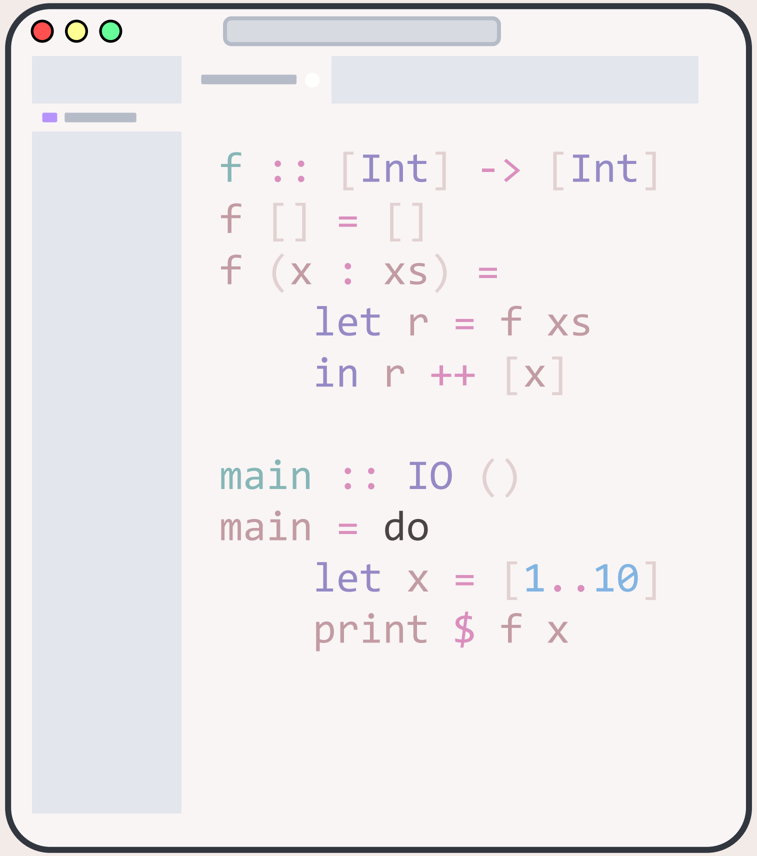
IT5100A

Industry Readiness:
Typed Functional Programming

Types

Foo Yong Qi

yongqi@nus.edu.sg



Types and Type Systems

Type Systems

A **tractable syntactic method** for **proving the absence of certain program behaviours** by classifying phrases according to the **kinds of values they compute**

Type Systems

A system to show that there won't be type errors

Types

A **type** is some **kind** of thing

- Gives meaning to something
- Description of what its inhabitants are like

```
>>> x = 1
>>> type(x)
<class 'int'>
>>> type('abc')
<class 'str'>
>>> class A: pass
>>> type(A())
<class '__main__.A'>
```

In Python, types more-or-less correspond to classes

Types

We can declare something to be of a type with type declarations

```python
# Python
def f(x: int) -> str:
    y: int = x * 2
    return f'{x} * 2 = {y}'
z: int
z = 3
s: str = f(z)
print(s) # 3 * 2 = 6
```

Types

Type declarations in Haskell are done with `::`

```haskell
f :: Int -> String
f x = show x ++ " * 2 = " ++ show y
    where y = x * 2
z :: Int
z = 3
s :: String
s = f z -- 3 * 2 = 6
```

Some basic Haskell types:

`Int`, `Char`, `[Char]` (a.k.a. `String`), `[Int]`, `Bool`, `Double`, etc.

Types of Functions

$$\frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \lambda x. e : S \to T}$$

Types of Functions

Assumption that x has type S

Judgement that e has type T

Typing environment (types of surrounding code)

$$\frac{\boxed{\Gamma},\ \boxed{x : S} \vdash \boxed{e : T}}{\boxed{\Gamma} \vdash \boxed{\lambda x.\, e : S \to T}}$$

Judgement that $\lambda x.\, e$ has type $S \to T$

Types of Functions

$$\frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \lambda x.\, e : S \to T}$$

Since, if `x` has type `int` then `x * 2` has type `int`,
Then `lambda x: x * 2` has type `int -> int`

Types of Functions

$$\frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \lambda x.\, e : S \to T}$$

If `f(x: int)` returns `str`, then `f` itself has type `int -> str`

Types of Functions

`->` is right-associative

```
a -> b -> c == a -> (b -> c)
```

Types of everything in Haskell are **fixed**, type declarations can be omitted due to **type inference**

```haskell
f :: Int -> String -- explicit type declaration
f x = show x ++ "!"

g x = x + 1 -- type of g is inferred
```

Good habit to declare types (in Python too)

