CIS 343 - Structure of Programming Languages

Nathan Bowman

Based on slides by Ira Woodring

Functional Programming Languages and Scheme Introduction (Follows the Sebesta Text Chapter 15)

Overview

Imperative languages are all based on the von Neumann architecture. These are by far the most used languages.

Instead of the von Neumann architecture, functional languages are based on a mathematical model of computing.

Overview

There have been many important computer scientists that have suggested functional programming might be a better alternative to imperative programming. John Backus, for instance, was a proponent of these languages.

In 1977 he won the ACM Turing Award for his work on Fortran. With this award is given an opportunity to present a lecture to the community. He lectured on the merits of functional programming. He noted that this paradigm:

Overview

- lacks side effects
- isn't affected by context
- (due to the above mentioned reasons) is easier to understand during and after development

Overview

Imperative languages involve the idea of state. As commands are executed state changes.

The solution to a problem is the memory's final state.

This is extremely hard to read, as we need to keep track of variables' state as we peruse the code.

This is not a problem in functional programs, as they have no state.

Overview

An example of a mathematical function is

```
f(x) = x^2 + 3*x + 2
```

Functions do not change. With this definition,

`f(2) -> 12`

always, no matter what

That makes it much easier to work with compared to a subroutine that has, for example, a static variable

Overview

It may not sound all that appealing to talk about what functional languages *don't* do (although referential transparency is great for readability and reliability)

One thing functional languages do well is **higher-order functions**

These are functions that

- take functions as parameters,
- return functions,
- or both

Overview

Higher-order functions give us a new way to decompose problems and create even more modular code

Modularity is great! Modular code is easier to write, maintain, and reuse

John Hughes wrote a famous paper called "Why Functional Programming Matters" that lays out some benefits of functional programming and nifty things you can do with it -- I encourage you to check it out if this introduction piques your interest at all

Overview

You have seen examples of higher-order functions in the past -- as with any great idea, various programming languages have picked up on it, even non-functional ones

Think about passing a comparison routine to a sorting routine

```
def compare(A, B):
    return A.name[0] < B.name[0]:

sort(students, compare)
```

Overview

This is great, because it allows us to write the `sort` once and allow it to sort practically anything (reuse!)

The `sort` function and `compare` function have essentially nothing to do with one another, so it makes sense to keep them separate (modularity!)

This would be hard to do without functions that take functions

Overview

While studying functional languages you will come across the concept of the **lambda expression**.

(If you remember back to our study of Python you may have picked up on this!)

This is based on the work of Alonzo Church, who was working on a formal model of functions called **lambda calculus**.

Overview

Lambda expressions are nameless functions. In lambda calculus a cube function would be

$$\lambda(x)x * x * x$$

In Python we expressed it as:

```
lambda x: x*x*x
```

We will see how these can be useful momentarily

Overview

We can also **compose** functions to form a new function

For example, we can rewrite the absolute value function as

```
|x| = sqrt(x^2)
```

If we already had functions for squaring and square roots, we could could write a new absolute value function by composing the other two

Overview

```
|x| = sqrt(x^2)

|x| = g(f(x)) where `f(x) = x^2` and `g(x) = sqrt(x)
```

Thinking of `g(f(x))` as a single function `(g.f)(x)`, the new `(g.f)` is a composition

In this case, we have created an absolute value function that we can reuse from two other functions we already had

Overview

In terms of code, this would be

```
new_abs = lambda x: math.sqrt(x**2)
```

new_abs is now a function of a single variable that returns the absolute value

This is one reason functional programming is so modular -- we make new functions by combining our old functions in useful ways

Overview

Recap:

- Functional languages are readable and reliable to due referential transparency
- We will make heavy use of higher-order functions
- Higher-order functions allow us to write more modular code, which is a big win