# COMP4075/G54RFP: Lecture 2 & 3

## *Pure Functional Programming: Exploiting Laziness*

Henrik Nilsson

University of Nottingham, UK

# Recap: Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

# Recap: Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

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

# Recap: 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.

# Recap: 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.

# Recap: 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.

# Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

```
sqr (dbl (2 + 3))
```

# Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

sqr (dbl (2 + 3))

$\Rightarrow$ $\boxed{\texttt{dbl (2 + 3)}}$ * (•)

# Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

```
  sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * (•)
⇒ ((2 + 3) + (•)) * (•)
```

# Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
    sqr (dbl (2+3))
```

```
sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * (•)
⇒ ((2 + 3) + (•)) * (•)
⇒ (5 + (•)) * (•)
```

# Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
    sqr (dbl (2+3))
```

```
sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * (•)
⇒ ((2 + 3) + (•)) * (•)
⇒ (5 + (•)) * (•)
⇒ 10 * (•)
```

# Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

```
  sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * (●)
⇒ ((2 + 3) + (●)) * (●)
⇒ (5 + (●)) * (●)
⇒ 10 * (●)
⇒ 100
```

# Infinite Data Structures (1)

```
take 0 _        = []
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)

`main`

`nats`

# Infinite Data Structures (2)

`main` $\Rightarrow^1$ `take 5 (•)`

`nats`

# Infinite Data Structures (2)

`main` $\Rightarrow^1$ `take 5 (•)`

`nats` $\Rightarrow^2$ `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)

$$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{4} 0 : \underline{\texttt{take 4 }(\bullet)}$$

$$\boxed{\underline{\texttt{nats}}} \Rightarrow^{2} \underline{\texttt{from 0}} \Rightarrow^{3} 0 : \boxed{\underline{\texttt{from 1}}}$$

# Infinite Data Structures (2)

$$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 (}\bullet\texttt{)}} \Rightarrow^{4} \texttt{0:}\underline{\texttt{take 4 (}\bullet\texttt{)}}$$

$$\boxed{\underline{\texttt{nats}}} \Rightarrow^{2} \underline{\texttt{from 0}} \Rightarrow^{3} \texttt{0:}\boxed{\underline{\texttt{from 1}}}$$

$$\Rightarrow^{5} \texttt{0:1:}\boxed{\underline{\texttt{from 2}}}$$

# Infinite Data Structures (2)

`main` $\Rightarrow^{1}$ `take 5 (•)` $\Rightarrow^{4}$ `0:take 4 (•)`

$\Rightarrow^{6}$ `0:1:take 3 (•)`

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

$\Rightarrow^{5}$ `0:1:from 2`

# Infinite Data Structures (2)

`main` $\Rightarrow^{1}$ `take 5 (•)` $\Rightarrow^{4}$ `0:take 4 (•)`

$\Rightarrow^{6}$ `0:1:take 3 (•)`

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

$\Rightarrow^{5}$ `0:1:from 2` $\Rightarrow^{7}$ `...`

# Infinite Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{4} \texttt{0:}\underline{\texttt{take 4 }(\bullet)}$

$\Rightarrow^{6} \texttt{0:1:}\underline{\texttt{take 3 }(\bullet)} \Rightarrow^{8} \ldots$

$\boxed{\underline{\texttt{nats}}} \Rightarrow^{2} \underline{\texttt{from 0}} \Rightarrow^{3} \texttt{0:}\boxed{\underline{\texttt{from 1}}}$

$\Rightarrow^{5} \texttt{0:1:}\boxed{\underline{\texttt{from 2}}} \Rightarrow^{7} \ldots$

# Infinite Data Structures (2)

$$\underline{\mathtt{main}} \Rightarrow^{1} \underline{\mathtt{take\ 5\ (\bullet)}} \Rightarrow^{4} \mathtt{0:}\underline{\mathtt{take\ 4\ (\bullet)}}$$

$$\Rightarrow^{6} \mathtt{0:1:}\underline{\mathtt{take\ 3\ (\bullet)}} \Rightarrow^{8} \mathtt{...}$$

$$\boxed{\underline{\mathtt{nats}}} \Rightarrow^{2} \underline{\mathtt{from\ 0}} \Rightarrow^{3} \mathtt{0:}\boxed{\underline{\mathtt{from\ 1}}}$$

$$\Rightarrow^{5} \mathtt{0:1:}\boxed{\underline{\mathtt{from\ 2}}} \Rightarrow^{7} \mathtt{...} \Rightarrow \mathtt{0:1:2:3:4:}\boxed{\underline{\mathtt{from\ 5}}}$$

# Infinite Data Structures (2)

main $\Rightarrow^{1}$ take 5 (•) $\Rightarrow^{4}$ 0:take 4 (•)

$\Rightarrow^{6}$ 0:1:take 3 (•) $\Rightarrow^{8}$ ...

$\Rightarrow$ 0:1:2:3:4:take 0 (•)

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

$\Rightarrow^{5}$ 0:1:from 2 $\Rightarrow^{7}$ ... $\Rightarrow$ 0:1:2:3:4:from 5

# Infinite Data Structures (2)

`main` $\Rightarrow^{1}$ `take 5 (•)` $\Rightarrow^{4}$ `0:take 4 (•)`

$\Rightarrow^{6}$ `0:1:take 3 (•)` $\Rightarrow^{8}$ `...`

$\Rightarrow$ `0:1:2:3:4:take 0 (•)` $\Rightarrow$ `[0,1,2,3,4]`

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

$\Rightarrow^{5}$ `0:1:from 2` $\Rightarrow^{7}$ `...` $\Rightarrow$ `0:1:2:3:4:from 5`

# Circular Data Structures (1)

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

ones = 1 : ones

main = take 5 ones
```