Compiler will reject program that is ill-typed

```haskell
f :: Int -> String -- explicit type declaration
f x = show x ++ "!"

g = f "1" -- compiler throws type error as f
          -- receives Int, not String
```

“If it compiles, it works”

Typing Rules

Bindings

```
x = e
```

Type of **x** must be same as type of **e**

Typing Rules

Conditionals

```
if x then y else z
```

- Type of `x` must be `Bool`
- Types of `y` and `z` must both be same as some α
- Entire expression has type α

Typing Rules

Function Applications

```
f x
```

- Type of `f` must be `α -> β`
- Type of `x` must be `α`
- Entire expression has type `β`

Typing Rules

Function Definitions

```
f x = e
```

- Type of `f` must be α `->` β
- Assuming `x` has type α, `e` must have type β

What are the Types?

```haskell
f :: Int -> Int -> [Int]
f x n =
  if n == 0 then
    []
  else
    let r = f x (n - 1)
    in  x : r
```

What are the Types?

```
x, n :: Int
```

```haskell
f :: Int -> Int -> [Int]
f x n =
  if n == 0 then
    []
  else
    let r = f x (n - 1)
    in  x : r
```

```
if ... :: [Int]
```

What are the Types?

n == 0 :: Bool

```haskell
f :: Int -> Int -> [Int]
f x n =
  if n == 0 then
    []
  else
    let r = f x (n - 1)
    in x : r
```

[] :: [Int] => **OK**

x : r :: [Int]

What are the Types?

```
(==) :: Int -> Int -> Bool
  n == 0 :: Bool => OK
```

```
f :: Int -> Int -> [Int]
f x n =
  if n == 0 then
    []
  else
    let r = f x (n - 1)
    in  x : r
```

What are the Types?

```haskell
f :: Int -> Int -> [Int]
f x n =
  if n == 0 then
    []
  else
    let r = f x (n - 1)
    in x : r
```

```haskell
(:) :: Int -> [Int] -> [Int]
x : r :: [Int]
x :: Int
=> r :: [Int]
```

What are the Types?

```
f :: Int -> Int -> [Int]
f x n =
  if n == 0 then
    []
  else
    let r = f x (n - 1)
    in  x : r
```

```
f :: Int -> Int -> [Int]
x, (n – 1) :: Int
r :: [Int] => OK
```

Let the types **guide** your programming

```haskell
f :: Int -> String
f x =
    let sx :: String = show x
        {- show :: Int -> String
           show x :: String -}
        y :: Int = x * 2
        {- (*) :: Int -> Int -> Int
           x * 2 :: Int -}
        sy :: String = show y
        {- (++) :: String -> String -> String
           sx ++ " * 2 = " :: String -}
    in sx ++ " * 2 = " ++ sy
```

Let the types **guide** your programming

```haskell
f :: Int -> String
f x =
    let sx :: String = show x
        {- show :: Int -> String
           show x :: String -}
        y :: Int = x * 2
        {- (*) :: Int -> Int -> Int
           x * 2 :: Int -}
        sy :: String = show y
        {- (++) :: String -> String -> String
           sx ++ " * 2 = " :: String -}
    in sx ++ " * 2 = " ++ sy
```

Let the types **guide** your programming

```haskell
f :: Int -> String
f x =
    let sx :: String = show x
        {- show :: Int -> String
           show x :: String -}
        y :: Int = x * 2
        {- (*) :: Int -> Int -> Int
           x * 2 :: Int -}
        sy :: String = show y
        {- (++) :: String -> String -> String
           sx ++ " * 2 = " :: String -}
    in sx ++ " * 2 = " ++ sy
```

Let the types **guide** your programming

```haskell
f :: Int -> String
f x =
    let sx :: String = show x
        {- show :: Int -> String
           show x :: String -}
        y :: Int = x * 2
        {- (*) :: Int -> Int -> Int
           x * 2 :: Int -}
        sy :: String = show y
        {- (++) :: String -> String -> String
           sx ++ " * 2 = " :: String -}
    in sx ++ " * 2 = " ++ sy
```

Let the types **guide** your programming

```
f :: Int -> String
f x =
    let sx :: String = show x
        {- show :: Int -> String
           show x :: String -}
        y :: Int = x * 2
        {- (*) :: Int -> Int -> Int
           x * 2 :: Int -}
        sy :: String = show y
        {- (++) :: String -> String -> String
           sx ++ " * 2 = " :: String -}
    in sx ++ " * 2 = " ++ sy
```

Let the types **guide** your programming

```haskell
f :: Int -> String
f x =
    let sx :: String = show x
        {- show :: Int -> String
           show x :: String -}
        y :: Int = x * 2
        {- (*) :: Int -> Int -> Int
           x * 2 :: Int -}
        sy :: String = show y
        {- (++) :: String -> String -> String
           sx ++ " * 2 = " :: String -}
    in sx ++ " * 2 = " ++ sy
```

