [I,L,R] = I

def LEFT EMPTY = TREE_ERROR
 or LEFT [I,L,R] = L

def RIGHT EMPTY = TREE_ERROR
 or RIGHT [I,L,R] = R
```

Note that we can use `EMPTY` in structure matching because it is the same as `NIL.`

7.10. Adding values to ordered binary trees

In an ordered binary tree, the left sub-tree contains nodes whose items come before the original node's item in some ordering and the right sub-tree, contains nodes whose items come after the original node's item in that ordering. Each sub-tree is itself an ordered binary tree.

Thus, to add an item to an ordered binary tree, if the tree is empty then make a new node with empty branches for the item:

```
TADD I EMPTY = NODE I EMPTY EMPTY
```

If the item comes before the root node item then add it to the left sub-tree:

```
TADD I (NODE NI L R) = NODE NI (TADD I L) R
                       if <less> I NI
```

Otherwise, add it to the right sub-tree:

```
TADD I (NODE NI L R) = NODE NI L (TADD I L)
                       if NOT (<less> I NI)
```

For example, for a binary tree of integers:

```
rec TADD I EMPTY =  NODE I EMPTY EMPTY
 or TADD I [NI,L,R] =
  IF LESS I NI
  THEN NODE NI (TADD I L) R
  ELSE NODE NITEM L (TADD I R)
```

For example, to add 7 to an empty tree:

```
TADD 7 EMPTY
```

The tree is empty so a new node is constructed:

```
[7,EMPTY,EMPTY]
```

To add 4 to this tree:

```
TADD 4 [7,EMPTY,EMPTY]
```

4 comes before 7 so it is added to the left sub-tree:

```
[7,(TADD 4 EMPTY),EMPTY] -> ... ->

[7,
 [4,EMPTY,EMPTY],
 EMPTY
]
```

To add 9 to this tree:

```
TADD 9 [7,
        [4,EMPTY,EMPTY],
        EMPTY
       ]
```

9 comes after 7 so it is added to the right sub-tree:

```
[7,
 [4,EMPTY,EMPTY],
 (TADD 9 EMPTY)
] -> ... ->

[7,
 [4,EMPTY,EMPTY],
 [9,EMPTY,EMPTY]
]
```

To add 3 to this tree:

```
TADD 3 [7,
        [4,EMPTY,EMPTY],
        [9,EMPTY,EMPTY]
       ]
```

3 comes before 7 so it is added to the left sub-tree:

```
[7,
 (TADD 3 [4,EMPTY,EMPTY]),
 [9,EMPTY,EMPTY]
]
```

3 comes before 4 so it is added to the left sub-tree:

```
[7,
 [4,
  (TADD 3 EMPTY),
  EMPTY
 ],
 [9,EMPTY,EMPTY]
] -> ... ->
```

```
[7,
 [4,
  [3,EMPTY,EMPTY],
  EMPTY
 ],
 [9,EMPTY,EMPTY]
]
```

To add 5 to this tree:

```
TADD 5 [7,
        [4,
         [3,EMPTY,EMPTY],
         EMPTY
        ],
        [9,EMPTY,EMPTY]
       ]
```

5 comes before 7 so it is added to the left sub-tree:

```
[7,
 (TADD 5 [4,
          [3,EMPTY,EMPTY],
          EMPTY
         ]),
 [9,EMPTY,EMPTY]
]
```

Now, 5 comes after 4 so it is added to the right sub-tree:

```
[7,
 [4,
  [3,EMPTY,EMPTY],
  (TADD 5 EMPTY)
 ],
 [9,EMPTY,EMPTY]
] -> ... ->

[7,
 [4,
  [3,EMPTY,EMPTY],
  [5,EMPTY,EMPTY]
 ],
 [9,EMPTY,EMPTY]
]
```

To add an arbitrary list of numbers to a tree, if the list is empty then return the tree. Otherwise add the tail of the list to the result of adding the head of the list to the tree:

```
rec TADDLIST [] TREE = TREE
 or TADDLIST (H::T) TREE = TADDLIST T (TADD H TREE)
```

Thus:

```
TADDLIST [7,4,9,3,5,11,6,8] EMPTY -> ... ->

[7,
 [4,
```

```
  [3,EMPTY,EMPTY],
  [5,
   EMPTY,
   [6,EMPTY,EMPTY]
  ]
 ],
 [9,
  [8,EMPTY,EMPTY],
  [11,EMPTY,EMPTY]
 ]
]
```

7.11. Binary tree traversal

Having added values to an ordered tree it may be useful to extract them in some order. This involves **walking** or **traversing** the tree picking up the node values. From our definition of an ordered binary tree, all the values in the left sub-tree for a node are less than the node value and all the values in the right sub-tree for a node are greater than the node value. Thus, to extract the values in ascending order we need to traverse the left sub-tree, pick up the node value and then traverse the right subtree:

```
TRAVERSE (NODE I L R) = APPEND (TRAVERSE L) (I::(TRAVERSE R))
```

Traversing an empty tree returns an empty list:

```
TRAVERSE EMPTY = []
```

Using lists instead of a tree type:

```
rec TRAVERSE EMPTY = []
 or TRAVERSE [I,L,R] = APPEND (TRAVERSE L) (I::(TRAVERSE R))
```

We will illustrate this with an example. To ease presentation, we may evaluate several applications at the same time at each stage:

```
TRAVERSE [7,
          [4,
           [3,EMPTY,EMPTY],
           [5,EMPTY,EMPTY]
          ],
          [9,EMPTY,EMPTY]
         ] -> ... ->

APPEND (TRAVERSE [4,
                  [3,EMPTY,EMPTY],
                  [5,EMPTY,EMPTY]
                 ])
       (7::(TRAVERSE [9,EMPTY,EMPTY])) -> ... ->

APPEND (APPEND (TRAVERSE [3,EMPTY,EMPTY])
               (4::(TRAVERSE [5,EMPTY,EMPTY])))
       (7::(APPEND (TRAVERSE EMPTY)
                   (9::(TRAVERSE EMPTY)))) -> ... ->

APPEND (APPEND (APPEND (TRAVERSE EMPTY)
                       (3::(TRAVERSE EMPTY)))
               (4::(APPEND (TRAVERSE EMPTY)
                           (5::(TRAVERSE EMPTY))))
```

```
       (7::(APPEND (TRAVERSE EMPTY)
                   (9::(TRAVERSE EMPTY)))) -> ... ->

APPEND (APPEND (APPEND []
                       (3::[]))
               (4::(APPEND []
                           (5::[])))
       (7::(APPEND []
                   (9::[]))) -> ... ->

APPEND (APPEND [3]
               (4::[5]))
       (7::[9]) -> ... ->

APPEND [3,4,5] [7,9] -> ... ->

[3,4,5,7,9]
```

7.12. Binary tree search

Once a binary tree has been constructed it may be searched to find out whether or not it contains a value. The search algorithm is very similar to the addition algorithm above. If the tree is empty then the search fails:

```
TFIND V EMPTY = FALSE
```

If the tree is not empty, and the required value is the node value then the search succeeds:

```
TFIND V (NODE NV L R) = TRUE if <equal> V NV
```

Otherwise, if the requiredvalue comes before the node value then try the left branch:

```
TFIND V (NODE NV L R) = TFIND V L if <less> V NV
```

Otherwise, try the right branch:

```
TFIND V (NODE NV L R) = TFIND V R if NOT (<less> V NV)
```

For example, for a binary integer tree:

```
rec TFIND V EMPTY = ""
 or TFIND V [NV,L,R] =
  IF EQUAL V NV
  THEN TRUE
  ELSE
   IF LESS V NV
   THEN TFIND V L
   ELSE TFIND V R
```

For example:

```
TFIND 5 [7,
         [4,
          [3,EMPTY,EMPTY],
          [5,EMPTY,EMPTY]
         ],
         [9,EMPTY,EMPTY]
        ] -> ... ->
```

```
TFIND 5 [4,
         [3,EMPTY,EMPTY],
         [5,EMPTY,EMPTY]
        ] -> ... ->

TFIND 5 [5,EMPTY,EMPTY] -> ... ->

TRUE
```

For example:

```
TFIND 2 [7,
         [4,
          [3,EMPTY,EMPTY],
          [5,EMPTY,EMPTY]
         ],
         [9,EMPTY,EMPTY]
        ] -> ... ->

TFIND 2 [4,
         [3,EMPTY,EMPTY],
         [5,EMPTY,EMPTY]
        ] -> ... ->

TFIND 2 [3,EMPTY,EMPTY] -> ... ->

TFIND 2 EMPTY -> ... ->

FALSE
```

7.13. Binary trees of composite values

Binary trees, like linear lists, may be used to represent ordered sequences of composite values. Each node holds one composite value from the sequence and the ordering is determined by one sub-value.

The tree addition functions above may be modified to work with composite values. For example, we might hold the circulation list of names in a binary tree in surname order. Adding a new name to the tree involves comparing the new surname with the node surnames:

```
rec CTADD N EMPTY = [N,EMPTY,EMPTY]
 or CTADD [F,S] [[NF,NS],L,R] =
  IF STRING_LESS S NS
  THEN [[NF,NS],(CTADD [F,S] L),R]
  ELSE [[NF,NS],L,(CTADD [F,S] R)]

rec CTADDLIST [] TREE = TREE
 or CTADDLIST (H::T) TREE = CTADDLIST T (CTADD H TREE)
```

For example:

```
CTADDLIST
 [["Mark","Monkey"],
  ["Graham","Goat"],
  ["Quentin","Quail"],
  ["James","Jaguar"],
  ["David","Duck]] EMPTY -> ... ->
```

```
[["Mark","Monkey"],
  [["Graham","Goat"],
    [["David","Duck"],EMPTY,EMPTY],
    [["James","Jaguar"],EMPTY,EMPTY]
  ],
  [["Quentin","Quail"],EMPTY,EMPTY]
]
```

The tree traversal function above may be applied to binary trees with arbitrary node values as it only inspects branches during traversal. For example:

```
TRAVERSE [["Mark","Monkey"],
           [["Graham","Goat"],
             [["David","Duck"],EMPTY,EMPTY],
             [["James","Jaguar"],EMPTY,EMPTY]
           ],
           [["Quentin","Quail"],EMPTY,EMPTY]
         ] -> ... ->

[["David","Duck"],
 ["Graham","Goat"],
 ["James","Jaguar"],
 ["Mark","Monkey"],
 ["Quentin","Quail"]]
```

Finally, the tree search function above may be modified to return some or all of a required composite value. For example, we might find the forename corresponding to a surname, using the surname to identify the required node:

```
rec CTFIND S EMPTY = ""
 or CTFIND S [[NF,NS],L,R] =
  IF STRING_EQUAL S NS
  THEN NF
  ELSE
   IF STRING_LESS S NS
   THEN CTFIND S L
   ELSE CTFIND S R
```

For example:

```
CTFIND "Duck" [["Mark","Monkey"],
                [["Graham","Goat"],
                  [["David","Duck"],EMPTY,EMPTY],
                  [["James","Jaguar"],EMPTY,EMPTY]
                ],
                [["Quentin","Quail"],EMPTY,EMPTY]
              ] -> ... ->

CTFIND "Duck" [["Graham","Goat"],
                  [["David","Duck"],EMPTY,EMPTY],
                  [["James","Jaguar"],EMPTY,EMPTY]
              ] -> ... ->

CTFIND "Duck" [["David","Duck"],EMPTY,EMPTY]

