

Tuples' components are extracted using pattern-matching, either in function parameters:

```
> f :: (Int,Bool) -> String
> f (n,b)
>   | b           = show n
>   | otherwise = "none"
```

or in local definitions:

```
> dozens :: Int -> Int
> dozens n
>   | r == 0    = d
>   | otherwise = d+1
> where
>   (d,r) = quotAndRem n 12
```

These patterns are called *irrefutable*, because they require no run-time pattern-matching tests—passing the compiler's type check guarantees a match.

Tuples can be used to simulate “uncurried” functions. For example,

```
> ucPlus :: Num a => (a,a) -> a
> ucPlus (a,b) = a+b
```

In Haskell's terms, `ucPlus` is a function of one argument, but it closely resembles a two-argument function in a conventional programming language.

Lists

Functional programming's most-used data structure is the *list*, which plays the rôle in Haskell which is played in conventional programming languages by the array.

One way to define a list is to *enumerate* its elements:

```
[3, 7+3, 4*6, 9, 2]
```

Square brackets are sufficient—but not necessary—to signify a list.

The *empty* list: `[]`

Lists' types

Lists are *homogeneous*—in any given list, all of the elements are of the same type.

A list's *type* is expressed by enclosing its elements' type in brackets, e.g.,

```
[True, False, 3>9] :: [Bool]
```

```
['I', 'o', 'w', 'a'] :: [Char]
```

```
[['N', 'e', 'w'], ['Y', 'o', 'r', 'k']] :: [[Char]]
```

```
[3, 7+3, 4*6, 9, 2] :: Num a => [a]
```

```
[(+), (-), (*), div] :: Integral a => [a -> a -> a]
```

Lists' elements can belong to *any type whatsoever*.

The list brackets `[]` appear in two rôles:

- **value constructor**, as in `[]`, `[3]`, `[m+n, m-n]`
- **type constructor**, as in `[Int]`, `[Bool]`, `[a]`

Strings

The type `String` is just a synonym for `[Char]`; as the Prelude says,

```
type String = [Char] -- type defines a new name for
-- an existing type
```

so we could say

```
['I', 'o', 'w', 'a'] :: String
[['N', 'e', 'w'], ['Y', 'o', 'r', 'k']] :: [String]
```

Literal strings get special syntactic treatment; these two lists would normally be written as

```
"Iowa"
["New", "York"]
```

Defining strings as lists of characters means that all of the general-purpose list software works on strings.

Outputting Strings

Hugs displays a result on the worksheet in the form in which it would be represented in a program as a literal value.

That principle works nicely except for strings, which are displayed with superfluous punctuation:

```
Prelude> "on two\nlines"
"on two\nlines"
```

To get rid of the extra punctuation, use `putStr`:

```
Prelude> putStr "on two\nlines"
on two
lines
```

Conversion to String

Values of many types have `String` representations, which are produced by the Prelude function `show`:

```
8 * 5           ~~~> 40
show (8 * 5)    ~~~> "40"
(8 * 5) ++ " acres" ~~~> type error
show (8 * 5) ++ " acres" ~~~> "40 acres"
```

There are actually many functions named `show`:

```
show :: Int -> String
show :: Float -> String
show :: Bool -> String
```

... etc.

Hugs uses the `show` functions to produce the `String` representations of values that it “prints” on the worksheet.

Which `show` function Hugs uses depends on the value's type. For some types, there is no `show` function:

```
Prelude> (+)
ERROR - Cannot find "show" function for:
*** Expression : (+)
*** Of type    : Integer -> Integer -> Integer
```

And no `show` function is polymorphic:

```
Prelude> []
ERROR - Cannot find "show" function for:
*** Expression : []
*** Of type    : [a]

Prelude> [] :: [Int]
[]
```

Arithmetic-sequence expressions

```
[3..9]    ~> [3,4,5,6,7,8,9]
[3,5..9]  ~> [3,5,7,9]
[0..]     ~> [0,1,2,3,...    -- the natural numbers
[3,5..]   ~> [3,5,7,9,11,13,15,...
```

The last two are *infinite* lists. Because of Haskell's laziness, their implementations (i.e., their representations in memory) are finite.

Sequence expressions are also defined for type Char:

```
['a'..'z'] ~> "abcdefghijklmnopqrstuvwxyz"
['a','c'..'m'] ~> "acegikm"
['a'..]    ~> "abcdefghijklmnopqrstuvwxyz{ }
~\DEL\128\129\130\131\132\133
~\134\135\136\137...\254\255"
```

The last one is not infinite, because type Char is a *bounded* type ('\255' is `maxBound :: Char`).

List comprehensions

Special notation, borrowed from set theory, which enables many lists to be specified more simply and clearly than with function applications.

Example:

```
[ n*2 | n <- [0..10], n `mod` 3 /= 0 ]
~> [2, 4, 8, 10, 14, 16, 20]
```

In English:

“the list of all $n*2$, where n is drawn from the list $[0..10]$ and n is not a multiple of 3”.

The parts:

- $n*2$ is the list comprehension's *body*—an expression that defines the value of each of the resulting list's elements
- $n <- [0..10]$ is a *generator*; it introduces a new variable (n), which is local to the list comprehension.
- $n \text{ `mod` } 3 \neq 0$ is a *filter*; only those values of n for which the filter is True are included in the result.

A few examples using list comprehensions:

```
> divisors :: Int -> [Int]
> divisors n = [ i | i <- [1..n],
>                  n `mod` i == 0 ]
> perfects :: [Int]
> perfects = [ n | n <- [1..],
>                2*n == sum (divisors n) ]
> gcd :: Int -> Int -> Int
> gcd a b = maximum [ d | d <- divisors a,
>                          b `mod` d == 0 ]
```

These examples use a couple of functions—`sum` and `maximum`—which are defined in the Prelude (`Prelude.hs`).

The Prelude is a rich source of list-processing functions, and more can be found in the module `List` (in file `List.hs`).

Two well-known higher-order functions for lists:

```
> map :: (a->b) -> [a] -> [b]
> map f xs = [ f x | x <- xs ]

> filter :: (a->Bool) -> [a] -> [a]
> filter p xs = [ x | x <- xs, p x ]
```

Examples:

```
map (*2) [1,3,4,4,2] ~> [2,6,8,8,4]
filter (<=3) [1,3,4,4,2] ~> [1,3,2]
```

A list comprehension can have more than one generator.

Example (Pythagorean triples):

```
[ (a,b,c) | a<-[1..20], b<-[1..20], c<-[1..20],
          a^2 + b^2 == c^2 ]
~> [(3,4,5), (4,3,5), (5,12,13), (6,8,10), (8,6,10),
(8,15,17), (9,12,15), (12,5,13), (12,9,15),
(12,16,20), (15,8,17), (16,12,20)]
```

Each generator introduces a **new variable** whose scope consists of

- the list comprehension's body
- any generators and filters that follow it

A more efficient, and less redundant, version of the Pythagorean-triples expression:

```
[ (a,b,c) | a<-[1..20], b<-[a+1..20],
          c<-[b+1..20], a^2 + b^2 == c^2 ]
~> [(3,4,5), (5,12,13), (6,8,10), (8,15,17),
(9,12,15), (12,16,20)]
```

List comprehensions resemble *for*-loops; multi-generator list comprehensions are like *nested for*-loops.

A C-like *analog* of the second Pythagorean-triples expression:

```
result = [];
for( a = 1; a <= 20; a++ )
  for( b = a+1; b <= 20; b++ )
    for( c = b+1; c <= 20; c++ )
      if( a*a + b*b == c*c )
        result = result ++ [(a,b,c)];
```