Week 2 : Evaluating a Function Application (Review)

A simple rule : One evaluates a function application $f(e_1, ..., e_n)$

- by evaluating the expressions $e_1, ..., e_n$ resulting in the values $v_1, ..., v_n$, then
- by replacing the application with the body of the function f, in which
- the actual parameters $v_1, ..., v_n$ replace the formal parameters of f.

This can be formalized as a rewriting of the program itself:

 $oldsymbol{def} f\left(x_{1},\,...,\,x_{n}
ight) =B\;;\,...\;f\left(v_{1},\,...,\,v_{n}
ight)$

$$oldsymbol{def} f\left(x_{1},\,...,\,x_{n}
ight) =B\ ;$$
 ... $\left[\mathrm{v}_{1}/\mathrm{x}_{1},\,...,\,\mathrm{v}_{n}/\mathrm{x}_{n}
ight] B$

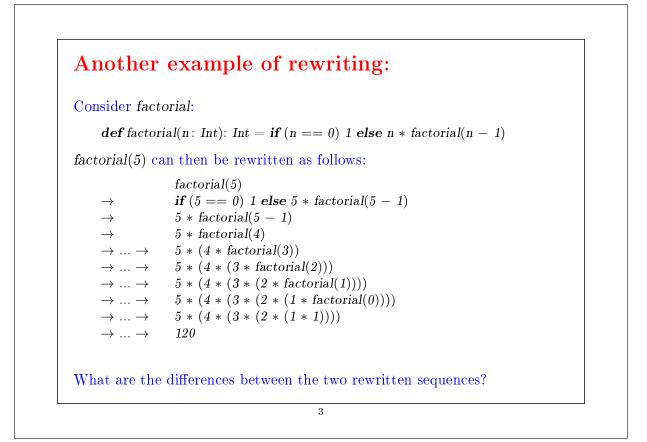
Here, $[v_1/x_1, ..., v_n/x_n] B$ denotes the expression B in which all occurences of x_i have been replaced by v_i .

 $[v_1/x_1, ..., v_n/x_n]$ is called a *substitution*.

