Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 20

GRAPH REDUCTION

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

CONTENTS

This lecture explores the technique of Graph Reduction for evaluating functional programs compiled into an intermediate representation – either those compiled into the λ calculus or those compiled into combinators

We focus on *interpretive* graph reduction of λ calculus expressions: covering both abstract and physical representations, how beta reduction is performed, and finding the next redex. The lecture ends with some observations about interpretive versus compiled graph reduction

Students are expected to read Chapters 10, 11 and 12 of "The Implementation of Functional Programming Languages", by Simon Peyton Jones, available for free download at this link: https://www.microsoft.com/en-us/research/uploads/prod/1987/01/slpj-book-1987.pdf

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

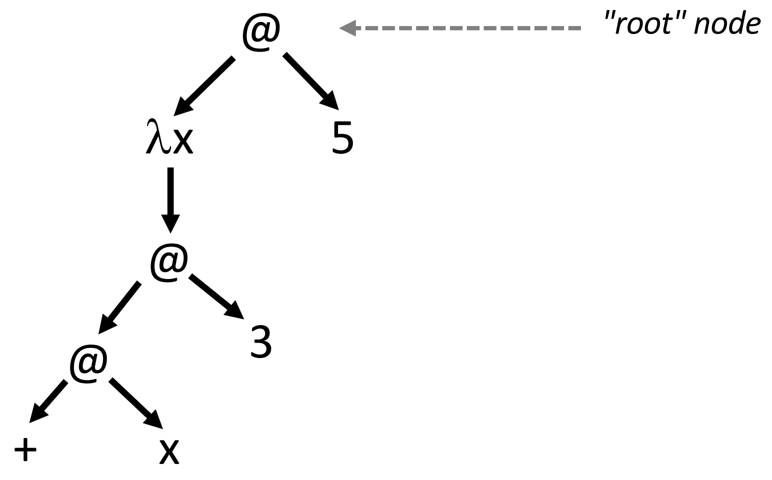
ABSTRACT REPRESENTATION

f x = x + 3

main = f 5

(λx.(x+3)) 5

Abstract representation:



We assume that a functional program has been compiled into an intermediate form that uses the (extended) λ calculus

For graph reduction, this intermediate form will be held as a graph structure (a tree data structure, possibly with shared nodes and cyclic references). We explain this representation in two stages – first we present the "abstract representation" and then we will present the "physical representation" (which will build on some of the material in the previous lecture on Lists, Trees and Graphs)

Here we present one possible "abstract representation" of the graph structure, using binary nodes (each graph node has at most two subgraphs). The most important aspect of this representation is the application node indicated by the character "@" which represents the application of its left subgraph (a function) to its right subgraph (an argument). Currying is used for multiple arguments

The term λx.e is represented by a λ node indicated by the character "λ" and the name of the formal parameter (e.g. "x") with just one subgraph which is the function body

This slide shows simple Miranda code, its λ calculus expression (in black), and its abstract representation as a graph structure. Notice that the top application node of a redex is called the "root" node of that redex

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

ABSTRACT REPRESENTATION

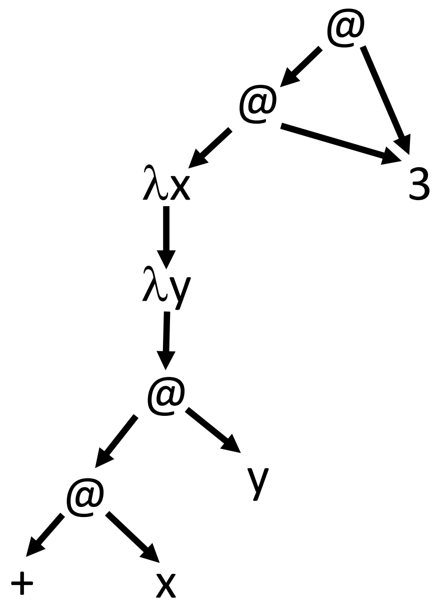
z = 3

f x y = x + y

main = f z z

(λz.((λx. (λy.(x+y))) z z)) 3

Abstract representation after one β reduction:



Here is a second example that illustrates how shared references to graph nodes can occur

In this example the function f is applied to the value z twice. The associated λ expression is shown in black

The graph representation shown below illustrates the result of sharing that occurs during a graph reduction implementation of β reduction – it shows the state of the graph representation **after** one β reduction has occurred

In a later slide we will explain exactly how β reduction occurs during interpretive graph reduction of λ expressions

This example also serves to highlight a feature of interpretive graph reduction, which is that during a sequence of evaluation steps it makes successive changes to the graph

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

Here is a third example that illustrates how cyclic references to graph nodes can occur

In this example the function f is recursive. The associated λ expression is shown in black with Y as a built-in operator (refer back to the previous lecture to recall how to represent recursive functions in the λ calculus)

In the next two slides we will first give the abstract graph representation without cyclic references (with Y as a built-in operator) and then give the abstract graph representation with cyclic references (and without Y)

ABSTRACT REPRESENTATION

```
f x  = 3, if (x=0)
     = 1 + f (x-1), otherwise
main = f 5
```

Associated λ expression with Y as a built-in operator:

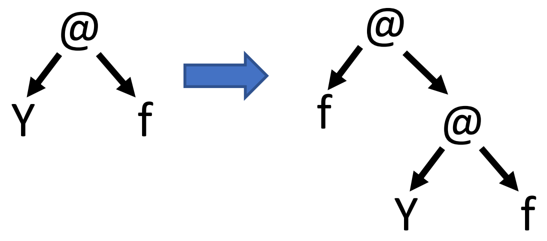
(Y (λf.(λx. (if (x=0) 3 (1+ f(x-1)))))) 5

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

This example illustrates an abstract graph representation of the λ calculus expression without using cyclic references

It uses the built-in operator Y for brevity, though using the λ calculus expression that defines Y would similarly give a non-cyclic graph representation

