

# Lazy evaluation illustrated

## for Haskell divers

*exploring some mental models and implementations*

Takenobu T.

Lazy,... zzz

..., It's fun!

## NOTE

- Meaning of terms are different by communities.
- There are a lot of good documents. Please see also references.
- This is written for GHC's Haskell.

# Contents

## 1. Introduction

- Basic mental models
- Lazy evaluation
- Simple questions

## 2. Expressions

- Expression and value
- Expressions in Haskell
- Classification by value and form
- WHNF

## 3. Internal representation of expressions

- Constructor
- Thunk
- Uniform representation
- let, case expression
- WHNF

## 4. Evaluation

- Evaluation strategies
- Evaluation in Haskell (GHC)
- Examples of evaluation steps
- Controlling the evaluation

## 5. Implementation of evaluator

- Lazy graph reduction
- STG-machine

## 6. Semantics

- Bottom
- Non-strict Semantics
- Strict analysis

## 7. Appendix

- References


# 1. Introduction

# 1. Introduction

Basic mental models

# How to evaluate a program in your brain ?

a program



```
code  
code  
code  
:
```

How to evaluate (execute, reduce) the program in your brain?

What "mental model" do you have?

# One of the mental models for C program

## C program

A sequence of statements

```
main (...) {  
  code..  
  code..  
  code..  
  code..  
}
```

A red bracket groups the four 'code..' lines, with a red question mark to its right.

A nested structure

```
x = func1( func2( a ) );
```

Red underlines are under 'func1' and 'func2( a )', with a red question mark below the space between them.

A sequence of arguments

```
y = func1( a(x), b(x), c(x) );
```

Red underlines are under 'a(x)', 'b(x)', and 'c(x)', with a red question mark below the space between 'a(x)' and 'b(x)'.

A function and arguments

```
z = func1( m + n );
```

Red underlines are under 'func1' and 'm + n', with a red question mark below the space between them.

How to evaluate (execute, reduce) the program in your brain?

What step, what order, ... ?



# One of the mental models for C program

C program

A program is a collection of statements.

A sequence of statements

```
main (...) {  
  code..  
  code..  
  code..  
  code..  
}
```

Statements are  
executed downward.

A nested structure

```
x = func1( func2( a ) );
```

from inner to outer

A sequence of arguments

```
y = func1( a(x), b(x), c(x) );
```

from left to right

A function and arguments

```
z = func1( m + n );
```

arguments first  
apply second

Each programmers have some mental models in their brain.

# One of the mental models for C program

Maybe, You have some implicit mental model in your brain for C program.

(1) A program is **a collection of statements**.

(2) There is **an order** between evaluations of elements.



(3) There is **an order** between completion and start of evaluations.



This is an example of an implicit **sequential order** model for programming languages.

# One of the mental models for Haskell program

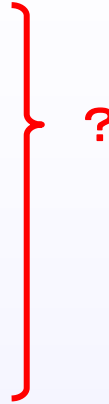
Haskell program

```
main = exp11 (exp12 exp13 exp14 )
```

```
exp13 = exp131 exp132
```

```
exp14 = exp141 exp142 exp143
```

```
:
```



How to evaluate (execute, reduce) the program in your brain?

What step, what order, ... ?

# One of the mental models for Haskell program

Haskell program

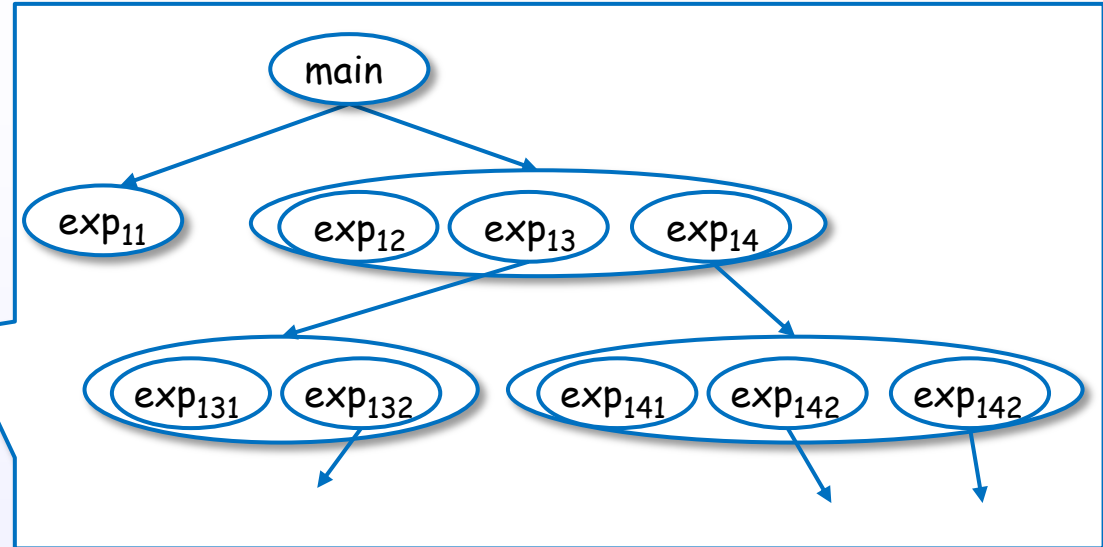
A program is a collection of expressions.

```
main = exp11 (exp12 exp13 exp14)
```

```
exp13 = exp131 exp132
```

```
exp14 = exp141 exp142 exp143
```

```
:
```



```
main = exp11 (exp12 (exp131 exp132) (exp141 exp142 exp143) )
```

A entire program is regarded as a single expression.

The subexpression is evaluated (reduced) in some order.

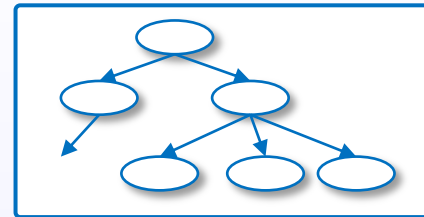
The evaluation is performed by replacement.

# One of the mental models for Haskell program

(1) A program is **a collection of expressions**.

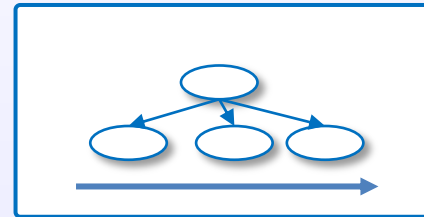
(2) A entire program is regarded as **a single expression**.

```
main = e (e (e (e e) e (e e e) ) )
```

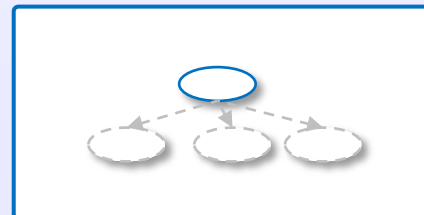
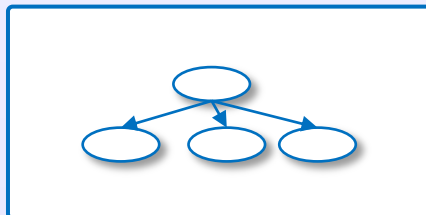


(3) The subexpressions are evaluated (reduced) in **some order**.

```
f = e (e (e (e e) e (e e e) ) )
```



(4) The evaluation is performed by **replacement**.

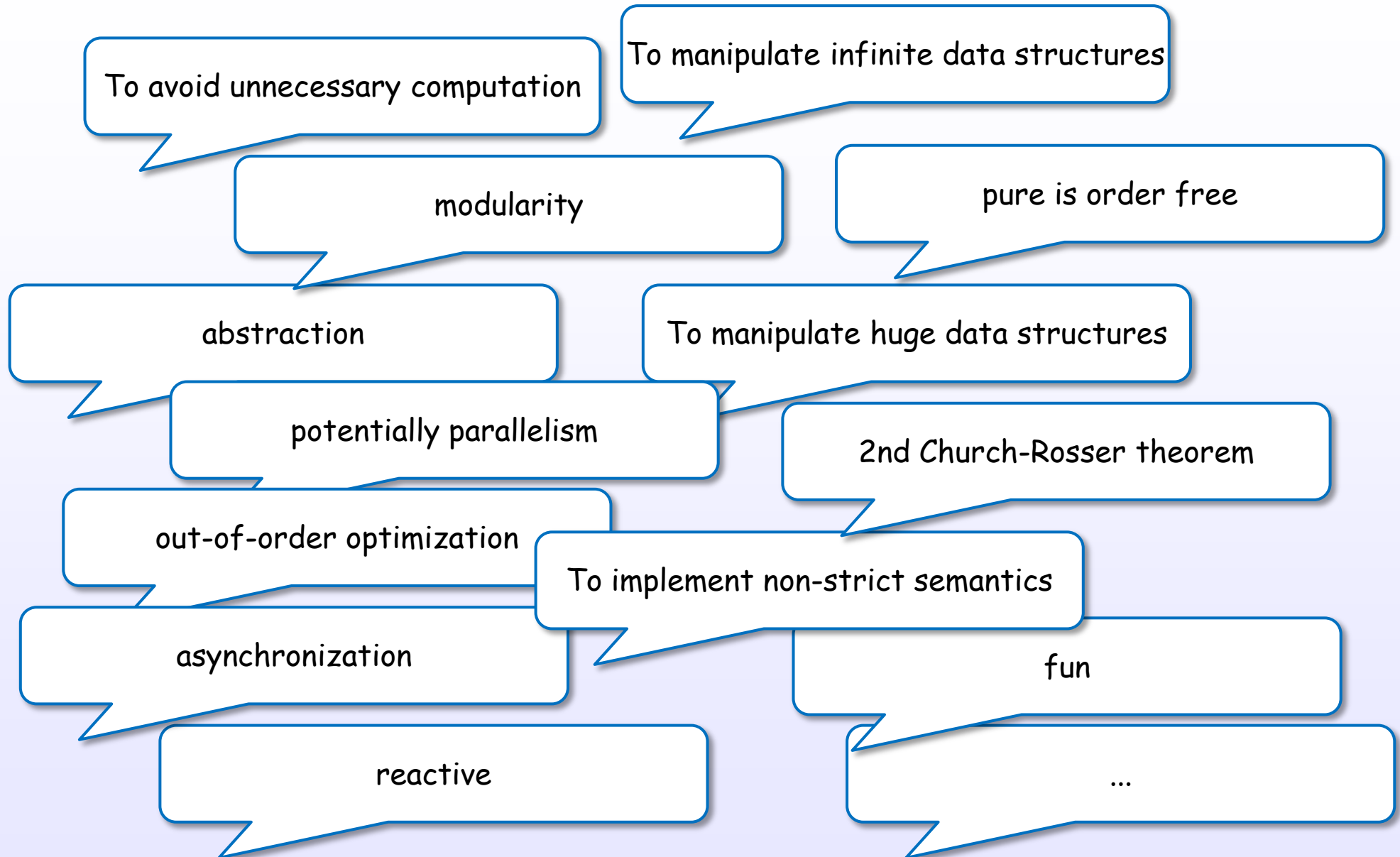


This is an example of an **expression reduction** model for Haskell.

# 1. Introduction

Lazy evaluation

# Why lazy evaluation?



There are various reason.

[slpj-book-1987]

# Haskell(GHC) 's lazy evaluation

evaluate **only if needed**

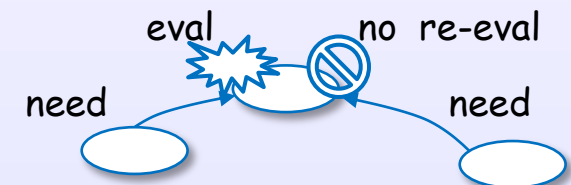
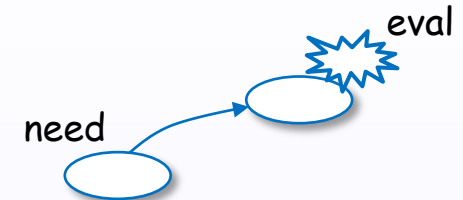
+

evaluate **only enough**

+

evaluate **at most once**

Lazy evaluation is this.



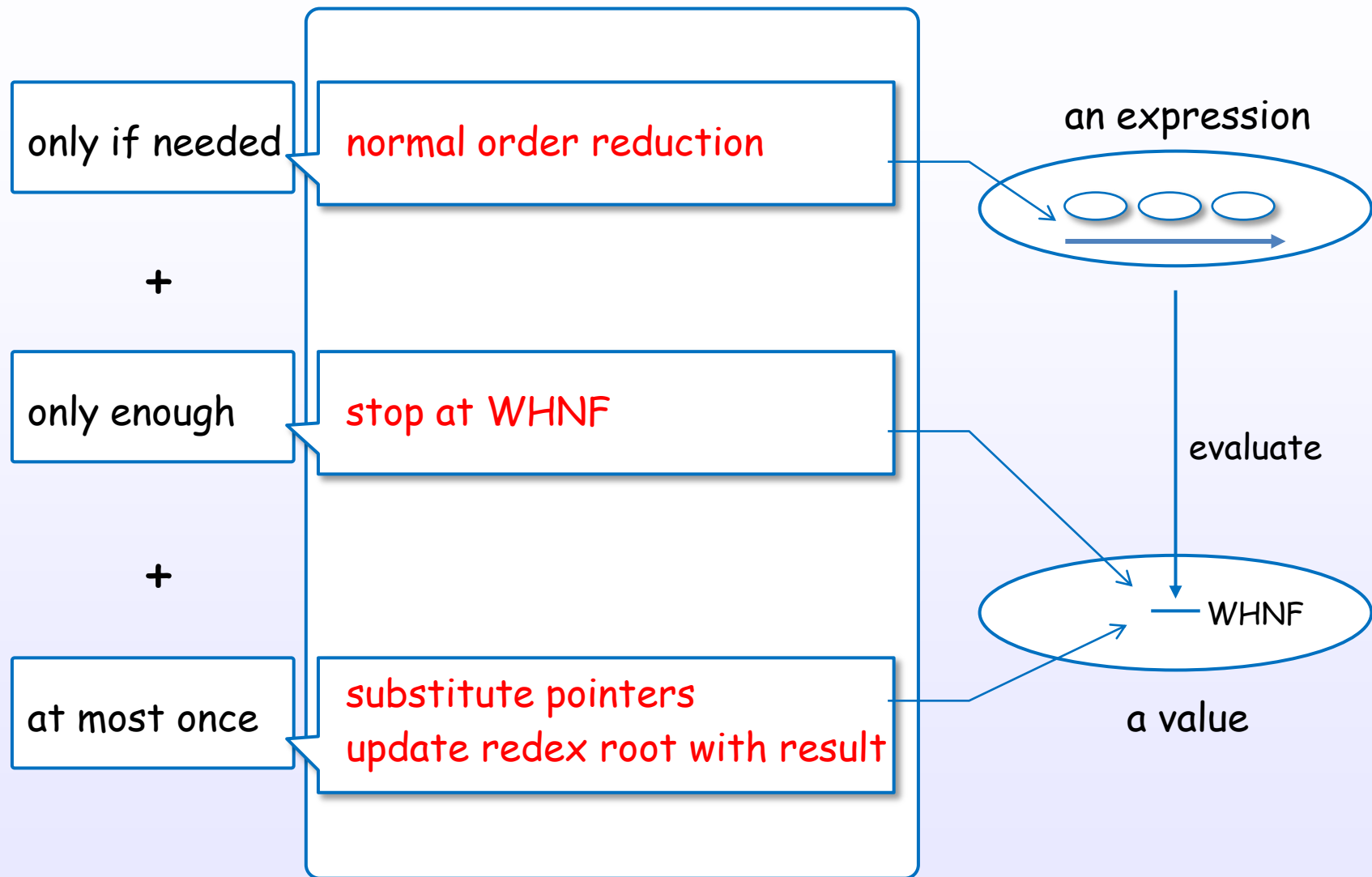
[slpj-book-1987], 194

[slpj-book-1987], Chap.12

[Bird, Chap.7]

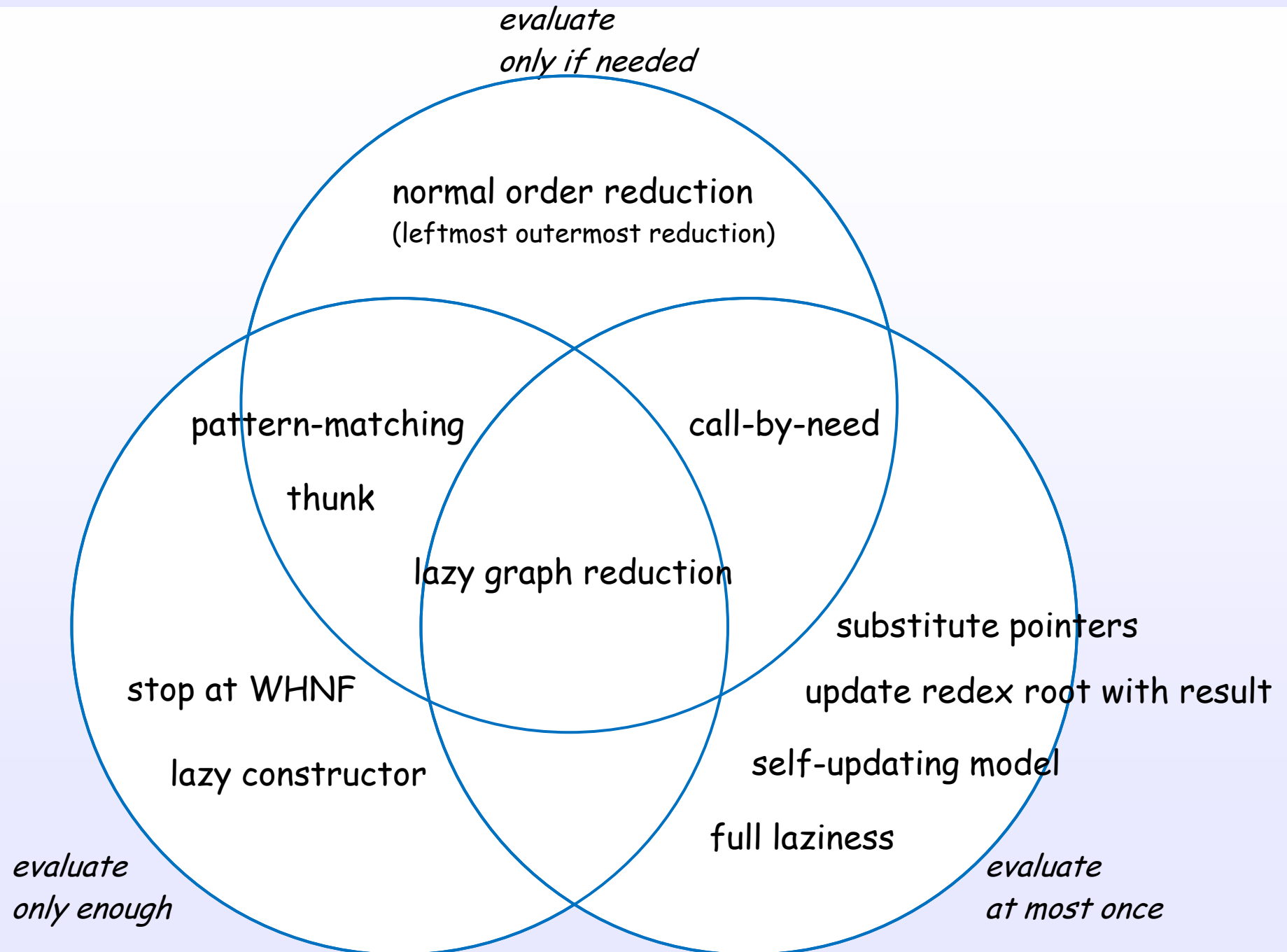


# Ingredient of Haskell(GHC) 's lazy evaluation



This strategy is implemented by lazy graph reduction.

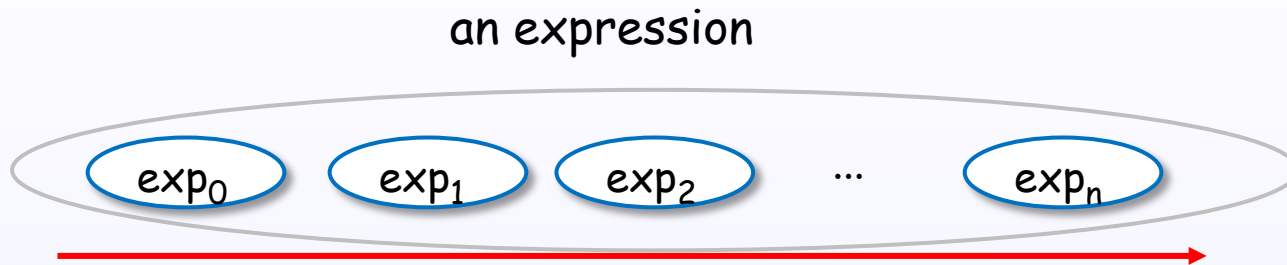
# Techniques of Haskell(GHC) 's lazy evaluation



# 1. Introduction

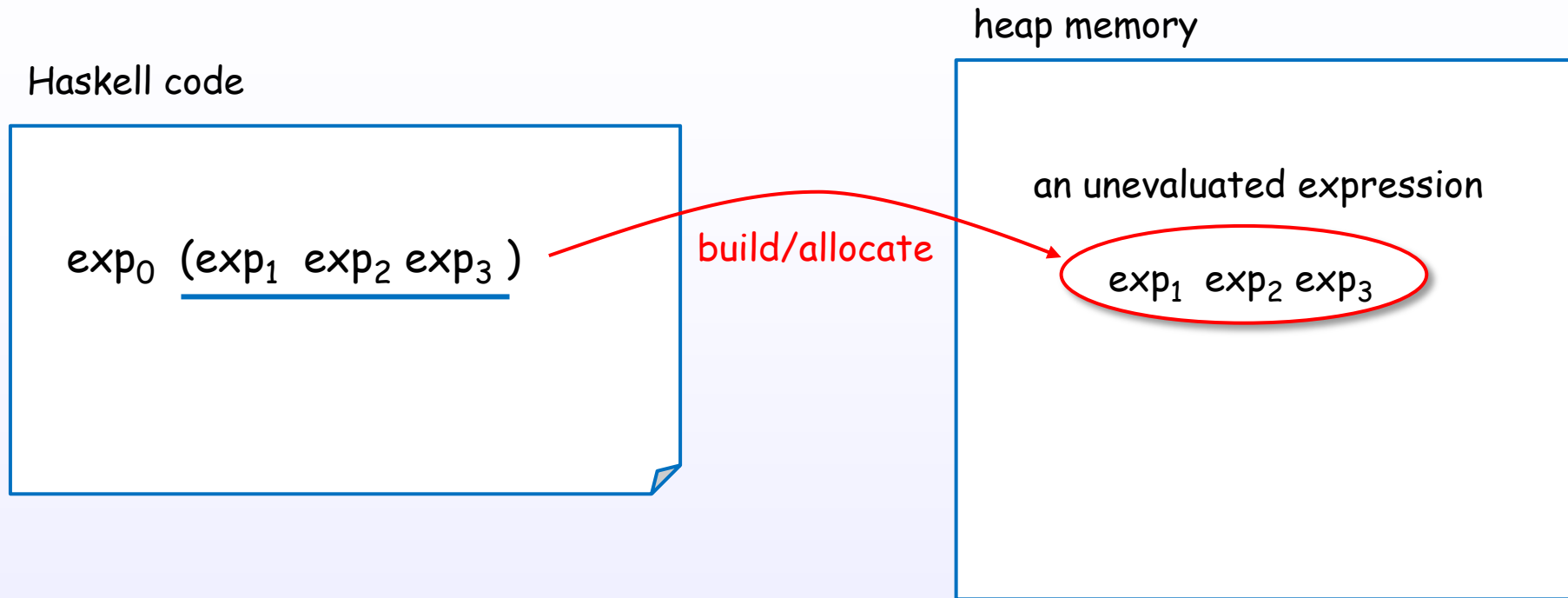
Simple questions

# What order?



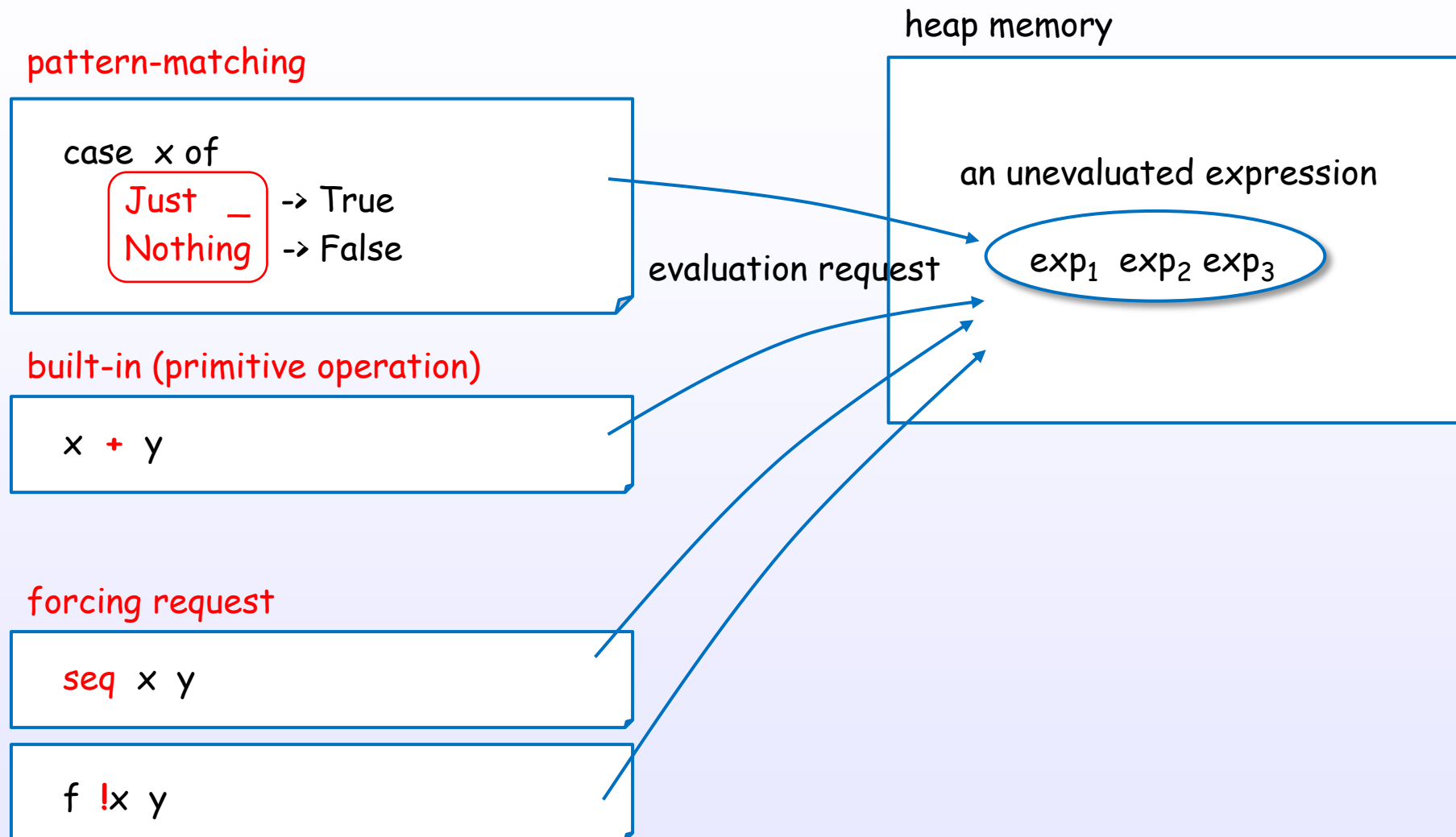
An expression is evaluated by normal order (leftmost outermost redex first).