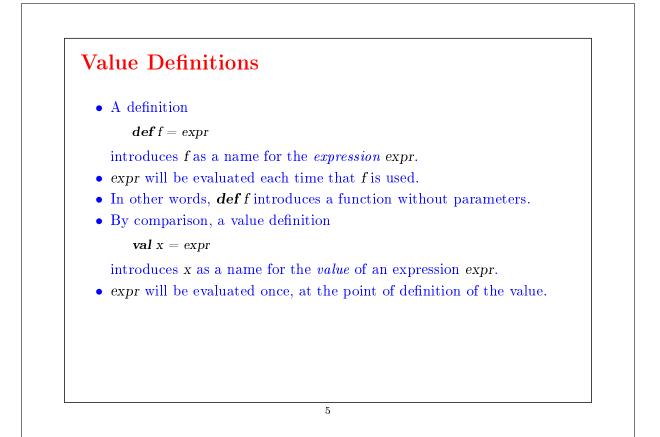
 \rightarrow

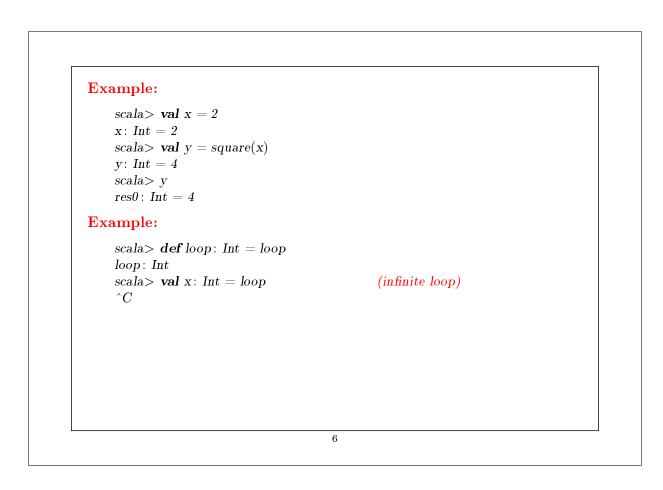
1

Example of rewriting: Consider gcd: def gcd(a: Int, b: Int): Int = if(b == 0) a else gcd(b, a % b)gcd(14, 21) Evaluated as follows : gcd(14, 21)if (21 == 0) 14 else gcd(21, 14 % 21)if (false) 14 else gcd(21, 14 % 21) gcd(21, 14 % 21) gcd(21, 14)if (14 == 0) 21 else gcd(14, 21 % 14) \rightarrow gcd(14, 21 % 14)gcd(14, 7) \rightarrow if (7 == 0) 14 else gcd(7, 14 % 7)gcd(7, 14 % 7)gcd(7, 0)if (0 == 0) 7 else gcd(0, 7% 0) \rightarrow 7 $\rightarrow \rightarrow$



Tail Recursion Implementation Detail : If a function calls itself as its last action, the function's stack frame can be reused. This is called *tail recursion*. ⇒ Tail recursive functions are iterative processes. In general, if the last action of a function consists of calling a function (which may be the same), one stack frame is sufficient for both functions. Such calls are called, *tail-calls*. Exercise: Design a tail recursive version of *factorial*.





Higher-Order Functions

Functional languages treat functions as *first-class values*.

This means that, like any other value, a function can be passed as a parameter and returned as a result.

This provides a flexible way to compose programs.

Functions that take other functions as parameters or that return functions as results are called *higher order functions*.

 $\overline{7}$

Example:

Take the sum of the integers between a and b:

 $\begin{array}{l} \textbf{def } sumInts(a: Int, \ b: Int): \ Double = \\ \textbf{if} \ (a > b) \ 0 \ \textbf{else} \ a + \ sumInts(a + 1, \ b) \end{array}$

Take the sum of the cubes of all the integers between a and b:

def cube(x: Int): Double = x * x * xdef sumCubes(a: Int, b: Int): Double = if (a > b) 0 else cube(a) + sumCubes(a + 1, b)

Take the sum of the reciprocals of the integers between a and b:

 $\begin{array}{l} \textbf{def sum} Reciprocals(a \colon Int, \ b \colon Int) \colon Double = \\ \textbf{if} \ (a > b) \ 0 \ \textbf{else} \ 1.0 \ / \ a + \ sum Reciprocals(a + 1, \ b) \end{array}$

These are special cases of $\sum_{n=a}^{b} f(n)$ for different values of f.

Can we factor out the common pattern?

Summing with Higher-Order Functions We define: $def sum(f: Int \Rightarrow Double, a: Int, b: Int): Double = \{$ **if** (a > b) 0else f(a) + sum(f, a + 1, b)We can then write: def sumInts(a: Int, b: Int): Double = sum(id, a, b)def sumCubes(a: Int, b: Int): Double = sum(cube, a, b) def sumReciprocals(a: Int, b: Int): Double = sum(reciprocal, a, b) where def id(x: Int): Double = xdef cube(x: Int): Double = x * x * xdef reciprocal(x: Int): Double = 1.0/xThe type $Int \Rightarrow Double$ is the type of a function that takes one argument of type Int and returns a result of type Double. 9

<section-header><list-item><list-item><list-item><list-item><list-item><list-item><list-item>



