Chapter 1

Lists

Abstract In programming, a list is an abstract data type, which contains a list of elements, such as a list of shopping items Figure 1.1, a list of students in a class, and a to-do list. Cornerstones in functional programming are immutable values and

Fig. 1.1 A list of shopping items.

functions, and in the previous chapters, we have looked at many ways where this is sufficient for solving many problems. However, often data is on the form of a list in which equivalent expressions need to be calculated and possibly collected into a single value. For example, for the shopping list, we may want to estimate the total shopping price by 1) replacing each item on the list with a price, and 2) summing the elements. In functional programming, this can be done with the programming concept of map and fold, where map produces a new list as the elements of the original list with a function applied to them, and fold sequentially combines the values of the

price-list by iteratively adding the price to a subtotal. These are important examples of the usage of lists, but there is more. In this chapter, you will learn how to

- define lists
- manipulate lists using indexing and its properties
- use the list module including the map and fold higher-order functions.

After you have read this chapter, you will be able to model situations such as

- a class with lists of student records.
- a drawing consisting of a list of elements, such as a house, some trees, etc.

Lists are unions of immutable values of the same type. A list can be expressed as a *sequence expression*,

Listing 1.1: The syntax for a list using the sequence expression.

```
1 [[<expr>{; <expr>}]]
```

For example, `[1; 2; 3]` is a list of integers, `["This"; "is"; "a"; "list"]` is a list of strings, `[(fun x -> x); (fun x -> x*x)]` is a list of functions, and `[]` is the empty list. Lists may also be given as ranges,

Listing 1.2: The syntax for a list using the range expressions.

```
1 [<expr> .. <expr> [.. <expr>]]
```

where `<expr>` in *range expressions* must be of integers, floats, or characters. Examples are `[1 .. 5]`, `[-3.0 .. 2.0]`, and `['a' .. 'z']`. Range expressions may include a step size, thus, `[1 .. 2 .. 10]` evaluates to `[1; 3; 5; 7; 9]`.

F# implements lists as linked lists, as illustrated in Figure 1.2. A linked list is a data

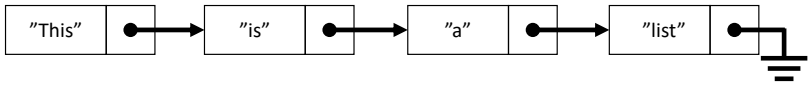

Fig. 1.2 A list is a linked list: Here is illustrated the linked list of `["This"; "is"; "a"; "list"]`.

structure consisting of a number of elements, where each element has an item of data and a pointer to the next element. For lists, every element can only have one other element pointing to it. The first element in a list is called its *head*, and the remainder of the list is called its *tail*. The last element in a list does not point to any other, and this is denoted by the *ground* symbol as shown in the figure.

A list type is identified with the `list` keyword, such that a list of integers has the type `int list`. Like strings, lists may be indexed and sliced using the *“[]”* notation, and the first element has index 0, and the last has the list's length minus one, see Listing 1.3 for examples. Note that if the index must be positive and less that than the length of the list. Otherwise an *out-of-bounds exception* is cast. See **??** for more details on exceptions. This is a very typical error, and it is adviced to **program in ways which completely avoids the possibility of out-of-bounds indexing, e.g., by using the List module.** An alternative to the “`[]`” indexing notation is the `List.tryItem` function to be described below on page 11, which does not cast exceptions but returns and option type. ⋆

Listing 1.3 listIndexing.fsx:
Lists are indexed as strings and has a `Length` property.

```
let lst = [3..9]
printfn "lst = %A, lst[1] = %A" lst lst[1]
printfn "First 2 elments of lst = %A" lst[..1]
printfn "Last 3 elements of lst = %A" lst[4..]
printfn "Element number 3 to 5 = %A" lst[2..4]
printfn "All elements = %A" lst[*]
```

```
$ dotnet fsi listIndexing.fsx
lst = [3; 4; 5; 6; 7; 8; 9], lst[1] = 4
First 2 elments of lst = [3; 4]
Last 3 elements of lst = [7; 8; 9]
Element number 3 to 5 = [5; 6; 7]
All elements = [3; 4; 5; 6; 7; 8; 9]
```

A list has a number of *properties*, which is summarized below:

`Head`: Returns the first element of a non-empty list.

