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.

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.

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:

```
sqr x = x * x
dbl x = x + x
main = sqr (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
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:

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

Starting from `main`:

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

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

Evaluation Orders (3)

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

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

(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 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.

Exercise 1

Consider:

```
f x = 1
g x = g x
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 (1)

- $\bot$, 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:

$$\begin{aligned} (0/0) + 1 &= \perp + 1 = \perp \\ 1 + (0/0) &= 1 + \perp = \perp \end{aligned}$$

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)` $\Rightarrow \bot$; `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, `foo` $\bot = 1$; 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:

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:

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

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

Lazy Evaluation (2)

Recall:

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

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

Lazy Evaluation (2)

Recall:

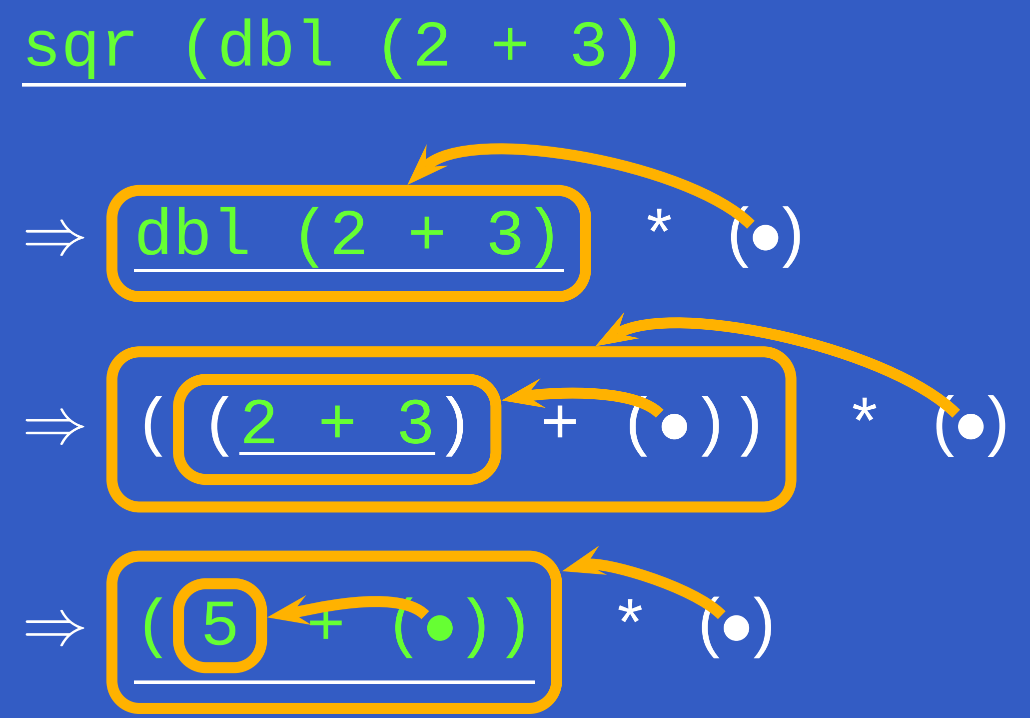
```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

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

Lazy Evaluation (2)

Recall:

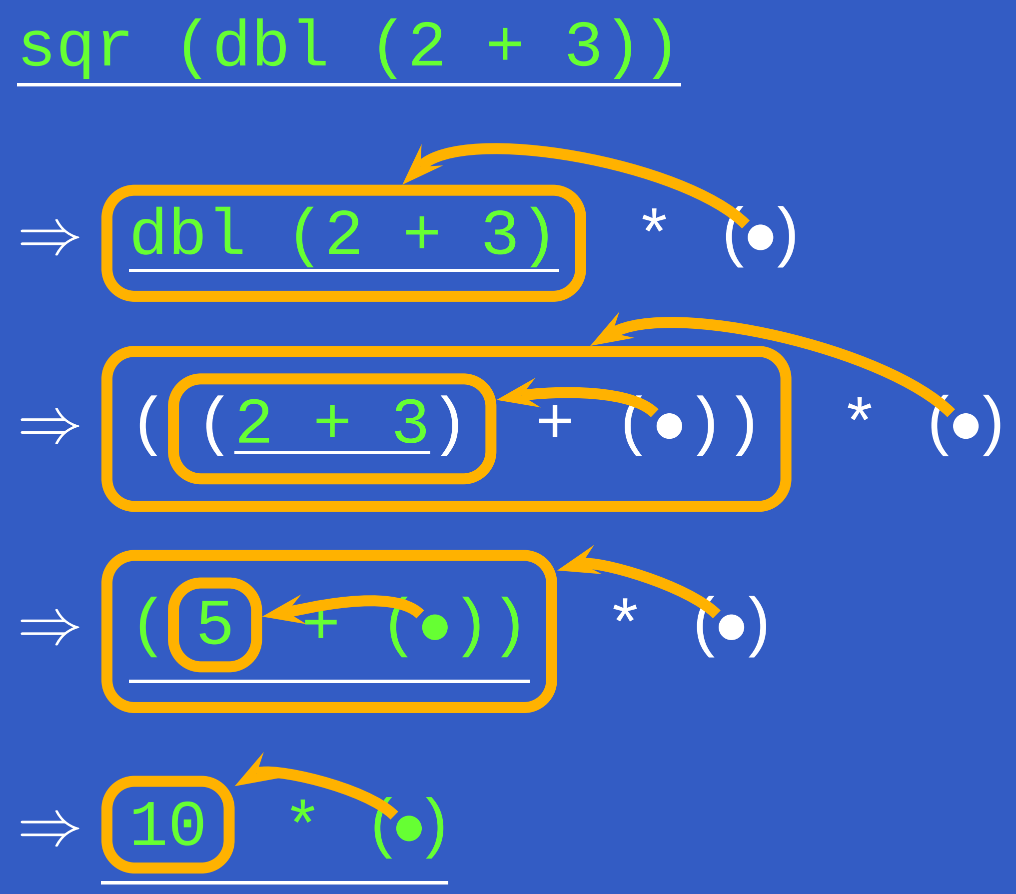
```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```



Lazy Evaluation (2)

Recall:

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



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
```

Exercise 2

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

```
f x y z  =  x * z
g x      =  f (x * x) (x * 2) x
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?

