

Lambda Abstraction

This is an old notation for function-values where $f = \lambda x.x + 2$

In Haskell, λ becomes `\` and $.$ becomes `->`

```
\x -> e
```

where x is a variable and e is an expression that mentions x

We can have nested abstractions `\x -> \y -> e` read as "the function taking x as input and returning a function that takes y as input and returns e as a result"

```
sqr = \n -> n * n
```

is the equivalent to:

```
sqr n = n * n
```

Factorial

A simple definition of factorial is:

```
fac 0 = 1
```

```
fac n = n * fac (n-1)
```

But what is `fac`?

```
fac = \n -> case n of
```

```
0 -> 1
```

```
m -> m * fac (m-1)
```

Here we use a Haskell case expression that does pattern matching in a general setting

Defining New Types - 3 Ways

- Type Synonyms

```
type Name = String
```

Haskell considers `String` and `Name` to be exactly the same type

- Wrapped Types

```
newType Name = N String
```

If s is a value of type `String`, then `N s` is a value of type `Name`. Haskell considers `String` and `Name` to be different types here.

- Algebraic Data Types

```
data Name = Official String String — NickName String
```

If f,s and n are values of type `String`, then `Official f s` and `NickName n` are different values of type `Name`.

User-Defined Data Types: enums

With the `data` keyword, we can easily define new enumerated types.

```
data Day = Monday — Tuesday — Wednesday — Thursday — Friday — Saturday — Sunday
```

We can define operations on values of this type by pattern matching:

```
weekend :: Day -> Bool
```

```
weekend Saturday = True
```

```
weekend Sunday = True
```

```
weekend _ = False
```

The identifiers Monday through Sunday are data constructors, and jsut like the types themselves, must begin with uppercase letters.

User-Defined Data Types: Recursive Structures

If lists were not builtin, we could define them with the data keyword:

```
data List = Empty — Node Int List
```

Using this def, the list [1,2,3] would be written:

```
Node 1 (Node 2 (Node 3 Empty))
```

Recursive types usually mean recursive functions:

```
length :: List -> Integer
```

```
length Empty = 0
```

```
length (Node _rest) = 1 + (length rest)
```

These lists arent as flexible as the builtins because they are not polymorphic but we can fix that by using a type variable.

```
data List t = Empty — Node t (list t)
```

No changes to the length function but the type becomes:

```
length :: (List a) -> Integer
```

Type Parameters

The types defined using type, newtype and data can have type parameters themselves:

```
type TwoList t = ([t],[t])
```

The type "list-of-a" ([a]) can be considered a parameterized type [] a

Whats In a Name?

```
data MyType = AToken — ANum Int — AList [Int]
```

- the name `MyType` after the data keyword is the type name
- the names `AToken`, `ANum`, `AList` are data-constructor names
- type names and data constructor names are in different namespaces so they can overlap e.g.

```
data Thing = Thing String — Thing Int
```

- The same principle applies to to newtypes:

```
newtype Nat = Nat Int
```

- We call these algebraic data types

Multiply-Parameterised Data Types

Here is a useful data type:

```
data Pair a b = Pair a b
```

```
divmod :: Integer -> Integer -> (Pair Integer Integer)
```

```
divmod x y = Pair (x / y) (x 'mod' y)
```

data-types in the Prelude (I)

- `data () = ()` -- Not legal syntax; builtin
- `data Bool = False | True`
- `data Char = ... 'a' | 'b' ...` -- Unicode values
- `data Maybe a = Nothing | Just a`
- `data Either a b = Left a | Right b`
- `data Ordering = LT | EQ | GT`
- `data [a] = [] | a : [a]` -- Not legal syntax; builtin

Example: failure

We can write functions such as `head` so that they fail outright:

`head (x:xs) = x` No `[]` pattern, so runtime fail for `head []`

Define type `Maybe` to represent an optional value of type `a`.

`data Maybe a = Nothing | Just a`

Now we can handle failure for `head` in a more manageable way:

`mhead :: [a] -> Maybe a`

`mhead [] = Nothing`

`mhead (x:Xs) = Just x`

Maybe

`Maybe` is a good way to deal with errors or exceptional cases. The `Maybe` type encapsulates an optional value. A value of type `Maybe a` either contains a value of type `a`, represented as `Just a` or it is empty, represented as `Nothing`.