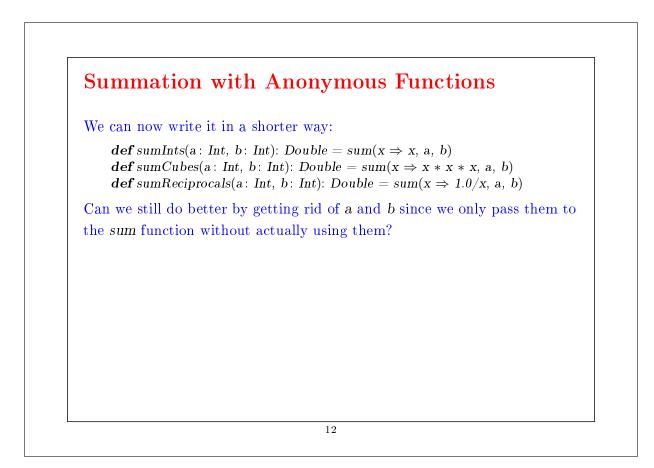
- In general, $(x_1: T_1, ..., x_n: T_n) \Rightarrow E$ is a function that relates the result of the expression E to the parameters $x_1, ..., x_n$ (such that E can refer to $x_1, ..., x_n$).
- An anonymous function $(x_1: T_1, ..., x_n: T_n) \Rightarrow E$ can always be expressed by using **def** as follows:

 $\{ def f (x_1: T_1, ..., x_n: T_n) = E; f \}$

where f is a fresh name (not yet used in the program).

• We say that anonymous functions are *syntactic sugar*.

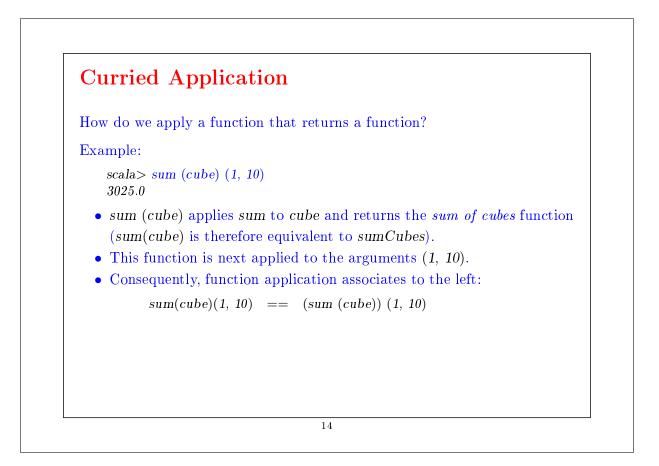




Currying

```
We rewrite sum as follows.
def sum(f: Int ⇒ Double): (Int, Int) ⇒ Double = {
    def sumF(a: Int, b: Int): Double =
        if (a > b) 0
        else f(a) + sumF(a + 1, b)
        sumF
    }
    sum is now a function that returns another function. More precisely,
    the specialized sum function sumF applies the function and sums the
    results. We can then define:
        def sumInts = sum(x ⇒ x)
        def sumCubes = sum(x ⇒ x * x * x)
        def sumReciprocals = sum(x ⇒ 1.0/x)
    These functions can be applied like the other functions:
        scala> sumCubes(1, 10) + sumReciprocals(10, 20)
```

13



Definition of Currying

The definition of functions that return functions is so useful in functional programming (FP) that there is a special syntax for it in Scala.

For example, the following definition of sum is equivalent to what we saw before, but shorter:

 $def sum(f: Int \Rightarrow Double)(a: Int, b: Int): Double = if (a > b) 0 else f(a) + sum(f)(a + 1, b)$

In general, a definition of a curried function

 $def f (args_1) \dots (args_n) = E$

where n > 1, is equivalent to

 $def f (args_1) \dots (args_{n-1}) = (def g (args_n) = E ; g)$

where g is a fresh identifier.

15

Or for short: $def f (args_1) \dots (args_{n-1}) = (args_n \Rightarrow E)$ By repeating the process n times $def f (args_1) \dots (args_{n-1}) (args_n) = E$ becomes equivalent to $def f = (args_1 \Rightarrow (args_2 \Rightarrow \dots (args_n \Rightarrow E) \dots))$ This style of definition and function application is called currying, named for its instigator, Haskell Brooks Curry (1900-1982), a twentieth century logician. In fact, the idea goes back to Moses Schönfinkel, but the word "currying" has won (perhaps because "schönfinkeling" is more difficult to pronounce).