Exercise 2

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

```
f x y z  =  x * z
g x      =  f (x * x) (x * 2) x
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)

```
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)

main

nats

Infinite Data Structures (2)

`main` $\Rightarrow^{1}$ `take 5 (•)`

`nats`

Infinite Data Structures (2)

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

$\underline{\texttt{nats}} \Rightarrow^{2} \underline{\texttt{from 0}}$

Infinite Data Structures (2)

`main` $\Rightarrow^1$ `take 5 (•)`

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

Infinite Data Structures (2)

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

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

Infinite Data Structures (2)

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

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

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

Infinite Data Structures (2)

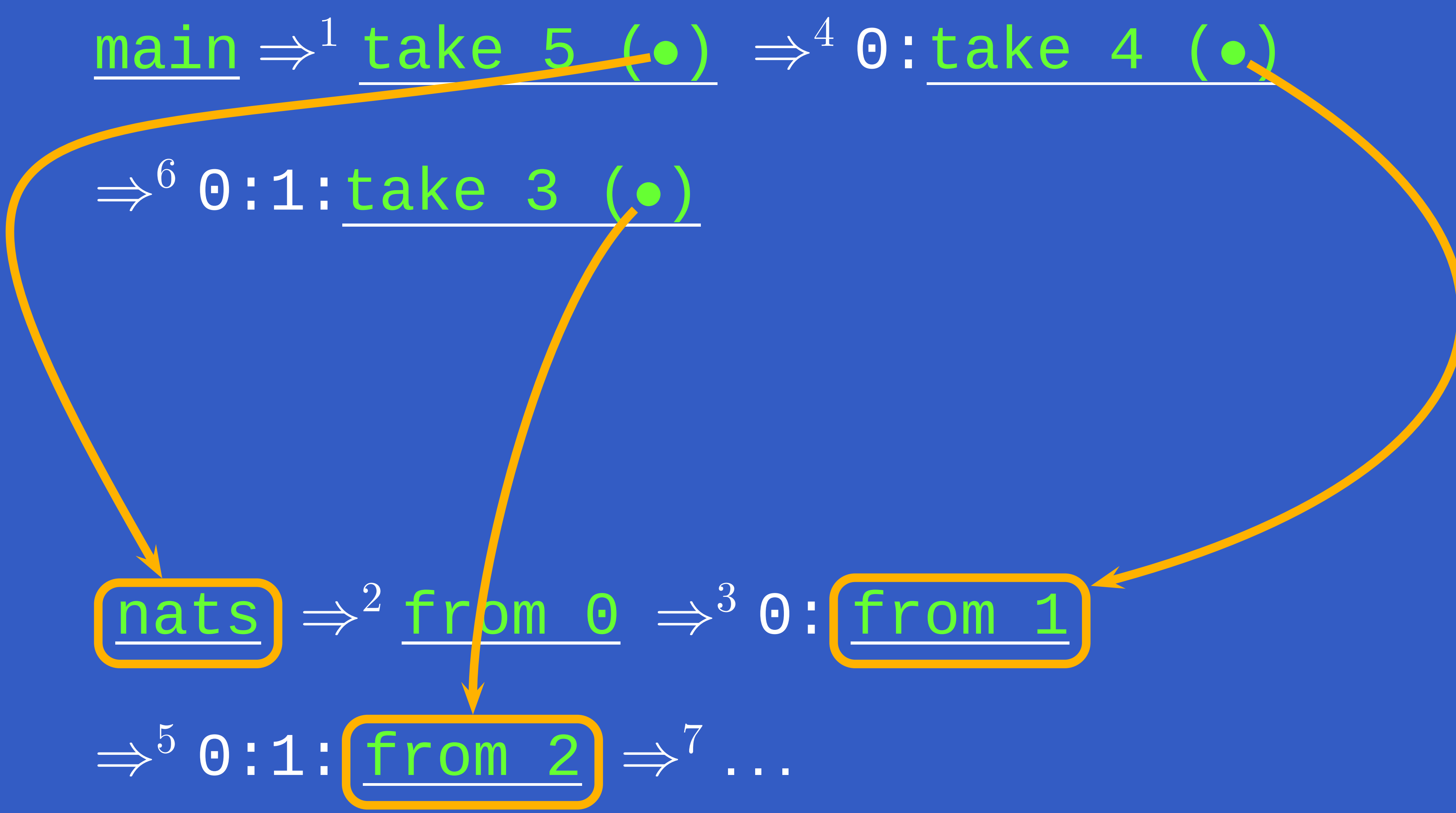
`main` $\Rightarrow^{1}$ `take 5 (•)` $\Rightarrow^{4}$ `0:take 4 (•)`

