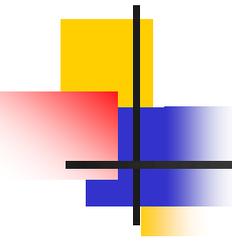


Functional Programming

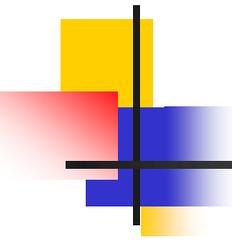
Slides taken from
http://turing.cs.pub.ro/fp_08



Lecture No. 12

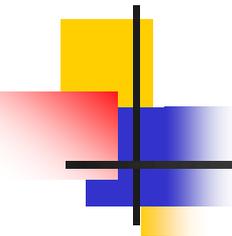
Analysis and Efficiency of Functional Programs

- Reduction strategies and lazy evaluation



1. Reduction strategies and lazy evaluation

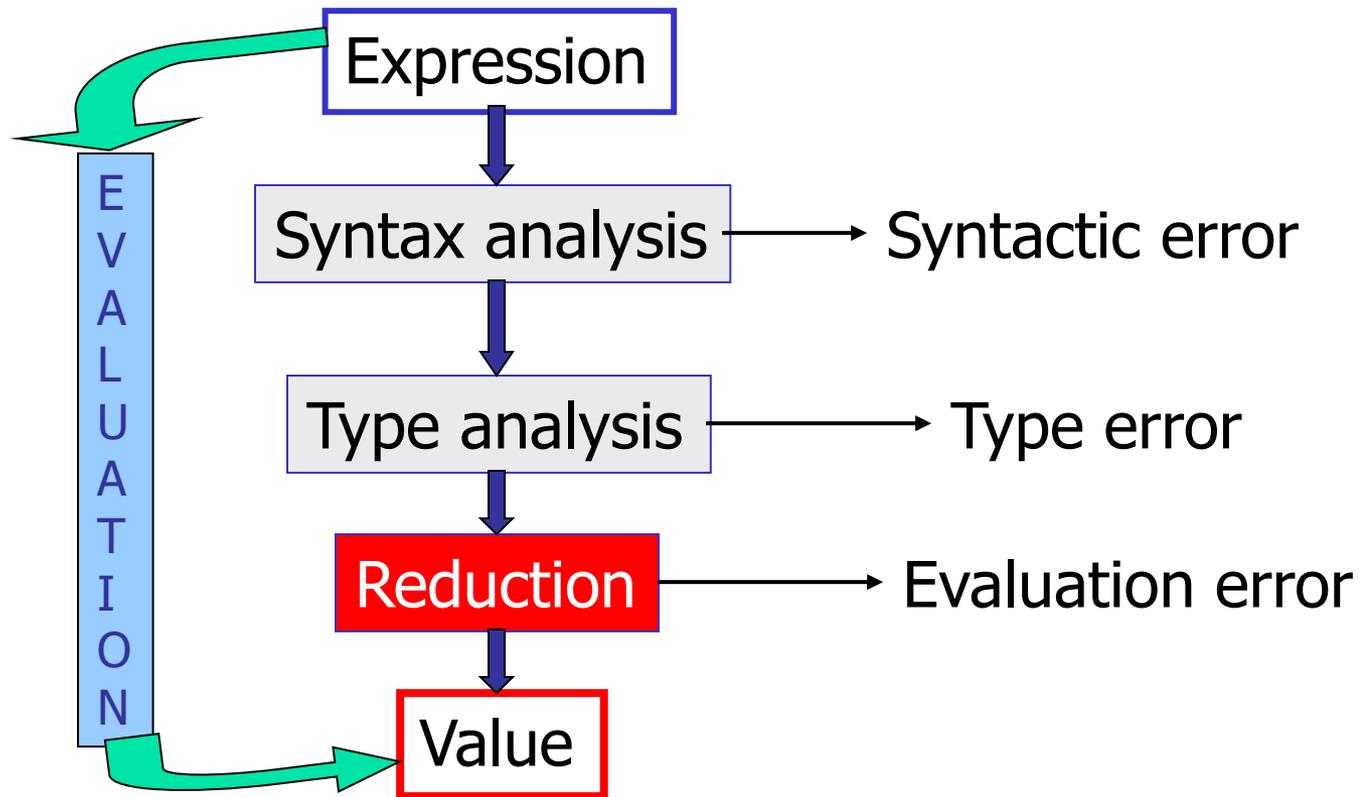
- Programming is not only about writing correct programs but also about **writing fast ones that require little memory**
- Aim: **how functional programs are commonly executed on a real computer** → a foundation for analyzing time and space usage
- Every implementationmore or less closely **follows the execution model of *lazy evaluation* (οκνηρή αποτίμηση).**