Listing 1.4: `Head`

```
> [1; 2; 3].Head;;
val it: int = 1
```

Hence, given a non-empty list `lst`, then `lst.Head = lst[0]`.

`IsEmpty`: Returns true if the list is empty and false otherwise.

Listing 1.5: `IsEmpty`

```
> [1; 2; 3].IsEmpty;;
val it: bool = false
```

Hence, given a list `lst`, then `lst.IsEmpty` is the same as `lst = []`.

`Length`: Returns the number of elements in the list.

Listing 1.6: `Length`

```
> [1; 2; 3].Length;;
val it: int = 3
```

`Tail`: Returns the list, except for its first element. The list must be non-empty.

Listing 1.7: `Tail`

```
> [1; 2; 3].Tail;;
val it: int list = [2; 3]
```

Hence, given a non-empty list `lst`, then `lst.Tail=lst[1..]`.

A new list may be generated by concatenated two other lists using *concatenation* operator, "`@`". Alternatively, a new list may be generated by prepending an element using the *cons* operators, "`::`". This is demonstrated in Listing 1.8. Since lists

Listing 1.8: Examples of list concatenation.

```
> [1] @ [2; 3];;
val it: int list = [1; 2; 3]

> [1; 2] @ [3; 4];;
val it: int list = [1; 2; 3; 4]

> 1 :: [2; 3];;
val it: int list = [1; 2; 3]
```

are represented as linked lists, some operations on lists are slow and some are fast. Operations on data structures are often analyzed for their *computational complexity*, which is a worst-case and relative measure of their running time on a given piece of hardware. The notation is sometimes called *Big-O* notation or *asymptotic notation*. For example, the algorithm for calculating the length of a linked list starts at the head and follows the links until the end. For a list of length n, this takes n steps, and hence the computational complexity $O(n)$. Conversely, the cons operator is very efficient and has computational complexity $O(1)$, since we only need to link a single element to the head of a list. Another example is concatenation which has computational complexity $O(n)$, where n is the length of the first list. A final example, indexing an element i of a list of length n is also $O(n)$, since in the worst case, $i = n$, and the linked list must be traversed from the head to its tail.

Technically speaking, if the true running time as a function of the length of the list is $f(n)$, then its computational complexity is $O(g(n))$, if there exists a positive real number A and a real number $n_0 > 0$ such that for all $n \leq n_0$,

$$f(n) \leq Ag(n). \tag{1.1}$$

I.e., if the computational complexity is $O(n)$, then for sufficiently large n, the running time grows no faster than Mn for some constant M. This constant will differ per computer, but the asymptotic notation describes the relative increase in running time as we increase the list size.

Pattern matching in the `match–with` expression is possible with combination of the "`[]`" and "`::`" notations in a manner similar to indexing and prepending elements.

For example, recursively iterating over list is often done as illustrated in Listing 1.9. The `match–with` expression recognizes explict naming of elements such as 3-

Listing 1.9 listRecursive.fsx:
Using `match–with` to recursively print the elements of a list.

```
let rec printList (lst : int list) : unit =
  match lst with
    [] ->
      printfn ""
    | elm::rest ->
      printf "%A " elm
      printList rest

printList [3; 4; 5]
```

```
$ dotnet fsi listRecursive.fsx
3 4 5
```

element list `[s;t;u]`, and the cons-notation `head::tail`, where `head` is the first element of a non-empty list, and `tail` is the possibly empty tail. In particular, patterns for single element list can either be `head::[]`, `[head]`, or `s when s.Length = 1`.

It is possible to make multidimensional lists as lists of lists, as shown in Listing 1.10. The example shows a *ragged multidimensional list*, since each row has a different

Listing 1.10 listMultidimensional.fsx:
A ragged multidimensional list, built as lists of lists, and its indexing.

```
let a = [[1;2];[3;4;5]]
let row =  a[0]
let elm = a[0][1]
printfn "Fst row = %A, snd element in fst row = %A" row elm
```

```
$ dotnet fsi listMultidimensional.fsx
Fst row = [1; 2], snd element in fst row = 2
```

number of elements. This is also illustrated in Figure 1.3.

