



Lists

- Homogeneous
 - [1,2,3] :: [Int]
- Dynamic: length may change during execution.
- [n..m] is the list [n, n+1, ..., m]: if n exceeds m, the list is empty.

Hugs > [2..7]

Lists

Homogeneous

[1,2,3] :: [Int]

- Dynamic: length may change during execution.
- [n..m] is the list [n, n+1, ..., m]: if n exceeds m, the list is empty.

Hugs > [2..7][2,3,4,5,6,7]

List Operations

- 1: [2,3,4] = [1,2,3,4]
- 4: [] = [4]
- 1:[2,3,4] = 1:2:[3,4] = 1:2:3:[4] = 1:2:3:4:[] = [1,2,3,4]
- head [1,2,3,4] = 1
- last [1,2,3,4] = 4
- tail [1,2,3,4] = [2,3,4]

More List Operations in Haskell

- [1,2] ++ [3,4] = [1,2,3,4]
- length [1,2,3,4] = 4
- init [1,2,3,4] = [1,2,3]Return all the elements of a list except the last one. The list must be finite and non-empty.



Finding Primitive Recursive Definitions

 A template for a primitive recursive definition over lists is:

```
fun [] = ... (base case)
fun (x:xs) = ... fun xs (recursive case)
```

 Question: given the value fun xs, how to define fun (x:xs)

CCE71 Programming Language



Polymorphism

- length: take a list as input, return the length of the list.
 - * Can be applied to any type of list.

```
length:: [a] -> Int
```

where a is a type variable.

Main> length [1,2,3,4]

4

length [] = 0

length(x:xs) = 1 + length xs

00574 D------

1

Example: sum

- Sum of all integers of a list
 - sum :: [Int] -> Int
 - * Base case: the value of sum at []
 - * A way of going from the value of sum xs to the value of sum (x:xs)

CS571 Programming Languages



Example: sum

- Sum of all integers of a list
 - sum :: [Int] -> Int
 - * Base case: the value of sum at [] sum [] = 0
 - * A way of going from the value of $\underset{\text{sum }xs}{\text{s}}$ to the value of $\underset{\text{sum }(x:xs)}{\text{s}}$

sum(x:xs) = x + sum xs

Languages



Example: sum

- Sum of all integers of a list
 - $sum :: [Int] -\!\!> Int$
 - * Base case: the value of sum at []
 - sum[] = 0
 - * A way of going from the value of sum xs to the value of sum (x:xs)

sum (x:xs) = x + sum xs

```
sum [3,2,7,5] = 3 + sum [2,7,5]
= 3 + (2 + sum [7,5])
= 3 + (2 + (7 + sum [5]))
= 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum []))) = 3 + (2 + (7 + (5 + sum [])))
```

= 3 + (2 + (7 + (5 + 0))) = 17

CCE71 Decomposing Languages



Examples: reverse

Reverse a list

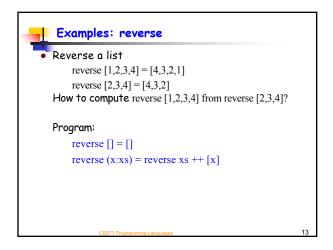
reverse [1,2,3,4] = [4,3,2,1]

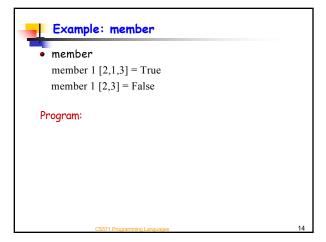
reverse [2,3,4] = [4,3,2]

How to compute reverse [1,2,3,4] from reverse [2,3,4]?

Program:

CS571 Programming Languages

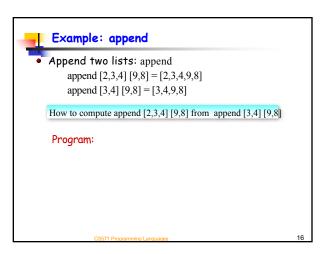




Example: member

• member
member 1 [2,1,3] = True
member 1 [2,3] = False

Program:
member a [] = False
member a (b:xs)
| a = b = True
| otherwise = member a xs



Example: append

• Append two lists: append
append [2,3,4] [9,8] = [2,3,4,9,8]
append [3,4] [9,8] = [3,4,9,8]

How to compute append [2,3,4] [9,8] from append [3,4] [9,8]

Program:
append :: [a] → [a] → [a]
append [] ys = ys
append (x:xs) ys = x: append xs ys

Wild-cards

• Wild-cards: a wild-card will match anything and is used when we don't care what a certain part of the input is head (x:_) = x tail (_:xs) = xs