# Circular Data Structures (2)

main

ones

# Circular Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)}$

$\boxed{\underline{\texttt{ones}}}$

# Circular Data Structures (2)

`main` $\Rightarrow^1$ `take 5 (•)`

`ones` $\Rightarrow^2$ `1 : •`

# Circular Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{3} 1 : \underline{\texttt{take 4 }(\bullet)}$

$\boxed{\underline{\texttt{ones}}} \Rightarrow^{2} \boxed{1 : \bullet}$

# Circular Data Structures (2)

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

$\Rightarrow^{4}$ 1:1:take 3 (•)

ones $\Rightarrow^{2}$ 1 : •

# Circular Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{3} \texttt{1:}\underline{\texttt{take 4 }(\bullet)}$

$\Rightarrow^{4} \texttt{1:1:}\underline{\texttt{take 3 }(\bullet)} \Rightarrow^{5} \ldots$

$\boxed{\underline{\texttt{ones}}} \Rightarrow^{2} \boxed{1 : \bullet}$

# Circular Data Structures (2)

$$\underline{\texttt{main}} \Rightarrow^1 \underline{\texttt{take 5 }(\bullet)} \Rightarrow^3 \texttt{1:}\underline{\texttt{take 4 }(\bullet)}$$

$$\Rightarrow^4 \texttt{1:1:}\underline{\texttt{take 3 }(\bullet)} \Rightarrow^5 \ldots$$

$$\Rightarrow \texttt{1:1:1:1:1:}\underline{\texttt{take 0 }(\bullet)}$$

$$\boxed{\underline{\texttt{ones}}} \Rightarrow^2 \boxed{\texttt{1 : }\bullet}$$

# Circular Data Structures (2)

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

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

$\Rightarrow$ `1:1:1:1:1:take 0 (•)` $\Rightarrow$ `[1,1,1,1,1]`

`ones` $\Rightarrow^2$ `1 : •`

# Exercise

Given the following tree type

```
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 root node.

# Exercise: Solution

```
treeOnes = Node treeOnes 1 treeOnes

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

treeDepths = treeFrom 0
```

# Circular Programming (1)

A non-empty tree type:

```
data Tree = Leaf Int | Node Tree Tree
```

# Circular Programming (1)

A non-empty tree type:

```
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.