# How to postpone?



To postpone the evaluation, an unevaluated expression is built in heap memory.

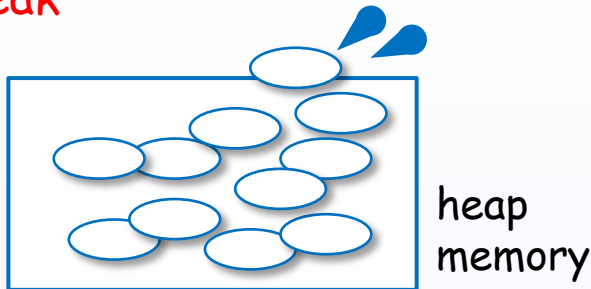
# When needed?



Pattern-matching or forcing request drive the evaluation.

# What are the pitfalls?

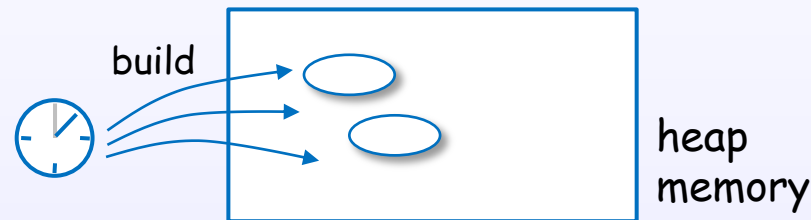
To consider hidden **space leak**



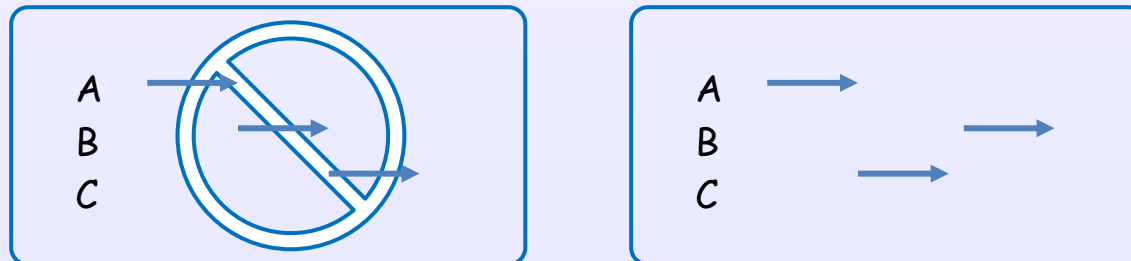
[hack.hands]

[CIS194]

To consider **performance cost** to postpone unevaluated expressions



To consider evaluation (execution) **order** in real world



You can avoid the pitfalls by controlling the evaluation.

## 2. Expressions

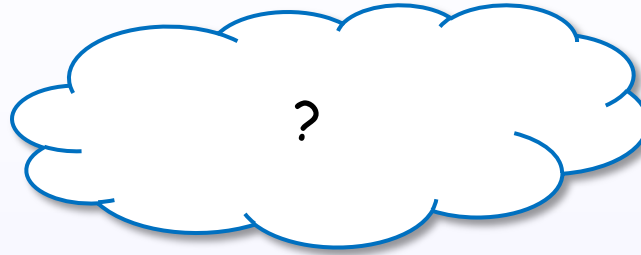


## 2. Expressions

Expression and value

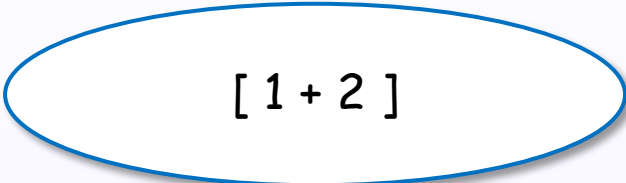
# What is an expression?

An expression



# An expression denotes a value

An expression



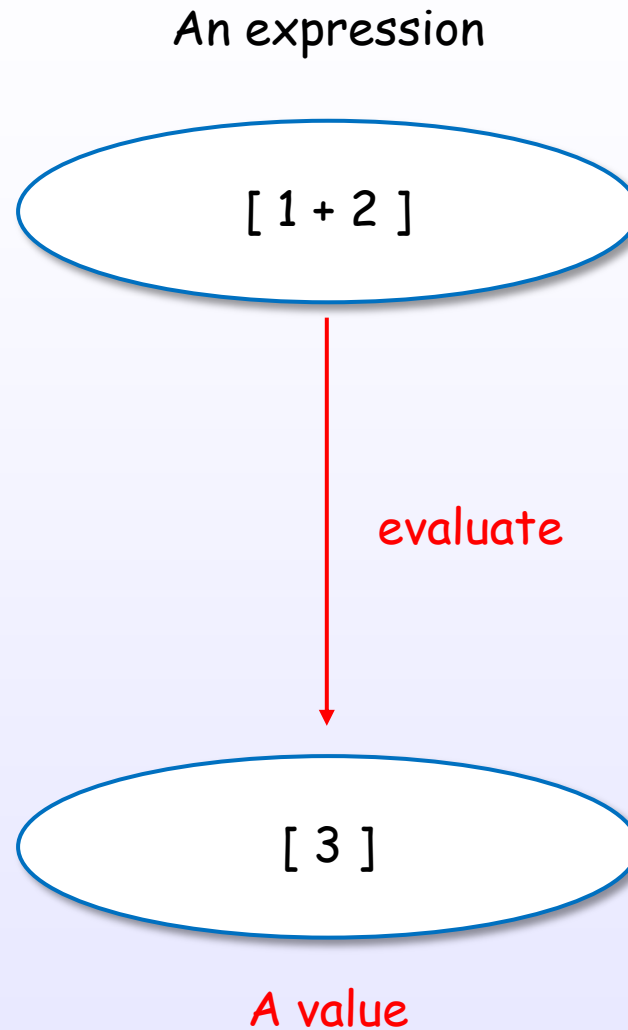
[ 1 + 2 ]

[HR2010]

[Bird, Chapter 2]

References : [1]

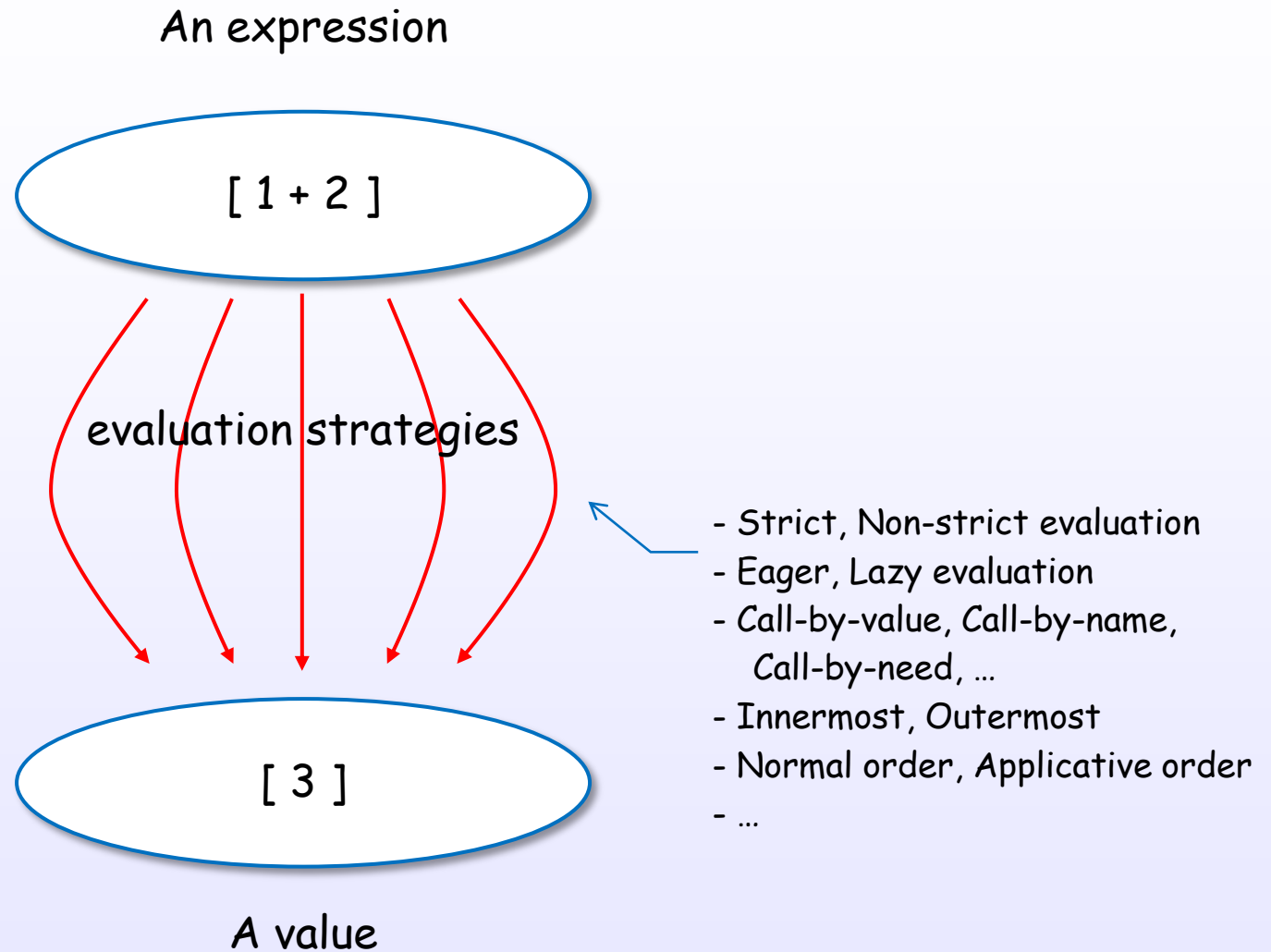
# An expression evaluates to a value



[HR2010]

[Bird, Chapter 2]

# There are many evaluation approaches

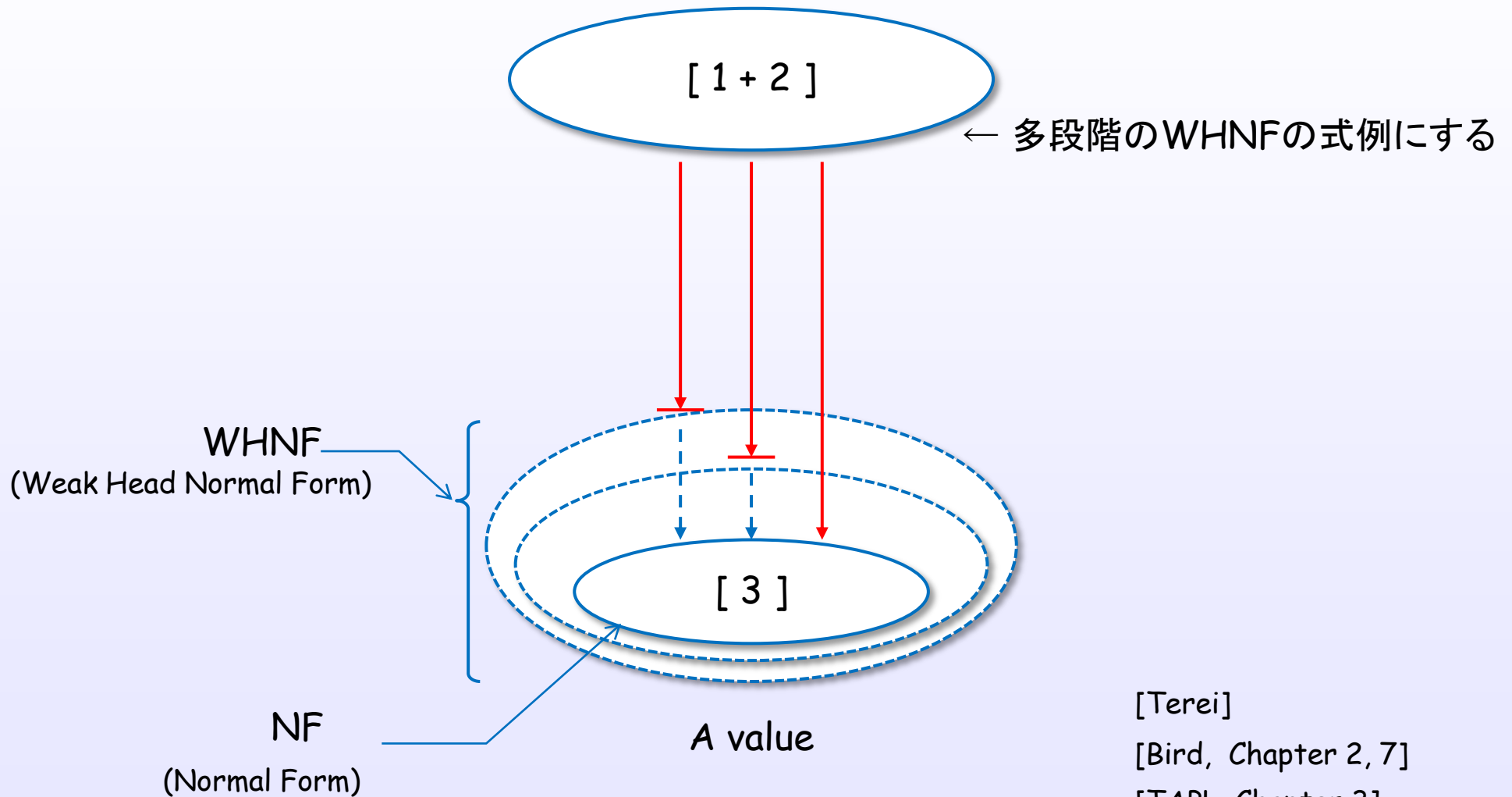


[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

## There are some evaluation levels

## An expression



[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

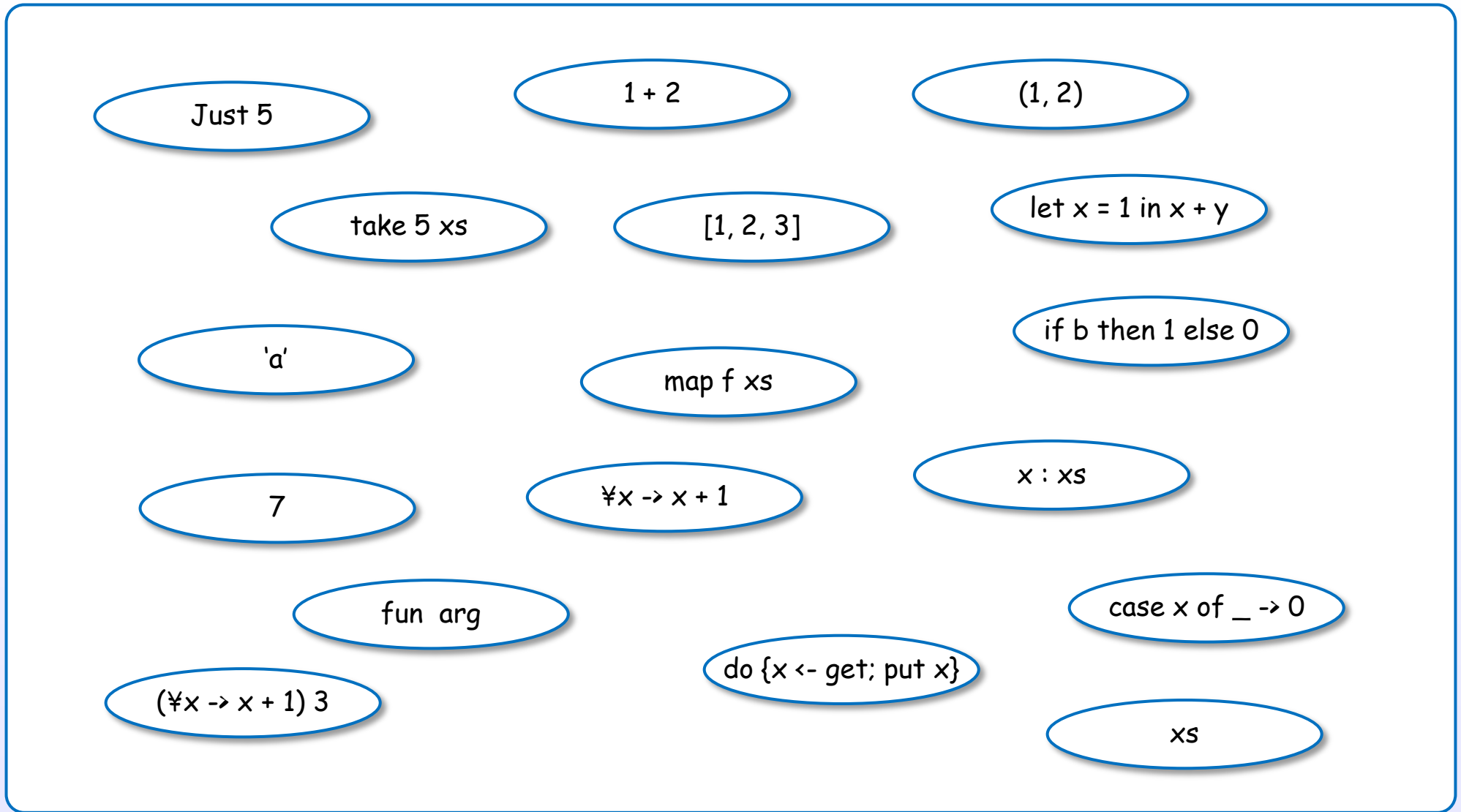
References : [1]

## 2. Expressions

### Expressions in Haskell

# There are many expressions in Haskell

## Expressions



categorizing

[HR2010]

[Bird, Chapter 2]

References : [1]



# Expression categories in Haskell

WHNF(a value)、  
unevaluated expression  
との関連づけを  
PAPもWHNFなので注意

## lambda abstraction

$\forall x \rightarrow x + 1$

## let expression

let  $x = 1$  in  $x + y$

## variable

$xs$

## conditional

if  $b$  then 1 else 0

## case expression

case  $x$  of  $\_ \rightarrow 0$

## do expression

do { $x \leftarrow \text{get}$ ; put  $x$ }

## general constructor, literal and some forms

7

[1, 2, 3]

(1, 2)

'a'

$x : xs$

Just 5

## function application

take 5  $xs$

$(\forall x \rightarrow x + 1)$  3

1 + 2

map  $f$   $xs$

fun arg

[HR2010]  
[Bird, Chapter 2]

# Specification is defined in Haskell 2010 Language Report

## Haskell 2010 Language Report, Chapter 3 Expressions [1]

<i>exp</i>	→	<i>infixexp</i> :: [context =>] type   <i>infixexp</i>	(expression type signature)
<i>infixexp</i>	→	<i>lexp</i> <i>qop</i> <i>infixexp</i>   - <i>infixexp</i>   <i>lexp</i>	(infix operator application) (prefix negation)
<i>lexp</i>	→	\ <i>apat</i> <sub>1</sub> ... <i>apat</i> <sub><i>n</i></sub> -> <i>exp</i>   let <i>decls</i> in <i>exp</i>   if <i>exp</i> [ <i>i</i> ] then <i>exp</i> [ <i>i</i> ] else <i>exp</i>   case <i>exp</i> of { <i>alts</i> }   do { <i>stmts</i> }   <i>fexp</i>	(lambda abstraction, $n \geq 1$ ) (let expression) (conditional) (case expression) (do expression)
<i>fexp</i>	→	[ <i>fexp</i> ] <i>aexp</i>	(function application)
<i>aexp</i>	→	<i>qvar</i>   <i>gcon</i>   <i>literal</i>   ( <i>exp</i> )   ( <i>exp</i> <sub>1</sub> , ... , <i>exp</i> <sub><i>k</i></sub> )   [ <i>exp</i> <sub>1</sub> , ... , <i>exp</i> <sub><i>k</i></sub> ]   [ <i>exp</i> <sub>1</sub> [, <i>exp</i> <sub>2</sub> ] .. [ <i>exp</i> <sub>3</sub> ] ]   [ <i>exp</i>   <i>qual</i> <sub>1</sub> , ... , <i>qual</i> <sub><i>n</i></sub> ]   ( <i>infixexp</i> <i>qop</i> )   ( <i>qop</i> _{-} <i>infixexp</i> )   <i>qcon</i> { <i>fbind</i> <sub>1</sub> , ... , <i>fbind</i> <sub><i>n</i></sub> }   <i>aexp</i> _{ <i>qcon</i> } { <i>fbind</i> <sub>1</sub> , ... , <i>fbind</i> <sub><i>n</i></sub> }	(variable) (general constructor)  (parenthesized expression) (tuple, $k \geq 2$ ) (list, $k \geq 1$ ) (arithmetic sequence) (list comprehension, $n \geq 1$ ) (left section) (right section)  (labeled construction, $n \geq 0$ ) (labeled update, $n \geq 1$ )

## 2. Expressions

Classification by value and form

# Classification by value

## Expressions

### unevaluated expressions

take 5 xs

$(\forall x \rightarrow x + 1) 3$

let x = 1 in x + y

1 + 2

map f xs

fun arg

if b then 1 else 0

case x of \_ -> 0

do {x <- get; put x}

### values

#### data values

7

'a'

[1, 2, 3]

(1, 2)

Just 5

Just (f x)

#### function values

$\forall x \rightarrow x + 1$

Values are data values or function values.

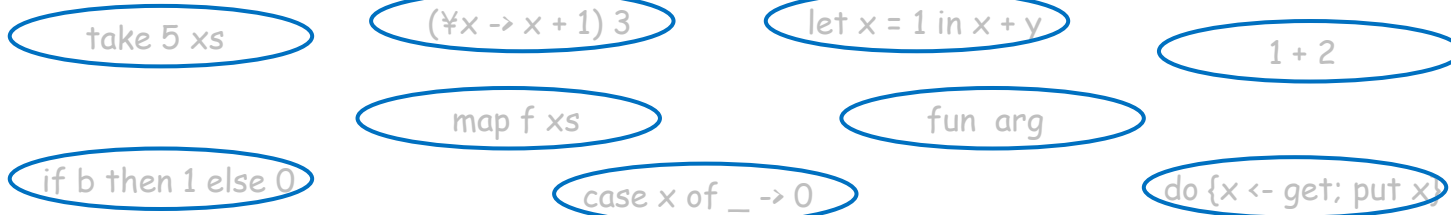
[STG]

References : [1]

# Classification by value

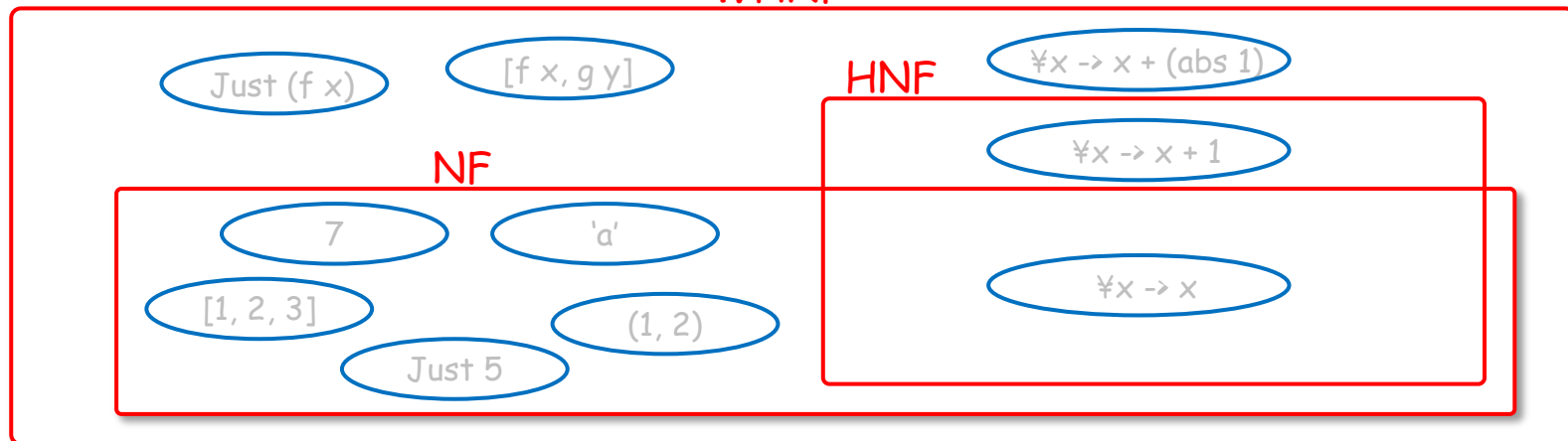
## Expressions

### unevaluated expressions



### values

#### WHNF



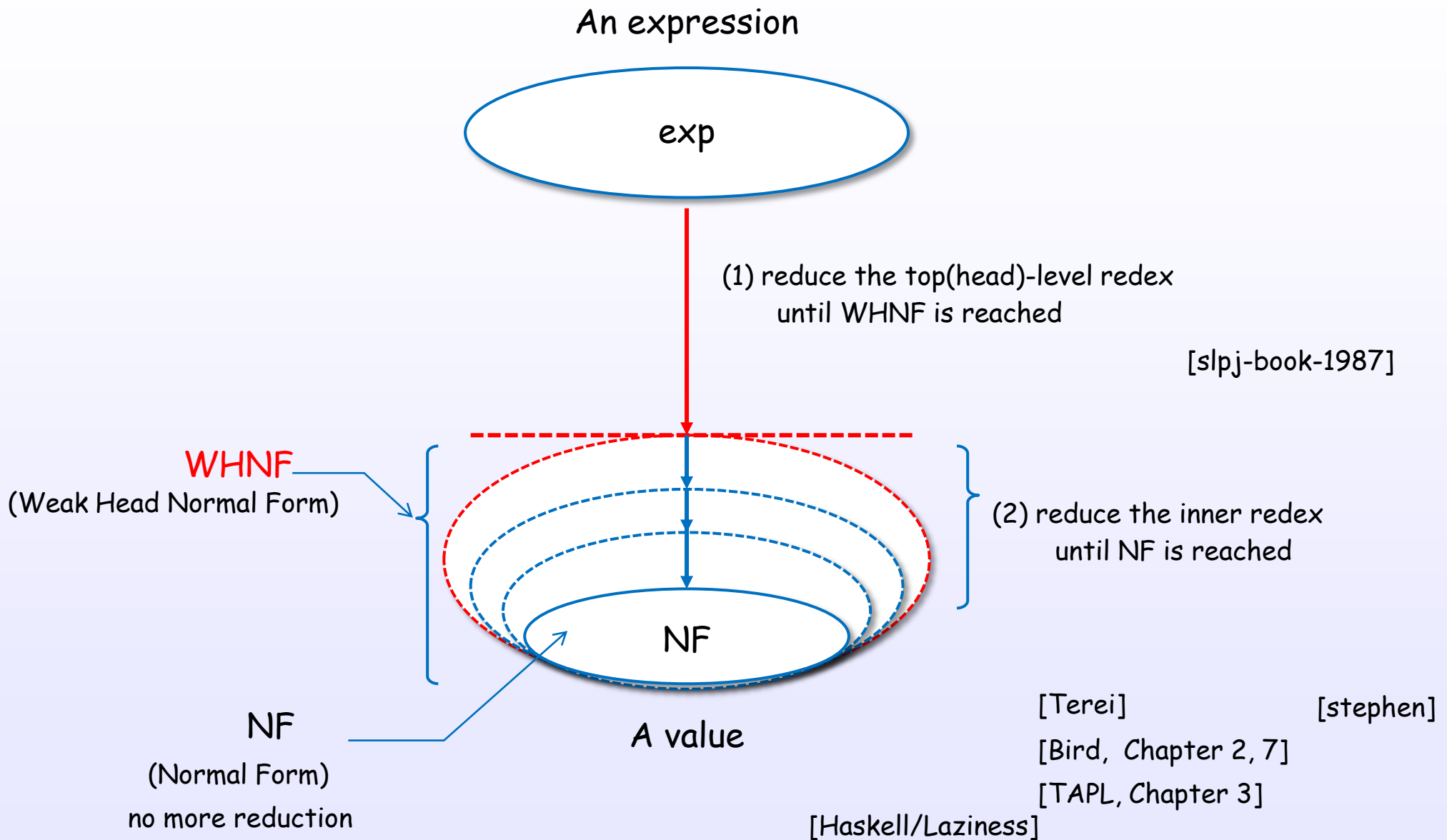
Values are NF, HNF or WHNF.

