```
def signed_type = ...
def SIGN_ERROR = MAKE_ERROR signed_type
def POS = TRUE
def NEG = FALSE
def MAKE_SIGNED N SIGN = make_obj signed_type (make_obj SIGN N)
```

So:

```
+<number> == MAKE_SIGNED <number> POS
-<number> == MAKE_SIGNED <number> NEG
```

For example:

```
+4 == MAKE_SIGNED 4 POS
-4 == MAKE_SIGNED 4 NEG
```

Note that there are two representations for 0:

```
+0 == MAKE_SIGNED 0 POS
-0 == MAKE_SIGNED 0 NEG
```

i) Define tester and selector functions for signed numbers:

```
def issigned N = ...      - true if N is a signed number
def ISSIGNED N = ...      - TRUE if N is a signed number
def sign N = ...          - N's sign as an untyped number
def SIGN N = ...          - N's sign as a typed number
def sign_value N = ...    - N's value as an unsigned number
def VALUE N = ...         - N's value as a signed number
def sign_iszero N = ...   - true if N is 0
```

Show that your functions work for representative positive and negative values, and 0.

ii) Define signed versions of `ISZERO, SUCC` and `PRED:`

```
def SIGN_ISZERO N = ...
def SIGN_SUCC N = ...
def SIGN_PRED N = ...
```

Show that your functions work for representative positive and negative values, and 0.

iii) Define a signed versions of '+':

```
def SIGN_+ X Y = ...
```

Show that your function works for representative positive and negative values, and 0.

6. LISTS AND STRINGS

6.1. Introduction

In this chapter we are going to look at the list data structure which is used to hold variable length sequences of values.

To begin with, we will discuss list construction and list element access. We will then use pair functions and the type representation techniques to add lists to our notation.

Next, we will develop elementary functions for manipulating linear lists and simpler list notations. We will also introduce strings as lists of characters.

We will then introduce structure matching for list functions and look at a variety of operations on linear lists. In imperative languages, we might use arrays in these applications.

Finally, we will consider at the use of mapping functions to generalise operations on linear lists.

6.2. Lists

Lists are general purpose data structures which are widely used within functional and logic programming. They were brought to prominence through LISP and are now found in various forms in many languages. Lists may be used in place of both arrays and record structures, for example to build stacks, queues and tree structures.

In this chapter, we are going to introduce lists into our functional notation and look at using lists in problems where arrays might be used in other languages. In chapter 7 we will look at using lists where record structures might be used in other languages.

Lists are variable length sequences of objects. A strict approach to lists, as in ML for example, treats them as variable length sequences of objects of the same type. We will take the relatively lax approach of LISP and Prolog and treat them as variable length sequences of objects of mixed type. Although this is less rigorous theoretically and makes a formal treatment more complex it simplifies presentation and provides a more flexible structure.

Formally, a list is either empty, denoted by the unique object:

```
NIL is a list
```

or it is a constructed pair with a **head** which is any object and a **tail** which is a list:

```
CONS H T is a list
if H is any object and T is a list
```

`CONS` is the traditional name for the list constructor, originally from LISP.

For example, from the object `3` and the list `NIL` we can construct the list:

```
CONS 3 NIL
```

with `3` in the head and `NIL` in the tail.

From the object `2` and this list we can construct the list:

```
CONS 2 (CONS 3 NIL)
```

with `1` in the head and the list `CONS 3 NIL` in the tail.

From the object `1` and the previous list we can construct the list:

```
CONS  1 (CONS 2 (CONS 3 NIL))
```

with `1` in the head and the list `CONS 2 (CONS 3 NIL))` in the tail, and so on.

Note that the tail of a list must be a list. Thus, all lists end, eventually, with the empty list.

Note that the head of a list may be any object including another list, enabling the construction of nested structures, particularly trees.

If the head of a list and the heads of all of its tail lists are not lists then the list is said to be **linear.** In LISP parlance, an object in a list which is not a list (or a function) is known as an **atom.**

The head and tail may be selected from non-empty lists:

```
HEAD (CONS H T) = H
TAIL (CONS H T) = T
```

but head and tail selection from an empty list is not permitted:

```
HEAD NIL = LIST_ERROR
TAIL NIL = LIST_ERROR
```

Consider, for example, the linear list of numbers we constructed above:

```
CONS 1 (CONS 2 (CONS 3 NIL))
```

The head of this list:

```
HEAD (CONS 1 (CONS 2 (CONS 3 NIL)))
```

is:

```
1
```

The tail of this list:

```
TAIL (CONS 1 (CONS 2 (CONS 3 NIL)))
```

is:

```
CONS 2 (CONS 3 NIL)
```

The head of the tail of the list:

```
HEAD (TAIL (CONS 1 (CONS 2 (CONS 3 NIL))))
```

is:

```
HEAD (CONS 2 (CONS 3 NIL))
```

which is:

```
2
```

The tail of the tail of the list:

```
TAIL (TAIL (CONS 1 (CONS 2 (CONS 3 NIL))))
```

is:

```
TAIL (CONS 2 (CONS 3 NIL))
```

which is:

```
CONS 3 NIL
```

The head of the tail of the tail of this list:

```
HEAD (TAIL (TAIL (CONS 1 (CONS 2 (CONS 3 NIL)))))
```

is:

```
HEAD (TAIL (CONS 2 (CONS 3 NIL)))
```

which is:

```
HEAD (CONS 3 NIL)
```

giving:

```
3
```

The tail of the tail of the tail of the list:

```
TAIL (TAIL (TAIL (CONS 1 (CONS 2 (CONS 3 NIL)))))
```

is:

```
TAIL (TAIL (CONS 2 (CONS 3 NIL)))
```

which is:

```
TAIL (CONS 3 NIL)
```

giving:

```
NIL
```

6.3. List representation

First we define the list type:

```
def list_type = three
```

and associated tests:

```
def islist = istype list_type

def ISLIST L = MAKE_BOOL (islist L)
```

and error object:

```
def LIST_ERROR = MAKE_ERROR list_type
```

A list value will consist of a pair made from the list head and tail so the general form of a list object with head `<head>` and tail `<tail>` will be:

```
λs.(s listtype
         λs.(s <head> <tail>))
```

Thus, we define:

```
def MAKE_LIST = make_obj listtype

def CONS H T =
 if islist T
 then MAKE_LIST λs.(s H T)
 else LIST_ERROR
```

