Scala Function Basics

Georgia Tech

Basic Function Definition

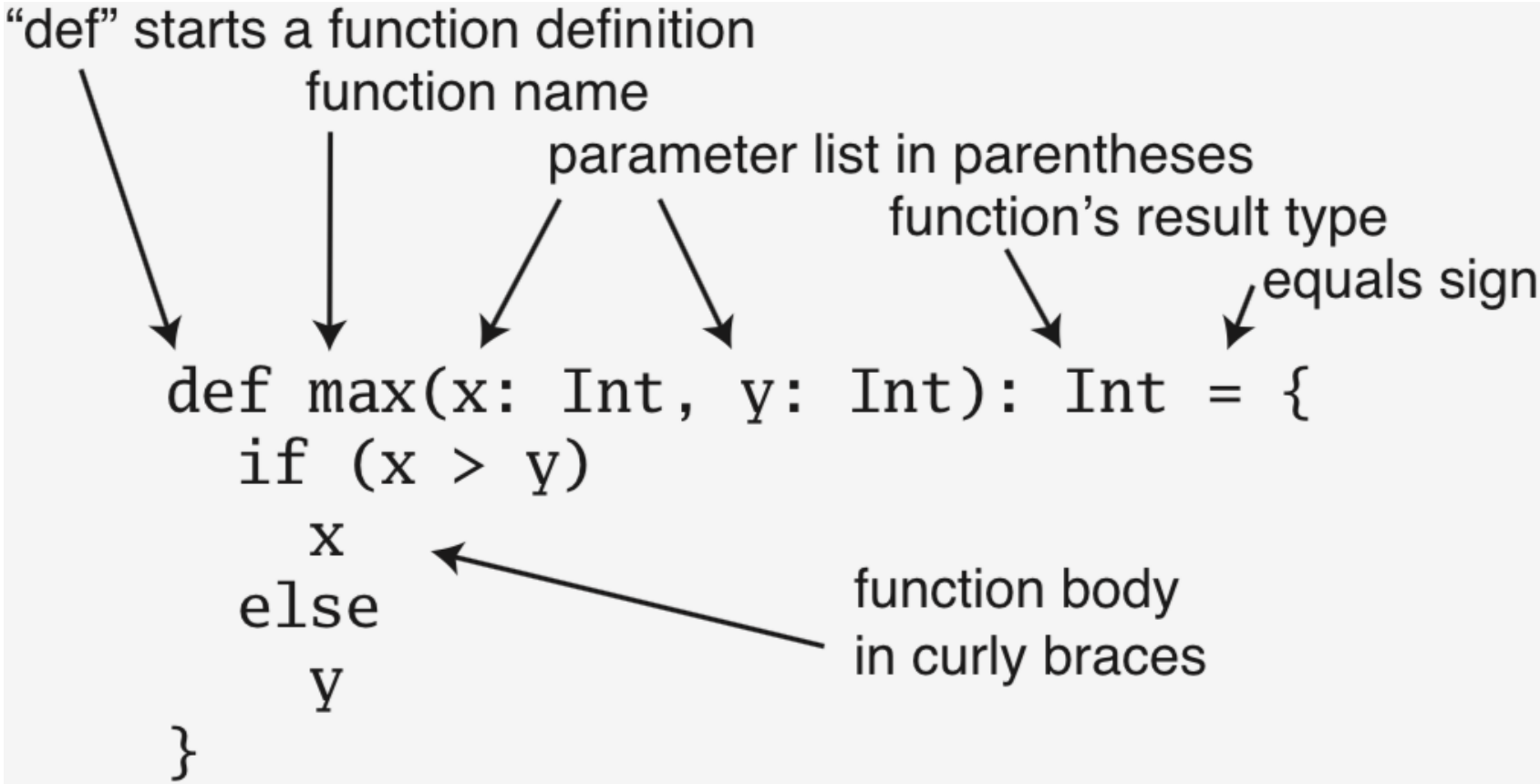

Figure 1: Scala Basic Function Definition, Programming in Scala, 3ed, page 69

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,
6        "Factorial only defined for non-negative integers")
7    iter(n, 1)
8  }
```

`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

Georgia Tech

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

Georgia Tech

Conclusion

You now know

- the basics of Scala functions,
- the substitution model of function evaluation,

Georgia Tech