# Circular Programming (1)

A non-empty tree type:

```
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?

# Circular Programming (1)

A non-empty tree type:

```
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:

```
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:

```
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`.

# A Simple Spreadsheet Evaluator (1)

| | a | b | c |
|---|---|---|---|
| 1 | c3 + c2 | | |
| 2 | a3 * b2 | 2 | a2 + b2 |
| 3 | 7 | | a2 + a3 |

s

$\Rightarrow$

| | a | b | c |
|---|---|---|---|
| 1 | 37 | | |
| 2 | 14 | 2 | 16 |
| 3 | 7 | | 21 |

s′

```
s' = array (bounds s)
           [ (r, evalCell s' (s ! r))
           | r <- indices s ]
```

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

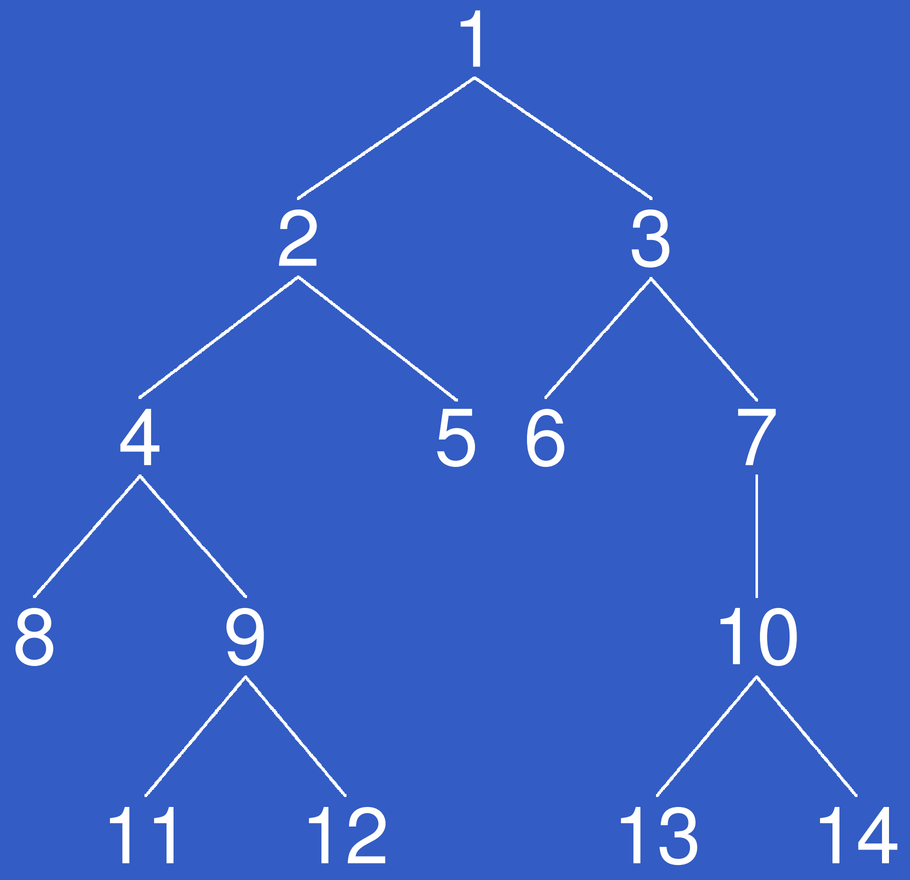
# A Simple Spreadsheet Evaluator (2)

As it is quite instructive, let us develop this evaluator together. Some definitions to get us started:

```
type CellRef = (Char, Int)

type Sheet a = Array CellRef a

data BinOp = Add | Sub | Mul | Div

data Exp = Lit Double
         | Ref CellRef
         | App BinOp Exp Exp
```

# 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:

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

Define:

| | |
|---|---|
| $\mathrm{width}\ t\ i$ | The width of a tree $t$ at level $i$ (0 origin). |
| $\mathrm{label}\ t\ i\ j$ | 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:

$$
\begin{aligned}
\text{label}\ t\ 0\ 0 &= 1 && (1)\\
\text{label}\ t\ (i+1)\ 0 &= \text{label}\ t\ i\ 0 + \text{width}\ t\ i && (2)\\
\text{label}\ t\ i\ (j+1) &= \text{label}\ t\ i\ j + 1 && (3)
\end{aligned}
$$