1.1 Reduction

- Executing a functional program, i.e. evaluating an expression \Rightarrow means to **repeatedly apply function definitions** until all function applications have been expanded
- Implementations of modern **Functional Programming** languages are based on a simplification technique called **reduction**.

Evaluation in a strongly typed language



- The **evaluation** of an **expression** passes through **three stages**
- Only those expression which are syntactically correct and well typed are submitted for **reduction**

What is reduction?

- Given a set of rules R_1, R_2, \dots, R_n (called reduction rules)
- and an expression e
- **Reduction** is the process of repeatedly simplifying e using the given reduction rules

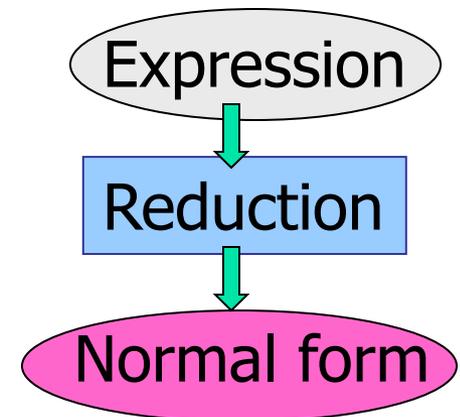
$$\begin{aligned} e &\Rightarrow e_1 \quad R_{i1} \\ &\Rightarrow e_2 \quad R_{i2} \end{aligned}$$

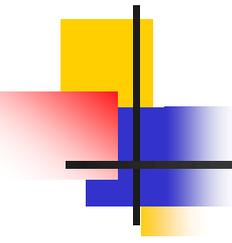
....

$$\Rightarrow e_k \quad R_{ik}, \quad R_{ij} \in \{R_1, R_2, \dots, R_n\}$$

until no rule is applicable

- e_k is called the **normal form** of e
- It is the simplest form of e





2 types of reduction rules

- Built-in rules:

addition, subtraction, multiplication, division

- User supplied rules:

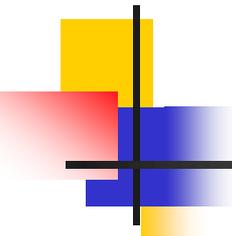
square x = $x * x$

double x = $x + x$

sum [] = 0

sum ($x:xs$) = $x + \text{sum } [x+1:xs]$

f $x y$ = square $x + \text{square } y$

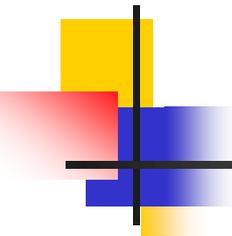


1.2 Reduction at work

- Each reduction step replaces a subexpression by an equivalent expression by applying one of the 2 types of rules

$$f\ x\ y \quad = \text{square } x + \text{square } y$$

$$\begin{aligned} f\ 3\ 4 &\Rightarrow (\text{square } 3) + (\text{square } 4) && (f) \\ &\Rightarrow (3*3) + (\text{square } 4) && (\text{square}) \\ &\Rightarrow 9 + (\text{square } 4) && (*) \\ &\Rightarrow 9 + (4*4) && (\text{square}) \\ &\Rightarrow 9 + 16 && (*) \\ &\Rightarrow 25 && (+) \end{aligned}$$



Reduction rules

- Every reduction replaces a subexpression, called **reducible expression** or **redex** for short, with an equivalent one, either by appealing to a **function definition** (like for square) or by using a **built-in function** like (+).
- *An expression without redexes is said to be in **normal form**.*
- The fewer reductions that have to be performed, the faster the program runs.
- **We cannot expect each reduction step to take the same amount of time** because its implementation on real hardware looks very different, but in terms of **asymptotic complexity**, this **number of reductions** is an accurate measure.

