Programming Paradigms

CT331 Week 6 Lecture 1

Finlay Smith

Finlay.smith@universityofgalway.ie

Functional Programming

Functional Programming

Given the same problem to solve, a program for this problem in any programming language can be considered equivalent to any other at machine level, in that the programs will result in changes to values contained in memory cells.

However, there can be quite significant differences at conceptual and implementation level.

Functional Programming

Conceptual … various types of programming languages are better suited to solving particular problems.

Functional Programming

Consider Building an embedded system:

- **Assembly:** Fast. Lightweight. Memory efficient. Use of architecture specific commands.
- **C:** Fast. Lightweight. Memory efficient.
- **Python:** Too much overhead.
- **Java:** Too much overhead.
- **GO/Rust:** Too much overhead.
- **Javascript (Node):** Too much overhead.
- **Scheme/Haskell:** Too much overhead. (Perhaps on a lisp machine)

Functional Programming

Consider Building a webserver:

- **Assembly:** Time to develop / test on each platform + see C
- **C:** Memory allocation for each connection? Long running process..memory leaks?
- **Python:** Memory safe but what about data types? Open to XSS attacks etc.
- **Java:** Garbage collections: memory safe! Strongly Typed: type safety. JVM on a small webserver...overhead for small projects.
- **GO/Rust:** Memory safe. Type safe. Lightweight runtime. Requires front end javascript. (ok for static content / specific usecases.)
- **Javascript (Node):** Memory safe. Type safe ??? (typescript etc). Lightweight. Isomorphic: Use same code on server as client. Single threaded...handling concurrency. Callback hell.
- **Scheme/Haskell:** Can define custom types. Memory safe. Can be lightweight. Less community support...

Functional Programming

Consider Building an AI system:

- **Assembly:** Time to develop / test on each platform + see C.
- **C:** Efficient calculation. Memory allocation for many data points? Possible memory leaks. Dealing with high level data will result in complex code. Multithreading?
- **Python:** High level data (OOP) + memory. Multiprocessing (threading). Scypi / Numpy / matplotlib available.
- **Java:** Memory safe. Type safety. Code becomes very long-winded.
- **GO/Rust:** Memory safe. Type safe. High level data…???
- **Javascript (Node):** suitable for distributed systems (GA's). single-threaded is a huge issue. Callback hell…
- **Scheme/Haskell:** Elegant code. Custom types. Memory safe. Lightweight. Mathematical system => mathematical code.

Lisp (Racket / Scheme)

Lisp

Lisp: List Processing

Scheme: Dialect of Lisp

Racket: Implementation of Scheme.

Lisp

Prefix Notation: (a.k.a Polish Notation)

```
(+ 3 4)

(* 5 6)

(- 4 (* 5 6))
```

Lisp - Function Vs Literal

Parenthesis means function:

(+ 3 4)	= 7
(* 5 6)	= 30
(- 4 (* 5 6))	= -26

Quote means literal:

(+ 3 4)	= 7
‘(+ 3 4)	= ‘(+ 3 4)

Rather than considering ‘+’ the name of a function, the quote means to take everything literally…’+’ is just a word.

Nothing is evaluated.

Lisp - S-Expressions

Both **code** and **data** are structured as nested lists in Lisp.

Symbolic Expressions (s-expressions, sexprs, sexps) are a notation for nested list structures.

Defined with a very simple recursive grammar, but produces a very flexible framework for computing..

An s-expressions is defined as:

1. An atom
2. An expression in the form (X . Y) where X

   and Y and s-expressions.

Lisp - S-Expressions: Atoms

An atom (historically) is something that is ‘indivisible’.

This is true for s-expressions in Lisp too.

Primitive data types like number, string, boolean etc. are atoms. (1 2 3…. A b c… \n \t … \0)

Lists and pairs (s-expressions) are not atoms.

Lisp - Cons Pairs

A pair is two pieces of data together.

Pairs are created by the `cons` function, short for construct:

The two values joined with cons are printed between parentheses, but with a dot (i.e., a period surrounded by whitespace) in between:

```
> (cons 1 2)

'(1 . 2)

> (cons "banana" "split")

'("banana" . "split")
```