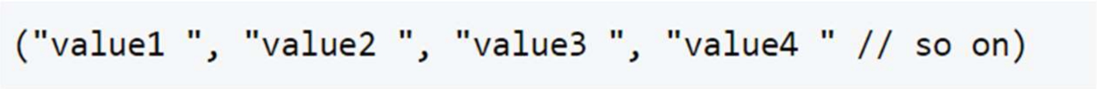
Tuples in Haskell

S6CSE, Amritapuri Campus

Tuples

- The tuple in Haskell is a fixed sized structure which can hold any type of data type inside it.
- They are fixed in number so we can use them where we know the number of values need to be stored.

Syntax of Tuple:

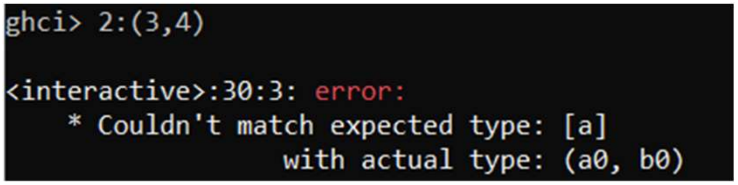


```
("value1 ", "value2 ", "value3 ", "value4 " // so on)
```

Example: (100, “hello”)

Tuples

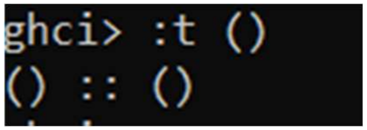
- Tuples, are used when you know **exactly how many values you want** to combine, and its type depends on how many components it has and the types of the components.
- They are denoted with **parentheses** and their components are separated by commas.
- A tuple can contain a combination of several types.
- Tuples have a *fixed* number of elements (*immutable*); you **can't cons** to a tuple.



```
ghci> 2:(3,4)

<interactive>:30:3: error:
    * Couldn't match expected type: [a]
                  with actual type: (a0, b0)
```

- The empty tuple ( ) is pronounced "unit"
  - the unit type is called **()** and its only value is also () , reflecting the 0-tuple interpretation.



```
ghci> :t ()
() :: ()
```

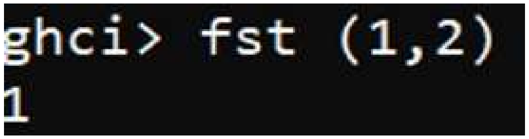
Tuples – as Singleton

- Note that Haskell uses parentheses for forming tuples and also for enforcing a particular order of evaluation within an expression.
- Any expression **x** is equivalent to the singleton tuple with first (and only) component **x**.
  - For example, the expressions "foo", ("foo"), and (("foo")) are all equivalent

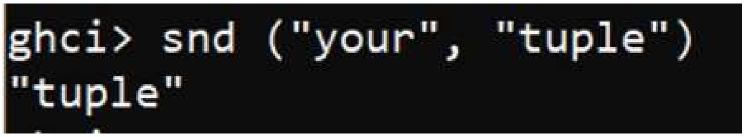
Tuples – as pair

- A tuple with two components is called a pair.

- The predefined functions fst and snd are applicable to pairs and return the first and second components of the pair, respectively.
  - Example: fst ((1, 2, 3), True) evaluates to (1, 2, 3)



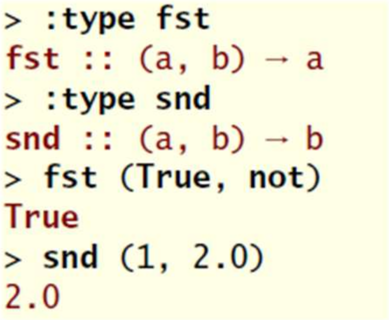
```
ghci> fst (1,2)
1
```



```
ghci> snd ("your", "tuple")
"tuple"
```