Let the types **guide** your programming

```haskell
f :: Int -> String
f x =
    let sx :: String = show x
        {- show :: Int -> String
           show x :: String -}
        y :: Int = x * 2
        {- (*) :: Int -> Int -> Int
           x * 2 :: Int -}
        sy :: String = show y
        {- (++) :: String -> String -> String
           sx ++ " * 2 = " :: String -}
    in sx ++ " * 2 = " ++ sy
```

Let the types **guide** your programming

```
f :: Int -> String
f x =
    let sx :: String = show x
        {- show :: Int -> String
           show x :: String -}
        y :: Int = x * 2
        {- (*) :: Int -> Int -> Int
           x * 2 :: Int -}
        sy :: String = show y
        {- (++) :: String -> String -> String
           sx ++ " * 2 = " :: String -}
    in sx ++ " * 2 = " ++ sy
```

Let the types **guide** your programming

```haskell
f :: Int -> String
f x =
    let sx :: String = show x
          {- show :: Int -> String
             show x :: String -}
          y :: Int = x * 2
          {- (*) :: Int -> Int -> Int
             x * 2 :: Int -}
          sy :: String = show y
          {- (++) :: String -> String -> String
             sx ++ " * 2 = " :: String -}
    in sx ++ " * 2 = " ++ sy
```

This is particularly useful when defining recursive functions

```haskell
sum' :: [Int] -> Int
sum' ls =
    if null ls then
        0 -- return type must be Int
    else
        let r :: Int = sum' (tail ls)
        {- tail :: [Int] -> [Int]
           tail ls :: [Int]
           sum' :: [Int] -> Int
           sum' (tail ls) :: Int -}
         hd :: Int = head ls
         {- head :: [Int] -> Int
            head ls :: Int -}
        in  hd + r
```

This is particularly useful when defining recursive functions

```
sum' :: [Int] -> Int
sum' ls =
    if null ls then
        0 -- return type must be Int
    else
        let r :: Int = sum' (tail ls)
        {- tail :: [Int] -> [Int]
           tail ls :: [Int]
           sum' :: [Int] -> Int
           sum' (tail ls) :: Int -}
            hd :: Int = head ls
            {- head :: [Int] -> Int
               head ls :: Int -}
        in  hd + r
```

This is particularly useful when defining recursive functions

```haskell
sum' :: [Int] -> Int
sum' ls =
    if null ls then
        0 -- return type must be Int
    else
        let r :: Int = sum' (tail ls)
        {- tail :: [Int] -> [Int]
           tail ls :: [Int]
           sum' :: [Int] -> Int
           sum' (tail ls) :: Int -}
            hd :: Int = head ls
            {- head :: [Int] -> Int
               head ls :: Int -}
        in  hd + r
```

This is particularly useful when defining recursive functions

```haskell
sum' :: [Int] -> Int
sum' ls =
    if null ls then
        0 -- return type must be Int
    else
        let r :: Int = sum' (tail ls)
        {- tail :: [Int] -> [Int]
           tail ls :: [Int]
           sum' :: [Int] -> Int
           sum' (tail ls) :: Int -}
            hd :: Int = head ls
            {- head :: [Int] -> Int
               head ls :: Int -}
        in  hd + r
```

This is particularly useful when defining recursive functions

```haskell
sum' :: [Int] -> Int
sum' ls =
    if null ls then
        0 -- return type must be Int
    else
        let r :: Int = sum' (tail ls)
        {- tail :: [Int] -> [Int]
           tail ls :: [Int]
           sum' :: [Int] -> Int
           sum' (tail ls) :: Int -}
            hd :: Int = head ls
            {- head :: [Int] -> Int
               head ls :: Int -}
        in  hd + r
```

This is particularly useful when defining recursive functions

```haskell
sum' :: [Int] -> Int
sum' ls =
    if null ls then
        0 -- return type must be Int
    else
        let r :: Int = sum' (tail ls)
        {- tail :: [Int] -> [Int]
           tail ls :: [Int]
           sum' :: [Int] -> Int
           sum' (tail ls) :: Int -}
            hd :: Int = head ls
            {- head :: [Int] -> Int
               head ls :: Int -}
        in  hd + r
```

This is particularly useful when defining recursive functions

```
sum' :: [Int] -> Int
sum' ls =
    if null ls then
        0 -- return type must be Int
    else
        let r :: Int = sum' (tail ls)
        {- tail :: [Int] -> [Int]
           tail ls :: [Int]
           sum' :: [Int] -> Int
           sum' (tail ls) :: Int -}
            hd :: Int = head ls
            {- head :: [Int] -> Int
               head ls :: Int -}
        in  hd + r
```

This is particularly useful when defining recursive functions

```haskell
sum' :: [Int] -> Int
sum' ls =
    if null ls then
        0 -- return type must be Int
    else
        let r :: Int = sum' (tail ls)
        {- tail :: [Int] -> [Int]
           tail ls :: [Int]
           sum' :: [Int] -> Int
           sum' (tail ls) :: Int -}
            hd :: Int = head ls
            {- head :: [Int] -> Int
               head ls :: Int -}
        in  hd + r
```

Enforcing type safety can be done on Python programs with pyright!