For example:

```
CONS 1 NIL => ... =>

MAKE_LIST λs.(s 1 NIL) => ... =>

λs.(s listtype
         λs.(s 1 NIL))
```

and:

```
CONS 2 (CONS 1 NIL) => ... =>

MAKE_LIST λs.(s 2 (CONS 1 NIL)) => ... =>

λs.(s listtype
         λs.(s 2 (CONS 1 NIL))) => ... =>

λs.(s listtype
         λs.(s 2 λs.(s listtype
                         λs.(s 1 NIL))))
```

The empty list will have both head and tail set to `LIST_ERROR`:

```
def NIL = MAKE_LIST λs.(s LIST_ERROR LIST_ERROR)
```

so `NIL` is:

```
λs.(s listtype
         λs.(s LIST_ERROR LIST_ERROR))
```

Now we can use the pair selectors to extract the head and tail:

```
def HEAD L =
 if islist L
 then (value L) select_first
 else LIST_ERROR

def TAIL L =
 if islist L
 then (value L) select_second
 else LIST_ERROR
```

For example:

```
HEAD (CONS 1 (CONS 2 NIL)) ==

(value (CONS 1 (CONS 2 NIL))) select_first ==

(value λs.(s listtype
               λs.(s 1 (CONS 2 NIL)))) select_first => ... =>
```

```
λs.(s 1 (CONS 2 NIL)) select_first =>

select_first 1 (CONS 2 NIL) => ... =>

1
```

and:

```
TAIL (CONS 1 (CONS 2 NIL)) ==

(value (CONS 1 (CONS 2 NIL))) select_second ==

(value λs.(s listtype
                λs.(s 1 (CONS 2 NIL)))) select_second => ... =>

λs.(s 1 (CONS 2 NIL)) select_second =>

select_second 1 (CONS 2 NIL) => ... =>

(CONS 2 NIL)
```

and:

```
HEAD (TAIL (CONS 1 (CONS 2 NIL))) -> ... ->

HEAD (CONS 2 NIL) ==

(value (CONS 2 NIL)) select_first ==

(value λs.(listtype λs.(s 2 NIL))) select_first => ... =>

λs.(s 2 NIL) select_first => ... =>

select_first 2 NIL => ... =>

2
```

Note that both `HEAD` and `TAIL` will return `LIST_ERROR` from the empty list. Thus:

```
HEAD NIL ==

(value NIL) select_first => ...=>

λs.(s LIST_ERROR LIST_ERROR) select_first =>

select_first LIST_ERROR LIST_ERROR => ... =>

LIST_ERROR
```

and:

```
TAIL NIL ==

(value NIL) select_second => ...=>

λs.(s LIST_ERROR LIST_ERROR) select_second =>

select_second LIST_ERROR LIST_ERROR => ... =>
```

```
LIST_ERROR
```

The test for an empty list checks for a list with an error object in the head:

```
def isnil L =
 if islist L
 then iserror (HEAD L)
 else false

def ISNIL L =
 if islist L
 then MAKE_BOOL (iserror (HEAD L))
 else LIST_ERROR
```

We will now define a variety of elementary operations on lists.

6.4. Linear length of a list

A list is a sequence of an arbitrary number of objects. To find out how many objects are in a linear list: if the list is empty then there are 0 objects:

```
LENGTH NIL = 0
```

and otherwise there are 1 more than the number in the tail:

```
LENGTH (CONS H T) = SUCC (LENGTH T)
```

For example:

```
LENGTH (CONS 1 (CONS 2 (CONS 3 NIL))) -> ... ->

SUCC (LENGTH (CONS 2 (CONS 3 NIL))) -> ... ->

SUCC (SUCC (LENGTH (CONS 3 NIL))) -> ... ->

SUCC (SUCC (SUCC (LENGTH NIL))) -> ... ->

SUCC (SUCC (SUCC 0))) ==

3
```

In our notation, this is:

```
rec LENGTH L =
 IF ISNIL L
 THEN 0
 ELSE SUCC (LENGTH (TAIL L))
```

For example, consider:

```
LENGTH (CONS 1 (CONS 2 NIL)) -> ... ->

SUCC (LENGTH (TAIL (CONS 1 (CONS 2 NIL)))) -> ... ->

SUCC (LENGTH (CONS 2 NIL)) -> ... ->

SUCC (SUCC (LENGTH (TAIL (CONS 2 NIL)))) -> ... ->
```

```
SUCC (SUCC (LENGTH NIL)) -> ... ->

SUCC (SUCC 0) ==

2
```

Note that the selection of the tail of the list is implicit in the case notation but must be made explicit in our current notation.

6.5. Appending lists

It is often useful to build one large linear list from several smaller lists. To append two lists together so that the second is a linear continuation of the first: if the first is empty then the result is the second:

```
APPEND NIL L = L
```

Otherwise the result is that of constructing a new list with the head of the first list as the head and the result of appending the tail of the first list to the second as the tail:

```
APPEND (CONS H T) L = CONS H (APPEND T L)
```

For example, to join:

```
CONS 1 (CONS 2 NIL))
```

to:

```
CONS 3 (CONS 4 NIL))
```

to get:

```
CONS 1 (CONS 2 (CONS 3 (CONS 4 NIL)))
```

we use:

```
APPEND (CONS 1 (CONS 2 NIL)) (CONS 3 (CONS 4 NIL)) -> ... ->

CONS 1 (APPEND (CONS 2 NIL) (CONS 3 (CONS 4 NIL))) -> ... ->

CONS 1 (CONS 2 (APPEND NIL (CONS 3 (CONS 4 NIL)))) -> ... ->

CONS 1 (CONS 2 (CONS 3 (CONS 4 NIL)))
```

In our notation this is:

```
rec APPEND L1 L2 =
 IF ISNIL L1
 THEN L2
 ELSE CONS (HEAD L1) (APPEND (TAIL L1) L2)
```

For example, consider:

```
APPEND (CONS 1 (CONS 2 NIL)) (CONS 3 NIL) -> ... ->

CONS (HEAD (CONS 1 (CONS 2 NIL)))
     (APPEND (TAIL (CONS 1 (CONS 2 NIL))) (CONS 3 NIL)) -> ... ->
```

```
CONS 1 (APPEND (TAIL (CONS 1 (CONS 2 NIL))) (CONS 3 NIL)) -> ... ->

CONS 1 (APPEND (CONS 2 NIL) (CONS 3 NIL)) -> ... ->

CONS 1 (CONS (HEAD (CONS 2 NIL))
        (APPEND (TAIL (CONS 2 NIL)) (CONS 3 NIL))) -> ... ->

CONS 1 (CONS 2
        (APPEND (TAIL (CONS 2 NIL)) (CONS 3 NIL))) -> ... ->

CONS 1 (CONS 2 (APPEND NIL (CONS 3 NIL))) -> ... ->

CONS 1 (CONS 2 (CONS 3 NIL))
```

Note again that the implicit list head and tail selection in the case notation is replaced by explicit selection in our current notation.