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:

# 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.

# 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 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.

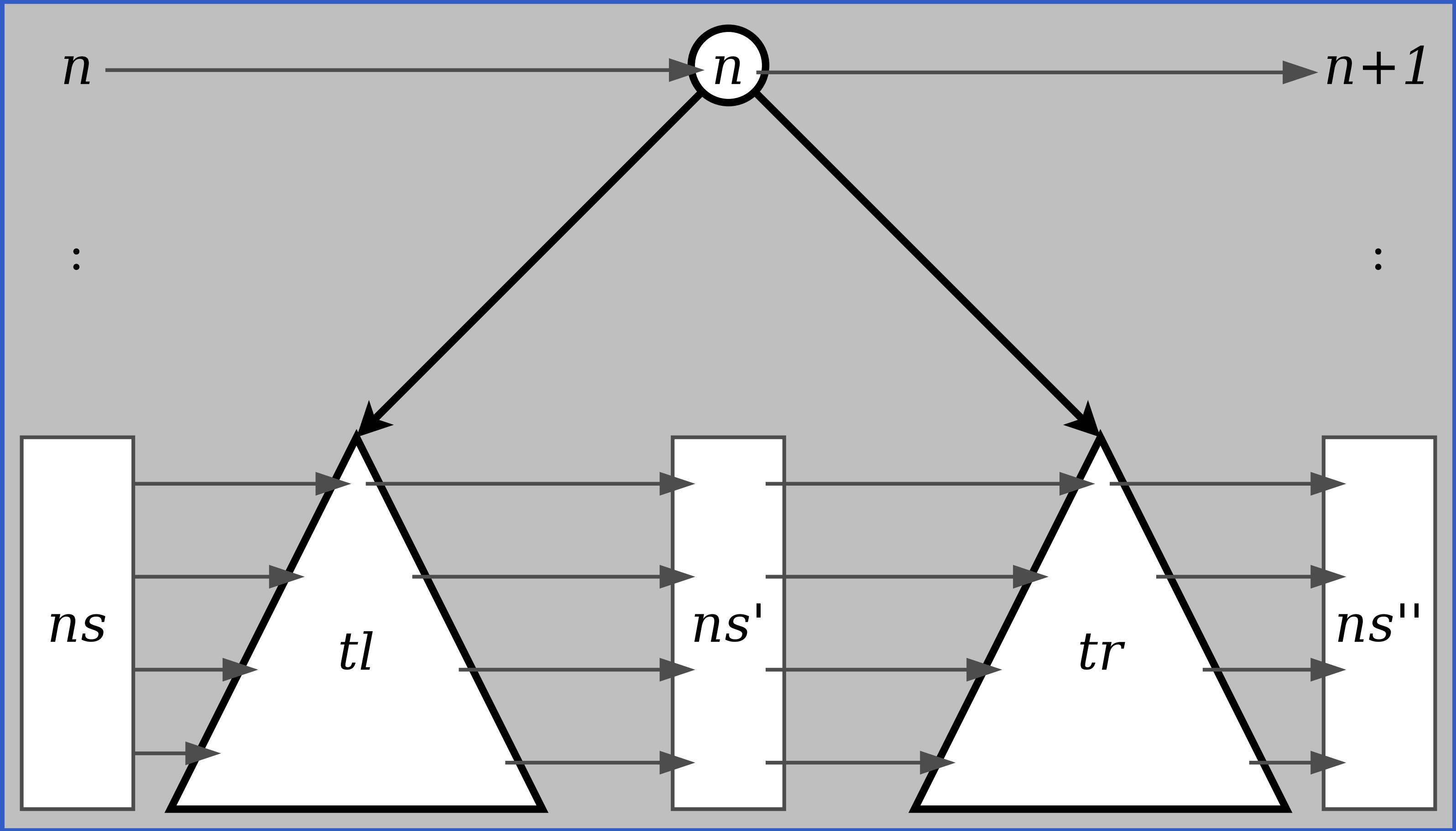
# Breadth-first Numbering (6)