Alternate reductions

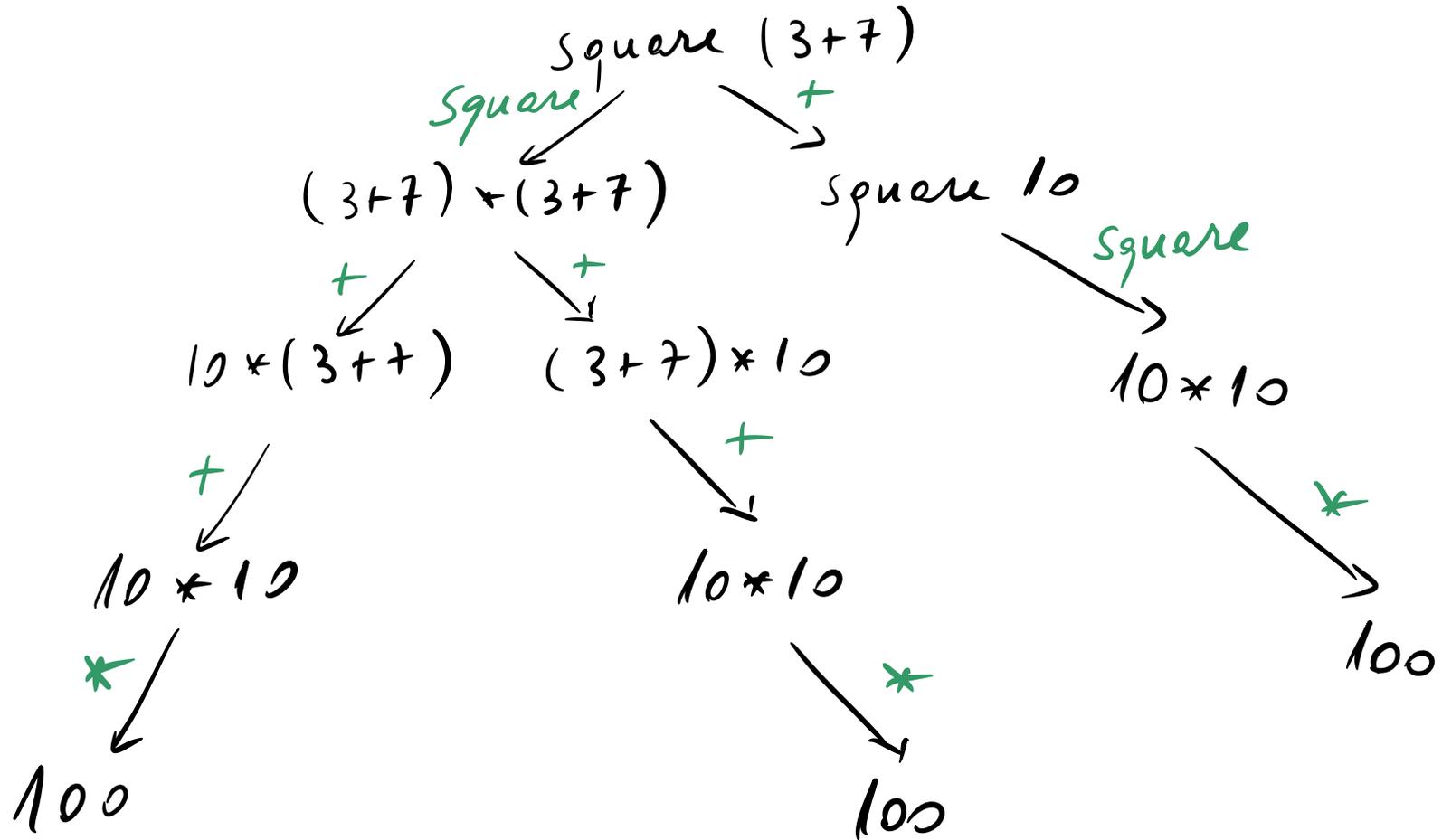
square (3+7)