6.6. List notation

These examples illustrate how inconvenient it is to represent lists in a functional form: there is an excess of brackets and `CONS`s! In LISP, a linear list may be represented as a sequence of objects within brackets with an implict `NIL` at the end but this overloads the function application notation. LISP's simple, uniform notation for data and functions is an undoubted strength for a particular sort of programming where programs manipulate the text of other programs but it is somewhat opaque.

We will introduce two new notations. First of all, we will follow ML and use the binary infix operator `::` in place of `CONS.` Thus:

```
<expression1>::<expression2> == CONS <expression1> <expression2>
```

LISP and Prolog use `.` as an infix concatenation operator.

For example:

```
CONS 1 (CONS 2 (CONS 3 NIL)) ==

CONS 1 (CONS 2 (3::NIL)) ==

CONS 1 (2::(3::NIL)) ==

1::(2::(3::NIL))
```

Secondly, we will adopt the notation used by ML and Prolog and represent a linear list with an implicit `NIL` at the end as a sequence of objects within square brackets `[` and `],` separated by commas:

```
X :: NIL == [X]
X :: [Y] == [X,Y]
```

For example:

```
CONS 1 NIL == 1::NIL == [1]

CONS 1 (CONS 2 NIL) == 1::(2::NIL) == 1::[2] == [1,2]

CONS 1 (CONS 2 (CONS 3 NIL)) == 1::(2::(3::NIL))) ==
1::(2::[3]) == 1::[2,3] == [1,2,3]
```

This leads naturally to:

```
NIL = []
```

which is a list of no objects with an implicit `NIL` at the end.

For example, a list of pairs:

```
CONS (CONS 5 (CONS 12 NIL))
     (CONS (CONS 10 (CONS 15 NIL))
           (CONS (CONS 15 (CONS 23 NIL))
                 (CONS (CONS 20 (CONS 45 NIL))
                       NIL)))
```

becomes the compact:

```
[[5,12],[10,15],[15,23],[20,45]]
```

As before, `HEAD` selects the head element:

```
HEAD (X::Y) = X
HEAD [X,L] = X
```

For example:

```
HEAD [[5,12],[10,15],[15,23],[20,45]] => ... =>

[5,12]
```

and `TAIL` selects the tail:

```
TAIL (X::Y) = Y
TAIL [X,L] = [L]
```

for example:

```
TAIL [[5,12],[10,15],[15,23],[20,45]] => ... =>

[[10,15],[15,23],[20,45]]
```

and:

```
TAIL [5] => ... =>

[]
```

Note that constructing a list with `CONS` is not the same as using `[` and `].` For lists constructed with `[` and `]` there is an assumed empty list at the end. Thus:

```
[<first item>,<second item>] ==

CONS <first item> (CONS <second item> NIL)
```

We may simplify long lists made from `::` by dropping intervening brackets. Thus:

```
<expression1>::(<expression2>::<expression3>) ==

<expression1>::<expression2>::<expression3>
```

For example:

```
1::(2::(3::(4::[]))) == 1::2::3::4::[] == [1,2,3,4]
```

6.7. Lists and evaluation

It is important to note that we have adopted tacitly a weaker form of β reduction with lists because we are not evaluating fully the expressions corresponding to list representations. A list of the form:

```
<expression1>::<expression2>
```

is shorthand for:

```
CONS <expression1> <expression2>
```

which is a function application and should, strictly speaking, be evaluated.

Here, we have tended to use a modified form of applicative order where we evaluate the arguments `<expression1>` and `<expression2>` to get values `<value1>` and `<value2>` but do not then evaluate the resulting function application:

```
CONS <value1> <value2>
```

any further.

Similarly, a list of the form:

```
[<expression1>,<expression2>]
```

has been evaluated to:

```
[<value1>,<value2>]
```

but no further even though it is equivalent to:

```
CONS <value1> (CONS <value2> NIL)
```

This should be born in brain until we discuss evaluation in more detail in chapter 9.

6.8. Deletion from a list

To add a new value to an unordered list we `CONS` it on the front. To delete a value, we must then search the list until we find it. Thus, if the list is empty then the value is not in it so return the empty list:

```
DELETE X [] = []
```

It is common in list processing to return the empty list if list access fails.

Otherwise, if the first value in the list is the required value then return the rest of the list:

```
DELETE X (H::T) = T if <equal> X H
```

Otherwise, join the first value onto the result of deleting the required value from the rest of the list:

```
DELETE X (H::T) =  H::(DELETE X T) if NOT (<equal> X H)
```

Note that the comparison `<equal>` depends on the type of the list elements.

Suppose we are deleting from a linear list of numbers. For example, suppose we want to delete `3`

from the list:

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

we use:

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

CONS 1 (DELETE 3 [2,3,4]) -> ... ->

CONS 1 (CONS 2 (DELETE 3 [3,4])) -> ... ->

CONS 1 (CONS 2 [4]) -> ... ->

CONS 1 [2,4] ==

[1,2,4]
```

In our notation, the function becomes:

```
rec DELETE V L =
 IF ISNIL L
 THEN NIL
 ELSE
  IF EQUAL V (HEAD L)
  THEN TAIL L
  ELSE (HEAD L)::(DELETE V (TAIL L))
```

For example, suppose we want to delete `10` from:

```
[5,10,15,20]
```

we use:

```
DELETE 10 [5,10,15,20]

(HEAD [5,10,15,20])::(DELETE 10 (TAIL [5,10,15,20])) -> ... ->

5::(DELETE 10 ([10,15,20]))

5::(TAIL [10,15,20]) -> ... ->

5::[15,20] => ... =>

[5,15,20]
```

Note again that implicit list head and tail selection in the case notation ie replaced by explicit selection in our current notation.

6.9. List comparison

Two lists are the same if they are both empty:

```
LIST_EQUAL [] [] = TRUE
LIST_EQUAL [] (H::T) = FALSE
LIST_EQUAL (H::T) [] = FALSE
```

Otherwise they are the same if the heads are equal and the tails are the same:

```
LIST_EQUAL (H1::T1) (H2::T2) = LIST_EQUAL T1 T2
                               if <equal> H1 H2

LIST_EQUAL (H1::T1) (H2::T2) = FALSE
                               if NOT (<equal> H1 H2)
```

Notice again that the comparison operation `<equal>` depends on the type of the list elements.

Here, we will compare lists of numbers, for example:

```
LIST_EQUAL [1,2,3] [1,2,3] -> ... ->

LIST_EQUAL [2,3] [2,3] -> ... ->

LIST_EQUAL [3] [3] -> ... ->

LIST_EQUAL [] [] -> ... ->

TRUE
```

In our notation this algortithm is:

```
rec LIST_EQUAL L1 L2 =
 IF AND (ISNIL L1) (ISNIL L2)
 THEN TRUE
 ELSE
  IF OR (ISNIL L1) (ISNIL L2)
  THEN FALSE
  ELSE
   IF EQUAL (HEAD L1) (HEAD L2)
   THEN LIST_EQUAL (TAIL L1) (TAIL L2)
   ELSE FALSE
```

For example consider:

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

{ EQUAL (HEAD [1,2,3]) (HEAD [1,2,4])) -> ... ->

  EQUAL 1 1 -> ... ->

  TRUE}

LIST_EQUAL (TAIL [1,2,3]) (TAIL [1,2,4]) -> ... ->

LIST_EQUAL [2,3] [2,4] -> ... ->

{ EQUAL (HEAD [2,3]) (HEAD [2,4])) -> ... ->

  EQUAL 2 2 -> ... ->

  TRUE}
```

```
LIST_EQUAL (TAIL [2,3]) (TAIL [2,4]) -> ... ->

LIST_EQUAL [3] [4] -> ... ->

{ EQUAL (HEAD [3]) (HEAD [4])) -> ... ->

  EQUAL 3 4 -> ... ->

  FALSE}

FALSE
```

6.10. Strings

Strings are the basis of text processing in many languages. Some provide strings as a distinct type, for example BASIC and ML. Others base strings on arrays of characters, for example Pascal and C. Here we will introduce strings as linear lists of characters.

We can define a test for stringness:

```
ISSTRING [] = TRUE
ISSTRING (H::T) = (ISCHAR H) AND (ISSTRING T)
```

For example:

```
ISSTRING ['a','p','e'] -> ... ->

(ISCHAR 'a') AND
 (ISSTRING ['p','e']) -> ... ->

(ISCHAR 'a') AND
 ((ISCHAR 'p') AND
   (ISSTRING ['e'])) -> ... ->

(ISCHAR 'a') AND
 ((ISCHAR 'p') AND
  ((ISCHAR 'e') AND
   (ISSTRING []))) -> ... ->

(ISCHAR 'a') AND
 ((ISCHAR 'p') AND
  ((ISCHAR 'e') AND
   TRUE) -> ... ->

TRUE
```

In our notation, this function is:

```
rec ISSTRING S =
 IF ISNIL S
 THEN TRUE
 ELSE AND (ISCHAR (HEAD S)) (ISSTRING (TAIL S))
```

We will represent a string as a sequence of characters within

```
 "Here is a string!"
```

In general, the string:

```
"<character> <characters>"
```

is equivalent to the list:

```
'<character>'::"<characters>"
```

For example:

```
"cat" ==

'c'::"at" ==

'c'::'a'::"t" ==

'c'::'a'::'t'::""
```

If we represent the empty string:

```
""
```

as:

```
[]
```

then the string notation mirrors the compact list notation. For example:

```
"cat" ==

'c'::'a'::'t'::"" ==

'c'::'a'::'t'::[] ==

['c','a','t']
```

6.11. String comparison

String comparison is a type specific version of list comparison. Two strings are the same if both are empty:

```
STRING_EQUAL "" "" = TRUE
STRING_EQUAL "" (C::S) = FALSE
STRING_EQUAL (C::S) "" = FALSE
```

or if the characters in the heads are the same and the strings in the tails are the same:

```
STRING_EQUAL (C1::S1) (C2::S2) = STRING_EQUAL S1 S2
                                 if CHAR_EQUAL C1 C2

STRING_EQUAL (C1::S1) (C2::S2) = FALSE
                                 if NOT (CHAR_EQUAL C1 C2)
```

For example:

```
STRING_EQUAL "dog" "dog" ==

STRING_EQUAL ('d'::"og") ('d'::"og") -> ... ->
```

```
STRING_EQUAL "og" "og" ==

STRING_EQUAL ('o'::"g") ('o'::"g") -> ... ->

STRING_EQUAL "g" "g" ==

STRING_EQUAL ('g'::"")  ('g'::"")  -> ... ->

STRING_EQUAL "" "" -> ... ->

TRUE
In our notation:

rec STRING_EQUAL S1 S2 =
 IF (ISNIL S1) AND (ISNIL S2)
 THEN TRUE
 ELSE
  IF (ISNIL S1) OR (ISNIL S2)
  THEN FALSE
  ELSE
   IF CHAR_EQUAL (HEAD S1) (HEAD S2)
   THEN STRING_EQUAL (TAIL S1) (TAIL S2)
   ELSE FALSE
```

Similarly, one string comes before another if its empty and the other is not:

```
STRING_LESS "" (C::S) = TRUE
STRING_LESS (C::S) "" = FALSE
```

or if the character in its head comes before the character in the others head:

```
STRING_LESS (C1::S1) (C2::S2) = TRUE
                                if CHAR_LESS C1 C2
```

or the head characters are the same and the first string's tail comes before the second string's tail:

```
STRING_LESS (C1::S1) (C2::S2) = (CHAR_EQUAL C1 C2) AND
                                (STRING_LESS S1 S2)
                                if NOT (CHAR_LESS C1 C2)
```

