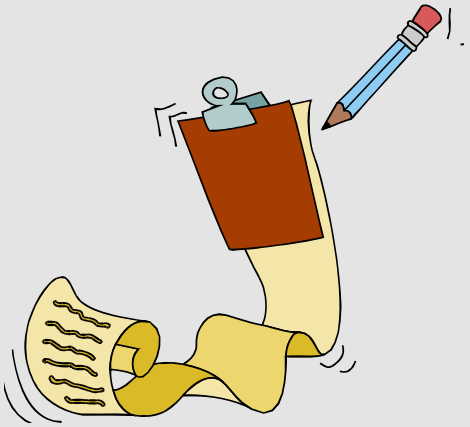
List



List in Haskell
Cons
Functions on lists
Tuples versus lists

List type

```
[False, True, False]        :: [Bool]
[‘a’,`e’,`i’,`o’,`u’]       :: [Char]
["one", "two", "three"]     ::[String]
[[‘a’,`b’],[`c’,`d’,`e’]]   :: [[Char]]

[(‘a’, False),(’b’, True)] ::[(Char, Bool)]
([1,2,3,4], [‘a’,’b’,’c’]) ::([Int],[Char])
[sum, product]              ::[Num a=>[a]->a]
```

`[T]` list of type `T`

Empty list , singleton

Empty list: `[]`

Singleton: `[1], ["Today"], [[]]`

Remarks:

a) No restriction of the type of elements in a list
b) No restriction on the length of a list-*infinite*
c) The type of elements in a list does not say anything about its length

Order and number of elements are important

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

[] ≠ [[]]

```
Prelude> [1,2,3]<[1,2,4]
True
Prelude> [1,2,3,6]<[1,2,3]
False
Prelude> [1,2,3,6]<[1,5,3]
True
```

Prelude functions on lists

```
Prelude> head [5,7,6,9,1]
5

Prelude> tail [5,7,6,9,1]
[7,6,9,1]

Prelude> [5,7,6,9,1]!! 2
6

Prelude> take 3 [5,7,6,9,1]
[5,7,6]

Prelude> drop 3 [5,7,6,9,1]
[9,1]
```

Prelude functions on lists

```
Prelude> length[5,7,6,9,1]
5

Prelude> sum [5,7,6,9,1]
28

Prelude> product [5,7,6,9,1]
1890

Prelude> [5,7,6,9,1]++ [4,23]
[5,7,6,9,1,4,23]

Prelude> reverse [5,7,6,9,1]
[1,9,6,7,5]
```

Length function

```
Prelude> length [1,2,3,5,8]
5
Prelude> length [“yes”, “no”]
2
Prelude> length [(‘a’, “First”)]
1
```

Polymorphic function

```
length [a]->Int

head:: [a]-> a
take:: Int ->[a]->[a]
```

Simple functions on list

```
take (2^2) [9,8,7,6,5,4,3]   = [9,8,7,6]

factorial n = product [1..n]

average ns = (sum ns) `div` (length ns)

splitAt :: Int ->[a] ->([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

What does this do?

```
test:: [Char] -> Bool
test ['a',_,_] = True
test _         = False
```

Writing functions on lists

Simple functions on lists: `fact list= product list`

Recursive functions on lists:

```
sum :: [Int]->Int
sum [] = 0
sum (head:tail) = head + sum tail
```

Compare with:

```
sum ::Int->Int
sum 0 = 0
sum n = n+ sum (n-1)
```

Steps: 1.Define the type
2. Define the base case
3. Define the recursive case
4. Make sure the base case is reached.

Example

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

What is the type of sum ?

```
Num a=> [a]-> a
```

How is this evaluated?

