Today’s Topics

Functionals
- Objects representing executable code

Higher-order functions
- Functions that accept functions as arguments or return them as results

Domain-specific languages
- PCAP: primitives, combination, abstraction pattern

Representing Code with Data

Consider a datatype representing language syntax
- Formula is the language of propositional logic formulas
- A Formula value represents program code in a data structure; i.e.
  \[ \text{new And(new Var(“x”), new Var(“y”))} \]
  has the same semantic meaning as the Java code
  \[ x && y \]
  but a Formula value is a first-class object
  - first-class: a value that can be passed, returned, stored, manipulated
  - the Java expression \( x && y \) is not first-class

Representing Code as Data

Recall the visitor pattern
- A visitor represents a function over a datatype
  - e.g. \texttt{new SizeVisitor()} represents size: \texttt{List \rightarrow int}
  \[
  \text{public class SizeVisitor<E> implements ListVisitor<E,Integer> {}
  \text{public Integer visit(Empty<E> l) { return 0; }}
  \text{public Integer visit(Cons<E> l) { return 1 + l.rest().accept(this); }}
  \text{}}
  \]
- A visitor represents code as a first-class object, too
- A visitor is an object that can be passed around, returned, and stored
- But it’s also a function that can be invoked

Today’s lecture will see more examples of code as data

Today’s Problem: Music

Interesting music tends to have a lot of repetition
- Let’s look at rounds, canons, fugues
- A familiar simple round is “Row Row Row Your Boat”: one voice starts, other voices enter after a delay
  \[ \text{Row row row your boat, gently down the stream, merrily merrily ...} \]
  \[ \text{Row row row your boat, gently down the stream...} \]
- Bach was a master of this kind of music
  - Recommended reading: \textit{Godel Escher Bach}, by Douglas Hofstadter

Recall our MIDI piano from early lectures
- A song could be represented by Java code doing a sequence of calls on a state machine:
  \[
  \text{machine.play(E); machine.play(D); machine.play(C); ...}
  \]
- We want to capture the code that operates this kind of machine as first-class data objects that we can manipulate, transform, and repeat easily

Music Data Type

Let’s start by representing simple tunes
- \texttt{Music = Note(duration:double, pitch:Pitch, instr:Instrument)
  + Rest(duration:double)
  + Concat(m1:Music, m2:Music)}
- duration is measured in beats
- Pitch represents note frequency (e.g. C, D, E, F, G; essentially the keys on the piano keyboard)
- Instrument represents the instruments available on a MIDI synthesizer

Design questions
- is this a tree or a list? what would it look like defined the other way?
- what is the “empty” Music object?
  - it’s usually good for a data type to be able to represent nothing
  - avoid null
- what are the rep invariants for Note, Rest, Concat?
A Few of Music’s Operations

**Notes**: Music = Note(duration:double, pitch:Pitch, instr:Instrument)
- **Duration**: Music → double
- **Transpose**: Music x int → Music
- **Music**: String x Instrument → Music

---

**Implementation Choices**

Creators can be constructors or factory methods
- Java constructors are limited interfaces can’t have them, and constructor can’t choose which runtime type to return
  - New C() must always be an object of type C,
  - so we can’t have a constructor Music(String, Instrument), whether Music is an interface or an abstract class

Observers & producers can be methods or visitors
- Methods break up function into many files; visitor is all in one place
- Adding a method requires changing source of classes (not always possible)
- Visitor keeps dependencies out of data type itself (e.g. MIDI dependence)
- Method has direct access to private rep; visitor needs to use observers

Producers can also be new subclasses of the datatype
- e.g. Music = ... + Transpose(m:Music, semitones:int)
- Defers the actual evaluation of the function
- Enables more sharing between values
- Adding a new subclass requires changing all visitors

---

**Multiple Voices**

For a round, the parts need to be sung simultaneously
- Music = Note(duration:double, pitch:Pitch, instr:Instrument)
  - Rest(duration:double)
  - Concat(m:Music, m2:Music)
  - Together(m1:Music, m2:Music)

Here’s where our decision to make Concat() tree-like becomes very useful
- Suppose we instead had:
  - Concat = List<Note + Rest>
  - Together = List<Concat>
- What kinds of music would we be unable to express?