[STG]

## 2. Expressions

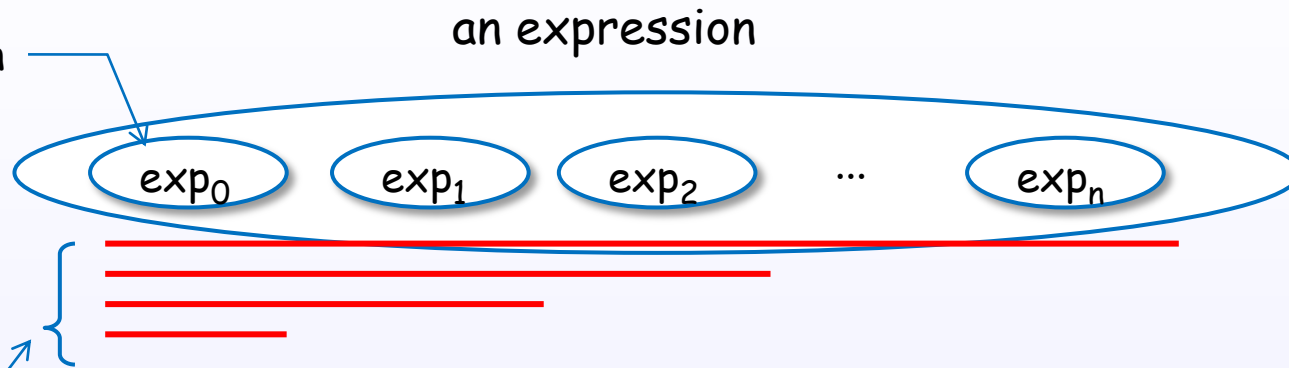
WHNF

# WHNF is one of the evaluated values.



# WHNF is a value which has evaluated top-level

top-level is  
constructor or  
lambda abstraction



no redex  
(can't evaluate anymore)

An expression has no top level redex, if it is in WHNF.

[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

[parconc, Ch.2] [stephen]  
[hack.hands]  
[slpj-book-1987]

[Terei]

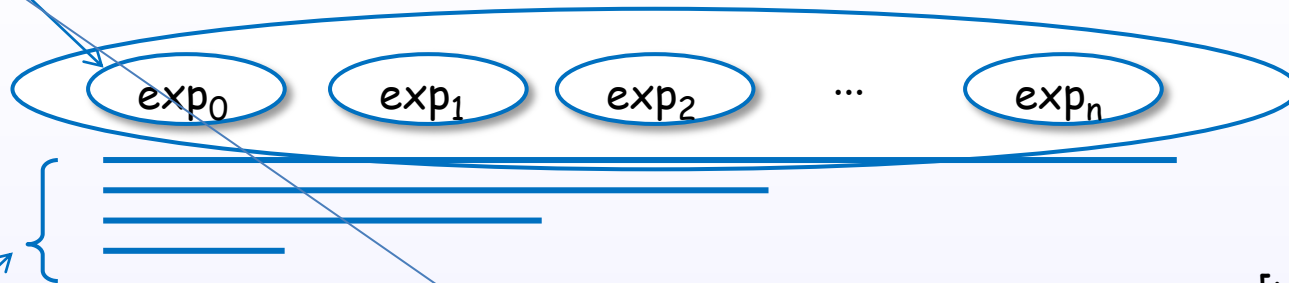
References : [1]



# WHNF

データ抽象、ビルトイン

an expression



[parconc, Ch.2]

An expression has no top level redex, if it is in WHNF.

[slpj-book-1987]

These are in weak head normal form,  
but not in normal form, since they contain inner redex. (p.198)

[stephen]  
[hack.hands]

[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

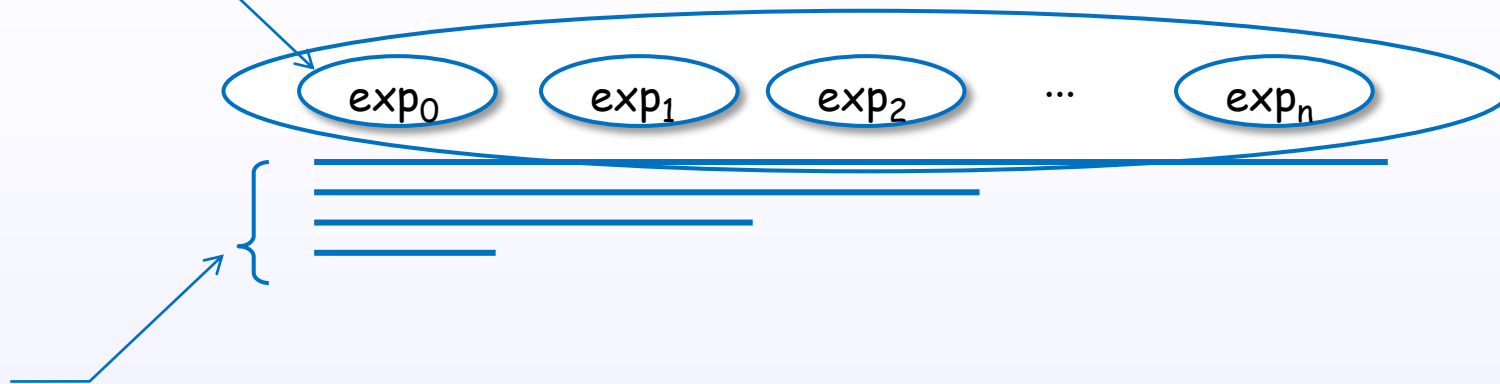
[Terei]

References : [1]

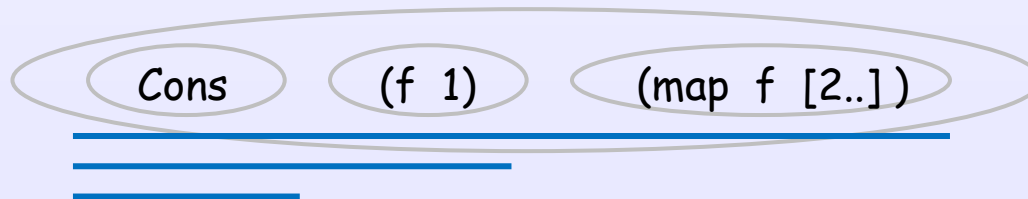
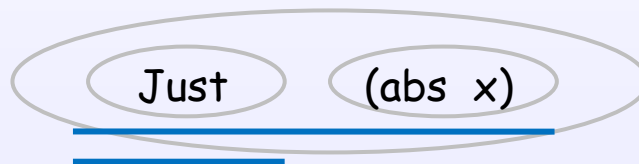
# Examples of WHNF

データ抽象、ビルトイン

an expression



more



[slpj-book-1987]

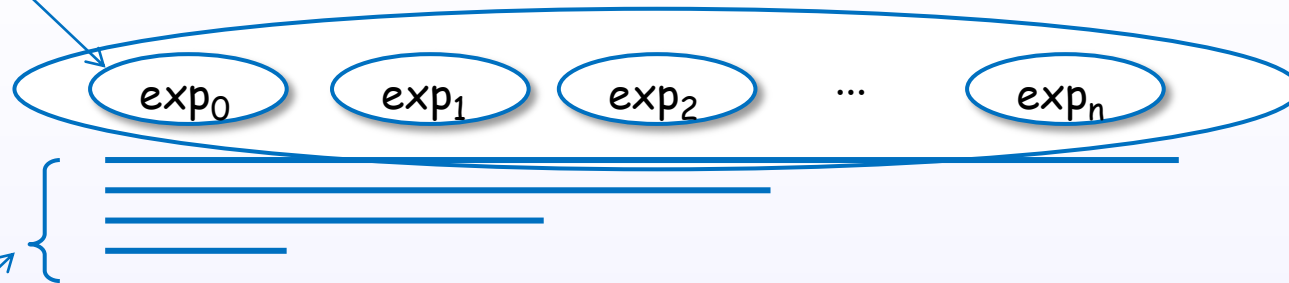
# HNF

データ抽象、ビルトイン

内側(body)が、簡

more

an expression



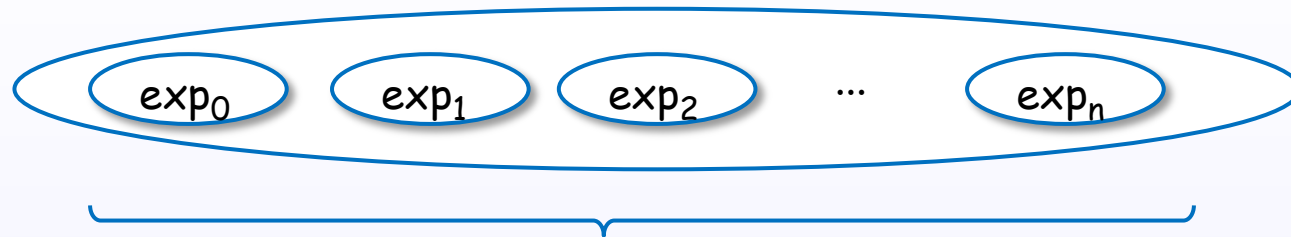
[slpj-book-1987]

[Terei]

References : [1]

# NF

an expression



redexが内部に無い

[slpj-book-1987]

[Terei]

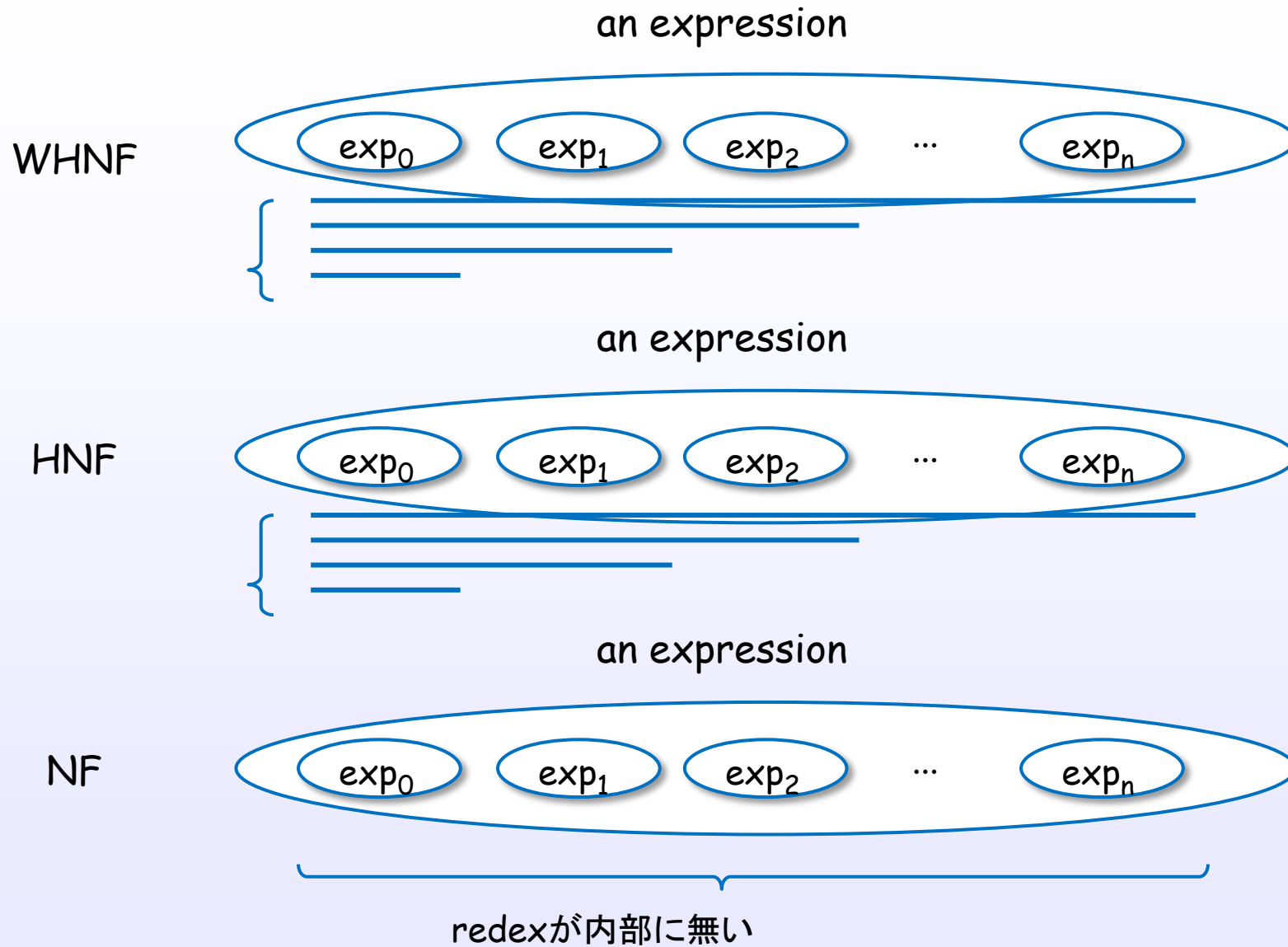
[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

[Terei]

References : [1]

# WHNF, HNF, NF



[slpj-book-1987]

References : [1]

## Definition of WHNF and HNF

## The implementation of functional programming languages [19]

### 11.3.1 Weak Head Normal Form

To express this idea precisely we need to introduce a new definition:

### DEFINITION

A lambda expression is in *weak head normal form* (WHNF) if and only if it is of the form

$$F \ E_1 \ E_2 \ \dots \ E_n$$

where  $n \geq 0$ ;

and either F is a variable or data object  
or F is a lambda abstraction or built-in function  
and  $(F \ E_1 \ E_2 \ \dots \ E_m)$  is not a redex for any  $m \leq n$ .

An expression has no *top-level redex* if and only if it is in weak head normal form.

### 11.3.3 Head Normal Form

Head normal form is often confusing and requires some discussion. The content of the `head` field is since for most purposes head normal form is sufficient. Nevertheless, we will stick to

### DEFINITION

A lambda expression is in *head normal form* (HNF) if and only if it is of the form

$$\lambda x_1. \lambda x_2. \dots \lambda x_n. (v \ M_1 \ M_2 \ \dots \ M_m)$$

where  $n, m \geq 0$ ;

**v** is a variable ( $x_i$ ), a data object, or a built-in function;

and  $(v \ M_1 \ M_2 \ \dots \ M_p)$  is not a redex for any  $p \leq m$ .

[slpj-book-1987]

### 3. Internal representation of expressions

### 3. Internal representation of expressions

Constructor



# A value or an unevaluated expression

## Expressions

unevaluated expressions

unevaluated expressions

values

data values

function values

値か否か。値は2種。

[STG]

# Constructor

Constructor is one of the key elements to understand WHNF and lazy evaluation in Haskell.

# Algebraic data type and value

data文で宣言する代数的データ型とその値

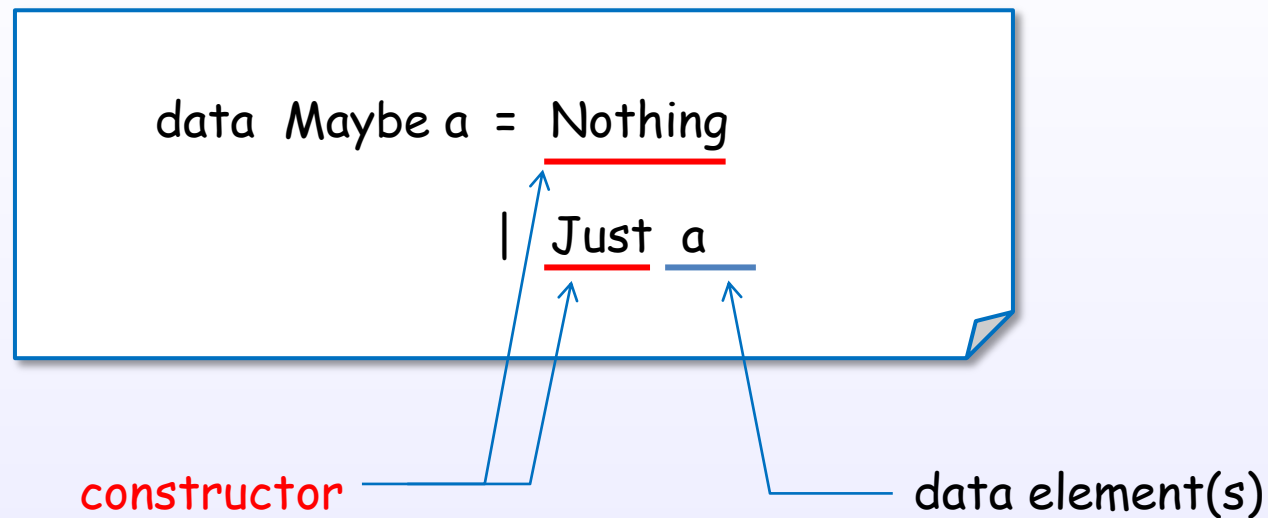
```
data Maybe a = Nothing  
              | Just a
```

Algebraic Data Type

Data **Values**

# Constructors are defined by data declaration

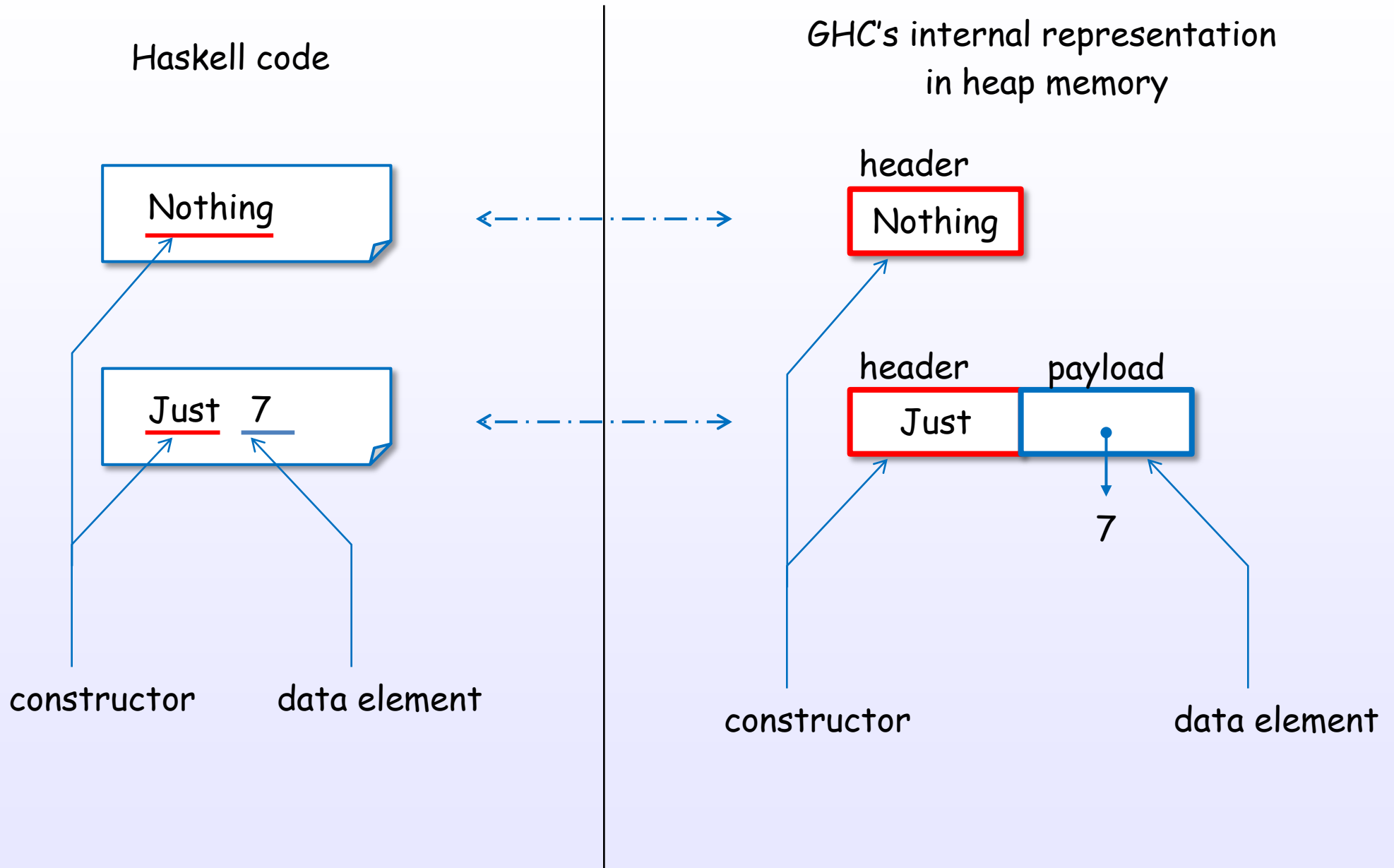
Constructorはdata文で宣言する代数的データ値



A constructor function builds a structured data value.

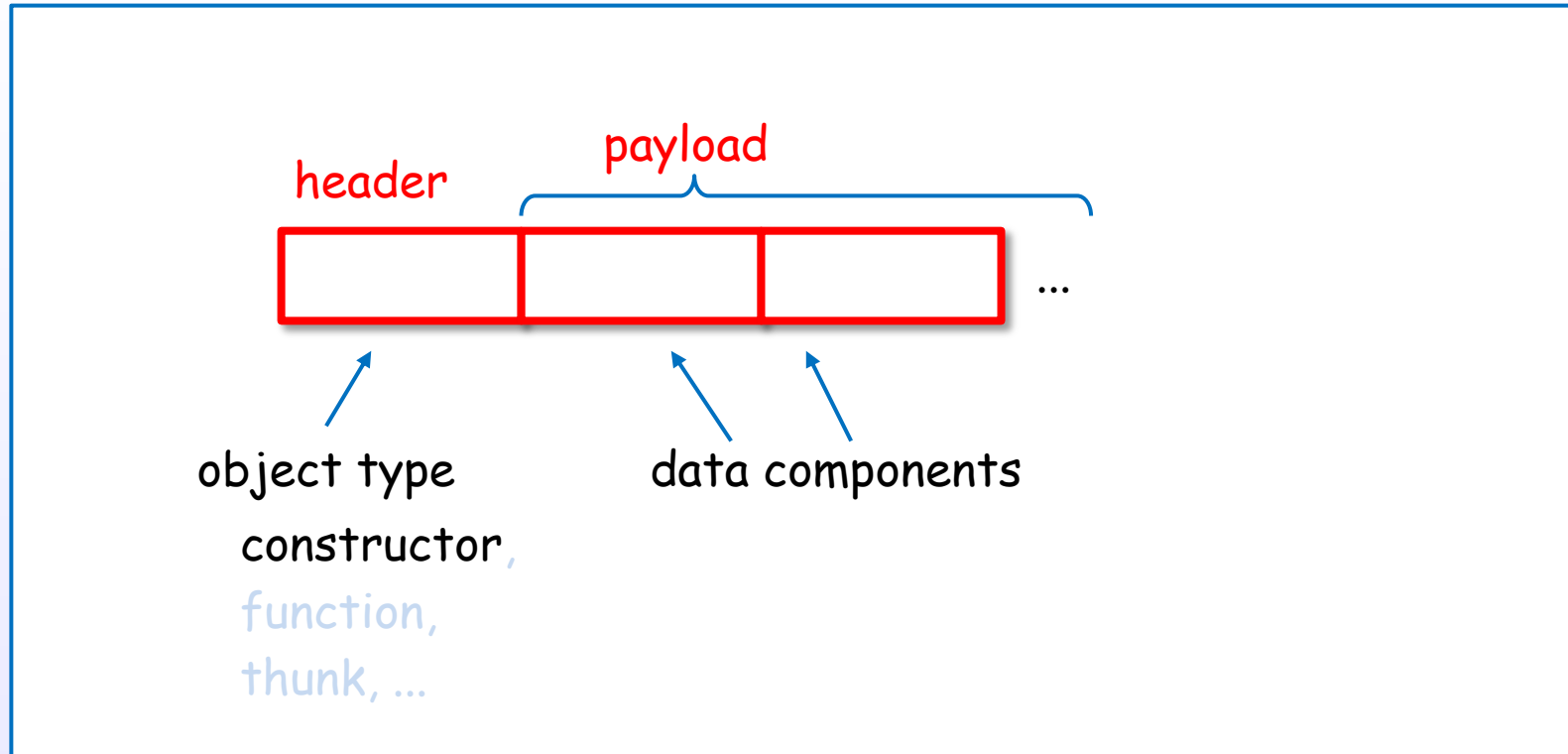
[slpj-book-1987] Ch.10

# Internal representation of Constructors for data values



# Constructors are represented uniformly

GHC's internal representation



in heap memory, stack, registers or static memory

# Representation of various constructors

Haskell code

```
data Bool = False
```

```
          | True
```

```
data Maybe a = Nothing
```

```
            | Just a
```

```
data Either a b = Left a
```

```
                | Right b
```

GHC's internal representation

False

True

Nothing

Just

a

Left

a

Right

b

# Primitive data types are also represented with constructor

Haskell code

```
data Int = I# 0#  
        | I# 1#  
        | :  
        | :
```

```
data Char = C# 'a'#  
          | C# 'b'#  
          | :  
          | :
```

GHC's internal representation

I#	0#
I#	1#
:	

C#	'a'#
C#	'b'#
:	

[Terei]