For example:

```
STRING_LESS "porridge" "potato" ==

STRING_LESS ('p'::"orridge") ('p'::"otato") -> ... ->

(CHAR_EQUAL 'p' 'p') AND
 (STRING_LESS "orridge" "otato") ==

(CHAR_EQUAL 'p' 'p') AND
 (STRING_LESS ('o'::"rridge") ('o'::"tato")) -> ... ->

(CHAR EQUAL 'p' 'p') AND
 ((CHAR_EQUAL 'o' 'o') AND
  (STRING_LESS "rridge" "tato")) ==

(CHAR EQUAL 'p' 'p') AND
 ((CHAR_EQUAL 'o' 'o') AND
```

```
  (STRING_LESS ('r'::"ridge") ('t'::"ato")) -> ... ->

(CHAR EQUAL 'p' 'p') AND
 ((CHAR_EQUAL 'o' 'o') AND
  TRUE)) -> ... ->

TRUE
```

In our notation this is:

```
rec STRING_LESS S1 S2 =
 IF ISNIL S1
 THEN  NOT (ISNIL S2)
 ELSE
  IF ISNIL L2
  THEN FALSE
  ELSE
   IF CHAR_LESS (HEAD S1) (HEAD S2)
   THEN TRUE
   ELSE (CHAR_EQUAL (HEAD S1) (HEAD S2)) AND
        (STRING_LESS1 (TAIL S1) (TAIL S2))
```

6.12. Numeric string to number conversion

Given a string of digits, we might wish to find the equivalent number. Note first of all that the number equivalent to a digit is found by taking `'0'`s value away from its value. For example:

```
value '0' => ... =>

forty_eight
```

so:

```
sub (value '0') (value '0') -> ... ->

sub forty_eight forty_eight => ... =>

zero
```

Similarly:

```
value '1' => ... =>

forty_nine
```

so:

```
sub (value '1') (value '0') -> ... ->

sub forty_nine forty_eight => ... =>

one
```

and:

```
value ’2’ => ... =>

fifty
```

so:

```
sub (value ’2’) (value ’0’) -> ... ->

sub fifty forty_eight => ... =>

two
```

and so on. Thus, we can define:

```
def digit_value d = sub (value d) (value ’0’)
```

Note also that the value of a single digit string is the value of the digit, for example:

value of "9"

gives value of ’9’

gives 9

The value of a two digit string is ten times the value of the first digit added to the value of the second digit, for example:

value of "98"

gives 10*value of "9"+value of ’8’

gives 90+8 = 98

The value of a three digit string is ten times the value of the first two digits added to the value of the third digit, which is ten times ten times the value of the first digit added to the second digit, added to the value of the third digit, for example:

value of "987"

gives 10*value of "98"+value of ’7’

gives 10*(10*value of "9"+value of ’8’)+value of ’7’

gives 10*(10*9+8)+7 = 987

In general, the value of an `N` digit string is ten times the value of the first `N-1` digits added to the value of the `N`th digit. The value of an empty digit string is `0.`

We will implement this inside out so we can work from left to right through the string. We will keep track of the value of the first `N-1` digits in another bound variable `v.` Each time we will multiply `v` by `10` and add in the value of the `N`th digit to get the value of `v` for processing the `N+1` the digit. When the string is empty we return `v.` To start with, `v` is `0.` For example:

value of "987" with 0

gives value of "87" with 10*0+value of ’9’

gives value of "7" with 10*(10*0+value of ’9’)+value of ’8’

gives value of "" with 10*(10*(10*0+value of ’9’)+value of ’8’)+value of ’7’

gives 987

Thus:

```
STRING_VAL V "" = V
STRING_VAL V D::T = STRING_VAL 10*V+(digit_value D) T
```

For a whole string, V starts at

0 for example:

```
STRING_VAL 0 "321" ==

STRING_VAL 0 (’3’::"21") -> ... ->

STRING_VAL 10*0+(digit_value ’3’) "21" -> ... ->

STRING_VAL 3 "21" ==

STRING_VAL 3 (’2’::"1") -> ... ->

STRING_VAL 10*3+(digit_value ’2’) "1" -> ... ->

STRING_VAL 32 "1" ==

STRING_VAL 32 (’1’::"") -> ... ->

STRING_VAL 32*10+(digit_value ’1’) "" -> ... ->

STRING_VAL 321 "" -> ... ->

321
```

In our notation, using untyped arithmetic, the function is:

```
rec string_val v L =
 IF ISNIL L
 THEN v
 ELSE string_val (add (mult v ten) (digit_value (HEAD L)))
                 (TAIL L)

def STRING_VAL S = MAKE_NUMB (string_val zero S)
```

For example:

```
STRING_VAL "987" ==

MAKE_NUMB (string_val zero "987") -> ... ->

MAKE_NUMB (string_val (add
                       (mult zero ten)
                       (digit_value ’9’))
                      "87") -> ... ->

MAKE_NUMB (string_val (add
                       (mult
```

```
                        (add
                         (mult zero ten)
                         (digit_value '9'))
                        ten)
                       (digit_value '8'))
                      "7") -> ... ->

MAKE_NUMB (string_val (add
                       (mult
                        (add
                         (mult
                          (add
                           (mult zero ten)
                           (digit_value '9'))
                          ten)
                         (digit_value '8'))
                        ten)
                       (digit_value '7'))
                      "") -> ... ->

MAKE_NUMB (add
           (mult
            (add
             (mult
              (add
               (mult zero ten)
               (digit_value '9'))
              ten)
             (digit_value '8'))
            ten)
           (digit_value '7'))
          ) -> ... ->

MAKE_NUMB (add
           (mult
            (add
             (mult nine ten)
             (digit_value '8'))
            ten)
           (digit_value '7'))
          ) -> ... ->

MAKE_NUMB (add
           (mult ninety_eight ten)
           (digit_value '7'))
          ) -> ... ->

MAKE_NUMB nine_hundred_and_eighty_seven => ... =>

987
```

6.13. Structure matching with lists

We have been defining list functions with a base case for an empty list argument and a recursion case for a non-empty list argument, but we have translated them into explicit list selection. We will now extend our structure matching notation to lists and allow cases for the empty list `NIL` and for bound variable lists built with `::` in place of bound variables.

In general, for:

```
rec <name> <bound variable> =
 IF ISNIL <bound variable>
 THEN <expression1>
 ELSE <expression2 using (HEAD <bound variable>)
             and (TAIL <bound variable>)>
```

we will write:

```
rec <name> [] = <expression1>
 or <name> (<head>::<tail>) = <expression2 using <head>
                           and <tail>>
```

where <head> and <tail> are bound variables.

