LiU-FP2010 Part II: Lecture 2

Lazy Functional Programming

Henrik Nilsson

University of Nottingham, UK
Imperative vs. Declarative (1)

- **Imperative Languages**:
  - Implicit state.
  - Computation essentially a sequence of side-effecting actions.
  - Examples: Procedural and OO languages
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*
Imperative vs. Declarative (2)

Another perspective:

- *Algorithm = Logic + Control*

- Declarative programming emphasises the logic (“what”) rather than the control (“how”).
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:
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.
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.
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:

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

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
\[\text{dbl } (2 + 3)\]
\[\text{sqr } (\text{dbl } (2 + 3))\]
Thus, in general, many possible reduction orders. Innermost, leftmost redex first is called \textit{Applicative Order Reduction} (AOR). Recall:

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

Starting from \texttt{main}:

\[
\begin{array}{c}
\text{main} \quad \Rightarrow \quad \text{sqr} \ (\text{dbl} \ (2 + 3)) \quad \Rightarrow \quad \text{sqr} \ (\text{dbl} \ 5) \\
\Rightarrow \quad \text{sqr} \ (5 + 5) \quad \Rightarrow \quad \text{sqr} \ 10 \quad \Rightarrow \quad 10 \ast 10 \quad \Rightarrow \quad 100
\end{array}
\]

This is just \textit{Call-By-Value}.
Evaluation Orders (3)

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

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

(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 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 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.
Exercise 1

Consider:

\[
\begin{align*}
f \ x &= 1 \\
g \ x &= g \ x \\
\text{main} &= f \ (g \ 0)
\end{align*}
\]

Attempt to evaluate \text{main} using both AOR and NOR. Which order is the more efficient in this case? (Count the number of reduction steps to normal form.)
• $\bot$, or “bottom”, the *undefined value*, representing *errors* and *non-termination*.

• A function $f$ is *strict* iff:

$$f(\bot) = \bot$$

For example, $+$ is strict in both its arguments:

$$(0/0) + 1 = \bot + 1 = \bot$$

$$1 + (0/0) = 1 + \bot = \bot$$
Again, consider:

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

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

- **AOR:** \( f(0/0) \Rightarrow \bot \); \( f(g\ 0) \Rightarrow \bot \)
  Conceptually, \( f\ \bot = \bot \); i.e., \( f \) is strict.

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

Thus, NOR results in non-strict semantics.
Lazy evaluation is a *technique for implementing NOR* more efficiently:
Lazy Evaluation (1)

Lazy evaluation is a technique for implementing NOR more efficiently:

- A redex is evaluated only if needed.
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.
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.
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:

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

Recall:

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

main =

\[
\text{sqr (dbl (2+3))}
\]
Recall:

$sqr \ x = x \times x$

$dbl \ x = x + x$

main =

$sqr \ (dbl \ (2+3))$

⇒

$dbl \ (2 + 3)$

⇒

$( (2 + 3) + (\cdot) ) \times (\cdot)$
Lazy Evaluation (2)

Recall:

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

Recall:

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

\[
\text{sqr } (\text{dbl } (2 + 3)) \\
\Rightarrow \text{dbl } (2 + 3) \\
\Rightarrow (2 + 3) + (\cdot) \\
\Rightarrow (5 + (\cdot)) \\
\Rightarrow 10
\]
Lazy Evaluation (2)

Recall:

\[ \text{sqr } x = x \times x \]
\[ \text{dbl } x = x + x \]

\text{main} =

\[ \text{sqr } (\text{dbl } (2+3)) \]

\[ \Rightarrow \text{dbl } (2+3) \]
\[ \Rightarrow (2+3) + 0 \]
\[ \Rightarrow 5 + 0 \]
\[ \Rightarrow 10 \]
\[ \Rightarrow 100 \]
Exercise 2

Evaluate \texttt{main} using AOR, NOR, and lazy evaluation:

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

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

How many reduction steps in each case?
Exercise 2

Evaluate \texttt{main} using AOR, NOR, and lazy evaluation:

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

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

How many reduction steps in each case?

\textbf{Answer:} 7, 8, 6 respectively
Infinite Data Structures (1)

\[
\begin{align*}
\text{take } 0 \, xs & = [] \\
\text{take } n \, [] & = [] \\
\text{take } n \, (x:xs) & = x : \text{take } (n-1) \, xs \\
\text{from } n & = n : \text{from } (n+1) \\
nats & = \text{from } 0 \\
\text{main} & = \text{take } 5 \, nats
\end{align*}
\]
main

nats
main \Rightarrow^1 \text{take 5} \ldots
Infinite Data Structures (2)

\[
\text{main} \Rightarrow^1 \text{take 5 (\textbullet)}
\]

\[
\text{nats} \Rightarrow^2 \text{from 0}
\]
Infinite Data Structures (2)

