FunStructs - A *Fun*ctional Data Structure Library for Java

Adam Burke

November 2021

Contents

1 About

*Fun*Structs provides *fun*ctional implementations of commonly used data structures in a highly object-oriented fashion. Taking advantage of inheritance, composition, and polymorphism, we can organize these data structures in a hierarchy that promises new and domain-specific functionality at each level. What we are left with is a lightweight Java library that greatly speeds up the development process by providing high-level abstractions over these data structures that make the most fundamental operations a cinch, allowing developers to

build full-fledged programs with minimal code, and little to no code duplication.

But how can we provide abstract functionality in an object-oriented setting?

Good question! I see you've been reading closely. This is where the *fun*part comes in. We borrow from the world of *fun*ctionalprogramming by using *Lambda Abstraction* to parameterize our functions that operate on data structures.

A Quick Note About Lambda Abstraction

I'm a big fan of Lambda, and I think you should be too. The Lambda calculus was invented by Alonzo Church. It is an incredibly small programming language that can define any complex computer program, however verbose. It serves as an alternative to the Turing Machine, and describes programs as a black box with inputs and outputs as opposed to Turing's state-based, mutable model. A theoretical function `func` takes 0 or more inputs $\{i_0, i_1, i_2, ..., i_n\}$, for $n \in Z+$, and produces an output o. We then say that `func` is *pure* if `func` does not mutate the states of i_k, for $k \in [0, n] \subseteq Z$. If `func` is *pure*, then we also know that o will always be the same for a fixed set of i_k. This makes our *fun*ctionalabstractions easy to test, and minimizes bugs, since behavior is always fully predictable based on inputs. There is no publicly-exposed state-based computation, and my testing ensures that all internal state-based computation is predictable in the scope of public-facing behavior.

Another side effect of writing *fun*ctional-oriented code in java is that all of my methods (or *fun*ctions, rather), return something. This may seem like a subtle distinction, but it makes a huge difference. There is no concept of `void` signatures in this library, so every function returns exactly one output. Where normal Java might provide a method that adds a number to the end of a list by mutating the original list to include the new number at the end, my *fun*ctionalcode would return a copy of that list, with the new element added at the end, allowing the output to be clearly predicted from the input and function documentation.

Conventions, Function Syntax, and More Nerd Stuff

To prevent early onset arthritis, I'll go ahead and formalize some shorthand so we can be on the same page.

- $A \equiv B$ means that A and B are logically, structurally, or linguistically equivalent, for statements A and B. When A and B have finite truth value, one can assume that logical equivalence is being denoted. When A and B are expressions representing data structures, one can assume that structural equivalence is denoted. When A and B are definitions with indeterminate truth value, one can assume that linguistic equivalence is denoted, and A and B have the same meaning. The context of this

statement will likely always clearly show which equivalence is denoted, but in exceptional circumstances we will explicitly state it. For example, $p \to q \equiv \neg p \vee q$, $\{x \in Z : 2|x\} \equiv \{x \in Z : x \equiv 0(mod 2)\}$, and "The black dog" $\equiv$ "The dog whose fur contains no color value", show these three types of equivalence, respectively.

- $[A := B] \equiv$ A is defined to be equal to B

`func` :: $I_0, I_1, I_2, ..., I_n \to O$ $\equiv$ a function named `func` that takes n inputs of type I_0, I_1, I_2, and so on, and outputs a value of type O. We call this the *signature* of `func`. For example, the function +, which adds two `Integer`s has the signature + :: $Integer$, $Integer \to Integer$

- When we want to represent an *Anonymous Function*, i.e. a Lambda with no specific name, we can then convert the definition of any function to an equivalent *lambda expression*. For a named function + :: $Integer$, $Integer \to Integer := +(i_1, i_2) = i_1 + i_2 \; \forall i_1 \in Z \; \forall i_2 \in Z$, we can convert that definition of + to an *anonymous function* using the *lambda syntax*: $\lambda i_1.\lambda i_2.i_1 + i_2$, where for a general function `func`, the lambda syntax appears as such: $\lambda i_0.\lambda i_1.\lambda i_2...\lambda i_n. * \textit{some computation on all } i_k *$. Writing so many λ's is tedious, so we instead use the *curried lambda syntax*: $\lambda i_0.\lambda i_1.\lambda i_2...\lambda i_n. \equiv \lambda i_0 i_1 i_2...i_n$ for functions with an arity greater than 1.

2 Build and Installation

3 Usage

4 Functionality

4.1 *Fun*Graph

A *fun*ctionalGraph interface, for abstract operations over the most general kind of data structure

4.2 *Fun*Tree

4.3 *Fun*List

A List is a directed graph that flows in a single direction. *Fun*List provides two implementations of lists. One is a recursively defined list, and one is an iterative list. The recursively defined list, i.e. a Linked List consists of a series of nodes, each with a reference to the next node, or some stopping point, i.e. the empty list. We define this generalized list L_X of some type X as such:

$$L_X := \{X, L_X\}$$

This recursive definition lets us define linked lists that recurse until the innermost list is $\emptyset$. We then can define the list $\{1, 2, 3, 4, 5\}$ as such:

$$\{1, \{2, \{3, \{4, \{5, \emptyset\}\}\}\}\}$$

The iterative list is a container class for a java `List`. Both implementations are strongly typed over some type `X`, meaning that a `FunList<Integer>` cannot contain some `String, Character,` or `FunList<Integer>` member, for example.

*Fun*List exposes the following functions:

- `<Y> FunList<Y> map(Function<X,Y> mapperFunc)`

  ### Purpose

  Returns a new `FunList<Y>` of equal size to the calling `FunList<X>` where each element $y_i =$ `mapperFunc`(x_i), for $i \in [0,$ `this.size()`$] \subseteq Z$

  ### Mathematical Representation

  `map` :: `FunList<X>`, `Function<X,Y>` $\rightarrow$ `FunList<Y>`
  `map`$(L_x, \lambda x.f(x)) := \{y_i \in Y : y_i = (\lambda x.f(x))x_i \, \forall x_i \in L_x \}, \forall i \in [0, L_x$`.size()`$] \subseteq Z \; \forall f ::$ `X` $\rightarrow$ `Y`

  ### Examples

  - $\{1, 2, 3\}$`.map`$(\lambda x.x + 1) \rightarrow \{2, 3, 4\}$
  - $\{$"a","bc","def"$\}$`.map`$(\lambda x.$`length(x)`$) \rightarrow \{1, 2, 3\}$
  - $\{\{$"bob","pop"$\}, \{$"$noon$","$deed$","$peep$"$\}, \{$"$level$","$radar$","$madam$","$kayak$"$\}\}$`.map`$(\lambda lx.$`revers`

4.4 *Fun*Matrix

5 Design

6 Testing

7 Examples

8 Education

9 Appendix