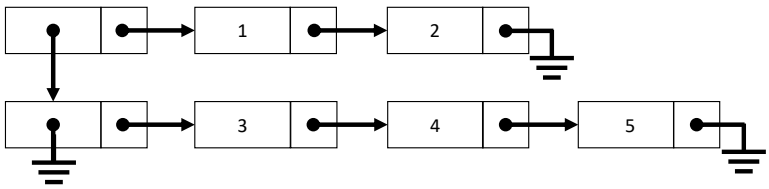

Fig. 1.3 A list is a ragged linked list: Here is illustrated the linked list of `[[1;2];[3;4;5]]`.

1.1 The List Module

Lists are so commonly used that a *module* is included with F#. A module is a library of functions and will be discussed in general in **??**. The list of functions in the list module is very long, and here, we briefly showcase some important ones. For the full list, see `https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-listmodule.html`. Below, the functions are grouped into 3: Simple functions, which take and/or give lists as argument and/or results; higher-order functions, which takes a function as an argument; and those that mimic the list's properties which can be useful in conjunction with the higher-order functions. Higher-order functions are discussed in detail in **??**.

Note that some arguments may result in an error, for example, when trying to find a non-existing element. For this, the list module has two types of functions: Those that return an option type, e.g., `None` if the element sought is not in the list. These functions all start with `try`. Similar functions exists with names excluding `try`, and they cast an exception, in case of an error. See **??** for more on exceptions. In both cases, the error has to be handled, and in this book, we favor option types.

Some often used simple functions in alphabetical order are:

`List.tryLast`: `'T list -> 'T option`.
Returns the last element of a list as an option type.

Listing 1.11: `List.tryLast`

```
> List.tryLast [1; -2; 0];;
val it: int option = Some 0
```

`List.rev`: `lst:'T list -> 'T list`.
Returns a new list with the same elements as in `lst` but in reversed order.

Listing 1.12: `List.rev`

```
> List.rev [1; 2; 3];;
val it: int list = [3; 2; 1]
```

`List.sort`: `lst:'T list -> 'T list`.
Returns a new list with the same elements as in `lst` but where the elements are sorted.

Listing 1.13: `List.sort`

```
> List.sort [3; 1; 2];;
val it: int list = [1; 2; 3]
```

`List.unzip`: `lst:('T1 * 'T2) list -> 'T1 list * 'T2 list`.
Returns a pair of lists of all the first elements and all the second elements of `lst`, respectively.

Listing 1.14: `List.unzip`

```
> List.unzip [(1, 'a'); (2, 'b'); (3, 'c')];;
val it: int list * char list = ([1; 2; 3], ['a'; 'b';
   'c'])
```

There exists an equivalent function `List.unzip3`, which separates elements from lists of triples.

`List.zip`: `lst1:'T1 list -> lst2:'T2 list -> ('T1 * 'T2) list`.
Returns a list of pairs, where elements in `lst1` and `lst2` are iteratively paired.

Listing 1.15: `List.zip`

```
> List.zip [1; 2; 3] ['a'; 'b'; 'c'];;
val it: (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
```

There exists an equivalent function `List.zip3`, which combines elements form three lists.

Some programming patterns on lists involve performing calculations and combining list elements, and they are so common that they have been standardized. Examples of this are the higher-order functions from the module, given below. As is common, the examples are given with anonymous functions, see page **??** in **??**.

`List.exists`: `f:('T -> bool) -> lst:'T list -> bool`.
Returns true if `f` is true for some element in `lst` and otherwise false.

Listing 1.16: `List.exists`

```
> List.exists (fun x -> x % 2 = 1) [0 .. 2 .. 4];;
val it: bool = false
```

`List.filter`: `f:('T -> bool) -> lst:'T list -> 'T list`.
Returns a new list with all the elements of `lst` for which `f` evaluates to true.