# List is also represented with constructor

List

```
[ 1, 2, 3 ]
```

syntactic desugar

```
1 : ( 2 : ( 3 : [] ) )
```

prefix notation by section

```
(:) 1 ( (:) 2 ( (:) 3 [] ) )
```

equivalent data constructor

```
Cons 1 ( Cons 2 ( Cons 3 Nil ) )
```

constructor

# List is also represented with constructor

List

```
[ 1, 2, 3 ]
```

syntactic desugar

```
1 : ( 2 : ( 3 : [] ) )
```

prefix notation by section

```
(:) 1 ( (:) 2 ( (:) 3 [] ) )
```

equivalent data constructor

```
Cons 1 ( Cons 2 ( Cons 3 Nil ) )
```

type declaration

*\* pseudo code*

```
data List a = []  
             | : a (List a)
```

```
data List a = Nil  
             | Cons a (List a)
```

# List is also represented with constructor

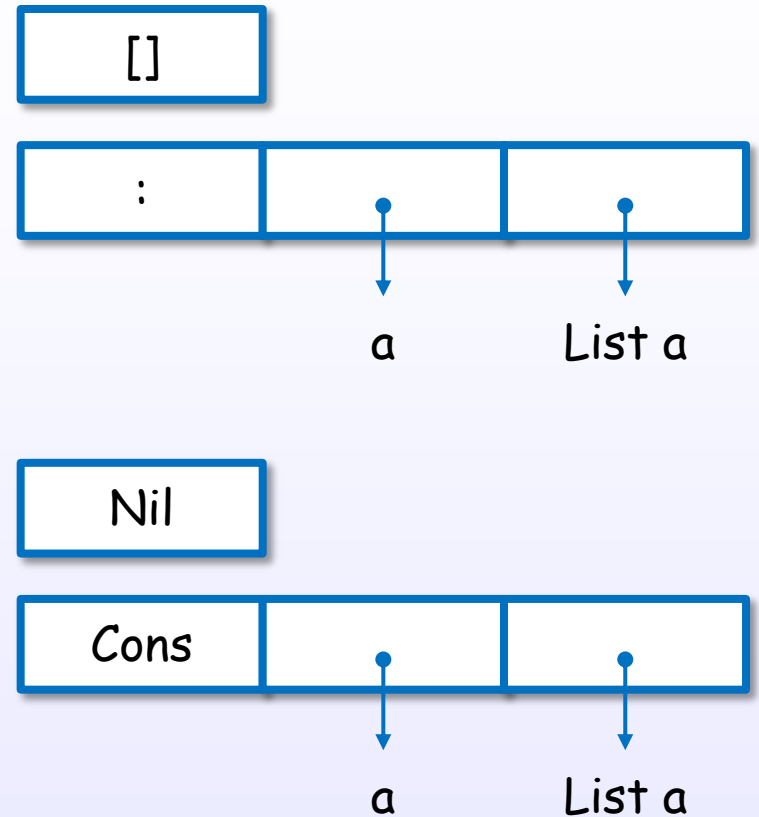
Haskell code

```
data List a = []  
           | : a (List a)
```

equivalent data constructor

```
data List a = Nil  
           | Cons a (List a)
```

GHC's internal representation



# List is also represented with constructor

Haskell code

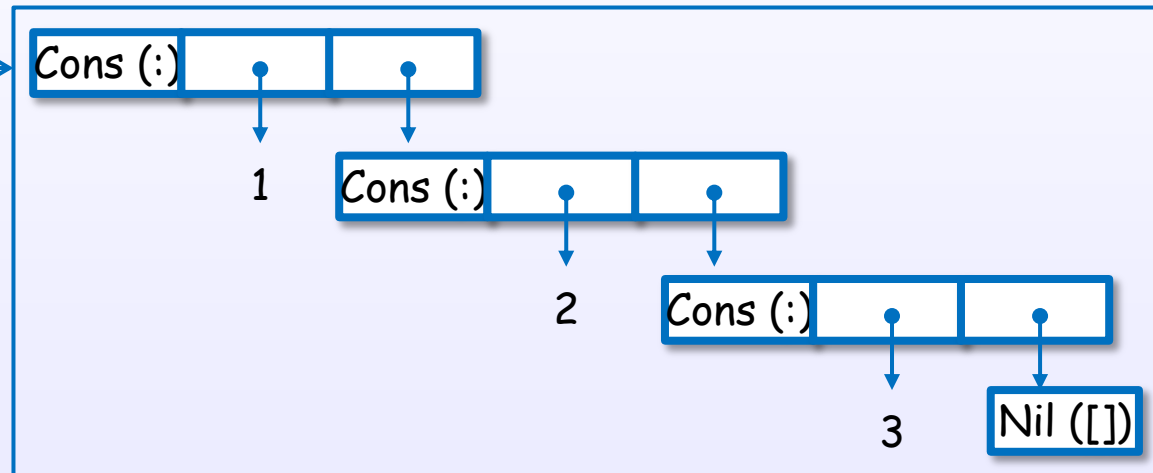
```
[ 1, 2, 3 ]
```

```
1 : ( 2 : ( 3 : [] ) )
```

```
(:) 1 ( (:) 2 ( (:) 3 [] ) )
```

```
Cons 1 ( Cons 2 ( Cons 3 Nil ) )
```

GHC's internal representation



# Tuple is also represented with constructor

Tuple (Pair)

( 7 , 8 )

prefix notation by section

(,) 7 8

equivalent data constructor

Pair 7 8

constructor

type declaration

*\*pseudo code*

data Pair a = (,) a a

data Pair a = Pair a a

# Tuple is also represented with constructor

Haskell code

```
(7, 8)
```

```
(,) 7 8
```

```
Pair 7 8
```

GHC's internal representation

```
Pair (,)
```

7

8

### 3. Internal representation of expressions

Thunk

# A value or an unevaluated expression

## Expressions

unevaluated expressions

unevaluated expressions

values

data values

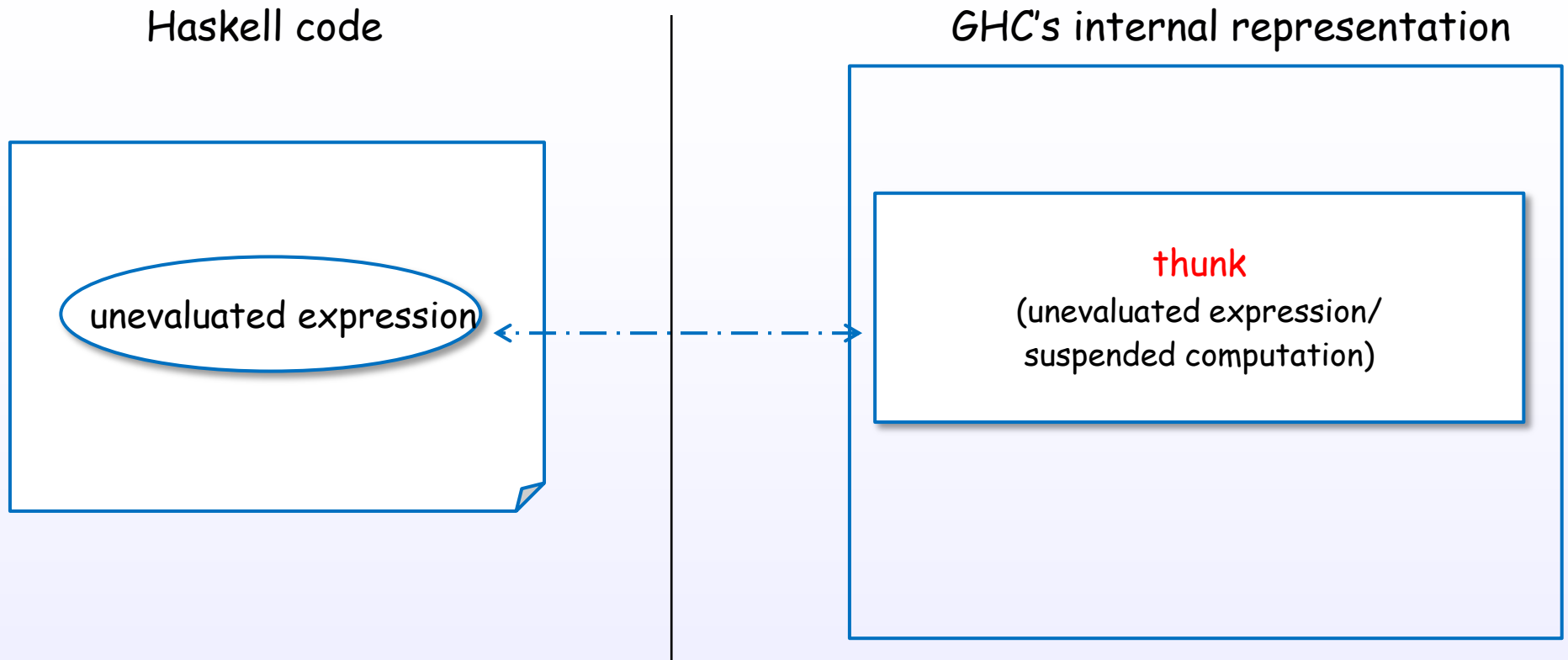
function values

値か否か。値は2種。

[STG]



# Thunk



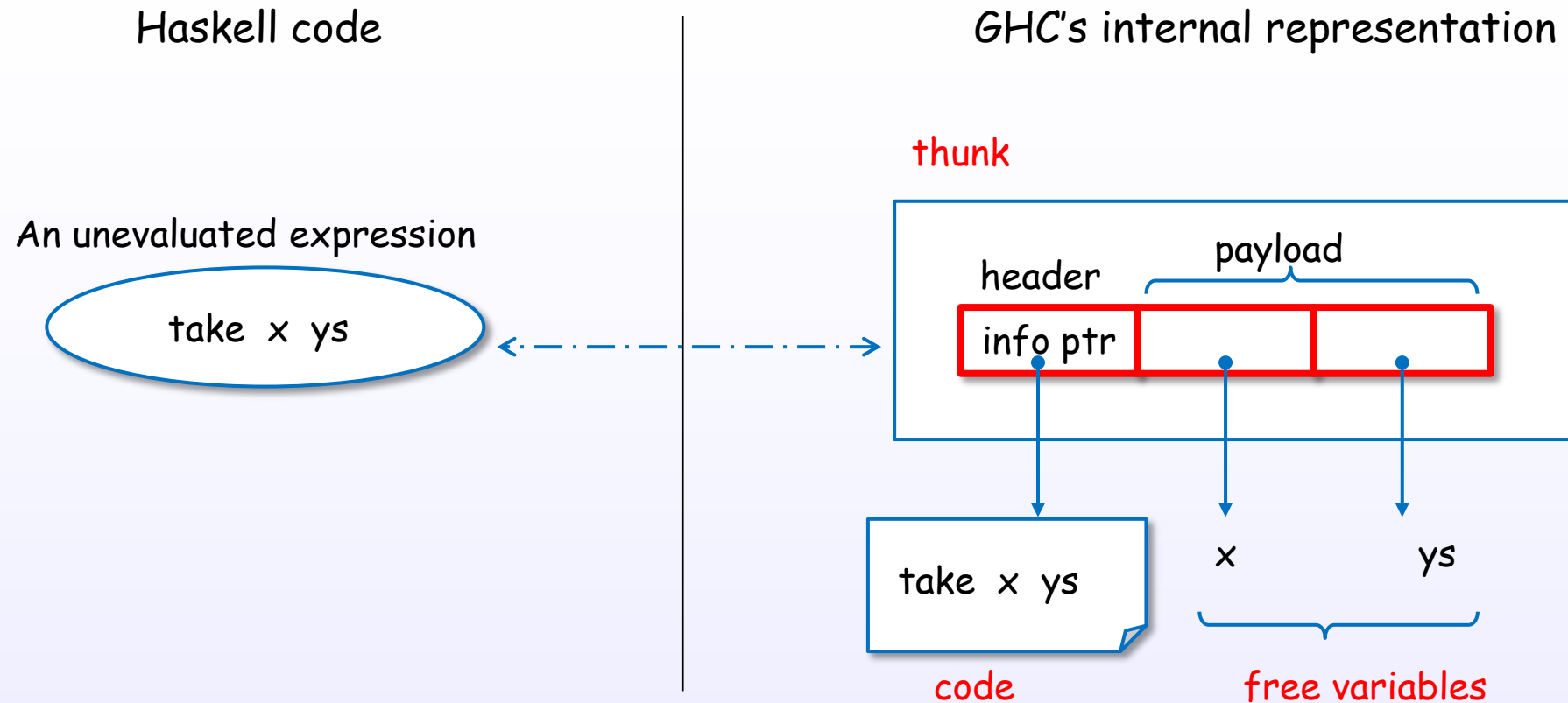
A thunk is an **unevaluated** expression in heap memory.  
A thunk is built to **postpone** the evaluation.

[parconc, Ch.2]

[hack.hands]

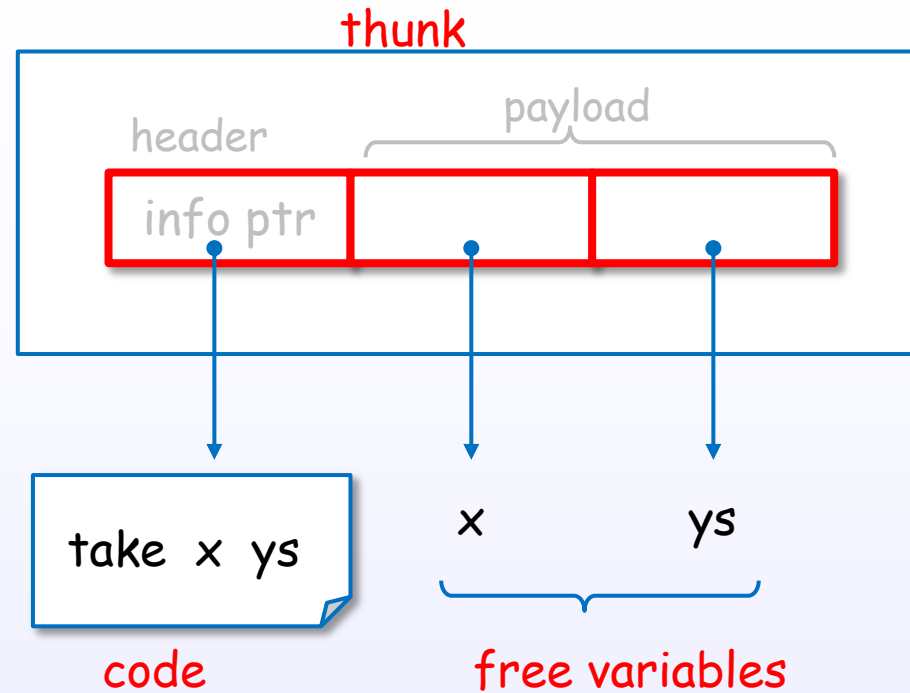
[Haskell/Laziness]

# Internal representation of thunk



A thunk is represented with header(code) + payload(free variables).

# A thunk is a package of code and free variables



A thunk is a package of code + free variables.

[CIS194]

# A thunk is evaluated by forcing request

Haskell code

An unevaluated expression

take x ys



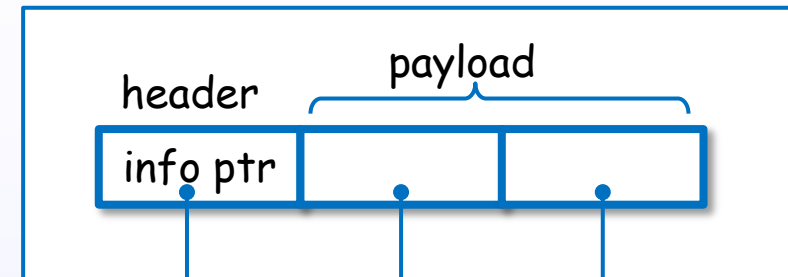
forcing

[ 3 ]

An evaluated expression

GHC's internal representation

thunk



take x ys

code

free variables



forcing



3

Nil ([])

### 3. Internal representation of expressions

Uniform representation

# A value or an unevaluated expression

## Expressions

unevaluated expressions

unevaluated expressions

values

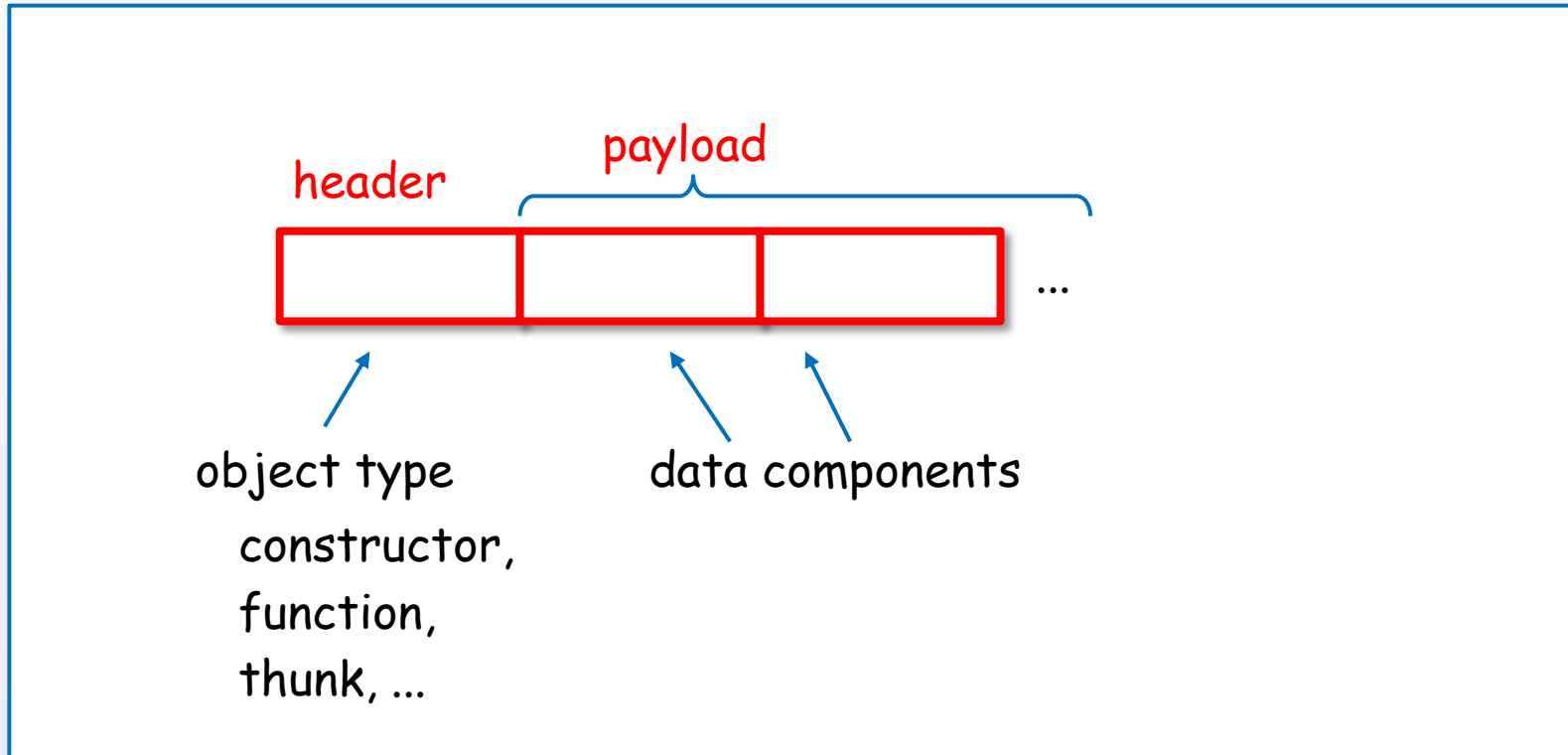
data values

function values

値か否か。値は2種。

[STG]

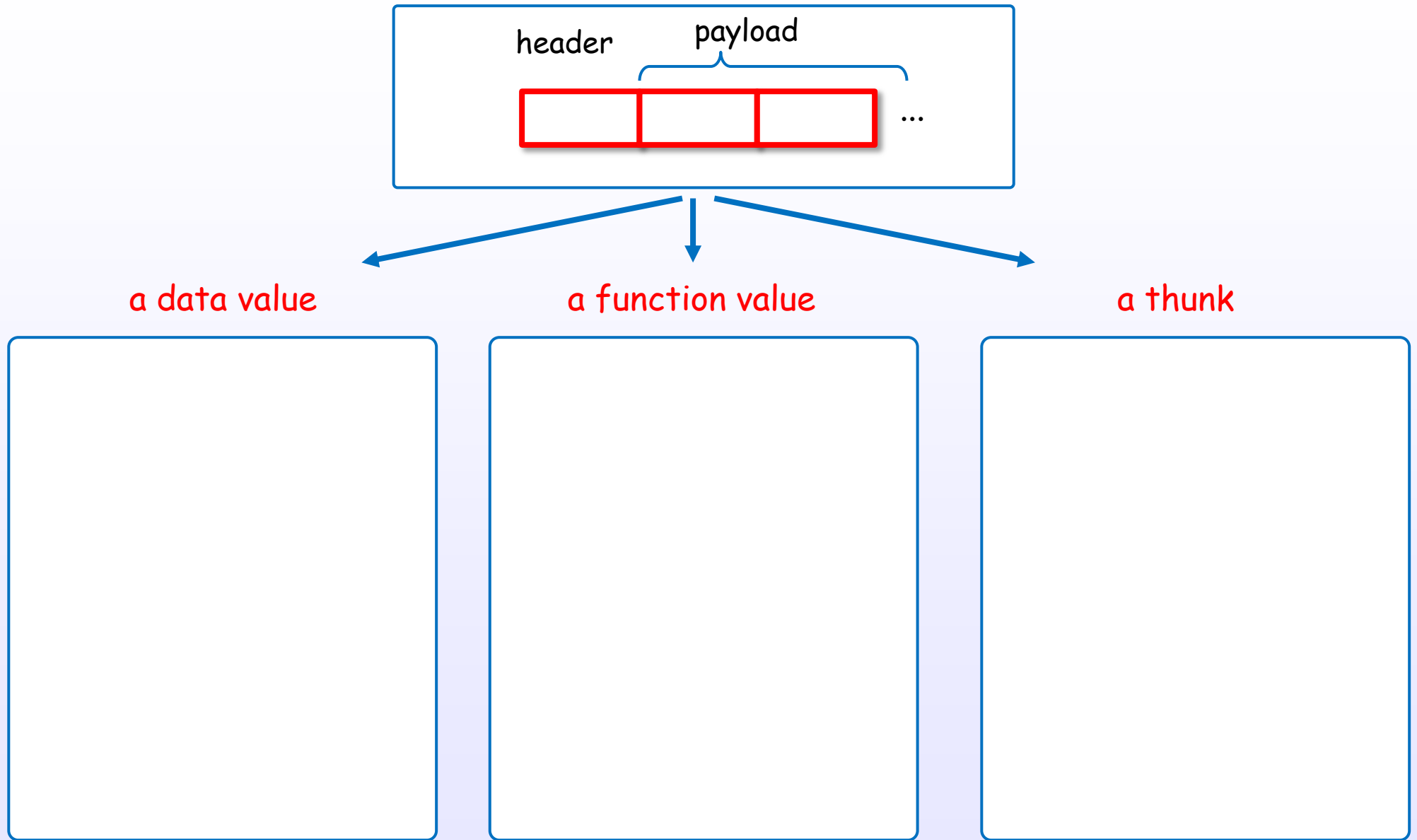
# Every object is represented uniformly in memory



in heap memory, stack, registers or static memory

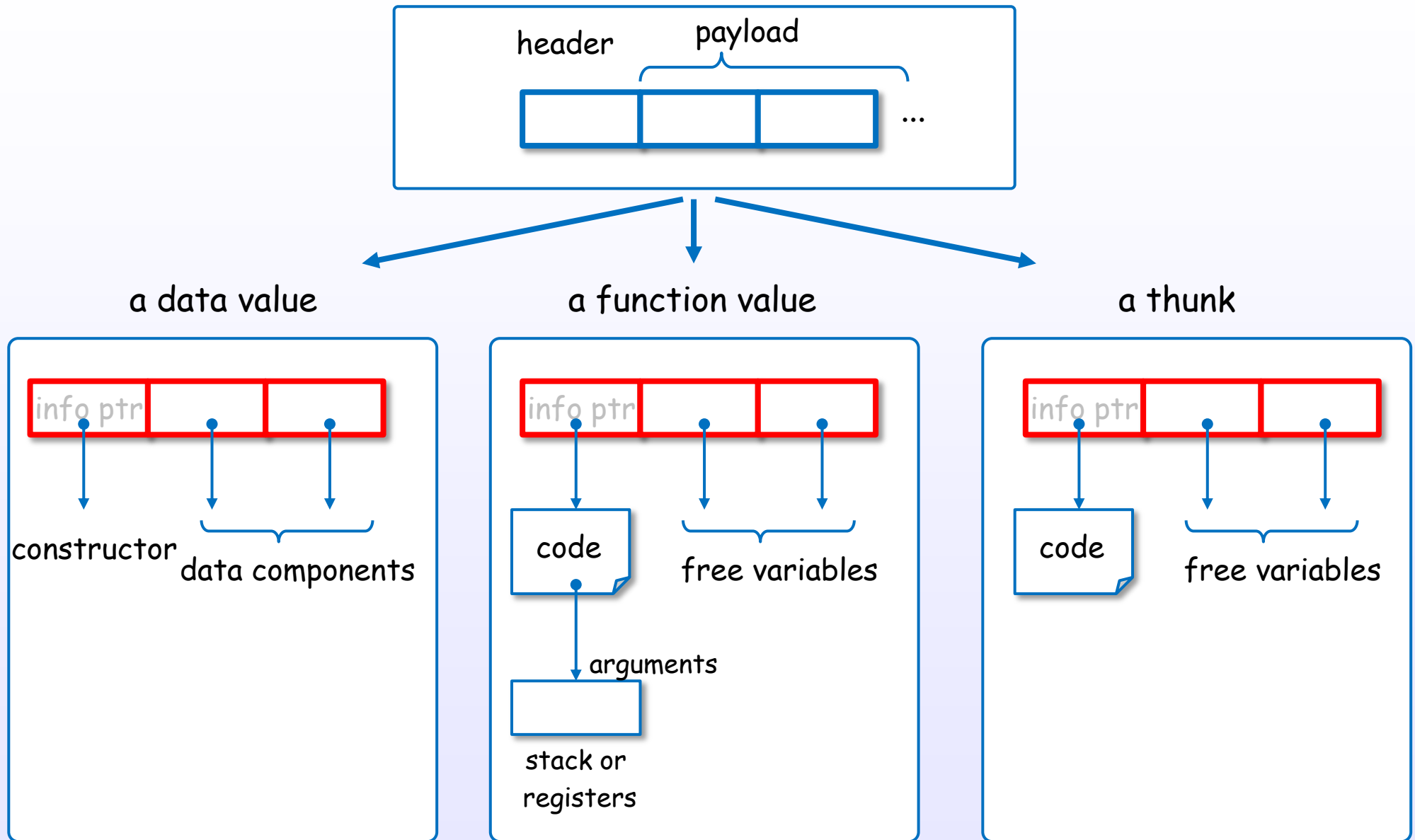
[STG]

# Every object is represented uniformly





# Every object is represented uniformly



いずれも、広義の、“closure” (= code + environment(free variables))

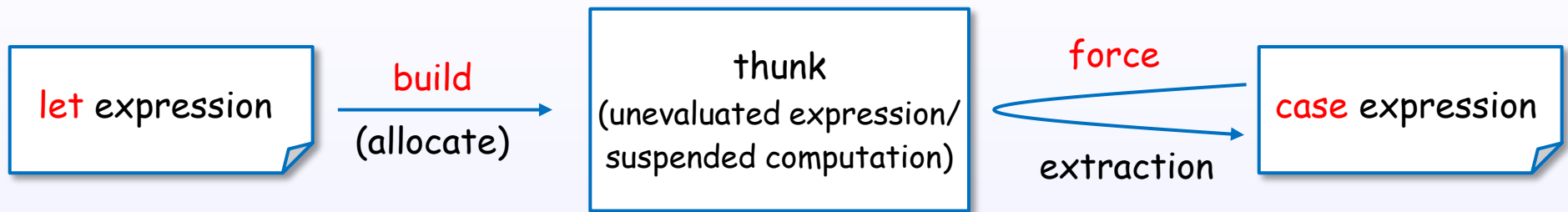
### 3. Internal representation of expressions

let, case expression

# let, case expression

let and case expressions are special role for evaluation

# let/case expressions and thunk



A let expression may build a thunk.

