Programming Languages and Techniques (CIS120)

Lecture 21

March 20, 2019

Transition to Java
Announcements

• HW05: GUI programming
  – Due: 3/26 at 11:59:59pm

• HW06: Pennstagram
  – Available soon
  – Due: Tuesday, April 2\textsuperscript{nd} at 11:59:59pm
  – Java programming

• Java Bootcamp!!
  – Wednesday, March 20, 6-8 pm
  – Towne 100
Goodbye OCaml...
...Hello Java!
Smoothing the transition

• Java Bootcamp!
  – Wednesday, March 20, 6-8 pm, Towne 100

• General advice for the next few lectures: Ask questions, but don’t stress about the details until you need them.

• Java resources:
  – Our lecture notes
  – CIS 110 website and textbook
  – Online Java textbook (http://math.hws.edu/javanotes/) linked from “CIS 120 Resources” on course website
  – Penn Library: Electronic access to “Java in a Nutshell” (and all other O’Reilly books)
  – Piazza
CIS 120 Overview

• Declarative (Functional) programming
  – persistent data structures
  – recursion is main control structure
  – frequent use of functions as data

• Imperative programming
  – mutable data structures (that can be modified “in place”)
  – iteration is main control structure

• Object-oriented (and reactive) programming
  – mutable data structures / iteration
  – heavy use of functions (objects) as data
  – pervasive “abstraction by default”
Java and OCaml together

Stephanie Weirich, Penn Prof. (CIS 120 co-developer, major contributor to Haskell)

Guy Steele, one of the principal designers of Java

Xavier Leroy, one of the principal designers of OCaml

Moral: Java and OCaml are not so far apart...
Recap: The Functional Style

• Core ideas:
  – immutable (persistent / declarative) data structures
  – recursion (and iteration) over tree structured data
  – functions as data
  – generic types for flexibility (i.e. ‘a list)
  – abstract types to preserve invariants (i.e. BSTs)
  – *simple model of computation (substitution)*

• Good for:
  – elegant descriptions of complex algorithms and/or data
  – small-scale compositional design
  – “symbol processing” programs (compilers, theorem provers, etc.)
  – parallelism, concurrency, and distribution
Functional programming

- Immutable lists primitive, tail recursion
- Datatypes and pattern matching for tree structured data
- First-class functions, transform and fold
- Generic types
- Abstract types through module signatures

- No primitive data structures, no tail recursion
- Trees must be encoded by objects, mutable by default
- First-class functions less common*
- Generic types
- Abstract types through public/private modifiers

*completely unsupported until recently (Java 8)
OCaml vs. Java for FP

OCaml provides a succinct, clean notation for working with generic, immutable, tree-structured data. Java requires a lot more "boilerplate".

interface Tree<A> {
    public boolean isEmpty();
}
class Empty<A> implements Tree<A> {
    public boolean isEmpty() {
        return true;
    }
}
class Node<A> implements Tree<A> {
    private final A v;
    private final Tree<A> lt;
    private final Tree<A> rt;
    Node(Tree<A> lt, A v, Tree<A> rt) {
        this.lt = lt; this.rt = rt; this.v = v;
    }
    public boolean isEmpty() {
        return false;
    }
}
class Program {
    public static void main(String[] args) {
        Tree<Integer> t =
            new Node<Integer>(new Empty<Integer>(),
                             3, new Empty<Integer>());
    boolean ans = t.isEmpty();
    }
}
Other Popular Functional Languages

**F#**: Most similar to OCaml, Shares libraries with C#

**Haskell** (CIS 552)
- Purity + laziness

**Swift**
- iOS programming

**Clojure**
- Dynamically typed
- Runs on JVM

**Racket**:
- LISP descendant;
  widely used in education

**Scala**
- Java / OCaml hybrid
Recap: The imperative style

• Core ideas:
  – computation as change of state over time
  – distinction between primitive and reference values
  – aliasing
  – linked data-structures and iteration control structure
  – generic types for flexibility (i.e. ‘a queue)
  – abstract types to preserve invariants (i.e. queue invariant)
  – Abstract Stack Machine model of computation

• Good for:
  – numerical simulations
  – implicit coordination between components (queues, GUI)
  – explicit interaction with hardware
Imperative programming

OCaml

• No null. Partiality must be made explicit with options.