$\Rightarrow^{6}$ `0:1:take 3 (•)`

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

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

Infinite Data Structures (2)



Infinite Data Structures (2)

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

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

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

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

Infinite Data Structures (2)

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

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

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 (•)

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 (2)

```
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)

main

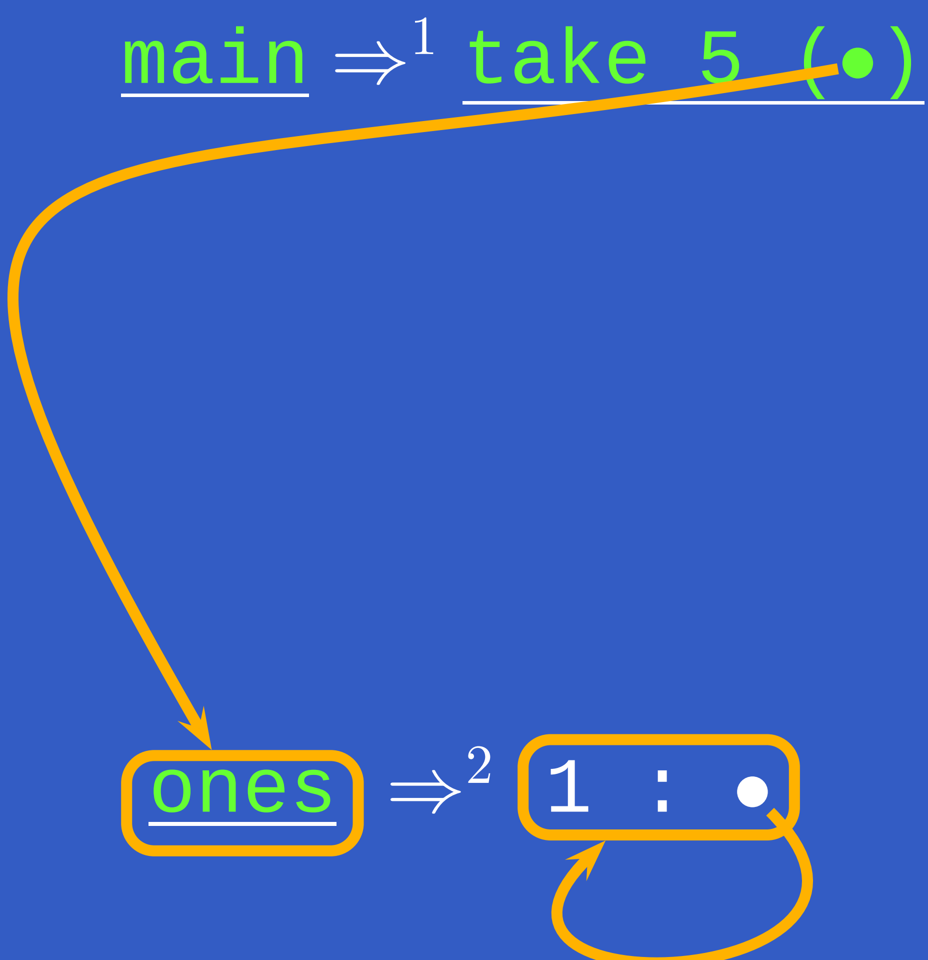
ones

Circular Data Structures (2)

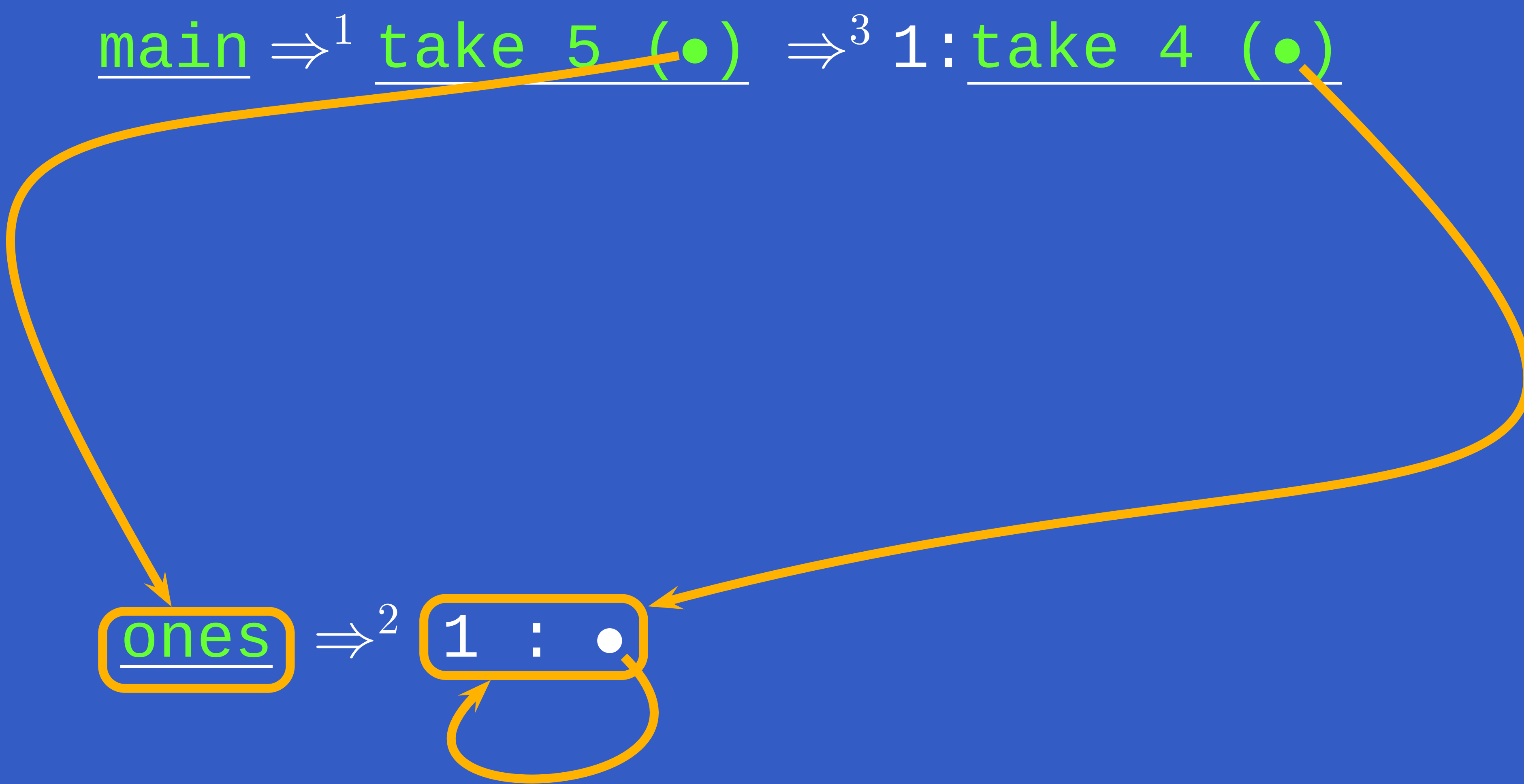
`main` $\Rightarrow^1$ `take 5 (•)`

`ones`

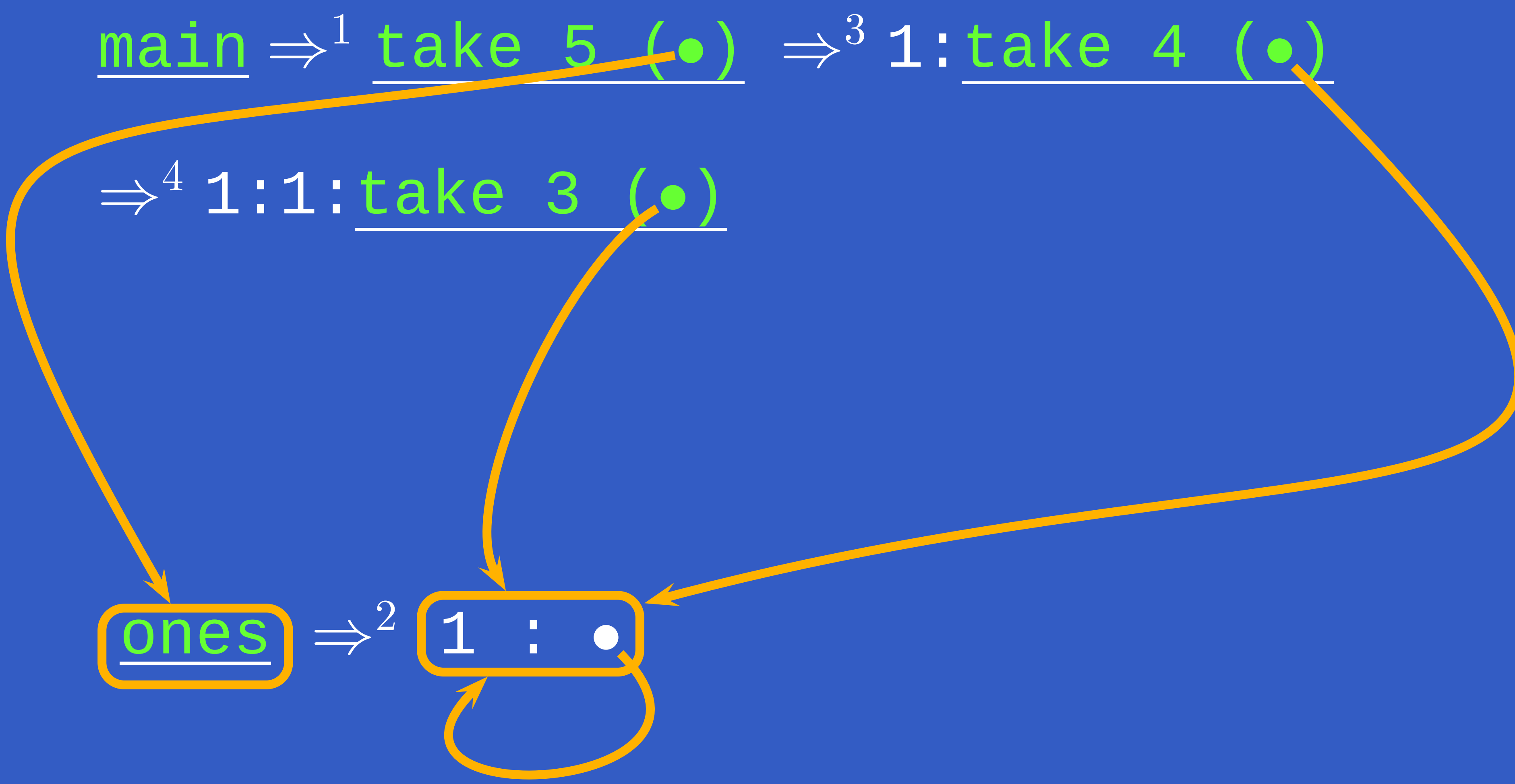
Circular Data Structures (2)



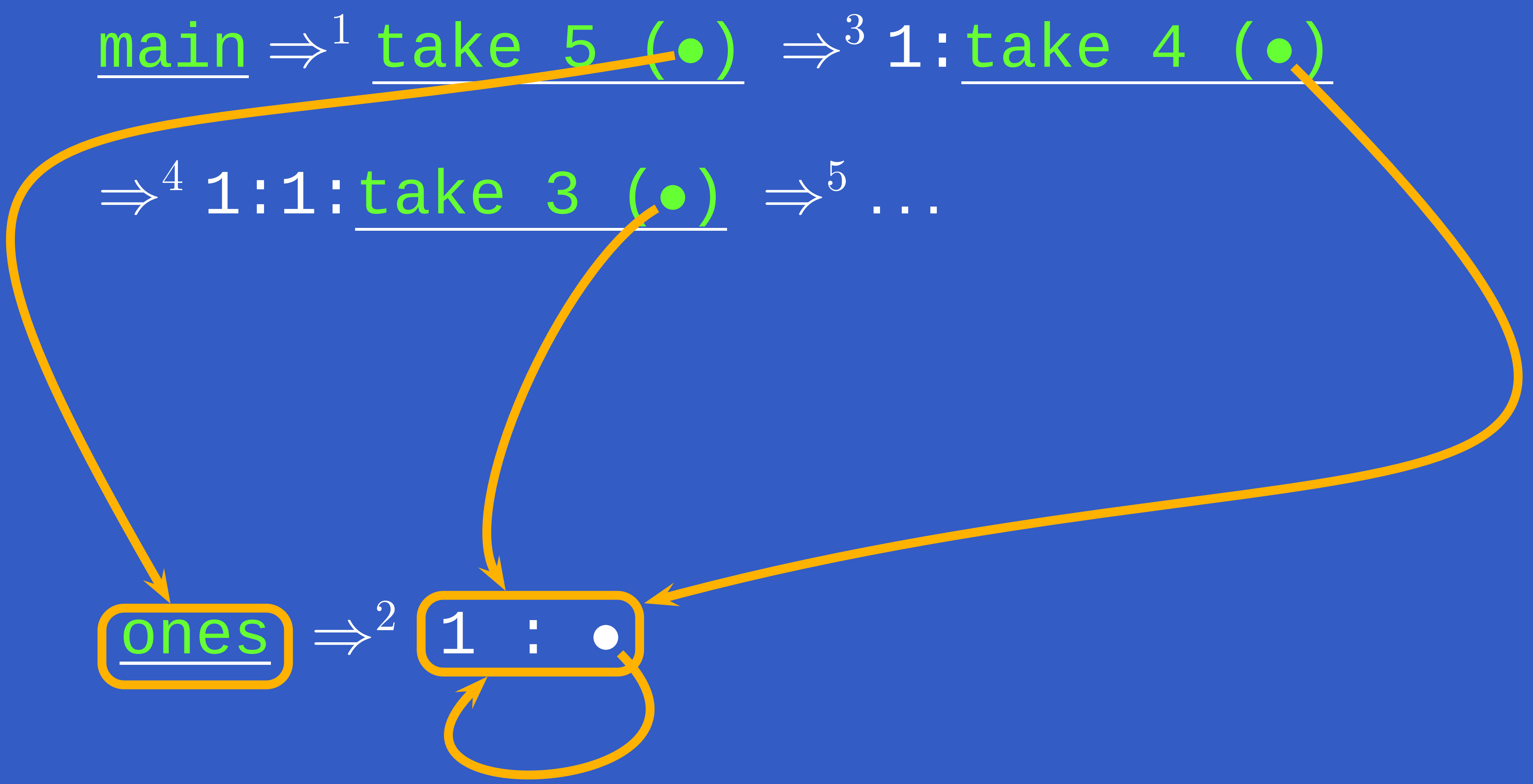