Example: null

 null: if a list is empty, return true; otherwise return false

```
null [] = True
null [1,2,3,4] = False
```

Program:

CCE71 Decembring Language

Example: null

 null: if a list is empty, return true; otherwise return false

```
null [] = True
null [1,2,3,4] = False
```

Program:

```
null :: [a] -> Bool
null [] = True
null (_:_) = False
```

CSE71 Decareommica Longuage

20



Example: delete

 delete x list: Removes the first occurrence of x from list delete 1 [2,1,3,1,4] = [2,3,1,4]

Program:

-

Example: delete

• delete x list: Removes the first occurrence of x from list

delete 1 [2,1,3,1,4] = [2,3,1,4]

Program

```
delete x [] = []
delete x (y:ys)
|x==y=ys
|otherwise = y: delete x ys
```

CS571 Programming Languages

22



Example: delete

 delete x list: Removes all occurrences of x from list delete 1 [2,1,3,1,4] = [2,3,4]

Program

CCE71 Decomposing Language



Example: delete

• delete x list: Removes all occurrences of x from list delete 1 [2,1,3,1,4] = [2,3,4]

Program:

```
delete x [] = []
delete x (y:ys)
|x==y = delete x ys
|otherwise = y: delete x ys
```

CS571 Programming Languages

24



Example: myLast

 Write a haskell function myLast list that finds the last element of a list list. Assume that the list is not empty (do not use the build-in function last).
 E.g. mylast [1,2,3,4] = 4

-

Example: myLast

 Write a haskell function myLast list that finds the last element of a list list. Assume that the list is not empty (do not use the build-in function last).
 E.g. mylast [1,2,3,4] = 4

```
myLast :: [a] -> a
myLast [x] = x
myLast (_:xs) = myLast xs
```

00574 P------

- - -



Example: myLastButOne

 Write a haskell function myLastButOne that finds the last but one element of a list. Assume that the list has at least 2 elements.
 E.g. mylastButOne [1,2,3,4] = 3

CS571 Programming Languages

Example: myLastButOne

 Write a haskell function myLastButOne that finds the last but one element of a list. Assume that the list has at least 2 elements.

```
E.g. mylastButOne [1,2,3,4] = 3

myLastButOne :: [a] -> a

myLastButOne [x,_] = x

myLastButOne (_:xs) = myLastButOne xs

Or
```

myLastButOne (x:[_]) = x myLastButOne (_:xs) = myLastButOne xs

CS571 Programming Languages

28



Example: findk

 Write a haskell function findk list k that finds the kth element of a list list. Assume that the size of the list is greater than k
 E.g. findk [1,2,3,4] 2 = 2

CCE71 Dengenoming Language



Example: findk

 Write a haskell function findk list k that finds the kth element of a list list. Assume that the size of the list is greater than k

```
E.g. findk [1,2,3,4] 2 = 2
```

```
findk :: [a] \rightarrow Int \rightarrow a
findk (x:_) 1= x
findk (_:xs) k = findk xs (k-1)
```

CS571 Programming Language

30



Example: deletek

 Write a haskell function deletek list k that remove the kth element of a list list. Assume that the size of the list is greater than k
 E.g. deletek [1,2,3,4] 2 = [1,3,4]

Example: deletek

Write a haskell function deletek list k that remove the kth element of a list list. Assume that the size of the list is greater than k

```
E.g. deletek [1,2,3,4] 2 = [1,3,4]
```

```
deletek :: [a] -> Int -> [a]
deletek (_:xs) 1=xs
deletek (x:xs) k=x: deletek xs (k-1)
```

00574 D------

. . .



Example: removedup

• Write a Haskell function removedup It that removes the duplicate elements from a list It E.g. >removedup [1,2,3,2,4]
[1,3,2,4]

CS571 Programming Languages



Example: removedup

 Write a Haskell function removedup It that removes the duplicate elements from a list It E.g. >removedup [1,2,3,2,4]

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

removedups [] = []

removedups (x:xs)

|elem x xs = removedups xs
```

CS571 Programming Languages

|otherwise = x:removedups xs

34



oddelem

• Write a haskell function oddelem It that returns all element occurring at odd position of a list It > oddelem [4,5,6,7,8,9] [4,6,8]

CoE74 Decaronming Longues



oddelem

• Write a haskell function oddelem It that returns all element occurring at odd position of a list It > oddelem [4,5,6,7,8,9]

```
[4,6,8]
oddelem [] = []
oddelem [x] = [x]
oddelem (x:y:ys) = x:oddelem ys
```

Cs571 Programming Languages

36