FUNCTIONAL PROGRAMMING
NO.5 LAZY EVALUATION

Tatsuya Hagino
hagino@sfc.keio.ac.jp

lecture slide URL

https://vu5.sfc.keio.ac.jp/slide/

Evaluation in Haskell

- Evaluation
  - Haskell evaluates the given expressions.
    - equals to program execution in usual programming languages.
  - Evaluation is to calculate the value according to the rule.

- Rewrite Model
  - Evaluation is done by rewriting a function application by its definition.

```
square n = n * n
```

function name

definition

```
square (1+3)
```

```
(1+3)*(1+3)
```

```
4*4
```

```
16
```

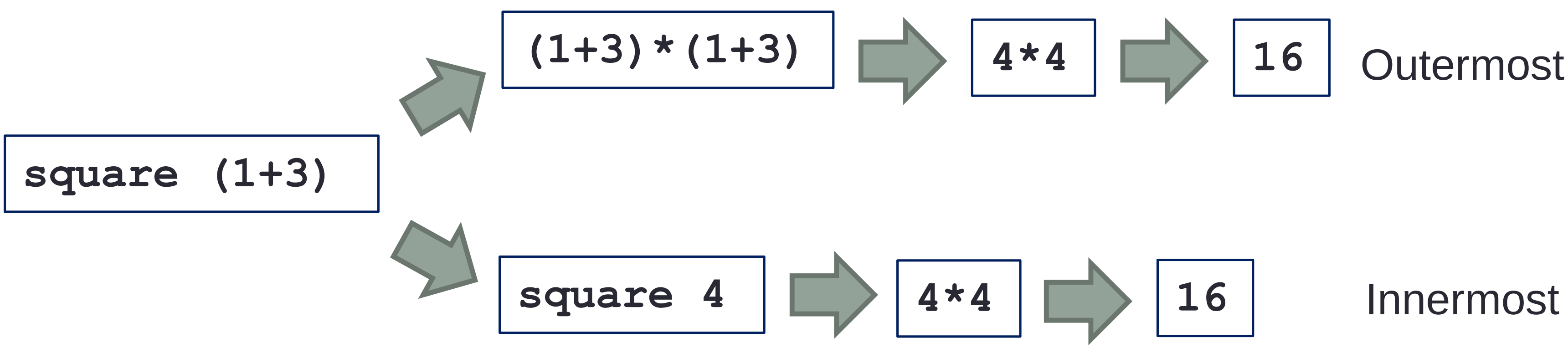
rewrite with definition

simplification

simplification

Innermost and Outermost simplification

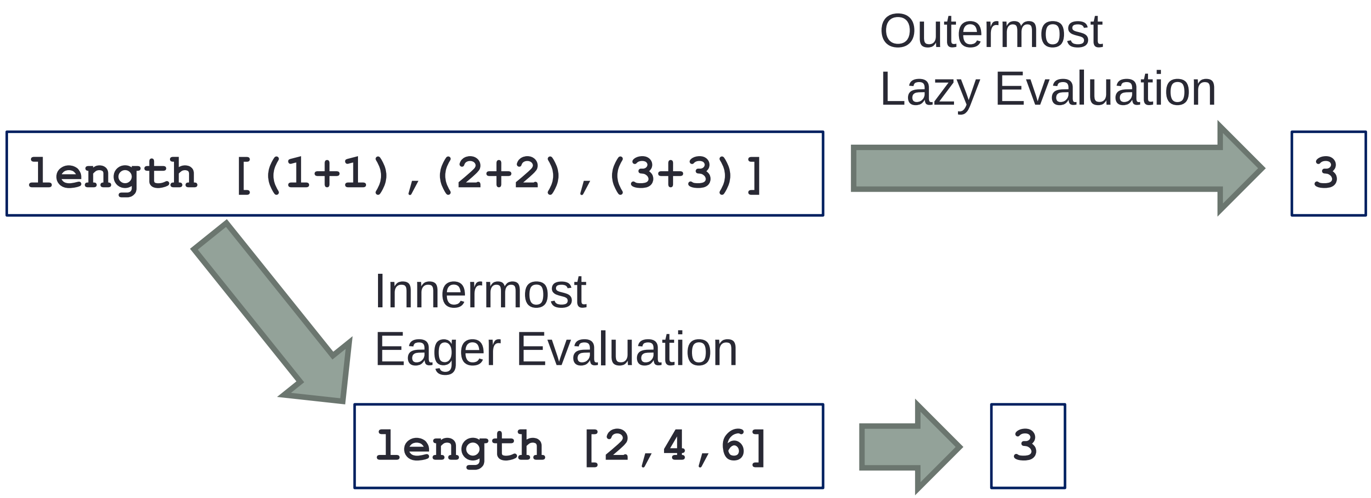
- Innermost simplification
  - Simplify inner expressions first
  - Simplify function arguments first, then expand the definition
- Outermost simplification
  - Simplify outer expressions first
  - Expand the definition without simplifying arguments



Outermost Simplification and Lazy Evaluation

- Haskell evaluates outermost expressions first
  - Delays full evaluation
  - lazy evaluation

- Example of lazy evaluation



Example of Lazy Evaluation

```
myIf :: Bool -> a -> a -> a
myIf True  t e = t
myIf False t e = e

f n = myIf (n==5) (1+2) (8/0)
```

```
f (4+1)
myIf ((4+1)==5) (1+2) (8/0)
myIf (5==5) (1+2) (8/0)
myIf True (1+2) (8/0)
(1+2)
3
```

Merit and Demerit of Lazy Evaluation

- Merit
  - Reduce unnecessary computation
  - Can handle infinite list
    - `ints n = n : (ints (n + 1))`
  - Give uniform interface
    - Even if a program is written: tree structure → list → process
    - List is not created as a whole
    - List processing can uniformly handle processing general structure.
    - List processing give the powerful tool.