Composite pattern
- The composite pattern means that groups of objects treated the same way as single objects (primitives)
  - T = C(..., T) + P(..., T) + P(..., T) + P(..., T) + ... + P(..., T)
- Music and Formula are composite data types.

---

**Distinguishing Voices**

We want each voice in the canon to be distinguishable
- e.g. an octave higher; or lower; or using a different instrument
- So these operations over Music also need to be first-class objects that can be passed to canon()?

Extend canon() to apply a function to the repeated melody
- **Canon**: Music x int x double x (Music -> Music) -> Music
  - Canon(rrryb, 4, 4, transpose OCTAVE) produces 4 voices, each one octave higher than the last?
  - Transpose = lambda m: transpose(m, semitones)

**Canon() is a higher-order function**
- A higher-order function takes a function as an argument or returns a function as its result
**Counterpoint**

A canon is a special case of a more general pattern

- **Counterpoint** is a special case of a delayed function composition.

- **Expressed as counterpoint**, a canon applies two functions to the music:
  - `delay` and `transform`

  
  \[
  \text{canon}(m, f, n) = \text{counterpoint}(m, f^{\text{delay}(d)}, n)
  \]

  where

  - `\text{delay} : \mathbb{Z} \rightarrow \mathbb{Z}`
  - `\text{delay}(d) = \lambda m: \text{delay}(m, d)`

- **Another general pattern**

  function composition: \((U \rightarrow V) \rightarrow (T \rightarrow U) \rightarrow (T \rightarrow V)\)

**Repeating Forever**

Music that repeats forever is useful for canons

- **Forever** plays a piece repeatedly:
  
  \[
  \text{play}(\text{forever}(m)) \rightarrow \text{repeat}(m, f, n)
  \]

**Repeating Forever**

A line of music can also be repeated by the same voice

- **Repeat** can be used to repeat a line of music:

  \[
  \text{repeat}(m, f, n) = \text{series}(m, \text{concat}, f, n)
  \]

**Accompaniment**

Accompaniment

- **Accompany** adds a second piece to a melody:

  \[
  \text{accompany}(m, b) = \text{together}(m, \text{repeat}(b, \text{identity}, \text{duration}(m)/\text{duration}(b)))
  \]

**Little Languages**

We’ve built a new language embedded in Java

- **Music data type** and its operations constitute a language for describing music generation.

- Instead of just solving one problem (like playing Row Row Row Your Boat), build a language or toolbox that can solve a range of related problems (e.g., Pachelbel’s canon).

- This approach gives you more flexibility if your original problem turns out to be the wrong one to solve (which is not uncommon in practice!)

- **Capture common patterns as reusable abstractions**

**Formula was an embedded language too**

- Formula combined with SAT solver is a powerful tool that solves a wide range of problems
Embedded Languages

Useful languages have three critical elements

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Formula language</th>
<th>Music language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitives</strong></td>
<td>3, false</td>
<td>Var, Bool</td>
<td>notes, rest</td>
</tr>
<tr>
<td><strong>Means of Combination</strong></td>
<td>+, *, ==, &amp;&amp;,</td>
<td></td>
<td>, ...</td>
</tr>
<tr>
<td><strong>Means of Abstraction</strong></td>
<td>variables, methods, classes</td>
<td>naming + methods in Java</td>
<td>naming + functions in Python</td>
</tr>
</tbody>
</table>

> 6.01 calls this PCAP (the Primitive-Combination-Abstraction pattern)

Summary

Review of many concepts we’ve seen in 6.005

- Abstract data types, recursive data types, interpreter/visitor, composite, immutability

Code as data

- Recursive datatypes, visitors, and functional objects are all ways to express behavior as data that can be manipulated and changed programatically

Higher-order functions

- Operations that take or return functional objects

Building languages to solve problems

- A language has greater flexibility than a mere program, because it can solve large classes of related problems instead of a single problem
- Composite, interpreter, visitor, and higher-order functions are useful for implementing powerful languages
- But in fact any well-designed abstract data type is like a new language