Chapter 11: Functional Languages

CSE 216 –PROGRAMMING ABSTRACTIONS

HTTPS://PPAWAR.GITHUB.IO/CSE216-F19/INDEX.HTML

Historical Origins

- The imperative and functional models grew out of work undertaken Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ~1930s
- Turing's model of computing was the *Turing machine* a sort of pushdown automaton using an unbounded storage "tape"
  - the Turing machine computes in an imperative way, by changing the values in cells of its tape – like variables just as a high level imperative program computes by changing the values of variables
  - https://www.youtube.com/watch?v=gJQTFhkhwPA

Historical Origins

- Church-Turing thesis
  - A function on the natural numbers is computable by a human being following an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine.
- Church's model of computing is called the *lambda calculus* (also written as **λ-calculus**)
  - is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution
  - based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter λ—hence the notation's name)

λ-calculus

- Terms are built using only the following rules producing expressions such as: producing expressions such as: $(\lambda x.\lambda y.(\lambda z.(\lambda x.z\ x)\ (\lambda y.z\ y))\ (x\ y))$

Syntax	Name	Description
x	Variable	A character or string representing a parameter or mathematical/logical value
(λx.M)	Abstraction	Function definition (M is a lambda term). The variable x becomes bound in the expression.
(M N)	Application	Applying a function to an argument. M and N are lambda terms.

- alpha equivalence: λa.a = λb.b
- beta substitution: (λa.aa) b = bb
- https://www.youtube.com/watch?v=eis11j_iGMs

Simple examples of λ-calculus

- λx.x+1 “plus one” function defines a function of one argument, whose formal parameter is named 'x'. The function body is: "x+1".
- Apply argument to function: (λx.x+1)3 -> Lambda expression
- Computation: (λx.x+1)3 ⇒ 3+1 ⇒ 4
- Abstract syntax tree where λ is the abstraction operator and apply is application operator

```
      apply
      /   \
     λ     3
    / \
   x   +
      / \
     x   1
```

Simple examples of λ-calculus

- Example with two applications: (λx.x+1)((λy.y+2)3)
- Abstract syntax tree where λ is the abstraction operator and apply is application operator

```
    apply         =>    apply     =>  apply   =>  +  =>  6
   /     \             /    \        /    \      / \
  λ       apply       λ      +      λ      5    5   1
 / \       / \       / \   / \     / \
x   +     λ   3     x   + 3   2   x   +
   / \   / \           / \           / \
  x   1 y   +         x   1         x   1
           / \
          y   2
```

Functional Programming Concepts

- Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language
- The key idea: do everything by composing functions
- So how do you get anything done in a functional language?
  - Recursion takes the place of iteration
  - First-call functions take value inputs
  - Higher-order functions take a function as input

Functional Programming Concepts

- So how do you get anything done in a functional language?
  - Recursion (especially tail recursion which uses accumulator) takes the place of iteration
  - In general, you can get the effect of a series of assignments

```
x := 0          ...
x := expr1      ...
x := expr2      ...
```

  from f3(f2(f1(0))), where each f expects the value of x as an argument, f1 returns expr1, and f2 returns expr2

Functional Programming Concepts

- Recursion even does a nifty job of replacing looping

```
x := 0; i := 1; j := 100;
while i < j do
     x := x + i*j; i := i + 1;
          j := j - 1
end while
return x
```

becomes f(0,1,100), where

```
f(x,i,j) == if i < j then
f (x+i*j, i+1, j-1) else x
```

Functional Programming Concepts

- Necessary features, many of which are missing in some imperative languages
  - 1st class and high-order functions
  - Extensive polymorphism – use function on as general a class of arguments
  - powerful list facilities
  - structured function returns – return structured types such as arrays from functions
  - garbage collection

LISP languages

- All of them use (symbolic) s-expression syntax: (+ 1 2).
- LISP is old - dates back to 1958 - only Fortran is older.
- Anything in parentheses is a function call (unless quoted)
  - (+ 1 2) evaluates to 3
  - (* 5 (+ 7 3)) evaluates to 50.
  - ((+ 1 2)) <- error, since 3 is not a function.
  - by default, s-expressions are evaluated. We can use the quote special form to stop that: (quote (+ 1 2))
  - Short form: '(+ 1 2) is a list containing +, 1, 2

Functional Programming Concepts

- Scheme is a particularly elegant Lisp
- Other functional languages
  - ML
  - Miranda
  - Haskell
  - FP
- Haskell is the leading language for research in functional programming

Evaluation Order

- applicative order:
  - evaluates arguments before passing them to a function:

(lambda (x) (* x x) (+ 1 2))

(lambda (x) (* x x) 3)

(* 3 3)

9

- normal order:
  - passes in arguments before evaluating them:

(lambda (x) (* x x) (+ 1 2))

(* (+ 1 2) (+ 1 2))

(* 3 3)

9

–Note: we might want normal order in some code.

(if-tuesday (do-tuesday)) // do-tuesday might print something and we want it only if it's Tuesday

Evaluation Order Example

- ((lambda (x y) (if x (+ y y) 0) t (* 10 10))
- Applicative order:
  - ((lambda (x y) (if x (+ y y) 0) t 100)
  - (if t (+ 100 100) 0)
  - (+ 100 100)
  - 200
  - –(four steps !)
- Normal Order:
  - (if t (+ (* 10 10) (* 10 10)) 0)
  - (+ (* 10 10) (* 10 10))
  - (+ 100 (* 10 10))
  - (+ 100 100)
  - 200
  - –(five steps !)

High-Order Functions

- Higher-order functions
  - Take a function as argument, or return a function as a result
  - Great for building things
  - Currying (after Haskell Curry, the same guy Haskell is named after)
    - ML, Miranda, OCaml, and Haskell have especially nice syntax for curried functions

Currying

A common operation, named for logician Haskell Curry, is to replace a multiargument function with a function that takes a single argument and returns a function that expects the remaining arguments:

```
(define curried-plus (lambda (a) (lambda (b) (+ a b))))
((curried-plus 3) 4)                    ⟹ 7
(define plus-3 (curried-plus 3))
(plus-3 4)                              ⟹ 7
```

Among other things, currying gives us the ability to pass a "partially applied" function to a higher-order function:

```
(map (curried-plus 3) '(1 2 3))         ⟹ (4 5 6)
```

- Some languages use currying as their main function-calling semantics (ML): **fun add a b : int = a + b;** ML's calling conventions make this easier to work with: **add 1**
  add 1 2

Pattern Matching

- It's common for FP languages to include pattern matching operations:
    - matching on value,
    - matching on type,
    - matching on structure (useful for lists).
  - ML example:

```
fun sum_even l =
 case l of
        nil => 0
        | b :: nil => 0
        | a :: b :: t => h + sum_even t;
```

Functional Programming in Perspective

- Problems
  - –difficult (but not impossible!) to implement efficiently on von Neumann machines
  - –lots of copying of data through parameters
  - –frequent procedure calls
  - –heavy space use for recursion
  - –requires garbage collection
  - –requires a different mode of thinking by the programmer
  - –difficult to integrate I/O into purely functional model
- Advantages of functional languages
  - programs are often surprisingly short
  - language can be extremely small and yet powerful