Appendix A

Lists

A.1 Introduction

This book intensely uses recursive list manipulations in purely functional settings. List can be treated as a counterpart to array in imperative settings, which are the bricks to many algorithms and data structures.

For the readers who are not familiar with functional list manipulation, this appendix provides a quick reference. All operations listed in this appendix are not only described in equations, but also implemented in both functional programming languages as well as imperative languages as examples. We also provide a special type of implementation in C++ template meta programming similar to [3] for interesting in next appendix.

Besides the elementary list operations, this appendix also contains explanation of some high order function concepts such as mapping, folding etc.

A.2 List Definition

Like arrays in imperative settings, lists play a critical role in functional setting[1]. Lists are built-in support in some programming languages like Lisp families and ML families so it needn't explicitly define list in those environment.

List, or more precisely, singly linked-list is a data structure that can be described below.

- A *list* is either empty;
- Or contains an element and a *list*.

Note that this definition is recursive. Figure A.1 illustrates a list with N nodes. Each node contains two part, a key element and a sub list. The sub list contained in the last node is empty, which is denoted as 'NIL'.

[1]Some reader may argue that 'lambda calculus plays the most critical role'. Lambda calculus is somewhat as assembly languages to the computation world, which is worthy studying from the essence of computation model to the practical programs. However, we don't dive into the topic in this book. Users can refer to [4] for detail.

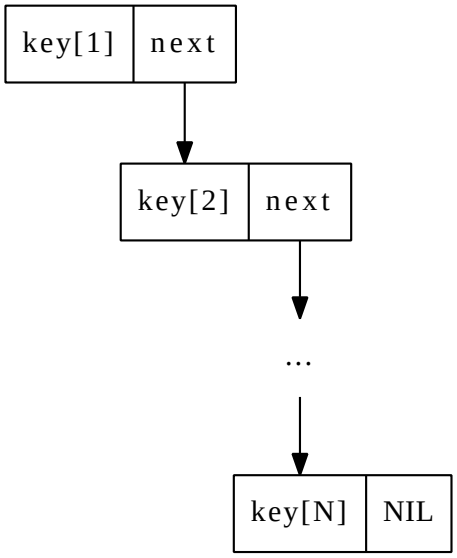


Figure A.1: A list contains N nodes

This data structure can be explicitly defined in programming languages support record (or compound type) concept. The following ISO C++ code defines list[2].

```
template<typename T>
struct List {
  T key;
  List* next;
};
```

A.2.1 Empty list

It is worth to mention about 'empty' list a bit more in detail. In environment supporting the nil concept, for example, C or java like programming languages, empty list can have two different representations. One is the trivial 'NIL' (or null, or 0, which varies from languages); the other is an non-NIL empty list as {}, the latter is typically allocated with memory but filled with nothing. In Lisp dialects, the empty is commonly written as `'()`. In ML families, it's written as `[]`. We use Φ to denote empty list in equations and use 'NIL' in pseudo code sometimes to describe algorithms in this book.

A.2.2 Access the element and the sub list

Given a list L, two functions can be defined to access the element stored in it and the sub list respectively. They are typically denoted as $first(L)$, and $rest(L)$ or $head(L)$ and $tail(L)$ for the same meaning. These two functions are named as `car` and `cdr` in Lisp for historic reason about the design of machine

[2]We only use template to parameterize the type of the element in this chapter. Except this point, all imperative source code are in ANSI C style to avoid language specific features.

registers [5]. In languages support Pattern matching (e.g. ML families, Prolog and Erlang etc.) These two functions are commonly realized by matching the `cons` which we'll introduced later. for example the following Haskell program:

```
head (x:xs) = x
tail (x:xs) = xs
```

If the list is defined in record syntax like what we did above, these two functions can be realized by accessing the record fields [3].

```
template<typename T>
T first(List<T> *xs) { return xs->key; }

template<typename T>
List<T>* rest(List<T>* xs) { return xs->next; }
```

In this book, L' is used to denote the $rest(L)$ sometimes, also we uses l_1 to represent $first(L)$ in the context that the list is literately given in form $L = \{l_1, l_2, ..., l_N\}$.

More interesting, as far as in an environment support recursion, we can define List. The following example define a list of integers in C++ compile time.

```
struct Empty;

template<int x, typename T> struct List {
  static const int first = x;
  typedef T rest;
};
```

This line constructs a list of $\{1, 2, 3, 4, 5\}$ in compile time.

```
typedef List<1, List<2, List<3, List<4 List<5, Empty> > > > > A;
```

A.3 Basic list manipulation

A.3.1 Construction

The last C++ template meta programming example actually shows literate construction of a list. A list can be constructed from an element with a sub list, where the sub list can be empty. We denote function $cons(x, L)$ as the constructor. This name is used in most Lisp dialects. In ML families, there are 'cons' operator defined as `::`, (in Haskell it's `:`).

We can define `cons` to create a record as we defined above in ISO C++, for example[4].

```
template<typename T>
List<T>* cons(T x, List<T>* xs) {
  List<T>* lst = new List<T>;
  lst->key = x;
  lst->next = xs;
  return lst;
}
```

[3] They can be also named as 'key' and 'next' or be defined as class methods.

[4] It is often defined as a constructor method for the class template, However, we define it as a standalone function for illustration purpose.

A.3.2 Empty testing and length calculating

It is trivial to test if a list is empty. If the environment contains nil concept, the testing should also handle nil case. Both Lisp dialects and ML families provide `null` testing functions. Empty testing can also be realized by pattern-matching with empty list if possible. The following Haskell program shows such example.

```
null [] = True
null _ = False
```

In this book we will either use $empty(L)$ or $L = \Phi$ where empty testing happens.

With empty testing defined, it's possible to calculate length for a list. In imperative settings, LENGTH is often implemented like the following.

function LENGTH(L)
  $n \leftarrow 0$
  while $L \neq NIL$ **do**
    $n \leftarrow n + 1$
    $L \leftarrow$ NEXT(L)

This ISO C++ code translates the algorithm to real program.

```
template<typename T>
int length(List<T>* xs) {
  int n = 0;
  for (; xs; ++n, xs = xs->next);
  return n
}
```

However, in purely funcitonal setting, we can't mutate a counter variable. the idea is that, if the list is empty, then its size is zero; otherwise, we can recursively calculate the length of the sub list, then add it by one to get the length of this list.

$$length(L) = \begin{cases} 0 & : \quad L = \Phi \\ 1 + length(L') & : \quad otherwise \end{cases} \tag{A.1}$$

Here $L' = rest(L)$ as mentioned above, it's $\{l_2, l_3, ..., l_N\}$ for list contains N elements. Note that both L and L' can be empty Φ. In this equation, we also use '=' to test if list L is empty. In order to know the length of a list, we need traverse all the elements from the head to the end, so that this algorithm is proportion to the number of elements stored in the list. It is a linear algorithm bound to $O(N)$ time.

Below are two programs in Haskell and in Scheme/Lisp realize this recursive algorithm.

```
length [] = 0
length (x:xs) = 1 + length xs
```

```
(define (length lst)
  (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```

How to testing if two lists are identical is left as exercise to the reader.

A.3.3 indexing

One big difference between array and list (singly-linked list accurately) is that array supports random access. Many programming languages support using `x[i]` to access the i-th element stored in array in constant $O(1)$ time. The index typically starts from 0, but it's not the all case. Some programming languages using 1 as the first index. In this appendix, we treat index starting from 0. However, we must traverse the list with i steps to reach the target element. The traversing is quite similar to the length calculation. Thus it's commonly expressed as below in imperative settings.

function GET-AT(L, i)
  while $i \neq 0$ **do**
    $L \leftarrow$ NEXT(L)
  return FIRST(L)

Note that this algorithm doesn't handle the error case such that the index isn't within the bound of the list. We assume that $0 \leq i < |L|$, where $|L| = length(L)$. The error handling is left as exercise to the reader. The following ISO C++ code is a line-by-line translation of this algorithm.

```
template<typename T>
T getAt(List<T>* lst, int n) {
  while(n--)
    lst = lst→next;
  return lst→key;
}
```

However, in purely functional settings, we turn to recursive traversing instead of while-loop.

$$getAt(L, i) = \begin{cases} First(L) & : \quad i = 0 \\ getAt(Rest(L), i - 1) & : \quad otherwise \end{cases} \tag{A.2}$$

In order to *get the i-th element*, the algorithm does the following:

- if i is 0, then we are done, the result is the first element in the list;
- Otherwise, the result is to *get the $(i-1)$-th element* from the sub-list.

This algorithm can be translated to the following Haskell code.

```
getAt i (x:xs) = if i == 0 then x else getAt i-1 xs
```

Note that we are using pattern matching to ensure the list isn't empty, which actually handles all out-of-bound cases with un-matched pattern error. Thus if $i > |L|$, we finally arrive at a edge case that the index is $i - |L|$, while the list is empty; On the other hand, if $i < 0$, minus it by one makes it even farther away from 0. We finally end at the same error that the index is some negative, while the list is empty;

The indexing algorithm takes time proportion to the value of index, which is bound to $O(N)$ linear time. This section only address the read semantics. How to mutate the element at a given position is explained in later section.

A.3.4 Access the last element

Although accessing the first element and the rest list L' is trivial, the opposite operations, that retrieving the last element and the initial sub list need linear time without using a tail pointer. If the list isn't empty, we need traverse it till the tail to get these two components. Below are their imperative descriptions.

```
function LAST(L)
    x ← NIL
    while L ≠ NIL do
        x ← FIRST(L)
        L ← REST(L)
    return x

function INIT(L)
    L' ← NIL
    while REST(L) ≠ NIL do
        L' ← APPEND(L', FIRST(L))
        L ← REST(L)
    return L'
```

The algorithm assumes that the input list isn't empty, so the error handling is skipped. Note that the INIT() algorithm uses the appending algorithm which will be defined later.

Below are the corresponding ISO C++ implementation. The optimized version by utilizing tail pointer is left as exercise.

```
template<typename T>
T last(List<T>* xs) {
  T x; /* Can be set to a special value to indicate empty list err. */
  for (; xs; xs = xs→next)
    x = xs→key;
  return x;
}

template<typename T>
List<T>* init(List<T>* xs) {
  List<T>* ys = NULL;
  for (; xs→next; xs = xs→next)
    ys = append(ys, xs→key);
  return ys;
}
```

While these two algorithms can be implemented in purely recursive manner as well. When we want to access *the last element*.

- If the list contains only one element (the rest sub-list is empty), the result is this very element;
- Otherwise, the result is *the last element* of the rest sub-list.

$$last(L) = \begin{cases} First(L) & : & Rest(L) = \Phi \\ last(Rest(L)) & : & otherwise \end{cases} \tag{A.3}$$

The similar approach can be used to *get a list contains all elements except for the last one.*

- The edge case: If the list contains only one element, then the result is an empty list;
- Otherwise, we can first *get a list contains all elements except for the last one* from the rest sub-list, then construct the final result from the first element and this intermediate result.

$$init(L) = \begin{cases} \Phi & : & L' = \Phi \\ cons(l_1, init(L')) & : & otherwise \end{cases} \tag{A.4}$$

Here we denote l_1 as the first element of L, and L' is the rest sub-list. This recursive algorithm needn't use appending, It actually construct the final result list from right to left. We'll introduce a high-level concept of such kind of computation later in this appendix.

Below are Haskell programs implement $last()$ and $init()$ algorithms by using pattern matching.

```
last [x] = x
last (_:xs) = last xs

init [x] = []
init (x:xs) = x : init xs
```

Where `[x]` matches the singleton list contains only one element, while `(_:xs)` matches any non-empty list, and the underscore (`_`) is used to indicate that we don't care about the element. For the detail of pattern matching, readers can refer to any Haskell tutorial materials, such as [8].

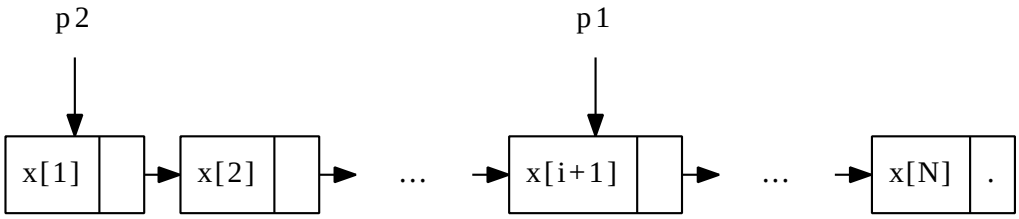
A.3.5 Reverse indexing

Reverse indexing is a general case for $last()$, finding the i-th element in a singly-linked list with the minimized memory spaces is interesting, and this problem is often used in technical interview in some companies. A naive implementation takes 2 rounds of traversing, the first round is to determine the length of the list N, then, calculate the left-hand index by $N - i - 1$. Finally a second round of traverse is used to access the element with the left-hand index. This idea can be give as the following equation.

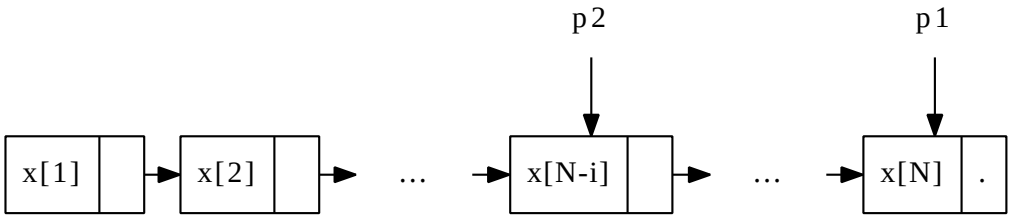
$$getAtR(L, i) = getAt(L, length(L) - i - 1)$$

There exists better imperative solution. For illustration purpose, we omit the error cases such as index is out-of-bound etc. The idea is to keep two pointers p_1, p_2, with the distance of i between them, that $rest^i(p_2) = p_1$, where $rest^i(p_1)$ means repleatedly apply $rest()$ function i times. It says that succeeds i steps from p_2 gets p_1. We can start p_2 from the head of the list and advance the two pointers in parallel till one of them (p_1) arrives at the end of the list. At that time point, pointer p_2 exactly arrived at the i-th element from right. Figure A.2 illustrates this idea.

It is straightforward to realize the imperative algorithm based on this 'double pointers' solution.



(a) p_2 starts from the head, which is behind p_1 in i steps.



(b) When p_1 reaches the end, p_2 points to the i-th element from right.

Figure A.2: Double pointers solution to reverse indexing.

function GET-AT-R(L, i)
  $p \leftarrow L$
  while $i \neq 0$ **do**
    $L \leftarrow$ REST(L)
    $i \leftarrow i - 1$
  while REST(L) $\neq$ NIL **do**
    $L \leftarrow$ REST(L)
    $p \leftarrow$ REST(p)
  return FIRST(p)

The following ISO C++ code implements the 'double pointers' right indexing algorithm.