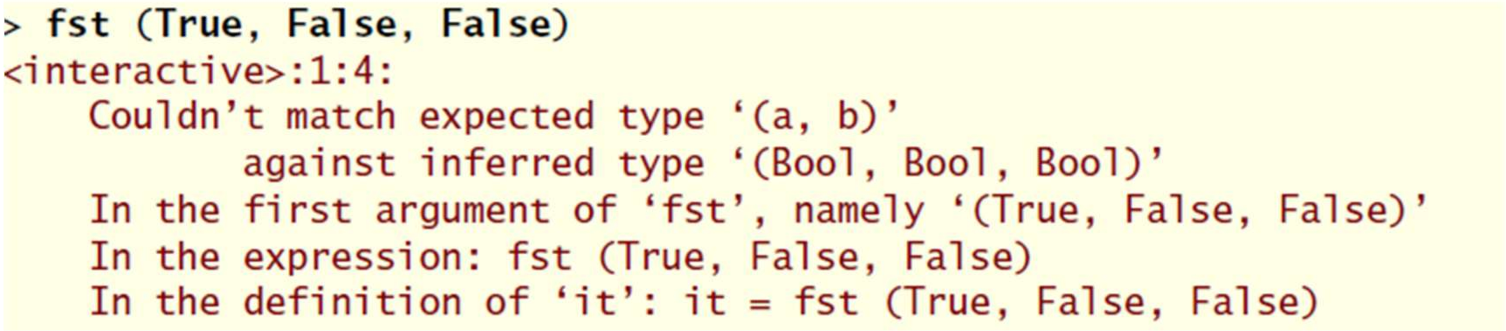
Deconstructing tuples with pattern matching

- The functions **fst** and **snd** are polymorphic.



```
> :type fst
fst :: (a, b) → a
> :type snd
snd :: (a, b) → b
> fst (True, not)
True
> snd (1, 2.0)
2.0
```

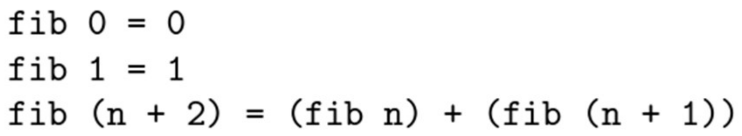
- **fst** and **snd** can only be applied to pairs, not to generic tuples:



```
> fst (True, False, False)
<interactive>:1:4:
    Couldn't match expected type '(a, b)'
           against inferred type '(Bool, Bool, Bool)'
    In the first argument of 'fst', namely '(True, False, False)'
    In the expression: fst (True, False, False)
    In the definition of 'it': it = fst (True, False, False)
```

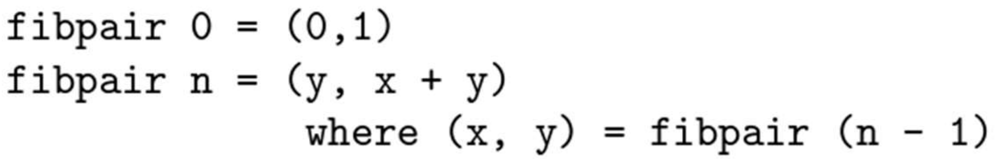
Tuples – Fibonacci Revisited

- Earlier in recursive functions, we presented a recursive script for computing the Fibonacci series.



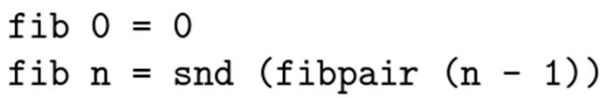
```
fib 0 = 0
fib 1 = 1
fib (n + 2) = (fib n) + (fib (n + 1))
```

- The following function fibpair provides the basis for a more efficient implementation of fib



```
fibpair 0 = (0,1)
fibpair n = (y, x + y)
            where (x, y) = fibpair (n - 1)
```

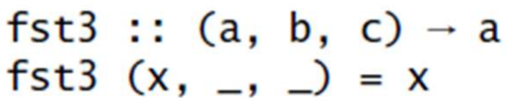
- We can then define fib as follows



```
fib 0 = 0
fib n = snd (fibpair (n - 1))
```