Circular Data Structures (2)



Circular Data Structures (2)



Circular Data Structures (2)



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 (•)

ones $\Rightarrow^2$ 1 : •

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 3

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 rote node.

Exercise 3: 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

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

s

⇒

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

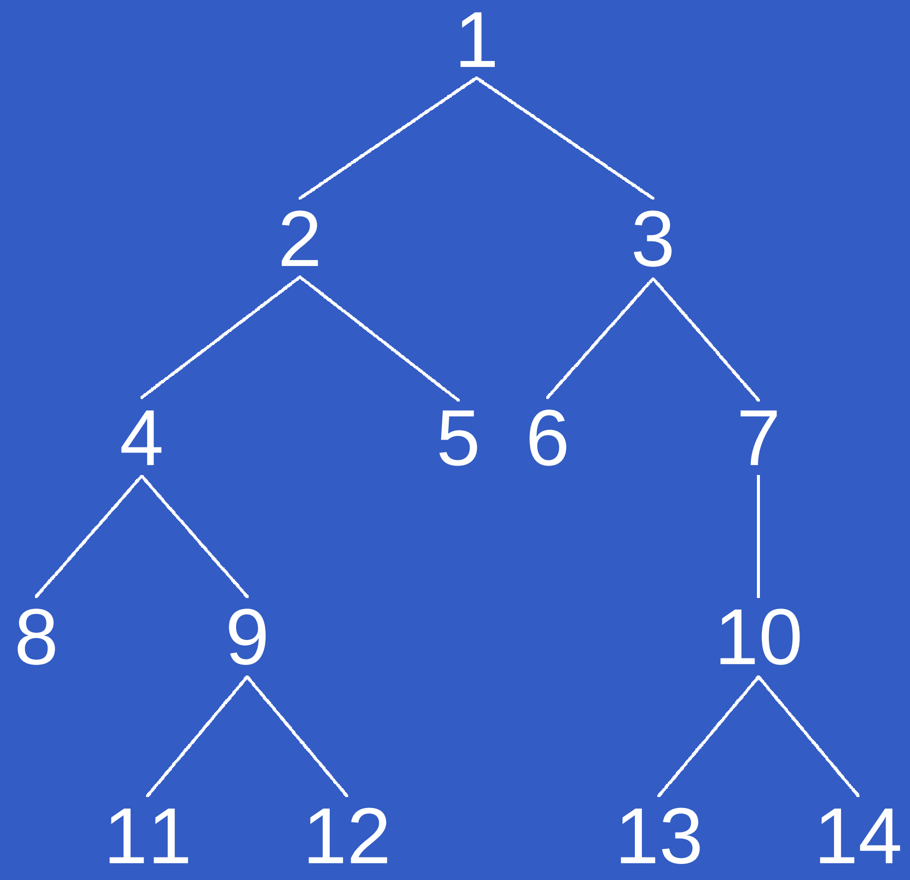
r