"David"
```

7.14. Binary tree efficiency

When an ordered binary tree is formed from a value sequence, each node holds an ordered sub-sequence. Every sub-node on the left branch of a node contains values which are less than the node's value and every sub-node on the right branch contains values which are greater than the node's value. Thus, when searching a tree for a node given a value, the selection of one branch discounts all the sub-nodes, and hence all the values, on the other branch. The number of comparisons required to find a node depends on how many layers of sub-nodes there are between the root and that node.

A binary tree is said to be **balanced** if for any node, the number of values in both branches is the same. For a balanced binary tree, if a node holds `N` values then there are `(N-1)/2` values in its left branch and `(N-1)/2` in the its right branch. Thus, the total number of branch layers depends on how often the number of values can be halved. This suggests that in general, if:

$$2^L <= N < 2^{L+1}$$

then:

$$N \text{ values} == \log_2(N)+1 == L+1 \text{ layers}$$

For example:

$$\begin{aligned}
&1 \text{ value} == 1 \text{ node} && == 1 \text{ layer} == \log_2(1)+1\\
&3 \text{ values} == 1 \text{ node} + 2 * 1 \text{ value} && == 2 \text{ layers} == \log_2(3)+1\\
&7 \text{ values} == 1 \text{ node} + 2 * 3 \text{ values} && == 3 \text{ layers} == \log_2(7)+1\\
&15 \text{ values} == 1 \text{ node} + 2 * 7 \text{ values} && == 4 \text{ layers} == \log_2(15)+1\\
&31 \text{ values} == 1 \text{ node} + 2 * 15 \text{ values} && == 5 \text{ layers} == \log_2(31)+1\\
&63 \text{ values} == 1 \text{ node} + 2 * 31 \text{ values} && == 6 \text{ layers} == \log_2(63)+1\\
&\ldots
\end{aligned}$$

For example, for a balanced tree of 1000 items it is necessary to go down 10 layers, making 10 comparisons, in the worst case.

Note that we have considered perfectly balanced trees. However, the algorithms discussed above do not try to maintain balance and so the 'shape' of a tree depends on the order in which values are added. In general, trees built with our simple algorithm will not be balanced. Indeed, in the worst case the algorithm builds a linear list, ironically, when the values are already in order:

```
TADDLIST [4,3,2,1] -> ... ->

[1,
 EMPTY,
 [2,
  EMPTY,
  [3,
   EMPTY,
   [4,EMPTY,EMPTY]
  ]
 ]
]
```

We will not consider the construction of balanced trees here.

7.15. Curried and uncurried functions

In imperative languages like Pascal and C we use procedures and functions declared with several formal parameters and we cannot separate a procedure or function from the name it is declared with. Here, however, all our functions are built from nested λ functions with single bound variables: names and definitions of name/function associations are just a convenient simplification. Our notation for function definitions and applications has led us to treat a name associated with a nested function as if it were a function with several bound variables. Now we have introduced another form of multiple bound variables through bound variable lists.

In fact, nested functions of single bound variables and functions with multiple bound variables are equivalent. The technique of defining multi-parameter functions as nested single parameter functions was popularised by the American mathematician Haskell Curry and nested single parameter functions are called **curried** functions.

We can construct functions to transform a curried function into an uncurried function and vice versa. For a function `f` with a bound variable list containing two bound variables:

```
def curry f x y = f [x,y]
```

will convert from uncurried form to curried form.

For example, consider:

```
def SUM_SQ1 [X,Y] = (X*X)+(Y*Y)
```

Then for:

```
def curry_SUM_SQ = curry SUM_SQ1
```

the right hand side expands as:

```
λf.λx.λy.(f [x,y]) SUM_SQ1 =>

λx.λy.(SUM_SQ1 [x,y])
```

so:

```
def curry_SUM_SQ x y = SUM_SQ1 [x,y]
```

Now, the use of `curry_SUM_SQ` with nested arguments is the same as the use of `SUM_SQ1` with an argument list.

Similarly, for a function `g` with a single bound variable `a,` which returns a function with single bound variable `b`

```
def uncurry g [a,b] = g a b
```

will convert from curried form to uncurried form.

For example, with

```
def SUM_SQ2 X Y = (X*X)+(Y*Y)
```

then for:

```
def uncurry_SUM_SQ = uncurry SUM_SQ2
```

the right hand side expands as:

```
λg.λ[a,b].(g a b) SUM_SQ2 =>
```

```
λ[a,b].(SUM_SQ2 a b)
```

so:

```
def uncurry_SUM_SQ [a,b] = SUM_SQ2 a b
```

Now, the use of `uncurry_SUM_SQ` with an argument list is equivalent to the use of `SUM_SQ2` with nested arguments.

The functions `curry` and `uncurry` are inverses. For an arbitrary function:

```
<function>
```

consider:

```
uncurry (curry <function>) ==

λg.λ[a,b].(g a b) (λf.λx.λy.(f [x,y]) <function>) ->

λg.λ[a,b].(g a b) λx.λy.(<function> [x,y]) =>

λ[a,b].(λx.λy.(<function> [x,y]) a b)
```

which simplifies to:

```
λ[a,b].(<function> [a,b]) ==

<function>
```

Here we have used a form of η reduction to simplify:

```
λ[<name1>,<name2>]).(<expression> [<name1>,<name2>])
```

to:

```
<expression>
```

Similarly:

```
curry (uncurry <function>) ==

λf.λx.λy.(f [x,y]) (λg.λ[a,b].(g a b) <function>) ->

λf.λx.λy.(f [x,y]) (λ[a,b].(<function a b) =>

λx.λy.(λ[a,b].(<function> a b) [x,y])
```

which simplifies to:

```
λx.λy.(<function> x y) ==

<function>
```

Again we have used a form of η reduction to simplify:

```
λ<name1>.λ<name2>.(<expression> <name1> <name2>)
```

to:

```
<expression>.
```

7.16. Partial application

We have been using a technique which is known as **partial application** where a multi-parameter function is used to construct another multi-parameter function by providing arguments for only some of the parameters. We have taken this for granted because we use nested single bound variable functions. For example, in chapter 5 we defined the function:

```
def istype t obj = equal t (type obj)
```

to test an objects type against an arbitrary type `t` and then constructed:

```
def isbool = istype bool_type

def isnumb = istype numb_type

def ischar = istype char_type

def islist = istype list_type
```

to test whether an object was a boolean, number, character or list by 'filling in' the bound variable `t` with an appropriate argument. This creates no problems for us because `istype` is a function of one bound variable which returns a function of one bound variable. However, many imperative languages with multi-parameter procedures and functions do not usually allow procedures and functions as objects in their own right, (POP-2 and PS-algol are exceptions), and so partial application is not directly available. However, an equivalent form is based on defining a new function or procedure with less parameters which calls the original procedure or function with some of its parameters 'filled in'.

For example, had `istype` been defined in Pascal as:

```
FUNCTION ISTYPE(T:TYPEVAL,OBJ:OBJECT):BOOLEAN
BEGIN
     ISTYPE := (T = TYPE(OBJ))
END,
```

assuming that Pascal allowed the construction of appropriate types, then `ISBOOL` might be defined by:

```
FUNCTION ISBOOL(O:OBJECT):BOOLEAN
BEGIN
     ISBOOL := ISTYPE(BOOL_TYPE,O)
END

and
ISNUMB
by:

FUNCTION ISNUMB(OBJ:OBJECT):BOOLEAN
BEGIN
     ISNUMB := ISTYPE(NUMB_TYPE,OBJ)
END
```

and so on. In our notation, it is as if we had defined:

```
def istype (t::o)= equal t (type o)

def isbool obj = istype (bool_type::obj)

def isnumb obj = istype (numb_type::obj)
```

and so on.

Here, we have explicitly provided a value for the second bound variable `o` from the new single bound variable `obj.`

In practical terms, curried functions are more flexible to use but less efficient to implement as more function entry and exit is necessitated.

7.17. Summary

In this chapter we have:

- represented composite values as lists
- developed selector functions to simplify composite value manipulation
- introduced generalised list structure matching
- introduced notations for local definitions
- considered the efficiency of naive linear list algorithms
- represented trees as lists
- developed functions to manipulate ordered binary trees
- considered the efficiency of binary trees
- met curried and uncurried functions, and partial application

Some of these topics arte summarised below.

Generalised structure matching

```
def <name> [<name1>,<name2>,<name3> ... ] =
      <expression using ‘<name1>’, ‘<name2>’, ‘<name3>’ ... >

def <name> <bound variable> =
      <expression using ‘HEAD <bound variable>’,
                        ‘HEAD (TAIL <bound variable>)’,
                        ‘HEAD (TAIL (TAIL <bound variable>))’ ... >
```

Local definitions