A case expression forces and deconstructs the thunk.

# A let expression builds a thunk

let expression

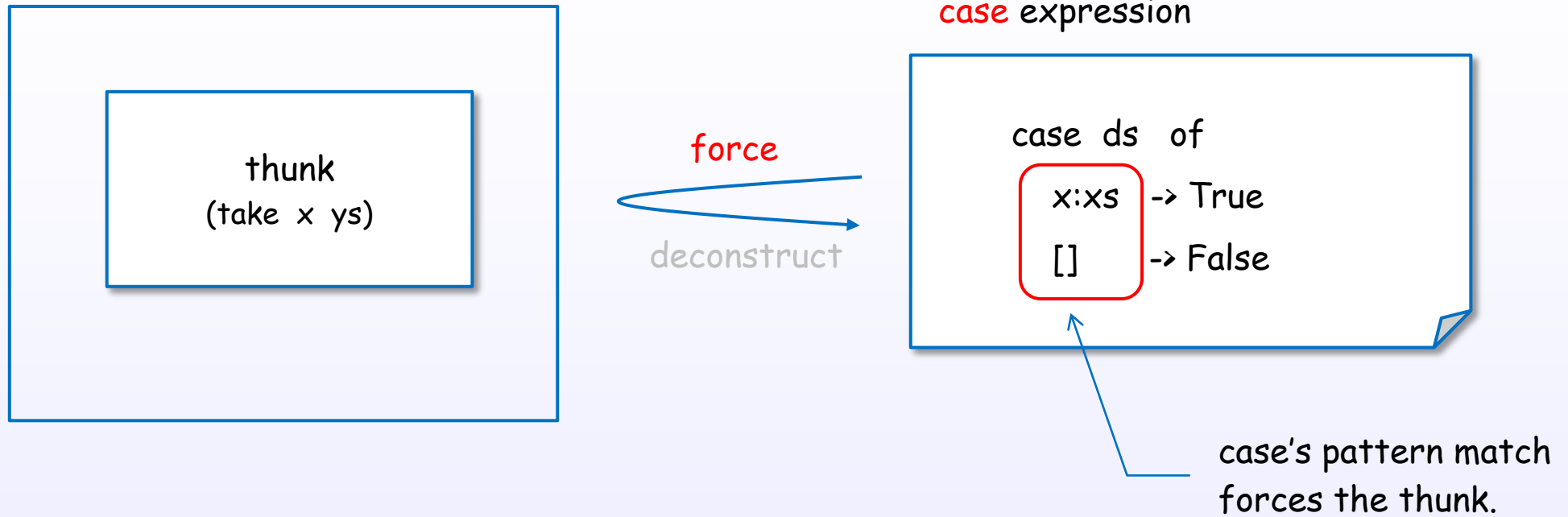
let ds = take x ys

build  
(allocate)

thunk  
(take x ys)

heap memory

# A case expression forces a thunk



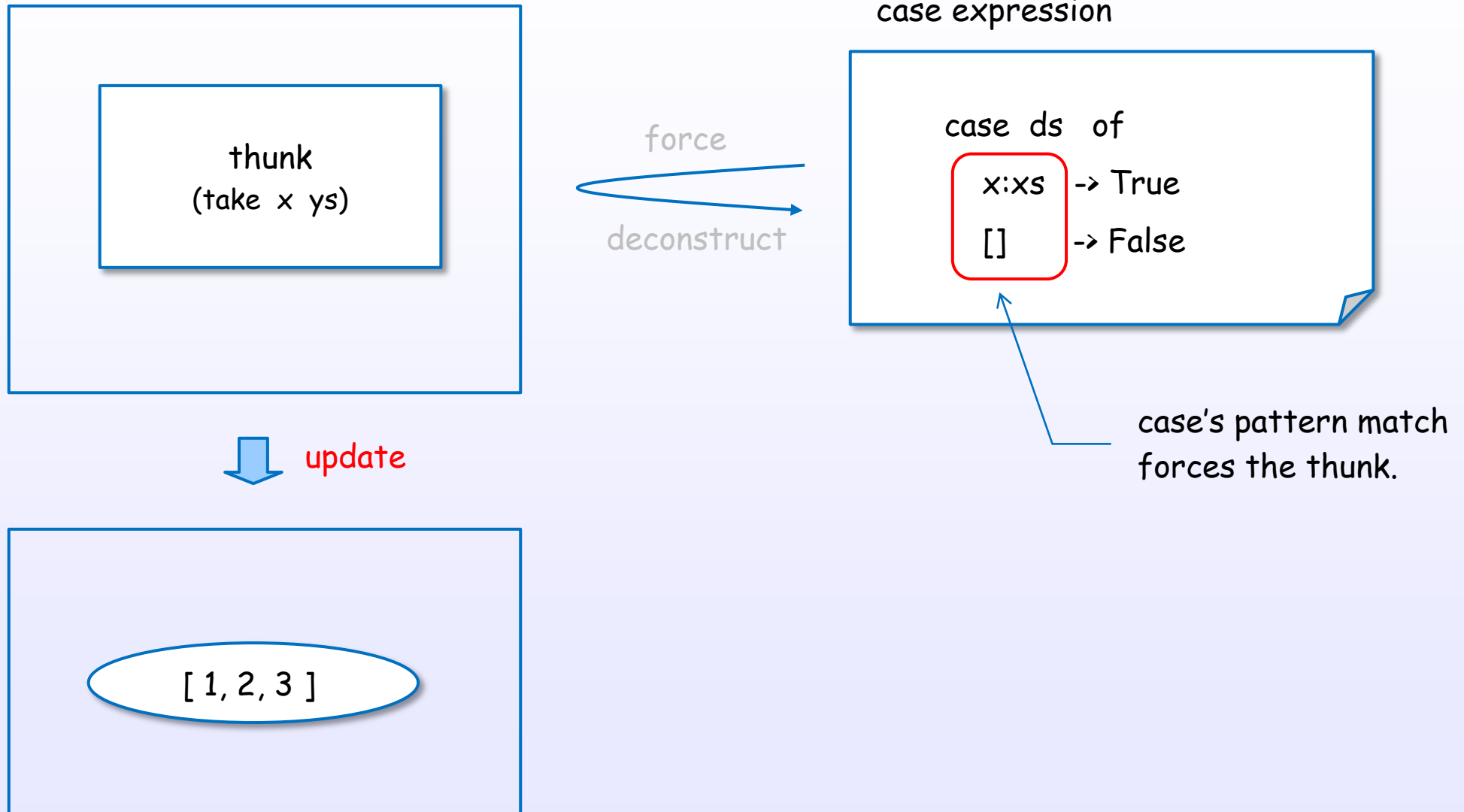
[Terei]

[CIS194]

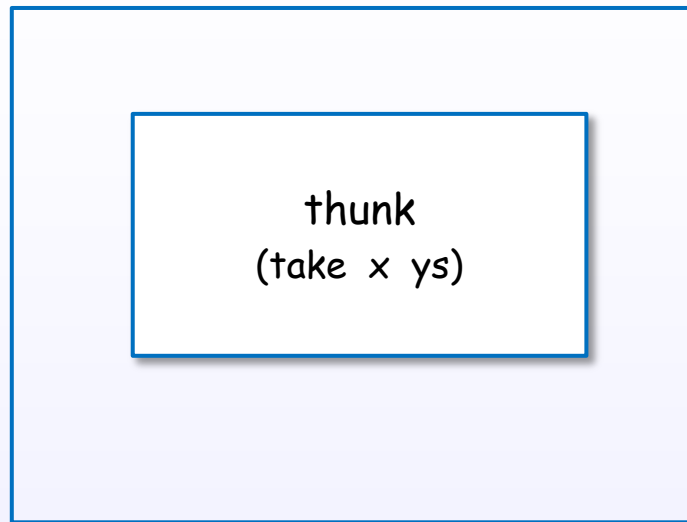
[STG]

References : [1]

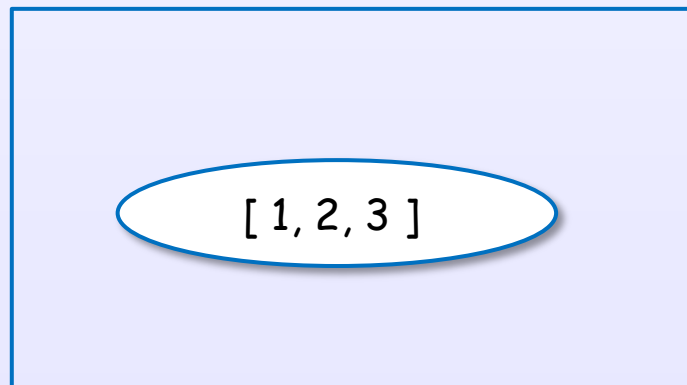
# A case expression forces a thunk



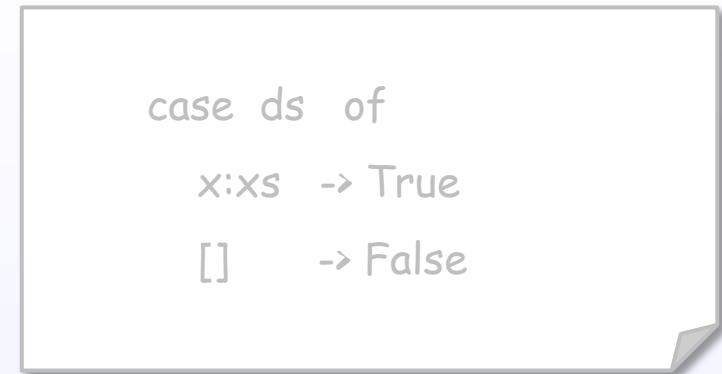
# A case expression forces a thunk



↓ update

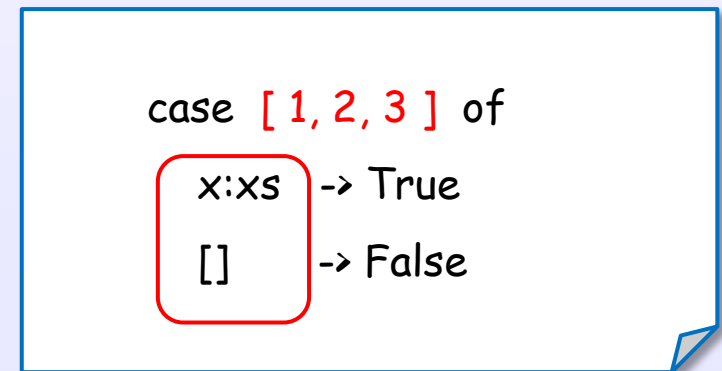


case expression



force  
→  
deconstruct

case expression



force  
→  
deconstruct



# Forcing and update

Haskell code

```
let x = expn
```

Expression graph



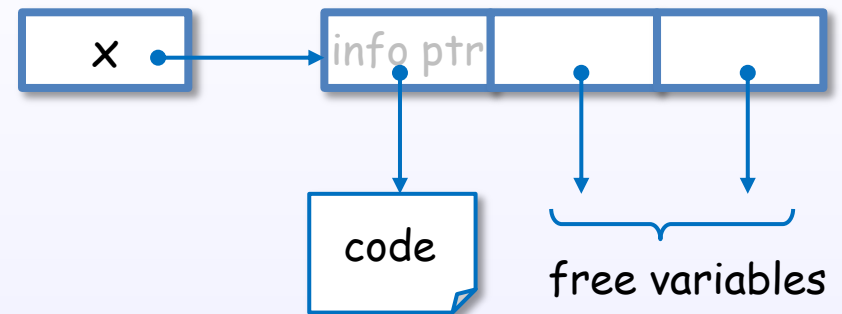
required



update



GHC internal representation



a value



when a variable is bound

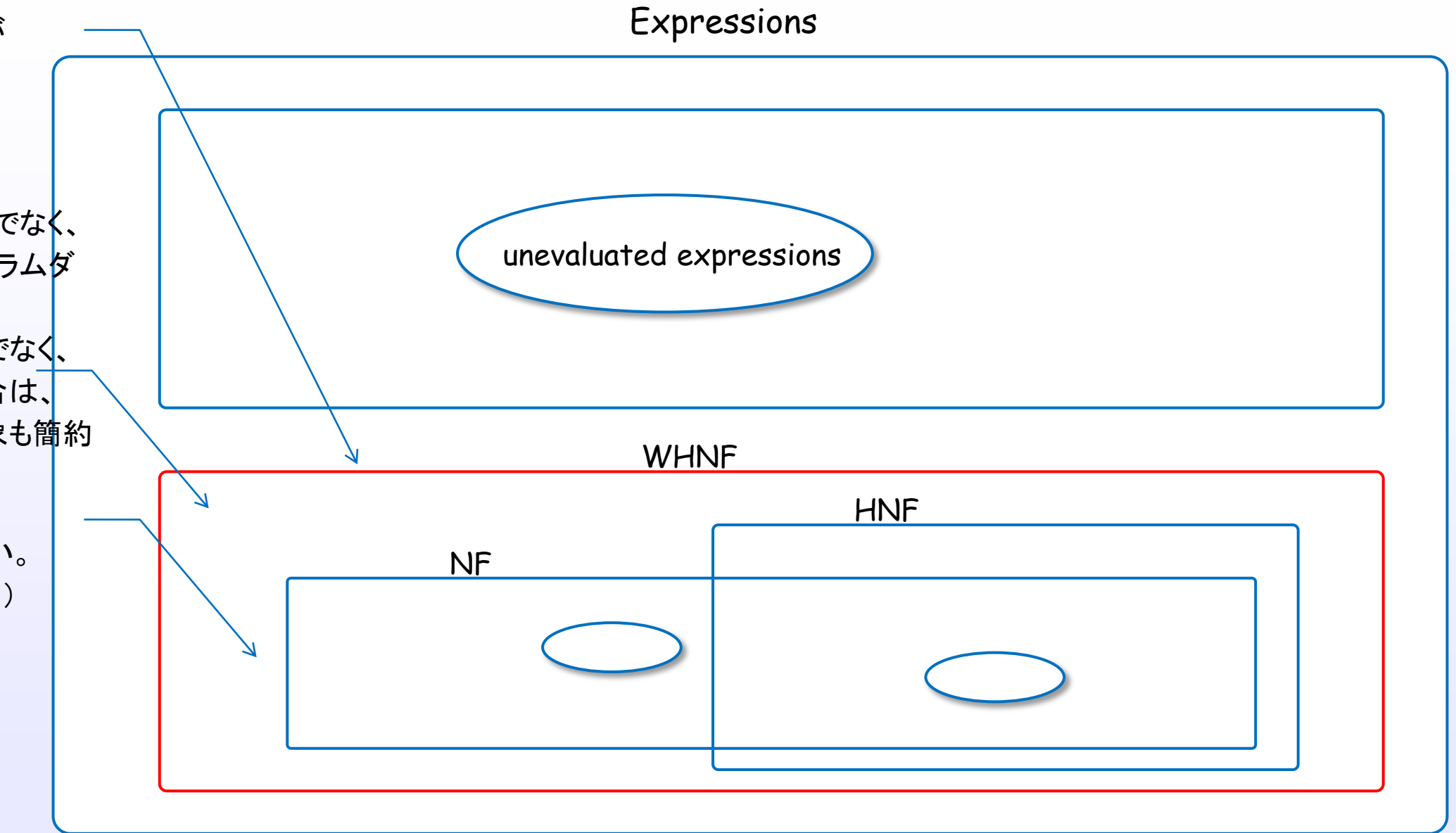
it is generally bound to an unevaluated closure allocated in the heap

このイメージを伝える

### 3. Internal representation of expressions

WHNF

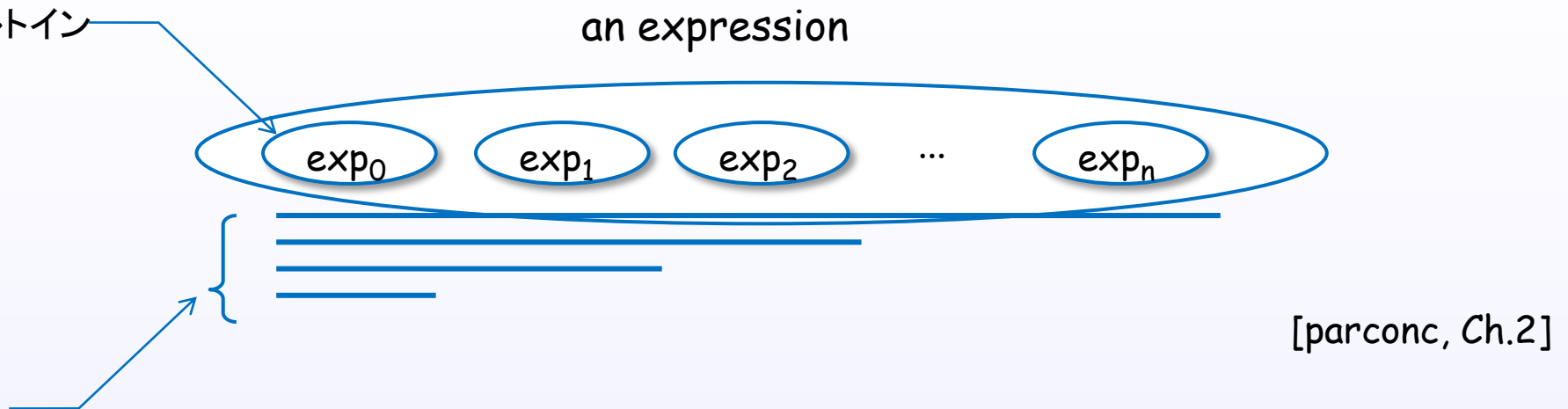
# A value has various form level (evaluation level)



値には、評価レベルがある。

[STG]

データ抽象、ビルトイン



more

An expression has no top level redex, if it is in WHNF.

[slpj-book-1987]

These are in weak head normal form,  
but not in normal form, since they contain inner redex. (p.198)

[stephen]

[hack.hands]

[Terei]

[Bird, Chapter 2, 7]

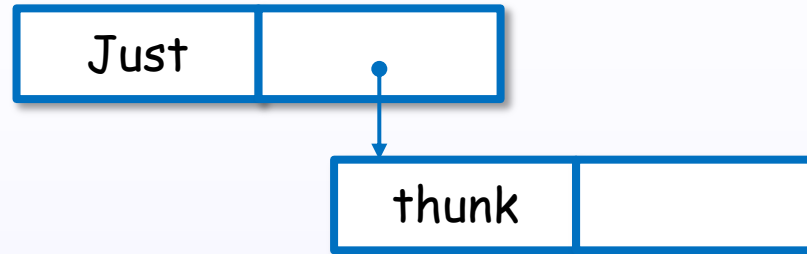
[TAPL, Chapter 3]

[Terei]

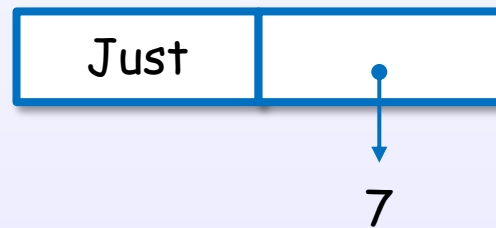
References : [1]

# Internal representation of WHNF

Just (thunk)



Just 7



constructor can contain unevaluated expression

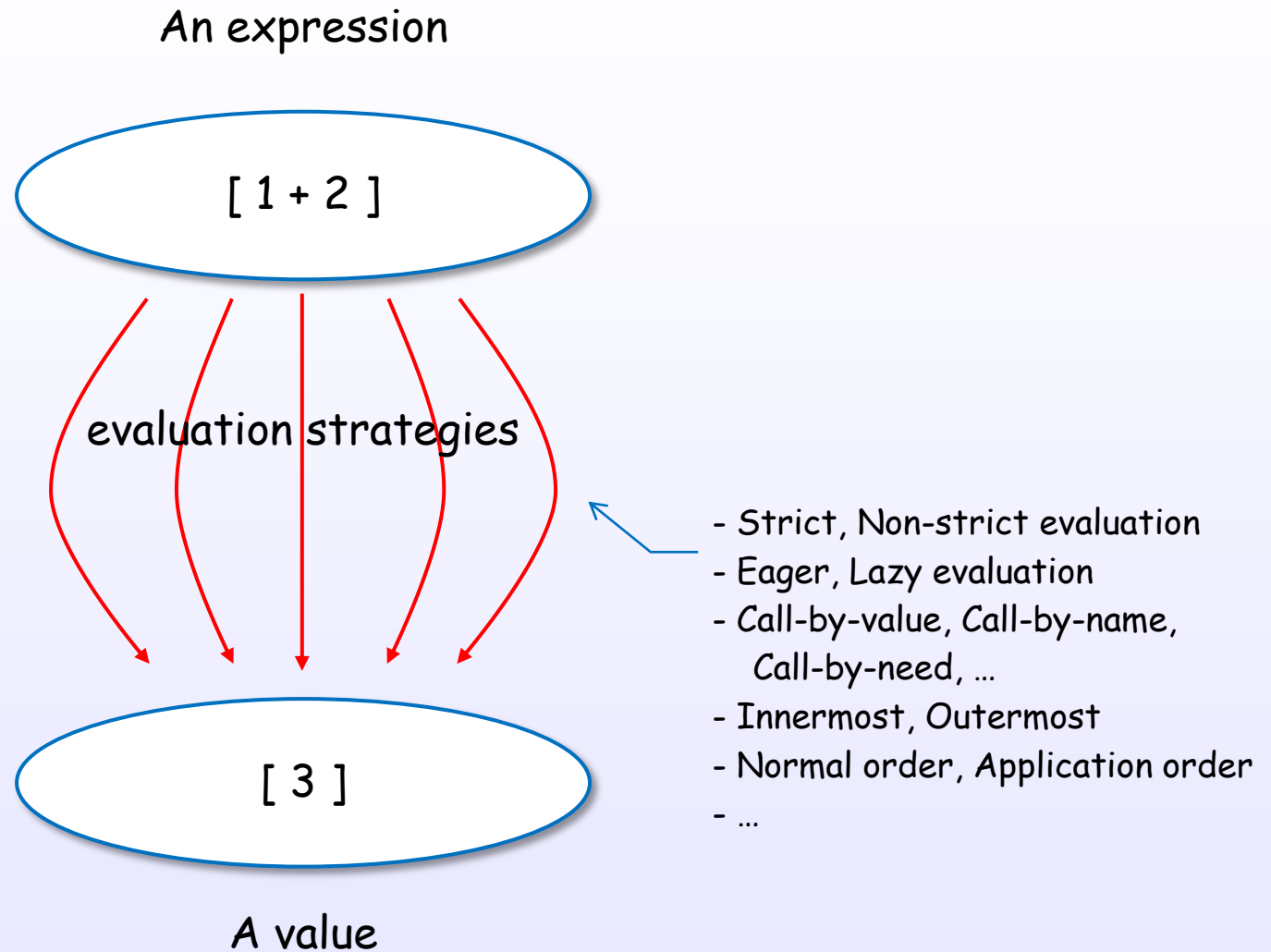
Lazy constructor

## 4. Evaluation

## 4. Evaluation

Evaluation strategies

# There are many evaluation approaches



[Bird, Chapter 2, 7]

[TAPL, Chapter 3]



# Evaluation concept layer

Denotational semantics

Operational semantics  
(Evaluation strategies / Reduction strategies)

Implementation techniques

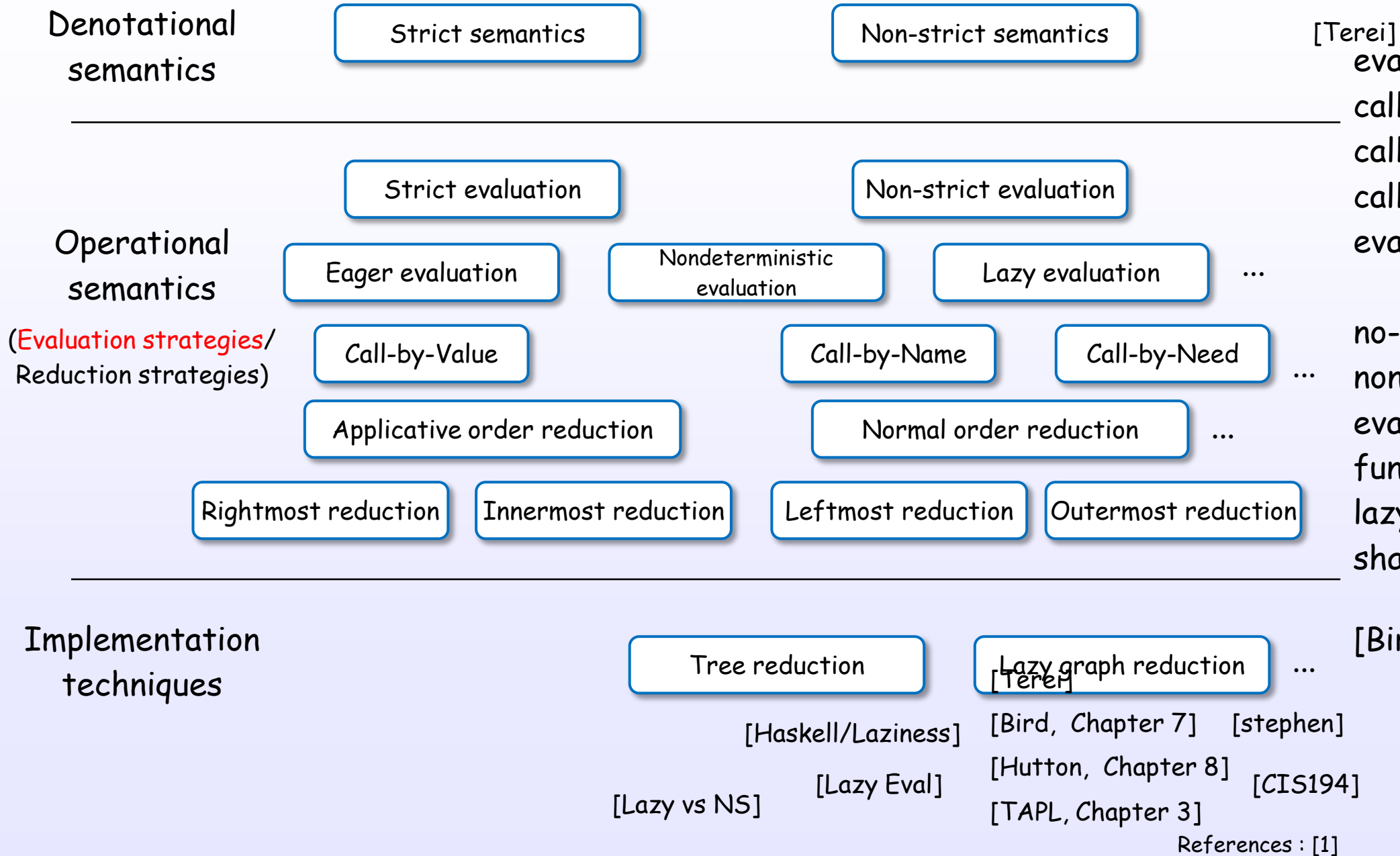
[Bird, Chapter 7]