Listing 1.17: `List.filter`

```
> List.filter (fun x -> x % 2 = 1) [0 .. 9];;
val it: int list = [1; 3; 5; 7; 9]
```

`List.fold`: `f:('S -> 'T -> 'S) -> acc:'S -> lst:'T list -> 'S`.
Updates an accumulator iteratively by applying `f` to each element in `lst`. The initial value of the accumulator is `elm`. For example, when `lst` consists of `n` elements `List.fold` calculates:

`f (...(f (f elm lst[0]) lst[1])...) lst[n-1]`.

Listing 1.18: `List.fold`

```
> let addSquares acc elm = acc + elm*elm
List.fold addSquares 0 [0 .. 9];;
val addSquares: acc: int -> elm: int -> int
val it: int = 285
```

There exists an equivalent function `List.fold2`, which iterates through two lists simultaneously.

`List.foldBack`: `f:('T -> 'S -> 'S) -> lst:'T list -> acc:'S -> 'S`.
Updates an accumulator iteratively backwards by applying `f` to each element in `lst`. The initial value of the accumulator is `elm`. For example, when `lst` consists of `n` elements `List.foldBack` calculates:

`f lst[0] (f lst[1] (...(f lst[n-1] elm)...))`.

Listing 1.19: `List.foldBack`

```
> let addSquares elm acc = acc + elm*elm
List.foldBack addSquares [0 .. 9] 0;;
val addSquares: elm: int -> acc: int -> int
val it: int = 285
```

There exists an equivalent function `List.foldBack2`, which iterates through two lists simultaneously.

`List.forall`: `f:('T -> bool) -> lst:'T list -> bool`.
Returns true if all elements in `lst` are true when `f` is applied to them.

Listing 1.20: `List.forall`

```
> List.forall (fun x -> x % 2 = 1) [0 .. 9];;
val it: bool = false
```

There exists an equivalent function `List.forall2`, which iterates through two lists simultaneously.

`List.init`: `m:int -> f:(int -> 'T) -> 'T list`.
Create a list with `m` elements, and whose value is the result of applying `f` to the index of the element.

Listing 1.21: `List.init`

```
> List.init 10 (fun i -> i * i);;
val it: int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]
```

`List.iter`: `f:('T -> unit) -> lst:'T list -> unit`.
Applies `f` to every element in `lst`.

Listing 1.22: `List.iter`

```
> List.iter (fun x -> printfn "%A " x) [0; 1; 2];;
0
1
2
val it: unit = ()
```

There exists an equivalent function `List.iter2`, which iterates through two lists simultaneously, and `List.iteri` and `List.iteri2`, which also receives the index while iterating.

`List.map`: `f:('T -> 'U) -> lst:'T list -> 'U list`.
Returns a list as a concatenation of applying `f` to every element of `lst`.

Listing 1.23: `List.map`

```
> List.map (fun x -> x*x) [0 .. 9];;
val it: int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]
```

There exist equivalent functions `List.map2` and `List.map3`, which iterates through two and three lists simultaneously, and `List.mapi` and `List.mapi2`, which also receives the index while iterating.

`List.tryFind`: `f:('T -> bool) -> lst:'T list -> 'T option`.
Returns the first element of `lst` for which `f` is true as an option type.

Listing 1.24: `List.tryFind`

```
> List.tryFind (fun x -> x % 2 = 1) [0 .. 2 .. 9];;
val it: int option = None
```

`List.tryFindIndex`: `f:('T -> bool) -> lst:'T list -> int option`.
Returns the index of the first element of `lst` for which `f` is true as an option type.

Listing 1.25: `List.tryFindIndex`

```
> List.findIndex (fun x -> x = 'k') ['a' .. 'z'];;
val it: int = 10
```

At times, e.g., in conjunction with the higher-order functions given above, it is useful to have the list operators and properties on function form. These are available in the list module as:

`List.concat`: `lstLst:'T list list -> 'T list`.
Concatenates a list of lists.

Listing 1.26: `List.concat`

```
> List.concat [[1; 2]; [3; 4]; [5; 6]];;
val it: int list = [1; 2; 3; 4; 5; 6]
```

`List.isEmpty`: `lst:'T list -> bool`.
Returns true if `lst` is empty.

