Lambda abstraction

Since functions are first class entities, we should expect to find some notation in the language to create them from scratch.
There are times when it is handy to just write a function "inline".
The notation is:

```
\ x -> e
```

where `x` is a variable,
and `e` is an expression that (usually) mentions `x`.
This notation reads as "the function taking `x` as input and returning `e` as a result". 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".
There is syntactic sugar for nested abstractions:

```
\ x  y -> e
```

It's just notation!

The following definition groups are equivalent:

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


add     =  \ x y -> x+y
add x   =    \ y -> x+y
add x y =           x+y
```

This notation is based on the so-called "lambda-calculus" (after reading week!).

Lambda application

In general, and application of a lambda abstraction to an argument looks like:

```
(\ x -> x + x) a
       ^--e--^
-- Applied:
(a + a)
```

The result is a copy of `e` where any free occurrence of `x` has been replaced by `a`.

Factorial: a comparison

A simple definition of factorial, ignoring negative numbers, is the following:

```
fac 0 = 1
fac n = n * fac (n-1)
```

But what is `fac`?

```
fac = ..... ?


fac = \n -> case n of
             0 -> 1
             m -> m * fac (m-1)
```

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 MyName = ExistingType
```

Haskell considers both `MyName` 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 `MyName` defined like this

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

"Wrapping" Existing Types

```
newtype NewName = NewCons ExistingType
```

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

- Advantages
  Typechecker treats `NewName` and `ExistingType` as different and incompatible.
  Can use type-class system to specify special handling for `NewName`.
  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 ... Type1k1
   | Dcon2 Type21 Type22 ... Type2k2
   ...
   | Dconk Typek1 Typek2 ... Typenkn
```

If `vi1`, ...`viki` are values of types `Typei1` ...`Typeiki`, then `Dconi vi1 ...  viki` is a value of type `ADTName`, and values built with different `Dconi` are always different

- 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 themselvesL

- `type TwoList t = ([t],[t])`
- `newtype BiList t = BiList ([t],[t])`
- `data ListPair t = LPair [t] [t]`

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*.
We are familiar by now with lists in Haskell: writing the list `[1,2,3]` is just a shorthand for writing `1:2:3:[]`.
If lists were not built-in, we could define them with `data`:

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

compare:

```
typedef struct {
  node *next;
  int value
} 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> {
  Node<T> *next;
  T value;
}
```

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.

History of Functional Programming (I)

- Combinatory logic, λ-calculus (1920s, 1930s)
  Foundations for *mathematics*, not computing!
- LISP, (late '50s, early '60s)
  Artificial Intelligence
- APL (early '60's)
  symbol-based, functions/operators as building blocks
- ML, SASL, NPL (1970s')
  type-inference, pattern-matching
- FP - John Backus Turing Award Speech (1977)
  inventor of Fortran and much parsing technology argues for functional progrmming
- Haskell starts (1987)

History of Functional Programming (II)

We focus on ML (early 70s')

- Robin Milner and colleagues
- Developing early theorem provers
- Provers based on a logic called
  the Logic of Computable Functions (LCF)
- Needed a very well-defined programming language to implement these
- Enter ML (Meta-Language), just such a programming language
  - Formal Semantics
  - Pattern Matching
  - Type Inference
- Still the basis for most modern theorem provers (HOL4/Isabelle/CoQ)
- Since evolved into SML and OCaml