



Strings

- Literal strings are written in double quotes. For example:

```
"Hello, 007!" "a" "Here is a \nnew line."
```

- If you enter a string as an expression at the Hugs prompt, it gives the string back as its value. For example:

```
Prelude> "This is a \nnew line."  
"This is a \nnew line."
```

- If you want the string itself to be output, use the standard function `putStr` as follows:

```
Prelude> putStr "This is a \nnew line."  
This is a  
new line.
```



show

- The standard function `show` takes a value (of any type) and returns a string representing that type.
- The `show` function is useful for constructing output strings from values of different types. (The `(++)` operator joins strings together.) For example:

```
Prelude> "The answer is " ++ show (6*7) ++ ". "  
"The meaning of life is 42."
```

- Without `show` we get a type error:

```
Prelude> "The answer is " ++ (6*7) ++ ". "  
ERROR: Cannot infer instance...
```



Tuples

- A tuple is a finite sequence of elements which can have different types. For example:

```
(1, "Smith", True)
```

has type

```
(Int, String, Bool)
```

- There are standard functions for extracting the first and second elements of a pair:

```
fst (x, y) == x
```

```
snd(x, y) == y
```

Lists

- A list is a sequence of elements of the same type. For example:

`[1, 2, 3]` is of type `[int]`

`["Smith", "Jones", "Brown"]` is of type `[String]`

- An empty list is represented by `[]`
- The infix operator `:` (cons) makes a list by putting an element in front of an existing list. For example:

```
1 : 2 : 3 : 4 : []  
== 1 : 2 : 3 : [4]  
== 1 : 2 : [3, 4]  
== 1 : [2, 3, 4]  
== [1, 2, 3, 4]
```

Lists

- Every list is either empty or has the form $h:t$, where h is called the *head* of the list and t is called the *tail*.
- The built-in functions `head` and `tail` can be used to extract the head and tail respectively from a list. For example:

```
head [1,2,3,4] == 1
```

```
tail [1,2,3,4] == [2,3,4]
```

- The infix operator `++` (append) joins two lists together:

```
[1,2,3] ++ [4,5,6] == [1,2,3,4,5,6]
```

- The infix operator `!!` selects the n^{th} element of a list:

```
[1,9,37,8] !! 2 == 37
```

Arithmetic Sequences

- Haskell provides a special ‘ellipsis’ or ‘dot-dot’ syntax for arithmetic progressions. For example:

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

- To specify step sizes other than 1, give the first 2 numbers. For example:

```
[1, 3..10] == [1,3,5,7,9]
```

```
[0,10..50] == [0,10,20,30,40,50]
```

- We can also have infinite lists. For example:

```
[1..] == [1,2,3,4,5,6, ...]
```

```
[1,3..] == [1,3,5,7,9, ...]
```

List Comprehensions

- List comprehensions are a convenient way to define lists where the elements must all satisfy certain properties.
- For example, the squares of all the values from 1 to 10 which are even is defined as follows:

```
[x^2 | x <- [1..10], even x]
```

- `x <- [1..10]` is the *generator* and `even x` is the *filter*
- With two generators, we get all possible pairs of values:

```
[(x,y) | x <- [1..3], y <- ['a'..'b']] ==  
[(1,'a'), (1,'b'), (2,'a'), (2,'b'), (3,'a'), (3,'b')]
```

Type Synonyms

- Haskell allows us to give our own names to types we have constructed.
- For example, we may represent a quadratic $ax^2 + bx + c$ as a triple `(Float, Float, Float)`
- Scripts can be made more readable by defining a more meaningful name for this type:

```
type Quadratic = (Float, Float, Float)
```

- Similarly, dates like 12/3/1977 can be represented as triples `(Int, Int, Int)` so we may define a type synonym:

```
type Date = (Int, Int, Int)
```