PYRIGHT

You may also use other static type checkers for Python

Let’s try it!

```
# main.py
def f(x: int, y: int) -> int:
    z = x / y
    return z
```

Checking with pyright…

```
$ pyright main.py
/home/main.py
  /home/main.py:4:12 - error:
    Expression of type "float" is incompatible with return
    type "int"
    "float" is incompatible with "int" (reportReturnType)
1 error, 0 warnings, 0 informations
```

Correct the program and check again

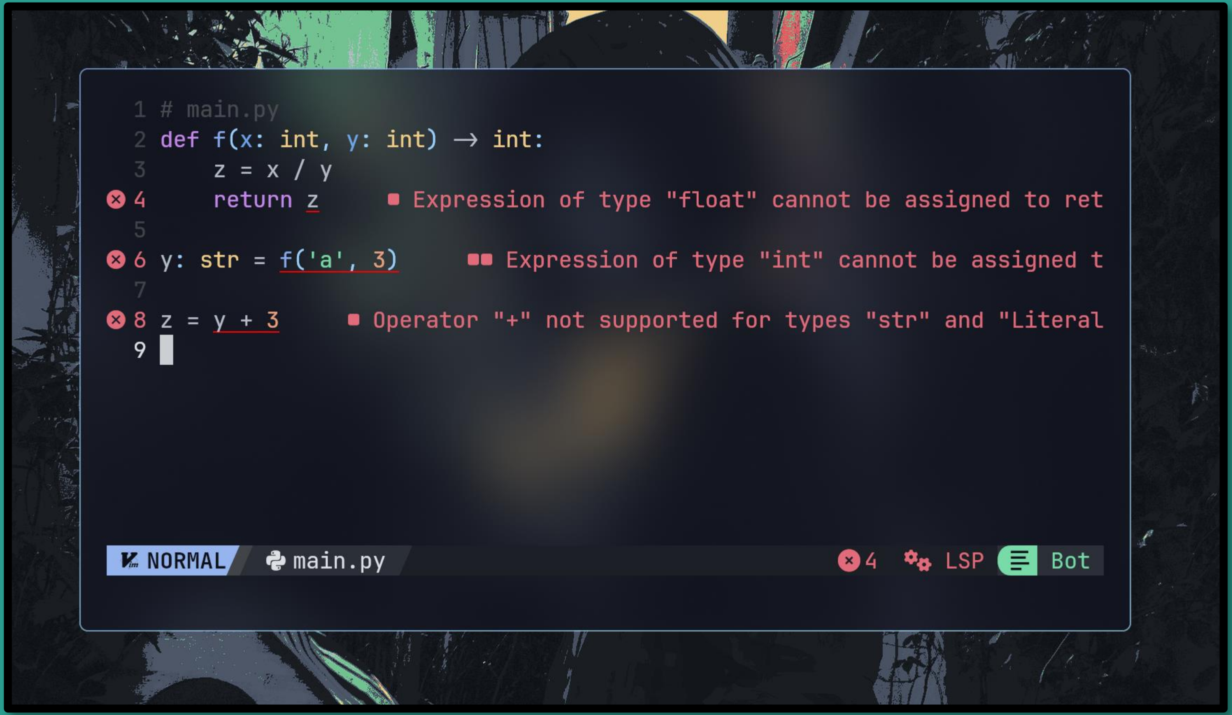
```python
# main.py
def f(x: int, y: int) -> int:
    z = x // y
    return z
```

Checking with pyright...

```
$ pyright main.py
0 errors, 0 warnings, 0 informations
```

Tip

Include pyright (or some Python language server with static type checking) with your editor to show type errors as you write your code



Polymorphism

Functions: terms depend on terms

$$f(x) = x \times 2$$

$$f(2) = 4 \qquad f(3) = 6$$

Polymorphism

When types/terms depend on **types**

Types depending on types

```python
# Python
@dataclass
class IntBox:
    value: int
```

```python
# Python
@dataclass
class StrBox:
    value: str
```

Core programming principle: when a pattern is found, **retain similarities** and **parameterize differences**

Similarity

```python
# Python
@dataclass
class IntBox:
    value: int
```

```python
# Python
@dataclass
class StrBox:
    value: str
```

Difference

Parameterize the **types**!