Consider the definition of the linear length of a list:

```
LENGTH [] = 0
LENGTH (H::T) = SUCC (LENGTH T)
```

which we wrote as:

```
rec LENGTH L =
 IF ISNIL L
 THEN 0
 ELSE SUCC (LENGTH (TAIL L))
```

We will now write:

```
rec LENGTH [] = 0
 or LENGTH (H::T) = SUCC (LENGTH T)
```

For example, suppose we have a list made up of arbitrarily nested lists which we want to flatten into one long linear list, for example:

```
FLAT [[1,2,3],[[4,5],[6,7,[8.9]]]] => ... =>

[1,2,3,4,5,6,7,8,9]
```

The empty list is already flat:

```
FLAT [] = []
```

If the list is not empty then if the head is not a list then join it to the flattened tail:

```
FLAT (H::T) = H::(FLAT T) if NOT (ISLIST H)
```

Otherwise, append the flattened head to the flattened tail:

```
FLAT (H::T) = APPEND (FLAT H) (FLAT T) if ISLIST H
```

In our old notation we would write:

```
rec FLAT L =
 IF ISNIL L
 THEN []
 ELSE
```

```
  IF NOT (ISLIST (HEAD L))
  THEN (HEAD L)::(FLAT (TAIL L))
  ELSE APPEND (FLAT (HEAD L)) (FLAT (TAIL L))
```

We will now write:

```
rec FLAT [] = []
 or FLAT (H::T) =
  IF NOT (ISLIST H)
  THEN H::(FLAT T)
  ELSE APPEND (FLAT H) (FLAT T)
```

Note that we may still need explicit conditional expressions in case definitions. For example, in `FLAT` above a conditional is need to distinguish the cases where the argument list has or does not have a list in its head. Structure matching can only distinguish between structural differences; it cannot distinguish between arbitrary values within structures.

Note that in LISP ther are no case definitions or structure matching so explicit list selection is necessary.

6.14. Ordered linear lists, insertion and sorting

For many applications, it is useful to hold data in some order to ease data access and presentation. Here we will look at ordered lists of data.

Firts of all, an ordered list is empty:

```
ORDERED [] = TRUE
```

or has a single element:

```
ORDERED [C] = TRUE
```

or has a head which comes before the head of the tail and an ordered tail:

```
ORDERED (C1::C2::L) = (<less> C1 C2) AND (ORDERED (CONS C2 L))
```

For example:

```
[1,2,3]
```

is ordered because `1` comes before `2` and `[2,3]` is ordered because `2` comes before `3` and `[3]` is ordered because it has a single element.

Thus, to insert an item into an ordered list: if the list is empty then the new list has the item as the sole element:

```
INSERT X [] = [X]
```

or if the item comes before the head of the list then the new list has the item as head and the old list as tail:

```
INSERT X (H::T) = X::H::T
                  if <less> X H
```

otherwise, the new list has the head of the old list as head and the item inserted into the tail of the old list as tail:

```
INSERT X (H::T) = H::(INSERT X T)
                  if NOT <less> X H
```

Note that this description tacitly assumes that all the items in the list are of the same type with a defined order relation <less>. For example, for lists of strings <less> will be STRING_LESS:

```
rec INSERT S [] = [S]
 or INSERT S (H::T) =
  IF STRING_LESS S H
  THEN S::H::T
  ELSE H::(INSERT H T)
```

For example:

```
INSERT "cherry" ["apple","banana","date"] => ... =>

"apple"::(INSERT "cherry" ["banana","date"]) -> ... ->

"apple"::"banana"::(INSERT "cherry" ["date"]) -> ... ->

"apple"::"banana"::"cherry"::["date"] ==

["apple","banana","cherry","date"]
```

Insertion forms the basis of a simple sort. The empty list is sorted:

```
SORT [] = []
```

and to sort a non-empty list, insert the head into the sorted tail:

```
SORT (H::T) = INSERT H (SORT T)
```

This is a general definition but must be made type specific. Once again, we will consider a list of strings:

```
rec SORT [] = []
 or SORT (H::T) = INSERT H (SORT T)
```

For example:

```
SORT ["cat","bat","ass"] => ... =>

INSERT "cat" (SORT ["bat","ass"]) -> ... ->

INSERT "cat" (INSERT "bat" (SORT ["ass"])) -> ... ->

INSERT "cat" (INSERT "bat" (INSERT "ass" [])) -> ... ->

INSERT "cat" (INSERT "bat" ["ass"]) -> ... ->

INSERT "cat" ["ass","bat"] => ... =>

["ass","bat","cat"]
```

6.15. Indexed linear list access

Arrays are often used where a linear sequence of objects is to be manipulated in terms of the linear positions of the objects in the sequence. In the same way, it is often useful to access an element of a linear list by specifying its position relative to the start of the list. For example, in:

```
["Chris","Jean","Les","Pat","Phil"]
```

the first name is

```
"Chris"
```

and we use:

```
HEAD ["Chris","Jean","Les","Pat","Phil"]
```

to access it, the second name is

```
"Jean"
```

and we use:

```
HEAD (TAIL ["Chris","Jean","Les","Pat","Phil"])
```

to access it, the third name is

```
"Les"
```

and we use:

```
HEAD (TAIL (TAIL ["Chris","Jean","Les","Pat","Phil"]))
```

to access it and so on. In general, to access the `<number>+1th` name we take the `TAIL <number>` times and then take the `HEAD:`

```
IFIND (SUCC N) (H::T) = IFIND N T
IFIND 0 (H::T) = H
```

Note that the first element is number `0,` the second element is number `1` and so on because we are basing our definition on how often the tail is taken to make the element the head of the remaining list.