Function Types

Question : Given,

 $def sum(f: Int \Rightarrow Double)(a: Int, b: Int): Double = \dots$

What is the type of sum ?

Note that functional types associate to the right. That is to say that

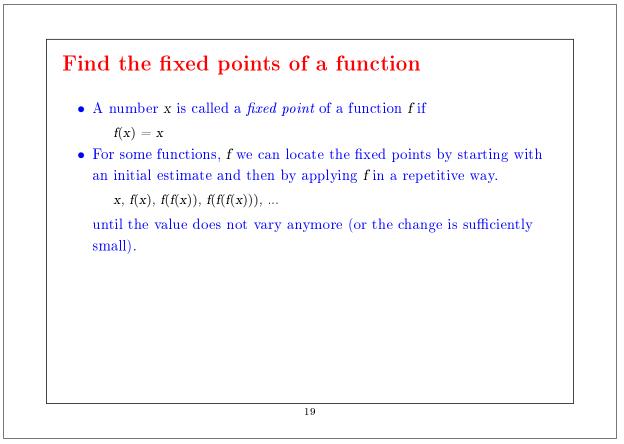
 $Int \Rightarrow Int \Rightarrow Int$

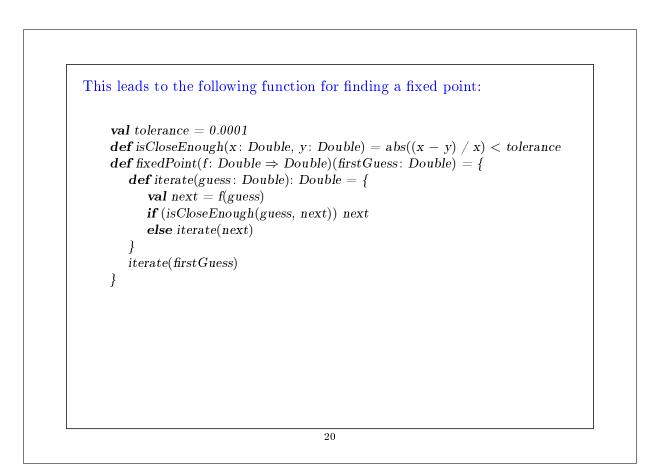
is equivalent to

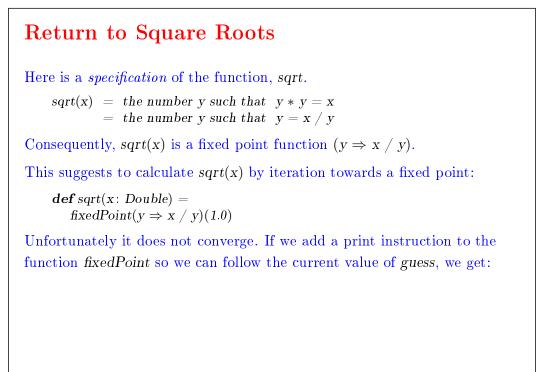
 $\mathit{Int} \Rightarrow (\mathit{Int} \Rightarrow \mathit{Int})$

17

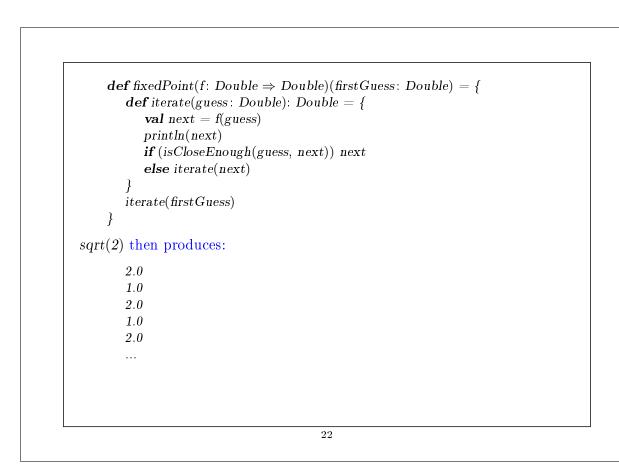
Exercises: 1. The sum function uses linear recursion. Can you write a tail-recursive version by replacing the ??? $def sum(f: Int \Rightarrow Double)(a: Int, b: Int): Double = \{$ def iter(a: Int, result: Double): Double = { **if** (??) ?? **else** iter(??, ??) } iter(??, ??) } 2. Write a product function that calculates the product of the values of a function for the points on a given interval. 3. Write factorial in terms of product. 4. Can you write a more general function, which generalizes both sum and product? 18