In Python and many OO languages, **Box** is called a **generic** or **parametrically polymorphic** type

```python
# Python 3.12
@dataclass
class Box[a]:
    value: a
```

```python
x: Box[int] = Box[int](1)
y: Box[str] = Box[str]('a')
z: Box[Box[int]] = Box(Box(1))
bad: Box[int] = Box[int]('a')
```

Types can depend on types!

Same principles apply: we can create **polymorphic functions**

```python
# Python 3.12
def singleton_int(x: int) -> list[int]:
    return [x]
def singleton_str(x: str) -> list[str]:
    return [x]

def singleton[a](x: a) -> list[a]:
    return [x]

x: list[int] = singleton(1)
y: list[str] = singleton('a')
bad: list[bool] = singleton(2)
```

Same principles apply: we can create **polymorphic functions**

```python
# Python 3.12
def singleton_int(x: int) -> list[int]:
    return [x]
def singleton_str(x: str) -> list[str]:
    return [x]


def singleton[a](x: a) -> list[a]:
    return [x]


x: list[int] = singleton(1)
y: list[str] = singleton('a')
bad: list[bool] = singleton(2)
```

Same implementation, different types

Same principles apply: we can create **polymorphic functions**

```python
# Python 3.12
def singleton_int(x: int) -> list[int]:
    return [x]
def singleton_str(x: str) -> list[str]:
    return [x]

def singleton[a](x: a) -> list[a]:
    return [x]

x: list[int] = singleton(1)
y: list[str] = singleton('a')
bad: list[bool] = singleton(2)
```

Make function receive type parameter

Same principles apply: we can create **polymorphic functions**

```python
# Python 3.12
def singleton_int(x: int) -> list[int]:
    return [x]
def singleton_str(x: str) -> list[str]:
    return [x]

def singleton[a](x: a) -> list[a]:
    return [x]

x: list[int] = singleton(1)
y: list[str] = singleton('a')
bad: list[bool] = singleton(2)
```

Same function returns different types!

Same principles apply: we can create **polymorphic functions**

```python
# Python 3.12
def singleton_int(x: int) -> list[int]:
    return [x]
def singleton_str(x: str) -> list[str]:
    return [x]

def singleton[a](x: a) -> list[a]:
    return [x]

x: list[int] = singleton(1)
y: list[str] = singleton('a')
bad: list[bool] = singleton(2)
```

Terms can depend on types!

Polymorphism in Haskell

In Haskell, types are capitalized, type parameters are lowercase

```
ghci> :t head
head :: [a] -> a

ghci> singleton x = [x]
ghci> :t singleton
singleton :: a -> [a]

ghci> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Huh?

```haskell
(.) :: x -> y -> e -> r
(.) g f x = g (f x)
```

Let us inspect the type signature of `(.)`

```haskell
(.) :: (d -> c) -> (a -> b) -> e -> r
(.) g f x = g (f x)
```

We know that `f` and `g` are some functions
Let their types be `a -> b` and `d -> c` for some `a`, `b`, `c`, `d` respectively

```haskell
(.) :: (d -> c) -> (a -> b) -> a -> r
(.) g f x = g (f x)
```

`f` has type `a -> b` so `x` must have type `a`

```
(.) :: (b -> c) -> (a -> b) -> a -> r
(.) g f x = g (f x)
```

`f x` has type `b` so `g` must have type `b -> c`

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) g f x = g (f x)
```

`g (f x)` has type `c` so `(.) g f x` must return `c`

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) g f x = g (f x)
```

Compiler does this automatically via **type inference**

Polymorphism

When to define polymorphic types/functions?

When **implementations** of classes/data types/functions are the **same** except the **types**

Example: container types!

```python
@dataclass
class IntTree:
    pass
@dataclass
class IntNode(IntTree):
    left: IntTree
    value: int
    right: IntTree
@dataclass
class IntLeaf(IntTree):
    value: int
```

```python
@dataclass
class StrTree:
    pass
@dataclass
class StrNode(StrTree):
    left: StrTree
    value: str
    right: StrTree
@dataclass
class StrLeaf(StrTree):
    value: str
```

Example: container types!

```python
@dataclass
class Tree[a]:
    pass
@dataclass
class Node[a](Tree[a]):
    left: Tree[a]
    value: a
    right: Tree[a]
@dataclass
class Leaf[a](Tree[a]):
    value: a
```

Example: generic operations on polymorphic types!