• Code is an expression that has a value. Sometimes computing that value has other effects.

• References are immutable by default, must be explicitly declared to be mutable

Java

• Most types have a null element. Partial functions can return null.

• Code is a sequence of statements that have effects, sometimes using expressions to compute values.

• References are mutable by default, must be explicitly declared to be constant
Explicit vs. Implicit Partiality

**OCaml identifiers**

- Cannot be changed once created; only mutable fields can change
  ```ocaml
type 'a ref = { mutable contents: 'a }
let x = { contents = counter () }
;; x.contents <- counter ()
```

- Cannot be null, must use options
  ```ocaml
  let y = { contents = Some (counter ()) }
  ;; y.contents <- None
  ```

- Accessing the value requires pattern matching
  ```ocaml
  ;; match y.contents with
  | None -> failwith "NPE"
  | Some c -> c.inc ()
  ```

**Java variables**

- Can be assigned to after initialization
  ```java
  Counter x = new Counter ();
  x = new Counter ();
  ```

- Can always be null
  ```java
  Counter y = new Counter ();
  y = null;
  ```

- Check for null is implicit whenever a variable is used
  ```java
  y.inc();
  ```

- If null is used as if it were an object (i.e. for a method call) then a NullPointerException occurs.
"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. ... This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

Sir Tony Hoare, QCon, London 2009
Java Core Language

differences between OCaml and Java
Structure of a Program

- All code lives in (perhaps implicitly named) **modules**.

- Modules may contain multiple **type definitions**, **let-bound value declarations**, and **top-level expressions** that are executed in the order they are encountered.

- All code lives in explicitly named **classes**.
- Classes are themselves types.
- Classes contain **field declarations** and **method definitions**.
- There is a single "entry point" of the program where it starts running, which must be a method called **main**.
Expressions vs. Statements

• OCaml is an *expression language*
  – Every program phrase is an expression (and returns a value)
  – The special value () of type `unit` is used as the result of expressions that are evaluated only for their side effects
  – Semicolon is an *operator* that combines two expressions (where the left-hand one returns type unit)

• Java is a *statement language*
  – Two sorts of program phrases: expressions (which compute values) and statements (which don’t)
  – Statements are *terminated* by semicolons
  – Any expression can be used as a statement (but not vice-versa)
Types

- As in OCaml, every Java *expression* has a type
- The type describes the value that an expression computes

<table>
<thead>
<tr>
<th>Expression form</th>
<th>Example</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable reference</td>
<td>x</td>
<td>Declared type of variable</td>
</tr>
<tr>
<td>Object creation</td>
<td>new Counter ()</td>
<td>Class of the object</td>
</tr>
<tr>
<td>Method call</td>
<td>c.inc()</td>
<td>Return type of method</td>
</tr>
<tr>
<td>Equality test</td>
<td>x == y</td>
<td>boolean</td>
</tr>
<tr>
<td>Assignment</td>
<td>x = 5</td>
<td><em>don’t use as an expression!!</em></td>
</tr>
</tbody>
</table>
## Type System Organization