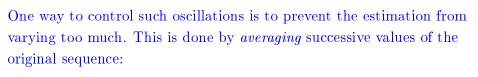






```
21
```

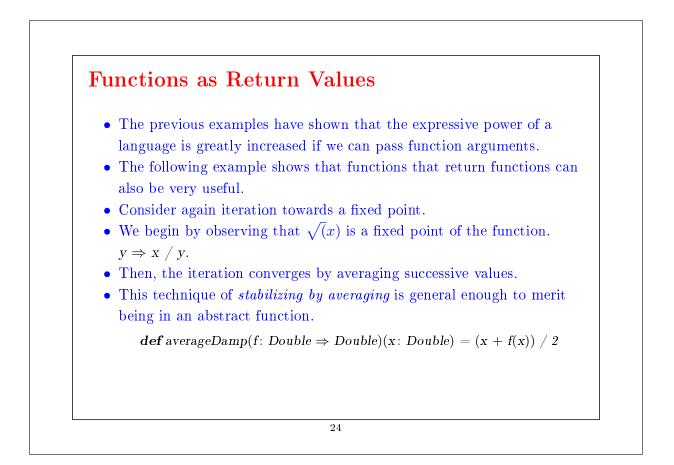


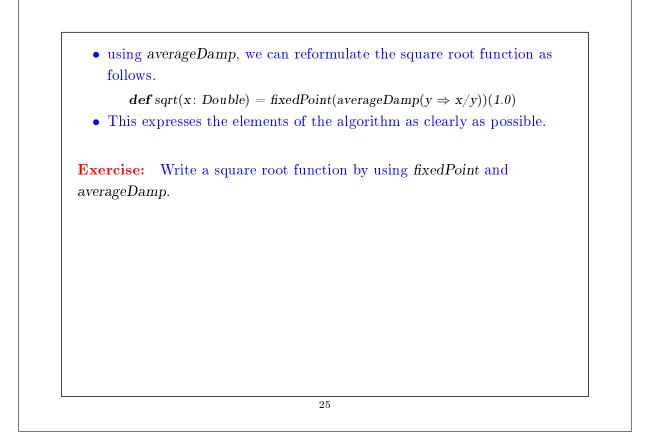


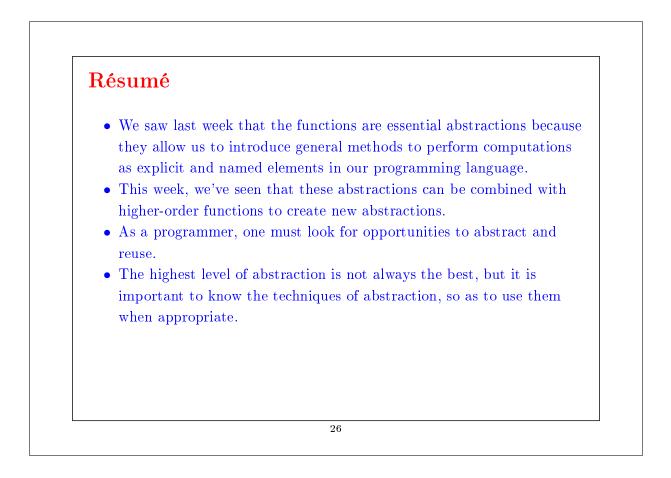
 $scala > def sqrt(x: Double) = fixedPoint(y \Rightarrow (y + x / y) / 2)(1.0)$ scala > sqrt(2.0)1.5 1.4166666666666666 1.4142156862745097 1.4142135623746899 1.4142135623746899