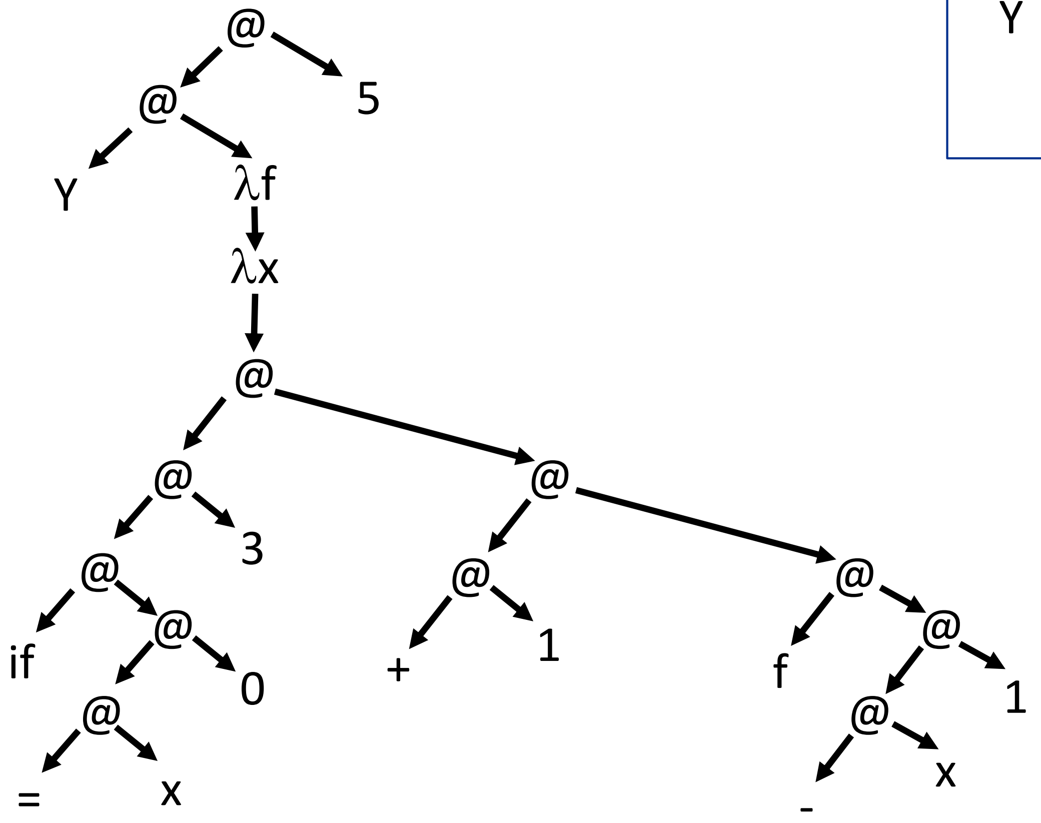
The non-cyclic implementation of the Y operator has the following reduction rule for Y:



ABSTRACT REPRESENTATION

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

Abstract representation without cyclic references:

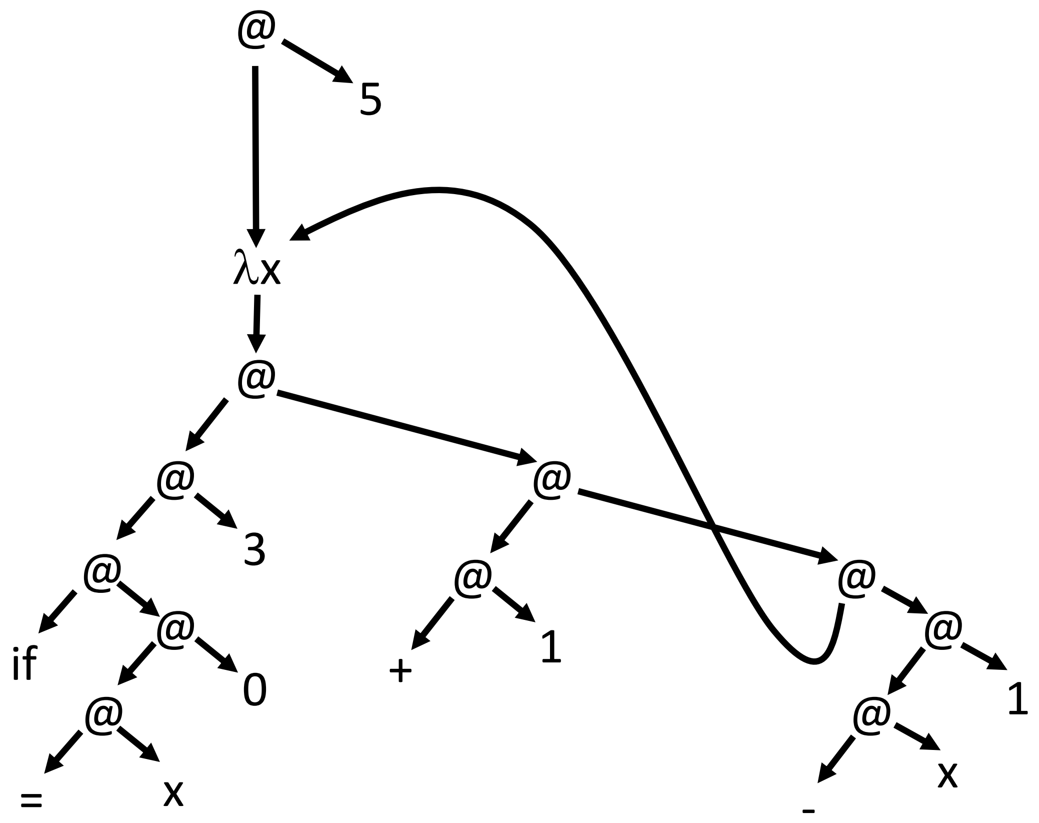


FUNCTIONAL PROGRAMMING GRAPH REDUCTION

ABSTRACT REPRESENTATION

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

Abstract representation **with** cyclic references:

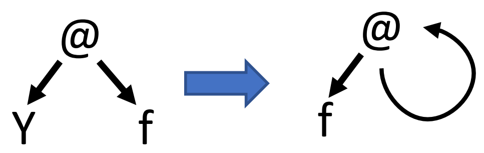


This example illustrates an alternative abstract graph representation of the λ calculus expression **with** cyclic references

With a graph structure it is possible to represent a recursive function without using either the Y operator or its λ calculus definition – all that is necessary is to indicate that the left subgraph of an application node (the root node of the recursive call of f to (x-1)) is the graph representing the function f, which already exists

Notice that because there is no Y operator in this example there is also no need for the λf construction

Some implementations incorporate Y but with a reduction rule that creates a cycle when it is evaluated. For example:



A cyclic representation relies on an understanding of how β reduction works with graph reduction, which will be explained in a later slide

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

PHYSICAL REPRESENTATION

- Simple scheme: each node or leaf is held in a small fixed-size area of contiguous memory called a "cell"
- A cell comprises a tag, a left field and a right field
- Cells are ONLY related by saving the memory address of a cell inside the left or right field of another cell ("pointers")
- If a memory address requires 4 bytes then the left and right fields must be at least 4 bytes
- The tag may be smaller (e.g. 1 byte)

We consider a very simple physical (sometimes called "concrete") graph representation of λ calculus expressions, where each node or leaf of the graph is held in a small fixed-size area of contiguous memory called a "cell"

We have seen this simple scheme of fixed-size memory cells in a previous lecture. A cell comprises three components:

- a tag
- a left field (also known as "Field 1")
- a right field (also known as "Field 2")

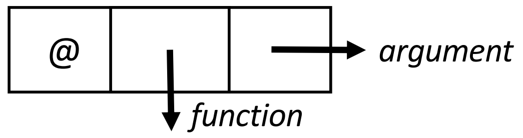
The location of cells in memory has no special meaning – they are related only by the use of the memory address of one cell being held in either the left or right field of another cell. We say that such an address is a "pointer" and that it "points" to the other cell

In a computer architecture that uses 4 bytes for a memory address, both the left and right fields must be at least 4 bytes. The tag can be smaller (e.g. 1 byte) but sometimes having a larger tag may improve memory access speed (due to cache alignment issues)

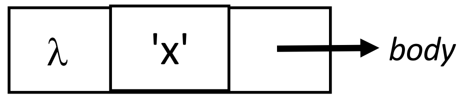
FUNCTIONAL PROGRAMMING
GRAPH REDUCTION

PHYSICAL REPRESENTATION

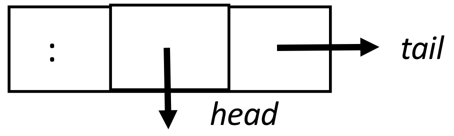
- An application cell:



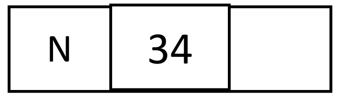
- A λ abstraction:



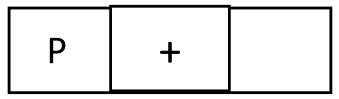
- A CONS cell (list cell):



- A Number:



- A built-in function/operator:



Here are some example cells with their tags:

- An application cell
- A λ abstraction
- A CONS cell
- A Number
- A Built-in function (also known as a "primitive" function)

In each case the tag is just a bit-pattern. For example, if the tag field is 1 byte then we might choose bit patterns as follows:

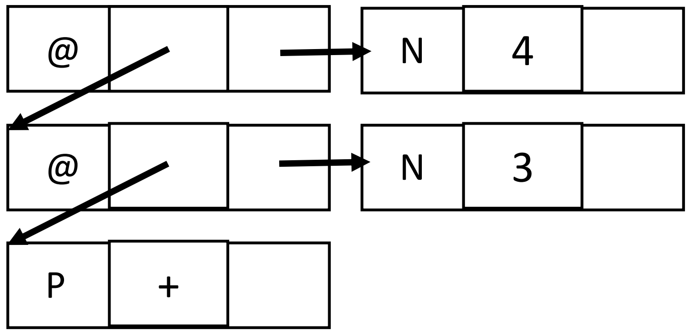
- @ to be 01000000
- λ to be 01101100
- : to be 00111010
- N to be 01001110
- P to be 01010000

Of course the values in the left and right fields are also just bit patterns – representing either an address or a number or a character or a built-in function. In each case the runtime system knows what to expect from the value of the tag

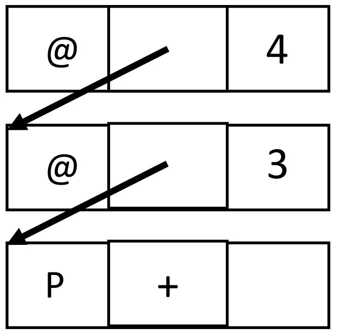
FUNCTIONAL PROGRAMMING GRAPH REDUCTION

PHYSICAL REPRESENTATION

- Using boxed numbers:



- Using unboxed numbers:



Some data items are small enough that they are capable of being represented in a single field

For example a number might be capable of representation in X bytes and a field might be Y bytes – if X < Y then any number could be held in a field and need not use an entire cell for its representation