```python
def reverse_int(ls: list[int]) ->
list[int]:
    return [] if not ls else \
           reverse_int(ls[1:]) + [ls[0]]
```

```python
def reverse_str(ls: list[str]) ->
list[str]:
    return [] if not ls else \
           reverse_str(ls[1:]) + [ls[0]]
```

Example: generic operations on polymorphic types!

```python
def reverse[a](ls: list[a]) -> list[a]:
    return [] if not ls else \
           reverse(ls[1:]) + [ls[0]]
```

Polymorphism

Class/function should be polymorphic if implementation is **independent of type**

Algebraic Data Types

How do we create our own data types in Haskell?

A data type is usually composed of:

- Type **and** type **and** … **and** type
- Type **or** type **or** … **or** type

A data type is usually composed of:

- Type **and** type **and** … **and** type
- Type **or** type **or** … **or** type

- `Fraction`: numerator (`Int`) **and** denominator (`Int`)
- `Student`: name (`String`) **and** ID (`Int`)
- `Bool`: True **or** False
- `String`: Empty string **or** (head (`Char`) **and** tail (`String`))
- `Tree a`: (Leaf with value (`a`)) **or** (Node with value (`a`), left subtree (`Tree a`) **and** right subtree (`Tree a`))

Algebraic Data Types

Types made of products (**and**) and/or sums (**or**)

Algebraic Data Types

In Haskell, types are **sums** of one or more **constructors**; constructors are **products** of zero or more types

```haskell
-- LHS Fraction is the name of type
-- RHS Fraction is the name of constructor
data Fraction = Fraction Int Int
half :: Fraction
half = Fraction 1 2

-- Constructor name not necessarily same as type name
data Student = S String Int
bob :: Student
bob = S "Bob" 123
```

```haskell
-- Two constructors, each with zero types
data Bool = True | False
true, false :: Bool
true = True
false = False

-- Node is a constructor with two types
data String = EmptyString
            | Node Char String
cat, empty :: String
cat = Node 'c' (Node 'a' (Node 't' EmptyString))
empty = EmptyString
```

We can also create polymorphic data types!

```haskell
data Box a = Box a

data LinkedList a = EmptyList
                  | Node a (LinkedList a)

data Tree a = Leaf a
            | TreeNode (Tree a) a (Tree a)

x :: Box Int
x = Box 1
y :: LinkedList Char
y = Node 'a' (Node 'b' EmptyList)
```

Constructors are actually functions

```
ghci> data Fraction = F Int Int
ghci> :t F
F :: Int -> Int -> Fraction
ghci> :t F 1
F 1 :: Int -> Fraction
ghci> :t F 1 2
F 1 2 :: Fraction
```

Automatically create accessor functions of fields of data types by using **record syntax**

```
ghci> data Student = S { name :: String, id :: Int }
ghci> x = S { name = "Alice", id = 123 }
ghci> y = S "Bob" 456
ghci> name x
"Alice"
ghci> id y
456
```

Mix-and-match!

```haskell
data Department = D {name' :: String, courses :: [Course]}
data Course = C { code :: String,
                  credits :: Int,
                  students :: [Student] }
data Student = UG { homeFac :: String,
                    name :: String,
                    id :: Int }
             | PG [String] String Int
```

```haskell
alice    = UG "SoC" "Alice" 123
bob      = PG ["SoC", "YLLSoM"] "Bob" 456
it5100a = C "IT5100A" 2 [alice]
it5100b = C "IT5100B" 2 [alice, bob]
cs       = D "Computer Science" [it5100a, it5100b]
```

More on Polymorphism

Mental model for polymorphic functions/types:

Function-like thing that **quantifies over types**

Lambda Calculus

λ creates a function over a parameter

$$\lambda x. e$$

Calling the function substitutes x for the argument

$$(\lambda x. e_1)e_2 \equiv_\beta e_1[x \coloneqq e_2]$$

$$
\begin{aligned}
(\lambda x\colon \texttt{Int}.\, x + 4\,)3 &\equiv_\beta (x + 4)[x \coloneqq 3] \\
&\equiv_\beta (3 + 4) \\
&\equiv_\beta 7
\end{aligned}
$$

```
ghci> (\x -> x + 4) 3
7
```

System F

Polymorphic functions receive type parameter; can call with type argument (usually implicit in Haskell)

$$\Lambda\alpha.\, e$$

Calling the function with a **type** substitutes α for the argument

$$(\Lambda\alpha.\, e)\tau \equiv_\beta e[\alpha \coloneqq \tau]$$

$$
\begin{aligned}
(\Lambda\alpha.\,\lambda x{:}\,\alpha.\,[x])\mathtt{Int} &\equiv_\beta (\lambda x{:}\,\alpha.\,[x])[\alpha := \mathtt{Int}] \\
&\equiv_\beta (\lambda x{:}\,\mathtt{Int}.\,[x])
\end{aligned}
$$