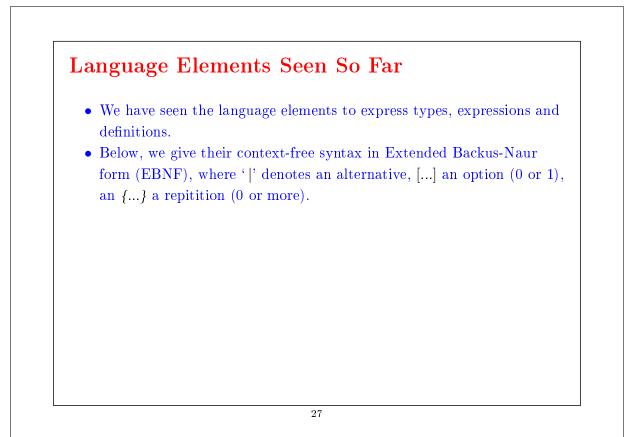
In fact, if we fold the fixed point function *fixedPoint* we find the same square root function that we found last week.





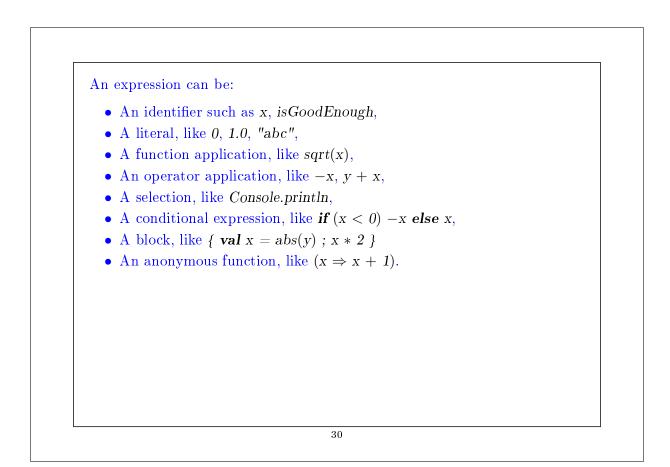






$\begin{array}{llllllllllllllllllllllllllllllllllll$	Гуреs :	
 A type can be: A numeric type: Int, Double (and Byte, Short, Char, Long, Float), The Boolean type with the values true and false, The String type, 	FunctionType SimpleType	$e = SimpleType '\Rightarrow' Type '(' [Types] ')' '\Rightarrow' Type$ = $Byte Short Char Int Long Double Float$ $Boolean String$
 A numeric type: Int, Double (and Byte, Short, Char, Long, Float), The Boolean type with the values true and false, The String type, 	Types	$= Type \ \{`, `Type\}$
 The Boolean type with the values true and false, The String type, 	A type can be:	
	 The Boolean The String to the string to	n type with the values true and false , type,

Expressions: Expr = InfixExpr | FunctionExpr | **if** '(' Expr ')' Expr **else** Expr InfixExpr= PrefixExpr | InfixExpr Operator InfixExpr Operator = ident $= ['+' | '-' | '!' | '^{-'}]$ SimpleExpr PrefixExpr SimpleExpr= ident | literal | SimpleExpr '.' ident | Block = Bindings ' \Rightarrow ' Expr FunctionExpr = ident [':' SimpleType] | '(' [Binding {',' Binding}] ')' Bindings Binding = ident [':' Type]Block $= {}^{(1)} { {Def ';' } Expr '}'$ 29



Definitions:

```
Def = FunDef | ValDef
FunDef = def ident ['(' [Parameters] ')'] [':' Type] '=' Expr
ValDef = val ident [': Type] '=' Expr
Parameter = ident ':' [ '⇒' ] Type
Parameters = Parameter {', ' Parameter}
A definition can be:
• A function definition like def square(x : Int) = x * x
• A value definition like val y = square(2)
```