This Scala notebook uses *BeakerX*, a Two Sigma Open Source project that enhances Jupyter.

[http://beakerx.com/ (http://beakerx.com/)](http://beakerx.com/)

In [1]:
```scala
scala.util.Properties.versionMsg
```

Out[1]:
```
Scala library version 2.11.12 -- Copyright 2002-2017, LAMP/EPFL
```

In [2]:
```scala
// Definitions from Lecture 04 used in this notebook:

/** Compute the maximum of two integers */
def max(x: Int, y: Int): Int = {
  if (x > y) x else y
} ensuring(result => result == x | result == y)

/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance.
 * Undefined for ticket prices over 1000 cents.
 */
def attendance(ticketPrice: Int): Int = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  max(0, 120 + 15 * (500 - ticketPrice) / 10)
} ensuring(result => result >= 0)

/**
* Returns cost to the theater of showing a film,
* as a function of ticketPrice.
*/
def cost(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  18000 + 4 * attendance(ticketPrice)
} ensuring(result => result >= 0)
```

Out[2]:
```
max: (x: Int, y: Int)Int
attendance: (ticketPrice: Int)Int
cost: (ticketPrice: Int)Int
```

Defining Constants

The movie theater profit functions that we implemented in the previous lecture contained several *magic constants* in their definitions: `120` attendees, `18000` ¢ base cost per showing, etc. Numeric literals included in functions with no explanation are called *magic constants* or *magic numbers* because, lacking any documentation or other context, they seem arcane to the reader.

A simple way to eliminate magic constants is to give them descriptive names. In Scala, we use the `val` keyword to define constants:

In [3]:
```scala
val basePerformanceCost: Int = 18000
```

Out[3]: 18000

We can also take advantage of Scala's built-in type inference and elide the type annotation for a constant definition. The fact that *explicit* types are optional in many contexts is the primary reason that Scala syntax puts the type after the name rather than before the name, and uses dedicated keywords such as `def` and `val` to denote definitions.

In [4]:
```scala
val basePerformanceCost = 18000
```

Out[4]: 18000

Note that although the constant `basePerformanceCost` defined above has no *explicit* type declaration, that does not mean it is *dynamically typed* or lacks a *static* type. We can apply the typing rules for expressions (from our lecture on static types) to determine that `18000` has the static type `Int` , and therefore `basePerformanceCost` also has the static type `Int` . The Scala compiler similarly infers the static types for constants with no explicit type given. The same logic is used to infer the result type for function definitions with no explicit result-type given.

Compound Expressions

Scala, like most programming languages, allows you to create local *bindings* within a function. In imperative languages like Java and Python, these would be called *local variables*, since the value of the binding can be changed throughout the function body. In Scala, we call these bindings *values*—not *variables*—since they are constant (i.e., immutable). We refer to an expression prefixed with a sequence of constant value definitions as a *compound expression*. We will refer to non-compound expressions as *simple expressions*. The term *expression* can now refer to either compound or simple expressions.

Let's restructure the definition of the `cost` function from Lecture 04 to use named constant values in a compound expression:

In [5]:
```scala
/**
* Returns cost to the theater of showing a film,
* as a function of ticketPrice.
*/
def cost(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)

  val fixedCost = 18000
  val perAttendeeCost = 4

  fixedCost + perAttendeeCost * attendance(ticketPrice)
} ensuring(result => result >= 0)
```

Out[5]: cost: (ticketPrice: Int)Int

Syntax for Compount Expressions in Functions

We now expand our Core Scala language to allow zero or more `val` definitions within functions:

```
def fnName(arg0: Type0, arg1: Type1, ...): ResultType = {
  require(preconditionPredicate, "Precondition error message")

  val localConstantA: TypeA = ???
  val localConstantB: TypeB = ???
  ...

  bodyExpression
} ensuring (result => postconditionPredicate, "Postcondition er
ror message")
```

The explicit type annotations `ResultType`, `TypeA`, and `TypeB` are all optional, as discussed above.

