Haskell Crash Course Part II

Recap: Haskell Crash Course II

- Core program element is an **expression**
- Every *valid* expression has a **type** (determined at compile-time)
- Every *valid* expression reduces to a **value** (computed at run-time)
Recap: Haskell

Basic values & operators

- Int, Bool, Char, Double
- +, -, ==, /=

Execution / Function Calls

- Just substitute equals by equals

Producing Collections

- Pack data into tuples & lists

Consuming Collections

- Unpack data via pattern-matching
Next: Creating and Using New Data Types

1. type Synonyms: Naming existing types

2. data types: Creating new types
Type Synonyms

Synonyms are just names ("aliases") for existing types

- think typedef in C

A type to represent Circle

A tuple \((x, y, r)\) is a circle with center at \((x, y)\) and radius \(r\)

```haskell
type Circle = (Double, Double, Double)
```
A type to represent *Cuboid*

A tuple \((\text{length}, \text{depth}, \text{height})\) is a *cuboid*

```
type Cuboid = (Double, Double, Double)
```
Using Type Synonyms

We can now use synonyms by creating values of the given types

circ0 :: Circle
circ0 = (0, 0, 100) -- ^ circle at "origin" with radius 100

cub0 :: Cuboid
cub0 = (10, 20, 30) -- ^ cuboid with length=10, depth=20, height=30

And we can write functions over synonyms too
area :: Circle -> Double
area (x, y, r) = pi * r * r

volume :: Cuboid -> Double
volume (l, d, h) = l * d * h

We should get this behavior

>>> area circ0
31415.926535897932

>>> volume cub0
6000
Suppose we have the definitions

```haskell
type Circle = (Double, Double, Double)
type Cuboid = (Double, Double, Double)
```

circ0 :: Circle
circ0 = (0, 0, 100)  -- ^ circle at "origin" with radius 100

cub0 :: Cuboid
cub0 = (10, 20, 30)  -- ^ cuboid with length=10, depth=20, height=30

area :: Circle -> Double
area (x, y, r) = pi * r * r

volume :: Cuboid -> Double
volume (l, d, h) = l * d * h

What is the result of

```haskell
>>> volume circ0
```

A. 0

B. Type error
Beware!

Type Synonyms

- Do not create new types
- Just name existing types

And hence, synonyms

- Do not prevent confusing different values
Creating New Data Types

We can avoid mixing up by creating new `data` types

```
-- | A new type `CircleT` with constructor `MkCircle`
data CircleT = MkCircle Double Double Double

-- | A new type `CuboidT` with constructor `MkCuboid`
data CuboidT = MkCuboid Double Double Double
```

`Constructors are the only way to create values`

- `MkCircle` creates `CircleT`
- `MkCuboid` creates `CuboidT`
QUIZ

Suppose we create a new type with a `data` definition

```haskell
-- | A new type `CircleT` with constructor `MkCircle`
data CircleT = MkCircle Double Double Double Double
```

What is the type of the `MkCircle` constructor?

A. `MkCircle :: CircleT`

B. `MkCircle :: Double -> CircleT`

C. `MkCircle :: Double -> Double -> CircleT`

D. `MkCircle :: Double -> Double -> Double -> CircleT`

E. `MkCircle :: (Double, Double, Double) -> CircleT`
Constructing Data

Constructors let us build values of the new type

\[
\begin{align*}
circ1 & :: \text{CircleT} \\
circ1 & = \text{MkCircle 0 0 100} \quad -- ^\text{circle at "origin" w/ radius 100} \\
cub1 & :: \text{Cuboid} \\
cub1 & = \text{MkCuboid 10 20 30} \quad -- ^\text{cuboid w/ len=10, dep=20, ht=30}
\end{align*}
\]
QUIZ

Suppose we have the definitions

```haskell
data CuboidT = MkCuboid Double Double Double

type Cuboid = (Double, Double, Double)

volume :: Cuboid -> Double
volume (l, d, h) = l * d * h
```

What is the result of

```haskell
>>> volume (MkCuboid 10 20 30)
```

A. 6000

B. Type error
Deconstructing Data

Constructors let us build values of new type ... but how to use those values?

How can we implement a function

volume :: Cuboid -> Double
volume c = ???

such that

>>> volume (MkCuboid 10 20 30)
6000
Deconstructing Data by Pattern Matching

Haskell lets us deconstruct data via pattern-matching

\[
\text{volume} :: \text{Cuboid} \to \text{Double} \\
\text{volume}\ c = \text{case}\ c\ \text{of} \\
\hspace{1cm}\text{MkCuboid}\ l\ d\ h \to l * d * h
\]

\text{case}\ e\ \text{of}\ \text{Ctor}\ x\ y\ z \to e1\ is\ read\ as\ as

\text{IF} - e\ evaluates\ to\ a\ value\ that\ matches\ the\ pattern\ \text{Ctor}\ vx\ vy\ vz

\text{THEN} - evaluate\ e1\ after\ naming\ x := vx,\ y := vy,\ z := vz
Pattern matching on Function Inputs

Very common to do matching on function inputs

```haskell
volume :: Cuboid -> Double
volume c = case c of
    MkCuboid l d h -> l * d * h

area :: Circle -> Double
area a = case a of
    MkCircle x y r -> pi * r * r
```

So Haskell allows a nicer syntax: *patterns in the arguments*

```haskell
volume :: Cuboid -> Double
volume (MkCuboid l d h) = l * d * h

area :: Circle -> Double
area (MkCircle x y r) = pi * r * r
```

Nice syntax *plus* the compiler saves us from *mixing up* values!