Polymorphic Functions

In many languages we can't write a generic sort routine, i.e. one that can sort arrays of integers as well as arrays of reals:

```
procedure Sort (var A : array of <type>; n : integer);
```

In Haskell (and many other FP languages) we can write polymorphic ("many shapes") functions.

Functions of polymorphic type are defined by using type variables in the signature:

```
length :: [a] -> Int
length s = ...  
```

Polymorphic Functions

length is a function from lists of elements of some (unspecified) type \(a\), to integer. I.e. it doesn't matter if we're taking the length of a list of integers or a list of reals or strings, the algorithm is the same.

- \(\text{length } [1,2,3] \Rightarrow 3\) (list of Int)
- \(\text{length } ["Hi ", "there", "]!\] \Rightarrow 3\) (list of String)
- \(\text{length } "Hi!" \Rightarrow 3\) (list of Char)

We have already used a number of polymorphic functions that are defined in the standard prelude.

```
head :: [a] -> a
head = ...  
tail :: [a] -> [a]
tail = ...  
(;) :: a -> [a] -> [a]
```

takes two arguments: an element of some type \(a\) and a list of elements of the same type. It returns a list of elements of that type.

```
read :: (\"") \rightarrow [a]
read (\"") = ...  
read (\"") = ...  
```

reads a function from "lists-of-things" to "things".

We have already used a number of polymorphic functions that are defined in the standard prelude.
Polymorphic Functions

Note that `head` and `tail` always take a list as their argument. `tail` always returns a list, but `head` can return any type of object, including a list. Note that it is because of Haskell's strong typing that we can only create lists of the same type of element. If we tried to do `[Int] ++ [True]` the Haskell type checker would complain that we were consing an `Int` onto a list of `Bots`, while the type of `:` is

```
(:) :: a -> [a] -> [a]
```

Context Predicates

Obviously, `remdups` should work for any list, not just lists of `Ints`. Removing duplicates from a list of strings is no different from removing duplicates from a list of integers. However, there's a complication. In order to remove different from removing duplicates from a list of integers, `Ints`. Removing duplicates from a list of strings is no different from removing duplicates from a list of `Ints`. Obviously, `remdups` should work for any list, not just lists of `Ints`.
Haskell uses context predicates to restrict polymorphic types:

```haskell
remdups :: Eq [a] => [a] -> [a]
```

Now, `remdups` may only be applied to lists of elements where the element type has `==` and `\=\` defined.

`Eq` is called a type class. `Ord` is another useful type class. It is used to restrict the polymorphic type of a function to types for which the relational operators (`<`, `<=`, `>`, `>=`) have been defined.

```haskell
signum :: (Num a, Ord a) => a -> Int
signum n | n == 0 = 0
          | n > 0 = 1
          | n < 0 = -1
```

`signum` can be applied to any type that is a number (hence `Num a`) and for which the relational operators are defined (hence `Ord a`). Without these restrictions, the polymorphic `signum` function could have been applied to lists, for example, which would not have made sense.

### Conclusion

We want to define functions that are as reusable as possible.

1. **Polymorphic functions** are reusable because they can be applied to arguments of different types.
2. **Curried functions** are reusable because they can be specialized; i.e, from a curried function we can create a new function by plugging in values for some of the arguments, and leaving others undefined.
3. **Curried functions** are reusable because they can be specialized.

We need to define functions that are as reusable as possible.
A polymorphic function is defined using type variables in the signature. A type variable can represent an arbitrary type. All occurrences of a particular type variable appearing in a type signature must represent the same type. An identifier will be treated as an operator symbol if it is enclosed in backquotes: `'`. An operator symbol can be treated as an identifier by enclosing it in parenthesis: `(+)`.

**Homework**
Define a polymorphic function `dup x` which returns a tuple with the argument duplicated. Example:

? dup 1
(1,1)

? dup "Hello, me again!"
("Hello, me again!", "Hello, me again!")

? dup (dup 3.14)
((3.14,3.14), (3.14,3.14))

**Homework**
Define a polymorphic function `copy n x` which returns a list of `n` copies of `x`.

Example:

? copy 5 "five"

["five","five","five","five","five"]

? copy 5 5

[5,5,5,5,5]

? copy 5 (dup 5)

[(5,5),(5,5),(5,5),(5,5),(5,5)]

**Homework**
Let `f` be a function from `Int` to `Int`, i.e. `f :: Int -> Int`. Define a function `total f x` so that `total f` is the function which at value `n` gives the total `f 0 + f 1 + ... + f n`. Example:

```
double x = 2*x
pow2 x = x^2
totDub = total double
totPow = total pow2
```

? totDub 5

30

? totPow 5

55