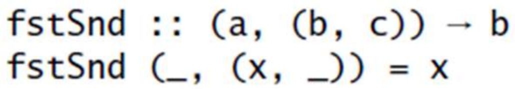
Tuples with pattern matching

- What is needed is a more general mechanism for accessing the components of a tuple.
  - This mechanism is a natural generalization of pattern matching.
- Pattern matching generalizes without problems to tuples of arbitrary length:

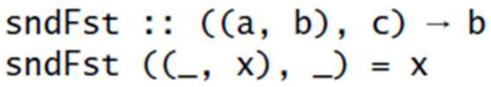


```
fst3 :: (a, b, c) → a
fst3 (x, _, _) = x
```

  - the second and third components of the triple are not used, hence they are matched by the pattern _ without giving them a name.
- Pattern matching can also be used for accessing "deep" components of a tuple.



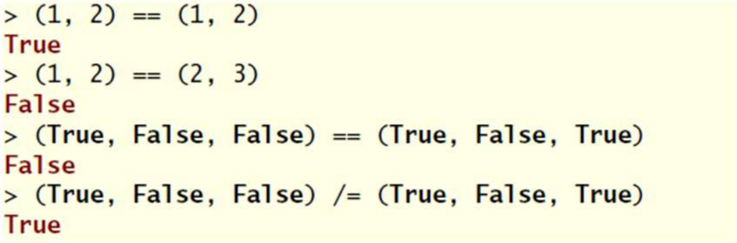
```
fstSnd :: (a, (b, c)) → b
fstSnd (_, (x, _)) = x
```



```
sndFst :: ((a, b), c) → b
sndFst ((_, x), _) = x
```

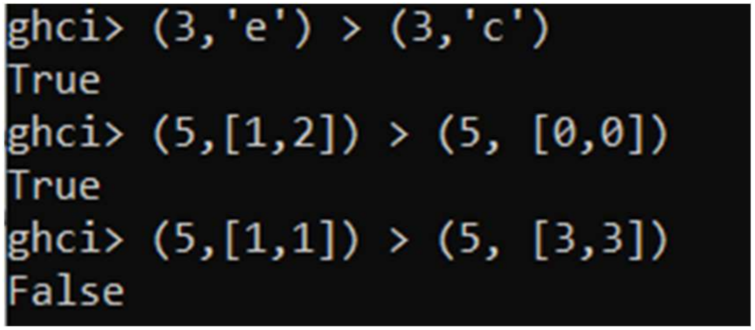
Comparing tuples

- Equality and inequality operators work seamlessly with tuples.

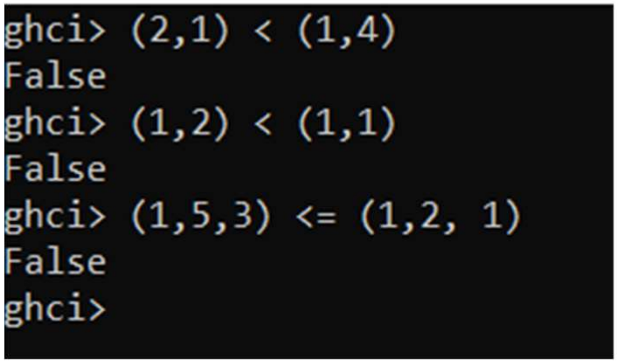


```
> (1, 2) == (1, 2)
True
> (1, 2) == (2, 3)
False
> (True, False, False) == (True, False, True)
False
> (True, False, False) /= (True, False, True)
True
```

- it is not possible to compare tuples with different types
- Tuples are ordered by default by means of lexicographic order.



```
ghci> (3,'e') > (3,'c')
True
ghci> (5,[1,2]) > (5, [0,0])
True
ghci> (5,[1,1]) > (5, [3,3])
False
```

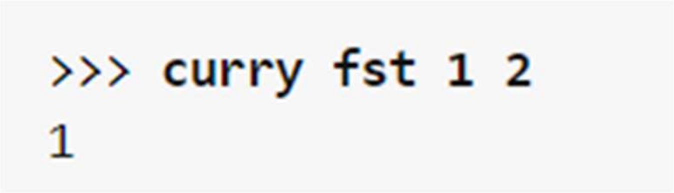


```
ghci> (2,1) < (1,4)
False
ghci> (1,2) < (1,1)
False
ghci> (1,5,3) <= (1,2, 1)
False
ghci>
```

Functions for Tuples

curry

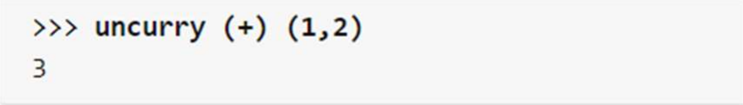
- curry converts an uncurried function to a curried function.