[Hutton, Chapter 8]

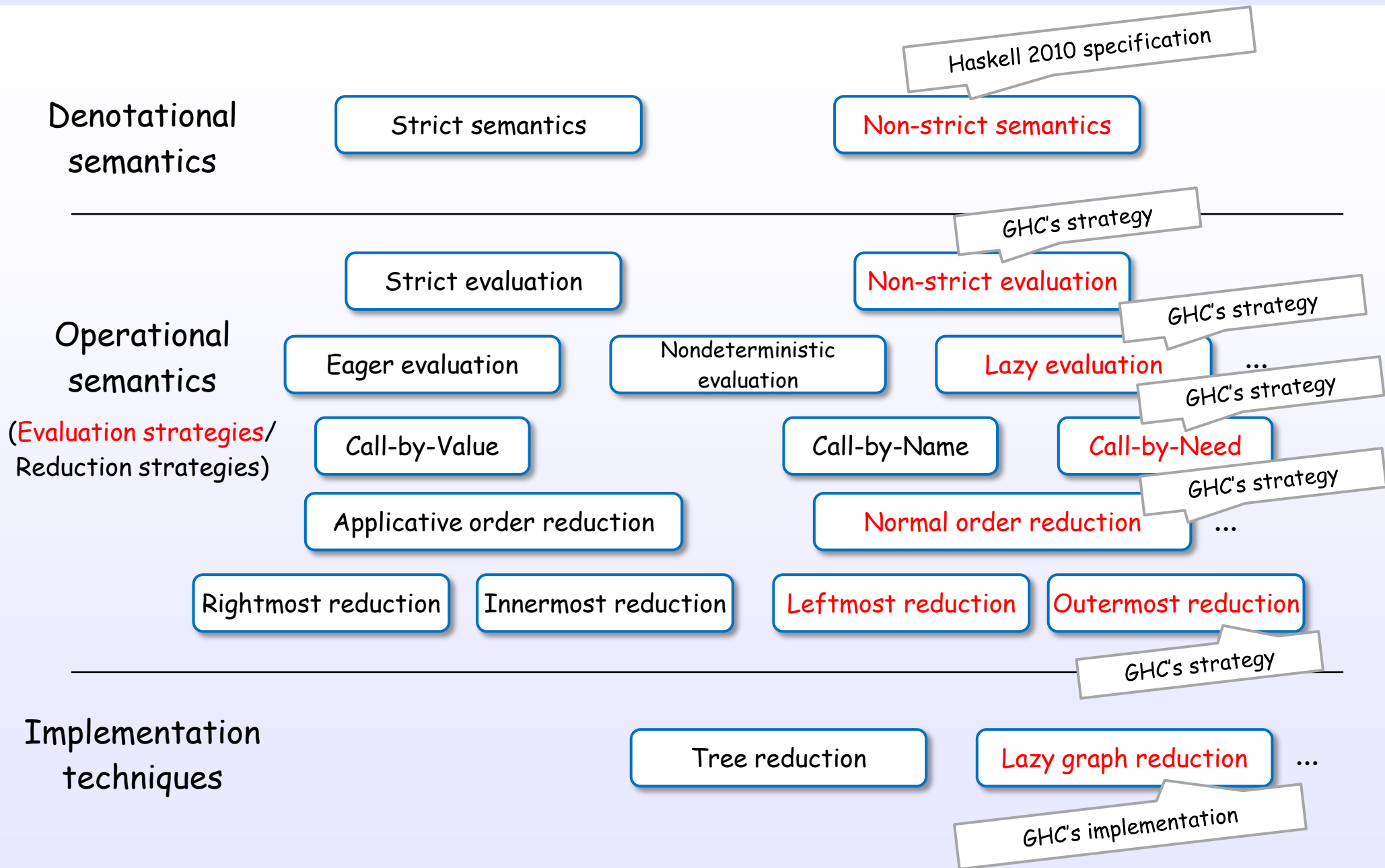
[TAPL, Chapter 3]

References : [1]

# Evaluation layer for GHC's Haskell



# Evaluation layer for GHC's Haskell



# Evaluation strategies and order

$a(b\ c) + d(e\ (f\ g))$

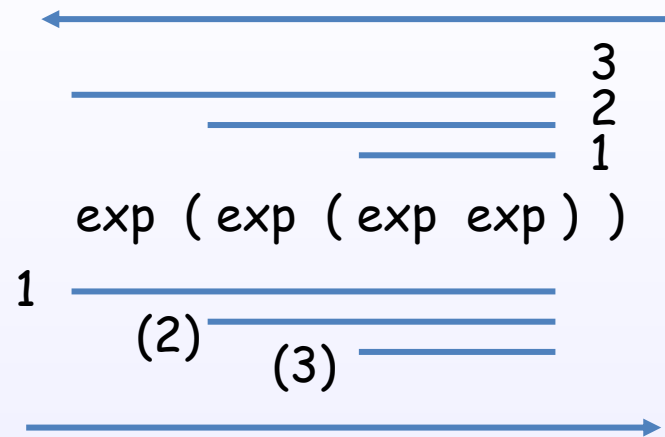
order

[Bird]  
[Hutton]

References : [1]

# Evaluation strategies and order

eager evaluation, call-by-value, innermost reduction, applicative order reduction

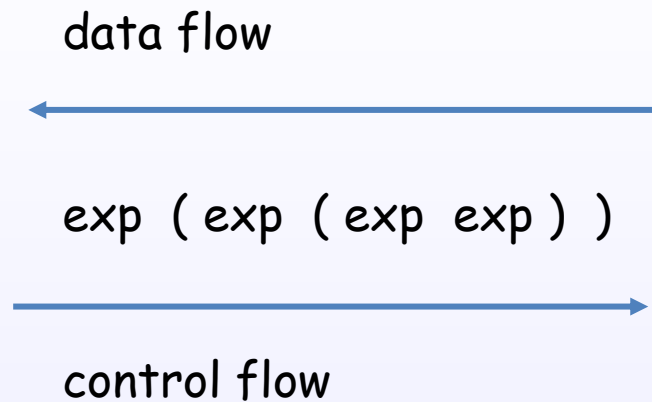


lazy evaluation, call-by-name, call-by-need, outermost reduction, normal order reduction

[Bird]  
[Hutton]

References : [1]

# Evaluation strategies and order



lazy evaluation, call-by-name, call-by-need, outermost reduction, normal order reduction

# Simple example of typical evaluations

## Eager evaluation (Strict evaluation)

default  
C, Java, JavaScript,  
Python, OCaml, Scheme, ...

square ( 1 + 2 )



argument  
evaluation  
first

square ( 3 )



3 \* 3



9

## Lazy evaluation (Non-strict evaluation)

default  
Haskell (GHC), ...

square ( 1 + 2 )



apply  
first

( 1 + 2 ) \* ( 1 + 2 )



( 3 ) \* ( 3 )



9

[Bird]  
[Hutton]

# Simple example of typical evaluations

Eager evaluation  
(Strict evaluation)

square ( 1 + 2 )



square ( 3 )



3 \* 3



9

argument  
evaluated

Lazy evaluation  
(Non-strict evaluation)

square ( 1 + 2 )



( 1 + 2 ) \* ( 1 + 2 )



( 3 ) \* ( 3 )



9

argument  
evaluation  
**delayed !**

[Bird]  
[Hutton]



## 4. Evaluation

Evaluation in Haskell (GHC)

# Haskell(GHC) 's lazy evaluation

ingredient of Haskell's "lazy evaluation"

evaluate only if needed

normal order reduction  
(= leftmost + outermost  
reduction)

+

evaluate only necessary  
part

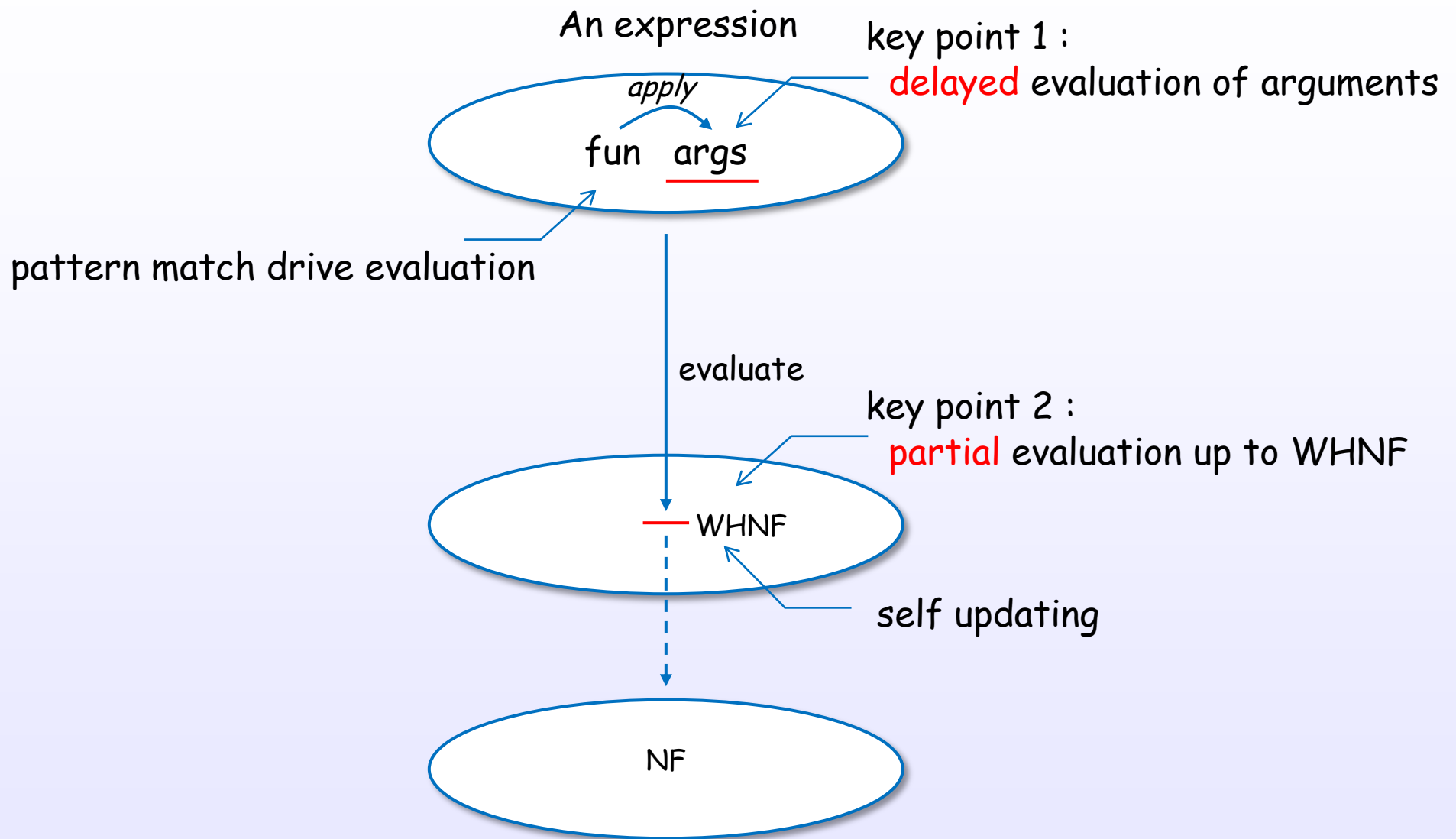
stopping at  
WHNF  
lazy constructor

+

evaluate only once

substitute pointers  
self-updating model  
model

# Key concept of Haskell's lazy evaluation



# 1. Example of GHC's evaluation

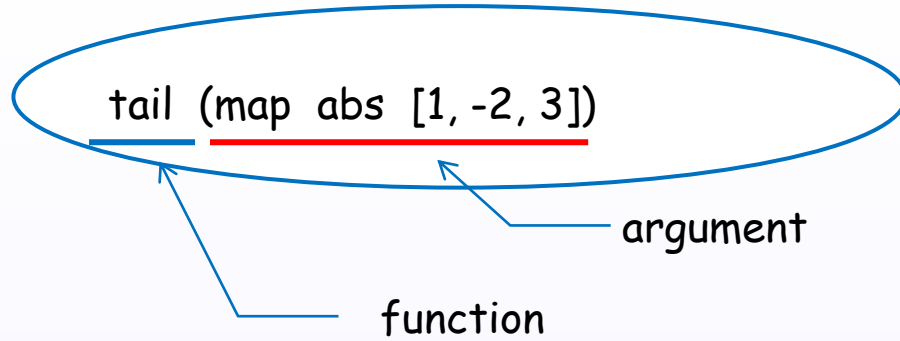


`tail (map abs [1, -2, 3])`

Are you ready for evaluation?

It's time to magic!

## 2. How to postpone the evaluation of arguments?



### 3. GHC transforms internally the expression

tail (map abs [1, -2, 3])

syntactic desugar

let thunk0 = map abs [1, -2, 3]  
in tail thunk0

## 4. a let expression builds a thunk

tail (map abs [1, -2, 3])

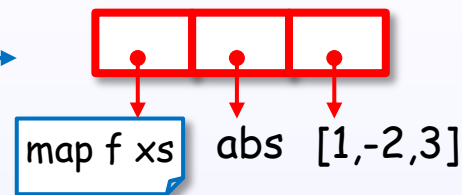
syntactic desugar

let **thunk0 = map abs [1, -2, 3]**  
in tail thunk0

build

heap memory

thunk



## 5. function apply to argument

tail (map abs [1, -2, 3])

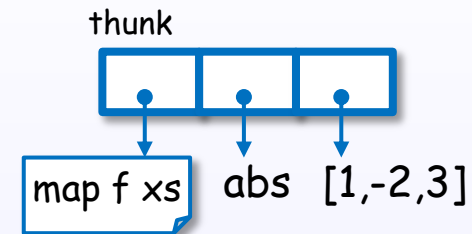
syntactic desugar

let thunk0 = map abs [1, -2, 3]

in tail thunk0

*apply*

heap memory





## 6. tail is defined here

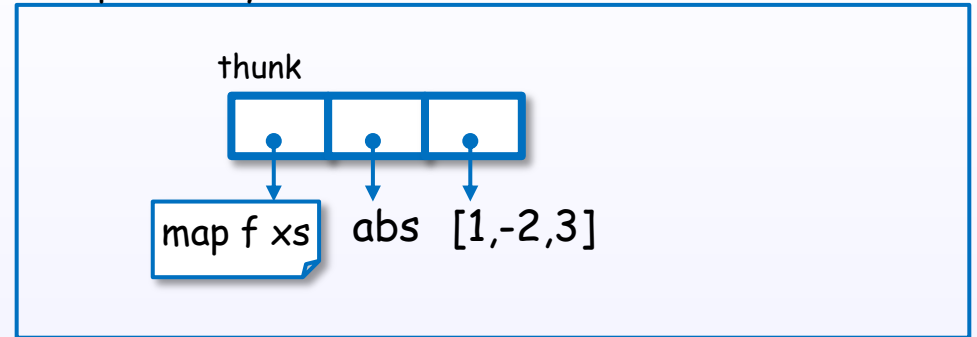
`tail (map abs [1, -2, 3])`

syntactic desugar

`let thunk0 = map abs [1, -2, 3]  
in tail thunk0`

`tail (_:xs) = xs`

heap memory



## 7. function is syntactic sugar

`tail (map abs [1, -2, 3])`

syntactic desugar

`let thunk0 = map abs [1, -2, 3]`

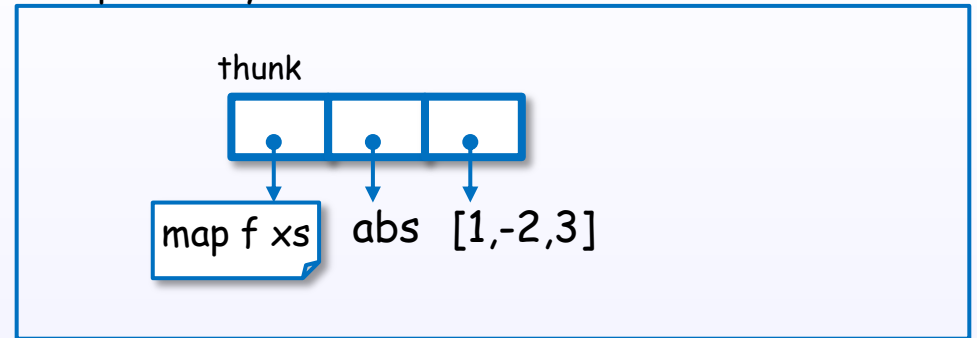
`in tail thunk0`

`tail (_:xs) = xs`

syntactic  
desugar

`tail y = case y of  
 (_:xs) -> xs`

heap memory



## 8. substitute function body (beta reduction)

tail (map abs [1, -2, 3])

syntactic desugar

let thunk0 = map abs [1, -2, 3]  
in tail thunk0

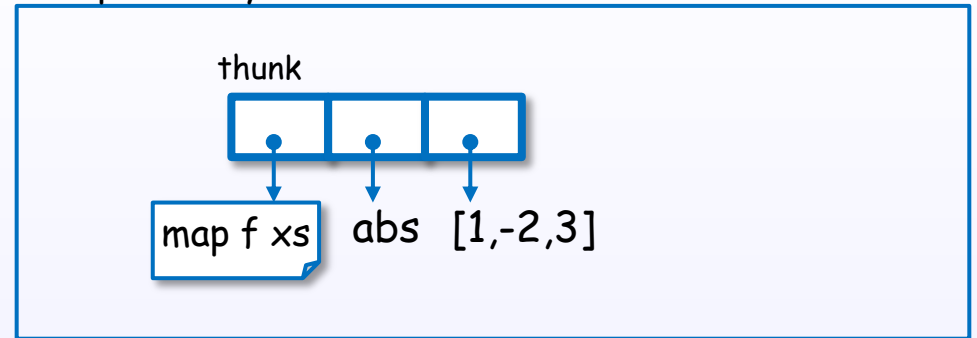
tail ( \_:xs) = xs

tail y = case y of  
 ( \_:xs) -> xs

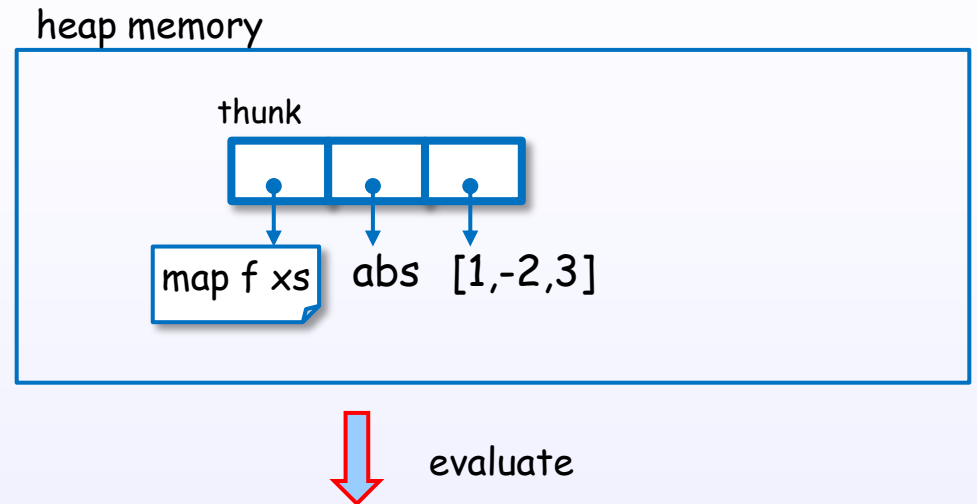
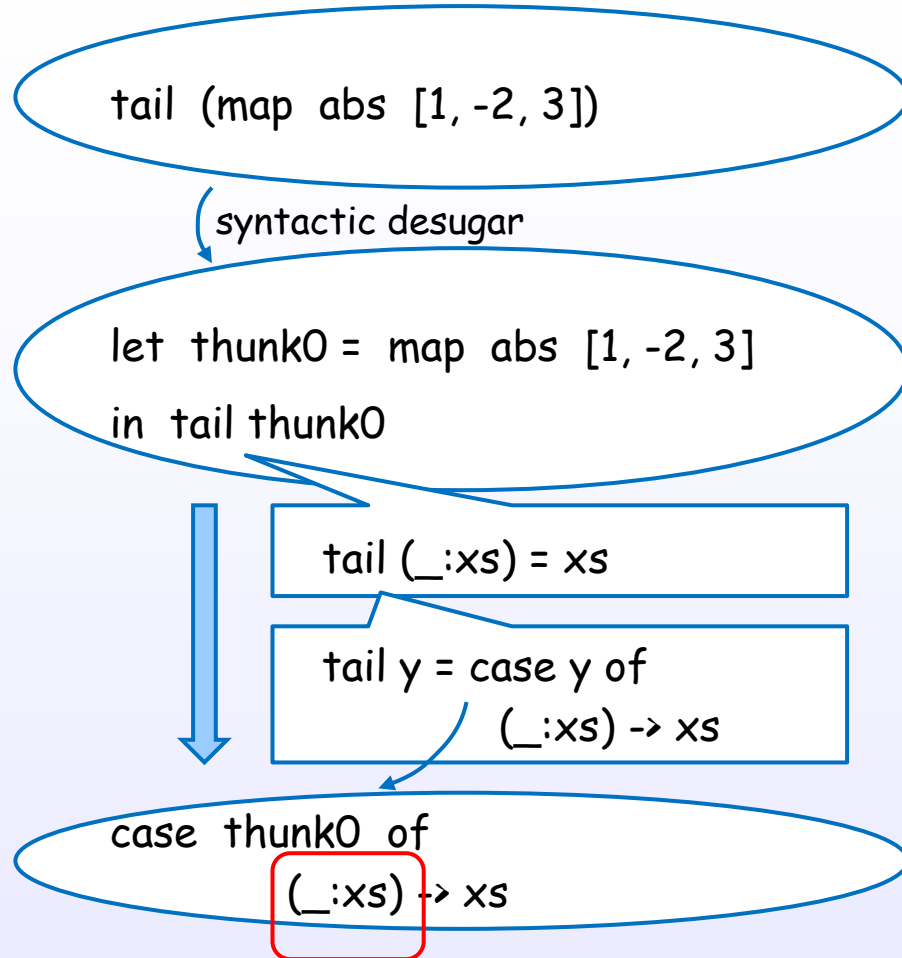
reduction

case thunk0 of  
 ( \_:xs) -> xs

heap memory



## 9. case pattern match drive evaluation



## 10. but, stop at WHNF

`tail (map abs [1, -2, 3])`

syntactic desugar

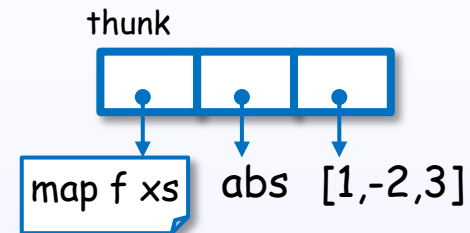
`let thunk0 = map abs [1, -2, 3]  
in tail thunk0`

