Imperative vs. Declarative (1)

- **Imperative Languages**:  
  - Implicit state.  
  - Computation essentially a sequence of side-effecting actions.  
  - Examples: Procedural and OO languages

- **Declarative Languages** (Lloyd 1994):  
  - No implicit state.  
  - A program can be regarded as a theory.  
  - Computation can be seen as deduction from this theory.  
  - Examples: Logic and Functional Languages.

Imperative vs. Declarative (2)

Another perspective:

- **Algorithm = Logic + Control**
  - Declarative programming emphasises the logic (“what”) rather than the control (“how”).
  - Strategy needed for providing the “how”:  
    - Resolution (logic programming languages)  
    - Lazy evaluation (some functional and logic programming languages)  
    - (Lazy) narrowing: (functional logic programming languages)

No Control?

Declarative languages for practical use tend to be only weakly declarative; i.e., not totally free of control aspects. For example:

- Equations in functional languages are directed.
- Order of patterns often matters for pattern matching.
- Constructs for taking control over the order of evaluation. (E.g. cut in Prolog, seq in Haskell.)
Relinquishing Control

Theme of this lecture: **relinquishing control by exploiting lazy evaluation.**

- Evaluation orders
- Strict vs. Non-strict semantics
- Lazy evaluation
- Applications of lazy evaluation:
  - Programming with infinite structures
  - Circular programming
  - Dynamic programming
  - Attribute grammars

Evaluation Orders (1)

Consider:

\[
\begin{align*}
sqr x &= x \times x \\
dbl x &= x + x \\
main &= \text{sqr (dbl (2 + 3))}
\end{align*}
\]

Roughly, any expression that can be evaluated or **reduced** by using the equations as rewrite rules is called a **reducible expression** or **redex**.

Assuming arithmetic, the redexes of the body of main are: 2 + 3

dbl (2 + 3)

sqr (dbl (2 + 3))

Evaluation Orders (2)

Thus, in general, many possible reduction orders. Innermost, leftmost redex first is called **Applicative Order Reduction** (AOR). Recall:

\[
\begin{align*}
sqr x &= x \times x \\
dbl x &= x + x \\
main &= \text{sqr (dbl (2 + 3))}
\end{align*}
\]

Starting from main:

\[
\begin{align*}
main &\Rightarrow \text{sqr (dbl (2 + 3))} \\
&\Rightarrow \text{sqr (dbl 5)} \\
&\Rightarrow \text{sqr (5 + 5)} \\
&\Rightarrow 10 \times 10 \\
&\Rightarrow 100
\end{align*}
\]

**Call-By-Value** (CBV) = AOR except no evaluation under \(\lambda\) (inside function bodies).

Evaluation Orders (3)

Outermost, leftmost redex first is called **Normal Order Reduction** (NOR):

\[
\begin{align*}
main &\Rightarrow \text{sqr (dbl (2 + 3))} \\
&\Rightarrow \text{dbl (2 + 3) * dbl (2 + 3)} \\
&\Rightarrow ((2 + 3) + (2 + 3)) \times \text{dbl (2 + 3)} \\
&\Rightarrow (5 + (2 + 3)) \times \text{dbl (2 + 3)} \\
&\Rightarrow (5 + 5) \times \text{dbl (2 + 3)} \\
&\Rightarrow 10 \times \text{dbl (2 + 3)} \\
&\Rightarrow ... \\
&\Rightarrow 10 \times 10 \\
&\Rightarrow 100
\end{align*}
\]

(Applications of arithmetic operations only considered redexes once arguments are numbers.) **Call-By-Name** (CBN) = NOR except no evaluation under \(\lambda\).
Why NOR or CBN? (1)

NOR and CBN seem rather inefficient. Any use?
  • Best possible termination properties.
    A pure functional languages is just the \(\lambda\)-calculus in disguise. Two central theorems:
    - Church-Rosser Theorem I: No term has more than one normal form.
    - Church-Rosser Theorem II: If a term has a normal form, then it can be found through NOR.

Why NOR or CBN? (2)

• More expressive power; e.g.:
  - “Infinite” data structures
  - Circular programming
  - Custom control constructs (great for EDSLs)

• More declarative code as control aspects (order of evaluation) left implicit.