```
template<typename T>
T getAtR(List<T>* xs, int i) {
  List<T>* p = xs;
  while(i--)
    xs = xs→next;
  for(; xs→next; xs = xs→next, p = p→next);
  return p→key;
}
```

The same idea can be realized recursively as well. If we want to access the i-th element of list L, we can examine the two lists L and $S = \{l_i, l_{i+1}, ..., l_N\}$ simultaneously, where S is a sub-list of L without the first i elements.

- The edge case: If S is a singleton list, then the i-th element from right is the first element in L;
- Otherwise, we drop the first element from L and S, and recursively exam-

ine L' and S'.

This algorithm description can be formalized as the following equations.

$$getAtR(L, i) = examine(L, drop(i, L)) \tag{A.5}$$

Where function $examine(L, S)$ is defined as below.

$$examine(L, S) = \begin{cases} first(L) & : & |S| = 1 \\ examine(rest(L), rest(S)) & : & otherwise \end{cases} \tag{A.6}$$

We'll explain the detail of *drop*() function in later section about list mutating operations. Here it can be implemented as repeatedly call *rest*() with specified times.

$$drop(n, L) = \begin{cases} L & : & n = 0 \\ drop(n - 1, rest(L)) & : & otherwise \end{cases}$$

Translating the equations to Haskell yields this example program.

```
atR :: [a] → Int → a
atR xs i = get xs (drop i xs) where
  get (x:_) [_] = x
  get (_:xs) (_:ys) = get xs ys
  drop n as@(_:as') = if n == 0 then as else drop (n-1) as'
```

Here we use dummy variable `_` as the placeholders for components we don't care.

A.3.6 Mutating

Strictly speaking, we can't mutate the list at all in purely functional settings. Unlike in imperative settings, mutate is actually realized by creating new list. Almost all functional environments support garbage collection, the original list may either be persisted for reusing, or released (dropped) at sometime (Chapter 2 in [6]).

Appending

Function *cons* can be viewed as building list by insertion element always on head. If we chains multiple *cons* operations, it can repeatedly construct a list from right to the left. Appending on the other hand, is an operation adding element to the tail. Compare to *cons* which is trivial constant time $O(1)$ operation, We must traverse the whole list to locate the appending position. It means that appending is bound to $O(N)$, where N is the length of the list. In order to speed up the appending, imperative implementation typically uses a field (variable) to record the tail position of a list, so that the traversing can be avoided. However, in purely functional settings we can't use such 'tail' pointer. The appending has to be realized in recursive manner.

$$append(L, x) = \begin{cases} \{x\} & : & L = \Phi \\ cons(first(L), append(rest(L), x)) & : & otherwise \end{cases} \tag{A.7}$$

That the algorithm handles two different appending cases:

- If the list is empty, the result is a singleton list contains x, which is the element to be appended. The singleton list notion $\{x\} = cons(x, \Phi)$, is a simplified form of *cons* the element with an empty list Φ;
- Otherwise, for the none-empty list, the result can be achieved by first appending the element x to the rest sub-list, then construct the first element of L with the recursive appending result.

For the none-trivial case, if we denote $L = \{l_1, l_2, ...\}$, and $L' = \{l_2, l_3, ...\}$ the equation can be written as.

$$append(L, x) = \begin{cases} \{x\} & : \quad L = \Phi \\ cons(l_1, append(L', x)) & : \quad otherwise \end{cases} \tag{A.8}$$

We'll use both forms in the rest of this appendix.

The following Scheme/Lisp program implements this algorithm.

```
(define (append lst x)
  (if (null? lst)
      (list x)
      (cons (car lst) (append (cdr lst) x))))
```

Even without the tail pointer, it's possible to traverse the list imperatively and append the element at the end.

```
function APPEND(L, x)
    if L = NIL then
        return CONS(x, NIL)
    H ← L
    while REST(L) ≠ NIL do
        L ← REST(L)
    REST(L) ← CONS(x, NIL)
    return H
```

The following ISO C++ programs implements this algorithm. How to utilize a tail field to speed up the appending is left as exercise to the reader for interesting.

```
template<typename T>
List<T>* append(List<T>* xs, T x) {
  List<T> *tail, *head;
  for (head = tail = xs; xs; xs = xs→next)
    tail = xs;
  if (!head)
    head = cons<T>(x, NULL);
  else
    tail→next = cons<T>(x, NULL);
  return head;
}
```

Mutate element at a given position

Although we have defined random access algorithm $getAt(L, i)$, we can't just mutate the element returned by this function in a sense of purely functional

settings. It is quite common to provide reference semantics in imperative programming languages and in some 'almost' functional environment. Readers can refer to [4] for detail. For example, the following ISO C++ example returns a reference instead of a value in indexing program.

```
template<typename T>
T& getAt(List<T>* xs, int n) {
  while (n--)
    xs = xs->next;
  return xs->key;
}
```

So that we can use this function to mutate the 2nd element as below.

```
List<int>* xs = cons(1, cons(2, cons<int>(3, NULL)));
getAt(xs, 1) = 4;
```

In an impure functional environment, such as Scheme/Lisp, to set the i-th element to a given value can be implemented by mutate the referenced cell directly as well.

```
(define (set-at! lst i x)
  (if (= i 0)
      (set-car! lst x)
      (set-at! (cdr lst) (- i 1) x)))
```

This program first checks if the index i is zero, if so, it mutate the first element of the list to given value x; otherwise, it deduces the index i by one, and tries to mutate the rest of the list at this new index with value x. This function doesn't return meaningful value. It is for use of side-effect. For instance, the following code mutates the 2nd element in a list.

```
(define lst '(1 2 3 4 5))
(set-at! lst 1 4)
(display lst)

(1 4 3 4 5)
```

In order to realize a purely functional $setAt(L, i, x)$ algorithm, we need avoid directly mutating the cell, but creating a new one:

- Edge case: If we want to set the value of the first element ($i = 0$), we construct a new list, with the new value and the sub-list of the previous one;
- Otherwise, we construct a new list, with the previous first element, and a new sub-list, which has the $(i-1)$-th element set with the new value.

This recursive description can be formalized by the following equation.

$$setAt(L, i, x) = \begin{cases} cons(x, L') & : & i = 0 \\ cons(l_1, setAt(L', i-1, x)) & : & otherwise \end{cases} \tag{A.9}$$

Comparing the below Scheme/Lisp implementation to the previous one reveals the difference from imperative mutating.

```
(define (set-at lst i x)
  (if (= i 0)
      (cons x (cdr lst))
      (cons (car lst) (set-at (cdr lst) (- i 1) x))))
```

Here we skip the error handling for out-of-bound error etc. Again, similar to the random access algorithm, the performance is bound to linear time, as traverse is need to locate the position to set the value.

insertion

There are two semantics about list insertion. One is to insert an element at a given position, which can be denoted as $insert(L, i, x)$. The algorithm is close to $setAt(L, i, x)$; The other is to insert an element to a sorted list, so that the the result list is still sorted.

Let's first consider how to insert an element x at a given position i. The obvious thing is that we need firstly traverse i elements to get to the position, the rest of work is to construct a new sub-list with x being the head of this sub-list. Finally, we construct the whole result by attaching this new sub-list to the end of the first i elements.

The algorithm can be described accordingly to this idea. If we want to insert an element x to a list L at i.

- Edge case: If i is zero, then the insertion turns to be a trivial 'cons' operation – $cons(x, L)$;
- Otherwise, we recursively *insert* x to the sub-list L' at position $i - 1$; then construct the first element with this result.

Below equation formalizes the insertion algorithm.

$$insert(L, i, x) = \begin{cases} cons(x, L) & : \quad i = 0 \\ cons(l_1, insert(L', i - 1, x)) & : \quad otherwise \end{cases} \tag{A.10}$$

The following Haskell program implements this algorithm.

```
insert xs 0 y = y:xs
insert (x:xs) i y = x : insert xs (i-1) y
```

This algorithm doesn't handle the out-of-bound error. However, we can interpret the case, that the position i exceeds the length of the list as appending. Readers can considering about it in the exercise of this section.

The algorithm can also be designed imperatively: If the position is zero, just construct the new list with the element to be inserted as the first one; Otherwise, we record the head of the list, then start traversing the list i steps. We also need an extra variable to memorize the previous position for the later list insertion operation. Below is the pseudo code.

function INSERT(L, i, x)
  if $i = 0$ **then**
    return CONS(x, L)
  $H \leftarrow L$
  $p \leftarrow L$

while $i \neq 0$ **do**
$p \leftarrow L$
$L \leftarrow \textsc{Rest}(L)$
$i \leftarrow i - 1$
$\textsc{Rest}(p) \leftarrow \textsc{Cons}(x, L)$
return H

And the ISO C++ example program is given by translating this algorithm.

```
template<typename T>
List<T>* insert(List<T>* xs, int i , int x) {
  List<T> *head, *prev;
  if (i == 0)
    return cons(x, xs);
  for (head = xs; i; --i, xs = xs→next)
    prev = xs;
  prev→next = cons(x, xs);
  return head;
}
```

If the list L is sorted, that is for any position $1 \leq i \leq j \leq N$, we have $l_i \leq l_j$. We can design an algorithm which inserts a new element x to the list, so that the result list is still sorted.

$$insert(x, L) = \begin{cases} cons(x, \Phi) & : & L = \Phi \\ cons(x, L) & : & x < l_1 \\ cons(l_1, insert(x, L')) & : & otherwise \end{cases} \tag{A.11}$$

The idea is that, to insert an element x to a sorted list L:

- If either L is empty or x is less than the first element in L, we just put x in front of L to construct the result;
- Otherwise, we recursively insert x to the sub-list L'.

The following Haskell program implements this algorithm. Note that we use $\leq$, to determine the ordering. Actually this constraint can be loosened to the strict less ($<$), that if elements can be compare in terms of $<$, we can design a program to insert element so that the result list is still sorted. Readers can refer to the chapters of sorting in this book for details about ordering.

```
insert y [] = [y]
insert y xs@(x:xs') = if y ≤ x then y : xs else x : insert y xs'
```

Since the algorithm need compare the elements one by one, it's also a linear time algorithm. Note that here we use the 'as' notion for pattern matching in Haskell. Readers can refer to [8] and [7] for details.

This ordered insertion algorithm can be designed in imperative manner, for example like the following pseudo code[5].

function $\textsc{Insert}(x, L)$
if $L = \Phi \vee x < \textsc{First}(L)$ **then**
return $\textsc{Cons}(x, L)$

[5]Reader can refer to the chapter 'The evolution of insertion sort' in this book for a minor different one

$H \leftarrow L$
while $\textsc{Rest}(L) \neq \Phi \wedge \textsc{First}(\textsc{Rest}(L)) < x$ **do**
$L \leftarrow \textsc{Rest}(L)$
$\textsc{Rest}(L) \leftarrow \textsc{Cons}(x, \textsc{Rest}(L))$
return H

If either the list is empty, or the new element to be inserted is less than the first element in the list, we can just put this element as the new first one; Otherwise, we record the head, then traverse the list till a position, where x is less than the rest of the sub-list, and put x in that position. Compare this one to the 'insert at' algorithm shown previously, the variable p uses to point to the previous position during traversing is omitted by examine the sub-list instead of current list. The following ISO C++ program implements this algorithm.

```
template<typename T>
List<T>* insert(T x, List<T>* xs) {
  List<T> *head;
  if (!xs || x < xs→key)
    return cons(x, xs);
  for (head = xs; xs→next && xs→next→key < x; xs = xs→next);
  xs→next = cons(x, xs→next);
  return head;
}
```

With this linear time ordered insertion defined, it's possible to implement quadratic time insertion-sort by repeatedly inserting elements to an empty list as formalized in this equation.

$$sort(L) = \begin{cases} \Phi & : \quad L = \Phi \\ insert(l_1, sort(L')) & : \quad otherwise \end{cases} \tag{A.12}$$

This equation says that if the list to be sorted is empty, the result is also empty, otherwise, we can firstly recursively sort all elements except for the first one, then ordered insert the first element to this intermediate result. The corresponding Haskell program is given as below.

```
isort [] = []
isort (x:xs) = insert x (isort xs)
```

And the imperative linked-list base insertion sort is described in the following. That we initialize the result list as empty, then take the element one by one from the list to be sorted, and ordered insert them to the result list.

function $\textsc{Sort}(L)$
$L' \leftarrow \Phi$
while $L \neq \Phi$ **do**
$L' \leftarrow \textsc{Insert}(\textsc{First}(L), L')$
$L \leftarrow \textsc{Rest}(L)$
return L'

Note that, at any time during the loop, the result list is kept sorted. There is a major difference between the recursive algorithm (formalized by the equation) and the procedural one (described by the pseudo code), that the former process the list from right, while the latter from left. We'll see in later section about 'tail-recursion' how to eliminate this difference.

The ISO C++ version of linked-list insertion sort is list like this.

```
template<typename T>
List<T>* isort(List<T>* xs) {
  List<T>* ys = NULL;
  for(; xs; xs = xs→next)
    ys = insert(xs→key, ys);
  return ys;
}
```

There is also a dedicated chapter discusses insertion sort in this book. Please refer to that chapter for more details including performance analysis and fine-tuning.

deletion

In purely functional settings, there is no deletion at all in terms of mutating, the data is persist, what the semantic deletion means is actually to create a 'new' list with all the elements in previous one except for the element being 'deleted'.

Similar to the insertion, there are also two deletion semantics. One is to delete the element at a given position; the other is to find and delete elements of a given value. The first can be expressed as $delete(L, i)$, while the second is $delete(L, x)$.

In order to design the algorithm $delete(L, i)$ (or 'delete at'), we can use the idea which is quite similar to random access and insertion, that we first traverse the list to the specified position, then construct the result list with the elements we have traversed, and all the others except for the next one we haven't traversed yet.

The strategy can be realized in a recursive manner that in order to *delete the i-th element from list L*,

- If i is zero, that we are going to delete the first element of a list, the result is obviously the rest of the list;
- If the list to be removed element is empty, the result is anyway empty;
- Otherwise, we can recursively *delete the $(i-1)$-th element from the sub-list L'*, then construct the final result from the first element of L and this intermediate result.

Note there are two edge cases, and the second case is major used for error handling. This algorithm can be formalized with the following equation.

$$delete(L, i) = \begin{cases} L' & : \quad i = 0 \\ \Phi & : \quad L = \Phi \\ cons(l_1, delete(L', i-1)) & : \end{cases} \tag{A.13}$$

Where $L' = rest(L), l_1 = first(L)$. The corresponding Haskell example program is given below.