If the list does not have `<number>` elements we will just return the empty list:

```
IFIND N [] = []
```

To summarise:

```
rec IFIND N [] = []
 or IFIND 0 (H::T) = H
 or IFIND (SUCC N) (H::T) = IFIND N T
```

For example:

```
IFIND 3 ["Chris","Jean","Les","Pat","Phil"] => ... =>

IFIND 2  ["Jean","Les","Pat","Phil"] => ... =>

IFIND 1  ["Les","Pat","Phil"] => ... =>

IFIND 0  ["Pat","Phil"] => ... =>

HEAD ["Pat","Phil"] => ... =>

"Pat"
```

Similarly, to remove a specified element from a list: if the list is empty then return the empty list:

```
IDELETE N [] = []
```

If the specified element is at the head of the list then return the tail of the list:

```
IDELETE 0 (H::T) = T
```

Otherwise, join the head of the list to the result of removing the element from the tail of the list, remembering that its position in the tail is one less than its position in the whole list:

```
IDELETE (SUCC N) (H::T) = H::(IDELETE N T)
```

To summarise:

```
rec IDELETE N [] = []
 or IDELETE 0 (H::T) = H
 or IDELETE (SUCC N) (H::T) = H::(IDELETE N T)
```

For example:

```
IDELETE 2 ["Chris","Jean","Les","Pat","Phil"] => ... =>

"Chris"::(IDELETE 1 ["Jean","Les","Pat","Phil"]) -> ... ->

"Chris"::"Jean":: (IDELETE 0 ["Les","Pat","Phil"]) -> ... ->

"Chris"::"Jean"::["Pat","Phil"]) ==

["Chris","Jean","Pat","Phil"]
```

New elements may be added to a list in a specified position. If the list is empty then return the empty list:

```
IBEFORE N E [] = []
```

If the specified position is at the head of the list then make the new element the head of a list with the old list as tail:

```
IBEFORE 0 E L = E::L
```

Otherwise, add the head of the list to the result of placing the new element in the tail of the list, remembering that the specified position is now one less than its position in the whole list:

```
IBEFORE (SUCC N) E (H::T) = H::(IBEFORE N E T)
```

To summarise:

```
rec IBEFORE N E [] = []
 or IBEFORE 0 E L = E::L
 or IBEFORE (SUCC N) E (H::T) = H::(IBEFORE N E T)

For example:

IBEFORE 2 "Jo" ["Chris","Jean","Les","Pat","Phil"] => ... =>

"Chris"::(IBEFORE 1 "Jo" ["Jean","Les","Pat","Phil"]) -> ... ->

"Chris"::"Jean":: (IBEFORE 0 "Jo" ["Les","Pat","Phil"]) -> ... ->
```

```
"Chris"::"Jean"::"Jo"::["Les","Pat","Phil"] ==

["Chris","Jean","Jo","Les","Pat","Phil"]
```

Finally, to replace the object in a specified position in a list, the list is empty then return the empty list:

```
IREPLACE N E [] = []
```

If the specified position is at the head then make the replacement the head:

```
IREPLACE 0 E (H::T) = E::T
```

Otherwise, join the head of the list to the result of replacing the element in the tail, remembering that the position in the tail is now one less than the position in the whole list:

```
IREPLACE (SUCC N) E (H::T) = H::(IREPLACE N E T)
```

Note that we have not considered what happens if the list does not contain the requisite item.

To summarise:

```
rec IREPLACE N E [] = []
 or IREPLACE 0 E (H::T) = E::T
 or IREPLACE (SUCC N) E (H::T) = H::(IREPLACE N E T)
```

For example:

```
IREPLACE 2 "Jo" ["Chris","Jean","Les","Pat","Phil"] => ... =>

"Chris"::(IREPLACE 1 "Jo" ["Jean","Les","Pat","Phil"]) -> ... ->

"Chris"::"Jean"::(IREPLACE 0 "Jo" ["Les","Pat","Phil"]) -> ... ->

"Chris"::"Jean"::"Jo"::["Pat","Phil"] ==

["Chris","Jean","Jo","Pat","Phil"]
```

Alternatively, we could use `DELETE` to drop the old element and `IBEFORE` to place the new element, so:

```
IREPLACE N E L = IBEFORE N E (IDELETE N L)
```

This is much simpler but involves scanning the list twice.

6.16. Mapping functions

Many functions have similar structures. We can take advantage of this to simplify function construction by defining abstract general purpose functions for common structures and inserting particular functions into them to make them carry out particular processes. For example, we have defined a general purpose `make_object` function in chapter 5 which we have then used to construct specialised `MAKE_BOOL`, `MAKE_NUMB`, `MAKE_LIST` and `MAKE_CHAR` functions.

For lists, such generalised functions are known as **mapping functions** because they are used to **map** a function onto the components of lists. The use of list mapping functions originated with LISP.

For example, consider a function which doubles every value in a list of numbers:

```
rec DOUBLE [] = []
 or DOUBLE (H::T) = (2*H)::(DOUBLE T)
```

so:

```
DOUBLE [1,2,3] => ... =>

(2*1)::(DOUBLE [2,3]) -> ... ->

2::(2*2)::(DOUBLE [3]) -> ... ->

2::4::(2*3)::(DOUBLE []) -> ... ->

2::4::6::[] ==

[2,4,6]
```

Now consider the function which turns all the words in a list into plurals:

```
rec PLURAL [] = []
 or PLURAL (H::T) = (APPEND H "s")::(PLURAL T)
```

so:

```
PLURAL ["cat","dog","pig"] => ... =>

(APPEND "cat" "s")::(PLURAL ["dog","pig"]) -> ... ->

"cats"::(APPEND "dog" "s")::(PLURAL ["pig"]) -> ... ->

"cats"::"dogs"::(APPEND "pig" "s")::(PLURAL []) -> ... ->

"cats"::"dogs"::"pigs"::[] ==

["cats","dogs","pigs"]
```

The functions DOUBLE and PLURAL both apply a function repeatedly to the consecutive heads of their list arguments. In LISP this is known as a **CAR mapping** because the function is mapped onto the CARs of the list. We can abstract a common structure from DOUBLE and PLURAL as:

```
rec MAPCAR FUNC [] = []
 or MAPCAR FUNC (H::T) = (FUNC H)::(MAPCAR FUNC T)
```

Thus, we can define DOUBLE as:

```
def DOUBLE = MAPCAR λX.(2*X)
```

so DOUBLEs definition expands as:

```
rec DOUBLE [] = []
 or DOUBLE (H::T) =(λX.(2*X) H)::(MAPCAR λX.(2*X) T)
```

Simplifying, we get:

```
def DOUBLE [] = []
 or DOUBLE (H::T) = (2*H)::(MAPCAR λX.(2*X) T)
```

which is equivalent to the original:

```
rec DOUBLE [] = []
 or DOUBLE (H::T) = (2*H)::(DOUBLE T)
```

because:

```
DOUBLE == MAPCAR λX.(2*X)
```

For example:

```
DOUBLE [1,2,3] => ... =>

(λX.(2*X) 1):)::(MAPCAR λX.(2*X) [2,3]) -> ... ->

2::(λX.(2*X) 2)::(MAPCAR λX.(2*X) [3]) -> ... ->

2::4::(λX.(2*X) 3)::(MAPCAR λX.(2*X) []) -> ... ->

2::4::6::[] ==

[2,4,6]
```