<table>
<thead>
<tr>
<th></th>
<th>OCaml</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>primitive types</strong></td>
<td>int, float, char, bool, ...</td>
<td>int, float, double, char, boolean, ...</td>
</tr>
<tr>
<td>(values stored</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;directly&quot; in the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>structured types</strong></td>
<td>tuples, datatypes, records, functions, arrays</td>
<td>objects, arrays</td>
</tr>
<tr>
<td>(a.k.a. reference</td>
<td></td>
<td></td>
</tr>
<tr>
<td>types — values</td>
<td>(objects encoded as records of functions)</td>
<td>(records, tuples, datatypes, strings, first-class</td>
</tr>
<tr>
<td>stored in the heap)</td>
<td></td>
<td>functions are special cases of classes)</td>
</tr>
<tr>
<td><strong>generics</strong></td>
<td>‘a list</td>
<td>List&lt;A&gt;</td>
</tr>
<tr>
<td><strong>abstract types</strong></td>
<td>module types (signatures)</td>
<td>interfaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>public/private modifiers</td>
</tr>
</tbody>
</table>
# Arithmetic & Logical Operators

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>=, ==</td>
<td>==</td>
</tr>
<tr>
<td>&lt;&gt;, !=</td>
<td>!=</td>
</tr>
<tr>
<td>&gt;, &gt;=, &lt;, &lt;=</td>
<td>&gt;, &gt;=, &lt;, &lt;=</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>mod</td>
<td>%</td>
</tr>
<tr>
<td>not</td>
<td>!</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Java: Operator Overloading

• The *meaning* of an operator in Java is determined by the *types* of the values it operates on:
  – Integer division
    \[ 4/3 \Rightarrow 1 \]
  – Floating point division
    \[ 4.0/3.0 \Rightarrow 1.3333333333333333 \]
  – Automatic conversion from int to float
    \[ 4/3.0 \Rightarrow 1.3333333333333333 \]

• Overloading is a general mechanism in Java
  – we’ll see more of it later
Equality

• like OCaml, Java has two ways of testing reference types for equality:
  – “pointer equality”
    \[ o1 == o2 \]
  – “deep equality”
    \[ o1.equals(o2) \]

• Normally, you should use == to compare primitive types and 
  “.equals” to compare objects

every object provides an “equals” method that should “do the right thing” depending on the class of the object
Strings

• **String** is a *built in* Java class
• Strings are sequences of (unicode) characters
  "" "Java" "3 Stooges" "富士山"
• + means String concatenation (overloaded)
  "3" + " " + "Stooges" ⇒ "3 Stooges"
• Text in a String is immutable (like OCaml)
  – but variables that store strings are not
  – String x = "OCaml";
  – String y = x;
  – Can't do anything to x so that y changes
• The `.equals` method returns true when two strings contain the same sequence of characters
What is the value of `ans` at the end of this program?

```java
String x = "CIS 120";
String z = "CIS 120";
boolean ans = x.equals(z);
```

Options:
- true
- false
- NullPointerException
What is the value of ans at the end of this program?

String x = "CIS 120";
String z = "CIS 120";
boolean ans = x.equals(z);

1. true
2. false
3. NullPointerException

Answer: true
This is the preferred method of comparing strings!
What is the value of `ans` at the end of this program?

```java
String x1 = "CIS ";
String x2 = "120";
String x = x1 + x2;
String z = "CIS 120";
boolean ans = (x == z);
```
What is the value of ans at the end of this program?

String x1 = "CIS ";
String x2 = "120";
String x = x1 + x2;
String z = "CIS 120";
boolean ans = (x == z);

1. true
2. false
3. NullPointerException

Answer: false
Even though x and z both contain the characters “CIS 120”, they are stored in two different locations in the heap.
What is the value of \( \text{ans} \) at the end of this program?

```java
String x = "CIS 120";
String z = "CIS 120";
boolean ans = (x == z);
```
What is the value of ans at the end of this program?

String x = "CIS 120";
String z = "CIS 120";
boolean ans = (x == z);

1. true
2. false
3. NullPointerException

Answer: true(!)
Why? Since strings are immutable, two identical strings that are known when the program is compiled can be aliased by the compiler (to save space).
Moral

Always use s1.equals(s2) to compare strings!

Compare strings with respect to their content, not where they happen to be allocated in memory...