main $\Rightarrow^1$ take 5 (•)

nats $\Rightarrow^2$ from 0 $\Rightarrow^3$ 0: from 1
Infinite Data Structures (2)

\[
\begin{align*}
\text{nats} & \Rightarrow^2 \text{from 0} \Rightarrow^3 0 : \text{from 1} \\
\text{main} & \Rightarrow^1 \text{take 5} (\bullet) \Rightarrow^4 0 : \text{take 4} (\bullet)
\end{align*}
\]
Infinite Data Structures (2)

main \Rightarrow^1 \text{take 5 (●)} \Rightarrow^4 0:\text{take 4 (●)}

\text{nats} \Rightarrow^2 \text{from 0} \Rightarrow^3 0:\text{from 1}

\Rightarrow^5 0:1:\text{from 2}
Infinite Data Structures (2)

\[
\begin{align*}
\text{main} & \Rightarrow^1 \text{take 5 (●)} \Rightarrow^4 0: \text{take 4 (●)} \\
\Rightarrow^6 0:1: \text{take 3 (●)} \\
\text{nats} & \Rightarrow^2 \text{from 0} \Rightarrow^3 0: \text{from 1} \\
\Rightarrow^5 0:1: \text{from 2}
\end{align*}
\]
Infinite Data Structures (2)

\[
\text{main} \Rightarrow \text{take 5 } (\bullet) \Rightarrow \text{from 0} \Rightarrow \text{take 4 } (\bullet) \Rightarrow \text{from 1} \Rightarrow \text{from 2} \Rightarrow \ldots
\]
Infinite Data Structures (2)

\[
\text{main} \Rightarrow^1 \text{take 5 (•)} \Rightarrow^4 0: \text{take 4 (•)} \\
\Rightarrow^6 0:1: \text{take 3 (•)} \Rightarrow^8 \ldots
\]

\[
\text{nats} \Rightarrow^2 \text{from 0} \Rightarrow^3 0: \text{from 1} \\
\Rightarrow^5 0:1: \text{from 2} \Rightarrow^7 \ldots
\]
Infinite Data Structures (2)

\[\text{main} \Rightarrow 1 \text{take 5 (●)} \Rightarrow 4 0:\text{take 4 (●)}\]

\[\Rightarrow 6 0:1:\text{take 3 (●)} \Rightarrow 8 \ldots\]

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

\[\Rightarrow 5 0:1:\text{from 2} \Rightarrow 7 \ldots \Rightarrow 0:1:2:3:4:\text{from 5}\]
Infinite Data Structures (2)

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

\[ \Rightarrow \text{nats} \Rightarrow \text{from 0} \Rightarrow \text{from 1} \Rightarrow \text{from 2} \Rightarrow \text{from 5} \]
Infinite Data Structures (2)

\[
\text{main} \Rightarrow^1 \text{take 5 (●)} \Rightarrow^4 0: \text{take 4 (●)} \\
\Rightarrow^6 0:1: \text{take 3 (●)} \Rightarrow^8 \ldots \\
\Rightarrow 0:1:2:3:4: \text{take 0 (●)} \Rightarrow [0, 1, 2, 3, 4]
\]

\[
\text{nats} \Rightarrow^2 \text{from 0} \Rightarrow^3 0: \text{from 1} \\
\Rightarrow^5 0:1: \text{from 2} \Rightarrow^7 \ldots \Rightarrow 0:1:2:3:4: \text{from 5}
\]
Circular Data Structures (2)

\[
\begin{align*}
\text{take } 0 \; \text{xs} &= [] \\
\text{take } n \; [] &= [] \\
\text{take } n \; (x:xs) &= x : \text{take } (n-1) \; xs \\
\text{ones} &= 1 : \text{ones} \\
\text{main} &= \text{take } 5 \; \text{ones}
\end{align*}
\]
Circular Data Structures (2)

main

ones
Circular Data Structures (2)

main \Rightarrow^1 \text{take 5 (●)}

\text{ones}
Circular Data Structures (2)

main ⇒\textsuperscript{1} take 5 ( )

ones ⇒\textsuperscript{2} 1 : •
Circular Data Structures (2)

main $\Rightarrow^1$ take 5 (•) $\Rightarrow^3$ 1:take 4 (•)

ones $\Rightarrow^2$ 1 : •
Circular Data Structures (2)

\[ \text{main} \Rightarrow^1 \text{take 5 (●)} \Rightarrow^3 1:\text{take 4 (●)} \]

\[ \Rightarrow^4 1:1:\text{take 3 (●)} \]

\[ \text{ones} \Rightarrow^2 1 : \]
Circular Data Structures (2)

