Tuples & Lists

- Tuples have almost the same syntax as Rust.

```
> joe = (32,185,”married”,”pilot”)
> :type joe
joe :: (Num a, Num b) => (a, b, [Char], [Char])
> (age, height, _, _) = joe
> age
=> 32
> :type age
age :: Num a => a
```

Type

Pattern Matching

Tuples & Lists

Repl.it - Intro

repl.it/@lutzhamel/Intro#main.hs

lutzhamel / Intro

Run ▶

Upgrade Share +

Files

main.hs
area.hs
curry.hs
fact.hs
intinc.hs
rect.hs

https://Intro.lutzhamel.repl.run

```
GHCi, version 8.6.5
ghci> joe = (32,185,"married","pilot")
ghci>
ghci> :type joe
joe :: (Num a, Num b) => (a, b, [Char], [Char])
ghci>
ghci> (age, height, _, _) = joe
ghci> age
=> 32
ghci> :type age
age :: Num a => a
=> 32
ghci>
```

Tuples & Lists

- Nested tuples – say we want to specify a circle with a center and a radius…

```
> circle = ((2.5,3.6),5.0)
> circle
=> ((2.5,3.6),5.0)
> :type circle
circle  :: (Fractional a, Fractional b1, Fractional b2) => ((a, b1), b2)
=> ((2.5,3.6),5.0)
> center, radius) = circle
➢  center
=> (2.5,3.6)
> radius=> 5.0
```

Tuples & Lists

Repl.it - Intro

repl.it/@lutzhamel/Intro#main.hs

lutzhamel / Intro

Run ▶ Upgrade Share +

Files

- main.hs
- area.hs
- curry.hs
- fact.hs
- intinc.hs
- rect.hs

https://Intro.lutzhamel.repl.run

```
GHCi, version 8.6.5
λ> circle = ((2.5,3.6),5.0)
λ> circle
=> ((2.5,3.6),5.0)
λ> :type circle
circle
  :: (Fractional a, Fractional b1, Fractional b2) => ((a, b1), b2)
=> ((2.5,3.6),5.0)
λ> (center, radius) = circle
λ> center
=> (2.5,3.6)
λ> radius
=> 5.0
λ>
```

Tuples & Lists

- In a list all elements are of the <u>same type</u>

```
> oddlist = [ 1, 3, 5, 7, 9 ]
> :type oddlist
oddlist :: Num a => [a]
>
```

different from tuples!

nested = [(1,2),(3,4)]
nested = [[1,2],[3,4]]
nested = [[1,2],[3,4,5]]
nested = [(1,2),(3,4,5)]

what is the type of these constructions?

Tuples & Lists

- There exists a special list → the <u>empty</u> list: [ ]

```
> mylist = []
> :type mylist
mylist :: [a]
>
```

Polymorphic type!

List Operators

- **null** – tests whether a list is empty
  - null :: [a] -> Bool

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

List Operators

- ++ - concatenates two lists
  - (++) :: [a] -> [a] -> [a]

Recall, any infix operator can be viewed as a function by putting parentheses around it.

```
 > [1,2,3]++[4,5,6]
=> [1,2,3,4,5,6]
 > ["not"]++["married"]
=> ["not","married"]
 > ['n','o','t']++"married"
=> "notmarried"
 >
```

But…

List Operators

- **:** - (cons operator) glue elements together to form a list
- the last elements has to always be a list
  - (:) :: a -> [a] -> [a]

```
λ> 1 : [2,3]
=> [1,2,3]
λ> 1 : 2 : 3 : [] == [1,2,3]
=> True
λ> 1 : rest = [1,2,3]
λ> rest
=> [2,3]
λ> 1 : rest = [4,5,6]
λ> rest
*** Exception: <interactive>:26:1-18: Non-exhaustive patterns in 1 : rest
λ> a : 2 : rest = [3,2,1,0]
λ> a
=> 3
λ> rest
=> [1,0]
λ>
```

List Operators

- **head** – return the <u>first</u> element of a list
  - head :: [a] -> a

```
 ⊱ head [1,2,3]
=> 1
 ⊱ head "Hello"
=> 'H'
 ⊱ head []
 ⊱ x = head []
 ⊱ x
*** Exception: Prelude.head: empty list
 ⊱
```

List Operators

- **tail** – return the list <u>without</u> its first element
  - tail :: [a] -> [a]

```
 > tail [1,2,3]
=> [2,3]
 > tail "Hello"
=> "ello"
 > x = tail []
 > x
*** Exception: Prelude.tail: empty list
 >
```

List Operators

```
(a) x = ["hello"] ++ ["there"]

(b) x = ["hello" ++ "there"]

(c) joe = (32, 185, "married", "pilot")
    jack = (29, 160, "not married", "cook")
    people = [joe, jack]

(d) l = [[1,2,3],["one", "two", "three"]]

(e) x = [1,2,3]
    h = head(x)
    t = tail(x)
    l = h : t
    l?

(f) y = 1::2::3
```