```
del (_:xs) 0 = xs
del [] _ = []
del (x:xs) i = x : del xs (i-1)
```

This is a linear time algorithm as well, and there are also alternatives for implementation, for example, we can first split the list at position $i-1$, to get 2 sub-lists L_1 and L_2, then we can concatenate L_1 and L_2'.

The 'delete at' algorithm can also be realized imperatively, that we traverse to the position by looping:

```
function DELETE(L, i)
    if i = 0 then
        return REST(L)
    H ← L
    p ← L
    while i ≠ 0 do
        i ← i − 1
        p ← L
        L ← REST(L)
    REST(p) ← REST(L)
    return H
```

Different from the recursive approach, The error handling for out-of-bound is skipped. Besides that the algorithm also skips the handling of resource releasing which is necessary in environments without GC (Garbage collection). Below ISO C++ code for example, explicitly releases the node to be deleted.

```
template<typename T>
List<T>* del(List<T>* xs, int i) {
  List<T> *head, *prev;
  if (i == 0)
    head = xs→next;
  else {
    for (head = xs; i; --i, xs = xs→next)
      prev = xs;
    prev→next = xs→next;
  }
  xs→next = NULL;
  delete xs;
  return head;
}
```

Note that the statement `xs->next = NULL` is neccessary if the destructor is designed to release the whole linked-list recursively.

The 'find and delete' semantic can further be represented in two ways, one is just find the first occurrence of a given value, and delete this element from the list; The other is to find *ALL* occurrence of this value, and delete these elements. The later is more general case, and it can be achieved by a minor modification of the former. We left the 'find all and delete' algorithm as an exercise to the reader.

The algorithm can be designed exactly as the term 'find and delete' but not 'find then delete', that the finding and deleting are processed in one pass traversing.

- If the list to be dealt with is empty, the result is obviously empty;
- If the list isn't empty, we examine the first element of the list, if it is identical to the given value, the result is the sub list;
- Otherwise, we keep the first element, and recursively find and delete the element in the sub list with the given value. The final result is a list constructed with the kept first element, and the recursive deleting result.

This algorithm can be formalized by the following equation.

$$delete(L, x) = \begin{cases} \Phi & : & L = \Phi \\ L' & : & l_1 = x \\ cons(l_1, delete(L', x)) & : & otherwise \end{cases} \tag{A.14}$$

This algorithm is bound to linear time as it traverses the list to find and delete element. Translating this equation to Haskell program yields the below code, note that, the first edge case is handled by pattern-matching the empty list; while the other two cases are further processed by if-else expression.

```
del [] _ = []
del (x:xs) y = if x == y then xs else x : del xs y
```

Different from the above imperative algorithms, which skip the error handling in most cases, the imperative 'find and delete' realization must deal with the problem that the given value doesn't exist.

```
function DELETE(L, x)
    if L = Φ then                                    ▷ Empty list
        return Φ
    if FIRST(L) = x then
        H ← REST(L)
    else
        H ← L
        while L ≠ Φ ∧ FIRST(L) ≠ x do                ▷ List isn't empty
            p ← L
            L ← REST(L)
        if L ≠ Φ then                                ▷ Found
            REST(p) ← REST(L)
    return H
```

If the list is empty, the result is anyway empty; otherwise, the algorithm traverses the list till either finds an element identical to the given value or to the end of the list. If the element is found, it is removed from the list. The following ISO C++ program implements the algorithm. Note that there are codes release the memory explicitly.

```
template<typename T>
List<T>* del(List<T>* xs, T x) {
  List<T> *head, *prev;
  if (!xs)
    return xs;
  if (xs→key == x)
    head = xs→next;
  else {
    for (head = xs; xs && xs→key != x; xs = xs→next)
      prev = xs;
    if (xs)
      prev→next = xs→next;
  }
  if (xs) {
    xs→next = NULL;
    delete xs;
  }
```

```
  return head;
}
```

concatenate

Concatenation can be considered as a general case for appending, that appending only adds one more extra element to the end of the list, while concatenation adds multiple elements.

However, It will lead to quadratic algorithm if implement concatenation naively by appending, which performs poor. Consider the following equation.

$$concat(L_1, L_2) = \begin{cases} L_1 & : & L_2 = \Phi \\ concat(append(L_1, first(L_2)), rest(L_2)) & : & otherwise \end{cases}$$

Note that each appending algorithm need traverse to the end of the list, which is proportion to the length of L_1, and we need do this linear time appending work $|L_2|$ times, so the total performance is $O(|L_1| + (|L_1| + 1) + ... + (|L_1| + |L_2|)) = O(|L_1||L_2| + |L_2|^2)$.

The key point is that the linking operation of linked-list is fast (constant $O(1)$ time), we can traverse to the end of L_1 only one time, and link the second list to the tail of L_1.

$$concat(L_1, L_2) = \begin{cases} L_2 & : & L_1 = \Phi \\ cons(first(L_1), concat(rest(L_1), L_2)) & : & otherwise \end{cases} \tag{A.15}$$

This algorithm only traverses the first list one time to get the tail of L_1, and then linking the second list with this tail. So the algorithm is bound to linear $O(|L_1|)$ time.

This algorithm is described as the following.

- If the first list is empty, the concatenate result is the second list;
- Otherwise, we concatenate the second list to the sub-list of the first one, and construct the final result with the first element and this intermediate result.

Most functional languages provide built-in functions or operators for list concatenation, for example in ML families `++` is used for this purpose.

```
[] ++ ys = ys
xs ++ [] = xs
(x:xs) ++ ys = x : xs ++ ys
```

Note we add another edge case that if the second list is empty, we needn't traverse to the end of the first one and perform linking, the result is merely the first list.

In imperative settings, concatenation can be realized in constant $O(1)$ time with the augmented tail record. We skip the detailed implementation for this method, reader can refer to the source code which can be download along with this appendix.

The imperative algorithm without using augmented tail record can be described as below.

```
function CONCAT(L_1, L_2)
    if L_1 = Φ then
        return L_2
    if L_2 = Φ then
        return L_1
    H ← L_1
    while REST(L_1) ≠ Φ do
        L_1 ← REST(L_1)
    REST(L_1) ← L_2
    return H
```

And the corresponding ISO C++ example code is given like this.

```
template<typename T>
List<T>* concat(List<T>* xs, List<T>* ys) {
  List<T>* head;
  if (!xs)
    return ys;
  if (!ys)
    return xs;
  for (head = xs; xs→next; xs = xs→next);
  xs→next = ys;
  return head;
}
```

A.3.7 sum and product

Recursive sum and product

It is common to calculate the sum or product of a list of numbers. They are quite similar in terms of algorithm structure. We'll see how to abstract such structure in later section.

In order to calculate the *sum of a list*:

- If the list is empty, the result is zero;
- Otherwise, the result is the first element plus the *sum of the rest of the list*.

Formalize this description gives the following equation.

$$sum(L) = \begin{cases} 0 & : \quad L = \Phi \\ l_1 + sum(L') & : \quad otherwise \end{cases} \tag{A.16}$$

However, we can't merely replace plus to times in this equation to achieve product algorithm, because it always returns zero. We can define the product of empty list as 1 to solve this problem.

$$product(L) = \begin{cases} 1 & : \quad L = \Phi \\ l_1 \times product(L') & : \quad otherwise \end{cases} \tag{A.17}$$

The following Haskell program implements sum and product.

```
sum [] = 0
sum (x:xs) = x + sum xs

product [] = 1
product (x:xs) = x * product xs
```

Both algorithms traverse the whole list during calculation, so they are bound to $O(N)$ linear time.

Tail call recursion

Note that both sum and product algorithms actually compute the result from right to left. We can change them to the normal way, that calculate the *accumulated* result from left to right. For example with sum, the result is actually accumulated from 0, and adds element one by one to this accumulated result till all the list is consumed. Such approach can be described as the following.

When accumulate result of a list by summing:

- If the list is empty, we are done and return the accumulated result;
- Otherwise, we take the first element from the list, accumulate it to the result by summing, and go on processing the rest of the list.

Formalize this idea to equation yields another version of sum algorithm.

$$sum'(A, L) = \begin{cases} A & : \quad L = \Phi \\ sum'(A + l_1, L') & : \quad otherwise \end{cases} \tag{A.18}$$

And sum can be implemented by calling this function by passing start value 0 and the list as arguments.

$$sum(L) = sum'(0, L) \tag{A.19}$$

The interesting point of this approach is that, besides it calculates the result in a normal order from left to right; by observing the equation of $sum'(A, L)$, we found it needn't remember any intermediate results or states when perform recursion. All such states are either passed as arguments (A for example) or can be dropped (previous elements of the list for example). So in a practical implementation, such kind of recursive function can be optimized by eliminating the recursion at all.

We call such kind of function as 'tail recursion' (or 'tail call'), and the optimization of removing recursion in this case as 'tail recursion optimization'[10] because the recursion happens as the final action in such function. The advantage of tail recursion optimization is that the performance can be greatly improved, so that we can avoid the issue of stack overflow in deep recursion algorithms such as sum and product.

Changing the sum and product Haskell programs to tail-recursion manner gives the following modified programs.

```
sum = sum' 0 where
    sum' acc [] = acc
    sum' acc (x:xs) = sum' (acc + x) xs

product = product' 1 where
    product' acc [] = acc
    product' acc (x:xs) = product' (acc * x) xs
```

In previous section about insertion sort, we mentioned that the functional version sorts the elements form right, this can also be modified to tail recursive realization.

$$sort'(A, L) = \begin{cases} A & : & L = \Phi \\ sort'(insert(l_1, A), L') & : & otherwise \end{cases} \tag{A.20}$$

The the sorting algorithm is just calling this function by passing empty list as the accumulator argument.

$$sort(L) = sort'(\Phi, L) \tag{A.21}$$

Implementing this tail recursive algorithm to real program is left as exercise to the reader.

As the end of this sub-section, let's consider an interesting problem, that how to design an algorithm to compute b^N effectively? (refer to problem 1.16 in [5].)

A naive brute-force solution is to repeatedly multiply b for N times from 1, which leads to a linear $O(N)$ algorithm.

function POW(b, N)
  $x \leftarrow 1$
  loop N times
    $x \leftarrow x \times b$
  return x

Actually, the solution can be greatly improved. Consider we are trying to calculate b^8. By the first 2 iterations in above naive algorithm, we got $x = b^2$. At this stage, we needn't multiply x with b to get b^3, we can directly calculate x^2, which leads to b^4. And if we do this again, we get $(b^4)^2 = b^8$. Thus we only need looping 3 times but not 8 times.

An algorithm based on this idea to compute b^N if $N = 2^M$ for some non-negative integer M can be shown in the following equation.

$$pow(b, N) = \begin{cases} b & : & N = 1 \\ pow(b, \frac{N}{2})^2 & : & otherwise \end{cases}$$

It is easy to extend this divide and conquer algorithm so that N can be any non-negative integer.

- For the trivial case, that N is zero, the result is 1;
- If N is even number, we can halve N, and compute $b^{\frac{N}{2}}$ first. Then calculate the square number of this result.
- Otherwise, N is odd. Since $N-1$ is even, we can first recursively compute b^{N-1}, the multiply b one more time to this result.

Below equation formalizes this description.

$$pow(b, N) = \begin{cases} 1 & : & N = 0 \\ pow(b, \frac{N}{2})^2 & : & 2|N \\ b \times pow(b, N-1) & : & otherwise \end{cases} \tag{A.22}$$

However, it's hard to turn this algorithm to be tail-recursive mainly because of the 2nd clause. In fact, the 2nd clause can be alternatively realized by squaring the base number, and halve the exponent.

$$pow(b, N) = \begin{cases} 1 & : & N = 0 \\ pow(b^2, \frac{N}{2}) & : & 2|N \\ b \times pow(b, N-1) & : & otherwise \end{cases} \tag{A.23}$$

With this change, it's easy to get a tail-recursive algorithm as the following, so that $b^N = pow'(b, N, 1)$.

$$pow'(b, N, A) = \begin{cases} A & : & N = 0 \\ pow'(b^2, \frac{N}{2}, A) & : & 2|N \\ pow'(b, N-1, A \times b) & : & otherwise \end{cases} \tag{A.24}$$

Compare to the naive brute-force algorithm, we improved the performance to $O(\lg N)$. Actually, this algorithm can be improved even one more step.

Observe that if we represent N in binary format $N = (a_m a_{m-1} ... a_1 a_0)_2$, we clear know that the computation for b^{2^i} is necessary if $a_i = 1$. This is quite similar to the idea of Binomial heap (reader can refer to the chapter of binomial heap in this book). Thus we can calculate the final result by multiplying all of them for bits with value 1.

For instance, when we compute b^{11}, as $11 = (1011)_2 = 2^3 + 2 + 1$, thus $b^{11} = b^{2^3} \times b^2 \times b$. We can get the result by the following steps.

1. calculate b^1, which is b;
2. Get b^2 from previous result;
3. Get b^{2^2} from step 2;
4. Get b^{2^3} from step 3.

Finally, we multiply the results of step 1, 2, and 4 which yields b^{11}.

Summarize this idea, we can improve the algorithm as below.

$$pow'(b, N, A) = \begin{cases} A & : & N = 0 \\ pow'(b^2, \frac{N}{2}, A) & : & 2|N \\ pow'(b^2, \lfloor \frac{N}{2} \rfloor, A \times b) & : & otherwise \end{cases} \tag{A.25}$$

This algorithm essentially shift N to right for 1 bit each time (by dividing N by 2). If the LSB (Least Significant Bit, which is the lowest bit) is 0, it means N is even. It goes on computing the square of the base, without accumulating the final product (Just like the 3rd step in above example); If the LSB is 1, it means N is odd. It squares the base and accumulates it to the product A; The edge case is when N is zero, which means we exhaust all the bits in N, thus the final result is the accumulator A. At any time, the updated base number b', the shifted exponent number N', and the accumulator A satisfy the invariant that $b^N = b'^{N'} A$.

This algorithm can be implemented in Haskell like the following.

