Defining new types (3 possibilities)

- *Type Synonyms*

  ```
  type Name = String
  ```

  Haskell considers both `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.
- *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`

Type Synonyms

```
type MyType = ExistingType
```

Haskell considers both `MyType` and `ExistingType` to be exactly the same type.

- Advantages
  Clearer code documentation
  Can use all existing functions defined for `ExistingType`
- Disadvantages
  Typechecker does not distinguish `ExistingType` from any type like `MyType` defined like this

  ```
  type Name = String ; name :: Name ; name = "Andrew"
  type Addr = String ; addr :: Addr ; addr = "TCD"
  name ++ addr  -- is well-typed
  ```

"Wrapping" Existing Types

```
newtype NewType = NewCons ExistingType
```

If `v` is a value of type `ExistingType`, then `NewCons v` is a value of type `NewType`.

- Advantages
  Typechecker treats `NewType` and `ExistingType` as different and incompatible.
  Can use type-class system to specify special handling for `NewType`.
  No runtime penalties in time or space !
- Disadvantages
  Needs to have explicit `NewCons` on front of values
  Need to pattern-match on `NewCons v` to define functions
  None of the functions defined for `ExistingType` can be used directly

Algebraic Data Types (ADTs)

```
data ADTName
   = Dcon1 Type11 Type12 ... Type1a
   | Dcon2 Type21 Type22 ... Type2b
   ...
   | DconN TypeN1 TypeN2 ... TypeNz
```

- If `vi1`, ... `vik` are values of types `Typei1` ... `Typeik`, then `Dconi vi1 ... vik` is a value of type `ADTName`, and values built with different `Dconi` are always different
- Note that a `Dconi` can have no `Typeij`, in which case `Dconi` itself is a value of type `ADTName`.

Algebraic Data Types (ADTs)

```
data ADTName
   = Dcon1 Type11 Type12 ... Type1a
   | Dcon2 Type21 Type22 ... Type2b
   ...
   | DconN TypeN1 TypeN2 ... TypeNz
```

- Advantages
  The only way to add genuinely *new* types to your program
- Disadvantages
  As per `newtype` — the need to use the `Dconi` data-constructors, and to pattern match
  Unlike `newtype`, these `data` types do have runtime overheads in space and time.

Type Parameters

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

- `type TwoList t = ([t],[t])`
- `newtype BiList t = BiList ([t],[t])`
- `data ListPair t = LPair [t] [t]`
- The type "list-of-`a`", (`[a]`) can be considered a parameterised type: `[] a`.
- The names `TwoList`, `BiList`, `ListPair`, and `[]` (in the type-language of Haskell) are considered to be *Type Constructors*. They take a type as argument and build a new type using that argument.

Defining Functions with ADT Patterns

Consider a generic example of a `data`-declaration:

```
data ADTName
   = Dcon1 Type11 Type12 ... Type1a
   | Dcon2 Type21 Type22 ... Type2b
   ...
   | DconN TypeN1 TypeN2 ... TypeNz
```

We can define a function `myfun :: ADTName -> a` as follows:

```
myfun (Dcon1 pat11 pat12 ... pat1a) = exp1
myfun (Dcon2 pat21 pat22 ... pat2b) = exp2
...
myfun (DconN patN1 patN2 ... patNz) = expN
```

Here `patIJ` has type `TypeIJ` and all `expK` have type `a`.

User-defined Datatypes (`data`): 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` thru `Sunday` are *Data Constructors*, and like the types themselves, must begin with *uppercase* letters (functions and parameters in Haskell begin with lowercase letters).

User-defined Datatypes (data): Recursive structures

Haskell also allows data types to be defined *recursively*.
If lists were not built-in, we could define them with `data`:

```
data List = Empty
          | Node Int List
```

compare:

```
typedef struct {
  int value;
  node *next;
} node;
```

User-defined Datatypes (data): Recursive structures

```
data List = Empty
          | Node Int List
```

Using this definition the list $\langle 1, 2, 3 \rangle$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)
```

Parameterised data types

Of course, those lists are not as flexible as the built-in lists, because they are not *polymorphic*. We can fix that by introducing a *type-variable*:

```
data List t = Empty
            | Node t (List t)
```

compare:

```
class Node<T> {
  T value;
  Node<T> *next;
}
```

No change to the length function, but the type becomes:

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

What's in a Name?

Consider the following `data` declaration:

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

- the name `MyType` after the `data` keyword is the *type* name.
- the names `AToken`, `ANum` and `AList` on the rhs are *data-constructor* names.
- type names and data-constructor names are in different namespaces so they can overlap, e.g.:
  `data Thing = Thing String | Thang Int`
- The same principle applies to newtypes:
  `newtype Nat = Nat Int`
- We call these **Algebraic Datatypes** (ADTs)
- For a nice explanation of the name (if interested) see: [1]

[1] https://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-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)
```

Actually, like lists, "tuples" (of various sizes) are built in to Haskell and have a convenient syntax:

```
divmod :: Integer -> Integer -> (Integer,Integer)
divmod x y = (x / y, x `mod` y)
```

As you would expect, we can use pattern matching to open up the tuple:

```
f (x,y,z) = x + y + z
```

data-types in the Prelude (I)

- `data () = () -- Not legal; for illustration`
- `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; for illustration`

data-types in the Prelude (II)

- `data IO a = ... -- abstract`
- `data (a,b) = (a,b)`
  `data (a,b,c) = (a,b,c)`
  `-- Not legal; for illustration`
- `data IOError -- internals system dependent`

data-types in the Prelude (III)

Standard numeric types.

The data declarations for these types cannot be expressed directly in Haskell since the constructor lists would be far too large.

- `data Int = minBound ...  -1 | 0 | 1 ...  maxBound`
- `data Integer = ...  -1 | 0 | 1 ...`
- `data Float`
- `data Double`

Another example: failure

A type that is often used in Haskell is one to model failure. While we can write functions such as `head` so that they fail outright:

```
head (x:xs) = x
```

It is sometimes useful to model failure in a more manageable way:

```
data Maybe a = Nothing
             | Just a
```

Every `Maybe` value represents either a success or failure:

```
mhead :: [a] -> Maybe a
mhead []     = Nothing
mhead (x:xs) = Just x
```

This technique is so common that `Maybe` and some useful functions are included in the standard Prelude.