Listing 1.27: `List.isEmpty`

```
> List.isEmpty [1; 2; 3];;
val it: bool = false
```

`List.length`: `lst:'T list -> int`.
Returns the length of the list.

Listing 1.28: `List.length`

```
> List.length [1; 2; 3];;
val it: int = 3
```

`List.tryHead`: `lst:'T list -> 'T option`.
Returns the first element in `lst` as an option type. .

Listing 1.29: `List.tryHead`

```
> List.tryHead [1; -2; 0];;
val it: int option = Some 1
```

`List.tryItem`: `i:int -> lst:'T list -> 'T option`.
Returns the *i*'th element of a list as an option type.

Listing 1.30: `List.tryItem`

```
> List.tryItem 10 [0..3];;
val it: int option = None
```

E.g., given a non-empty list `lst`, `Some lst.Head = List.tryHead lst` and `Some lst[2] = List.tryItem 2 lst`. Note, the module does not contain an equivalent of `lst.Tail`.

1.2 Programming Intermezzo: Word Statistics

Natural language processing is the field of studying the natural language. This is a vast field and has seen considerable breakthroughs in our understanding of how humans communicate through writing. A common view of texts is as lists of words, so let us consider one of the most basic questions, the study of natural language may ask:

Problem 1.1

What is the longest word in Hans Christian Andersen's fairy tale "The emperor's new clothes"?

The fairy tale, as published online on `https://www.gutenberg.org` in English contains 1885 words. For the sake of demonstration, we will just consider the first seven words,

> "Many years ago, there was an Emperor,. . . "

and ignore punctuation. In such a small example, we quickly realize, that the answer should be "Emperor" which consists of 7 characters, but if we were to analyze the whole text, then the answer would not be as readily found by eye. Thus, we will write a program. To begin, we create a list of the relevant words. In **??**, we will discuss, how to read from a file, but here, we will enter them by hand:

```
let wLst = ["Many";"years";"ago";"there";"was";"an";"Emperor"]
```

Strings have the `Length` property, which if we can read it of each string in the list, then we can decide, which is the longest. The `List.map` function allows us to make a new list as a copy of the old, but where each element `elm` is result of `f elm` for some function `f`. Thus, we make an anonymous function `fun w -> (w,w.Length)` and apply it to every element by `List.map`:

```
let wLen = List.map (fun (w: string) -> (w,w.Length)) wLst
```

Note that we chose to make a list of pairs, such that the word and its length are joined into a single data structure to safeguard us from possibly future mix-ups of words and lengths. What remains is to find the longest. For this, we must traverse, and while doing so, we must maintain an accumulator containing the longest word, we have seen so far. This is a `List.fold` operation. `List.fold` traverses from the head and applies a function to update the accumulator given the present state of the accumulator and the next element. An function for this is `maxWLen`,

```
let maxWLen acc elm = if snd acc > snd elm then acc else elm
```

The initial value of the accumulator must be sensible, in the case that the list is empty, and which we are sure will be disregarded as soon as it is compared to any word. Here, such a value is `("",0)`. Finally, we are ready to call `List.fold`:

```
let longest = List.fold maxWLen ("",0) wLen
```

Thus, the problem is solved with the program shown in Listing 1.31.

Listing 1.31 longestWord.fsx:
Using the list module to find the longest word in an H.C. Andersen fairy tale.

```
let wLst = ["Many";"years";"ago";"there";"was";"an";"Emperor"]
let wLen = List.map (fun (w: string) -> (w,w.Length)) wLst
let maxWLen acc elm = if snd acc > snd elm then acc else elm
let longest = List.fold maxWLen ("",0) wLen
printfn "The longest word is %A" longest
```

```
$ dotnet fsi longestWord.fsx
The longest word is ("Emperor", 7)
```

1.3 Key concepts and terms in this chapter

In this chapter, you have read about

- How to create lists with the "`[]`" notation
- That lists are implemented as linked elements which implies that prepending is fast, but concatenation and indexing are slow.
- The list properties such as `Head`, `Tail`, and `Length`
- The list module with important higher-order functions such as `List.map` and `List.fold`.