CSci 4223  
Principles of Programming Languages

Lecture 18

Prof. Clarkson  
Spring 2013

Review
• Ruby: classes, methods, variables, arrays, hashes, blocks, procs, subclassing,  
duck typing, dynamic dispatch
• Today:  
  – Functional programming vs. object-oriented programming

Logistics
• HW4: 1-week extension  
  – This overrides any individual extensions previously granted
• Midterm 2: Monday, April 8, during class, in Fungar 103  
  – Same format, policies, etc. as Midterm 1  
  • Can use laptop to interact with SML and Ruby  
  • Can bring 1 double-sided page of notes  
  • Otherwise “closed book”  
  – Review session: Sunday night during recitation

Implementing an “expression” language
Well-known and compelling example of a common pattern:
• Expressions for a small language
• Different variants of expressions: ints, additions, negations, …
• Different operations to perform: eval, toString, hasZero, …
Leads to a matrix (2D-grid) of variants and operations
• Implementation will involve deciding what “should happen” for each entry in the grid regardless of the PL

<table>
<thead>
<tr>
<th>eval</th>
<th>toString</th>
<th>hasZero</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Standard approach in FP
• Define a datatype, with one constructor for each variant
• Define a function for each operation
• So “fill out the grid” via one function per column
  – Can use a wildcard pattern if there is a default for multiple entries in a column

<table>
<thead>
<tr>
<th>Int</th>
<th>Add</th>
<th>Negate</th>
</tr>
</thead>
</table>

Standard approach in OOP
• Define a class, with one abstract method for each operation
  – (No need to indicate abstract methods if dynamically typed)
• Define a subclass for each variant
• So “fill out the grid” via one class per row
  – Can use a method in the superclass if there is a default for multiple entries in a column

<table>
<thead>
<tr>
<th>Int</th>
<th>Add</th>
<th>Negate</th>
</tr>
</thead>
</table>

See lec18_stage1.sml

See lec18_stage1.rb and lec18_stage1.java
FP vs. OOP

- FP and OOP doing the same thing in **exact opposite** way
  - Organize the program “by rows” or “by columns”
- Which is "most natural" may depend on what you are doing (e.g., an interpreter vs. a GUI) or personal taste
- Code layout is important, but there’s no perfect way since software has many dimensions of structure
  - Tools, IDEs can help with multiple “views” (e.g., rows / columns)

### Extending the expression language

- We want to add new:
  - operations (**noNegConstants**)
  - variants (**Mult**)
- FP and OOP each make one of these extensions easy, and the other (more) difficult

### Extension: FP

- SML:
  - Easy to add a new operation, e.g. **noNegConstants**
    - Just write a new function
  - Hard to add a new variant, e.g. **Mult**
    - Have to edit all existing functions
  - But ML type-checker gives a to-do list if we avoided wildcard patterns in Stage 1
- See lec18_stage2.sml

### Extension: OOP

- Java/Ruby:
  - Easy to add a new variant, e.g. **Mult**
    - Just write a new class
  - Hard to add a new operation, e.g. **noNegConstants**
    - Have to modify all existing classes
  - But Java type-checker gives a to-do list if we avoided non-abstract methods in Stage 1
- See lec18_stage2.java and lec18_stage2.rb

### Planning for extension

- FP makes new operations easy
  - So if you know you want new operations, use FP
  - FP can support new variants somewhat awkwardly if you “plan ahead”
    - See **datatype 'a ext_exp and eval_ext** in lec18_stage2.sml
- OOP makes new variants easy
  - So if you know you want new variants, use OOP
  - OOP can support new operations somewhat awkwardly if you “plan ahead”
    - The popular **Visitor Pattern** (not discussed here)

...once again, FP and OOP are **exact opposites**
Thoughts on Extensibility

- Reality: the future is hard to predict
  - Might not know what kind of extensibility you need
  - Might even need both kinds!
    - Languages like Scala try, it’s a hard problem

- Extensibility is a double-edged sword
  - Pro: code more reusable without being changed later
  - Con: original code more difficult to reason about locally or change later (could break extensions)
  - So some language features specifically designed to make code less extensible
    - e.g., Java’s final prevents subclassing/overriding

Summary

- "The 2-D grid" is a fundamental truth about software, essential to understanding how OOP and FP relate

- OOP vs. FP isn’t just a matter of taste

- Software extensibility is easy in some ways and hard in others
  - Which ways are which depend on how code is structured
  - Which in turn heavily influenced by programming paradigm