```
pow b n = pow' b n 1 where
  pow' b n acc | n == 0 = acc
               | even n = pow' (b*b) (n `div` 2) acc
               | otherwise = pow' (b*b) (n `div` 2) (acc*b)
```

Compare to previous algorithm, which minus N by one to change it to even when N is odd, this one halves N every time. It exactly runs m rounds, where m is the number of bits of N. However, the performance is still bound to $O(\lg N)$. How to implement this algorithm imperatively is left as exercise to the reader.

Imperative sum and product

The imperative sum and product are just applying plus and times while traversing the list.

function SUM(L)
  $s \leftarrow 0$
  while $L \neq \Phi$ **do**
    $s \leftarrow s +$ FIRST(L)
    $L \leftarrow$ REST(L)
  return s

function PRODUCT(L)
  $p \leftarrow 1$
  while $L \neq \Phi$ **do**
    $p \leftarrow p \times$ FIRST(L)
    $L \leftarrow$ REST(L)
  return p

The corresponding ISO C++ example programs are list as the following.

```
template<typename T>
T sum(List<T>* xs) {
  T s;
  for (s = 0; xs; xs = xs->next)
    s += xs->key;
  return s;
}

template<typename T>
T product(List<T>* xs) {
  T p;
  for (p = 1; xs; xs = xs->next)
    p *= xs->key;
  return p;
}
```

One interesting usage of product algorithm is that we can calculate factorial of N by calculating the product of $\{1, 2, ..., N\}$ that $N! = product([1..N])$.

A.3.8 maximum and minimum

Another very useful use case is to get the minimum or maximum element of a list. We'll see that their algorithm structures are quite similar again. We'll

generalize this kind of feature and introduce about higher level abstraction in later section. For both maximum and minimum algorithms, we assume that the given list isn't empty.

In order to find the minimum element in a list.

- If the list contains only one element, (a singleton list), the minimum element is this one;
- Otherwise, we can firstly find the minimum element of the rest list, then compare the first element with this intermediate result to determine the final minimum value.

This algorithm can be formalized by the following equation.

$$min(L) = \begin{cases} l_1 & : & L = \{l_1\} \\ l_1 & : & l_1 \leq min(L') \\ min(L') & : & otherwise \end{cases} \tag{A.26}$$

In order to get the maximum element instead of the minimum one, we can simply replace the $\leq$ comparison to $\geq$ in the above equation.

$$max(L) = \begin{cases} l_1 & : & L = \{l_1\} \\ l_1 & : & l_1 \geq max(L') \\ max(L') & : & otherwise \end{cases} \tag{A.27}$$

Note that both maximum and minimum actually process the list from right to left. It remind us about tail recursion. We can modify them so that the list is processed from left to right. What's more, the tail recursion version brings us 'on-line' algorithm, that at any time, we hold the minimum or maximum result of the list we examined so far.

$$min'(L, a) = \begin{cases} a & : & L = \Phi \\ min(L', l_1) & : & l_1 < a \\ min(L', a) & : & otherwise \end{cases} \tag{A.28}$$

$$max'(L, a) = \begin{cases} a & : & L = \Phi \\ max(L', l_1) & : & a < l_1 \\ max(L', a) & : & otherwise \end{cases} \tag{A.29}$$

Different from the tail recursion sum and product, we can't pass constant value to min', or max' in practice, this is because we have to pass infinity ($min(L, \infty)$) or negative infinity ($max(L, -\infty)$) in theory, but in a real machine neither of them can be represented since the length of word is limited.

Actually, there is workaround, we can instead pass the first element of the list, so that the algorithms become applicable.

$$\begin{aligned} min(L) &= min(L', l_1) \\ max(L) &= max(L', l_1) \end{aligned} \tag{A.30}$$

The corresponding real programs are given as the following. We skip the none tail recursion programs, as they are intuitive enough. Reader can take them as exercises for interesting.

```
min (x:xs) = min' xs x where
    min' [] a = a
    min' (x:xs) a = if x < a then min' xs x else min' xs a

max (x:xs) = max' xs x where
    max' [] a = a
    max' (x:xs) a = if a < x then max' xs x else max' xs a
```

The tail call version can be easily translated to imperative min/max algorithms.

function MIN(L)
  $m \leftarrow$ FIRST(L)
  $L \leftarrow$ REST(L)
  while $L \neq \Phi$ **do**
    if FIRST(L) $< m$ **then**
      $m \leftarrow$ FIRST(L)
    $L \leftarrow$ REST(L)
  return m

function MAX(L)
  $m \leftarrow$ FIRST(L)
  $L \leftarrow$ REST(L)
  while $L \neq \Phi$ **do**
    if $m <$ FIRST(L) **then**
      $m \leftarrow$ FIRST(L)
    $L \leftarrow$ REST(L)
  return m

The corresponding ISO C++ programs are given as below.

```
template<typename T>
T min(List<T>* xs) {
  T x;
  for (x = xs->key; xs; xs = xs->next)
    if (xs->key < x)
      x = xs->key;
  return x;
}

template<typename T>
T max(List<T>* xs) {
  T x;
  for (x = xs->key; xs; xs = xs->next)
    if (x < xs->key)
      x = xs->key;
  return x;
}
```

Another method to achieve tail-call maximum(and minimum) algorithm is by discarding the smaller element each time. The edge case is as same as before; for recursion case, since there are at least two elements in the list, we can take the first two for comparing, then drop one and go on process the rest. For a list with more than two elements, denote L'' as $rest(rest(L)) = \{l_3, l_4, ...\}$, we have

the following equation.

$$max(L) = \begin{cases} l_1 & : & |L| = 1 \\ max(cons(l_1, L'')) & : & l_2 < l_1 \\ max(L') & : & otherwise \end{cases} \tag{A.31}$$

$$min(L) = \begin{cases} l_1 & : & |L| = 1 \\ min(cons(l_1, L'')) & : & l_1 < l_2 \\ min(L') & : & otherwise \end{cases} \tag{A.32}$$

The relative example Haskell programs are given as below.

```
min [x] = x
min (x:y:xs) = if x < y then min (x:xs) else min (y:xs)

max [x] = x
max (x:y:xs) = if x < y then max (y:ys) else max (x:xs)
```

Exercise A.1

- Given two lists L_1 and L_2, design a algorithm $eq(L_1, L_2)$ to test if they are equal to each other. Here equality means the lengths are same, and at the same time, every elements in both lists are identical.
- Consider varies of options to handle the out-of-bound error case when randomly access the element in list. Realize them in both imperative and functional programming languages. Compare the solutions based on exception and error code.
- Augment the list with a 'tail' field, so that the appending algorithm can be realized in constant $O(1)$ time but not linear $O(N)$ time. Feel free to choose your favorite imperative programming language. Please don't refer to the example source code along with this book before you try it.
- With 'tail' field augmented to list, for which list operations this field must be updated? How it affects the performance?
- Handle the out-of-bound case in insertion algorithm by treating it as appending.
- Write the insertion sort algorithm by only using less than ($<$).
- Design and implement the algorithm that find all the occurrence of a given value and delete them from the list.
- Reimplenent the algorithm to calculate the length of a list in tail-call recursion manner.
- Implement the insertion sort in tail recursive manner.
- Implement the $O(\lg N)$ algorithm to calculate b^N in your favorite imperative programming language. Note that we only need accumulate the intermediate result when the bit is not zero.

A.4 Transformation

In previous section, we list some basic operations for linked-list. In this section, we focus on the transformation algorithms for list. Some of them are corner stones of abstraction for functional programming. We'll show how to use list transformation to solve some interesting problems.

A.4.1 mapping and for-each

It is every-day programming routine that, we need output something as readable string. If we have a list of numbers, and we want to print the list to console like '3 1 2 5 4'. One option is to convert the numbers to strings, so that we can feed them to the printing function. One such trivial conversion program may like this.

$$toStr(L) = \begin{cases} \Phi & : & L = \Phi \\ cons(str(l_1), toStr(L')) & : & otherwise \end{cases} \tag{A.33}$$

The other example is that we have a dictionary which is actually a list of words grouped in their initial letters, for example: [[a, an, another, ...], [bat, bath, bool, bus, ...], ..., [zero, zoo, ...]]. We want to know the frequency of them in English, so that we process some English text, for example, 'Hamlet' or the 'Bible' and augment each of the word with a number of occurrence in these texts. Now we have a list like this:

```
[[(a, 1041), (an, 432), (another, 802), ... ],
 [(bat, 5), (bath, 34), (bool, 11), (bus, 0), ...],
 ...,
 [(zero 12), (zoo, 0), ...]]
```

If we want to find which word in each initial is used most, how to write a program to work this problem out? The output is a list of words that every one has the most occurrence in the group, which is categorized by initial, something like '[a, but, can, ...]'. We actually need a program which can transfer a list of group of augmented words into a list of words.

Let's work it out step by step. First, we need define a function, which takes a list of word - number pairs, and find the word has the biggest number augmented. Sorting is overkill. What we need is just a special $max'()$ function, Note that the $max()$ function developed in previous section can't be used directly. Suppose for a pair of values $p = (a, b)$, function $fst(p) = a$, and $snd(p) = b$ are accessors to extract the values, $max'()$ can be defined as the following.

$$max'(L) = \begin{cases} l_1 & : & |L| = 1 \\ l_1 & : & snd(max'(L')) < snd(l_1) \\ max'(L') & : & otherwise \end{cases} \tag{A.34}$$

Alternatively, we can define a dedicated function to compare word-number of occurrence pair, and generalize the $max()$ function by passing a compare function.

$$less(p_1, p_2) = snd(p_1) < snd(p_2) \tag{A.35}$$

$$maxBy(cmp, L) = \begin{cases} l_1 & : & |L| = 1 \\ l_1 & : & cmp(l_1, maxBy(cmp, L')) \\ maxBy(cmp, L') & : & otherwise \end{cases} \tag{A.36}$$

Then $max'()$ is just a special case of $maxBy()$ with the compare function comparing on the second value in a pair.

$$max'(L) = maxBy(\neg less, L) \tag{A.37}$$

Here we write all functions in purely recursive way, they can be modified in tail call manner. This is left as exercise to the reader.

With $max'()$ function defined, it's possible to complete the solution by processing the whole list.

$$solve(L) = \begin{cases} \Phi & : & L = \Phi \\ cons(fst(max'(l_1)), solve(L')) & : & otherwise \end{cases} \tag{A.38}$$

Map

Compare the $solve()$ function in (A.38) and $toStr()$ function in (A.33), it reveals very similar algorithm structure. although they targets on very different problems, and one is trivial while the other is a bit complex.

The structure of $toStr()$ applies the function $str()$ which can turn a number into string on every element in the list; while $solve()$ first applies $max'()$ function to every element (which is actually a list of pairs), then applies $fst()$ function, which essentially turns a list of pairs into a string. It is not hard to abstract such common structure like the following equation, which is called as *mapping*.

$$map(f, L) = \begin{cases} \Phi & : & L = \Phi \\ cons(f(l_1)), map(f, L')) & : & otherwise \end{cases} \tag{A.39}$$

Because map takes a 'converter' function f as argument, it's called a kind of high-order function. In functional programming environment such as Haskell, mapping can be implemented just like the above equation.

```
map :: (a→b)→[a]→[b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

The two concrete cases we discussed above can all be represented in high order mapping.

$$\begin{aligned} toStr &= map \quad str \\ solve &= map \quad (fst \cdot max') \end{aligned}$$

Where $f \cdot g$ means function composing, that we first apply g then apply f. For instance function $h(x) = f(g(x))$ can be represented as $h = f \cdot g$, reading like function h is composed by f and g. Note that we use Curried form to omit the argument L for brevity. Informally speaking, If we feed a function which needs 2 arguments, for instance $f(x, y) = z$ with only 1 argument, the result turns to be a function which need 1 argument. For instance, if we feed f with

only argument x, it turns to be a new function take one argument y, defined as $g(y) = f(x, y)$, or $g = fx$. Note that x isn't a free variable any more, as it is bound to a value. Reader can refer to any book about functional programming for details about function composing and Currying.

Mapping can also be understood from the domain theory point of view. Consider function $y = f(x)$, it actually defines a mapping from domain of variable x to the domain of value y. (x and y can have different types). If the domains can be represented as set X, and Y, we have the following relation.

$$Y = \{f(x)|x \in X\} \tag{A.40}$$

This type of set definition is called Zermelo Frankel set abstraction (as known as ZF expression) [7]. The different is that here the mapping is from a list to another list, so there can be duplicated elements. In languages support list comprehension, for example Haskell and Python etc (Note that the Python list is a built-in type, but not the linked-list we discussed in this appendix), mapping can be implemented as a special case of list comprehension.

```
map f xs = [ f x | x ← xs]
```

List comprehension is a powerful tool. Here is another example that realizes the permutation algorithm in list comprehension. Many textbooks introduce how to implement all-permutation for a list, such as [7], and [9]. It is possible to design a more general version $perm(L, r)$, that if the length of the list L is N, this algorithm permutes r elements from the total N elements. We know that there are $P_N^r = \frac{N!}{(N-r)!}$ solutions.

$$perm(L, r) = \begin{cases} \{\Phi\} & : \quad r = 0 \vee |L| < r \\ \{\{l\} \cup P | l \in L, P \in perm(L - \{l\}, r - 1)\} & : \quad otherwise \end{cases} \tag{A.41}$$

In this equation, $\{l\} \cup P$ means $cons(l, P)$, and $L - \{l\}$ denotes $delete(L, l)$, which is defined in previous section. If we take zero element for permutation, or there are too few elements (less than r), the result is a list contains a empty list; Otherwise for non-trivial case, the algorithm picks one element l from the list, and recursively permutes the rest $N - 1$ elements by picking up $r - 1$ ones; then it puts all the possible l in front of all the possible $r - 1$ permutations. Here is the Haskell implementation of this algorithm.

```
perm _ 0 = [[]]
perm xs r | length xs < r = [[]]
          | otherwise = [ x:ys | x ←xs, ys ← perm (delete x xs) (r-1)]
```

We'll go back to the list comprehension later in section about filtering.

Mapping can also be realized imperatively. We can apply the function while traversing the list, and construct the new list from left to right. Since that the new element is appended to the result list, we can track the tail position to achieve constant time appending, so the mapping algorithms is linear in terms of the passed in function.

function MAP(f, L)
  $L' \leftarrow \Phi$
  $p \leftarrow \Phi$
  while $L \neq \Phi$ **do**

```
        if p = Φ then
            p ← Cons(f( First(L) ), Φ)
            L' ← p
        else
            Next(p) ← Cons(f( First(L) ), Φ)
            p ← Next(p)
        L ← Next(L)
    return L'
```

Because It is a bit complex to annotate the type of the passed-in function in ISO C++, as it involves some detailed language specific features. See [11] for detail. In fact ISO C++ provides the very same mapping concept as in `std::transform`. However, it needs the reader have knowledge of function object, iterator etc, which are out of the scope of this book. Reader can refer to any ISO C++ STL materials for detail.