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

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

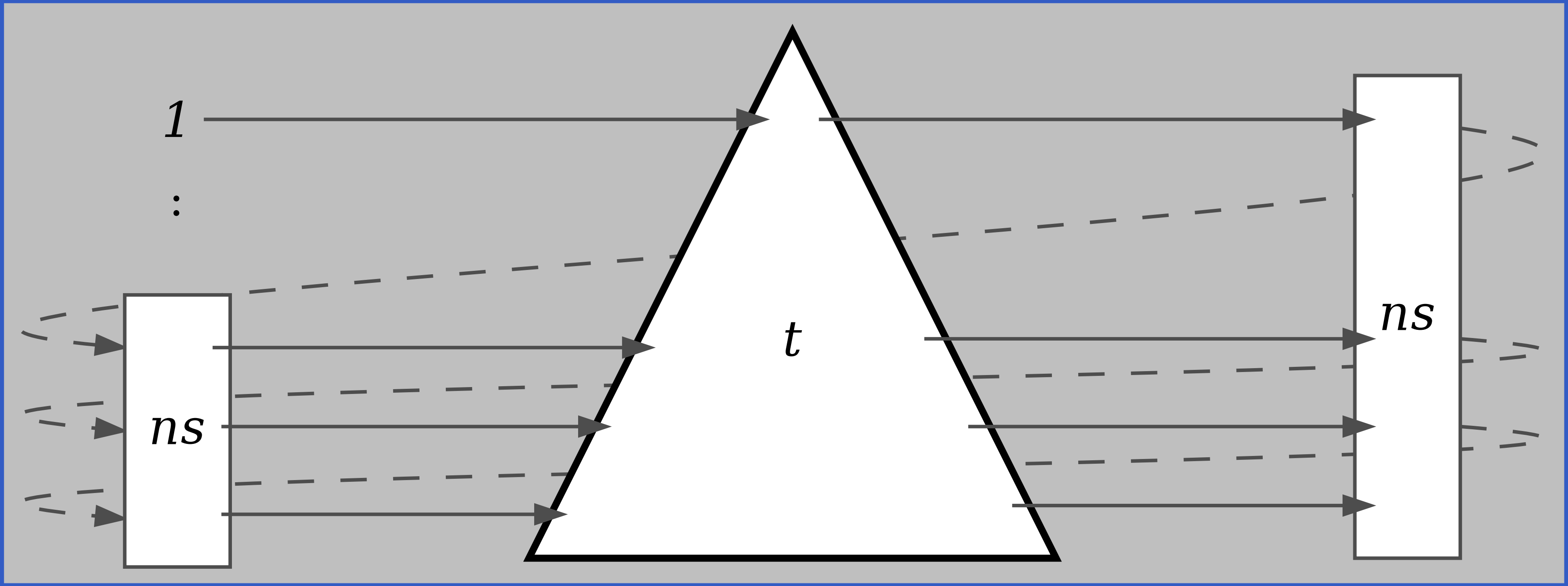
*Eqns (1) & (2)*

*Eqn (3)*

# Breadth-first Numbering (7)



# Breadth-first Numbering (8)



# 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.

# 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 perfect match: no need to worry about finding a suitable evaluation order.

# 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 perfect match: no need to worry about finding a suitable evaluation order.

In effect, using laziness to implement limited form of ***memoization***.