`tail (_:xs) = xs`

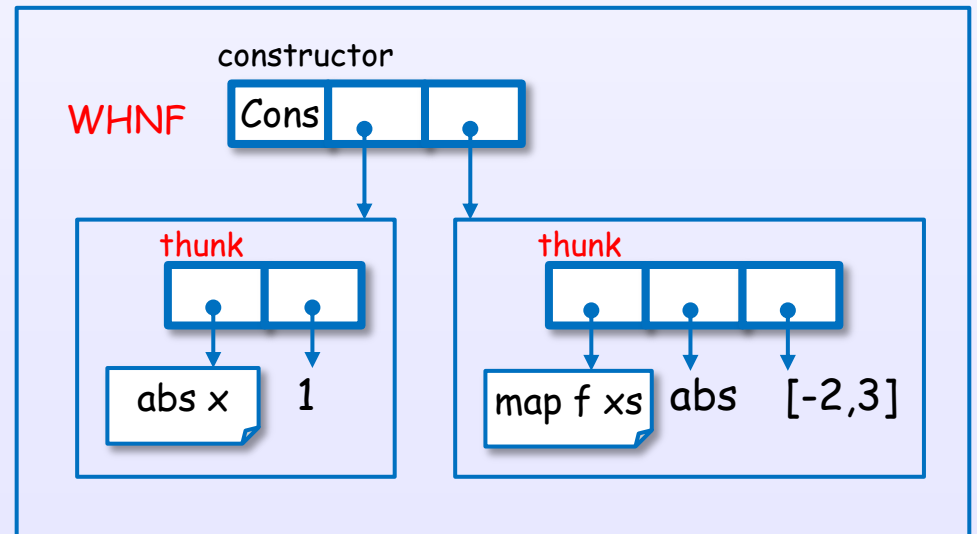
`tail y = case y of  
 (_:xs) -> xs`

`case thunk0 of  
 (_:xs) -> xs`

heap memory



evaluate



# 11. bind variables to result

tail (map abs [1, -2, 3])

syntactic desugar

let thunk0 = map abs [1, -2, 3]  
in tail thunk0

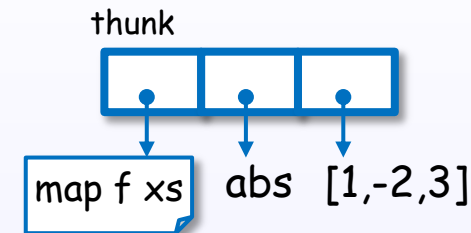
tail (\_:xs) = xs

tail y = case y of  
(\_:xs) -> xs

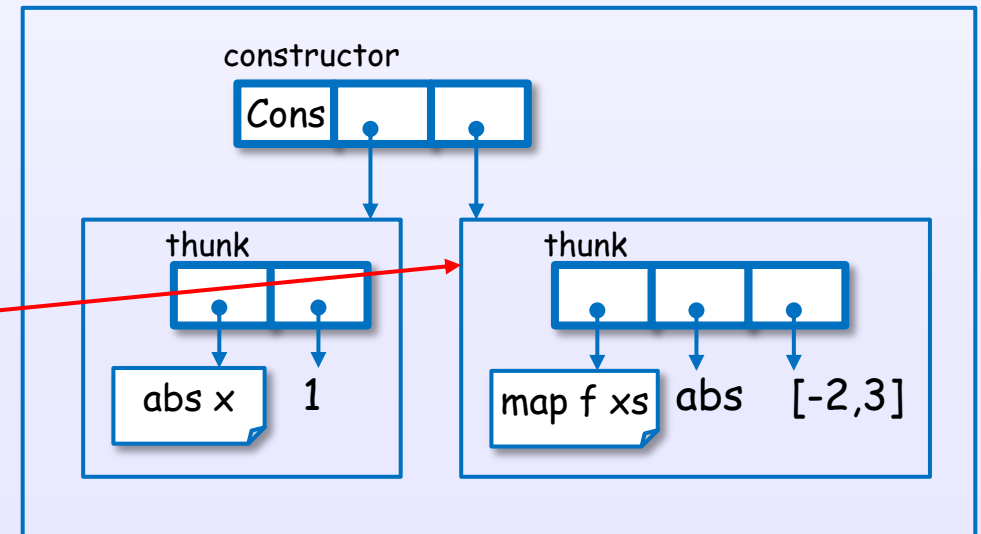
case thunk0 of  
(\_:xs) -> xs

case (abs 1) : (map abs [-2, 3]) of  
(\_:xs) -> xs

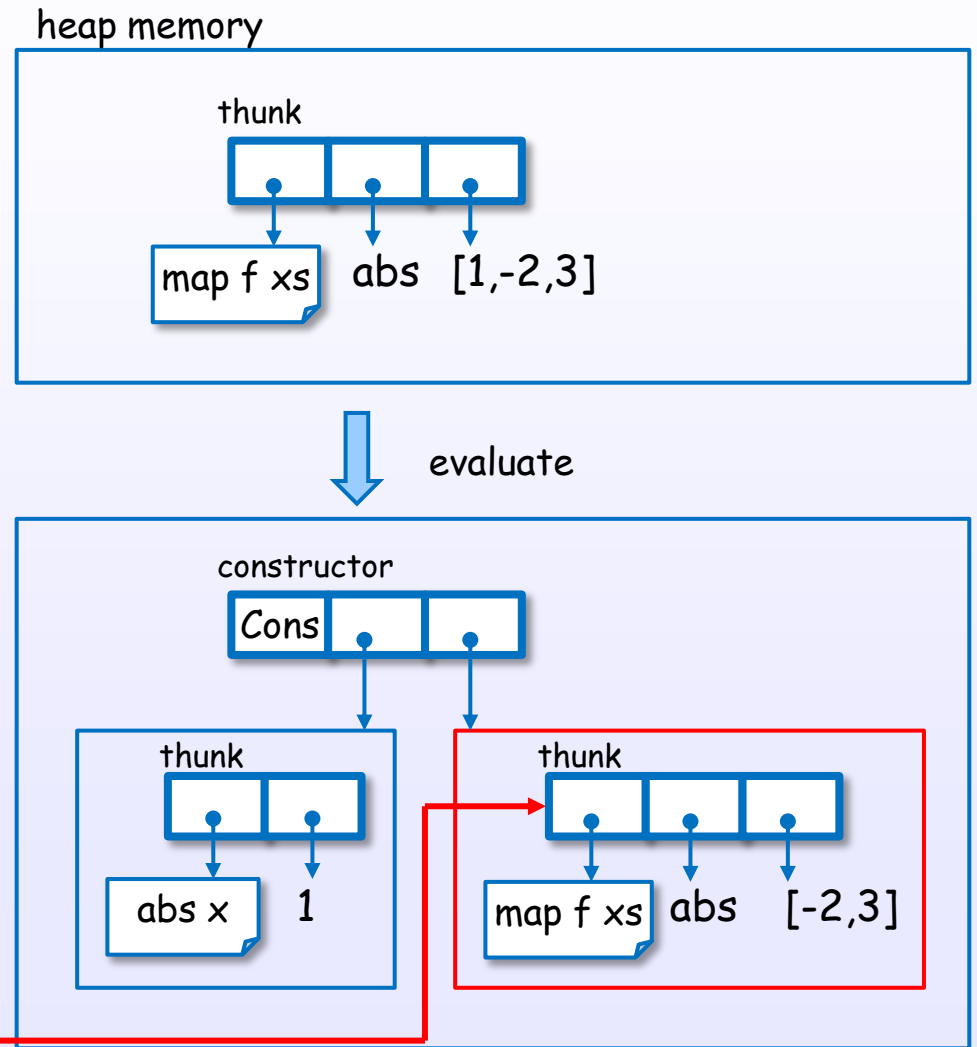
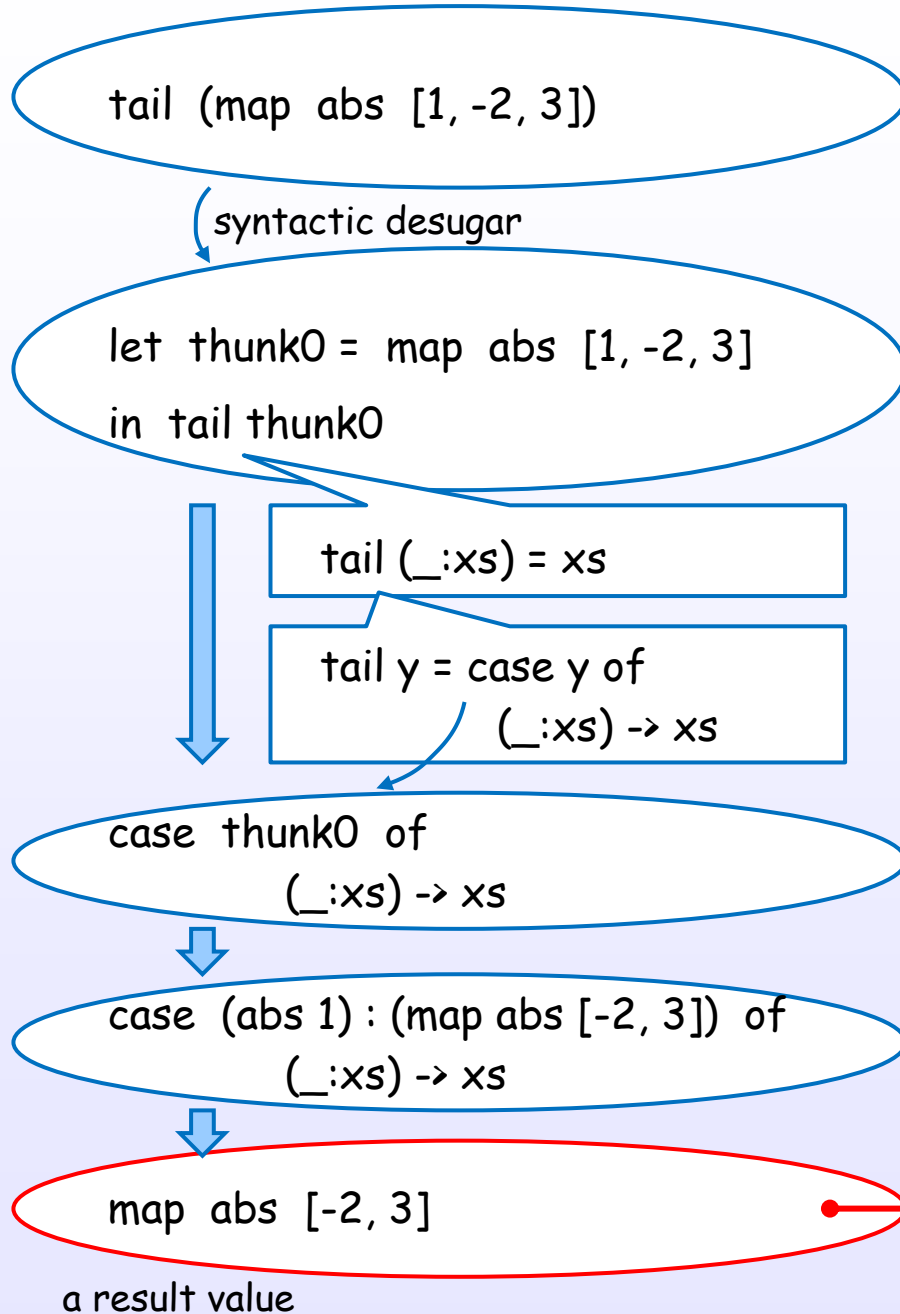
heap memory



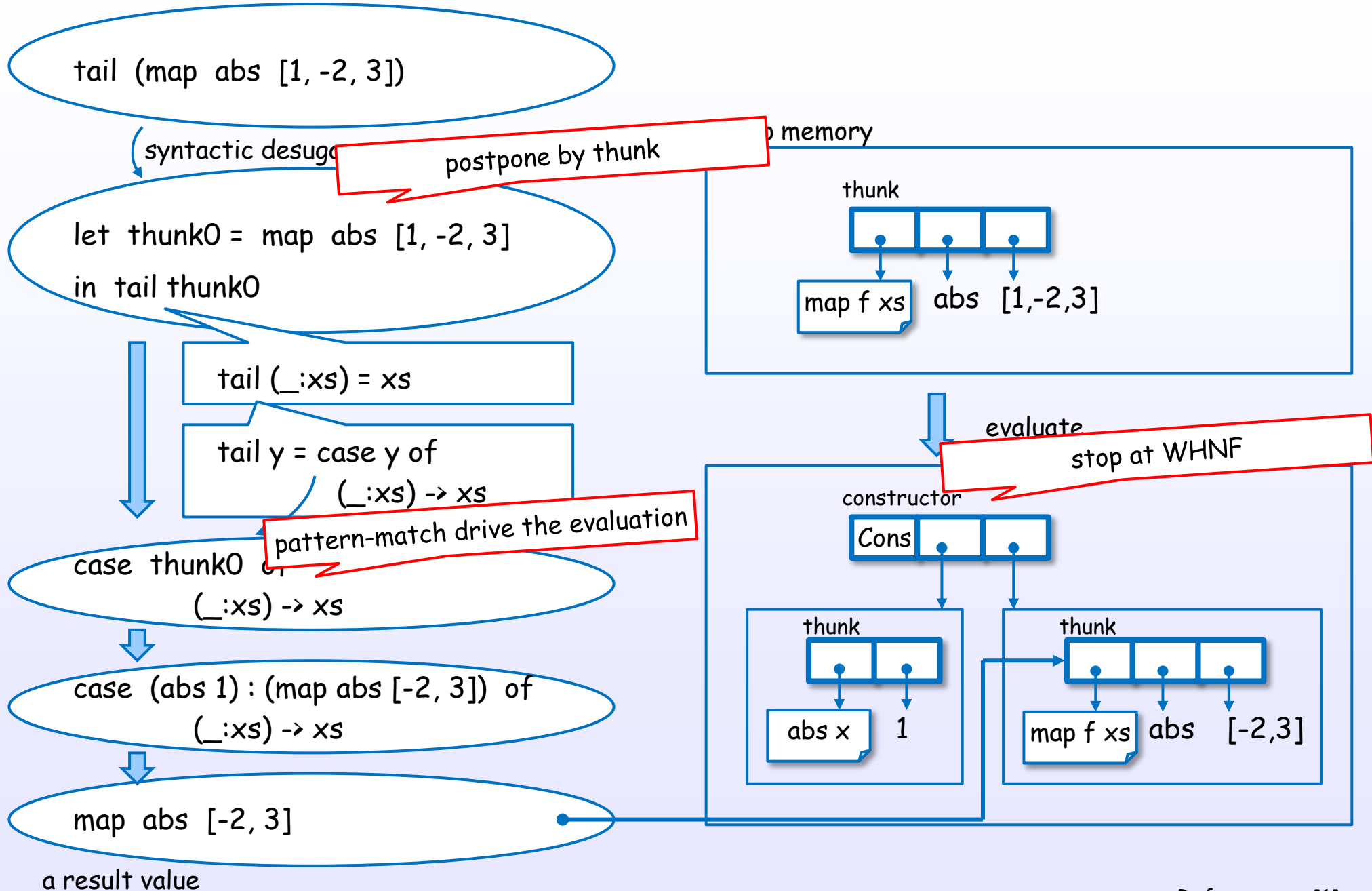
evaluate



## 12. return the value



# key points





# Pattern match

[CIS194]

# Pattern match

strict pattern

lazy pattern

case expression

function definition

let bounding pattern

Irrefutable Patterns

[stephen]

## 4. Evaluation

Examples of evaluation steps

# Example of repeat

repeat 1



1 : repeat 1



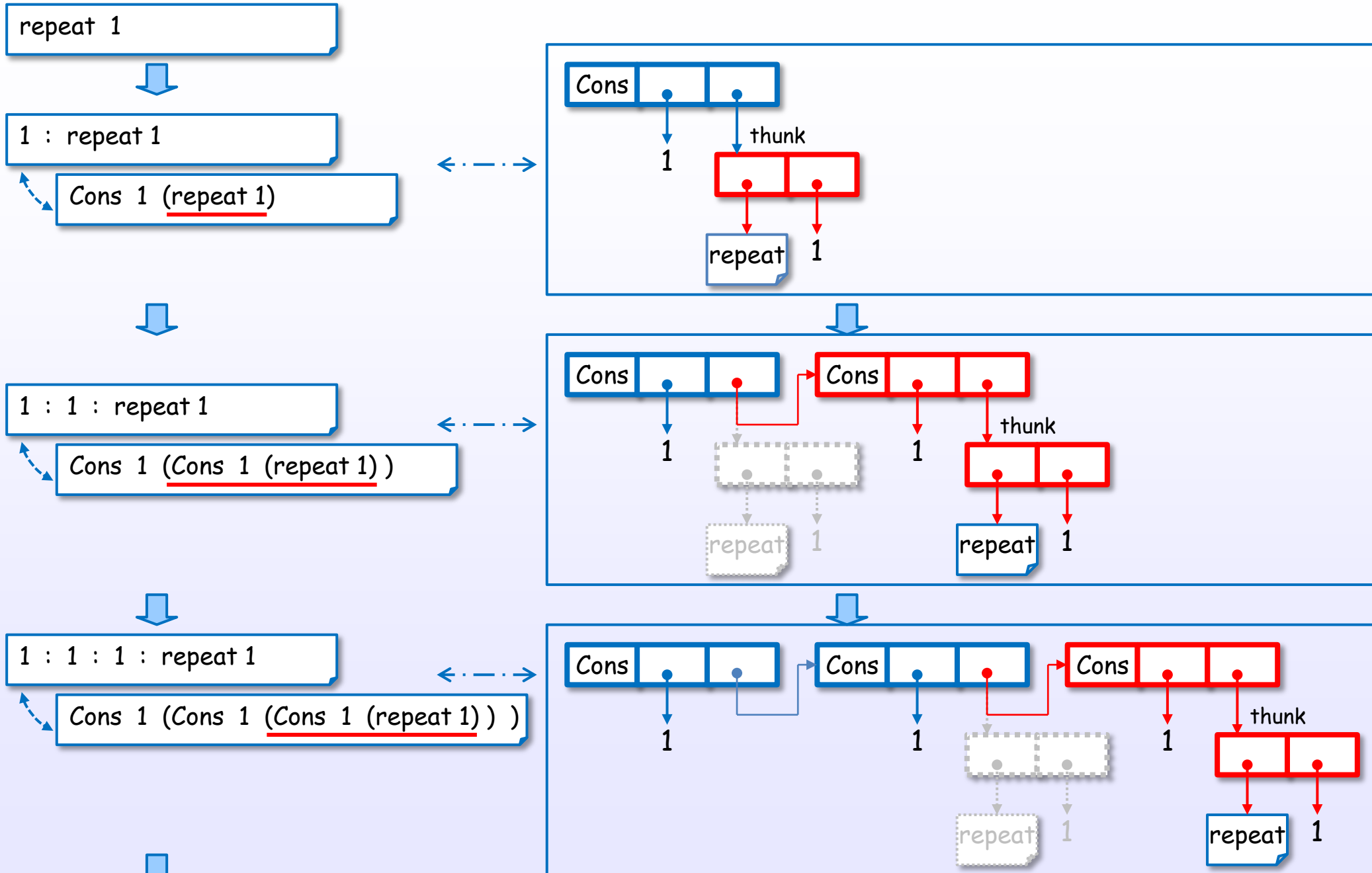
1 : 1 : repeat 1



1 : 1 : 1 : repeat 1



# Example of repeat



# Example of map

```
map f [1, 2, 3]
```



```
f 1 : map f [2, 3]
```



```
f 1 : f 2 : map f [3]
```



```
f 1 : f 2 : f 3
```



...

# Example of map

map f [1, 2, 3]



f 1 : map f [2, 3]



Cons (f 1) (map f [2, 3])



f 1 : f 2 : map f [3]



Cons (f 1) (Cons (f 2) (map f [3]))



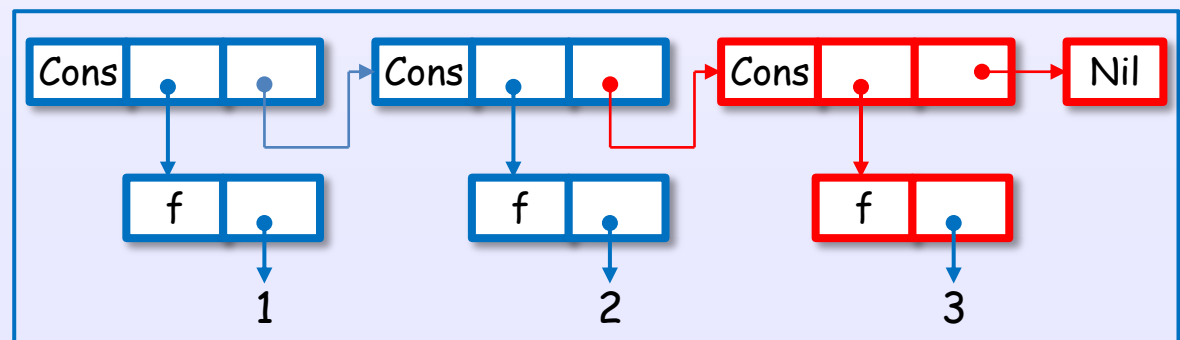
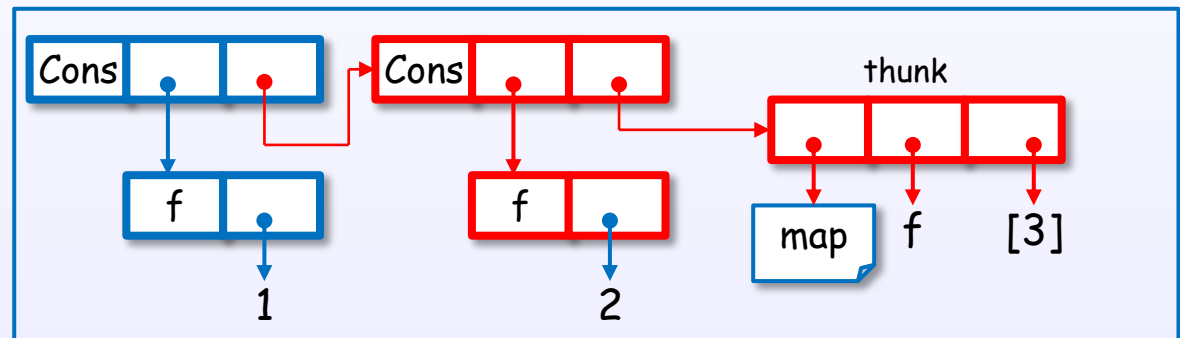
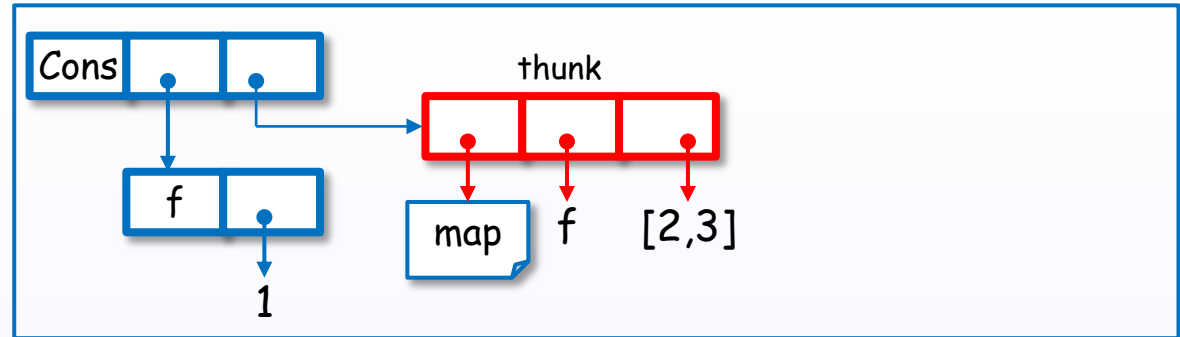
f 1 : f 2 : f 3



Cons (f 1) (Cons (f 2) (Cons (f 3) Nil))



...



...

# Example of foldl (non-strict)

`foldl (+) 0 [1 .. 100]`



`foldl (+) (0 + 1) [2 .. 100]`



`foldl (+) ((0 + 1) + 2) [3 .. 100]`



`foldl (+) ((((0 + 1) + 2) + 3) [4 .. 100]`



...



# Example of foldl (non-strict)

```
foldl (+) 0 [1 .. 100]
```



```
foldl (+) (0 + 1) [2 .. 100]
```

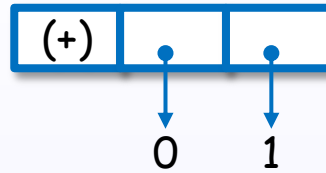
```
let thunk1 = (0 + 1)  
in foldl (+) thunk1 [2 .. 100]
```



heap memory

\*show only accumulation value

thunk1

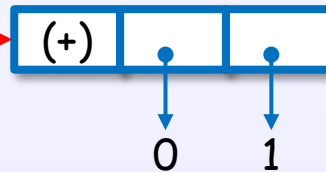


```
foldl (+) ((0 + 1) + 2) [3 .. 100]
```

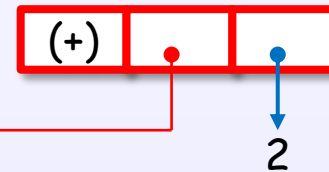
```
let thunk2 = (thunk1 + 2)  
in foldl (+) thunk2 [3 .. 100]
```



thunk1



thunk2

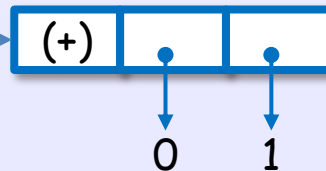


```
foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]
```

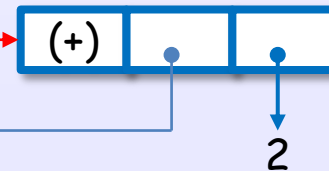
```
let thunk3 = (thunk2 + 3)  
in foldl (+) thunk3 [4 .. 100]
```



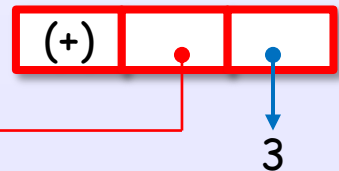
thunk1



thunk2



thunk3



increasing heap ...



...



# Example of foldl' (strict)

foldl' (+) 0 [1 .. 100]



foldl' (+) (0 + 1) [2 .. 100]



foldl' (+) (1 + 2) [3 .. 100]



foldl' (+) (3 + 3) [4 .. 100]



...

# Example of foldl' (strict)

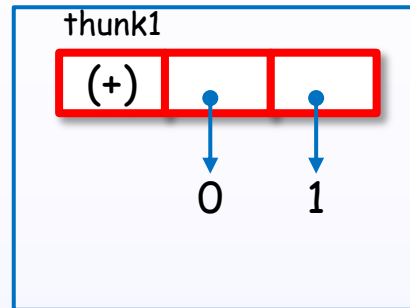
foldl' (+) 0 [1 .. 100]



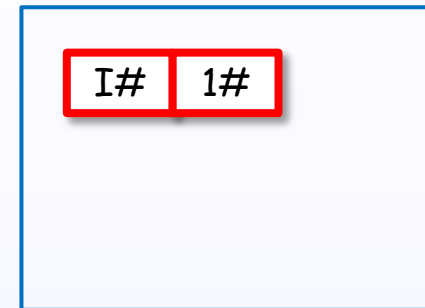
foldl' (+) (0 + 1) [2 .. 100]

let thunk1 = (0 + 1)  
in thunk1 `pseq`  
foldl' (+) thunk1 [2 .. 100]

heap memory

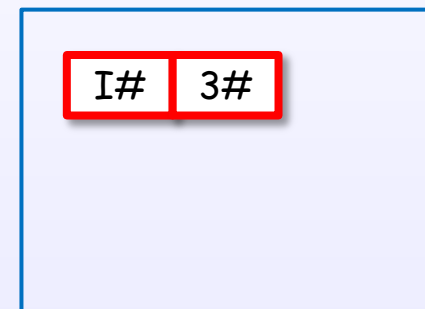
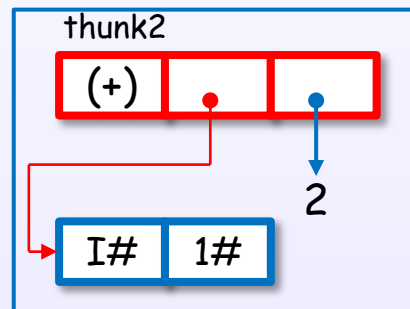


update  
by pseq



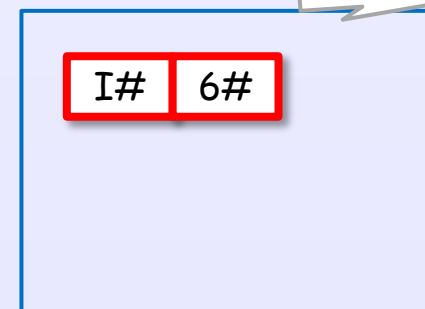
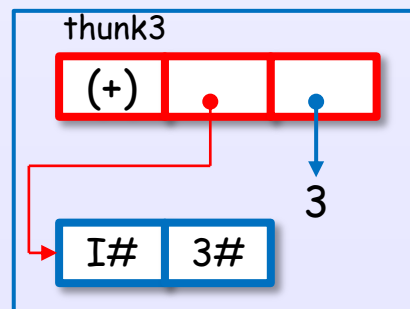
foldl' (+) (1 + 2) [3 .. 100]

let thunk2 = (1 + 2)  
in thunk2 `pseq`  
foldl' (+) thunk2 [3 .. 100]



foldl' (+) (3 + 3) [4 .. 100]

let thunk3 = (3 + 3)  
in thunk3 `pseq`  
foldl' (+) thunk3 [4 .. 100]



fixed heap size



...