For brevity purpose, we switch to Python programming language for example code. So that the type inference can be avoid in compile time. The definition of a simple singly linked-list in Python is give as the following.

```
class List:
    def __init__(self, x = None, xs = None):
        self.key = x
        self.next = xs

def cons(x, xs):
    return List(x, xs)
```

The mapping program, takes a function and a linked-list, and maps the functions to every element as described in above algorithm.

```
def mapL(f, xs):
    ys = prev = List()
    while xs is not None:
        prev.next = List(f(xs.key))
        prev = prev.next
        xs = xs.next
    return ys.next
```

Different from the pseudo code, this program uses a dummy node as the head of the resulting list. So it needn't test if the variable stores the last appending position is NIL. This small trick makes the program compact. We only need drop the dummy node before returning the result.

For each

For the trivial task such as printing a list of elements out, it's quite OK to just print each element without converting the whole list to a list of strings. We can actually simplify the program.

```
function Print(L)
    while L ≠ Φ do
        print First(L)
        L ← Rest(L)
```

More generally, we can pass a procedure such as printing, to this list traverse, so the procedure is performed *for each* element.

function FOR-EACH(L, P)
  while $L \neq \Phi$ **do**
    P(FIRST(L))
    $L \leftarrow$ REST(L)

For-each algorithm can be formalized in recursive approach as well.

$$foreach(L, p) = \begin{cases} u & : \quad L = \Phi \\ do(p(l_1), foreach(L', p)) & : \quad otherwise \end{cases} \tag{A.42}$$

Here u means unit, it's can be understood as doing nothing, The type of it is similar to the 'void' concept in C or java like programming languages. The $do()$ function evaluates all its arguments, discards all the results except for the last one, and returns the last result as the final value of $do()$. It is equivalent to `(begin ...)` in Lisp families, and `do` block in Haskell in some sense. For the details about unit type, please refer to [4].

Note that the for-each algorithm is just a simplified mapping, there are only two minor difference points:

- It needn't form a result list, we care the 'side effect' rather than the returned value;
- For each focus more on traversing, while mapping focus more on applying function, thus the order of arguments are typically arranged as $map(f, L)$ and $foreach(L, p)$.

Some Functional programming facilities provides options for both returning the result list or discarding it. For example Haskell Monad library provides both `mapM`, `mapM_` and `forM`, `forM_`. Readers can refer to language specific materials for detail.

Examples for mapping

We'll show how to use mapping by an example, which is a problem of ACM/ICPC[12]. For sake of brevity, we modified the problem description a bit. Suppose there are N lights in a room, all of them are off. We execute the following process N times:

1. We switch all the lights in the room, so that they are all on;
2. We switch the 2, 4, 6, ... lights, that every other light is switched, if the light is on, it will be off, and it will be on if the previous state is off;
3. We switch every third lights, that the 3, 6, 9, ... are switched;
4. ...

And at the last round, only the last light (the N-th light) is switched.

The question is how many lights are on finally?

Before we show the best answer to this puzzle, let's first work out a naive brute-force solution. Suppose there are N lights, which can be represented as a list of 0, 1 numbers, where 0 means the light is off, and 1 means on. The initial state is a list of N zeros: $\{0, 0, ..., 0\}$.

We can label the light from 1 to N. A mapping can help us to turn the above list into a labeled list[6].

$$map(\lambda_i \cdot (i, 0), \{1, 2, 3, ...N\})$$

This mapping augments each natural number with zero, the result is a list of pairs: $L = \{(1, 0), (2, 0), ..., (N, 0)\}$.

Next we operate this list of pairs N times from 1 to N. For every time i, we switch the second value in this pair if the first label can be divided by i. Consider the fact that $1 - 0 = 1$, and $1 - 1 = 0$, we can realize switching of 0, 1 value x by $1 - x$. At the i-th operation, for light (j, x), if $i|j$, (or $j \mod i = 0$), we then perform switching, otherwise, we leave the light untouched.

$$switch(i, (j, x)) = \begin{cases} (j, 1 - x) & : \quad j \mod i = 0 \\ (j, x) & : \quad otherwise \end{cases} \tag{A.43}$$

The i-th operation on all lights can be realized as mapping again:

$$map(switch(i), L) \tag{A.44}$$

Note that, here we use Curried form of $switch()$ function, which is equivalent to

$$map(\lambda_{(j,x)} \cdot switch(i, (j, x)), L)$$

Here we need define a function $proc()$, which can perform the above mapping on L over and over by N times. One option is to realize it in purely recursive way as the following, so that we can call it like $proc(\{1, 2, ..., N\}, L)$[7].

$$proc(I, L) = \begin{cases} L & : \quad I = \Phi \\ operate(I', map(switch(i_1), L)) & : \quad otherwise \end{cases} \tag{A.45}$$

Where $I = cons(i_1, I')$ if I isn't empty.

At this stage, we can sum the second value of each pair in list L to get the answer. The sum function has been defined in previous section, so the only thing left is mapping.

$$solve(N) = sum(map(snd, proc(\{1, 2, ..., N\}, L))) \tag{A.46}$$

Translating this naive brute-force solution to Haskell yields below program.

```
solve' = sum ∘ (map snd) ∘ proc  where
    proc n = operate [1..n] $ map (λi → (i, 0)) [1..n]
    operate [] xs = xs
    operate (i:is) xs = operate is (map (switch i) xs)
```

Let's see what's the answer for there are 1, 2, ..., 100 lights.

```
[1,1,1,2,2,2,2,2,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,5,
6,6,6,6,6,6,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8,
8,8,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,10]
```

[6] Readers who are familiar with functional programming, may use zipping to achieve this. We'll explain zipping in later section.

[7] This can also be realized by folding, which will be explained in later section.

This result is interesting:

- the first 3 answers are 1;
- the 4-th to the 8-th answers are 2;
- the 9-th to the 15-th answers are 3;
- ...

It seems that the i^2-th to the $((i+1)^2-1)$-th answers are i. Actually, we can prove this fact as the following.

Proof. Given N lights, labeled from 1 to N, consider which lights are on finally. Since the initial states for all lights are off, we can say that, the lights which are manipulated odd times are on. For every light i, it will be switched at the j round if i can be divided by j (denote as $j|i$). So only the lights which have odd number of factors are on at the end.

So the key point to solve this puzzle, is to find all numbers which have odd number of factors. For any positive integer N, denote S the set of all factors of N. S is initialized to Φ. if p is a factor of N, there must exist a positive integer q that $N = pq$, which means q is also a factor of N. So we add 2 different factors to the set S if and only if $p \neq q$, which keeps $|S|$ even all the time unless $p = q$. In such case, N is a perfect square number, and we can only add 1 factor to the set S, which leads to an odd number of factors. □

At this stage, we can design a fast solution by finding the number of perfect square numbers under N.

$$solve(N) = \lfloor \sqrt{N} \rfloor \tag{A.47}$$

The next Haskell command verifies that the answer for 1, 2, ..., 100 lights are as same as above.

```
map (floor.sqrt) [1..100]
[1,1,1,2,2,2,2,2,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,
6,6,6,6,6,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,
8,8,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,10]
```

Mapping is generic concept that it doesn't only limit in linked-list, but also can be applied to many complex data structures. The chapter about binary search tree in this book explains how to map on trees. As long as we can traverse a data structure in some order, and the empty data structure can be identified, we can use the same mapping idea. We'll return to this kind of high-order concept in the section of folding later.

A.4.2 reverse

How to reverse a singly linked-list with minimum space is a popular technical interview problem in some companies. The pointer manipulation must be arranged carefully in imperative programming languages such as ANSI C. However, we'll show that, there exists an easy way to write this program:

1. Firstly, write a pure recursive straightforward solution;

2. Then, transform the pure recursive solution to tail-call manner;
3. Finally, translate the tail-call solution to pure imperative pointer operations.

The pure recursive solution is simple enough that we can write it out immediately. In order to *reverse a list* L.

- If L is empty, the reversed result is empty. This is the trivial edge case;
- Otherwise, we can first reverse the rest of the sub-list, then append the first element to the end.

This idea can be formalized to the below equation.

$$reverse(L) = \begin{cases} \Phi & : & L = \Phi \\ append(reverse(L'), l_1) & : & otherwise \end{cases} \tag{A.48}$$

Translating it to Haskell yields below program.

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

However, this solution doesn't perform well, as appending has to traverse to the end of list, which leads to a quadratic time algorithm. It is not hard to improve this program by changing it to tail-call manner. That we can use a accumulator to store the intermediate reversed result, and initialize the accumulated result as empty. So the algorithm is formalized as $reverse(L) = reverse'(L, \Phi)$.

$$reverse'(L, A) = \begin{cases} A & : & L = \Phi \\ reverse'(L', \{l_1\} \cup A) & : & otherwise \end{cases} \tag{A.49}$$

Where $\{l_1\} \cup A$ means $cons(l_1, A)$. Different from appending, it's a constant $O(1)$ time operation. The core idea is that we repeatedly take the element one by one from the head of the original list, and put them in front the accumulated result. This is just like we store all the elements in a stack, them pop them out. This is a linear time algorithm.

Below Haskell program implements this tail-call version.

```
reverse' [] acc = acc
reverse' (x:xs) acc = reverse' xs (x:acc)
```

Since the nature of tail-recursion call needn't book-keep any context (typically by stack), most modern compilers are able to optimize it to a pure imperative loop, and reuse the current context and stack etc. Let's manually do this optimization so that we can get a imperative algorithm.

function REVERSE(L)
  $A \leftarrow \Phi$
  while $L \neq \Phi$ **do**
    $A \leftarrow$ CONS(FIRST(L), A)
    $L \leftarrow$ REST(L)

However, because we translate it directly from a functional solution, this algorithm actually produces a new reversed list, but does not mutate the original one. It is not hard to change it to an in-place solution by reusing L. For example, the following ISO C++ program implements the in-place algorithm. It takes $O(1)$ memory space, and reverses the list in $O(N)$ time.

```
template<typename T>
List<T>* reverse(List<T>* xs) {
  List<T> *p, *ys = NULL;
  while (xs) {
    p = xs;
    xs = xs→next;
    p→next = ys;
    ys = p;
  }
  return ys;
}
```

Exercise A.2

- Implement the algorithm to find the maximum element in a list of pair in tail call approach in your favorite programming language.

A.5 Extract sub-lists

Different from arrays which are capable to slice a continuous segment fast and easily, It needs more work to extract sub lists from singly linked list. Such operations are typically linear algorithms.

A.5.1 take, drop, and split-at

Taking first N elements from a list is semantically similar to extract sub list from the very left like $sublist(L, 1, N)$, where the second and the third arguments to $sublist$ are the positions the sub-list starts and ends. For the trivial edge case, that either N is zero or the list is empty, the sub list is empty; Otherwise, we can recursively take the first $N - 1$ elements from the rest of the list, and put the first element in front of it.

$$take(N, L) = \begin{cases} \Phi & : \quad L = \Phi \vee N = 0 \\ cons(l_1, take(N - 1, L')) & : \quad otherwise \end{cases} \tag{A.50}$$

Note that the edge cases actually handle the out-of-bound error. The following Haskell program implements this algorithm.

```
take _ [] = []
take 0 _ = []
take n (x:xs) = x : take (n-1) xs
```

Dropping on the other hand, drops the first N elements and returns the left as result. It is equivalent to get the sub list from right like $sublist(L, N+1, |L|)$,

where $|L|$ is the length of the list. Dropping can be designed quite similar to taking by discarding the first element in the recursive case.

$$drop(N, L) = \begin{cases} \Phi & : & L = \Phi \\ L & : & N = 0 \\ drop(N-1, L') & : & otherwise \end{cases} \tag{A.51}$$

Translating the algorithm to Haskell gives the below example program.

```
drop _ [] = []
drop 0 L = L
drop n (x:xs) = drop (n-1) xs
```

The imperative taking and dropping are quite straight-forward, that they are left as exercises to the reader.

With taking and dropping defined, extracting sub list at arbitrary position for arbitrary length can be realized by calling them.

$$sublist(L, from, count) = take(count, drop(from - 1, L)) \tag{A.52}$$

or in another semantics by providing left and right boundaries:

$$sublist(L, from, to) = drop(from - 1, take(to, L)) \tag{A.53}$$

Note that the elements in range $[from, to]$ is returned by this function, with both ends included. All the above algorithms perform in linear time.

take-while and drop-while

Compare to taking and dropping, there is another type of operation, that we either keep taking or dropping elements as far as a certain condition is met. The taking and dropping algorithms can be viewed as special cases for take-while and drop-while.

Take-while examines elements one by one as far as the condition is satisfied, and ignore all the rest of elements even some of them satisfy the condition. This is the different point from filtering which we'll explained in later section. Take-while stops once the condition tests fail; while filtering traverses the whole list.

$$takeWhile(p, L) = \begin{cases} \Phi & : & L = \Phi \\ \Phi & : & \neg p(l_1) \\ cons(l_1, takeWhile(p, L')) & : & otherwise \end{cases} \tag{A.54}$$

Take-while accepts two arguments, one is the predicate function p, which can be applied to element in the list and returns Boolean value as result; the other argument is the list to be processed.

It is easy to define the drop-while symmetrically.

$$dropWhile(p, L) = \begin{cases} \Phi & : & L = \Phi \\ L & : & \neg p(l_1) \\ dropWhile(p, L') & : & otherwise \end{cases} \tag{A.55}$$

The corresponding Haskell example programs are given as below.

```
takeWhile _ [] = []
takeWhile p (x:xs) = if p x then x : takeWhile p xs else []

dropWhile _ [] = []
dropWhile p xs@(x:xs') = if p x then dropWhile p xs' else xs
```

split-at

With taking and dropping defined, splitting-at can be realized trivially by calling them.

$$splitAt(i, L) = (take(i, L), drop(i, L)) \tag{A.56}$$

A.5.2 breaking and grouping

breaking

Breaking can be considered as a general form of splitting, instead of splitting at a given position, breaking examines every element for a certain predicate, and finds the longest prefix of the list for that condition. The result is a pair of sub-lists, one is that longest prefix, the other is the rest.

There are two different breaking semantics, one is to pick elements satisfying the predicate as long as possible; the other is to pick those don't satisfy. The former is typically defined as *span*, while the later as *break*.