Reduction Rule for Compound Expressions

1. While there are one or more `val` bindings in the compound expression:
   A. Compute the right-hand-side value of the first `val` binding using the rules for reducing simple expressions.
   B. Substitute the reduced value on the right-hand-side for all occurrences of the left-hand-side symbolic name throughout the remainder of this compound expression.
   C. Drop the first `val` binding since it has now been fully reduced and substituted.
2. Reduce the resulting simple expression using the rules for reducing expressions.

Example

Let's define a function to compute the circumference of a circle:

In [6]:

```
def circumference(radius: Double): Double = {
  require(radius > 0, "Radius must be positive")

  val diameter = radius + radius
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal places

  diameter * pi
} ensuring (result => result > 0, "Circumference is positive")
```

Out[6]:

```
circumference: (radius: Double)Double
```

Now let's step through the reduction for the expression `circumference(5.0 - 3.0)`. For now we'll treat all `require` and `ensuring` clauses in our reduction as no-ops.

```
circumference(5.0 - 3.0)
↦
circumference(2.0)
↦
{
  require({2.0} > 0, "Radius must be positive")

  val diameter = {2.0} + {2.0}
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  diameter * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val diameter = {2.0} + {2.0}
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  diameter * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val diameter = {2.0} + {2.0}
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  diameter * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val diameter = 4.0
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  diameter * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  {4.0} * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val pi = 3.14

  {4.0} * pi
```

```
  } ensuring (result => result > 0, "Circumference is positive")
  ↦
  {
    {4.0} * {3.14}
  } ensuring (result => result > 0, "Circumference is positive")
  ↦
  {
    12.56
  } ensuring (result => result > 0, "Circumference is positive")
  ↦
  12.56
```

Let's verify that our hand-evaluated answer is correct:

In [7]:
```
circumference(5.0 - 3.0)
```

Out[7]: 12.56

A note on braces

As mentioned previously, curly braces `{}` can wrap any expression in Core Scala. They can be used very similarly to parentheses `()` , but have slightly different syntactic behavior. We can mostly ignore the difference in Core Scala, but I recommend reading the section on Semicolon Inference in the *Programming in Scala* book for a better description of the difference in the Scala language:

[https://www.artima.com/pins1ed/classes-and-objects.html#4.2 (https://www.artima.com/pins1ed/classes-and-objects.html#4.2)](https://www.artima.com/pins1ed/classes-and-objects.html#4.2)

Semantics of require, ensuring, and assert

We previously hand-waived the semantics for the `require` and `ensuring` constructs used for implementing preconditions and postconditions. We'll define those now, and also introduce the `assert` construct, which is used to define `ensuring` .

Error States

To help us distinguish between different error states, we'll introduce new syntax for expressing errors with descriptions in our reductions:

```
⊥("Error Message") // error with description
```

Note that whenever the error value ⊥ (pronounced "bottom") appears in an expression, the entire expression reduces to that error value.

```
... ⊥(msg) ...
↦
⊥(msg)
```

We'll introduce more flexible syntax for expression errors in a later lecture when we discuss `try` / `catch` .

require

```
{
  require(condition, message)
  trailingCode
}

condition: Boolean
message: String
```

The rules for evaluating a `require` assertion are similar to an `if` expression. We first reduce the `condition` expression to a `Boolean` value. When `condition` reduces to `true` , then the `require` is a no-op, and the whole `require` clause is reduced away. When `condition` reduces to `false` , then we then reduce the whole expression to `⊥(message)` . Note that the evaluation of `message` is deferred and contingent on the value of `condition` .

Passing Precondition

```
{
  require(true, message)
  trailingCode
}
↦
{
  trailingCode
}
```

In [8]:
```
require(true, "message")
"foo"
```

Out[8]: foo

Failing Precondition

```
{
  require(false, message)
  trailingCode
}
↦
⊥(message)
```

In [9]:
```
require(false, "message")
"foo"
```

```
java.lang.IllegalArgumentException: requirement failed: message
  at scala.Predef$.require(Predef.scala:224)
  ... 46 elided
```

assert

The semantics of `assert` in Core Scala are identical to `require`. The only difference is syntactic: `assert` comes *after* the `val` declarations in a compound expression.

```
{
  assert(condition, message)
  trailingCode
}

condition: Boolean
message: String
```

The only difference between the two constructs in the Scala language is the type of the error resulting from a failure.