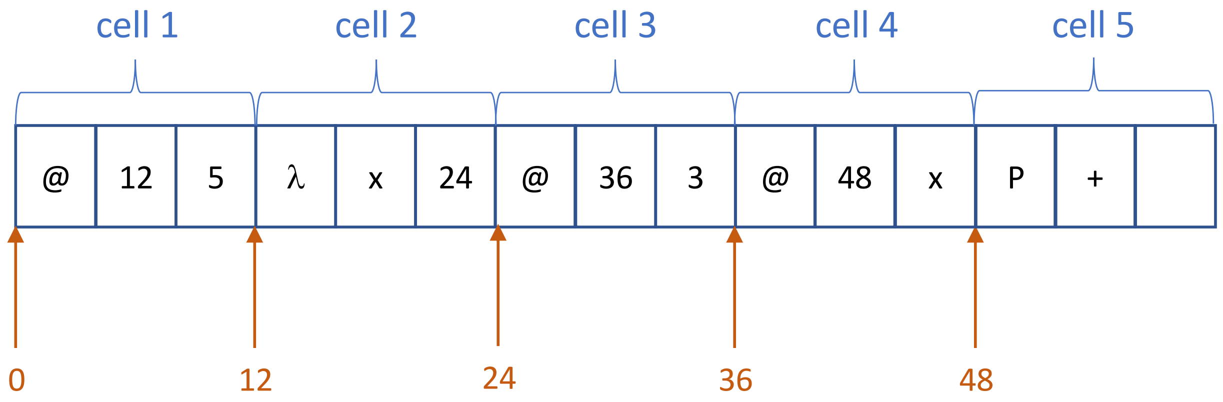
The diagrams on the left illustrate the expression (3 + 4) using firstly a "boxed" representation, where each number is held in a cell, and secondly an "unboxed" representation where each number is held (more efficiently) in a field

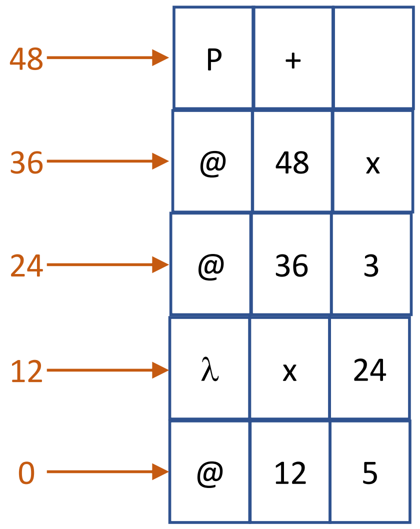
In a program some numbers might be boxed and some unboxed. A disadvantage of the unboxed representation is that it is not possible to share a pointer to a field (we only have pointers to cells). We will see later how (to preserve laziness) a β reduction typically overwrites the root node of the redex with the resulting value (or an indirection to a resulting expression) – if this is a single value it will naturally be boxed

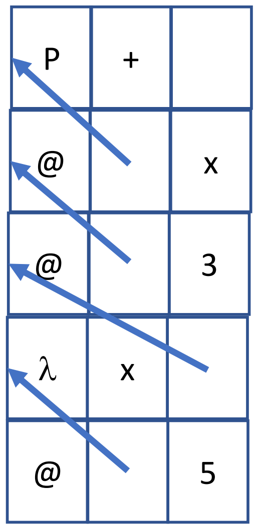
It is not only numbers that can be unboxed – it can occur for any data item that is sufficiently small

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

PHYSICAL REPRESENTATION







All of the cells that represent the program are held in an area of memory called the "heap"

Graph reduction successively transforms the contents of the heap (the cells), creating new cells and modifying the contents of existing cells according to the implementation mechanism for each of the normal λ calculus reduction rules. This continues until normal form (or more likely weak head normal form) is reached

This slide gives an example of the physical representation in the heap of a program that applies (λx.(x+3)) to the argument 5. It assumes that tags are the same size as fields (4 bytes each), and that addresses start at 0 and a separate address is available for each byte. The memory addresses for the start of the five cells are therefore 0, 12, 24, 36 and 48

Notice how a field might contain either a number or a variable name of type num or an address – the *types* are always known at runtime, but runtime representations may differ and the runtime system must distinguish these cases, either by convention or with additional information (perhaps using spare bits in the tag field)

The second and third diagrams are alternative views. They all represent identical layouts in the heap

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

PERFORMING A β REDUCTION

1. COPY the function body

2. Substitute each free occurrence of the formal parameter with a POINTER to the actual argument

3. OVERWRITE the root node of the redex with an indirection to the root node of the copy

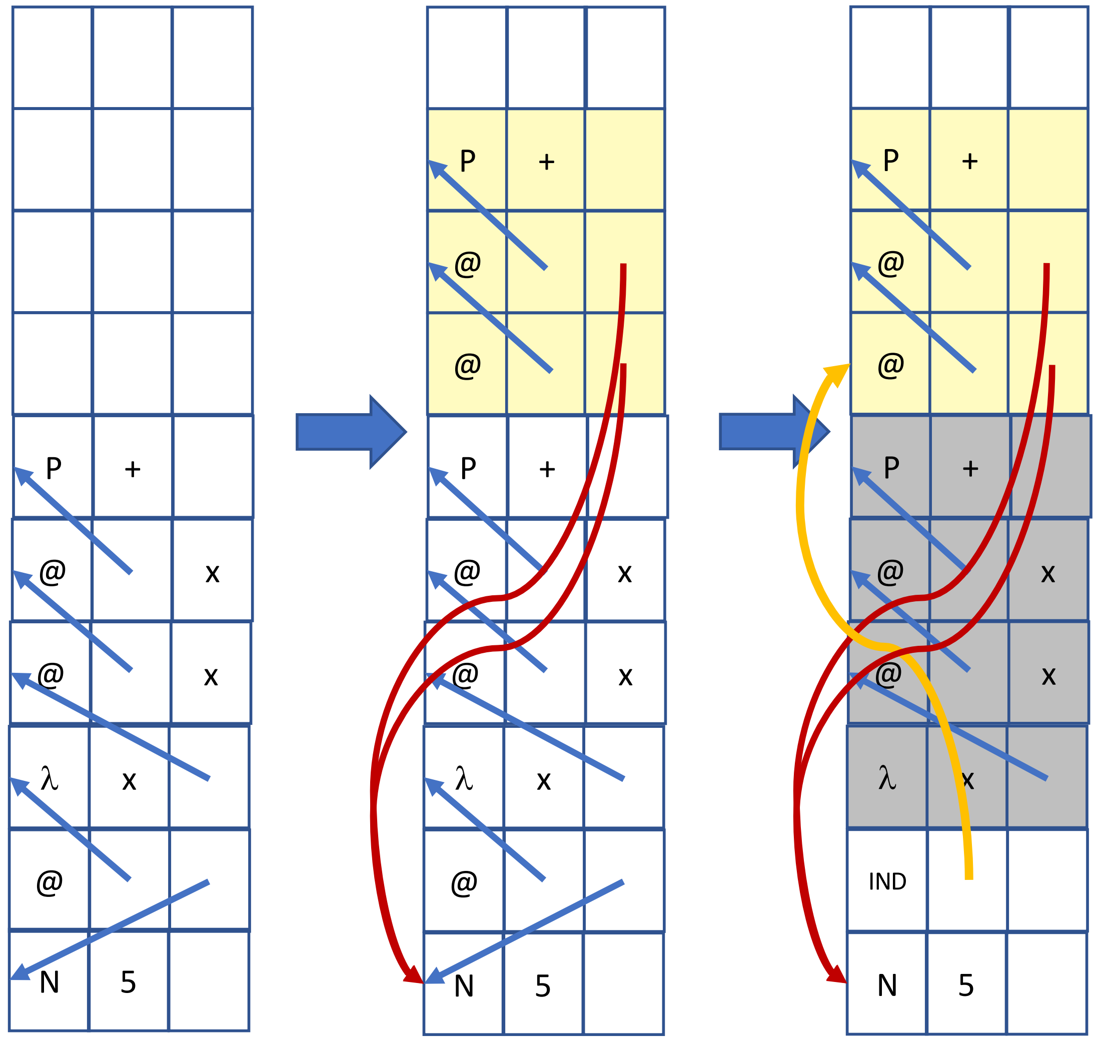
Graph reduction performs a β reduction using the following steps:

1. Make a COPY of the function body – this is important because the lambda abstraction might be shared
2. Substitute a POINTER to the actual argument for each free occurrence of the formal parameter name in the function body (unless the actual argument is unboxed, in which case just copy the argument value) – this is important because the argument might be bulky, or contain redexes whose evaluation we only want to perform at most once
3. OVERWRITE the root node of the redex with an INDIRECTION to the root node of the copy

An indirection node has tag "IND" – the root node of the redex is overwritten (in-place update) because the redex might be shared, and an indirection is used because the result might be a bulky data structure or contain redexes that need further evaluation

FUNCTIONAL PROGRAMMING
GRAPH REDUCTION

PERFORMING A β REDUCTION



Here's a simple example of a β reduction of (λx.(x+x)) 5 where the number 5 is boxed

The cells coloured yellow are the copy that has been created of the function body, with each formal parameter name "x" replaced by a POINTER to the boxed argument cell (these two pointers are shown in red)

The second step is to overwrite the root node of the redex with an indirection (using tag IND) to the root node of the copy of the function body. Notice how this causes the previous contents of the root node of the redex (the @ tag, and two pointers) to be deleted and a new pointer (coloured orange) to be created

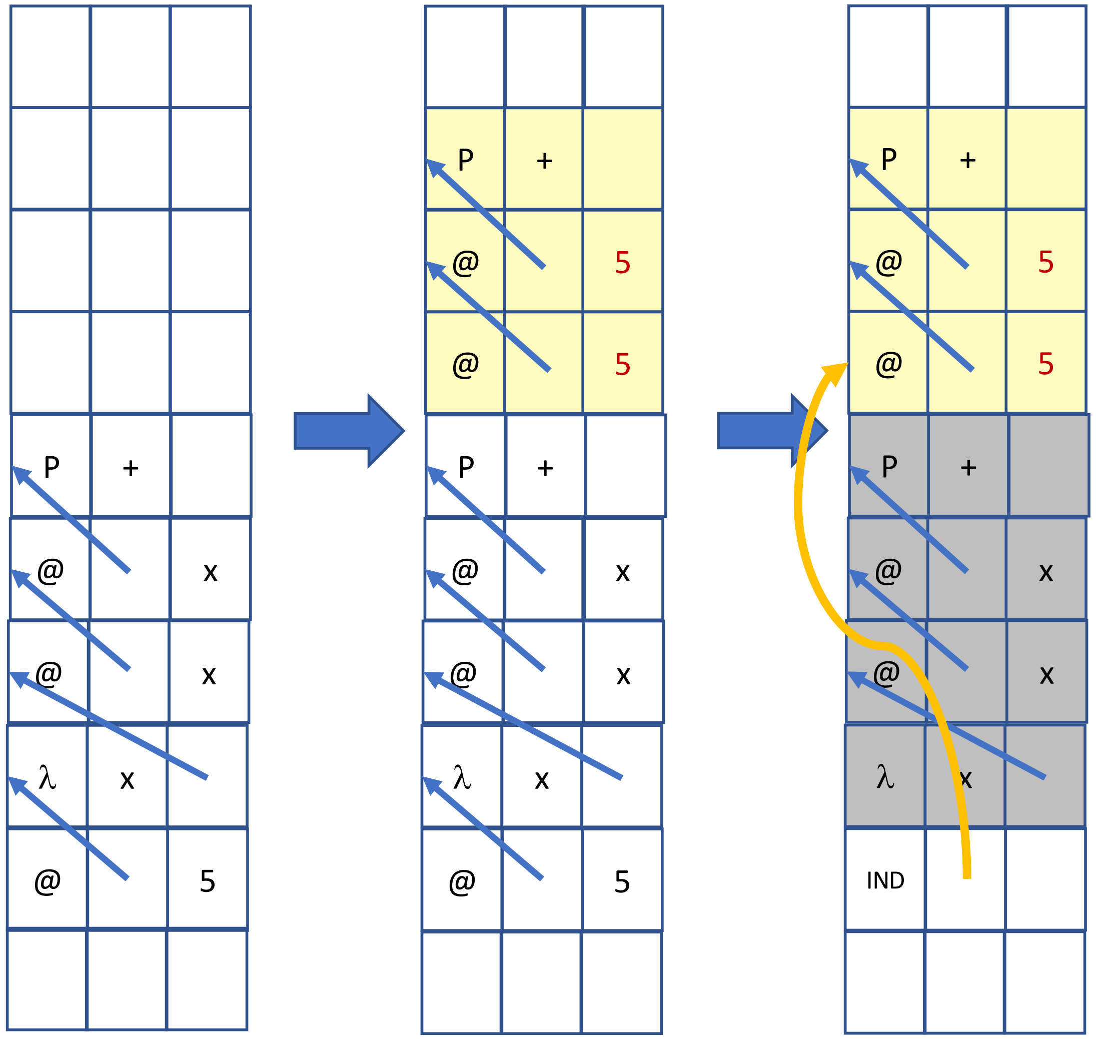
Notice that the cells now coloured grey cannot be reached by following pointers from the indirection cell – if there is no other way to reach them from the rest of the program (which in this case is true) then they are "garbage" cells that can be re-used. We will discuss this in a later lecture on garbage collection