Span can be described, for example, in such recursive manner: In order to span a list L for predicate p:

- If the list is empty, the result for this edge trivial case is a pair of empty lists (Φ, Φ);
- Otherwise, we test the predicate against the first element l_1, if l_1 satisfies the predicate, we denote the intermediate result for spanning the rest of list as $(A, B) = span(p, L')$, then we put l_1 in front of A to get pair $(\{l_1\} \cup A, B)$, otherwise, we just return (Φ, L) as the result.

For breaking, we just test the negate of predicate and all the others are as same as spanning. Alternatively, one can define breaking by using span as in the later example program.

$$span(p, L) = \begin{cases} (\Phi, \Phi) & : & L = \Phi \\ (\{l_1\} \cup A, B) & : & p(l_1) = True, (A, B) = span(p, L') \\ (\Phi, L) & : & otherwise \end{cases} \tag{A.57}$$

$$break(p, L) = \begin{cases} (\Phi, \Phi) & : & L = \Phi \\ (\{l_1\} \cup A, B) & : & \neg p(l_1), (A, B) = break(p, L') \\ (\Phi, L) & : & otherwise \end{cases} \tag{A.58}$$

Note that both functions only find the longest *prefix*, they stop immediately when the condition is fail even if there are elements in the rest of the list meet the predicate (or not). Translating them to Haskell gives the following example program.

```
span _ [] = ([], [])
span p xs@(x:xs') = if p x then let (as, bs) = span xs' in (x:as, bs) else ([], xs)

break p = span (not ∘ p)
```

Span and break can also be realized imperatively as the following.

function SPAN(p, L)
  $A \leftarrow \Phi$
  while $L \neq \Phi \wedge p(l_1)$ **do**
    $A \leftarrow$ CONS(l_1, A)
    $L \leftarrow$ REST(L)
  return (A, L)

function BREAK(p, L)
  return SPAN($\neg p, L$)

This algorithm creates a new list to hold the longest prefix, another option is to turn it into in-place algorithm to reuse the spaces as in the following Python example.

```
def span(p, xs):
    ys = xs
    last = None
    while xs is not None and p(xs.key):
        last = xs
        xs = xs.next
    if last is None:
        return (None, xs)
    last.next = None
    return (ys, xs)
```

Note that both span and break need traverse the list to test the predicate, thus they are linear algorithms bound to $O(N)$.

grouping

Grouping is a commonly used operation to solve the problems that we need divide the list into some small groups. For example, Suppose we want to group the string 'Mississippi', which is actual a list of char { 'M', 's', 's', 'i', 's', 's', 'i', 'p', 'p', 'i'}. into several small lists in sequence, that each one contains consecutive identical characters. The grouping operation is expected to be:

```
group('Mississippi') = { 'M'', 'i', 'ss', 'i', 'ss', 'i', 'pp', 'i'}
```

Another example, is that we have a list of numbers:

$$L = \{15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2\}$$

We want to divide it into several small lists, that each sub-list is ordered descending. The grouping operation is expected to be :

$$group(L) = \{\{15, 9, 0\}, \{12, 11, 7\}, \{10, 5\}, \{6\}, \{13, 1\}, \{4\}, \{8, 3\}, \{14, 2\}\}$$

Both cases play very important role in real algorithms. The string grouping is used in creating Trie/Patricia data structure, which is a powerful tool in string searching area; The ordered sub-list grouping can be used in nature merge sort. There are dedicated chapters in this book explain the detail of these algorithms.

It is obvious that we need abstract the grouping condition so that we know where to break the original list into small ones. This predicate can be passed to the algorithm as an argument like $group(p, L)$, where predicate p accepts two consecutive elements and test if the condition matches.

The first idea to solve the grouping problem is traversing – takes two elements at each time, if the predicate test succeeds, put both elements into a small group; otherwise, only put the first one into the group, and use the second one to initialize another new group. Denote the first two elements (if there are) are l_1, l_2, and the sub-list without the first element as L'. The result is a list of list $G = \{g_1, g_2, ...\}$, denoted as $G = group(p, L)$.

$$group(p, L) = \begin{cases} \{\Phi\} & : & L = \Phi \\ \{\{l_1\}\} & : & |L| = 1 \\ \{\{l_1\} \cup g'_1, g'_2, ...\} & : & p(l_1, l_2), G' = group(p, L') = \{g'_1, g'_2, ...\} \\ \{\{l_1\}, g'_1, g'_2, ...\} & : & otherwise \end{cases} \tag{A.59}$$

Note that $\{l_1\} \cup g'_1$ actually means $cons(l_1, g'_1)$, which performs in constant time. This is a linear algorithm performs proportion to the length of the list, it traverses the list in one pass which is bound to $O(N)$. Translating this program to Haskell gives the below example code.

```
group _ [] = [[]]
group _ [x] = [[x]]
group p (x:xs@(x':_)) | p x x' = (x:ys):yss
                      | otherwise = [x]:r
  where
    r@(ys:yss) = group p xs
```

It is possible to implement this algorithm in imperative approach, that we initialize the result groups as $\{l_1\}$ if L isn't empty, then we traverse the list from the second one, and append to the last group if the two consecutive elements satisfy the predicate; otherwise we start a new group.

function GROUP(p, L)
  if $L = \Phi$ **then**
    return $\{\Phi\}$
  $x \leftarrow$ FIRST(L)
  $L \leftarrow$ REST(L)
  $g \leftarrow \{x\}$
  $G \leftarrow \{g\}$
  while $L \neq \Phi$ **do**
    $y \leftarrow$ FIRST(L)
    if $p(x, y)$ **then**
      $g \leftarrow$ APPEND(g, y)
    else
      $g \leftarrow \{y\}$
      $G \leftarrow$ APPEND(G, g)

$x \leftarrow y$
$L \leftarrow \text{NEXT}(L)$
return G

However, different from the recursive algorithm, this program performs in quadratic time if the appending function isn't optimized by storing the tail position. The corresponding Python program is given as below.

```
def group(p, xs):
    if xs is None:
        return List(None)
    (x, xs) = (xs.key, xs.next)
    g = List(x)
    G = List(g)
    while xs is not None:
        y = xs.key
        if p(x, y):
            g = append(g, y)
        else:
            g = List(y)
            G = append(G, g)
        x = y
        xs = xs.next
    return G
```

With the grouping function defined, the two example cases mentioned at the beginning of this section can be realized by passing different predictions.

$$group(=, \{m, i, s, s, i, s, s, i, p, p, i\}) = \{\{M\}, \{i\}, \{ss\}, \{i\}, \{ss\}, \{i\}, \{pp\}, \{i\}\}$$

$$\begin{aligned} &group(\geq, \{15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2\}) \\ &= \{\{15, 9, 0\}, \{12, 11, 7\}, \{10, 5\}, \{6\}, \{13, 1\}, \{4\}, \{8, 3\}, \{14, 2\}\} \end{aligned}$$

Another solution is to use the *span* function we have defined to realize grouping. We pass a predicate to span, which will break the list into two parts: The first part is the longest sub-list satisfying the condition. We can repeatedly apply the span with the same predication to the second part, till it becomes empty.

However, the predicate function we passed to span is *an unary function*, that it takes an element as argument, and test if it satisfies the condition. While in grouping algorithm, the predicate function is *a binary function*. It takes two adjacent elements for testing. The solution is that, we can use currying and pass the first element to the binary predicate, and use it to test the rest of elements.

$$group(p, L) = \begin{cases} \{\Phi\} & : & L = \Phi \\ \{\{l_1\} \cup A\} \cup group(p, B) & : & otherwise \end{cases} \tag{A.60}$$

Where $(A, B) = span(\lambda_x \cdot p(l_1, x), L')$ is the result of spanning on the rest sub-list of L.

Although this new defined grouping function can generate correct result for the first case as in the following Haskell code snippet.

```
groupBy (==) "Mississippi"
["m","i","ss","i","ss","i","pp","i"]
```

However, it seems that this algorithm can't group the list of numbers into ordered sub lists.

```
groupBy (≥) [15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]
[[15,9,0,12,11,7,10,5,6,13,1,4,8,3,14,2]]
```

The reason is because that, as the first element 15 is used as the left parameter to $\geq$ operator for span, while 15 is the maximum value in this list, so the span function ends with putting all elements to A, and B is left empty. This might seem a defect, but it is actually the correct behavior if the semantic is to group equal elements together.

Strictly speaking, the equality predicate must satisfy three properties: reflexive, transitive, and symmetric. They are specified as the following.

- Reflexive. $x = x$, which says that any element is equal to itself;
- Transitive. $x = y, y = z \Rightarrow x = z$, which says that if two elements are equal, and one of them is equal to another, then all the tree are equal;
- Symmetric. $x = y \Leftrightarrow y = x$, which says that the order of comparing two equal elements doesn't affect the result.

When we group character list "Mississippi", the equal (=) operator is used, which obviously conforms these three properties. So that it generates correct grouping result. However, when passing ($\geq$) as equality predicate, to group a list of numbers, it violets both reflexive and symmetric properties, that is reason why we get wrong grouping result.

This fact means that the second algorithm we designed by using span, limits the semantic to strictly equality, while the first one does not. It just tests the condition for every two adjacent elements, which is much weaker than equality.

Exercise A.3

1. Implement the in-place imperative taking and dropping algorithms in your favorite programming language, note that the out of bound cases should be handled. Please try both languages with and without GC (Garbage Collection) support.
2. Implement take-while and drop-while in your favorite imperative programming language. Please try both dynamic type language and static type language (with and without type inference). How to specify the type of predicate function as generic as possible in static type system?
3. Consider the following definition of span.

$$span(p, L) = \begin{cases} (\Phi, \Phi) & : & L = \Phi \\ (\{l_1\} \cup A, B) & : & p(l_1) = True, (A, B) = span(p, L') \\ (A, \{l_1\} \cup B) & : & otherwise \end{cases}$$

   What's the difference between this algorithm and the the one we've shown in this section?
4. Implement the grouping algorithm by using span in imperative way in your favorite programming language.

A.6 Folding

We are ready to introduce one of the most critical concept in high order programming, folding. It is so powerful tool that almost all the algorithms so far in this appendix can be realized by folding. Folding is sometimes be named as reducing (the abstracted concept is identical to the buzz term 'map-reduce' in cloud computing in some sense). For example, both STL and Python provide reduce function which realizes partial form of folding.

A.6.1 folding from right

Remind the sum and product definition in previous section, they are quite similar actually.

$$sum(L) = \begin{cases} 0 & : \quad L = \Phi \\ l_1 + sum(L') & : \quad otherwise \end{cases}$$

$$product(L) = \begin{cases} 1 & : \quad L = \Phi \\ l_1 \times product(L') & : \quad otherwise \end{cases}$$

It is obvious that they have same structure. What's more, if we list the insertion sort definition, we can find that it also shares this structure.

$$sort(L) = \begin{cases} \Phi & : \quad L = \Phi \\ insert(l_1, sort(L')) & : \quad otherwise \end{cases}$$

This hint us that we can abstract this essential common structure, so that we needn't repeat it again and again. Observing *sum*, *product*, and *sort*, there are two different points which we can parameterize.

- The result of the trivial edge case varies. It is zero for sum, 1 for product, and empty list for sorting.
- The function applied to the first element and the intermediate result varies. It is plus for sum, multiply for product, and ordered-insertion for sorting.

If we parameterize the result of trivial edge case as initial value z (stands for abstract zero concept), the function applied in recursive case as f (which takes two parameters, one is the first element in the list, the other is the recursive result for the rest of the list), this common structure can be defined as something like the following.

$$proc(f, z, L) = \begin{cases} z & : \quad L = \Phi \\ f(l_1, proc(f, z, L')) & : \quad otherwise \end{cases}$$

That's it, and we should name this common structure a better name instead of the meaningless 'proc'. Let's see the characteristic of this common structure. For list $L = \{x_1, x_2, ..., x_N\}$, we can expand the computation like the following.

$$\begin{aligned} proc(f, z, L) &= f(x_1, proc(f, z, L') \\ &= f(x_1, f(x_2, proc(f, z, L'')) \\ &... \\ &= f(x_1, f(x_2, f(..., f(x_N, f(f, z, \Phi))...) \\ &= f(x_1, f(x_2, f(..., f(x_N, z))...) \end{aligned}$$

Since f takes two parameters, it's a binary function, thus we can write it in infix form. The infix form is defined as below.

$$x \oplus_f y = f(x, y) \tag{A.61}$$

The above expanded result is equivalent to the following by using infix notion.

$$proc(f, z, L) = x_1 \oplus_f (x_2 \oplus_f (...(x_N \oplus_f z))...)$$

Note that the parentheses are necessary, because the computation starts from the right-most $(x_N \oplus_f z)$, and repeatedly fold to left towards x_1. This is quite similar to folding a Chinese hand-fan as illustrated in the following photos. A Chinese hand-fan is made of bamboo and paper. Multiple bamboo frames are stuck together with an axis at one end. The arc shape paper is fully expanded by these frames as shown in Figure A.3 (a); The fan can be closed by folding the paper. Figure A.3 (b) shows that some part of the fan is folded from right. After these folding finished, the fan results a stick, as shown in Figure A.3 (c).

(a) A folding fan fully opened.

(b) The fan is partly folded on right.

(c) The fan is fully folded, closed to a stick.

Figure A.3: Folding a Chinese hand-fan

We can considered that each bamboo frame along with the paper on it as an element, so these frames forms a list. A unit process to close the fan is to rotate

a frame for a certain angle, so that it lays on top of the collapsed part. When we start closing the fan, the initial collapsed result is the first bamboo frame. The close process is folding from one end, and repeatedly apply the unit close steps, till all the frames is rotated, and the folding result is a stick closed form.

