Scala Functions

Georgia Tech

Basic Function Definition

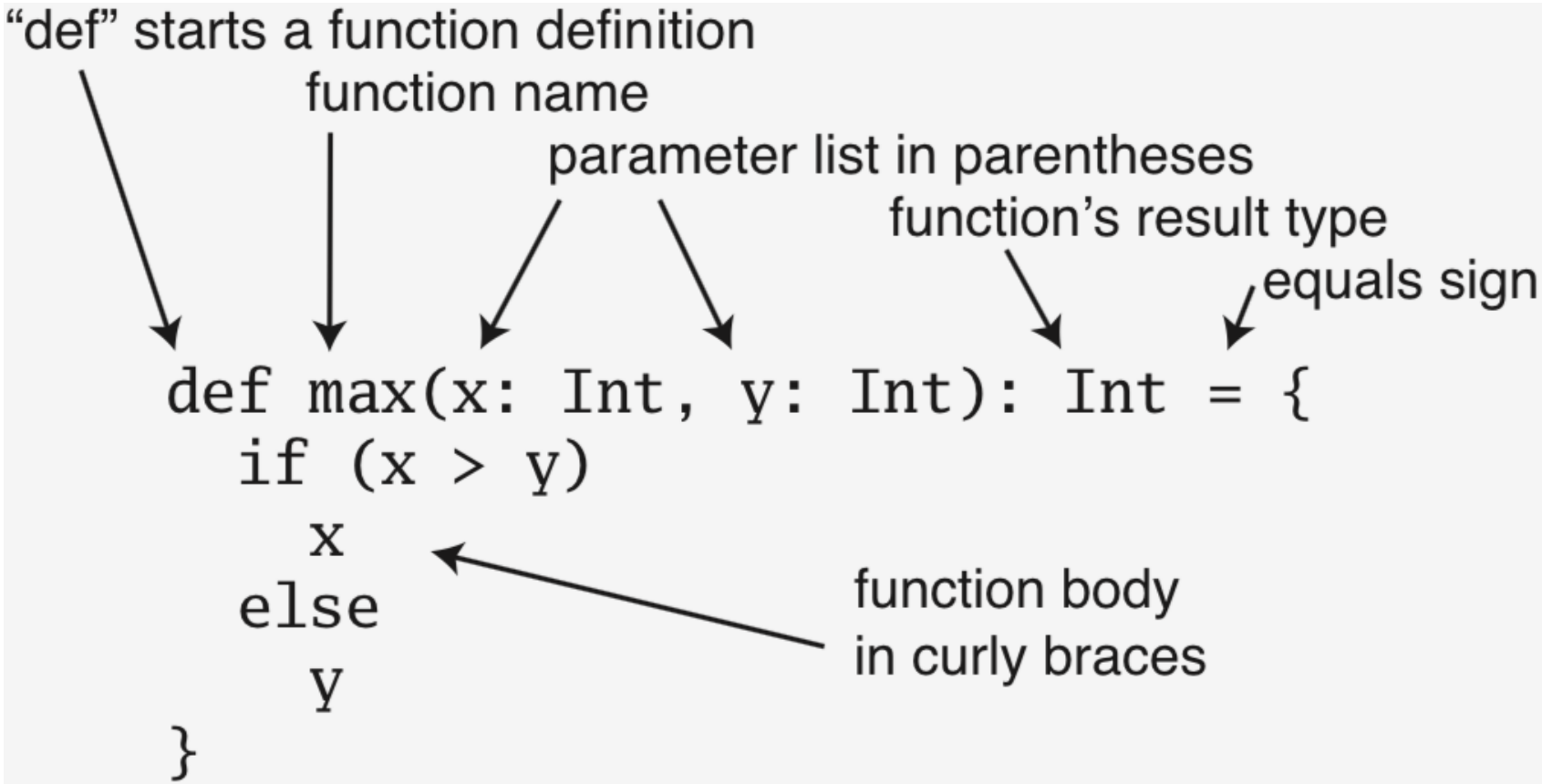


Figure 1: Scala Basic Function Definition, Programming in Scala, 3ed, page 69

Functions Return Values

Notice the mandatory = between the "header" and "body"

```
1 def double(x: Int): Int = 2 * x
```

- Also notice that you don't need {} if body is single expression

A function that doesn't return a useful value is called a procedure and returns the special value () of type Unit. Style guide says always annotate return type of procedures

```
1 def say(something: String): Unit = {
2   println(something)
3 }
```

Georgia Tech

Local Functions

You can nest functions within functions. Here iter can only be called within facIter

```
1  def facIter(n: BigInt): BigInt = {
2    def iter(i: BigInt, accum: BigInt): BigInt =
3      if (i <= 1) accum
4      else iter(i - 1, i * accum)
5    require(n >= 0, "Factorial defined for non-negative integers")
6    iter(n, 1)
7  }
```

`require` takes a `Boolean` expression and an optional `String` description. If `Boolean` expression is `false`, throws an `IllegalArgumentException` with the description as the exception message

