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)

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:

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

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:

\[
\begin{align*}
2 + 3 \\
\text{dbl} (2 + 3) \\
\text{sqrt} (\text{dbl} (2 + 3))
\end{align*}
\]

Evaluation Orders (2)

Thus, in general, many possible reduction orders.

Innermost, leftmost redex first is called **Applicative Order Reduction** (AOR). Recall:

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

Starting from `main`:

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

This is just **Call-By-Value**.

Evaluation Orders (3)

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

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

(Applications of arithmetic operations only considered redexes once arguments are numbers.)

Demand-driven evaluation or **Call-By-Need**

Why Normal Order Reduction? (1)

NOR seems rather inefficient. Any use?

- Best possible termination properties.

A pure functional language is just the λ-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 NOR will find it.
Why Normal Order Reduction? (2)

- More expressive power; e.g.:
  - “Infinite” data structures
  - Circular programming
- More declarative code as control aspects (order of evaluation) left implicit.

Strict vs. Non-strict Semantics (1)

- ⊥, 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:

\[ (0/0) + 1 = \perp + 1 = \perp \]
\[ 1 + (0/0) = 1 + \perp = \perp \]

Exercise 1

Consider:

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

Attempt to evaluate 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 (2)

Again, consider:

\[ f \ x = 1 \]
\[ g \ x = g \ x \]

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

- AOR: \( f \ (0/0) = \perp; \ f \ (g \ 0) \to \perp \)

  Conceptually, \( f \perp = \perp \); i.e., \( f \) is strict.

- NOR: \( f \ (0/0) = 1; \ f \ (g \ 0) \to 1 \)

  Conceptually, \( foo \perp = 1 \); i.e., \( foo \) is non-strict.

Thus, NOR results in non-strict semantics.

Lazy Evaluation (1)

Lazy evaluation is a technique for implementing NOR 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.

Lazy Evaluation (2)

Recall:

\[ \text{sqr} \ x = x * x \]
\[ \text{dbl} \ x = x + x \]
\[ \text{main} = \text{sqr} \ (\text{dbl} \ (2+3)) \]

\[ \Rightarrow \text{dbl} \ (2+3) \]
\[ \Rightarrow (2+3) \]
\[ \Rightarrow 5 \]
\[ \Rightarrow 100 \]

Exercise 2

Evaluate main using AOR, NOR, and lazy evaluation:

\[ f \ y z = x * z \]
\[ g \ x = f \ (x * x) \ (x * 2) \ x \]
\[ \text{main} = g \ (1 + 2) \]

(Only consider an applications 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)

\[ n \]ats = from 0
\[ \text{main} = \text{take} \ 5 \ n \]ats

\[ \Rightarrow 1 \]
\[ \Rightarrow \text{take} \ 4 \ [\] \]
\[ \Rightarrow 4 \]
\[ \Rightarrow 0:1: \text{take} \ 3 \ [\] \]
\[ \Rightarrow 6 \]
\[ \Rightarrow 0:1:2:3: \text{take} \ 2 \ [\] \]
\[ \Rightarrow 8 \]
\[ \Rightarrow 0:1:2:3:4: \text{take} \ 1 \ [\] \]
\[ \Rightarrow 10 \]
\[ \Rightarrow 0:1:2:3:4: \text{take} \ 0 \ [\] \]
\[ \Rightarrow 12 \]
\[ \Rightarrow 0:1:2:3:4: \hat{=} \]
\[ \Rightarrow 0:1:2:3:4: \]

Infinite Data Structures (2)

main \{ ^1 \text{take} \ 5 \} \{ ^4 \text{take} \ 4 \} \{ ^8 \text{take} \ 3 \} \{ ^{16} \text{take} \ 2 \} \{ ^{32} \text{take} \ 1 \} \Rightarrow [0,1,2,3,4]

\[ \Rightarrow ^{0:1:2:3:4:} \text{take} \ 0 \] \Rightarrow [0,1,2,3,4]

\[ \Rightarrow ^{0:1:2:3:4:} \text{from} \ 0 \Rightarrow ^{0:1:2:3:4:} \text{from} \ 1 \]

\[ \Rightarrow 0:1:2:3:4: \text{from} \ 5 \]
Circular Data Structures (2)

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

\[
ones = 1 : ones
\]

\[
main = take 5\ ones
\]

Exercise 3

Given the following tree type

\[
data\ Tree = Empty \mid \ Node\ Tree\ Int\ Tree
\]

\[
define:
\]
\[
\bullet\ An\ infinite\ tree\ where\ every\ node\ is\ labelled\ by 1.
\]
\[
\bullet\ An\ infinite\ tree\ where\ every\ node\ is\ labelled\ by\ its\ depth\ from\ the\ root\ node.
\]

Circular Programming (1)

A non-empty tree type:

\[
data\ Tree = Leaf\ Int \mid \ Node\ Tree\ Tree
\]

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

How many passes over the tree are needed?

\[One!\]

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:

\[
fmr :: Int -> Tree -> (Tree, Int)
fmr m (Leaf i) = (Leaf m, i)
fmr m (Node tl tr) = (Node tl' tr', min ml mr)
\]

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:

\[(t', m) = \text{fmr} m\ t\]

Thus:

\[
\text{findMinReplace}\ t = t'
\]

where

\[(t', m) = \text{fmr} m\ t\]

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

A Simple Spreadsheet Evaluator

The evaluated sheet is again simply the solution to the stated equation. No need to worry about evaluation order. Any caveats?

Breadth-first Numbering (1)

Consider the problem of numbering a possibly infinitely deep tree in breadth-first order:
Breadth-first Numbering (2)

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

```haskell
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).

Breadth-first Numbering (3)

The following system of equations defines breadth-first numbering:

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

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

Breadth-first Numbering (4)

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.

Breadth-first Numbering (5)

- As there manifestly are no cyclic dependencies 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.

Breadth-first Numbering (6)

```haskell
bfn :: Tree a -> Tree Integer
bfn t = t'
  where
      (ns, t') = bfnAux (1 : ns) t

bfnAux :: [Integer] -> Tree a
        -> (Tree Integer)
    where
      bfnAux Empty     Empty = (Empty, Empty)
      bfnAux (n : ns) (Node tl _ tr) =
        (n + 1 : ns', tr')
        where
          (ns', tl') = bfnAux ns tl
          (ns'', tr') = bfnAux ns' tr
```

Breadth-first Numbering (7)

![Diagram](image)

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 $S_i$ 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 $S_i$ 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 $3^{s-3}$ (non-trivial) calls.

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

The Triangulation Problem (3)

• Let $C_{is}$ denote the minimal triangulation cost of $S_i$.

• Let $D(v_i,v_j)$ denote the length of a chord between $v_i$ and $v_j$ (length is 0 for non-chords; i.e. adjacent $v_i$ and $v_j$).

• 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 $s < 4$, $S_i = 0$.

The Triangulation Problem (4)

These equations can be transcribed 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)
    | k <- [1..s-2] ]
    | i <- [0..n-1], s <- [4..n] ]
  | 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.

Reading


• Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In Functional Programming Languages and Computer Architecture, FPCA’87, 1987

Attribute Grammars (2)

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