Why NOR or CBN? (3)

• More reuse. E.g. consider:
  
  \[
  \text{any :: (a -> Bool) -> [a] -> Bool}
  \text{any p = or . map p}
  \]

Under AOR/CBV, we would have to inline all functions to avoid doing too much work:

\[
\text{any :: (a -> Bool) -> [a] -> Bool}
\text{any p [] = False}
\text{any p (y:ys) = y || any p ps}
\]

(Assume \((||)\) has “short-circuit” semantics.) No reuse.

(See references for in-depth discussion.)

Exercise 1

Consider:

\[
f x = 1
g x = g x
main = f (g 0)
\]

Attempt to evaluate \texttt{main} using both AOR and NOR. Which order is the more efficient in this case? (Count the number of reduction steps to normal form.)
Strict vs. Non-strict Semantics (1)

- \(\perp\), or “bottom”, the **undefined value**, representing **errors** and **non-termination**.
- A function \(f\) is strict iff:
  \[
  f \perp = \perp
  \]

For example, \(+\) is strict in both its arguments:

\[
\begin{align*}
(0/0) + 1 &= \perp + 1 = \perp \\
1 + (0/0) &= 1 + \perp = \perp
\end{align*}
\]

Lazy Evaluation (1)

**Lazy evaluation** or **Call-by-Need** is a technique for **implementing** CBN more efficiently:

- A redex is evaluated **only if needed**.
- **Sharing** employed to avoid duplicating redexes.
- Once evaluated, a redex is **updated** with the result to avoid evaluating it more than once.

As a result, under lazy evaluation, any one redex is evaluated **at most once**.

Strict vs. Non-strict Semantics (2)

Again, consider:

\[
\begin{align*}
f \ x &= 1 \\
g \ x &= g \ x
\end{align*}
\]

What is the value of \(f \ (0/0)\)? Or of \(f \ (g \ 0)\)?

- AOR: \(f \ (0/0) \Rightarrow \perp; \ f \ (g \ 0) \Rightarrow \perp\)
  - Conceptually, \(f \perp = \perp\); i.e., \(f\) is strict.
- NOR: \(f \ (0/0) \Rightarrow 1; \ f \ (g \ 0) \Rightarrow 1\)
  - Conceptually, \(f \perp = 1\); i.e., \(f\) is non-strict.

Thus, NOR results in non-strict semantics.

Lazy Evaluation (2)

Recall:

\[
\begin{align*}
sqr \ x &= x \times x \\
dbl \ x &= x + x
\end{align*}
\]

\[
\begin{align*}
\text{main} = &\quad \text{sqr} \ (\text{dbl} \ (2 + 3)) \\
&\Rightarrow \text{dbl} \ (2 + 3) \ast (\bullet) \\
&\Rightarrow (\ (2 + 3) + (\bullet)) \ast (\bullet) \\
&\Rightarrow (5 + (\bullet)) \ast (\bullet) \\
&\Rightarrow 10 \ast (\bullet) \\
&\Rightarrow 100
\end{align*}
\]
Exercise 2

Evaluate `main` using AOR, NOR, and lazy evaluation:

\[
\begin{align*}
  f \ x \ y \ z &= x \times z \\
  g \ x &= f \ (x \times x) \ (x \times 2) \ x \\
  \text{main} &= g \ (1 + 2)
\end{align*}
\]

(Only consider an application of an arithmetic operator a redex once the arguments are numbers.)

How many reduction steps in each case?

**Answer:** 7, 8, 6 respectively

Infinite Data Structures (1)

```haskell
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

from n = n : from (n+1)
nats = from 0

main = take 5 nats
```

Infinite Data Structures (2)

```haskell
main ⇒ \[\text{take 5} \ (\bullet) \ ⇒ 0 : \text{take 4} \ (\bullet) \ ⇒ 0 : 1 : \text{take 3} \ (\bullet) \ ⇒ 0 : 1 : 2 : 3 : 4 : \text{take 0} \ (\bullet) \ ⇒ [0, 1, 2, 3, 4] \]

\[\text{nats} \ ⇒ 2 \ : \text{from 0} \ ⇒ 3 : \text{from 1} \ ⇒ 5 : 1 : \text{from 2} \ ⇒ 7 : 0 : 1 : 2 : 3 : 4 : \text{from 5} \]
```

