

Comp 311

Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

September 25, 2018

Announcements

- Homework 1 was due today before class
- Homework 2 will be posted after class today
 - Assignment description PDF on Piazza
 - No provided “skeleton” code
 - Simple interface (compilation/linking) check provided
- Due in 2 weeks

Additional Syntax for Homework 2

Repeated Parameters

- Scala allows the last entry in a parameter list to stand for zero or more arguments
- Arguments are placed in a sequence of the given type

```
def squares(xs: Int*) =  
  for (x <- xs)  
  yield x*x
```

Repeated Parameters

- Scala allows the last entry in a parameter list to stand for zero or more arguments
- Arguments are placed in a sequence of the given type

`squares(4, 2, 6, 5, 8)`

`squares()`

`squares(4, 2, 6, 8)`

`squares(3)`

`squares(4, 3, 7)`

Repeated Parameters

- Scala allows the last entry in a parameter list to stand for zero or more arguments
- Arguments are placed in a sequence of the given type

```
def fnName(a1: T1, ..., aN: TN*) = expr
```

Repeated Parameters

- Scala allows the last entry in a parameter list to stand for zero or more arguments
- Arguments are placed in a sequence of the given type

```
squares(1, 2, 3, 4, 5) ↦*  
ArrayBuffer(1, 4, 9, 16, 25)
```

ArrayBuffers

- Buffers in Scala enable incremental creation of sequences
 - Random access to elements
 - Support destructive append, prepend, insert
 - We have not talked about destructive operations yet
 - Just pretend they are Arrays for now
- ArrayBuffers are simply Buffers implemented using Arrays
 - Similar idea to Java's ArrayList class

Repeated Parameters

- If you have an array and you wish to pass it to a repeated parameter, include the suffix :_*

```
val myArray = Array(1, 2, 3)
    squares(myArray: _*)
```

Guidelines on Repeated Parameters

- Use repeated parameters to provide factory methods for collections classes

`List(1,2,3,4,5)`

- Use repeated parameters for methods that map over an immediately provided set of values

`squares(1,2,3,4,5)`

- Use repeated parameters for folds over an immediately provided set of values

`sum(1,2,3,4,5)`

Named Arguments

- With *named arguments*, the arguments to a function can be passed in any order
- Each argument must be prefixed with the name of the parameter and an equals sign:

```
def speed(distance: Double, time: Double) =  
    distance/time
```

```
speed(distance = 2.0, time = 5.0)
```

Named Arguments

If positional arguments are mixed with named arguments, the positional arguments must come first

```
def speed(distance: Double, time: Double) =  
    distance/time
```

```
speed(2.0, time = 5.0)
```

Guidelines on Named Arguments

- Named arguments add bulk to function applications
- Use when it's unclear which arguments correspond to which parameters, e.g.:
 - There are multiple arguments of the same type
 - There is no natural order for the arguments
 - The expected order of the arguments is difficult to remember

Default Parameter Values

- Function parameters can include default values:

```
case class Circle(radius: Double = 1) extends Shape {  
  val pi = 3.14  
  
  def area = { pi * radius * radius }  
  def makeLikeMe(that: Shape): Circle = this  
}
```

- The argument for a parameter with a default value can be omitted at the call site:

`Circle()`

Guidelines of Default Parameter Values

- Consider default parameter values instead of static overloading
- Use when there is a common argument value that is usually used
 - A default I/O source, file location, etc.

Exception Handling

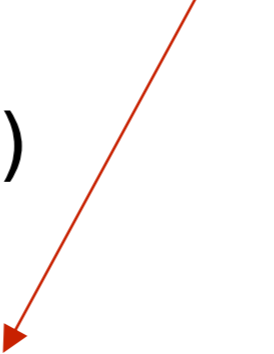
- Scala handles error by *throwing* exceptions
- An exception can be *caught* by a scope that is a *dynamic* parent of the throwing scope
- We'll delay the detailed reduction semantics for try/catch until a later date

```
try // expression...
catch {
  case Pattern => // expression...
  // zero or more additional cases...
}
```


Try/Catch Examples

Patterns can be restricted to only match a specific type

```
try {  
    require(false)  
}  
catch {  
    case e: IllegalArgumentException =>  
        assertEquals(e.getMessage,  
                    "requirement failed")  
}
```



Try/Catch Examples

```
try abs(Int.MinValue)
catch {
  // precondition failure
  case e: IllegalArgumentException =>
    assertEquals(e.getMessage,
                 "requirement failed")
  // postcondition failure
  case e: AssertionError =>
    assertEquals(e.getMessage,
                 "assertion failed")
}
```

Custom Error Messages

```
try {  
    require(false, "my message")  
}  
catch {  
    case e: IllegalArgumentException =>  
        assertEquals(e.getMessage,  
            "requirement failed: my message")  
}
```

Custom Error Messages

```
def abs(x: Int) = {  
  if (x < 0) -x else x  
} ensuring(_ > 0, "negative absolute value")
```

Non-trivial Postconditions

```
def add(x: Int, y: Int) = {  
  x + y  
} ensuring(result => {  
  if (x >= 0 && y >= 0)  
    result >= max(x, y)  
  else if (x < 0 && y < 0)  
    result < min(x, y)  
  else  
    result >= min(x, y)  
}, "integer overflow during addition")
```

String Formatting

Scala strings support `printf`-style formatting:

```
"Customer %s ordered %d units"  
  .format(accountName, 500)
```

String Interpolation

Scala strings also support *interpolation* of values:

Prefix the string with an **S**

`s"My name is $name."`

`s"My name is ${name}."`

`s"Max value: ${max(x, y)}."`

Prefix variable names with **\$**

General expressions must be wrapped in braces