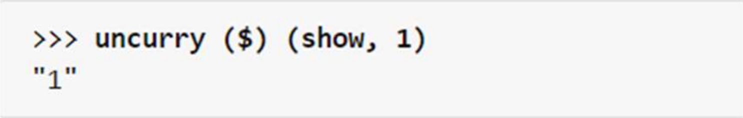
```
>>> curry fst 1 2
1
```

uncurry

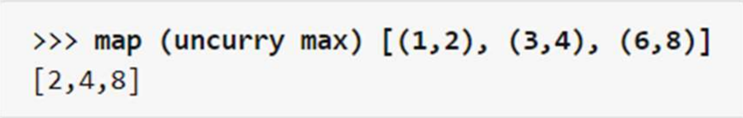
- uncurry converts a curried function to a function on pairs.



```
>>> uncurry (+) (1,2)
3
```

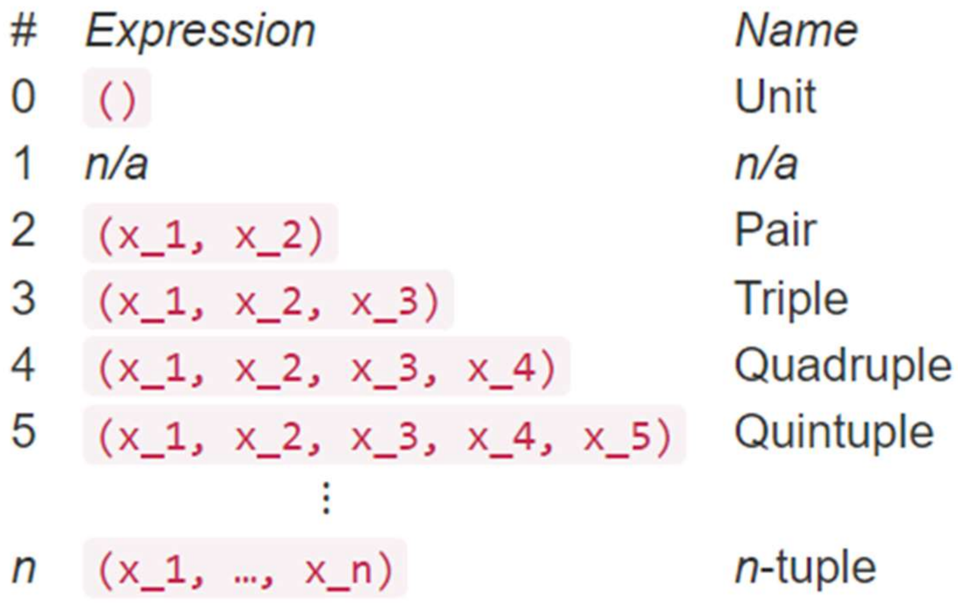


```
>>> uncurry ($) (show, 1)
"1"
```



```
>>> map (uncurry max) [(1,2), (3,4), (6,8)]
[2,4,8]
```

Special names for some tuples.



#	*Expression*	*Name*
0	`()`	Unit
1	*n/a*	*n/a*
2	`(x_1, x_2)`	Pair
3	`(x_1, x_2, x_3)`	Triple
4	`(x_1, x_2, x_3, x_4)`	Quadruple
5	`(x_1, x_2, x_3, x_4, x_5)`	Quintuple
	⋮	
n	`(x_1, …, x_n)`	*n*-tuple

Next - Higher Order Functions