For example

```
tup1 :: ???
tup1 = ('a', 5)

tup2 :: (Char, Double, Int)
tup2 = ('a', 5.2, 7)
```

QUIZ

What is the type `???` of `tup3`?

```
tup3 :: ???
tup3 = ((7, 5.2), True)
```

A. `(Int, Bool)`

B. `(Int, Double, Bool)`

C. `(Int, (Double, Bool))`

D. `((Int, Double), Bool)`

E. `(Tuple, Bool)`

Extracting Values from Tuples

We can **create** a tuple of three values `e1` , `e2` , and `e3` ...

```
tup = (e1, e2, e3)
```

... but how to **extract** the values from this tuple?

Pattern Matching

```
fst3 :: (t1, t2, t3) -> t1
fst3 (x1, x2, x3) = x1

snd3 :: (t1, t2, t3) -> t2
snd3 (x1, x2, x3) = x2

thd3 :: (t1, t2, t3) -> t3
thd3 (x1, x2, x3) = x3
```

QUIZ

What is the value of `quiz` defined as

```
tup2 :: (Char, Double, Int)
tup2 = ('a', 5.2, 7)

snd3 :: (t1, t2, t3) -> t2
snd3 (x1, x2, x3) = x2

quiz = snd3 tup2
```

A. `'a'`

B. `5.2`

C. `7`

D. `('a', 5.2)`

E. `(5.2, 7)`

Lists

Unbounded Sequence of values of type `T`

```
[T]
```

For example

```
chars :: [Char]
chars = ['a','b','c']

ints :: [Int]
ints = [1,3,5,7]

pairs :: [(Int, Bool)]
pairs = [(1,True),(2,False)]
```

QUIZ

What is the type of `things` defined as

```
things :: ???
things = [ [1], [2, 3], [4, 5, 6] ]
```

A. `[Int]`

B. `([Int], [Int], [Int])`

C. `[(Int, Int, Int)]`

D. `[[Int]]`

E. `List`

List's Values Must Have The SAME Type!

The type `[T]` denotes an unbounded sequence of values of type `T`

Suppose you have a list

```
oops = [1, 2, 'c']
```

There is no `T` that we can use

- As last element is not `Int`
- First two elements are not `Char` !

Result: Mysterious Type Error!

Constructing Lists

There are two ways to construct lists

```
[]      -- creates an empty list
h:t     -- creates a list with "head" 'h' and "tail" t
```

For example

```
>>> 3 : []
[3]

>>> 2 : (3 : [])
[2, 3]

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

Cons Operator `:` is Right Associative

`x1 : x2 : x3 : x4 : t` means `x1 : (x2 : (x3 : (x4 : t)))`

So we can just avoid the parentheses.

Syntactic Sugar

Haskell lets you write `[x1, x2, x3, x4]` instead of `x1 : x2 : x3 : x4`

```
: []
```

Functions Producing Lists

Lets write a function `copy3` that

- takes an input `x` and
- returns a list with *three* copies of `x`

```
copy3 :: ???
copy3 x = ???
```

When you are done, you should see the following

```
>>> copy3 5
[5, 5, 5]

>>> copy3 "cat"
["cat", "cat", "cat"]
```

PRACTICE: Clone

Write a function `clone` such that `clone n x` returns a list with `n` copies of `x`.

```
clone :: ???
clone n x = ???
```

When you are done you should see the following behavior

```
>>> clone 0 "cat"
[]

>>> clone 1 "cat"
["cat"]

>>> clone 2 "cat"
["cat", "cat"]

>>> clone 3 "cat"
["cat", "cat", "cat"]

>>> clone 3 100
[100, 100, 100]
```

How does `clone` execute?

(Substituting equals-by-equals!)

```
clone 3 100
  =*> ???
```

EXERCISE: Range

Write a function `range` such that `range i j` returns the list of values `[i, i+1, ..., j]`

```
range :: ???
range i j = ???
```

When we are done you should get the behavior

```
>>> range 4 3
[]

>>> range 3 3
[3]

>>> range 2 3
[2, 3]

>>> range 1 3
[1, 2, 3]

>>> range 0 3
[0, 1, 2, 3]
```

Functions Consuming Lists

So far: how to *produce* lists.

Next how to *consume* lists!

Example

Lets write a function `firstElem` such that `firstElem xs` returns the *first* element `xs` if it is a non-empty list, and `0` otherwise.

```
firstElem :: [Int] -> Int
firstElem xs = ???
```

When you are done you should see the following behavior:

```
>>> firstElem []
0

>>> firstElem [10, 20, 30]
10

>>> firstElem [5, 6, 7, 8]
5
```

QUIZ

Suppose we have the following `mystery` function

```
mystery :: [a] -> Int
mystery []     = 0
mystery (x:xs) = 1 + mystery xs
```

What does `mystery [10, 20, 30]` evaluate to?

A. `10`

B. `20`

C. `30`

D. `3`

E. `0`

EXERCISE: Summing a List

Write a function `sumList` such that `sumList [x1, ..., xn]` returns `x1 + ... + xn`

```
sumList :: [Int] -> Int
sumList = ???
```

When you are done you should get the following behavior:

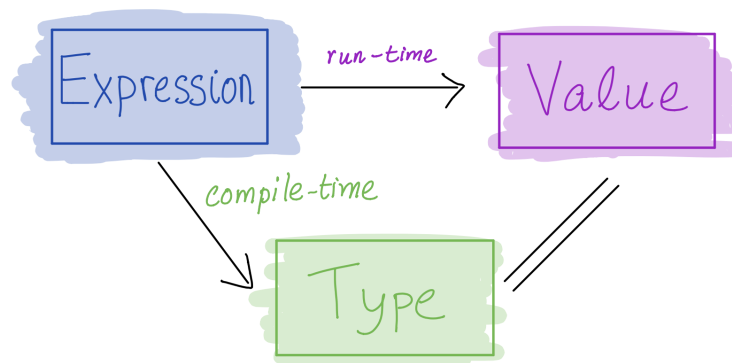
```
>>> sumList []
0

>>> sumlist [3]
3

>>> sumlist [2, 3]
5

>>> sumlist [1, 2, 3]
6
```

Recap



- Core program element is an **expression**
- Every *valid* expression has a **type** (determined at compile-time)
- Every *valid* expression reduces to a *value* (computed at run-time)

Execution

- Basic values & operators
- Execution / Function Calls just *substitute equals by equals*
- Pack data into *tuples* & *lists*

- Unpack data via *pattern-matching*

(https://ucsd-cse230.github.io/fa20/feed.xml)
(https://twitter.com/ranjitjhala)
(https://plus.google.com/u/0/104385825850161331469)
(https://github.com/ranjitjhala)

Generated by Hakyll (http://jaspervdj.be/hakyll), template by Armin Ronacher (http://lucumr.pocoo.org), suggest improvements here (https://github.com/ucsd-progsys/liquidhaskell-blog/).