Week 2, Lecture 1: Functions = Values

Today's Topics

- Referential transparency
- Lazy evaluation
- Operators
- Currying
- Higher-order functions
- Composition

Recap: Expressions, Functions, and Functional Programming

Some vocabulary

- **expression (noun)**: a "piece of computation" (for example, 1 + 2 or f 3)
- **evaluate (verb)**: try to compute the result of an expression
  - The evaluation of an expression can terminate with a result, or it can diverge (run forever).
- **side effect (noun)**: during evaluation, a side effect is something that changes state
  - Examples: modifying the value of a variable, printing a message to the screen
- A sub-expression is **referentially transparent** if it can be replaced with its result without affecting the result of the surrounding expression.
- A "pure" expression has no side effects.
  - All pure expressions are referentially transparent.
- In a purely functional programming language (such as Haskell), computation is equivalent to evaluating only pure expressions.

Because Haskell is a purely functional programming language:

- All "variables" are constant.
  - Because we are not allowed to have side effects.
- Programs can be easier to reason about.
  - Because we are not allowed to have side effects.
- Without side effects, we can't* print things out (!!)
  - *Actually, we can, but it is going to take us a while before we can understand how.

Referential transparency allows us to write programs in a new way: lazily!

Referential Transparency Referential transparency is an important property in functional programming. It means that you can replace a referentially transparent expression with its result without affecting the behavior of the program. This makes it easier to reason about the program and enables optimizations, such as lazy evaluation.

Lazy Evaluation Lazy evaluation is an evaluation strategy used in Haskell where expressions are only evaluated when their values are needed. This can lead to more efficient programs, as computations that are not required for the final result will not be performed.

Operators In Haskell, operators are functions that can be used in infix notation. For example, `+`, `*`, and `/` are operators. You can also create your own operators by enclosing a sequence of symbols in backticks, like `` `myOperator` ``.

Currying Currying is a technique in functional programming where a function that takes multiple arguments is transformed into a series of functions that each take a single argument. In Haskell, all functions are curried by default.

Higher-order Functions Higher-order functions are functions that take other functions as arguments or return functions as results. They are a powerful tool in functional programming, enabling complex behavior to be built up from simpler functions.

Composition Function composition is a technique in functional programming where the output of one function is used as the input for another function. In Haskell, you can use the `(.)` operator to compose functions, such as `f . g`, which means "apply the function `g` first, then apply the function `f` to the result of `g`."

Lazy Evaluation

Some vocabulary

- **eager (or strict) evaluation (noun)**: evaluate subexpressions first
  - This is the form of computation that is most familiar to you.
  - With eager evaluation:
    - expressions are evaluated before they are assigned
      - `x = 1 + 2` $\rightarrow$ `x = 3`
    - arguments are evaluated before the function call
      - `f(1 + 2, 3 * 4)` $\rightarrow$ `f(3, 12)`
- **lazy evaluation (noun)**: evaluate subexpressions only when needed
  - With lazy evaluation:
    - expressions are not evaluated before they are assigned
      - `x = 1 + 2` (x is bound to the expression `1 + 2`)
    - arguments are not evaluated before the function call

      -
        ```
        ignoreFirstArgument :: Integer -> Integer -> Integer
        ignoreFirstArgument x y = y
        ignoreFirstArgument (1 + 2) (3 * 4)
        $\rightarrow$ 3 * 4
        ```
      - x is bound to the expression `1 + 2`

Brief tangent: wildcard patterns

-
  ```haskell
  ignoreFirstArgument :: Integer -> Integer -> Integer
  ignoreFirstArgument _ y = y
  ```
  - wildcard pattern: Any value will match, but the function body does not plan to reference the value.

Some benefits of lazy evaluation

- No unnecessary computation.
  - The program evaluates only what is needed.
- Fewer redundant computations.
  - Thanks to referential transparency, Haskell can "remember" the results of some computations.
- Evaluation can happen in parallel!
  - Without side-effects, independent expressions can be evaluated in any order.
- We can easily write efficient code that operates over large (even potentially infinite!) data structures.
  - The program can construct only the part of the data structure that needs to be evaluated, not the entire structure.

Operators

In Haskell, operators are essential for working with expressions. To understand how they work, we need to know some vocabulary:

- Operator: A symbol used to perform a specific operation (e.g., +, -, *, /).
- Operand: The values the operator works with.
- Application: The use of an operator on operands (e.g., 1 + 2).
- Precedence: Determines the order in which operators are applied in an expression.
- Associativity: Resolves ambiguity when multiple applications of the same operator are present in an expression.

In Haskell, operators have specific precedence and associativity. For example:

```
5 - 1 - 2 * 2
```

This expression is interpreted as `(5 - 1) - (2 * 2)` because the `*` operator has higher precedence than `-`. The `-` operator is left-associative, which means that the left-most application is evaluated first.

Haskell has several predefined operators, including:

Operation name(s)	Operator(s)	Associativity
function application		left
exponentiation	^, **	right
multiplication, division	*, /	left

Operation name(s)	Operator(s)	Associativity
(in)equality	==, /=, <, >, <=, >=	left
logical and	&&	left
logical or		

Function applications have the highest precedence, allowing expressions like `show 4 ++ show 2` instead of `(show 4) ++ (show 2)`.

Operators are just functions in Haskell. A binary operator is a function that takes two arguments. To use an operator as a function, place it in parentheses, like `1 + 2` can be written as `(+) 1 2`. Similarly, any two-argument function can be an operator using infix notation. Surround the function with back tick marks (usually on the tilde key to the left of the 1 and below the escape key on most keyboards) and place the arguments on either side, like

```
4 `div` 2
```

To define custom operators, use fixity declarations. A fixity declaration sets the associativity and precedence for the operator. For example:

```
infixl 7 `addMod60`

addMod60 :: Integer -> Integer -> Integer
addMod60 x y = (x + y) `mod` 60
```

This creates a new operator `addMod60` with a precedence level of 7 and left associativity. Fixity declarations help create more expressive and readable code in Haskell.

Currying and Higher-order Functions

In Haskell, all functions take only one argument. This may seem counterintuitive, but it's due to a concept called currying. Currying allows multi-argument functions to be represented as a series of single-argument functions. For example, consider the `average` function:

```
average :: Double -> Double -> Double
average x y = (x + y) / 2
```

We can interpret the type of `average` in two ways:

1. `average` is a function that takes two `Double` values and returns a `Double`.
2. `average` is a function that takes a `Double` and returns a function that takes a `Double` and returns a `Double`.

Function declaration (`->`) is an operator on types, with the left operand being the type of the parameter and the right operand being the type of the result. It is right associative.

Currying is named after Haskell Curry, a mathematician and logician who (re)discovered the concept, along with Moses Schönfinkel. Curried functions are multi-argument functions that take their arguments one at a time.

Higher-order functions are functions that take other functions as input or return other functions as output. A programming language that supports higher-order functions is said to have first-class functions because functions are also values in that language.

For example, the `increment` function:

```
increment :: Integer -> Integer
increment n = 1 + n
increment n = (+) 1 n
increment n = ((+) 1) n
increment n = (+ 1) n
increment = (+ 1)
```

Function composition is a powerful technique for combining simple functions to create more complex ones. Consider these functions:

```
increment :: Integer -> Integer
increment = (+ 1)
double :: Integer -> Integer
double = (* 2)
square :: Integer -> Integer
square = (^ 2)
```

Function composition allows you to create new functions by combining existing ones:

```
increment (double 1)
double (increment 1)
increment (square 1)
```

Function composition can be represented as an operator, spelled `.` in Haskell:

```
(increment . double) 1
(double . increment) 1
(increment . square) 1
```

This allows for concise and expressive code. In summary, currying and higher-order functions are essential concepts in Haskell, enabling elegant function composition and powerful programming techniques.