Actually, the sum and product algorithms exactly do the same thing as closing the fan.

$$\begin{aligned} sum(\{1,2,3,4,5\}) &= 1+(2+(3+(4+5))) \\ &= 1+(2+(3+9)) \\ &= 1+(2+12) \\ &= 1+14 \\ &= 15 \end{aligned}$$

$$\begin{aligned} product(\{1,2,3,4,5\}) &= 1\times(2\times(3\times(4\times 5))) \\ &= 1\times(2\times(3\times 20)) \\ &= 1\times(2\times 60) \\ &= 1\times 120 \\ &= 120 \end{aligned}$$

In functional programming, we name this process *folding*, and particularly, since we execute from the most inner structure, which starts from the right-most one. This type of folding is named *folding right*.

$$foldr(f,z,L) = \begin{cases} z & : \quad L=\Phi \\ f(l_1, foldr(f,z,L')) & : \quad otherwise \end{cases} \tag{A.62}$$

Let's see how to use fold-right to realize sum and product.

$$\begin{aligned} \sum_{i=1}^{N} x_i &= x_1+(x_2+(x_3+...+(x_{N_1}+x_N))...) \\ &= foldr(+,0,\{x_1,x_2,...,x_N\}) \end{aligned} \tag{A.63}$$

$$\begin{aligned} \prod_{i=1}^{N} x_i &= x_1\times(x_2\times(x_3\times...+(x_{N_1}\times x_N))...) \\ &= foldr(\times,1,\{x_1,x_2,...,x_N\}) \end{aligned} \tag{A.64}$$

The insertion-sort algorithm can also be defined by using folding right.

$$sort(L) = foldr(insert,\Phi,L) \tag{A.65}$$

A.6.2 folding from left

As mentioned in section of 'tail recursive' call. Both pure recursive sum and product compute from right to left and they must book keep all the intermediate results and contexts. As we abstract fold-right from the very same structure, folding from right does the book keeping as well. This will be expensive if the list is very long.

Since we can change the realization of sum and product to tail-recursive call manner, it quite possible that we can provide another folding algorithm, which processes the list from left to right in normal order, and enable the tail-call optimization by reusing the same context.

Instead of induction from sum, product and insertion, we can directly change the folding right to tail call. Observe that the initial value z, actually represents the intermediate result at any time. We can use it as the accumulator.

$$foldl(f, z, L) = \begin{cases} z & : \quad L = \Phi \\ foldl(f, f(z, l_1), L') & : \quad otherwise \end{cases} \tag{A.66}$$

Every time when the list isn't empty, we take the first element, apply function f on the accumulator z and it to get a new accumulator $z' = f(z, l_1)$. After that we can repeatedly folding with the very same function f, the updated accumulator z', and list L'.

Let's verify that this tail-call algorithm actually folding from left.

$$\begin{aligned} \sum_{i=1}^{5} i &= foldl(+, 0, \{1, 2, 3, 4, 5\}) \\ &= foldl(+, 0 + 1, \{2, 3, 4, 5\}) \\ &= foldl(+, (0 + 1) + 2\{3, 4, 5\} \\ &= foldl(+, ((0 + 1) + 2) + 3, \{4, 5\}) \\ &= foldl(+, (((0 + 1) + 2) + 3) + 4, \{5\}) \\ &= foldl(+, ((((0 + 1) + 2 + 3) + 4 + 5, \Phi) \\ &= 0 + 1 + 2 + 3 + 4 + 5 \end{aligned}$$

Note that, we actually delayed the evaluation of $f(z, l_1)$ in every step. (This is the exact behavior in system support lazy-evaluation, for instance, Haskell. However, in strict system such as standard ML, it's not the case.) Actually, they will be evaluated in sequence of $\{1, 3, 6, 10, 15\}$ in each call.

Generally, folding-left can be expanded in form of

$$foldl(f, z, L) = f(f(...(f(f(z, l_1), l_2), ..., l_N) \tag{A.67}$$

Or in infix manner as

$$foldl(f, z, L) = ((...(z \oplus_f l_1) \oplus_f l_2) \oplus_f ...) \oplus l_N \tag{A.68}$$

With folding from left defined, sum, product, and insertion-sort can be transparently implemented by calling $foldl$ as $sum(L) = foldl(+, 0, L)$, $product(L) = foldl(+, 1, L)$, and $sort(L) = foldl(insert, \Phi, L)$. Compare with the folding-right version, they are almost same at first glares, however, the internal implementation differs.

Imperative folding and generic folding concept

The tail-call nature of folding-left algorithm is quite friendly for imperative settings, that even the compiler isn't equipped with tail-call recursive optimization, we can anyway implement the folding in while-loop manually.

function FOLD(f, z, L)
  while $L \neq \Phi$ **do**
    $z \leftarrow f(z,$ FIRST(L))
    $L \leftarrow$ REST(L)
  return z

Translating this algorithm to Python yields the following example program.

```
def fold(f, z, xs):
    for x in xs:
        z = f(z, x)
    return z
```

Actually, Python provides built-in function 'reduce' which does the very same thing. (in ISO C++, this is provided as reduce algorithm in STL.) Almost no imperative environment provides folding-right function because it will cause stack overflow problem if the list is too long. However, there still exist cases that the folding from right semantics is necessary. For example, one defines a container, which only provides insertion function to the head of the container, but there is no any appending method, so that we want such a $fromList$ tool.

$$fromList(L) = foldr(insertHead, empty, L)$$

Calling $fromList$ with the insertion function as well as an empty initialized container, can turn a list into the special container. Actually the singly linked-list is such a container, which performs well on insertion to the head, but poor to linear time if appending on the tail. Folding from right is quite nature when duplicate a linked-list while keeps the elements ordering. While folding from left will generate a reversed list.

In such cases, there exists an alternative way to implement imperative folding right by first reverse the list, and then folding the reversed one from left.

function FOLD-RIGHT(f, z, L)
  return FOLD(f, z, REVERSE(L))

Note that, here we must use the tail-call version of reversing, or the stack overflow issue still exists.

One may think that folding-left should be chosen in most cases over folding-right because it's friendly for tail-recursion call optimization, suitable for both functional and imperative settings, and it's an online algorithm. However, folding-right plays a critical role when the input list is infinity and the binary function f is lazy. For example, below Haskell program wraps every element in an infinity list to a singleton, and returns the first 10 result.

```
take 10 $ foldr (λx xs →[x]:xs) [] [1..]
[[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]
```

This can't be achieved by using folding left because the outer most evaluation can't be finished until all the list being processed. The details is specific to lazy evaluation feature, which is out of the scope of this book. Readers can refer to [13] for details.

Although the main topic of this appendix is about singly linked-list related algorithms, the folding concept itself is generic which doesn't only limit to list, but also can be applied to other data structures.

We can fold a tree, a queue, or even more complicated data structures as long as we have the following:

- The empty data structure can be identified for trivial edge case; (e.g. empty tree)
- We can traverse the data structure (e.g. traverse the tree in pre-order).

Some languages provide this high-level concept support, for example, Haskell achieve this via *monoid*, readers can refer to [8] for detail.

There are many chapters in this book use the widen concept of folding.

A.6.3 folding in practice

We have seen that *max*, *min*, and insertion sort all can be realized in folding. The brute-force solution for 'drunk jailer' puzzle shown in mapping section can also be designed by mixed use of mapping and folding.

Remind that we create a list of pairs, each pair contains the number of the light, and the on-off state. After that we process from 1 to N, switch the light if the number can be divided. The whole process can be viewed as folding.

$$fold(step, \{(1,0),(2,0),...,(N,0)\}, \{1,2,...,N\})$$

The initial value is the very first state, that all the lights are off. The list to be folding is the operations from 1 to N. Function *step* takes two arguments, one is the light states pair list, the other is the operation time i. It then maps on all lights and performs switching. We can then substitute the *step* with mapping.

$$fold(\lambda_{L,i} \cdot map(switch(i), L), \{(1,0),(2,0),...,(N,0)\}, \{1,2,...,N\})$$

We'll simplify the λ notation, and directly write $map(switch(i), l)$ for brevity purpose. The result of this folding is the final states pairs, we need take the second one of the pair for each element via mapping, then calculate the summation.

$$sum(map(snd, fold(map(switch(i), L), \{(1,0),(2,0),...,(N,0)\}, \{1,2,...,N\}))) \tag{A.69}$$

There are materials provides plenty of good examples of using folding, especially in [1], folding together with fusion law are well explained.

concatenate a list of list

In previous section A.3.6 about concatenation, we explained how to concatenate two lists. Actually, concatenation of lists can be considered equivalent to summation of numbers. Thus we can design a general algorithm, which can concatenate multiple lists into one big list.

What's more, we can realize this general concatenation by using folding. As sum can be represented as $sum(L) = foldr(+, 0, L)$, it's straightforward to write the following equation.

$$concats(L) = foldr(concat, \Phi, L) \tag{A.70}$$

Where L is a list of list, for example $\{\{1,2,3\},\{4,5,6\},\{7,8,9\},...\}$. Function $concat(L_1, L_2)$ is what we defined in section A.3.6.

In some environments which support lazy-evaluation, such as Haskell, this algorithm is capable to concatenate infinite list of list, as the binary function ++ is lazy.

Exercise A.4

- What's the performance of *concats* algorithm? is it linear or quadratic?
- Design another linear time *concats* algorithm without using folding.
- Realize mapping algorithm by using folding.

A.7 Searching and matching

Searching and matching are very important algorithms. They are not only limited to linked list, but also applicable to a wide range of data structures. We just scratch the surface of searching and matching in this appendix. There are dedicated chapters explain about them in this book.

A.7.1 Existence testing

The simplest searching case is to test if a given element exists in a list. A linear time traverse can solve this problem. In order to determine element x exists in list L:

- If the list is empty, it's obvious that the element doesn't exist in L;
- If the first element in the list equals to x, we know that x exists;
- Otherwise, we need recursively test if x exists in the rest sub-list L';

This simple description can be directly formalized to equation as the following.

$$x \in L = \begin{cases} False & : & L = \Phi \\ True & : & l_1 = x \\ x \in L' & : & otherwise \end{cases} \tag{A.71}$$

This is definitely a linear algorithm which is bound to $O(N)$ time. The best case happens in the two trivial clauses that either the list is empty or the first element is what we are finding; The worst case happens when the element doesn't exist at all or it is the last element. In both cases, we need traverse the whole list. If the probability is equal for all the positions, the average case takes about $\frac{N+1}{2}$ steps for traversing.

This algorithm is so trivial that we left the implementation as exercise to the reader. If the list is ordered, one may expect to improve the algorithm to logarithm time but not linear. However, as we discussed, since list doesn't support constant time random accessing, binary search can't be applied here. There is a dedicated chapter in this book discusses how to evolve the linked list to binary tree to achieve quick searching.

A.7.2 Looking up

One extra step from existence testing is to find the interesting information stored in the list. There are two typical methods to augment extra data to the element. Since the linked list is chain of nodes, we can store satellite data in the node, then provide $key(n)$ to access the key of the node, $rest(n)$ for the rest sub-list, and $value(n)$ for the augmented data. The other method, is to pair the key

and data, for example $\{(1, hello), (2, world), (3, foo), ...\}$. We'll introduce how to form such pairing list in later section.

The algorithm is almost as same as the existence testing, that it traverses the list, examines the key one by one. Whenever it finds a node which has the same key as what we are looking up, it stops, and returns the augmented data. It is obvious that this is linear strategy. If the satellite data is augmented to the node directly, the algorithm can be defined as the following.

$$lookup(x, L) = \begin{cases} \Phi & : & L = \Phi \\ value(l_1) & : & key(l_1) = x \\ lookup(x, L') & : & otherwise \end{cases} \tag{A.72}$$

In this algorithm, L is a list of nodes which are augmented with satellite data. Note that the first case actually means looking up failure, so that the result is empty. Some functional programming languages, such as Haskell, provide `Maybe` type to handle the possibility of fail. This algorithm can be slightly modified to handle the key-value pair list as well.

$$lookup(x, L) = \begin{cases} \Phi & : & L = \Phi \\ snd(l_1) & : & fst(l_1) = x \\ lookup(x, L') & : & otherwise \end{cases} \tag{A.73}$$

Here L is a list of pairs, functions $fst(p)$ and $snd(p)$ access the first part and second part of the pair respectively.

Both algorithms are in tail-call manner, they can be transformed to imperative looping easily. We left this as exercise to the reader.

A.7.3 finding and filtering

Let's take one more step ahead, looking up algorithm performs linear search by comparing the key of an element is equal to the given value. A more general case is to find an element matching a certain predicate. We can abstract this matching condition as a parameter for this generic linear finding algorithm.

$$find(p, L) = \begin{cases} \Phi & : & L = \Phi \\ l_1 & : & p(l_1) \\ find(p, L') & : & otherwise \end{cases} \tag{A.74}$$

The algorithm traverses the list by examining if the element satisfies the predicate p. It fails if the list is empty while there is still nothing found. This is handled in the first trivial edge case; If the first element in the list satisfies the condition, the algorithm returns the whole element (node), and user can further handle it as he like (either extract the satellite data or do whatever); otherwise, the algorithm recursively perform finding on the rest of the sub-list. Below is the corresponding Haskell example program.

```
find _ [] = Nothing
find p (x:xs) = if p x then Just x else find p xs
```

Translating this to imperative algorithm is straightforward. Here we use 'NIL' to represent the fail case.

function FIND(p, L)
  while $L \neq \Phi$ **do**

```
        if p(FIRST(L)) then
            return FIRST(L)
        L ← REST(L)
    return NIL
```

And here is the Python example of finding.

```
def find(p, xs):
    while xs is not None:
        if p(xs.key):
            return xs
        xs = xs.next
    return None
```

It is quite possible that there are multiple elements in the list which satisfy the precondition. The finding algorithm designed so far just picks the first one it meets, and stops immediately. It can be considered as a special case of finding all elements under a certain condition.

Another viewpoint of finding all elements with a given predicate is to treat the finding algorithm as a black box, the input to this box is a list, while the output is another list contains all elements satisfying the predicate. This can be called as filtering as shown in the below figure.

Figure A.4: The input is the original list $\{x_1, x_2, ..., x_N\}$, the output is a list $\{x'_1, x'_2, ..., x'_M\}$, that for $\forall x'_i$, predicate $p(x'_i)$ is satisfied.

This figure can be formalized in another form in taste of set enumeration. However, we actually enumerate among list instead of a set.

$$filter(p, L) = \{x | x \in L \wedge p(x)\} \tag{A.75}$$

Some environment such as Haskell (and Python for any iterable), supports this form as list comprehension.

```
filter p xs = [ x | x ← xs, p x]
```

And in Python for built-in list as

```
def filter(p, xs):
    return [x for x in xs if p(x)]
```

Note that the Python built-in list isn't singly-linked list as we mentioned in this appendix.

In order to modify the finding algorithm to realize filtering, the found elements are appended to a result list. And instead of stopping the traverse, all the rest of elements should be examined with the predicate.

$$filter(p, L) = \begin{cases} \Phi & : & L = \Phi \\ cons(l_1, filter(p, L')) & : & p(l_1) \\ filter(p, L') & : & otherwise \end{cases} \tag{A.76}$$

This algorithm returns empty result if the list is empty for trivial edge case; For non-empty list, suppose the recursive result of filtering the rest of the sub-list is A, the algorithm examine if the first element satisfies the predicate, it is put in front of A by a 'cons' operation ($O(1)$ time).