Georgia Tech

Functions are First Class

First class values in a programming language can be

- stored in variables
- passed as arguments to functions, and
- returned from functions

Georgia Tech

Function Literals

Just as other types have literal values, function values can be created with literals

```
1 val doubleFun: Int => Int = {(x: Int) => {2 * x}}
```

- Notice the type annotation. `doubleFun` is a function with a domain of `Int` and codomain of `Int`

Above is full literal notation. What can be inferred can be left off. Could be written as

```
1 val doubleFun:Int => Int = x => 2 * x
```

or

```
1 val doubleFun = (x: Int) => 2 * x
```

Georgia Tech

Higher-Order Functions

- A first order function takes non-function value parameters and returns a non-function value
- A higher-order function takes function value parameters or returns a function value
- Function literals are most useful as arguments to higher-order functions `List.filter` takes a function of one parameter of the list's element type and returns a `Boolean`

```
1 val evens = List(1,2,3,4,5,6,7,8).filter(x => x % 2 == 0)
```

If each parameter appears once in the function literal's body, can use placeholder syntax

```
1 val evens2 = List(1,2,3,4,5,6,7,8).filter(_ % 2 == 0)
```

Georgia Tech

Repeated Parameters

Repeated parameters, or "var-args" parameters, are annotated with a * after the type

```
1 def max(x: Int, xs: Int*): Int = { xs.foldLeft(x)((x, y) => if (x > y)
      x else y)
2 }
```

Must pass a multiple single arguments to a repeated parameter

```
1 val varArgsMax = max(3, 5, 7, 1)
```

- In application of `max` above, `x` is `3`, `xs` is `Array(5, 7, 1)`

To pass a sequence to a varargs parameter, use `: _*`

```
1 val seqMax = max(0, List(2, 4, 6, 8, 0): _*)
```

Georgia Tech

Functional Function Evaluation

The result of a pure function depends only on its inputs

A pure function is referentially transparent, i.e., a function application can be replaced with the value it produces without changing the meaning of the program

Application of pure functions to their arguments can be understood with the substitution model of evaluation:

1. Evaluate arguments left to right
2. Replace function call with function body, substituting arguments for parameters in body

Georgia Tech

Recursive Function Evaluation

```
1  def fac(n: Int): Int = if (n <= 1) 1 else n * fac(n - 1)
```

Applying the steps of applicative-order evaluation gives:

[5/n]fac(n) ($[v_1/p_1, ...v_n/p_n]expr$ means substitute v_i for p_i in $expr$)

- => fac(5)
- => 5 * fac(4)
- => 5 * 4 * fac(3)
- => 5 * 4 * 3 * fac(2)
- => 5 * 4 * 3 * 2 * fac(1)
- => 5 * 4 * 3 * 2 * 1
- => 5 * 4 * 3 * 2
- => 5 * 4 * 6
- => 5 * 24
- => 120

Notice the expanding-contracting pattern. This mirrors stack usage – calling `fac` with a large argument will overflow the stack

Iterative Recursive Functions Evaluation

Recursive calls in tail position are turned into loops (only one stack frame is used). This is called tail call optimization

`facIter` uses an iterative local function whose recursive call is in tail position

```
1  def facIter(n: BigInt): BigInt = {
2    def iter(i: BigInt, accum: BigInt): BigInt =
3      if (i <= 1) accum
4      else iter(i - 1, i * accum)
5    iter(n, 1)
6  }
```

Iterative Recursive Functions Evaluation

```
1  def facIter(n: BigInt): BigInt = {
2    def iter(i: BigInt, accum: BigInt): BigInt =
3      if (i <= 1) accum
4      else iter(i - 1, i * accum)
5    iter(n, 1)
6  }
```

[5/n]facIter(n)

- => iter(5, 1)

[5/i, 1/accum]iter(i, accum)

- => iter(5, 1)
- => iter(4, 5)
- => iter(3, 20)
- => iter(2, 60)
- => iter(1, 120)
- => 120

% Scala Functional Abstraction

Georgia Tech

Functional Lists

Scala's list type has an API familiar to Java programmers, and an API modeled on the original cons list in Lisp, which is an elegant representation of linked lists. Recall that one way to create a list in Scala is to use the `::` operator (pronounced "cons"):

```
1  scala> var xs = 1::Nil
2  xs: List[Int] = List(1)
3
4  scala> xs = 2::xs
5  xs: List[Int] = List(2, 1)
6
7  scala> xs = 3::xs
8  xs: List[Int] = List(3, 2, 1)
```

Notice that you add elements to the head of the list. The special value `Nil` represents an empty node which signals the end of the list, which you can also think of as a list with no elements because it contains on value and doesn't point to a successor node.