```
r = array (bounds s)
          [ ((i,j), eval r (s!(i,j)))
          | (i,j) <- indices s ]
```

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:

```
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 jth 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)

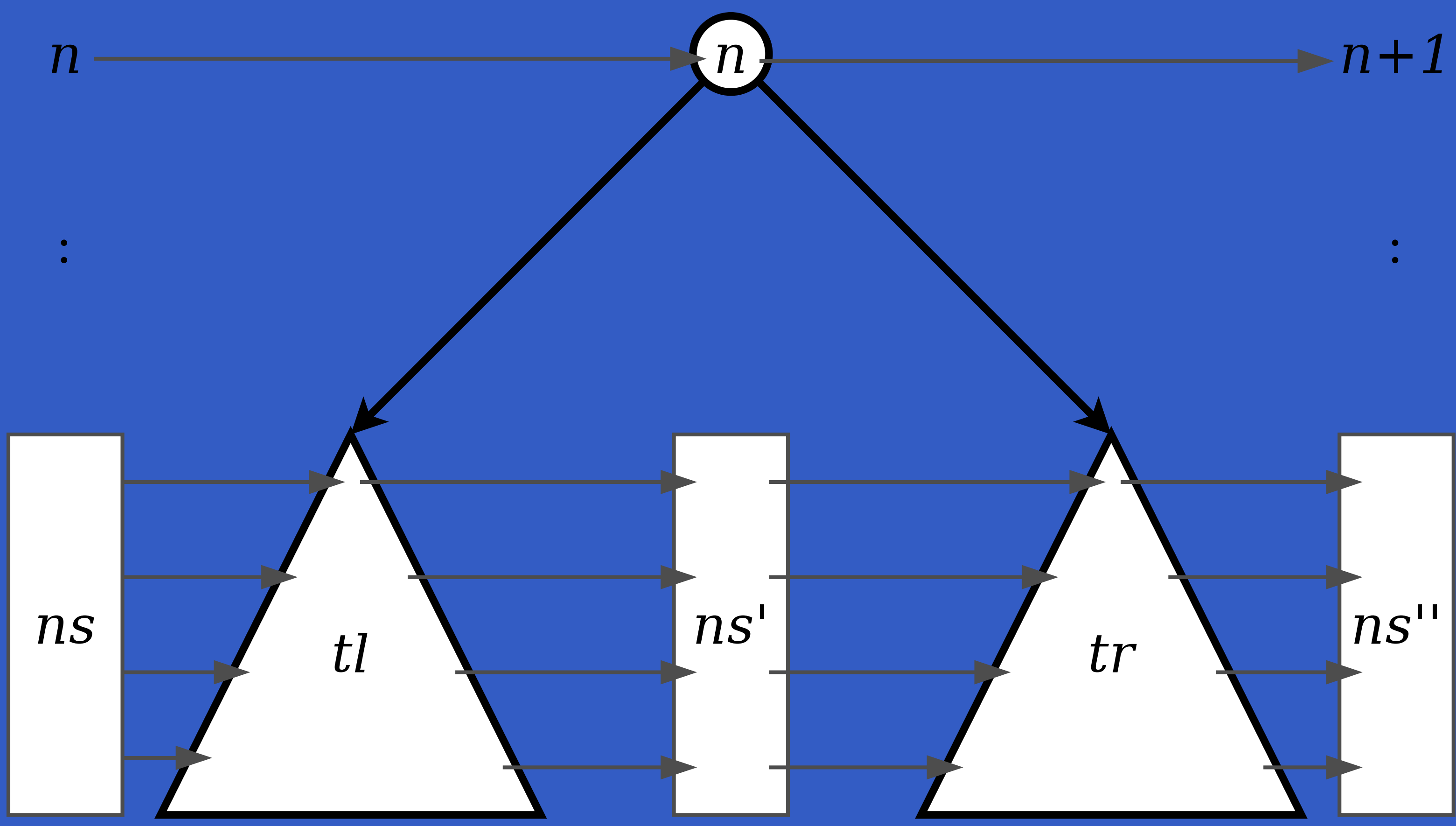
```
bfn :: Tree a -> Tree Integer
bfn t = t'
    where
        (ns, t') = bfnAux (1 : ns) t

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
```