- Demerit
  - Evaluation order is difficult to control.
  - Difficult to debug
    - No stack trace like C

Exercise 5-1

- Write tarai function in Haskell and C, and compare the execution time.

tarai.hs

```
main = print (tarai 20 10 5)

tarai :: Int -> Int -> Int -> Int
tarai x y z = if x <= y then y
                        else tarai (tarai (x-1) y z)
                                   (tarai (y-1) z x)
                                   (tarai (z-1) x y)
```

tarai.c

```
#include <stdio.h>

int tarai(int x, int y, int z) {
  if (x <= y) return y;
  else return tarai(tarai(x-1,y,z),tarai(y-1,z,x),tarai(z-1,x,y));
}

main() {
  printf("%d\n", tarai(20,10,5));
}
```

Exercise 5-2

- Create an infinite sequence 1,2,3,4,5,6,7,8,9,10,.....
- Print out first 20 elements of the sequence.

ints.hs

```
main = print $ take 20 $ ints 1

ints n = n:(ints(n+1))
```

- **`ints 1`** can be written as **`[1..]`**

ints2.hs

```
main = print $ take 20 [1..]
```

Exercise 5-3

- Create an infinite sequence of odd numbers
  - 1, 3, 5, 7, 9, 11, ...
- Print out first 20 elements of the sequence.

odds.hs

```
main = print $ take 20 $ odds 1

odds n =
```

- Write the same program by filtering odd numbers from the infinite sequence `[1..]`

odds2.hs

```
main = print $ take 20 $ filter ...
```

Recursive Call

- Functional programming language do not have while or if statements to loop. Instead, it uses recursive calls.

```
tarai :: Int -> Int -> Int -> Int
tarai x y z = if x <= y then y
                        else tarai (tarai (x-1) y z)
                                   (tarai (y-1) z x)
                                   (tarai (z-1) x y)
```

recursive all of tarai

```
ints n = n:(ints(n+1))
```

recursive all of ints

- Recursive call
  - Call a function itself directly or indirectly.

Exercise 5-4

fact.hs

```
import System.Environment

main = do args <- getArgs
          print $ fact $ read $ head args

fact 0 = 1
fact n = ...
```

- The factorial of n is the result of multiplying numbers from 1 to n.
  - $1! = 1$
  - $2! = 1 \times 2$
  - $3! = 1 \times 2 \times 3$
  - $4! = 1 \times 2 \times 3 \times 4$
  - $5! = 1 \times 2 \times 3 \times 4 \times 5$

- The factorial can be defined recursively.
  - $5! = 5 \times 4!$
  - $4! = 4 \times 3!$

- Please complete the above program of calculating the factorial of a give number.

Exercise 5-5

- Fibonacci sequence f_0, f_1, f_2, f_3, ... is
  - $f_0 = 1$
  - $f_1 = 1$
  - $f_n = f_{n-1} + f_{n-2}$
- Print out the first 20 elements of the Fibonacci sequence.

fib.hs

```
main = print $ take 20 $ map fib [1..]

fib 0 = 1
fib 1 = 1
fib n =
```

- Try output first 100 elements and see what happens.

Exercise 5-6

- Output all the divisors of a given number.

factor.hs

```
import System.Environment

main = do args <- getArgs
          print $ factors $ read $ head args

factors n = filter divisible [1..n]
  where divisible m = ...
```

- To calculate the remainder:
  - `x `mod` y`

```
% ./factor 144
[1,2,3,4,6,8,9,12,16,18,24,36,48,72,144]
```

Exercise 5-7

- p is a prime number when p is only divisible by 1 and p itself (i.e. only two divisors).
- Print out the first 100 prime numbers.

prime.hs

```
main = print $ take 100 $ filter isprime [1..]

factors n = ...

isprime n = ...
```

```
% ./prime
[2,3,5,7,...,541]
```

Exercise 5-8

- There are prime numbers which are twins.
  - 3 and 5 are prime numbers and the gap is 2.
  - 5 and 7 are also twins.
  - 7 and 11 are not, since the gap is 4.
- Print out the first 20 twin prime number pairs.

twin.hs

```
main = print $ take 20 $ twin $ ...

isprime n = ...

twin :: [Int] -> [(Int,Int)]
twin ...
```

- Pairs can be created by:
  - `(x, y)`

```
% ./twin
[(3,5), (5,7), (11,13), (17,19),..., (311,313)]
```