Passing Assertion

```
{
  assert(true, message)
  trailingCode
}
↦
{
  trailingCode
}
```

In [10]:
```
assert(true, "message")
"foo"
```

Out[10]: foo

Failing Assertion

```
{
  assert(false, message)
  trailingCode
}
↦
⊥(message)
```

In [11]:
```
assert(false, "message")
"foo"
```

```
java.lang.AssertionError: assertion failed: message
  at scala.Predef$.assert(Predef.scala:170)
  ... 46 elided
```

ensuring

We define the `ensuring` construct in using `assert`:

```
{ value } ensuring(result => condition, message)
↦
{
  val result = {value}
  assert(condition, message)
  result
}
```

Note that if the symbol `result` is used in the `condition`, then it gets reduced to `{value}` via the `val` definition.

Passing Postcondition

In [12]:
```
{ "foo" } ensuring(result => true, "message")
```

Out[12]:
```
foo
```

Failing Postcondition

In [13]:
```
{ "foo" } ensuring(result => false, "message")
```

```
java.lang.AssertionError: assertion failed: message
  at scala.Predef$Ensuring$.ensuring$extension3(Predef.scala:261)
  ... 46 elided
```

Static Type for ensuring

The static type for an `ensuring` clause is the same as the result type of its input expression.

```
{ x } ensuring (result => false, "msg")
```

For example, given the expression above, the static type of the whole expression (including an `ensuring` clause) is the same as the static type of the expression `x`. This makes sense semantically since the purpose of the `ensuring` clause is to add assertions, not to change the result. It also follows from the new reduction rules for `ensuring` given above.

Note that even though the predicate given in the example above always evaluates to `false` (thus resulting in an error), the static type of the expression is still the type of the value that would result if the predicate evaluated to `true`. In other words, the static type of the expression does not change based on the assertion's predicate.

Type Rules and Assertions

The assertion constructs `require` and `assert` have no type in Core Scala. This follows from the fact that these constructs are not allowed inside a Core Scala simple expression.

However, in the full Scala language you *can* use `require` or `assert` in a position requring an expression, and thus these constructs have types.

In [14]:
```
def f() = require(false)
def g() = assert(false)
```

Out[14]:
```
f: ()Unit
g: ()Unit
```

We will talk about the `Unit` type later in the course, but for now you can think of it like `Void` in Java or `None` in Python.

Circumference Example Revisited

Now that we have proper reduction rules for `require` and `ensuring`, let's revisit our hand-evaluation of `circumference(5.0 - 3.0)`:

```
circumference(5.0 - 3.0)
↦
circumference(2.0)
↦
{
  require({2.0} > 0, "Radius must be positive")

  val diameter = {2.0} + {2.0}
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  diameter * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  require(true, "Radius must be positive")

  val diameter = {2.0} + {2.0}
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  diameter * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val diameter = {2.0} + {2.0}
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  diameter * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val diameter = 4.0
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  diameter * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val pi = 157.0 / 50.0  // approximation of π to 2 decimal pla
ces

  {4.0} * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val pi = 3.14
```

```
  val pi = 3.14

  {4.0} * pi
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  {4.0} * {3.14}
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  12.56
} ensuring (result => result > 0, "Circumference is positive")
↦
{
  val result = {12.56}
  assert(result > 0, "Circumference is positive")
  result
}
↦
{
  assert({12.56} > 0, "Circumference is positive")
  {12.56}
}
↦
{
  assert(true, "Circumference is positive")
  {12.56}
}
↦
{
  {12.56}
}
↦
12.56
```

No more hand-waiving! Much better.