# The Triangulation Problem (1)

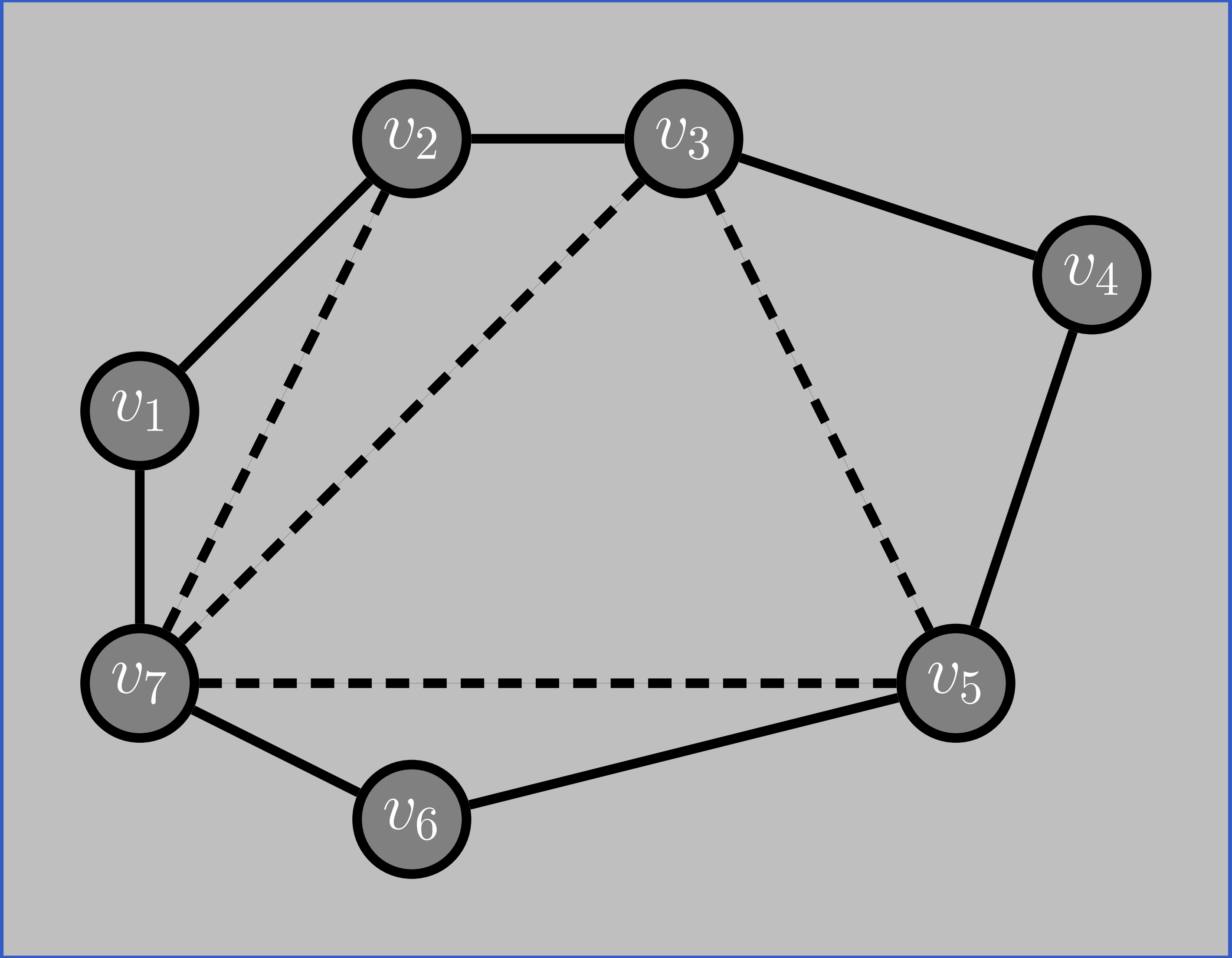
Select a set of ***chords*** that divides a convex polygon into triangles such that:

- 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.