```
ghci> :set -XTypeApplications -fprint-explicit-foralls
ghci> :{
ghci| f :: forall a. a -> [a]
ghci| f x = [x]
ghci| :}
ghci> :t f
f :: forall a. a -> [a]
ghci> :t f @Int
f @Int :: Int -> [Int]
ghci> f @Int 1
[1]
```

Polymorphic types can be seen as functions at the type-level: functions that receive types and return types

```
ghci> data Pair a b = P a b
ghci> :k Pair
Pair :: * -> * -> *
ghci> :k Pair Int
Pair Int :: * -> *
ghci> :k Pair Int String
Pair Int String :: *
```

Types like `Pair` are also known as **type constructors**

Every type has a kind, “normal types” have kind *, type constructors have kind * -> * etc

```
ghci> data Pair a b = P a b
ghci> :k Pair
Pair :: * -> * -> *
ghci> :k Pair Int
Pair Int :: * -> *
ghci> :k Pair Int String
Pair Int String :: *
```

Crazy: type constructors are like functions at the type level

Can we have higher-ordered type constructors?

```
ghci> :set -fprint-explicit-foralls
ghci> data Crazy f a = C (f a)
ghci> :k Crazy
Crazy :: forall {k}. (k -> *) -> k -> *
ghci> :k Crazy []
Crazy [] :: * -> *
ghci> :k Crazy [] Int
Crazy [] Int :: *
ghci> x :: Crazy [] Int = C [1]
ghci> data Box a = B a
ghci> y :: Crazy Box Int = C (B 2)
```

Moral of the story: Polymorphism is when something can receive a type and give you a type/term

1. Can we have types that depend on terms?
2. Are there other kinds of polymorphism?

1. Yes: dependent types
2. Yes: subtype (OOP), ad-hoc (overloading, next week)

ADTs in Python

Types are type declarations, constructors are classes

```
type List[a] = Node[a] | Empty

@dataclass
class Empty:
    pass

@dataclass
class Node[a]:
    head: a
    tail: List[a]

x: List[int] = Node(1, Node(2, Empty()))
```

Alternative formulation: types are classes, constructors are subclasses containing typed fields

```python
from typing import Any
@dataclass
class List[a]: pass

@dataclass
class Empty(List[Any]):
    pass

@dataclass
class Node[a](List[a]):
    head: a
    tail: List[a]

x: List[int] = Node(1, Node(2, Empty()))
```

Important Difference

Haskell: constructors are not types

Python: “constructors” are classes which are also types

Observation: subclass need not conform to declaration of superclass

```python
class List[a]: pass

class Empty(List[Any]):
    pass
```

Another example: expressions in a programming language

```
class Expr[a]:
    def eval(self) -> a:
        raise Exception
```

Create expressions that evaluate to **`int`s**; note the inheritance relationship

```python
@dataclass
class LitNumExpr(Expr[int]):
    n: int
    def eval(self) -> int:
        return self.n

@dataclass
class AddExpr(Expr[int]):
    lhs: Expr[int]
    rhs: Expr[int]
    def eval(self) -> int:
        return self.lhs.eval() + self.rhs.eval()
```

Expressions that evaluate to other things are also possible!

```python
@dataclass
class EqExpr[a](Expr[bool]):
    lhs: Expr[a]
    rhs: Expr[a]
    def eval(self) -> bool:
        return self.lhs.eval() == self.rhs.eval()
@dataclass
class CondExpr[a](Expr[a]):
    cond: Expr[bool]
    true: Expr[a]
    false: Expr[a]
    def eval(self) -> a:
        if self.cond.eval():
            return self.true.eval()
        return self.false.eval()
```

How do we customize our Haskell constructors to create specific specializations of polymorphic types?

Generalized Algebraic Data Types (GADTs)

```haskell
data LinkedList a where
   EmptyList :: LinkedList a -- this is a different a!
   Node :: b -> LinkedList b -> LinkedList b
```

```haskell
data Expr a where
   LitNumExpr :: Int -> Expr Int
   AddExpr    :: Expr Int -> Expr Int -> Expr Int
   EqExpr     :: Expr a -> Expr a -> Expr Bool
   CondExpr   :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Pattern Matching

Destructure data and match against patterns

`case` expression: match a term over a pattern

```haskell
fac :: Int -> Int
fac n = case n of -- match n against these patterns:
    0 -> 1
    x -> x * fac (x - 1) -- any other Int
```

Patterns can even destructure constructors!

```haskell
fst' :: (a, b) -> a
snd' :: (a, b) -> b
fst' p = case p of
    (x, _) -> x
snd' p = case p of
    (_, y) -> y

data Fraction = F Int Int
numerator :: Fraction -> Int
numerator f = case f of
    F x _ -> x
```