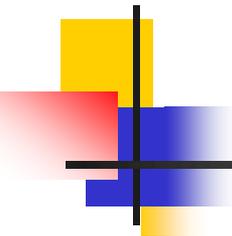
$$\begin{aligned} & \text{square}(3+7) \\ &= \text{square}(10) \quad (+) \\ &= 10 * 10 \quad \text{square} \\ &= 100 \quad (*) \\ & \quad \underbrace{\hspace{2em}} \\ & \text{Normal form} \end{aligned}$$

square (3+7)

$$\begin{aligned} & \text{square}(3+7) \\ &= (3+7) * (3+7) \quad (\text{square}) \\ &= 10 * (3+7) \quad (+) \\ &= 10 * 10 \quad (+) \\ &= 100 \quad (*) \\ & \quad \underbrace{\hspace{2em}} \\ & \text{Normal form} \end{aligned} \quad \begin{array}{l} \text{Reduction} \\ \text{rules} \end{array}$$

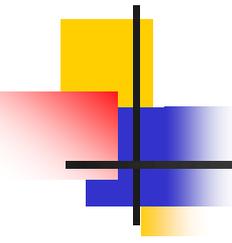
Possible ways of evaluating square





Comments

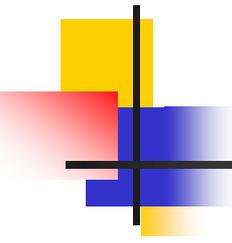
- There are usually **several ways** for reducing a given expression to normal form
- Each way corresponds to **a route in the evaluation tree**: from the root (original expression) to a leaf (reduced expression)
- There are *three different ways* for reducing *square (3+7)*
- **Questions:**
 - Are all the answers obtained by following distinct routes identical?
 - Which is the best route?
 - Can we find an algorithm which always follows the best route?



Q&A

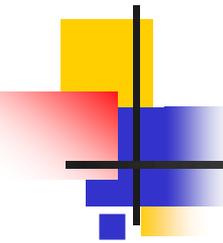
- **Q: Are all the values obtained by following distinct routes identical?**
- A: If two values are obtained by following two different routes, then these values must be identical

- **Q: Which is the best route?**
- Ideally, we are looking for the shortest route. Because this will take the least number of reduction steps and, therefore, is the most efficient.



Q&A

- **Q: Can we find an algorithm which always follows the best route?**
- In any tree of possible evaluations, there are usually two extremely interesting routes based on:
 - An **Eager** Evaluation Strategy (**πρόθυμη αποτίμηση**)
 - A **Lazy** Evaluation Strategy (**Οκνηρή αποτίμηση**)



1.3 Eager evaluation

Given an expression such as:

$$f a$$

where f is a function and a is an argument.

- The Lazy Evaluation strategy reduces such an expression by attempting to **apply the definition of f first**.
- The Eager Evaluation Strategy reduces this expression by **attempting to simplify the argument a first**.

Example of Eager Evaluation

$$\begin{aligned} & \text{square}(3+7) \\ &= \text{square}(10) \\ &= 10 * 10 \\ &= 100 \end{aligned}$$

By (+)
By square
By *

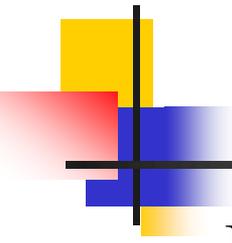
} Reduction rules

Example of Lazy Evaluation

$$\begin{aligned} & \text{Square}(3+7) \\ &= (3+7) * (3+7) \\ &= 10 * (3+7) \\ &= 10 * 10 \\ &= 100 \end{aligned}$$