```
let <name> = <expression1>
in <expression2> ==

<expression2>
where <name> = <expression> ==

λ<name>.<expression2> <expression1>
```

Curried and uncurried functions

```
λ<name1>.λ<name2>...λ<nameN>.<body> ==

λ[<name1>,<name2> ... <nameN>].<body>
```

7.18. Exercises

1) The time of day might be represented as a list with three integer fields for hours, minutes and seconds:

```
[<hours>,<minutes>,<seconds>]
```

For example:

```
[17,35,42] == 17 hours 35 minutes 42 seconds
```

Note that:

```
24 hours = 0 hours
1 hour == 60 minutes
1 minute == 60 seconds
```

i) Write functions to convert from a time of day to seconds and from seconds to a time of day. For example:

```
TOO_SECS [2,30,25] => ... => 9025

FROM_SECS 48975 => ... => [13,36,15]
```

ii) Write a function which increments the time of day by one second. For example:

```
TICK [15,27,18] => ... => [15,27,19]
TICK [15,44,59] => ... => [15,45,0]
TICK [15,59,59] => ... => [16,0,0]
TICK [23,59,59] => ... => [0,0,0]
```

iii) In a shop, each transaction at a cash register is time stamped. Given a list of transaction details, where each is a string followed by a time of day, write a function which sorts them into ascending time order. For example:

```
TSORT [["haggis",[12,19,57]],
       ["clouty dumpling",[18,22,48]],
       ["white pudding",[10,12,35]],
       ["oatcakes",[15,47,19]]] => ... =>

[["white pudding",[10,12,35]],
 ["haggis",[12,19,57]],
 ["oatcakes",[15,47,19]]
 ["clouty dumpling",[18,22,48]]]
```

2) i) Write a function which compares two integer binary trees.

ii) Write a function which indicates whether or not one integer binary tree contains another as a sub-tree.

iii) Write a function which traverses a binary tree to produce a list of node values in descending order.

3) Strictly bracketed integer arithemtic expressions:

```
<expression> ::= (<expression> + <expression>) |
                 (<expression> + <expression>) |
                 (<expression> + <expression>) |
                 (<expression> + <expression>) |
                 <number>
```

might be represented by a nested list structure so:

```
(<expression1> + <expression2>) == [<expression1>,"+",<expression2>]
(<expression1> - <expression2>) == [<expression1>,"-",<expression2>]
(<expression1> * <expression2>) == [<expression1>,"*",<expression2>]
(<expression1> / <expression2>) == [<expression1>,"/",<expression2>]
<number> == <number>
```

For example:

```
3 == 3
(3 * 4) == [3,"*",4]
((3 * 4) - 5) == [[3,"*",4],"-",5]
((3 * 4) - (5 + 6)) == [[3,"*",4],"-",[5,"+",6]]
```

Write a function which evaluates a nested list representation of an arithmetic expression. For example:

```
EVAL 3 => ... => 3
EVAL [3,"*",4] => ... => 12
EVAL [[3,"*",4],"-",5] => ... => 7
EVAL [[3,"*",4],"-",[5,"+",6]] => ... => 11
```

8. EVALUATION

8.1. Introduction

In this chapter we are going to look at evaluation order in more detail.

First of all we will consider the relative merits of applicative and normal order evaluation and see that applicative order is generally more efficient than normal order. We will also see that applicative order evaluation may lead to non-terminating evaluation sequences where normal order evaluation terminates, with our representations of conditional expressions and recursion.

We will then see that the halting problem is undecideable in undecidable so it is not possible to tell whether or not the evaluation of an arbitrary λ expression terminates. We will also survey the Church-Rosser theorems which show that normal and applicative order evaluation order are equivalent but that normal order is more likely to terminate.

Finally, we will look at lazy evaluation which combines the best features of normal and applicative orders.

8.2. Termination and normal form

A lambda expression which cannot be reduced any further is said to be in **normal form**. Our definition of β reduction in chapter 2 implied that evaluation of an expression terminates when it is no longer a function application. This won't reduce an expression to normal form. Technically we should go on evaluating the function body until it contains no more function applications. Otherwise, expressions which actually reduce to the same normal form appear to be different. For example, consider: