Functional Programming Paradigm CS315

Joie Ann M. Mac

Reference

- Chapter 15 – Functional Programming Languages
  - Concepts of Programming Languages by Sebesta

Topics

- Introduction
- Mathematical Functions
- Fundamentals of Functional Programming Languages
- The First Functional Programming Language: LISP
- Support for Functional Programming in Primarily Imperative Languages
- Comparison of Functional and Imperative Languages

Imperative vs Functional

- The design of the imperative languages is based directly on the *von Neumann architecture*
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on *mathematical functions*
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Mathematical Functions

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*
- Function definitions are often written as a function name, followed by a list of parameters in parentheses, followed by the mapping expression

  `cube(x)≡x*x*x, where x∈R`

- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

  `λ(x) x * x * x`

for the function `cube(x)=x*x*x`

Lambda Expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

  e.g., `(λ(x) x * x * x)(2)`

  which evaluates to `8`

More examples:

Square of a number

- Function form: square(x) ≡ x * x
  - This means the function takes a number x and returns x squared (i.e., multiplied by itself).
- Lambda expression: λ(x) x * x
  - This means "take x and return x squared.“

Add two numbers

- Function form: add(x, y) ≡ x + y
  - This function takes two numbers, x and y, and returns their sum.
- Lambda expression: λ(x, y) x + y
  - This means "take x and y, and return their sum."

Functional Forms

- A higher-order function, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both
- Common kinds:
  - function composition
  - apply-to-all

Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

  Form: $h \equiv f \circ g$

  which means $h(x) \equiv f(g(x))$

  For $f(x) \equiv x + 2$ and

  $g(x) \equiv 3 * x$,

  $h \equiv f \circ g$ yields

  $(3 * x) + 2$

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

  Form: α

  For `h(x) ≡ x * x`

  `α(h, (2, 3, 4))` yields `(4, 9, 16)`

Fundamentals of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
  - In an imperative language, operations are done and the results are stored in variables for later use
  - Management of variables is a constant concern and source of complexity for imperative programming

Fundamentals of Functional Programming Languages

- In an FPL, variables are not necessary, as is the case in mathematics
- Iterative constructs are not possible
  - alternative: **recursion**
- Programs are function definitions and function application specifications
  - execution consists of evaluating function applications

Fundamentals of Functional Programming Languages

- **Referential Transparency**
  - in an FPL, the evaluation of a function always produces the same result given the same parameters
  - in purely functional programming language, semantics are far simpler

Fundamentals of Functional Programming Languages

- Functional languages provide:
  - a set of primitive functions
  - a set of functional forms to construct complex functions
  - structure to represent data

Fundamentals of Functional Programming Languages

- First functional language: LISP
  - **LIS**t **P**rocessing
  - syntax used for data and code not similar to imperative languages
- Other functional languages
  - Haskell
  - Scheme
  - Common LISP
  - ML
  - F#

Support for Functional Programming in Primarily Imperative Languages

- Support for functional programming is increasingly creeping into imperative languages
- Most important restriction:
  - support for higher-order functions

Support for Functional Programming in Primarily Imperative Languages

```javascript
function add(a, b) {
    return a + b;
}
```

Regular Function
Definition:

Support for Functional Programming in Primarily Imperative Languages

- To create an **anonymous function** in JavaScript, you simply remove the function's name:

javascript

```javascript
(function(a, b) {
    return a + b;
});
```

javascript

```javascript
let add = function(a, b) {
    return a + b;
};
console.log(add(2, 3)); // Output: 5
```

Anonymous Function (Lambda Expression):

Support for Functional Programming in Primarily Imperative Languages

- Anonymous functions (lambda expressions)
  - In C#, the syntax for a lambda expression is:

csharp

```csharp
(parameter) => expression
```

Support for Functional Programming in Primarily Imperative Languages

- Anonymous functions (lambda expressions)
  - the lambda expression:

```
i => (i % 2) == 0
```

  - checks whether a number is even or odd
  - returns true or false depending on whether the parameter is even or odd

Support for Functional Programming in Primarily Imperative Languages

- Python supports the higher-order functions `filter` and `map` (often use lambda expressions as their first parameters)

```
map(lambda x : x ** 3, [2,4,6,8])
```

- Returns `[8, 64, 216, 512]`

Comparing Functional and Imperative Languages

- Functional Languages:
  - Simple syntax
  - Simple semantics
  - Less efficient execution
- Imperative Languages:
  - Complex syntax
  - Complex semantics
  - Efficient execution

Comparing Functional and Imperative Languages

- Readability:
- C (Imperative)

```
int sum_cubes(int n){
   int sum = 0;
      for(int i = 1; i <= n; i++)
         sum += i * i * i;
      return sum;
  }
```

- Haskell (Functional)

```
sumCubes n = sum (map (^3) [1..n])
```

Comparing Functional and Imperative Languages

- Readability:
- C (Imperative)

```
int sum_cubes(int n){
    int sum = 0;
        for(int i = 1; i <= n; i++)
            sum += i * i * i;
        return sum;
  }
```

- Haskell (Functional)

```
sumCubes n = sum (map (^3) [1..n])
```

Attendance:

https://forms.gle/CSorA
tJCbzSLLpUZ6