Introduction to Haskell

Functional programming in Haskell

Ivan Trepakov

NSU Sys.Pro

What is Haskell?

What is Haskell?

Functional

- Functions as first-class citizens
- Higher order functions
- Declarative style

What is Haskell?

Functional

- Functions as first-class citizens
- Higher order functions
- Declarative style

Pure

- Side-effect separation
- Equational reasoning
- Simplified parallelism

What is Haskell?

Functional

- Functions as first-class citizens
- Higher order functions
- Declarative style

Lazy

- Infinite data structures
- Compositional programming style
- Tricky to evaluate complexity

Pure

- Side-effect separation
- Equational reasoning
- Simplified parallelism

What is Haskell?

Functional

- Functions as first-class citizens
- Higher order functions
- Declarative style

Pure

- Side-effect separation
- Equational reasoning
- Simplified parallelism

Lazy

- Infinite data structures
- Compositional programming style
- Tricky to evaluate complexity

Statically typed

- “If a program compiles, it probably works”
- Expressive type system
- Type inference

Basics

Anatomy of declaration

Here is sample Haskell declaration:

```
x :: Int    -- Type declaration
x = 42      -- Value declaration
```

`name = expression` is a *binding* (not assignment)

- `::` reads as “has type”
- `=` reads as “defined to be”

Anatomy of declaration

Here is sample Haskell declaration:

```
x :: Int    -- Type declaration
x = 42      -- Value declaration
```

`name = expression` is a *binding* (not assignment)

- `::` reads as “has type”
- `=` reads as “defined to be”

Multiple declarations with the same name are not allowed!

Compiler will let us know about it with error:

```
Multiple declarations of 'x'
```

Anatomy of declaration

Here is sample Haskell declaration:

```
x :: Int    -- Type declaration
x = 42      -- Value declaration
```

`name = expression` is a *binding* (not assignment)

- `::` reads as “has type”
- `=` reads as “defined to be”

Multiple declarations with the same name are not allowed!
Compiler will let us know about it with error:

```
Multiple declarations of 'x'
```

What does this declaration mean?
And what is its type if any?

```
y = y + 1
```

Anatomy of declaration

Here is sample Haskell declaration:

```
x :: Int    -- Type declaration
x = 42      -- Value declaration
```

`name = expression` is a *binding* (not assignment)

- `::` reads as “has type”
- `=` reads as “defined to be”

Multiple declarations with the same name are not allowed!
Compiler will let us know about it with error:

```
Multiple declarations of 'x'
```

What does this declaration mean?
And what is its type if any?

```
y = y + 1
y :: Int
```

Built-in types

Built-in types

```
-- Fixed-precision integer
i :: Int
i = 12
```

Guaranteed[1] to be at least $[-2^{29}, 2^{29} - 1]$, but usually is machine word sized

```
-- Actual bounds
minInt, maxInt :: Int
minInt = minBound
maxInt = maxBound
```

[1] See Haskell 2010 Language Report, Section 6.4 Numbers

Built-in types

```haskell
-- Fixed-precision integer
i :: Int
i = 12
```

Guaranteed[1] to be at least $[-2^{29}, 2^{29} - 1]$, but usually is machine word sized

```haskell
-- Actual bounds
minInt, maxInt :: Int
minInt = minBound
maxInt = maxBound
```

```haskell
-- Arbitrary-precision integer
n :: Integer
n = 2 ^ (2 ^ (2 ^ (2 ^ 2)))

numDigits :: Int
numDigits = length (show n)

-- >>> numDigits
-- 19729
```

[1] See Haskell 2010 Language Report, Section 6.4 Numbers

Built-in types

```
-- Double-precision floatint point
d1, d2 :: Double
d1 = 3.1415
d2 = 6.2831e-4

-- Boolean
b1, b2 :: Bool
b1 = True
b2 = False
```

Built-in types

```haskell
-- Double-precision floatint point
d1, d2 :: Double
d1 = 3.1415
d2 = 6.2831e-4

-- Boolean
b1, b2 :: Bool
b1 = True
b2 = False
```

```haskell
-- Unicode code point (character)
c1, c2, c3 :: Char
c1 = 'A'
c2 = 'λ'
c3 = '🌎'

-- String (list of characters)
s :: String
s = "Hello world! 🌎"
```

Q&A