By Square
By +
By +
By *

Reduction rules



1.4 A&D of reduction strategies

- **LAZY EVALUATION** = **Outer-most** Reduction Strategy
 - Reduces **outermost redexes** = redexes that are not inside another redex.
- **EAGER EVALUATION** = **Inner-most** Reduction Strategy
 - Reduces **innermost redexes**
 - An **innermost redex** is a redex that has no other redex as subexpression inside.
- Advantages and drawbacks

Repeated Reductions of Subexpressions

Eager eval

$$\begin{aligned} &= \text{square } 10 \\ &= 10 * 10 \\ &= 100 \end{aligned}$$

square (3+7)

Lazy eval

$$\begin{aligned} &= (3+7) * (3+7) \\ &= 10 * (3+7) \\ &= 10 * 10 \\ &= 100 \end{aligned}$$

- Eager Evaluation is better because it did not repeat the reduction of the subexpression $(3+7)$!

Repeated Reductions of Subexpressions

square (sum [1:100])

= square 5050 *100 times +*
*= 5050 * 5050* *square*
= 25502500 ***

*= sum [1..100] * sum [1..100]*
*= 5050 * sum [1..100]* *square*
*= 5050 * 5050* *100 times +*
= 25502500 *100 times +*

- Eager Evaluation requires 102 reductions ***
- Lazy Evaluation requires 202 reductions
- The Eager Strategy did not repeat the reduction of the subexpression *sum [1..100]*!

Performing Redundant Computations

■ Lazy evaluation is better as it avoids performing redundant computations

■ first (2+2, square 15)

■ Lazy Evaluation

⇒ 2 + 2

(first)

⇒ 4

(+)

■ Eager Evaluation

⇒ first (4, square 15)

(+)

⇒ first(4, 15*15)

(square)

⇒ first(4, 225)

(*)

⇒ 4

(first)

Termination

- For some expressions like $\text{loop} = 1 + \text{loop}$ no reduction sequence may terminate; they do not have a normal form.
- But there are also expressions where some reduction sequences terminate and some do not

- $\text{first}(5, 1 / 0)$

- **Lazy Evaluation**

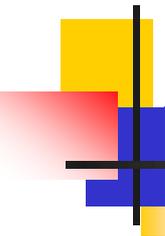
$\Rightarrow 5$ (first)

- **Eager Evaluation**

$\Rightarrow \text{first}(5, \text{bottom})$ (/)

$\Rightarrow \text{bottom}$ (Attempts to compute $1/0$)

■ Lazy evaluation is better as it avoids infinite loops in some cases



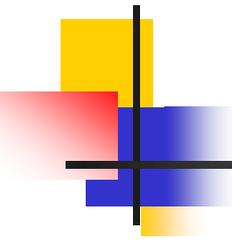
Eager evaluation (Πρόθυμη Αποτίμηση)

- **Advantages:**

- Repeated reductions of sub-expressions is avoided.

- **Drawbacks:**

- Have to evaluate all the parameters in a function call, whether or not they are required to produce the final result.
- It may not terminate.



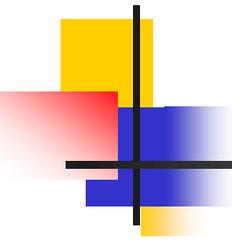
Lazy evaluation (Οκνηρή Αποτίμηση)

- **Advantages:**

- A sub-expression is not reduced unless it is absolutely essential for producing the final result.
- If there is any reduction order that terminates, then **Lazy Evaluation will terminate**.

- **Drawbacks:**

- The **reductions of some sub-expressions** may be unnecessarily repeated.