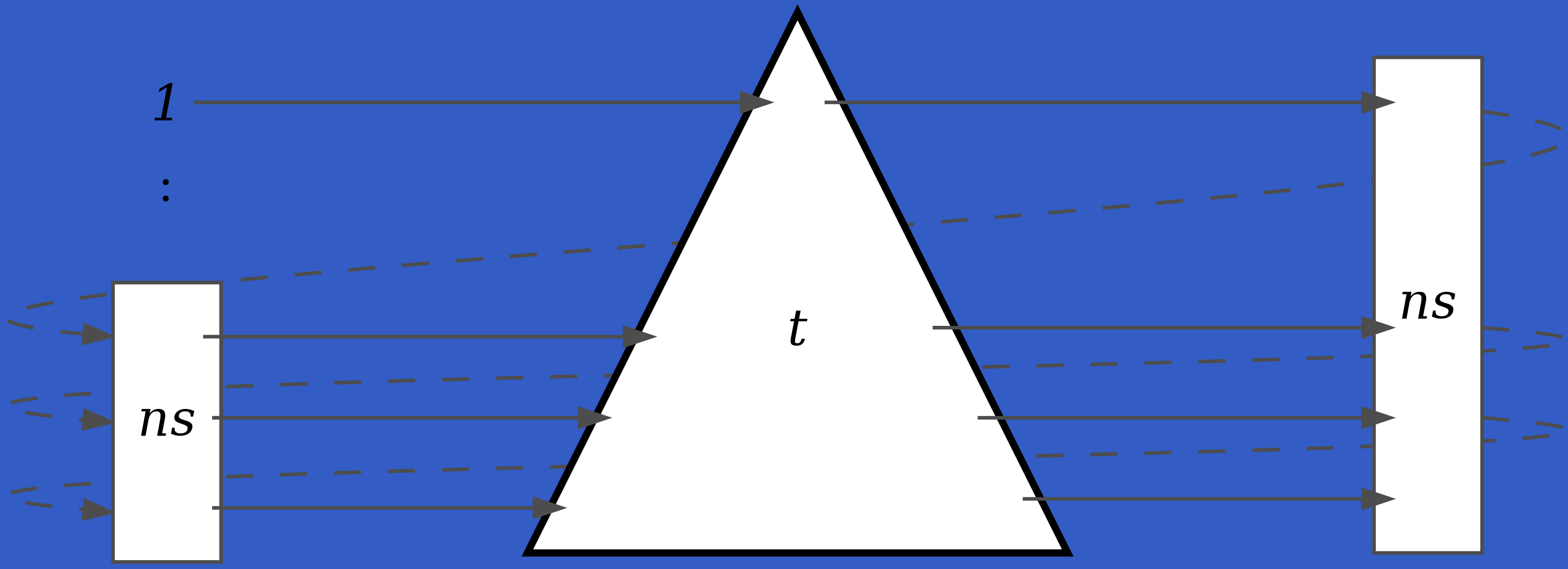
Eqns (1) & (2)

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.

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

The Triangulation Problem (1)

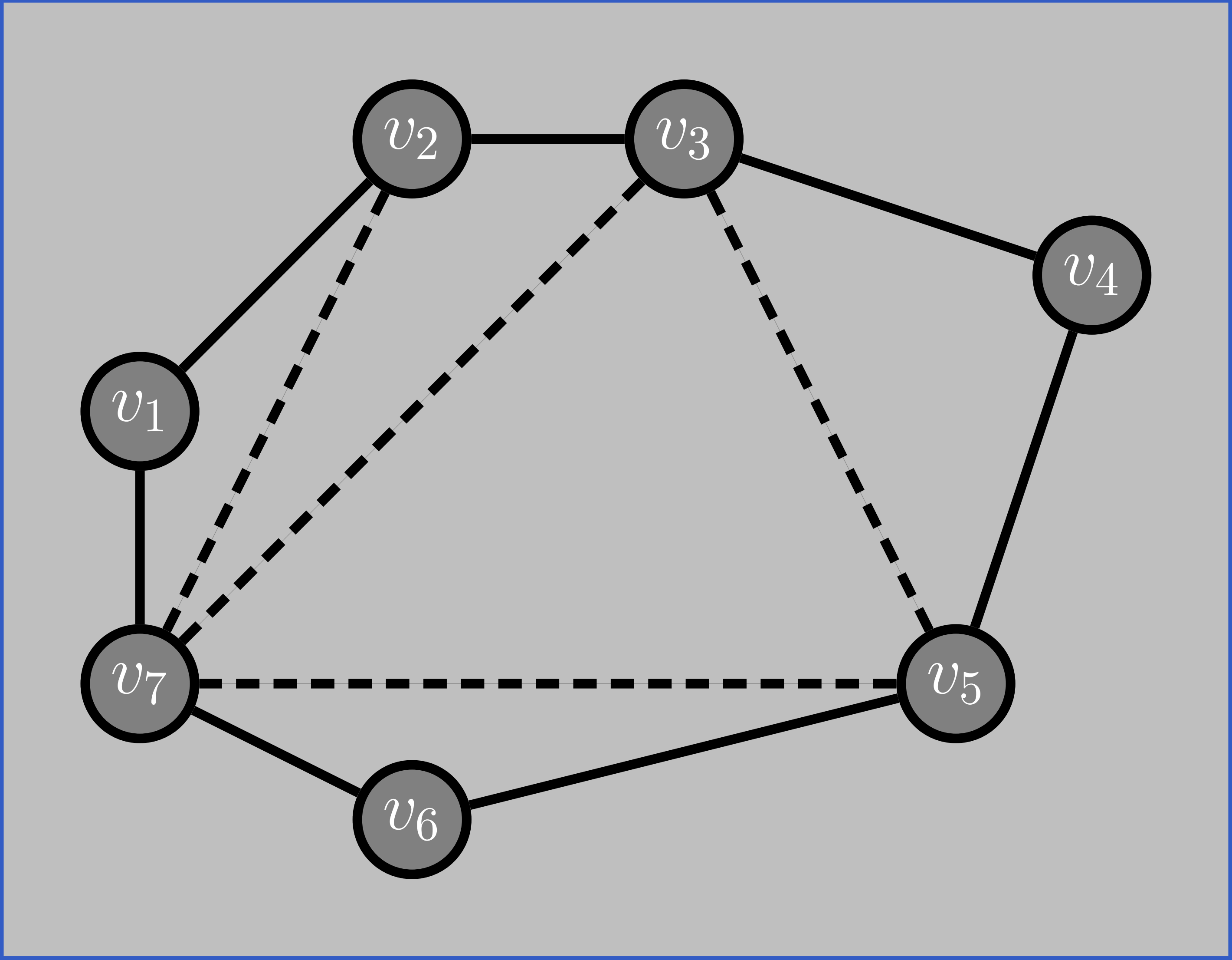
Select a set of ***chords*** that divides a convex polygon into triangles such that:

- 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.

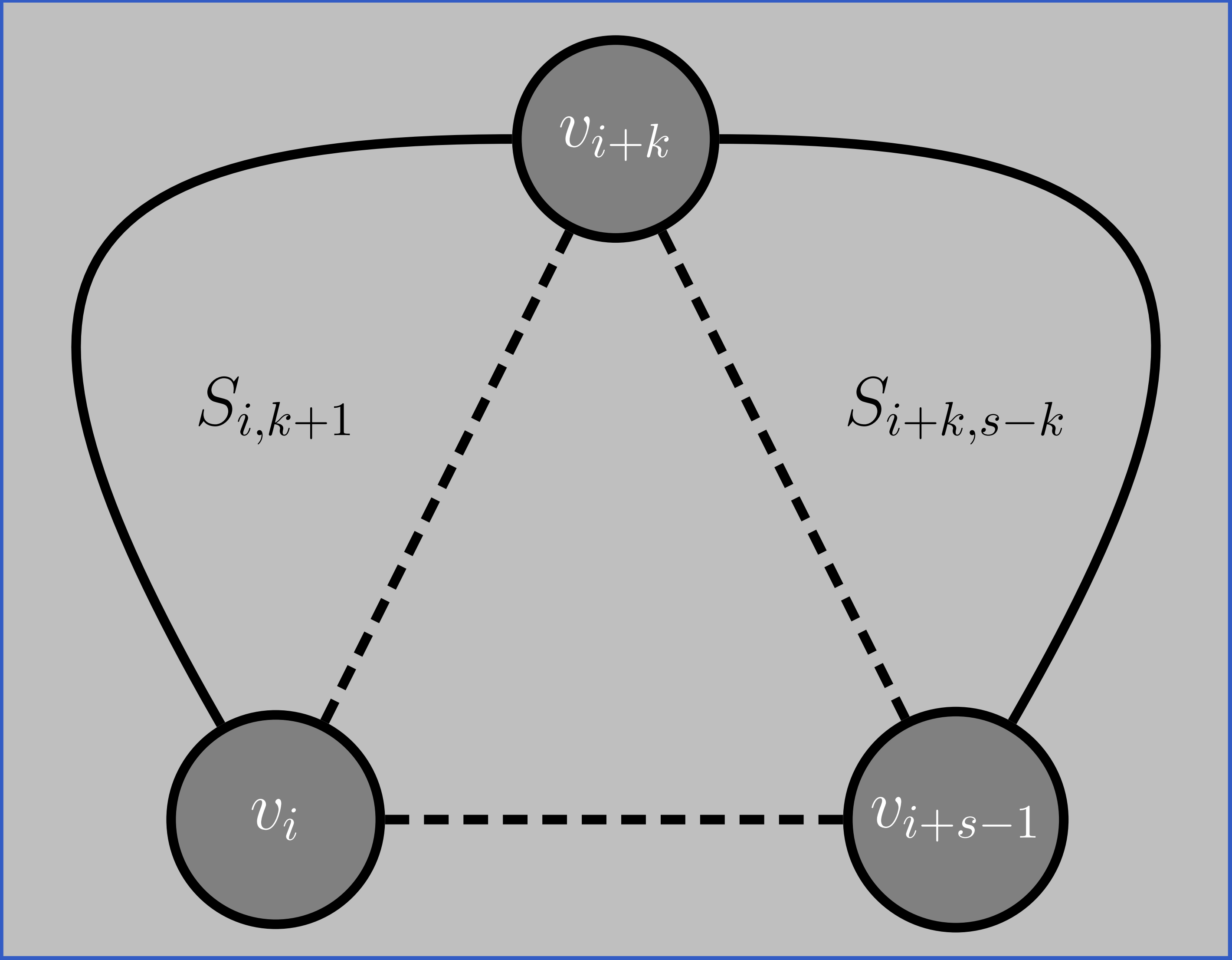
The Triangulation Problem (2)



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 $s \geq 4$:

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

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

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.