Georgia Tech

Linked List Structure

The code on the previous slide produces a list that looks like:

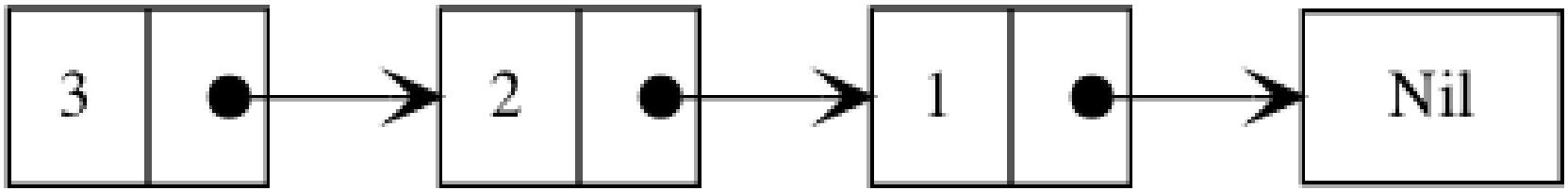


Figure 2: A singly-linked list.

Each node is a cons cell that contains an element, and a link to the rest of the list. The `head` and `tail` methods return these two components of the first cons cell in the list.

```
1  scala> xs.head
2  res2: Int = 3
3
4  scala> xs.tail
5  res3: List[Int] = List(2, 1)
```

Georgia Tech

The End of the List

- `isEmpty` is equivalent to comparison to `Nil`.

```
1  scala> val leer = List()
2  leer: List[Nothing] = List()
3
4  scala> leer.isEmpty
5  res4: Boolean = true
6
7  scala> leer == Nil
8  res5: Boolean = true
```

Georgia Tech

Functional List Idioms

A common functional idiom for processing a List uses only

- the three primary first-order methods `head`, `tail`, and `isEmpty`
- `if` expressions, and
- recursion

Here's a function to generate a `String` representation of a list:

```
1  def listToString[T](list: List[T]): String = {
2    def toStringHelper(list: List[T], accum: String): String =
3      // Nil is the end of a list, base case for recursion
4      if (list == Nil) accum
5      // Recurse on the tail of the list, accumulate result
6      else toStringHelper(list.tail, accum + list.head)
7    toStringHelper(list, "")
8  }
```

As an exercise, use the substitution model to evaluate `listToString(List("R", "E", "S", "P", "E", "C", "T"))` with pencil and paper.

Georgia Tech

Function Values

Function values, like all values in Scala, are instances of classes.

Function1, . . . , Function22 [^1]

[^1] The FunctionN classes and the 22 limit are going away in Scala 3.

Georgia Tech

Closures

```
1  def makeDecorator(
2      leftBrace: String,
3      rightBrace: String): String => String =
4    (middle: String) => leftBrace + middle + rightBrace
5
6  val squareBracketer = makeDecorator("[", "]")
```

In the function literal

```
1  (middle: String) => leftBrace + middle + rightBrace
```

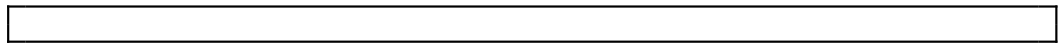
- `middle` is *bound* variable because it's in the parameter list
- `leftBrace` and `rightBrace` are *free* variables

A function literal with only bound variables is called a closed term.

A function literal with free variables is called an open term becuase values for the free variables must be captures from an enclosing environment, thereby *closing* the term.

Georgia Tech

Abstractions with Higher-order Functions

- 

Partial Application

Partially Applied Functions

A `def` is not a function value.

```
1 def dubbel(x: String): String = s"two ${x}s"
2
3 // Won't compile because dubbel is not a function value
4 val wontCompile = dubbel
```

To turn the dubbel method in to a Function value, partially apply it

```
1 val dubbelFun = dubbel _
```

Don't forget the space between the name of the function and the underscore.

The partial function application above is equivalent to:

```
1 val dubbelFun = (x: String) => dubbel(x)
```

Georgia Tech

Partial Function Short Forms

You can leave off the underscore if target type is a function. These three are equivalent

```
1 List("Honey", "Boo", "Boo").foreach(x => print(x))
2 List("Honey", "Boo", "Boo").foreach(print _)
3 List("Honey", "Boo", "Boo").foreach(print)
```

- The third example above works because `foreach` takes a function value, so print is lifted to a function (another term for partial function application)

Note that this form is not technically a partially applied function, it's just a short-form of a function literal using placeholder syntax:

```
1 List("Honey", "Boo", "Boo").foreach(print(_))
```

Georgia Tech

Schönfinkeling, a.k.a., Currying

Scala syntax for curried functions: multiple param lists

```
1 def curry(chicken: String)(howard: String): String =
2   s"Love that $chicken from $howard!"
```

Above is equivalent to:

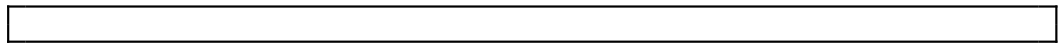
```
1 def explicitCurry(chicken: String): String => String =
2  (howard: String) => s"Love that $chicken from $howard!"
```

You can partially apply second parameter list to get another function

```
1 val eleganceFrom = curry("elegence")_
2 eleganceFrom("provability")
```

Georgia Tech

Control Abstraction with Higher-Order Functions

- 

By-Name Parameters