Haskell Intro

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

Learning Group Activity

With your learning group, share answers from the learning group activity. If you did Question 6, collaborate with the other person who did Question 6 to give a collective tutorial on GHCi to your group (should be easy).

Blake will go around and check participation. If you have questions on the LGA (and your group members cannot answer), ask Blake. If he cannot answer, feel free to Email me or the list.

Background on Functional Languages

Functional vs. Imperative Languages

- **Imperative Language:** Typical programming language you are already used to (like C or C++). Sequence of instructions is executed to change the *state* of the program.
- **Pure Functional Language:** Functions main unit of programming. Functions are **referentially transparent**; that is, every time you call a function with the same arguments, you are *guaranteed* the same result (no dependence on state).
- **Immutible structures** are typically used in functional languages to make this possible (e.g., once you create a list you cannot change it).
- Easy to *prove* functions correct (contrast to imperative languages)
- Haskell is a purely functional language.

Lazy Evaluation

In Haskell, expressions are evaluated **lazily**: parts of expressions are not evaluated until they need to be.

Example

Suppose you wanted to **determine the length** of this list:

```
[12 + 3, 4 / 2, 2 - 15, 5]
```

You could determine this two ways:

1. You could evaluate the whole list first, and get this result:

   ```
   [15, 2, -13, 5]
   ```

   Then, you can see the list has a length 4. This is **eager evaluation**.
2. Or, you could see the list has four elements, without preforming any of the computations. This is called **lazy evaluation**.

Lazy Evaluation: What Does It Give Us?

Under lazy evaluation, we can have:

- Expressions evaluated without wasting time on parts that are not needed to evaluate.
- **Lists of infinite length**! Think about a list that defines all of the Fibonacci numbers.
- **Immutible structures** which act efficiently. Suppose you had a function `doubleAll` which took a list and returned a *new copy* with each of the elements multiplied by 2.
  - **Eager:** `doubleAll(doubleAll(doubleAll(lst)))` would create 3 new lists
  - **Lazy:** same computation can act on a new list only at the very end

Haskell says... meh, I'll do it later!

Lazy Evaluation can differ from Eager?

Is it possible for a lazy evaluation to differ from an equivalent eager evaluation?

Lazy Evaluation can differ from Eager?

Is it possible for a lazy evaluation to differ from an equivalent eager evaluation?

Yes, in error handling:

- Under eager evaluation, determining the length of `[3, 1/0]` would result in an error, whereas in lazy evaluation, the length would be 2.

Starting with Haskell

GHCi

Note

You are probably best paying attention to lecture, then going home and replicating this. Up to you.

To load up GHCi, type `ghci` at your terminal. You will need Haskell installed on your computer, or use the ALAMODE Linux Lab (BB 136) machines. You should be left at a `Prelude>` prompt if you did that correctly:

```
GHCi, version 8.2.2: http://www.haskell.org/ghc/
:? for help
Prelude>
```

This is a REPL (Read-Evaluate-Print-Loop). GHCi reads what you type, the result is evaluated, GHCi prints the result, looping until exit.

Infix and Prefix Operators

Haskell comes with two kinds of operators: **infix** and **prefix**.

- **Infix operators** are placed **in-between** the operands:

```
GHCi> 4 + 4
8
```

- **Prefix operators** are placed **before** the operands:

```
GHCi> mod 10 3
1
```

Infix and Prefix Operators

Haskell comes with two kinds of operators: **infix** and **prefix**.

- **Infix operators** are placed **in-between** the operands:

```
GHCi> 4 + 4
8
```

- **Prefix operators** are placed **before** the operands:

```
GHCi> mod 10 3
1
```

Postfix Operators

As you may have guessed, postfix operators are operators placed *after* the operands, but Haskell does not have any of these.

Using Prefix Operators as Infix (and visa-versa)

- You can use prefix operators as infix by surrounding them in **backticks**:

```
GHCi> 10 `mod` 3
1
```

- Likewise, you can use infix operators as prefix by surrounding them in **parenthesis**:

```
GHCi> (+) 4 4
8
```

Parenthesis Denote Order of Operation

```
GHCi> (2 * 50) + 1
101
GHCi> 2 * (50 + 1)
102
GHCi> 3 + 4 * 7
31
GHCi> (3 + 4) * 7
49
```

Parenthesis Denote Order of Operation

```
GHCi> (2 * 50) + 1
101
GHCi> 2 * (50 + 1)
102
GHCi> 3 + 4 * 7
31
GHCi> (3 + 4) * 7
49
```

Careful! You need to surround negative numbers in parenthesis when combined with other expressions (- is a prefix operator)

```
GHCi> 5 * (-3)
-15
```

Boolean Operators

Haskell comes with a standard set of boolean operators:

- `&&`: and (infix)
- `||`: or (infix)
- `not`: not (prefix)

Boolean Operators

Haskell comes with a standard set of boolean operators:

- `&&`: and (infix)
- `||`: or (infix)
- `not`: not (prefix)

and some infix equality tests:

- `==`: equal
- `/=`: not equal
- `>`: greater than
- `>=`: greater or equal
- ... same for less

Boolean Operators

Haskell comes with a standard set of boolean operators:

- `&&`: and (infix)
- `||`: or (infix)
- `not`: not (prefix)

```
GHCi> True && False
False
GHCi> not False
True
GHCi> "hello" /= "goodbye"
True
```

and some infix equality tests:

- `==`: equal
- `/=`: not equal
- `>`: greater than
- `>=`: greater or equal
- ... same for less

Calling Functions

To call a function, write the function name, with arguments following, separated by spaces.

```
GHCi> max 9 7
9
```

Calling Functions

To call a function, write the function name, with arguments following, separated by spaces.

```
GHCi> max 9 7
9
```

Calling functions takes **highest precedence**:

```
GHCi> max 9 7 + 3
10
```

Perhaps this is what you wanted:

```
GHCi> (max 9 7) + 3
12
```

Defining Functions

Open a file `clocktools.hs` and write this function in it:

```
subtract45 m = mod (m - 45) 60
```

This defines a function named `subtract45` that takes a number of minutes `m`, and computes what subtracting 45 minutes from this number would lead to, without going below 0 or above 59.

Defining Functions

Open a file `clocktools.hs` and write this function in it:

```
subtract45 m = mod (m - 45) 60
```

This defines a function named `subtract45` that takes a number of minutes `m`, and computes what subtracting 45 minutes from this number would lead to, without going below 0 or above 59.

Load the file in GHCi using `:l`:

```
GHCi> :l clocktools
GHCi> subtract45 15
30
```

Multiple Arguments

Let's solve the general case of subtracting n minutes! Add to `clocktools.hs`:

```
subtractMinutes n m = mod (m - n) 60
```

Multiple Arguments

Let's solve the general case of subtracting n minutes! Add to `clocktools.hs`:

```
subtractMinutes n m = mod (m - n) 60
```

```
GHCi> :r
GHCi> subtractMinutes 10 5
55
```

Multiple Arguments

Let's solve the general case of subtracting n minutes! Add to `clocktools.hs`:

```
subtractMinutes n m = mod (m - n) 60
```

```
GHCi> :r
GHCi> subtractMinutes 10 5
55
```

```
GHCi> (subtractMinutes 10) 5
55
```

Whoa, what happened there?

Currying

- Haskell takes advantage of **currying** to support functions with multiple arguments. That is, functions take a single argument and return a function ready to take the next argument.
- We call the function ready to take the next argument a **partially applied function**.

What can we do with this?

Currying

- Haskell takes advantage of **currying** to support functions with multiple arguments. That is, functions take a single argument and return a function ready to take the next argument.
- We call the function ready to take the next argument a **partially applied function**.

What can we do with this?

```
subtract45 = subtractMinutes 45
```

Whoa. Is that concise, expressive, or both?

End of Lecture: Roadmap

Roadmap

- Blake lecturing again on Tuesday.
- Divide up the learning group assignment amongst your group **before you leave class**.
- Any questions? If they cannot be answered here, feel free to send to mailing list.