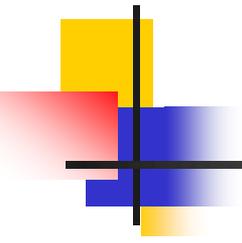


Duplicated Reduction of Subexpressions

- The reduction of the expression $(3+4)$ is duplicated when we attempt to use lazy evaluation to reduce $\text{square } (3+4)$
- This problem arises for any definition where a variable on the left-hand side appears more than once on the right-hand side.

$$\text{square } x = x * x$$

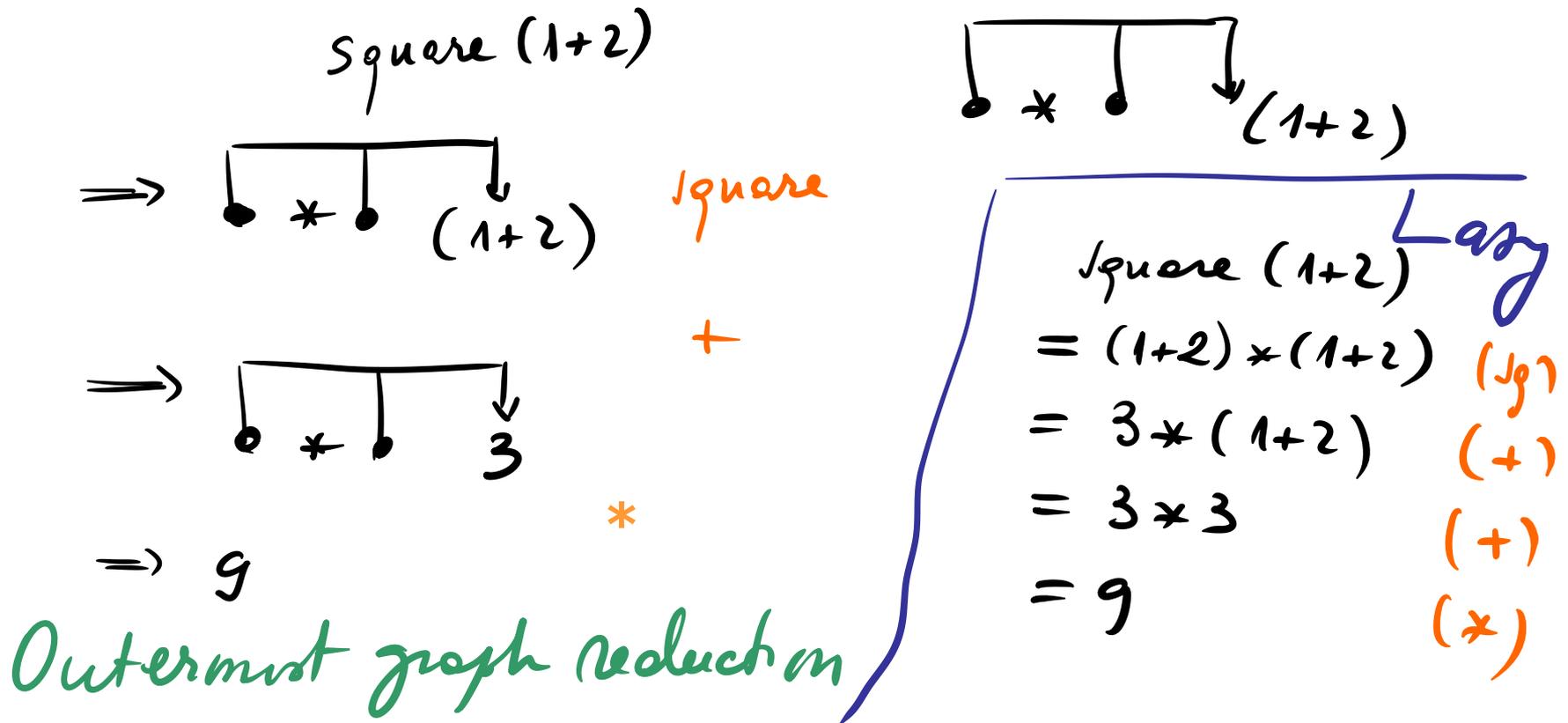
$$\text{cube } x = x * x * x$$



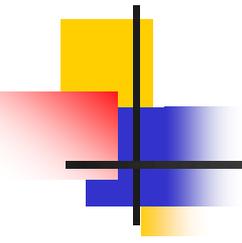
1.5 Graph Reduction

- Aim: **Keep All good features of Lazy Evaluation** and at the same time avoiding duplicated reductions of sub-expressions.
- Method: By representing expressions as graphs so **that all occurrences of a variable (graph nodes) are pointing to the same value (graph node).**

Graph Reduction



Graph Reduction Strategy combines all the benefits of both Eager and Lazy evaluations with none of their drawbacks.



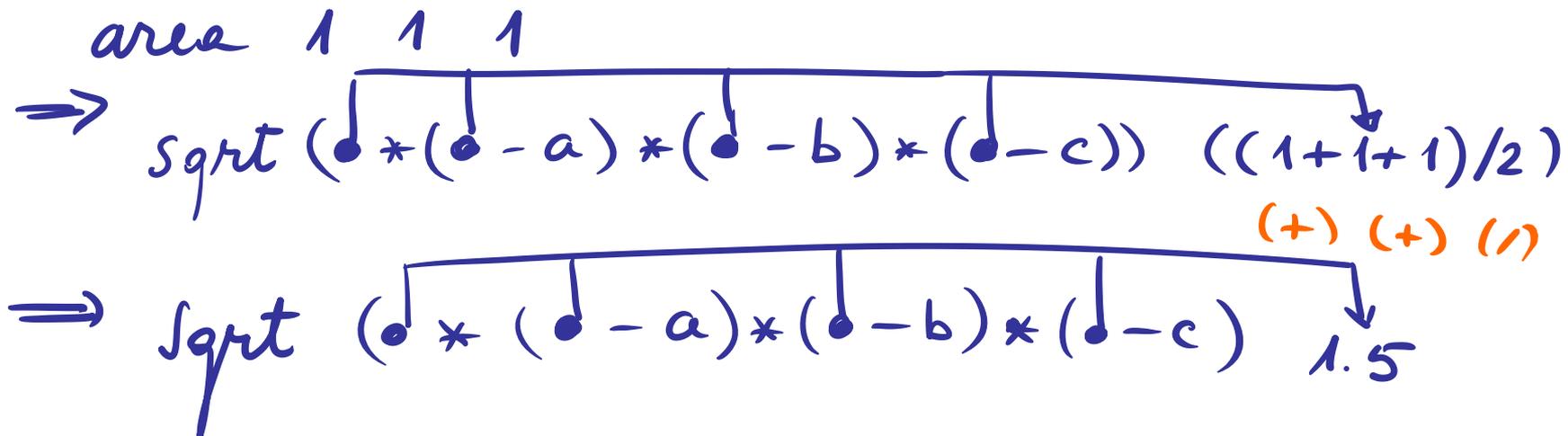
Graph Reduction

- The **outermost graph reduction** of square $(3 + 4)$
now reduces every argument at most once.
- For this reason, it always takes fewer reduction steps than the innermost reduction
- Sharing of expressions is also introduced with **let** and **where** constructs.

Graph Reduction for let

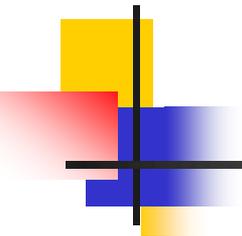
Heron's formula for the area of a triangle with sides a, b and c:

$$\text{area } a \ b \ c = \text{let } s = (a+b+c)/2 \text{ in} \\ \text{sqrt } (s*(s-a)*(s-b)*(s-c))$$



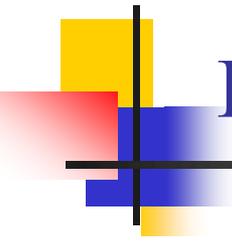
⇒ 0.433 012 702 (√3/4)

- Let-bindings simply give names to nodes in the graph



Graph Reduction

- Any implementation of Haskell is in some form based on **outermost graph reduction** which thus provides a good *model for reasoning about the asymptotic complexity of time and memory allocation*
- The **number of reduction steps** to reach normal form corresponds to the **execution time** and the **size of the terms in the graph** corresponds to the **memory used**.



Reduction of higher order functions and currying

$(.) \quad :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
 $f . g = \lambda x \rightarrow f (g x)$

`id x = x`

`a = id (+1) 41`

`twice f = f . f`

`b = twice (+1) (13*3)`

where both **id** and **twice** are only defined with one argument.

The solution is to **see multiple arguments as subsequent applications to one argument** –
currying (διαδοχικές εφαρμογές σε ένα όρισμα)

Reduction of higher order functions and currying

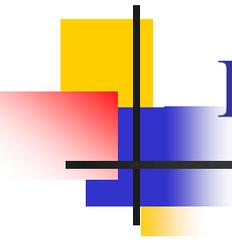
- **Currying**

$$a = (\text{id } (+1)) 41$$

$$b = (\text{twice } (+1)) (13*3)$$

$$\begin{aligned} \text{id } x &= x \\ a &= \text{id } (+1) 41 \\ \text{twice } f &= f . f \\ b &= \text{twice } (+1) (13*3) \end{aligned}$$

- To reduce an arbitrary application *expression1* *expression2*, call-by-need first **reduce *expression1*** until this **becomes a function** whose definition can be unfolded with **the argument *expression2***.



Reduction of higher order functions and currying

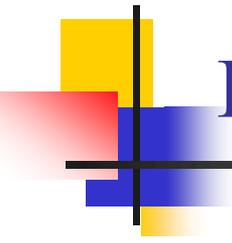
$a = (\text{id } (+1)) 41$

a

$\Rightarrow (\text{id } (+1)) 41$ (a)

$\Rightarrow (+1) 41$ (id)

$\Rightarrow 42$ (+)

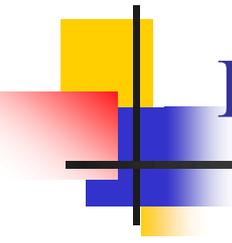


Reduction of higher order functions and currying

`b = (twice (+1)) (13*3)`

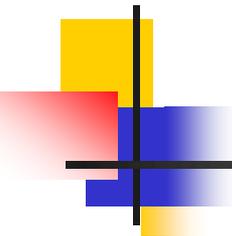
`b`

<code>⇒ (twice (+1)) (13*3)</code>	<code>(b)</code>
<code>⇒ ((+1).(+1)) (13*3)</code>	<code>(twice)</code>
<code>⇒ (+1) ((+1) (13*3))</code>	<code>(.)</code>
<code>⇒ (+1) ((+1) 39)</code>	<code>(*)</code>
<code>⇒ (+1) 40</code>	<code>(+)</code>
<code>⇒ 41</code>	<code>(+)</code>



Reduction of higher order functions and currying

- Functions are useful as **data structures**.
- In fact, all **data structures are represented as functions in the pure lambda calculus**, the root of all functional programming languages.



ΕΠΑΝΑΛΗΨΗ

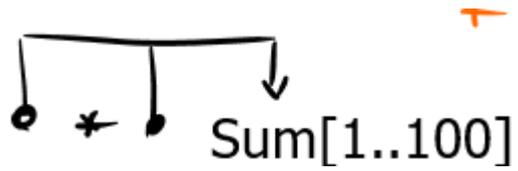
- LAZY: $f a$ (η f εφαρμόζεται πρώτα...)
- EAGER: $f a$ (το a αποτιμάται πρώτα...)
- Outer-Most Graph-Reduction: LAZY + EAGER
- Η f εφαρμόζεται πρώτα: Στην έκφραση των operations που προκύπτει, κάποια operands αντιστοιχούν σε κόμβους γράφου
- Πολλοί κόμβοι δείχνουν στην ίδια έκφραση που αποτιμάται μία φορά και αποθηκεύεται σε shared memory cell. Έτσι αποφεύγονται οι επαναλήψεις της LAZY τεχνικής...

Repeated Reductions of Subexpressions

Square (sum [1:100])

= square 5050 *100 times +*
 = 5050 * 5050 *square*
 = 25502500 ***

Eager: 102



Outer-Most Graph-Reduction: 101

= sum [1..100] * sum [1..100]
 = 5050 * *square* sum [1..100]
100 times +
 = 5050 * 5050
100 times +
 = 25502500 ***

Lazy: 202