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

This example is almost identical to the previous example, except that in this case the argument value is unboxed

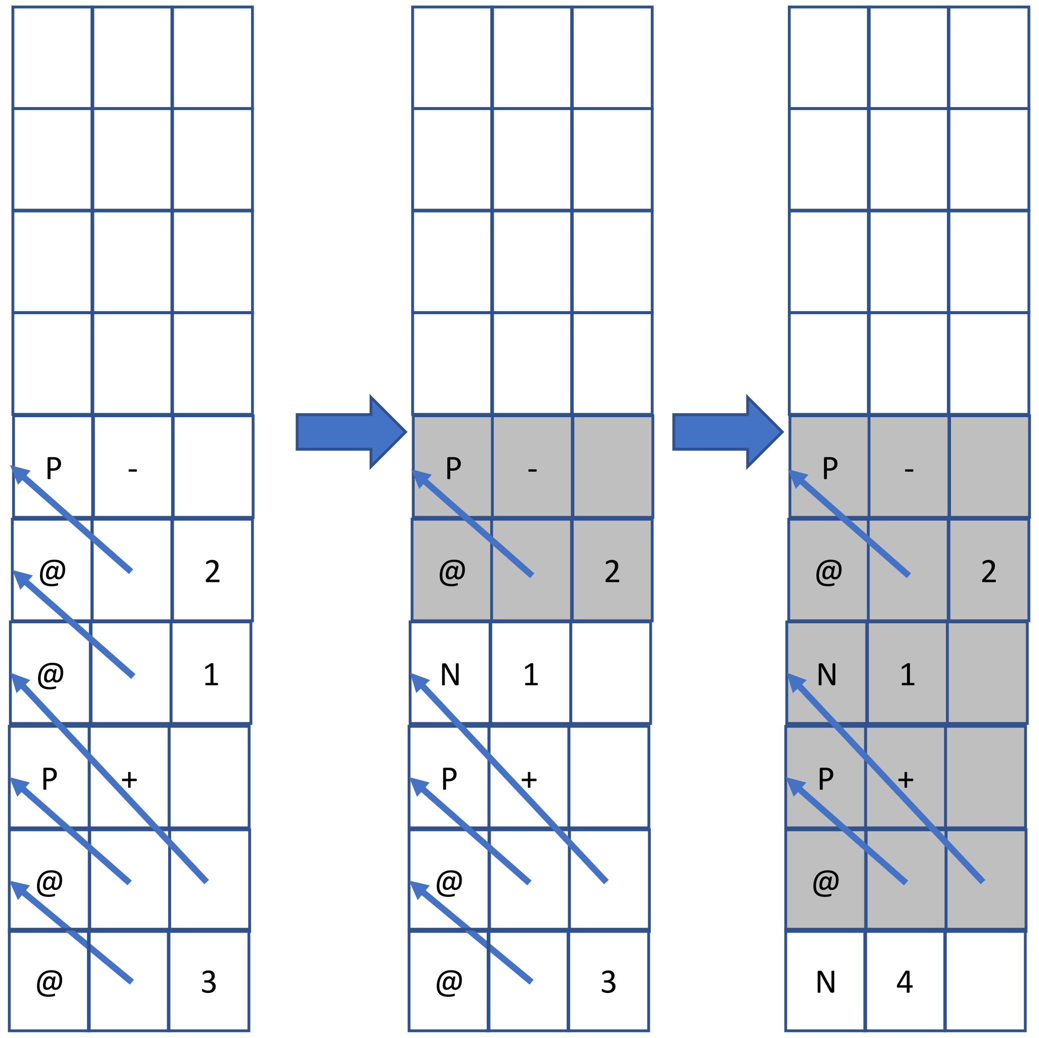
For unboxed arguments we simply use copies of the argument instead of using a pointer to the argument cell

PERFORMING A β REDUCTION



FUNCTIONAL PROGRAMMING
GRAPH REDUCTION

PERFORMING A δ REDUCTION



Performing a δ reduction is different to performing a β reduction in one important aspect:

Built-in operators may be **strict** in one or more of their arguments, which means that the operator cannot produce a result until that argument (or those arguments) have been evaluated

E.g. the operator "+" requires both of its arguments to be fully evaluated (to numbers) before it can add those numbers together to generate a result

The built-in operator "if" is only strict in its first argument, which must be evaluated to a Boolean before "if" can determine whether to return a pointer to its second argument or to its third argument

This is very different to the behaviour of lazy-evaluation β reduction which does not attempt to evaluate the argument of a function in advance