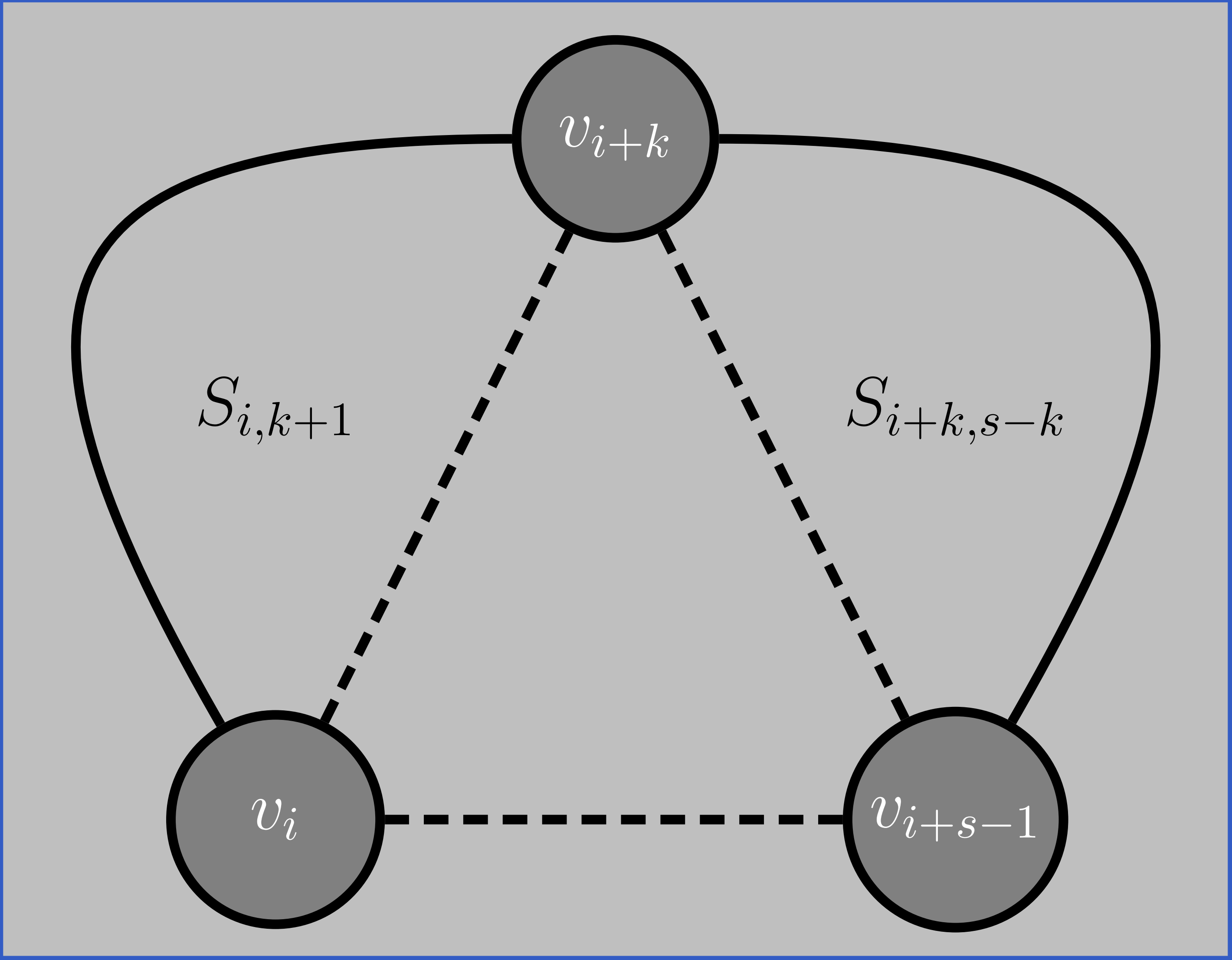
# The Triangulation Problem (2)



# 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 $s \geq 4$:

$$C_{is} = \min_{k \in [1, s-2]} \left\{ \begin{array}{l} C_{i,k+1} + C_{i+k,s-k} \\ +D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \end{array} \right\}$$

- For $s < 4$, $C_{is} = 0$.

# The Triangulation Problem (6)

These equations can be transliterated straight into 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

- John W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 1994.
- John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–197, April 1989.
- Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming Languages and Computer Architecture, FPCA'87*, 1987

# Reading

- Geraint Jones and Jeremy Gibbons. *Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips.* Technical Report TR-31-92, Oxford University Computing Laboratory, 1992.
- Alfred Aho, John Hopcroft, Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.