Concepts of Programming Languages

Lecture 15 - Functional Programming

Patrick Donnelly

Montana State University

Spring 2014

Administrivia

Assignments:

Homework #3 : due 03.31

Reading:

Chapter 15

It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.

A. Perlis

Introduction

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

Overview of Functional Languages

They emerged in the 1960's with Lisp

Functional programming mirrors *mathematical functions*:
domain = input, range = output

Variables are mathematical *symbols*:
not associated with memory locations.

Pure functional programming is *state-free*: no assignment

Mathematical Functions

Definition

A ***mathematical function*** is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*.

Example

The function Square has R (the reals) as domain and range.

Square: $R \rightarrow R$

Square(n): n^2

Total Function

A function is *total* if it is defined for all values of its domain. Otherwise, it is *partial*. E.g., Square is total.

Functional Forms

Definition

A higher-order function, or ***functional form***, is one that either takes functions as parameters or yields a function as its result, or both

Functional 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 $h(x) \equiv f(g(x))$

Example

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

Example

For $h(x) \equiv x * x$
$\alpha(h, (2, 3, 4))$ 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

Definition

Referential transparency: a function's result depends only upon the values of its parameters, thus the evaluation of a function always produces the same result given the same parameters.

LISP Data Types and Structures

Data object types: originally only atoms and lists

List form: parenthesized collections of sublists and/or atoms

e.g., `(A B (C D) E)`

Originally, LISP was a typeless language

LISP lists are stored internally as single-linked lists

LISP Interpretation

Lambda notation is used to specify functions and function definitions.

Function applications and data have the same form.

Example

If the list (A B C) is interpreted as data it is a simple list of three atoms, `A, B`, and `C`

If it is interpreted as a function application, it means that the function named `A` is applied to the two parameters, `B` and `C`

The first LISP interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation

Common LISP

A combination of many of the features of the popular dialects of LISP around in the early 1980s A large and complex language–the opposite of Scheme Features include:

- records
- arrays
- complex numbers
- character strings
- powerful I/O capabilities
- packages with access control
- iterative control statements

ML

A static-scoped functional language with syntax that is closer to Pascal than to LISP

Uses type declarations, but also does type inferencing to determine the types of undeclared variables

It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions

Does not have imperative-style variables

Its identifiers are untyped names for values

Includes exception handling and a module facility for implementing abstract data types

Includes lists and list operations

Currying

ML functions actually take just one parameter–if more are given, it considers the parameters a tuple (commas required)

Definition

Process of ***currying*** replaces a function with more than one parameter with a function with one parameter that returns a function that takes the other parameters of the original function

An ML function that takes more than one parameter can be defined in curried form by leaving out the commas in the parameters

```
fun add a b = a + b;
```

A function with one parameter, `a`. Returns a function that takes `b` as a parameter. Call: `add 3 5;`

Partial Evaluation

Curried functions can be used to create new functions by partial evaluation

Partial evaluation means that the function is evaluated with actual parameters for one or more of the leftmost actual parameters

```
fun add5 x add 5 x;
```

Takes the actual parameter 5 and evaluates the `add` function with 5 as the value of its first formal parameter. Returns a function that adds 5 to its single parameter

```
val num = add5 10; (* sets num to 15 *)
```

Haskell

Similar to ML (syntax, static scoped, strongly typed, type inferencing, pattern matching)

Different from ML (and most other functional languages) in that it is purely functional (e.g., no variables, no assignment statements, and no side effects of any kind)

Lazy Evaluation

Definition

A language is ***strict*** if it requires all actual parameters to be fully evaluated.

Definition

A language is ***nonstrict*** if it does not have the strict requirement

Nonstrict languages are more efficient and allow some interesting capabilities – infinite lists

Definition

Lazy evaluation = delaying argument evaluation in a function call until the argument is needed.

- Advantage: flexibility

F#

Based on Ocaml, which is a descendant of ML and Haskell

Fundamentally a functional language, but with imperative features and supports OOP

Has a full-featured IDE, an extensive library of utilities, and interoperates with other .NET languages

Includes tuples, lists, discriminated unions, records, and both mutable and immutable arrays

Supports generic sequences, whose values can be created with generators and through iteration

Why F# is Interesting

It builds on previous functional languages

It supports virtually all programming methodologies in widespread use today

It is the first functional language that is designed for interoperability with other widely used languages

At its release, it had an elaborate and well-developed IDE and library of utility software

Support for Functional Programming

Support for functional programming is increasingly creeping into imperative languages:

- Anonymous functions (lambda expressions)
  - JavaScript: leave the name out of a function definition
  - C#: i => (i % 2) == 0 (returns true or false depending on whether the parameter is even or odd)
  - Python: lambda a, b : 2 * a - b
- 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]`

Support for Functional Programming

- Python supports partial function applications

```
from operator import add
add5 = partial (add, 5)
```

(the first line imports add as a function)

Use: `add5(15)`

- Ruby Blocks
  - Are effectively subprograms that are sent to methods, which makes the method a higher-order subprogram
  - A block can be converted to a subprogram object with lambda

```
times = lambda |a, b| a * b
```

Use: `x = times.(3, 4)` (sets x to 12)

  - Times can be curried with

```
times5 = times.curry.(5)
```

Use: `x5 = times5.(3)` (sets x5 to 15)

Comparing Functional and Imperative Languages

Imperative Languages:

- Efficient execution
- Complex semantics
- Complex syntax
- Concurrency is programmer designed

Functional Languages:

- Simple semantics
- Simple syntax
- Less efficient execution
- Programs can automatically be made concurrent

Summary

Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution

LISP began as a purely functional language and later included imperative features

Scheme is a relatively simple dialect of LISP that uses static scoping exclusively

Common LISP is a large LISP-based language

ML is a static-scoped and strongly typed functional language that uses type inference

Summary

Haskell is a lazy functional language supporting infinite lists and set comprehension.

F# is a .NET functional language that also supports imperative and object-oriented programming

Some primarily imperative languages now incorporate some support for functional programming

Purely functional languages have advantages over imperative alternatives, but still are not very widely used