Even better: bring pattern matching up to definition

```haskell
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)

fst' :: (a, b) -> a
snd' :: (a, b) -> b
fst' (x, _) = x
snd' (_, y) = y

data Fraction = F Int Int
numerator :: Fraction -> Int
numerator (F x _) = x
```

List type declaration (sort of)

```haskell
data [a] = [] | a : [a]
```

Pattern match against different list constructors!

```haskell
sum' :: [Int] -> Int
sum' [] = 0
sum' (x : xs) = x + sum' xs

len :: [a] -> Int
len [] = 0
len (_ : xs) = 1 + len xs
```

You can use patterns in almost all bindings

```haskell
len :: [a] -> Int
len [] = 0
len ls =
    let (_ : xs) = ls
    in  1 + len xs
```

Compiler can catch non-exhaustive pattern matches

```haskell
-- Main.hs
emp :: [a] -> [a]
emp [] = []
```

```
$ ghc Main.hs
Main.hs:3:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘emp’: Patterns of type
    ‘[a]’ not matched: (_:_)
  |
3 | emp [] = []
  | ^^^^^^^^^^^
```

Important: terms are matched against patterns top-down

```haskell
fac :: Int -> Int
fac n = n * fac (n - 1)
fac 0 = 1 -- redundant as pattern above matches all
          -- possible integers
```

Can perform pattern matching against GADTs

```haskell
eval :: Expr a -> a
eval (LitNumExpr n)    = n
eval (AddExpr a b)     = eval a + eval b
eval (EqExpr a b)      = eval a == eval b
eval (CondExpr a b c) = if eval a then eval b else eval c
```

Can perform pattern matching against GADTs

```haskell
eval :: Expr a -> a
eval (LitNumExpr n)    = n
eval
eval
eval
```

```
$ ghc Main.hs
Main.hs:13:28: error:
    • Could not deduce (Eq a1) arising from a use of ‘==’
      from the context: a ~ Bool
        bound by a pattern with constructor:
                   EqExpr :: forall a. Expr a -> Expr a -> Expr Bool,
                 in an equation for ‘eval’
        at app/Main.hs:13:7-16
      Possible fix:
        add (Eq a1) to the context of the data constructor ‘EqExpr’
    • In the expression: eval a == eval b
      In an equation for ‘eval’: eval (EqExpr a b) = eval a == eval b
   |
13 | eval (EqExpr a b) = eval a == eval b
   |
```

Tell Haskell to enable the GADTs language extension

```haskell
{-# LANGUAGE GADTs #-}
data Expr a where
  LitNumExpr ::                Int -> Expr Int
  AddExpr    ::                Expr Int -> Expr Int -> Expr Int
  EqExpr     :: Eq a =>        Expr a -> Expr a -> Expr Bool
  CondExpr   ::                Expr Bool -> Expr a -> Expr a -> Expr a
```

Ensure that a can be compared for equality (covered next week)

Pattern Matching in Python

`match` block with `case` clauses

```python
def factorial(n: int) -> int:
  match n:
    case 0: return 1
    case n: return n * factorial(n - 1)
```

Pattern Matching in Python

`match` block with `case` clauses

```python
def sum(ls: list[int]) -> int:
    match ls:
        case []:
            return 0
        case (x, *xs):
            return x + sum(xs)
        case _:
            raise TypeError
```

Structural pattern matching is very useful in Python

```
@dataclass
class Tree[a]: pass

@dataclass
class Node[a](Tree[a]):
    val: a
    left: Tree[a]
    right: Tree[a]

@dataclass
class Leaf[a](Tree[a]):
    val: a
```

```
def preorder[a](tree: Tree[a]) -> list[a]:
    match tree:
        case Node(v, l, r):
            return [v] + preorder(l) \
                       + preorder(r)
        case Leaf(v):
            return [v]
        case _:
            raise TypeError
```

Exhaustiveness checks difficult; include catch-all case or use union types for ADTs

```python
type Tree[a] = Node[a] \
             | Leaf[a]
@dataclass
class Node[a](Tree[a]):
    val: a
    left: Tree[a]
    right: Tree[a]

@dataclass
class Leaf[a](Tree[a]):
    val: a
```

```python
def preorder[a](tree: Tree[a]) -> list[a]:
    match tree:
        case Node(v, l, r):
            return [v] + preorder(l) \
                       + preorder(r)
        case Leaf(v):
            return [v]
        # no need for further cases
```

When to use pattern matching?

`if .. then .. else ..`

Do different things based on **condition**

`case .. of ..`

Do different things based on **value/structure of data**

Thank you

Foo Yong Qi

yongqi@nus.edu.sg

https://yongqi.foo/