Similarly, we can redefine PLURAL as:

```
def PLURAL = MAPCAR λW.(APPEND W "s")
```

so expanding the definition gives:

```
rec PLURAL [] = []
 or PLURAL (H::T) = (λW.(APPEND W "s") H)::
                     (MAPCAR λW.(APPEND W "s") T)
```

so, simplifying:

```
def PLURAL [] = []
 or PLURAL (H::T) = (APPEND H "s")::
                     (MAPCAR λW.(APPEND W "s") T)
```

which is equivalent to:

```
rec PLURAL [] = []
 or PLURAL (H::T) = (APPEND H "s")::(PLURAL T)
```

because:

```
PLURAL == MAPCAR λW.(APPEND W "s")
```

For example:

```
PLURAL ["cat","dog","pig"] => ... =>

(λW.(APPEND W "s") "cat")::
 (MAPCAR λW.(APPEND W "s") ["dog","pig"]) -> ... ->

"cats"::(λW.(APPEND W "s") "dog")::
 (MAPCAR λW.(APPEND W "s") ["pig"]) -> ... ->
```

```
"cats"::"dogs"::(λW.(APPEND W "s") "pig")::
 (MAPCAR λW.(APPEND W "s") []) -> ... ->

"cats"::"dogs"::"pigs"::[] ==

["cats","dogs","pigs"]
```

Consider the function which compares two equal length linear lists of strings component by component and constructs a boolean list showing where they are the same and where they differ:

```
rec COMP [] [] = []
 or COMP (H1::T1) (H2::T2) = (STRING_EQUAL H1 H2)::(COMP T1 T2)
```

so:

```
COMP ["hey","diddle","diddle"] ["hey","daddle","diddle"] => ... =>

(STRING_EQUAL "hey" "hey")::
 (COMP ["diddle","diddle"] ["daddle","diddle"]) -> ... ->

TRUE::(STRING_EQUAL "diddle" "daddle")::
 (COMP ["diddle"] ["diddle"]) -> ... ->

TRUE::FALSE::(STRING_EQUAL "diddle" "diddle")::
 (COMP [] []) -> ... ->

TRUE::FALSE::TRUE::[] ==

[TRUE,FALSE,TRUE]
```

Now consider the function that adds together corresponding components of two equal length linear numeric lists:

```
rec SUM2 [] [] = []
 or SUM2 (H1::T1) (H2::T2) = (H1+H2)::(SUM2 T1 T2)
```

so:

```
SUM2 [1,2,3] [4,5,6] => ... =>

(1+4)::(SUM2 [2,3] [5,6]) -> ... ->

5::(2+5)::(SUM2 [3] [6]) -> ... ->

5::7::(3+6)::(SUM2 [] []) -> ... ->

5::7::9::[] ==

[5,7,9]
```

The functions `COMP` and `SUM2` both apply a function repeatedly to the consecutive heads of two list arguments to construct a new list. We can abstract a common structure from `COMP` and `SUM2` as:

```
rec MAPCARS FUNC [] [] = []
 or MAPCARS FUNC (H1::T1) (H2::T2) = (FUNC H1 H2)::(MAPCARS FUNC T1 T2)
```

Thus:

```
def COMP = MAPCARS λX.λY.(STRING_EQUAL X Y)
def SUM2 = MAPCARS λX.λY.(X+Y)
```

6.17. Summary

In this chapter we have:

- introduced the list type
- developed a representation for the list type and typed list operations
- developed elementary functions for manipulating linear lists
- introduced simplified list notations
- introduced strings as character lists with simplified notation
- introduced list case definitions and structure matching
- developed functions for constructing ordered linear lists
- developed functions for indexed linear list access
- developed mapping functions to generalise linear list operations

Some of these topics are summarised below.

List notation

```
<expression1>::<expression2> == CONS <expression1> <expression2>

[<expression1>,<expression2>] == <expression1>::[<expression2>]

[<expression>] == <expression>::NIL

[] == NIL

<expression1>::(<expression2>::<expression3>) ==

<expression1>::<expression2>::<expression3>
```

String notation

```
"<character> <characters>" == <character>::"<characters>"

"" = []
```

List case definition

```
rec <name> [] = <expression1>
 or <name> (<head>::<tail>) =
      <expression2 using ‘<head>’ and ‘<tail>’> ==

rec <name> <bound variable> =
 IF ISNIL <bound variable>
 THEN <expression1>
 ELSE <expression2 using ‘HEAD <bound variable>’ and
```

```
'TAIL <bound variable>'>
```

6.18. Exercises

1) Define a concatenation function for linear lists whose elements are atoms of the same type.

2) i) Write a fucntion which indicates whether or not a list starts with a sub-list. For example:

```
STARTS "The" "The cat sat on the mat." => ... =>
TRUE

STARTS "A" "The cat sat on the mat." => ... =>
FALSE
```

ii) Write a function which indicates whether or not a list contains a given sub-list. For example:

```
CONTAINS "the" "The cat sat on the mat." => ... =>
TRUE

CONTAINS "the" "All cats sit on all mats." => ... =>
FALSE
```

iii) Write a function which counts how often a sub-list appears in another list. For example:

```
COUNT "at" "The cat sat on the mat." => ... =>
3
```

iv) Write a function which removes a sub-list from the start of a list, assuming that you know that the sub-list starts the list. For example:

```
REMOVE "The " "The cat sat on the mat." => ... =>
"cat sat on the mat."
```

v) Write a function which deletes the first occurence of a sub-list in another list. For example:

```
DELETE "sat" "The cat sat on the mat." => ... =>
"The cat  on the mat."

DELETE "lay" "The cat sat on the mat." => ... =>
"The cat sat on the mat."
```

vi) Write a function which inserts a sub-list after the first occurence of another sub-list in a list. For example:

```
INSERT "sat" "cat " "The cat  on the mat." => ... =>
"The cat sat on the mat."

INSERT "sat" "fish " "The cat  on the mat." => ... =>
"The cat  on the mat."
```

vii) Write a function which replaces a sub-list with another sub-list in a list. For example:

```
REPLACE "sat" "lay" "The cat sat on the mat." => ... =>
"The cat lay on the mat."

REPLACE "sit" "lay" "The cat sat on the mat." => ... =>
"The cat sat on the mat."
```

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