```
sum [1,2,3] =
       applying sum
       = 1+sum[2,3]=
      applying sum
       = 1+ (2 + sum[3])=
       applying sum
       = 1+ (2 + 3+ sum[])=
        applying sum
      = 1+ (2 + (3+ sum[]))=
       applying sum
       = 1+ (2+(3+0))=
       applying +
       = 6
```

Example

Reversing a list

```
rev ::[a]->[a]

rev::[Int]->[Int]
rev [] =[]
rev (x:xs)= rev xs ++ [x]
```

cons

Lists are not primitive notions.
They are **constructed** one element at a time, starting from empty list using the “cons” operator.

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

```
1: 2:3 :[] is 1:(2:(3:[]))
```

Important--when using cons:
a) The elements of the list must have the same type.
b) You can only cons (:) something onto a list, not the other way around (you cannot cons a list onto an element). So, the final item on the right must be a list, and the items on the left must be independent elements, not lists.

cons

cons used to construct patterns:

```
test ::[Char]-> Bool
test ('a':_)= True
test _       = False
```

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

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

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

Quick quiz

1. Which of these are valid Haskell and which are not? Rewrite in cons notation.

a) `[1,2,3,[]]`
b) `[1,[2,3],4]`
c) `[[1,2,3],[]]`

2. Which of these are valid Haskell, and which are not? Rewrite in comma and bracket notation.

a) `[]:[[1,2,3],[4,5,6]]`
b) `[]:[]`
c) `[]:[]:[]`
d) `[1]:[]:[]`
e) `["hi"]:[1]:[]`

3. Why is the following list invalid in Haskell?

`[[1,2],3,[4,5]]`

Tuples

Examples:

```
(True, 1)
("Hello world", False)
(4, 5, "Six", True, 'b')
```

- Tuples offer another way of storing multiple values in a single value.

- Tuples and lists have different characteristics::
  - Tuples have a fixed number of elements (immutable); you can't cons to a tuple.
  - The elements of a tuple do not need to be all of the same type.
  - Tuples are marked by parentheses with elements delimited by commas.

Tuples

```
Prelude> :t ()
() :: ()
```

It's called a **unit** type:

The unit type is like the Null construct in other languages.

One element tuples are not allowed.

We use n-tuple to denote a tuple of size n. Commonly, we call 2-Tuples pairs and 3-tuples triples.
Tuples of greater sizes aren't actually all
that common, but we can logically extend the naming system to quadruples, quintuples, and so on.

The type of a tuple

The type of a tuple is defined by its size and by the types of objects it contains.

Examples:

```
("Hello",32) :: (String,Int)
and (47,"World") :: (Int,String)
```

This has implications for building up lists of tuples.

`[("a",1),("b",9),("c",9)]` this is a valid list

`[("a",1), (2,"b"),(9,"c")]` this is not a valid list

Tuples and lists

Nesting tuples and lists:

```
((2,3), True)
((2,3), [2,3])
[(1,2), (3,4), (5,6)]
```

Lists can be built by consing new elements onto them. Cons a number onto a list of numbers, you will get back a list of numbers. There is no such way to build up tuples.

Why do you think that is?

Quiz

Which of these are valid Haskell, and why?

```
a)  1:(2,3)
b)  (2,4):(2,3)
c)  (2,4):[]
d)  [(2,4),(5,5),('a','b')]
e)  ([2,4],[2,2])
```

Tuples and functions

Tuples can be used to return more than one value from a function.

```
addsub x y = (x + y, x - y)
```

Comparing tuples

```
Prelude> ( 2,3,8) <(1,2,6)
False
Prelude> ( 2,3,8) <(2, 3,9)
True
Prelude> ( 'x', 3, 12.78) <(2, 's',9)

<interactive>:10:8:
    No instance for (Num Char) arising from the literal ‘3’
    In the expression: 3
    In the first argument of ‘(<)’, namely ‘('x', 3, 12.78)’
    In the expression: ('x', 3, 12.78) < (2, 's', 9)
Prelude> ( 'x', 3, 12.78) <('z', 5,9)
True
Prelude> ((2,3),"AbC",45) <= ((2.0,3), "abC",23)
True
Prelude> (9,0)>=(8,12)
True
Prelude>
```

The magic of Haskell patterns

```
fst :: (a,b) -> a

fst (x,_) = x
```

So simple and so powerful!

Extracting an element from a quadruple:

```
get3 (_, _, element, _) = element
```