The corresponding Haskell program is given as below.

```
filter _ [] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

Although we mentioned that the next found element is 'appended' to the result list, this algorithm actually constructs the result list from the right most to the left, so that appending is avoided, which ensure the linear $O(N)$ performance. Compare this algorithm with the following imperative quadratic realization reveals the difference.

```
function FILTER(p, L)
    L' ← Φ
    while L ≠ Φ do
        if p(FIRST(L)) then
            L' ← APPEND(L', FIRST(L))          ▷ Linear operation
        L ← REST(L)
```

As the comment of appending statement, it's typically proportion to the length of the result list if the tail position isn't memorized. This fact indicates that directly transforming the recursive filter algorithm into tail-call form will downgrade the performance from $O(N)$ to $O(N^2)$. As shown in the below equation, that $filter(p, L) = filter'(p, L, \Phi)$ performs as poorly as the imperative one.

$$filter'(p, L, A) = \begin{cases} A & : & L = \Phi \\ filter'(p, L', A \cup \{l_1\}) & : & p(l_1) \\ filter'(p, L', A) & : & otherwise \end{cases} \tag{A.77}$$

One solution to achieve linear time performance imperatively is to construct the result list in reverse order, and perform the $O(N)$ reversion again (refer to the above section) to get the final result. This is left as exercise to the reader.

The fact of construction the result list from right to left indicates the possibility of realizing filtering with folding-right concept. We need design some combinator function f, so that $filter(p, L) = foldr(f, \Phi, L)$. It requires that function f takes two arguments, one is the element iterated among the list; the other is the intermediate result constructed from right. $f(x, A)$ can be defined as that it tests the predicate against x, if succeed, the result is updated to $cons(x, A)$, otherwise, A is kept same.

$$f(x, A) = \begin{cases} cons(x, A) & : & p(x) \\ A & : & otherwise \end{cases} \tag{A.78}$$

However, the predicate must be passed to function f as well. This can be achieved by using currying, so f actually has the prototype $f(p, x, A)$, and filtering is defined as following.

$$filter(p, L) = foldr(\lambda_{x,A} \cdot f(p, x, A), \Phi, L) \tag{A.79}$$

Which can be simplified by η-conversion. For detailed definition of η-conversion, readers can refer to [2].

$$filter(p, L) = foldr(f(p), \Phi, L) \tag{A.80}$$

The following Haskell example program implements this equation.

```
filter p = foldr f [] where
    f x xs = if p x then x : xs else xs
```

Similar to mapping and folding, filtering is actually a generic concept, that we can apply a predicate on any traversable data structures to get what we are interesting. readers can refer to the topic about monoid in [8] for further reading.

A.7.4 Matching

Matching generally means to find a given pattern among some data structures. In this section, we limit the topic within list. Even this limitation will leads to a very wide and deep topic, that there are dedicated chapters in this book introduce matching algorithms. So we only select the algorithm to test if a given list exists in another (typically longer) list.

Before dive into the algorithm of finding the sub-list at any position, two special edge cases are used for warm up. They are algorithms to test if a given list is either prefix or suffix of another.

In the section about span, we have seen how to find a prefix under a certain condition. prefix matching can be considered as a special case in some sense. That it compares each of the elements between the two lists from the beginning until meets any different elements or pass the end of one list. Define $P \subseteq L$ if P is prefix of L.

$$P \subseteq L = \begin{cases} True & : & P = \Phi \\ False & : & p_1 \neq l_1 \\ P' \subseteq L' & : & otherwise \end{cases} \tag{A.81}$$

This is obviously a linear algorithm. However, We can't use the very same approach to test if a list is suffix of another because it isn't cheap to start from the end of the list and keep iterating backwards. Arrays, on the other hand which support random access can be easily traversed backwards.

As we only need the yes-no result, one solution to realize a linear suffix testing algorithm is to reverse both lists, (which is linear time), and use prefix testing instead. Define $L \supseteq P$ if P is suffix of L.

$$L \supseteq P = reverse(P) \subseteq reverse(L) \tag{A.82}$$

With $\subseteq$ defined, it enables to test if a list is infix of another. The idea is to traverse the target list, and repeatedly applying the prefix testing till any success or arrives at the end.

function IS-INFIX(P, L)
  while $L \neq \Phi$ **do**
    if $P \subseteq L$ **then**
      return TRUE
    $L \leftarrow$ REST(L)

return FALSE

Formalize this algorithm to recursive equation leads to the below definition.

$$infix?(P, L) = \begin{cases} True & : & P \subseteq L \\ False & : & L = \Phi \\ infix?(P, L') & : & otherwise \end{cases} \tag{A.83}$$

Note that there is a tricky implicit constraint in this equation. If the pattern P is empty, it is definitely the infix of any target list. This case is actually covered by the first condition in the above equation because empty list is also the prefix of any list. In most programming languages support pattern matching, we can't arrange the second clause as the first edge case, or it will return false for $infix?(\Phi, \Phi)$. (One exception is Prolog, but this is a language specific feature, which we won't covered in this book.)

Since prefix testing is linear, and it is called while traversing the list, this algorithm is quadratic $O(N * M)$. where N and M are the length of the pattern and target lists respectively. There is no trivial way to improve this 'position by position' scanning algorithm to linear even if the data structure changes from linked-list to randomly accessible array.

There are chapters in this book introduce several approaches for fast matching, including suffix tree with Ukkonen algorithm, Knuth-Morris-Pratt algorithm and Boyer-Moore algorithm.

Alternatively, we can enumerate all suffixes of the target list, and check if the pattern is prefix of any these suffixes. Which can be represented as the following.

$$infix?(P, L) = \exists S \in suffixes(L) \wedge P \subseteq S \tag{A.84}$$

This can be represented as list comprehension, for example the below Haskell program.

```
isInfixOf x y = (not ∘ null) [ s | s ← tails(y), x `isPrefixOf` s]
```

Where function `isPrefixOf` is the prefixing testing function defined according to our previous design. function `tails` generate all suffixes of a list. The implementation of `tails` is left as an exercise to the reader.

Exercise A.5

- Implement the linear existence testing in both functional and imperative approaches in your favorite programming languages.
- Implement the looking up algorithm in your favorite imperative programming language.
- Realize the linear time filtering algorithm by firstly building the result list in reverse order, and finally reverse it to resume the normal result. Implement this algorithm in both imperative looping and functional tail-recursion call.
- Implement the imperative algorithm of prefix testing in your favorite programming language.
- Implement the algorithm to enumerate all suffixes of a list.

A.8 zipping and unzipping

It is quite common to construct a list of paired elements. For example, in the naive brute-force solution for 'Drunk jailer' puzzle which is shown in section of mapping, we need to represent the state of all lights. It is initialized as $\{(1,0),(2,0),...,(N,0)\}$. Another example is to build a key-value list, such as $\{(1,a),(2,an),(3,another),...\}$.

In 'Drunk jailer' example, the list of pairs is built like the following.

$$map(\lambda_i \cdot (i,0), \{1,2,...,N\})$$

The more general case is that, There have been already two lists prepared, what we need is a handy 'zipper' method.

$$zip(A,B) = \begin{cases} \Phi & : \quad A = \Phi \vee B = \Phi \\ cons((a_1,b_1), zip(A',B')) & : \quad otherwise \end{cases} \tag{A.85}$$

Note that this algorithm is capable to handle the case that the two lists being zipped have different lengths. The result list of pairs aligns with the shorter one. And it's even possible to zip an infinite list with another one with limited length in environment support lazy evaluation. For example with this auxiliary function defined, we can initialize the lights state as

$$zip(\{0,0,...\}, \{1,2,...,N\}$$

In some languages support list enumeration, such as Haskell (Python provides similar `range` function, but it manipulates built-in list, which isn't linked-list actually), this can be expressed as `zip (repeat 0) [1..n]`. Given a list of words, we can also index them with consecutive numbers as

$$zip(\{1,2,...\}, \{a,an,another,...\})$$

Note that the zipping algorithm is linear, as it uses constant time 'cons' operation in each recursive call. However, directly translating *zip* into imperative manner would down-grade the performance to quadratic unless the linked-list is optimized with tail position cache or we in-place modify one of the passed-in list.

function ZIP(A, B)
  $C \leftarrow \Phi$
  while $A \neq \Phi \wedge B \neq \Phi$ **do**
    $C \leftarrow$ APPEND(C, (FIRST(A), FIRST(B)))
    $A \leftarrow$ REST(A)
    $B \leftarrow$ REST(B)
  return C

Note that, the appending operation is proportion to the length of the result list C, so it will get more and more slowly along with traversing. There are three solutions to improve this algorithm to linear time. The first method is to use a similar approach as we did in infix-testing, that we construct the result list of pairs in reverse order by always insert the paired elements on head; then perform a linear reverse operation before return the final result; The second method is

to modify one passed-in list, for example A, in-place while traversing. Translate it from list of elements to list of pairs; The third method is to remember the last appending position. Please try these solutions as exercise.

The key point of linear time zipping is that the result list is actually built from right to left, which is similar to the infix-testing algorithm. So it's quite possible to provide a folding-right realization. This is left as exercise to the reader.

It is natural to extend the zipper algorithm so that multiple lists can be zipped to one list of multiple-elements. For example, Haskell standard library provides, `zip`, `zip3`, `zip4`, ..., till `zip7`. Another typical extension to zipper is that, sometimes, we don't want to list of pairs (or tuples more generally), instead, we want to apply some combinator function to each pair of elements.

For example, consider the case that we have a list of unit prices for every fruit: apple, orange, banana, ..., as $\{1.00, 0.80, 10.05, ...\}$, with same unit of Dollar; And the cart of customer holds a list of purchased quantity, for instance $\{3, 1, 0, ...\}$, means this customer, put 3 apples, an orange in the cart. He doesn't take any banana, so the quantity of banana is zero. We want to generate a list of cost for the customer, contains how much should pay for apple, orange, banana,... respectively.

The program can be written from scratch as below.

$$paylist(U, Q) = \begin{cases} \Phi & : \quad U = \Phi \vee Q = \Phi \\ cons(u_1 \times q_1, paylist(U', Q')) & : \quad otherwise \end{cases}$$

Compare this equation with the zipper algorithm. It is easy to find the common structure of the two, and we can parameterize the combinator function as f, so that the 'generic' zipper algorithm can be defined as the following.

$$zipWith(f, A, B) = \begin{cases} \Phi & : \quad A = \Phi \vee B = \Phi \\ cons(f(a_1, b_1), zipWith(f, A', B')) & : \quad otherwise \end{cases} \tag{A.86}$$

Here is an example that defines the inner-product (or dot-product)[14] by using $zipWith$.

$$A \cdot B = sum(zipWith(\times, A, B)) \tag{A.87}$$

It is necessary to realize the inverse operation of zipping, that converts a list of pairs, to different lists of elements. Back to the purchasing example, It is quite possible that the unit price information is stored in a association list like $U = \{(apple, 1.00), (orange, 0.80), (banana, 10.05), ...\}$, so that it's convenient to look up the price with a given product name, for instance, $lookup(melon, U)$. Similarly, the cart can also be represented clearly in such manner, for example, $Q = \{(apple, 3), (orange, 1), (banana, 0), ...\}$.

Given such a 'product - unit price' list and a 'product - quantity' list, how to calculate the total payment?

One straight forward idea derived from the previous solution is to extract the unit price list and the purchased quantity list, then calculate the inner-product of them.

$$pay = sum(zipWith(\times, snd(unzip(P)), snd(unzip(Q)))) \tag{A.88}$$

Although the definition of *unzip* can be directly written as the inverse of *zip*, here we give a realization based on folding-right.

$$unzip(L) = foldr(\lambda_{(a,b),(A,B)} \cdot (cons(a, A), cons(b, B)), (\Phi, \Phi), L) \quad (A.89)$$

The initial result is a pair of empty list. During the folding process, the head of the list, which is a pair of elements, as well as the intermediate result are passed to the combinator function. This combinator function is given as a lambda expression, that it extracts the paired elements, and put them in front of the two intermediate lists respectively. Note that we use implicit pattern matching to extract the elements from pairs. Alternatively this can be done by using fst, and snd functions explicitly as

$$\lambda_{p,P} \cdot (cons(fst(p), fst(P)), cons(snd(p), snd(P)))$$

The following Haskell example code implements *unzip* algorithm.

```
unzip = foldr λ(a, b) (as, bs) → (a:as, b:bs) ([], [])
```

Zip and unzip concepts can be extended more generally rather than only limiting within linked-list. It is quite useful to zip two lists to a tree, where the data stored in the tree are paired elements from both lists. General zip and unzip can also be used to track the traverse path of a collection to mimic the 'parent' pointer in imperative implementations. Please refer to the last chapter of [8] for a good treatment.

Exercise A.6

- Design and implement iota (I) algorithm, which can enumerate a list with some given parameters. For example:
  - $iota(..., N) = \{1, 2, 3, ..., N\}$;
  - $iota(M, N) = \{M, M + 1, M + 2, ..., N\}$, Where $M \leq N$;
  - $iota(M, M + a, ..., N) = \{M, M + a, M + 2a, ..., N\}$;
  - $iota(M, M, ...) = repeat(M) = \{M, M, M, ...\}$;
  - $iota(M, ...) = \{M, M + 1, M + 2, ...\}$.

  Note that the last two cases demand generate infinite list essentially. Consider how to represents infinite list? You may refer to the streaming and lazy evaluation materials such as [5] and [8].
- Design and implement a linear time imperative zipper algorithm.
- Realize the zipper algorithm with folding-right approach.
- For the purchase payment example, suppose the quantity association list only contains those items with the quantity isn't zero, that instead of a list of $Q = \{(apple, 3), (banana, 0), (orange, 1), ...\}$, it hold a list like $Q = \{(apple, 3), (orange, 1), ...\}$. The 'banana' information is filtered because the customer doesn't pick any bananas. Write a program, taking the unit-price association list, and this kind of quantity list, to calculate the total payment.

A.9 Notes and short summary

In this appendix, a quick introduction about how to build, manipulate, transfer, and searching singly linked list is briefed in both purely functional and imperative approaches. Most of the modern programming environments have been equipped with tools to handle such elementary data structures. However, such tools are designed for general purpose cases, Serious programming shouldn't take them as black-boxes.

Since linked-list is so critical that it builds the corner stones for almost all functional programming environments, just like the importance of array to imperative settings. We take this topic as an appendix to the book. It is quite OK that the reader starts with the first chapter about binary search tree, which is a kind of 'hello world' topic, and refers to this appendix when meets any unfamiliar list operations.