COMP0020 Functional Programming

Lecture 17

Graph Reduction

Contents

- Representation of lambda expressions :
  - Abstract
  - Physical
- Performing a beta reduction
- Reduction orders
- Lazy and strict evaluation
- Parallel evaluation

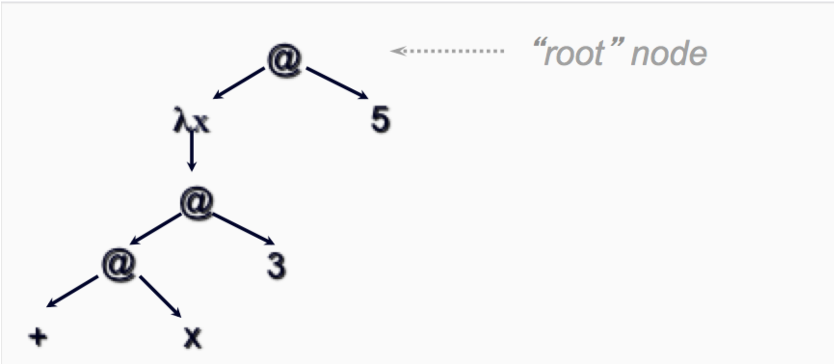
Abstract representation (1)

$f\ x\ =\ x\ +\ 3$

$main\ =\ f\ 5$

$(\lambda\ x\ .\ (x\ +\ 3))\ 5$

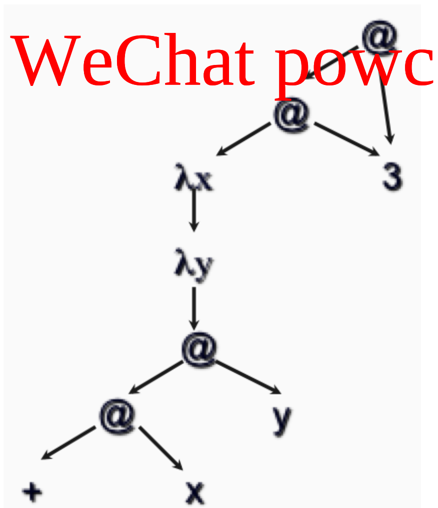
Syntax tree :



Abstract representation (2)

$$z = 3$$
$$f\ x\ y\ =\ x\ +\ y$$
$$main\ =\ f\ z\ z$$
$$(\lambda\ z\ .\ ((\lambda\ x\ .\ (\lambda\ y\ .\ (x\ +\ y)))\ z\ z\ ))\ 3 \qquad (\textit{sharing})$$

Syntax tree after first beta reduction :



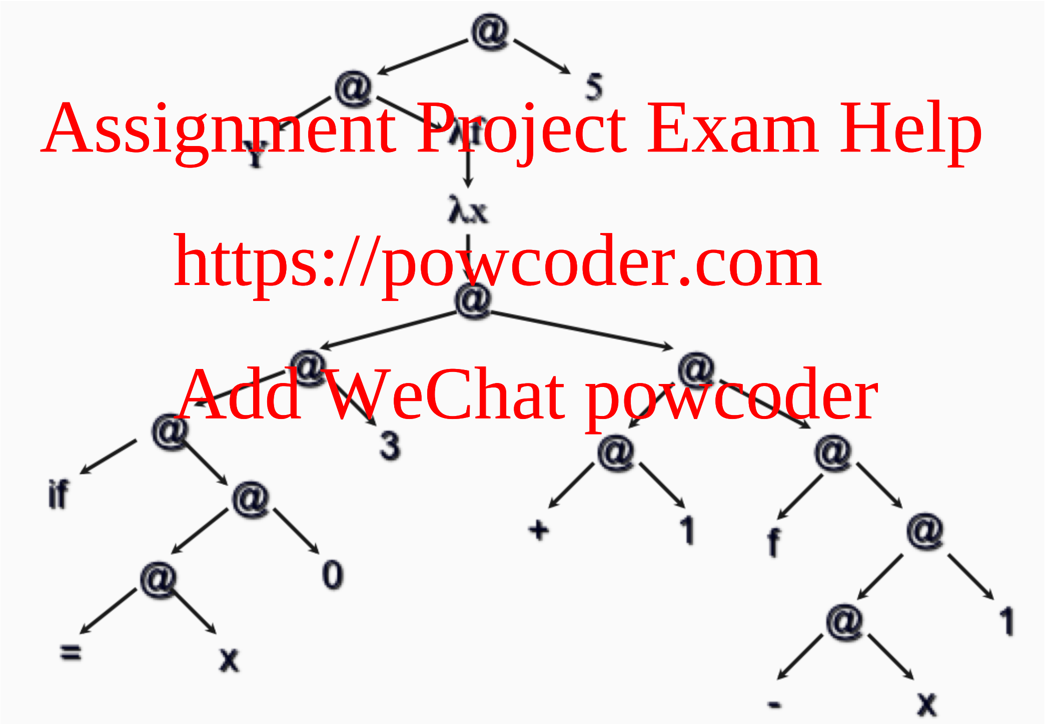
Abstract representation (3)

Now consider a recursive function definition :

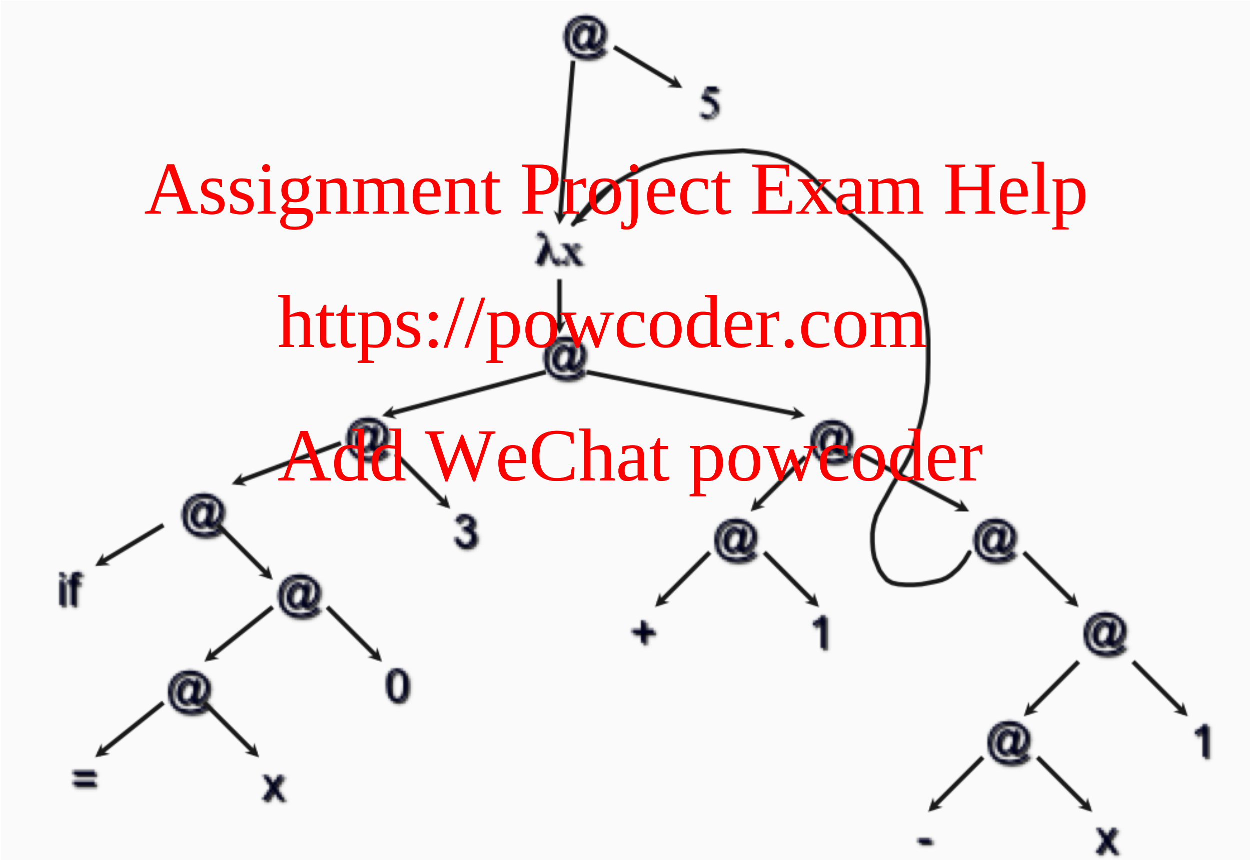
$$
\begin{aligned}
f\ x &= 3,\ \mathit{if}\ (x = 0)\\
&= 1 + f\ (x - 1),\ \mathit{otherwise}\\
\mathit{main} &= f\ 5
\end{aligned}
$$

$$(Y\ (\lambda\ f\ .\ (\lambda\ x\ .\ (\mathit{if}\ (x = 0)\ 3\ (1 + (f\ (x - 1))))))\ )\ 5$$

(Y (λ f . (λ x . (if (x=0) 3 (1+(f (x-1))))))) 5



Alternative abstract representation



Physical representation

- Each node (and leaf) of the graph is held in memory
- Position in memory is irrelevant

  - Just because two nodes occupy adjacent memory locations, that doesn't mean they are related
  - Related nodes are connected using pointers (addresses)

- Set aside a chunk of memory just to hold these nodes
  - The “HEAP"

Physical representation (2)

- Each node requires sufficient memory to store :
  - A tag (@, *I*, "*op*", *IND*, ...)
  - A pointer to a left subtree (alternatively, a number representing a built-in operator or a variable)
  - A pointer to a right subtree (alternatively, a variable or a constant such as a number)
- We call each such small chunk of memory a "cell"
- The heap therefore consists of many cells

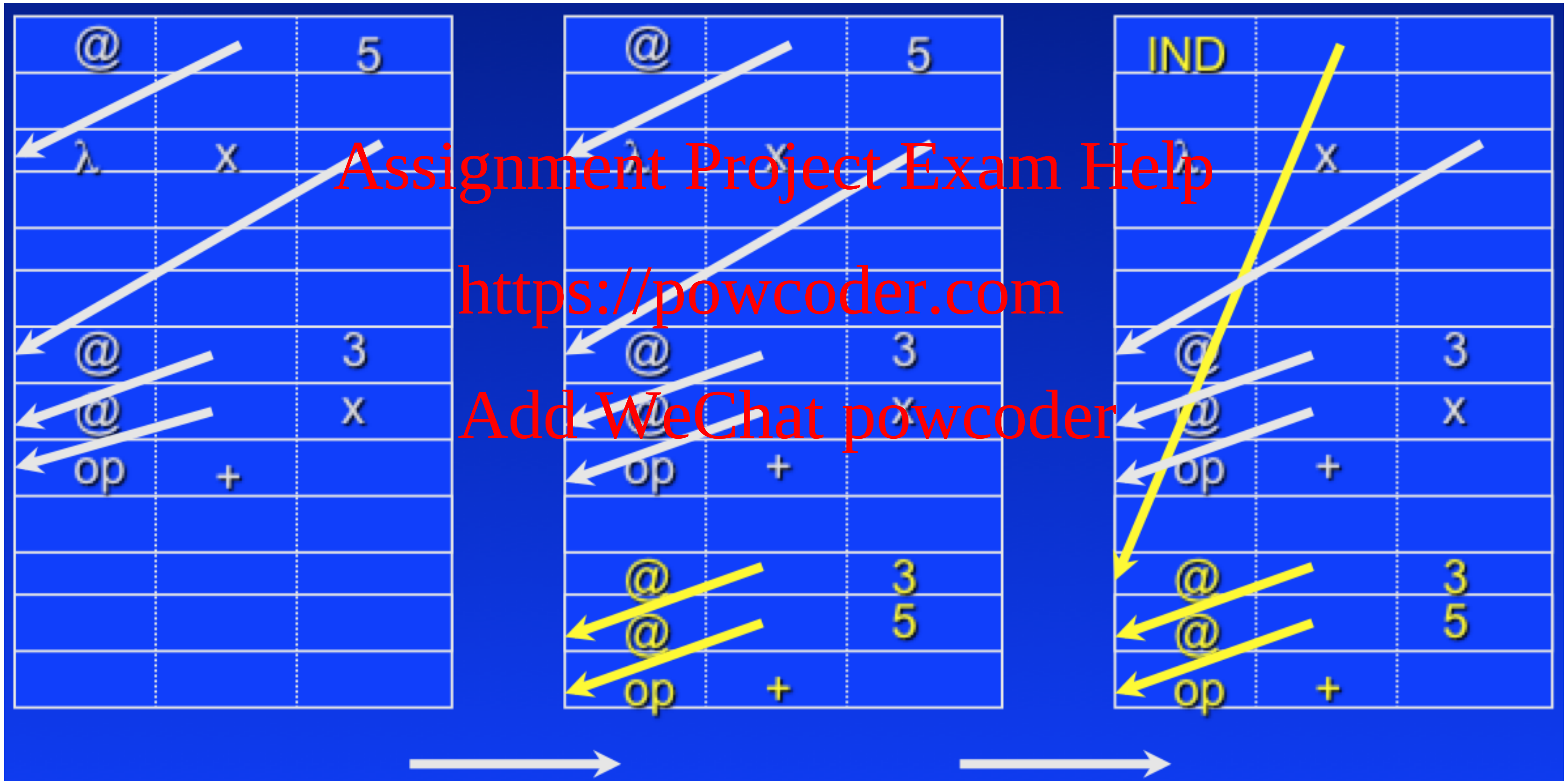
Physical representation (3)

- The heap :

Performing a beta reduction

- Make a COPY of the function body
- Substitute the actual parameters for the formal parameters
- OVERWRITE the root node of the expression with an <u>indirection</u> to the root node of the copy (*preserves sharing*)

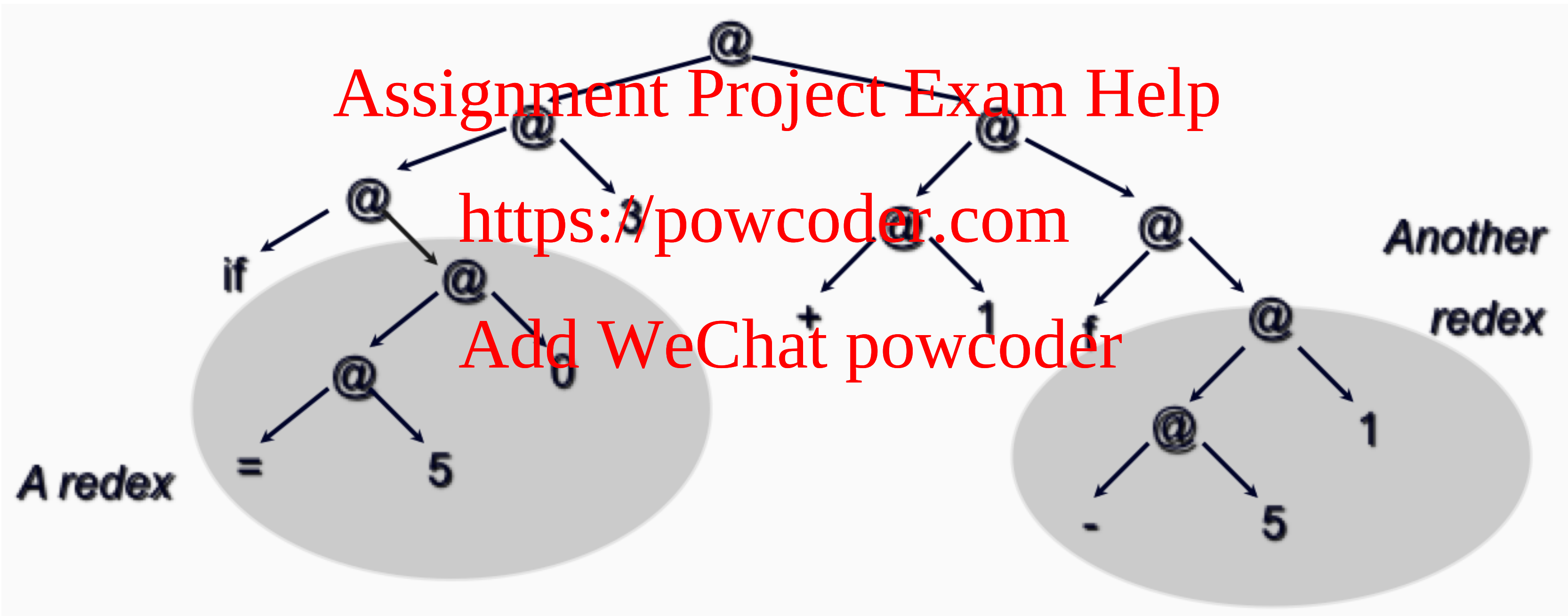
Beta reduction (2)



Beta reduction (2)

- A "redex" is a reducible expression, i.e. an expression that can be reduced by α, β, η or δ-rule reduction
- A program may contain many redexes
- Evaluation of a program is a process of successive transformations (reductions) of the graph until the final value is found :
  1. Decide which redex to reduce next
  2. Reduce it
  3. Loop to 1.

Redexes



Reduction orders

- There will often be many redexes that could be evaluated next
- The evaluator must choose an EVALUATION ORDER : this may have a great effect on the performance of the system (but not on its (partial) correctness)

Reduction orders

- NORMAL ORDER EVALUATION
  - starts at the top node and follows the left pointers until the first redex is found
  - leftmost outermost
  - lazy evaluation.
- APPLICATIVE ORDER EVALUATION
  - leftmost innermost redex
  - strict evaluation
- PARALLEL EVALUATION
  - referential transparency
  - redexes can be reduced in any order
  - may be reduced concurrently !

Summary

- Physical representation of lambda expressions
- Performing a beta reduction
- Reduction orders
- Lazy and strict evaluation
- Parallel evaluation

END OF LECTURE