Circular Data Structures (2)

```haskell
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

ones = 1 : ones

main = take 5 ones
```
Circular Data Structures (2)

\[
\text{main} \Rightarrow \text{take 5} \Rightarrow \text{1:take 4} \Rightarrow \text{1:1:take 3} \Rightarrow \text{1:1:1:take 0} \Rightarrow [1,1,1,1,1]
\]

Exercise 3

Given the following tree type

\[
\text{data Tree} = \text{Empty} \mid \text{Node Tree Int Tree}
\]

define:

- An infinite tree where every node is labelled by 1.
- An infinite tree where every node is labelled by its depth from the root node.

Exercise 3: Solution

\[
\text{treeOnes = Node treeOnes 1 treeOnes}
\]

\[
\text{treeFrom n = Node (treeFrom (n + 1)) n (treeFrom (n + 1))}
\]

\[
\text{treeDepths = treeFrom 0}
\]

Circular Programming (1)

A type of non-empty trees:

\[
\text{data Tree} = \text{Leaf Int} \mid \text{Node Tree Tree}
\]

Suppose we would like to write a function that replaces each leaf integer in a given tree with the \textit{smallest} integer in that tree.

How many passes over the tree are needed?

\textit{One!}
Circular Programming (2)

Write a function that replaces all leaf integers by a given integer, and returns the new tree along with the smallest integer of the given tree:

```haskell
fmr :: Int -> Tree -> (Tree, Int)
fmr m (Leaf i) = (Leaf m, i)
fmr m (Node tl tr) =
  (Node tl' tr', min ml mr)
where
  (tl', ml) = fmr m tl
  (tr', mr) = fmr m tr
```

Circular Programming (3)

For a given tree \( t \), the desired tree is now obtained as the solution to the equation:

\[
(t', m) = fmr m t
\]

Thus:

```haskell
findMinReplace t = t'
where
  (t', m) = fmr m t
```

Intuitively, this works because \( fmr \) can compute its result without needing to know the value of \( m \).

Circular Programming (4)

Operational view:

```haskell
fmr (snd •) = (min (min 3 1) 2)
```

A Simple Spreadsheet Evaluator

```
a b c
1 37
2 14 2 16
3 7 21
```

The evaluated sheet is again simply the solution to the stated equation. No need to worry about evaluation order. Any caveats?
Consider the problem of numbering a possibly infinitely deep tree in breadth-first order:

1
2 3
4 5 6 7
8 9
10
11 12 13 14

The following algorithm is due to G. Jones and J. Gibbons (1992), but the presentation differs. Consider the following tree type:

```
data Tree a = Empty
  | Node (Tree a) a (Tree a)
```

Define:

- `width t i` The width of a tree `t` at level `i` (0 origin).
- `label t i j` The `j`th label at level `i` of a tree `t` (0 origin).

The following system of equations defines breadth-first numbering:

1. `label t 0 0 = 1`
2. `label t (i + 1) 0 = label t i 0 + width t i`
3. `label t i (j + 1) = label t i j + 1`

Note that `label t i 0` is defined for *all* levels `i` (as long as the widths of all tree levels are finite).

The code that follows sets up the defining system of equations:

- **Streams** (infinite lists) of labels are used as a *mediating data structure* to allow equations to be set up between adjacent nodes within levels and between the last node at one level and the first node at the next.
- **Idea:** the tree numbering function for a subtree takes a stream of labels for the *first node* at each level, and returns a stream of labels for the *node after the last node* at each level.
As there manifestly are *no cyclic dependences* among the equations, we can entrust the details of solving them to the lazy evaluation machinery in the safe knowledge that a solution will be found.