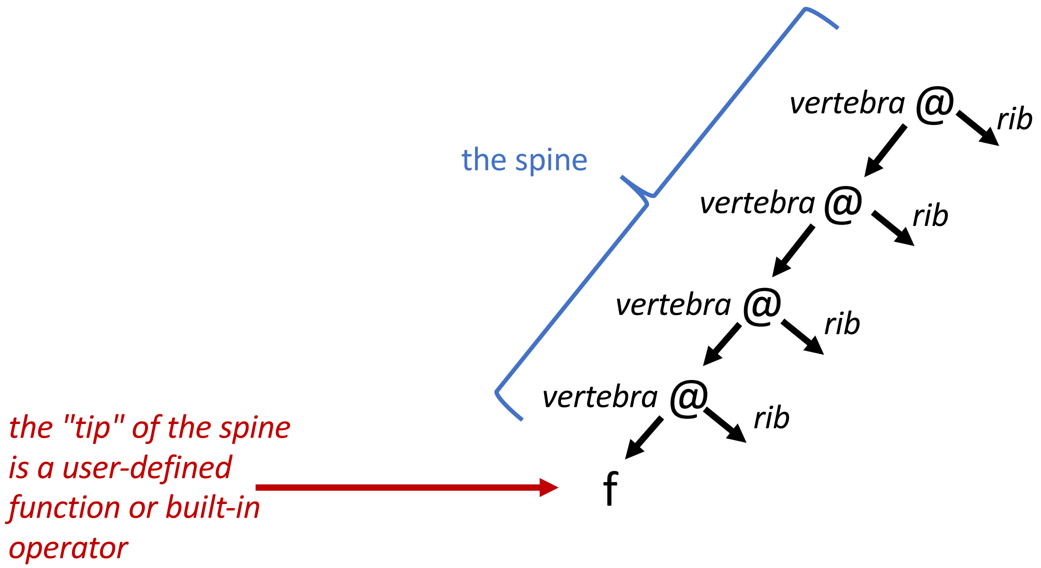
This example shows the evaluation of the expression ((2-1) + 3). The δ reduction can't proceed until (2-1) is evaluated, so the interpreter calls itself recursively to evaluate (2-1) and overwrite the root of the sub-redex with its value – it then also checks that the second argument is evaluated (it is!) and then overwrites the root of the main redex with the result (4)

Notice how in both cases the δ reduction results in an unboxed result with no indirection

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

FINDING THE NEXT REDEX

- Find the next redex – reduce it – loop
- Start at the root node of the program (or the root node of a subgraph to be evaluated)
- Reduce to WHNF
- Structural terminology:


the spine
vertebra
@
rib
vertebra
@
rib
vertebra
@
rib
vertebra
@
rib
f
the "tip" of the spine is a user-defined function or built-in operator

Graph reduction is a repetitive process that acts as follows:

1. find the next redex
2. reduce it
3. loop to 1

Finding the next redex is the key to whether the implementation is strict or lazy (e.g. whether Applicative Order or Normal Order evaluation is performed)

Graph reduction typically starts at the root node of the whole program (or the root node of a subgraph to be evaluated) and inspects the structure

Most modern implementations reduce programs to Weak Head Normal Form. As a reminder, a term M is in Weak Head Normal Form (WHNF) if it is either (i) $M \equiv \lambda x_1 \ldots x_n . (x\ N_1 \ldots N_m)$ *where* $n,m \geq 0$ *and* x *is a variable or (ii)* $M \equiv \lambda x.N$

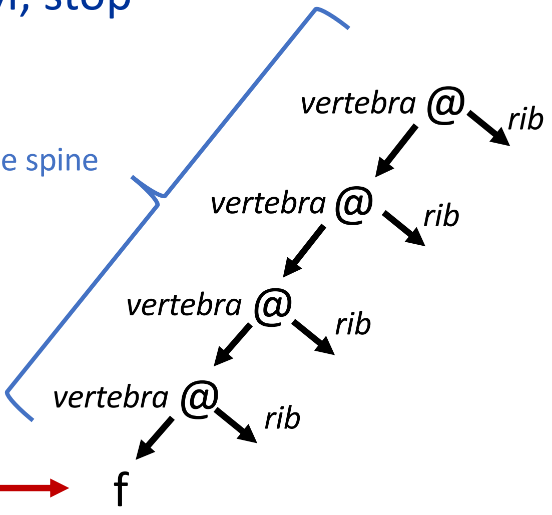
Thus, if the program (or subgraph to be reduced) is just a variable name, the graph reduction stops; similarly, if it is just a λ abstraction (not applied to an argument) then the process stops. Graph reduction only acts if it sees either a function or a built-in operator applied to arguments

The diagram illustrates some terminology for a graph representation: the *spine* representing a function(at the *tip*) applied to arguments, the *vertebrae* (each an application node) and the *ribs* (pointers to the arguments)

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

FINDING THE NEXT REDEX

- start at the root
- descend (unwind) the spine until the outermost leftmost function or operator is found
- If f is a user-defined function and there is at least one argument, perform a β reduction (if no arguments, stop)
- Otherwise, determine how many arguments f needs (N) and how many exist (M). If N>M, stop
- Otherwise determine which arguments are strict, recursively evaluate each strict rib, then perform a δ reduction
- Loop



the "tip" of the spine is a user-defined function or built-in operator

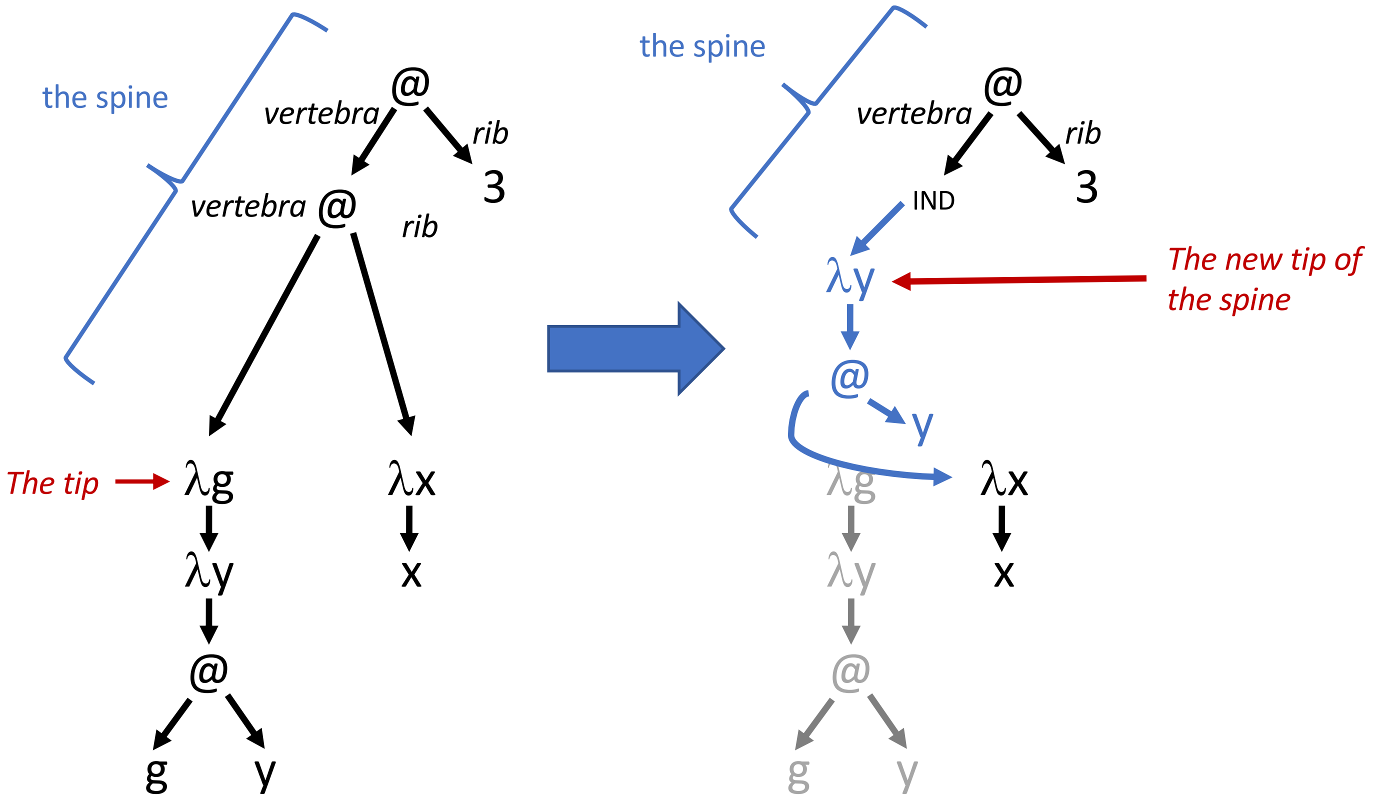
To determine the next redex, graph reduction starts at the root node and follows the pointers in the left field of each vertebra, descending (often called "unwinding") the spine until it finds either a user-defined function or a built-in operator at the tip

- If f is a user-defined function (i.e. a λ abstraction) and if there is at least one argument on the spine, then perform a β reduction. If there are no arguments, stop.
- Otherwise f is a built-in operator, so determine the number of arguments it needs (N) and the number of arguments available (M). If N>M, stop.
- Otherwise find out which of its arguments are strict (i.e. must be evaluated first) – call the graph reducer recursively on the root node of the subgraph for each such strict rib, then perform a δ reduction
- Then loop: repeat the whole process, starting again with the root node of the program

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

EXAMPLE

((λg.λy.(g y)) (λx.x)) 3



Here is an example where the function at the tip of the spine is a user-defined function

The graph reduction interpreter starts at the top node of the program and descends until it finds the tip

The tip is a λ abstraction with more than one argument existing on the spine, so perform a β reduction: copy the function body (the new copy is shown in blue, the original body – sometimes called the "template" is shown in grey) and replace each occurrence in that copy of the formal parameter with a pointer to the actual argument

The root of that β reduction is the vertebra for the first rib of the original spine (which applies the function to the first argument on the spine). This vertebra is therefore overwritten with an indirection node which points to the root node of the new copy of the function body

The process starts again – the graph reduction interpreter descends the spine from the root node of the program, and discovers a new tip

This process continues until the program graph is in weak head normal form

FUNCTIONAL PROGRAMMING GRAPH REDUCTION

INTERPRETIVE VERSUS COMPILED REDUCTION

- Interpretive β reduction using template copying is inefficient:
  - case analysis on tags
  - variable testing to see if it is the formal parameter
  - copying subgraphs that won't change
- Alternatively a compiler could produce native code
- Difficulty with free variables
- Can solve with an "environment" of bindings for free variables
- In the next lecture we'll see a better solution

We have seen how interpretive β reduction of λ expressions can be achieved using a "template copying" approach – this is inefficient for many reasons:

- at each node of the function body, the interpreter must do case analysis on the tag of the node
- each variable name must be tested to see whether it is the formal parameter name (and, if so, updated)
- subgraphs with no free occurrences of the formal parameter are copied, which is not strictly necessary

An alternative would be for the compiler to generate native code for each function body – that code would then apply functions to arguments and build graph as necessary

Unfortunately this is not easy where there are free variables in the function body. Consider the function λx.E where E=λy.(y-x): λx. (λy. (y-x))

A compiler would need to generate code for different versions of E depending on the value of x. For example, if x=3 the code must build graph for λy.(y-3).

It is possible to create code that produces a result with an indirection to a value held in an "environment" of bindings for free variables, though this also has overheads, and we will see a better solution in the next lecture

FUNCTIONAL PROGRAMMING
GRAPH REDUCTION

SUMMARY

- Representation of lambda expressions :
    - Abstract
    - Physical
- Performing a β reduction
- Performing a δ reduction
- Finding the next redex
- Interpretive versus compiled reduction

In summary, this lecture has explored the technique of Graph Reduction for evaluating compiled functional programs – either those compiled into the λ calculus or those compiled into combinators

The lecture focused on *interpretive* graph reduction of λ calculus expressions: covering both abstract and physical representations, how β reduction is performed, and finding the next redex. The lecture ended with some observations about interpretive versus compiled reduction

Students are reminded to read Chapters 10, 11 and 12 of "The Implementation of Functional Programming Languages", by Simon Peyton Jones