\[
\text{main} \Rightarrow^1 \text{take 5 (•)} \Rightarrow^3 1:\text{take 4 (•)} \\
\Rightarrow^4 1:1:\text{take 3 (•)} \Rightarrow^5 \ldots
\]

\[
\text{ones} \Rightarrow^2 1 : \bullet
\]
Circular Data Structures (2)

\[
\text{main} \Rightarrow^1 \text{take 5 (\bullet)} \Rightarrow^3 1:\text{take 4 (\bullet)} \\
\Rightarrow^4 1:1:\text{take 3 (\bullet)} \Rightarrow^5 \ldots \\
\Rightarrow 1:1:1:1:1:1:\text{take 0 (\bullet)}
\]
Circular Data Structures (2)

\[
\begin{align*}
\text{main} & \Rightarrow^1 \text{take 5 } (\bullet) \Rightarrow^3 1 : \text{take 4 } (\bullet) \\
& \Rightarrow^4 1 : 1 \text{:take 3 } (\bullet) \Rightarrow^5 \ldots \\
& \Rightarrow 1 : 1 : 1 : 1 : 1 : \text{take 0 } (\bullet) \Rightarrow [1, 1, 1, 1, 1, 1]
\end{align*}
\]

\[\text{ones} \Rightarrow^2 1 : \bullet\]
Exercise 3

Given the following tree type

```haskell
data Tree = Empty | 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 rote node.
Exercise 3: Solution

```
treeOnes = Node treeOnes 1 treeOnes

treeFrom n = Node (treeFrom (n + 1)) n (treeFrom (n + 1))

treeDepths = treeFrom 0
```
A non-empty tree type:

```haskell
data Tree = Leaf Int | Node Tree Tree
```
Circular Programming (1)

A non-empty tree type:

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

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.
A non-empty tree type:

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

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?
Circular Programming (1)

A non-empty tree type:

```haskell
data Tree = Leaf Int | 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:

\[
\text{fmr} :: \text{Int} \rightarrow \text{Tree} \rightarrow (\text{Tree}, \text{Int})
\]

\[
\text{fmr} \ m \ (\text{Leaf} \ i) = (\text{Leaf} \ m, i)
\]

\[
\text{fmr} \ m \ (\text{Node} \ tl \ tr) =
\]

\[
(\text{Node} \ tl' \ tr', \ \text{min} \ ml \ mr)
\]

where

\[
(tl', ml) = \text{fmr} \ m \ tl
\]

\[
(tr', mr) = \text{fmr} \ m \ tr
\]
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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c3 + c2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a3 * b2</td>
<td>2</td>
<td>a2 + b2</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td></td>
<td>a2 + a3</td>
</tr>
</tbody>
</table>

\[ r = \text{array (bounds } s) \]
\[
\begin{array}{l}
[ ((i,j), \text{eval } r (s!(i,j))) \\
| (i,j) <- \text{indices } s ]
\end{array}
\]

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:

```
1
/  \
2   3
/ \  /  \
4  5 6  7
/ \  /  /  \
8 9 10 11 12 13 14
```
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:

\[
\begin{align*}
\text{label} \ t \ 0 \ 0 &= 1 \quad (1) \\
\text{label} \ t \ (i+1) \ 0 &= \text{label} \ t \ i \ 0 + \text{width} \ t \ i \quad (2) \\
\text{label} \ t \ i \ (j+1) &= \text{label} \ t \ i \ j \ + \ 1 \quad (3)
\end{align*}
\]

Note that \(\text{label} \ t \ i \ 0\) 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:
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.
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.
• 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.
Breadth-first Numbering (6)

\[ bfn :: \text{Tree } a \rightarrow \text{Tree Integer} \]

\[ bfn t = t' \]

\[ \text{where} \]

\[ (ns, t') = bfnAux (1 : ns) t \]

\[ bfnAux :: [\text{Integer}] \rightarrow \text{Tree } a \rightarrow ([\text{Integer}], \text{Tree Integer}) \]

\[ bfnAux \text{ ns Empty } = (\text{ns, Empty}) \]

\[ bfnAux (n : ns) \text{ (Node } tl \_ \text{ _ tr}) = (n + 1 : ns'', \text{ Node } tl' \ n \ tr') \]

\[ \text{where} \]

\[ (ns'', tl'') = bfnAux \text{ ns } tl \]

\[ (ns'''', tr''') = bfnAux \text{ ns''' } tr \]
Breadth-first Numbering (8)
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)
The Triangulation Problem (3)

- 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)
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 $s \geq 4$:
  \[
  C_{is} = \min_{k \in [1, s-2]} \left\{ \begin{array}{c}
  C_{i,k+1} + C_{i+k,s-k} \\
  + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1})
  \end{array} \right\}
  \]
- 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
```
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.
The earlier examples on Circular Programming and Breadth-first Numbering can be seen as instances of this idea.
Reading


Reading