\[
\text{bfn :: Tree } a \rightarrow \text{ Tree Integer} \\
bfn \ t \ = \ t' \\
\quad \text{where} \\
\quad (\text{ns}, \ t') \ = \ \text{bfnAux} \ \ (1 : \ \text{ns}) \ \ t
\]

\[
\text{bfnAux :: [Integer] } \rightarrow \text{ Tree } a \\
\quad \rightarrow \ (([\text{Integer}], \ \text{Tree Integer}) \\
\quad \text{bfnAux ns Empty} \ = \ (\text{ns}, \ \text{Empty}) \\
\quad \text{bfnAux (n : ns)} \ (\text{Node } tl' \ _\ tr') \ = \ ((n + 1) : \ \text{ns'}, \ \\
\quad \quad \quad \quad \quad \quad \text{Node } tl' \ n \ tr')
\]

\[
\begin{align*}
(\text{ns'}, \ tl') & = \text{bfnAux ns tl} \\
(\text{ns'}, \ tr') & = \text{bfnAux ns' tr}
\end{align*}
\]
Dynamic Programming

**Dynamic Programming:**
- Create a **table** of all subproblems that ever will have to be solved.
- Fill in table without regard to whether the solution to that particular subproblem will be needed.
- Combine solutions to form overall solution.

**Lazy Evaluation** is a perfect match as saves us from having to worry about finding a suitable evaluation order.

The Triangulation Problem (1)

Select a set of **chords** that divides a convex polygon into triangles such that:
- no two chords cross each other
- the sum of their length is minimal.

We will only consider computing the minimal length.

See Aho, Hopcroft, Ullman (1983) for details.

The Triangulation Problem (2)

Let $S_{is}$ denote the subproblem of size $s$ starting at vertex $v_i$ of finding the minimum triangulation of the polygon $v_i, v_{i+1}, \ldots, v_{i+s-1}$ (counting modulo the number of vertices).

Subproblems of size less than 4 are trivial.

Solving $S_{is}$ is done by solving $S_{i,k+1}$ and $S_{i+k,s-k}$ for all $k$, $1 \leq k \leq s - 2$.

The obvious recursive formulation results in $3^{s-4}$ (non-trivial) calls.

But for $n \geq 4$ vertices there are only $n(n - 3)$ non-trivial subproblems!
The Triangulation Problem (4)

\[ v_i \]

\[ v_{i+k} \]

\[ S_{i,k+1} \]

\[ S_{i+k,s-k} \]

\[ v_{i+s-1} \]

The Triangulation Problem (5)

- Let \( C_{is} \) denote the minimal triangulation cost of \( S_{is} \).
- Let \( D(v_p, v_q) \) denote the length of a chord between \( v_p \) and \( v_q \) (length is 0 for non-chords; i.e. adjacent \( v_p \) and \( v_q \)).
- For \( s \geq 4 \):
  \[
  C_{is} = \min_{k \in [1,s-2]} \left\{ C_{i,k+1} + C_{i+k,s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \right\}
  \]
- For \( s < 4 \), \( S_{is} = 0 \).

The Triangulation Problem (6)

These equations can be transliterated straight into Haskell:

```haskell
triCost :: Polygon -> Double
triCost p = cost!(0,n) where
  cost = array ((0,0), (n-1,n))
    [ ( (i,s),
        minimum [ cost!(i, k+1)
            + cost!((i+k) `mod` n, s-k)
            + dist p i ((i+k) `mod` n)
            + dist p ((i+k) `mod` n)
            + ((i+s-1) `mod` n)
            | k <- [1..s-2] ]
            | i <- [0..n-1], s <- [4..n] ] ++
        [ ( (i,s), 0.0 )
            | i <- [0..n-1], s <- [0..3] ]
        ]
  n = snd (bounds b) + 1
```

Attribute Grammars (1)

Lazy evaluation is also very useful for evaluation of Attribute Grammars:

- The attribution function is defined recursively over the tree:
  - takes inherited attributes as extra arguments;
  - returns a tuple of all synthesised attributes.
- As long as there exists some possible attribution order, lazy evaluation will take care of the attribute evaluation.
Attribute Grammars (2)

- The earlier examples on Circular Programming and Breadth-first Numbering can be seen as instances of this idea.

Reading (1)

- Lennart Augustsson. More Points for Lazy Evaluation. 2 May 2011. [http://augustss.blogspot.co.uk/2011/05/more-points-for-lazy-evaluation-in.html](http://augustss.blogspot.co.uk/2011/05/more-points-for-lazy-evaluation-in.html)

Reading (2)