References : [1]

# Example of foldl (non-strict) and foldl' (strict)

foldl (+) (0 + 1) [2 .. 100]



foldl (+) ((0 + 1) + 2) [3 .. 100]



foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]



foldl' (+) (0 + 1) [2 .. 100]



foldl' (+) (1 + 2) [3 .. 100]



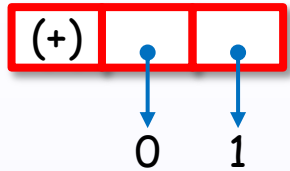
foldl' (+) (3 + 3) [4 .. 100]



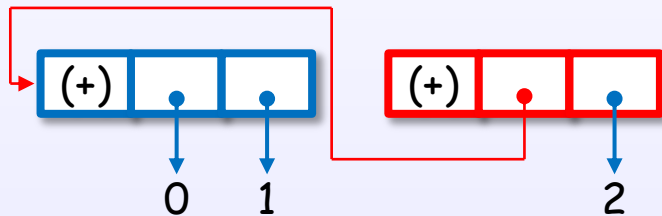
# Example of foldl (non-strict) and foldl' (strict)

foldl (+) (0 + 1) [2 .. 100]

heap memory

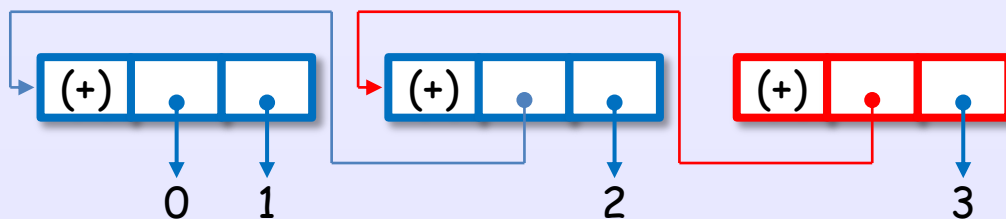


foldl (+) ((0 + 1) + 2) [3 .. 100]

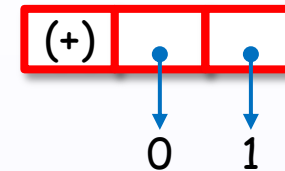


foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]

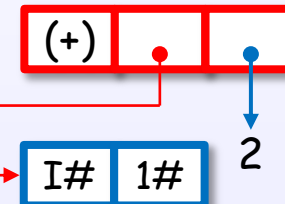
increasing heap ...



foldl' (+) (0 + 1) [2 .. 100]

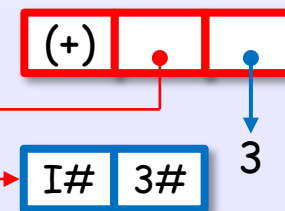


foldl' (+) (1 + 2) [3 .. 100]



foldl' (+) (3 + 3) [4 .. 100]

fixed heap size



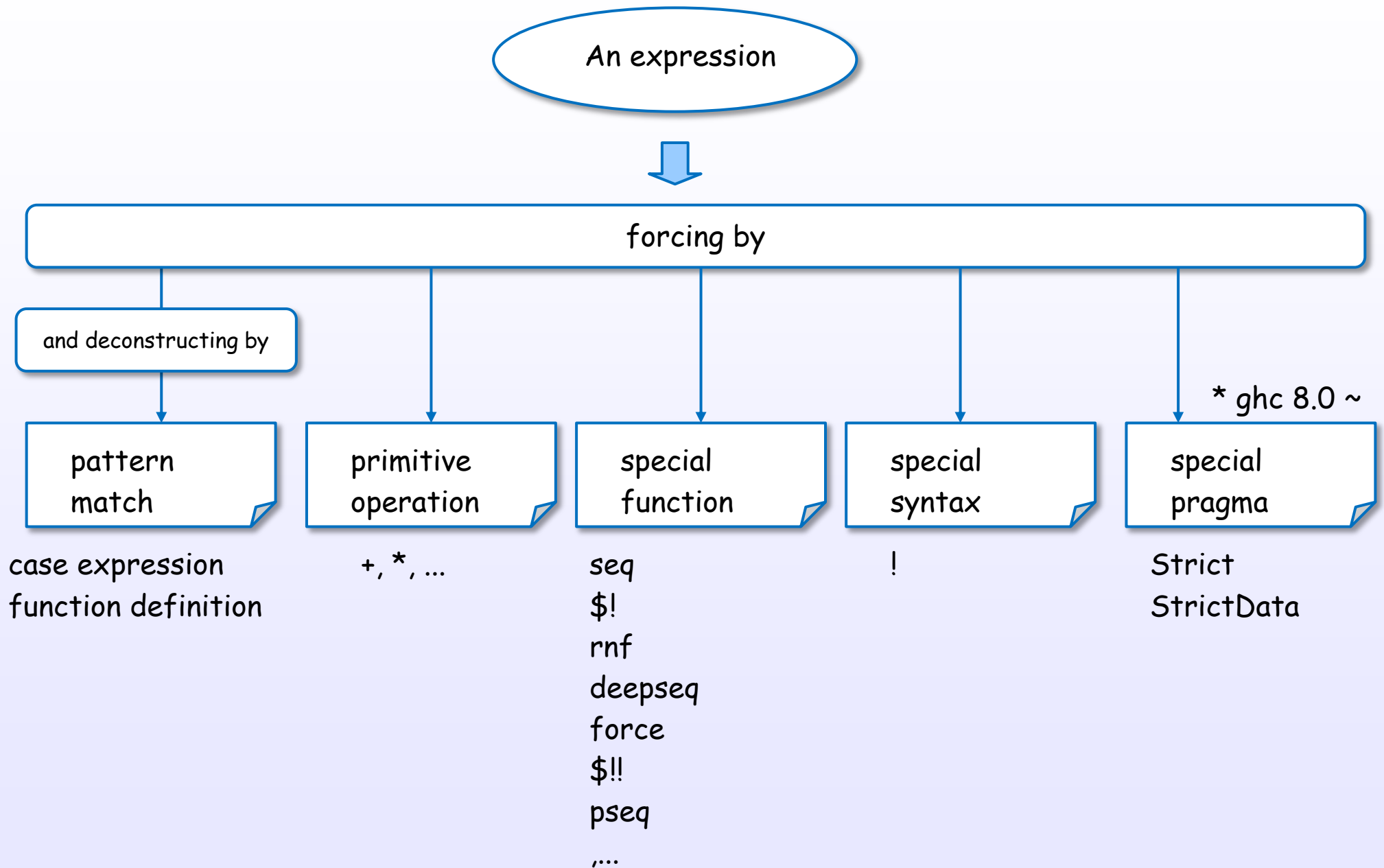
# Example of nest-function

```
take 5 ( map f xs )
```

## 4. Evaluation

Controlling the evaluation

# How to drive evaluation





# Example of the evaluation by pattern match

case expression

```
case ds of
  x:xs -> f x xs
  []    -> False
```

case expression  
function definition

# Example of the evaluation by primitive operation

primitive operation

$$f \ x \ y = x \text{ + } y$$

$+, *, \dots$

# Example of the evaluation by special function

special function

$f \times y = \text{seq} \times y$

seq  
\$!  
rnf  
deepseq  
force  
\$!!  
pseq  
...

to WHNF  
to NF

[parconc, Ch.2]

[RWH, Ch.24-25]

Please refer the document more detail. [xx]  
hoogle or hackage

[stephen]

[hack.hands]

[Bird, Chapter 7]

[CIS194]

References : [1]

# Example of the evaluation by special function

seq のObject図イメージ

force

rnf

rwhnf

deepseq のObject図イメージ

# Example of the evaluation by special function

表で整理

to WHNF

seq

rwhnf

pseq

\$!

to NF

force

rnf

deepseq

\$!!

# Example of the evaluation by special syntax

special syntax

```
{-# LANGUAGE BangPatterns #-}
```

```
f !xs = g xs
```

BangPattern

```
{-# LANGUAGE BangPatterns #-}
```

```
data ...
```

[RWH, Ch.25]

[stephen]

Please refer the document more detail. [xx]

[user guide, 7.19]

# Example of the evaluation by special pragma

special pragma

```
{-# LANGUAGE Strict #-}
```

```
f xs = g xs
```

\* ghc 8.0 ~

```
{-# LANGUAGE StrictData #-}
```

```
f xs = g xs
```

Strict  
StrictData

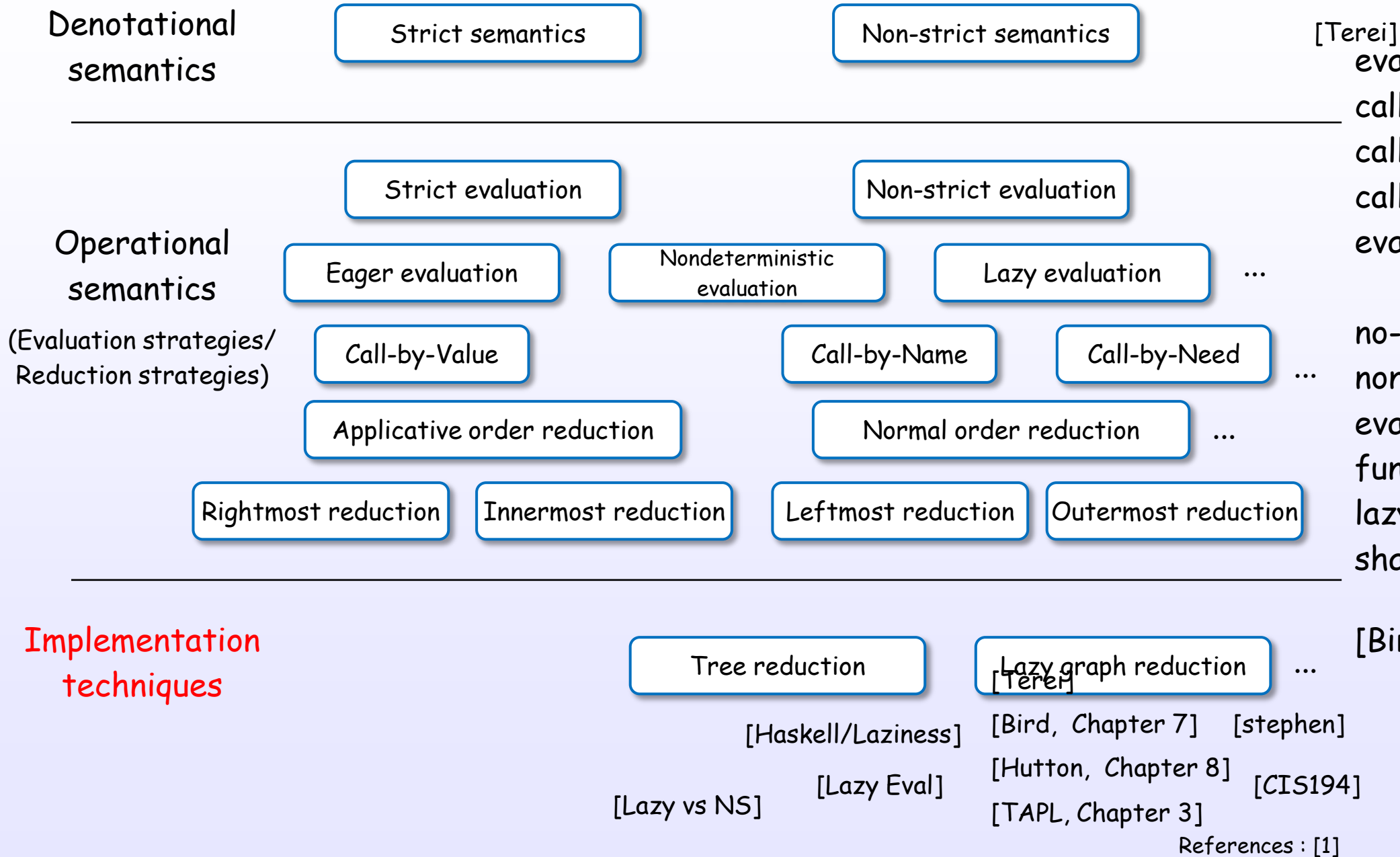
Please refer the document more detail. [xx]

[wiki]

## 5. Implementation of evaluator



# Evaluation layer for GHC's Haskell

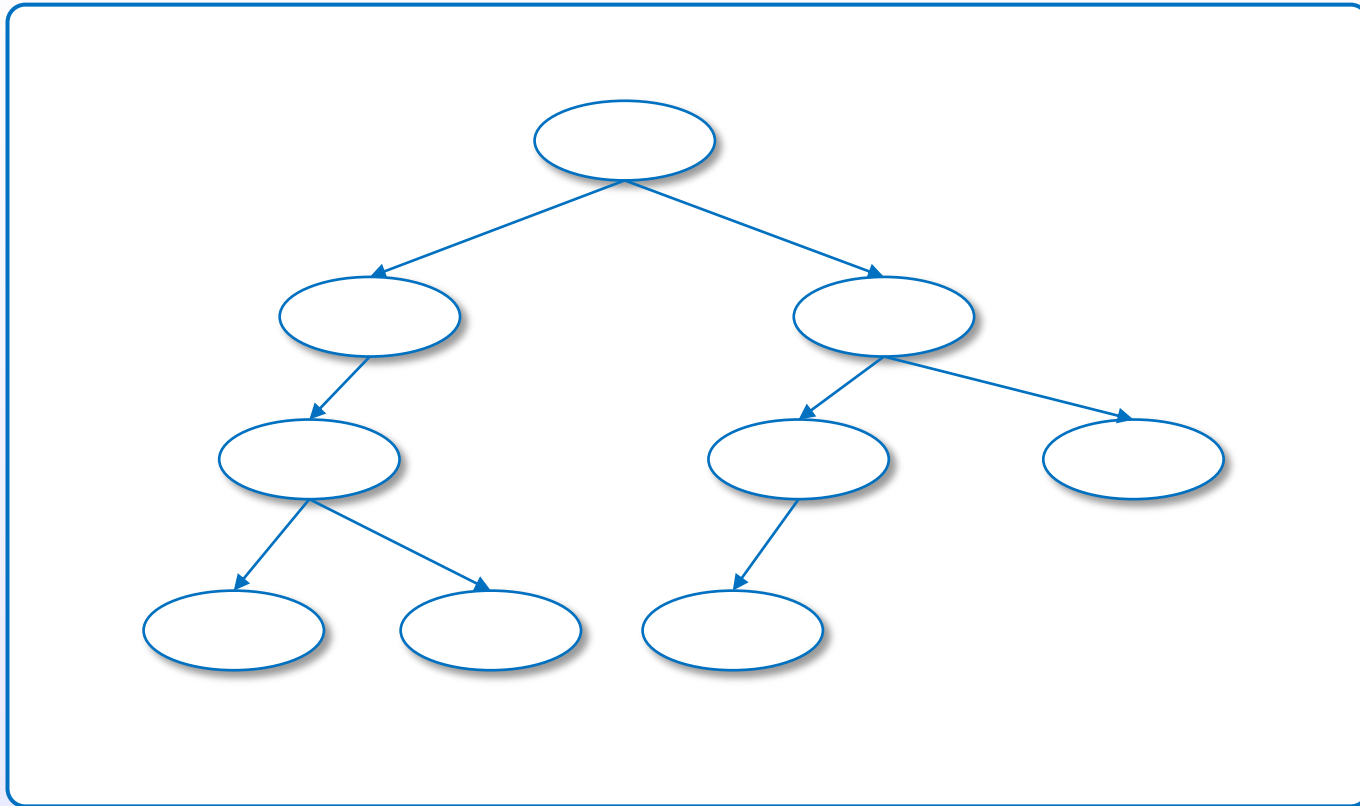


## 5. Implementation of evaluator

Lazy graph reduction

# Tree

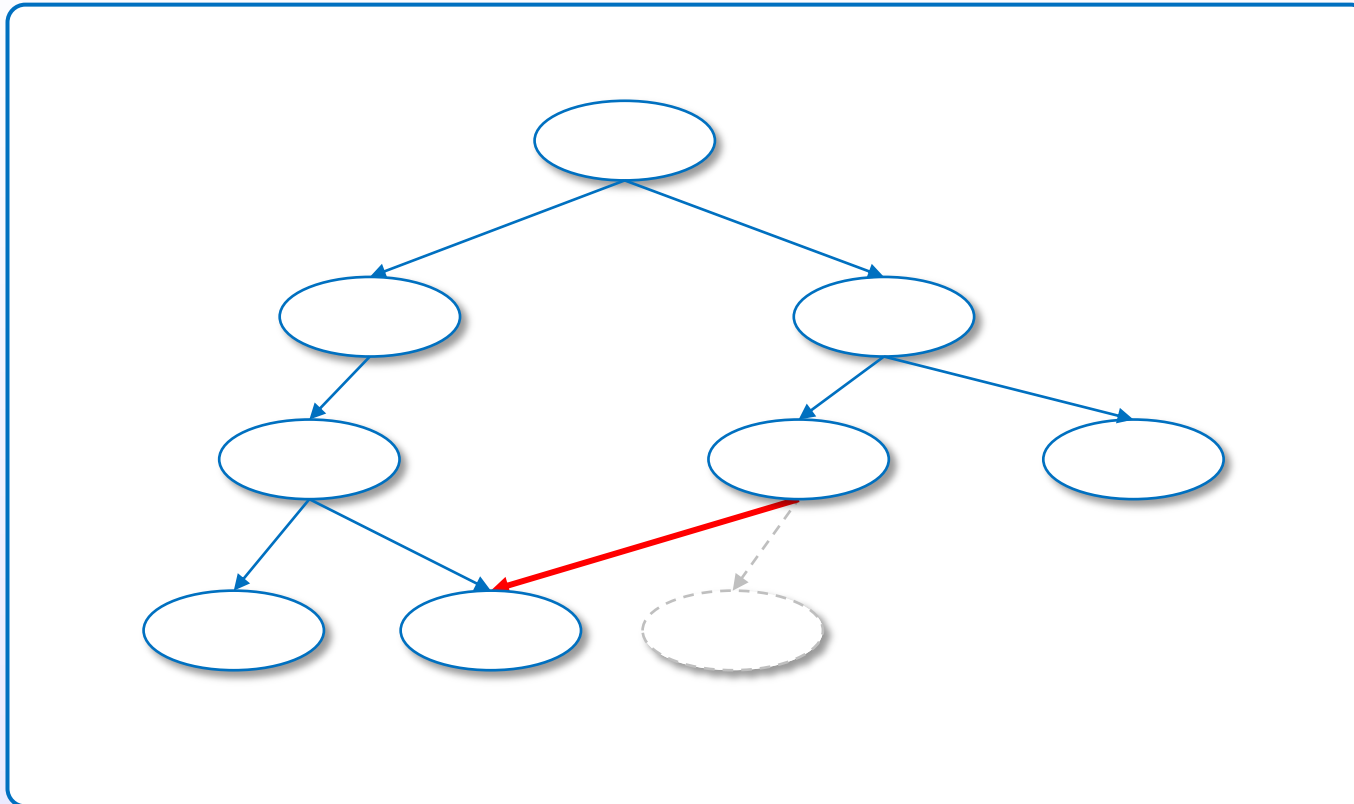
AST represents an expression



Stack base

# Graph

Share the term, looped  
not Tree, but Graph



Heap base

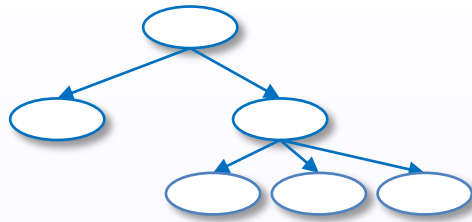
[Terei]

[hack.hands]

[CIS194]

References : [1]

# Tree and graph reduction



Tree reduction



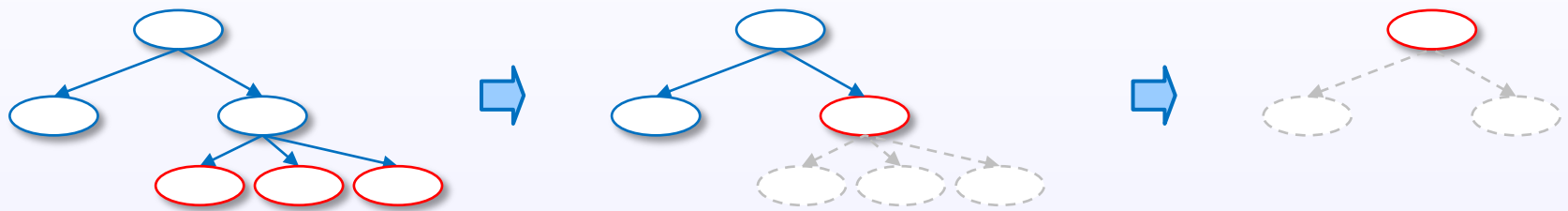
Graph reduction



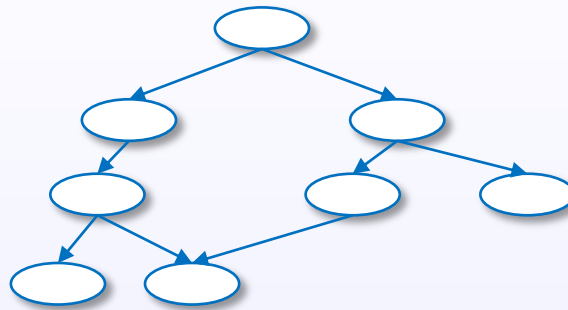
copy arguments

share arguments by pointers

# Graph reduction



# Graph reduction and lazy



## 5. Implementation of evaluator

STG-machine



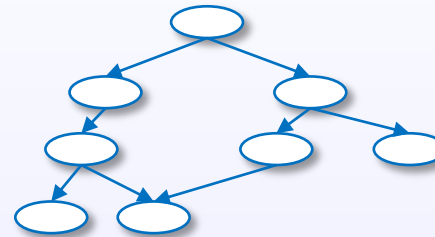
# Abstract machine

# Layer

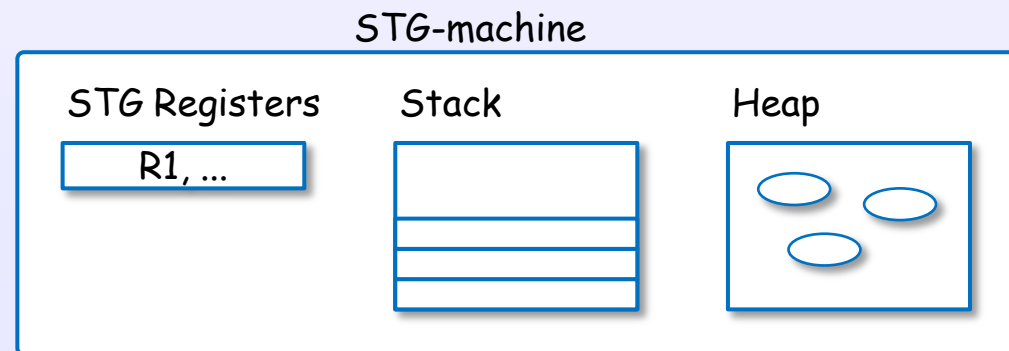
Haskell code

take 5 [1..10]

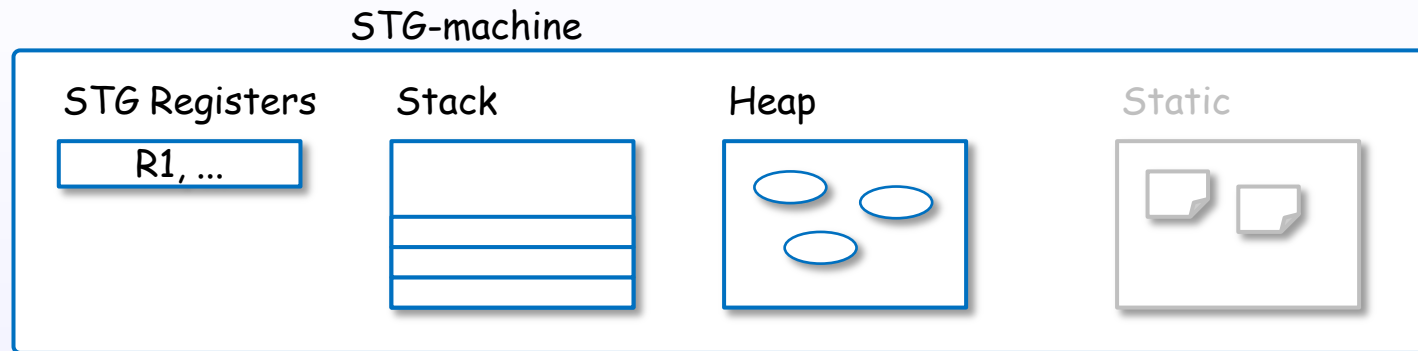
Internal representation  
by graph



Evaluation (execution, reduction)  
by STG-machine



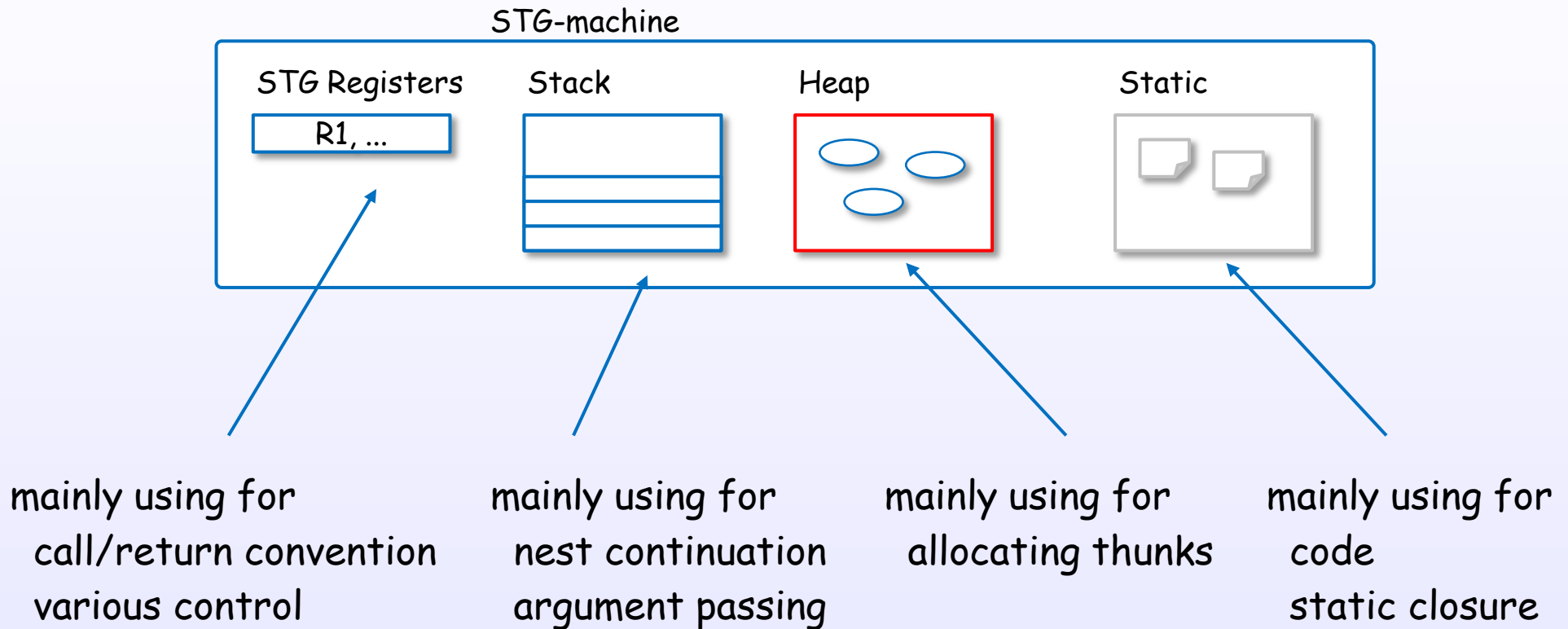
# STG-machine



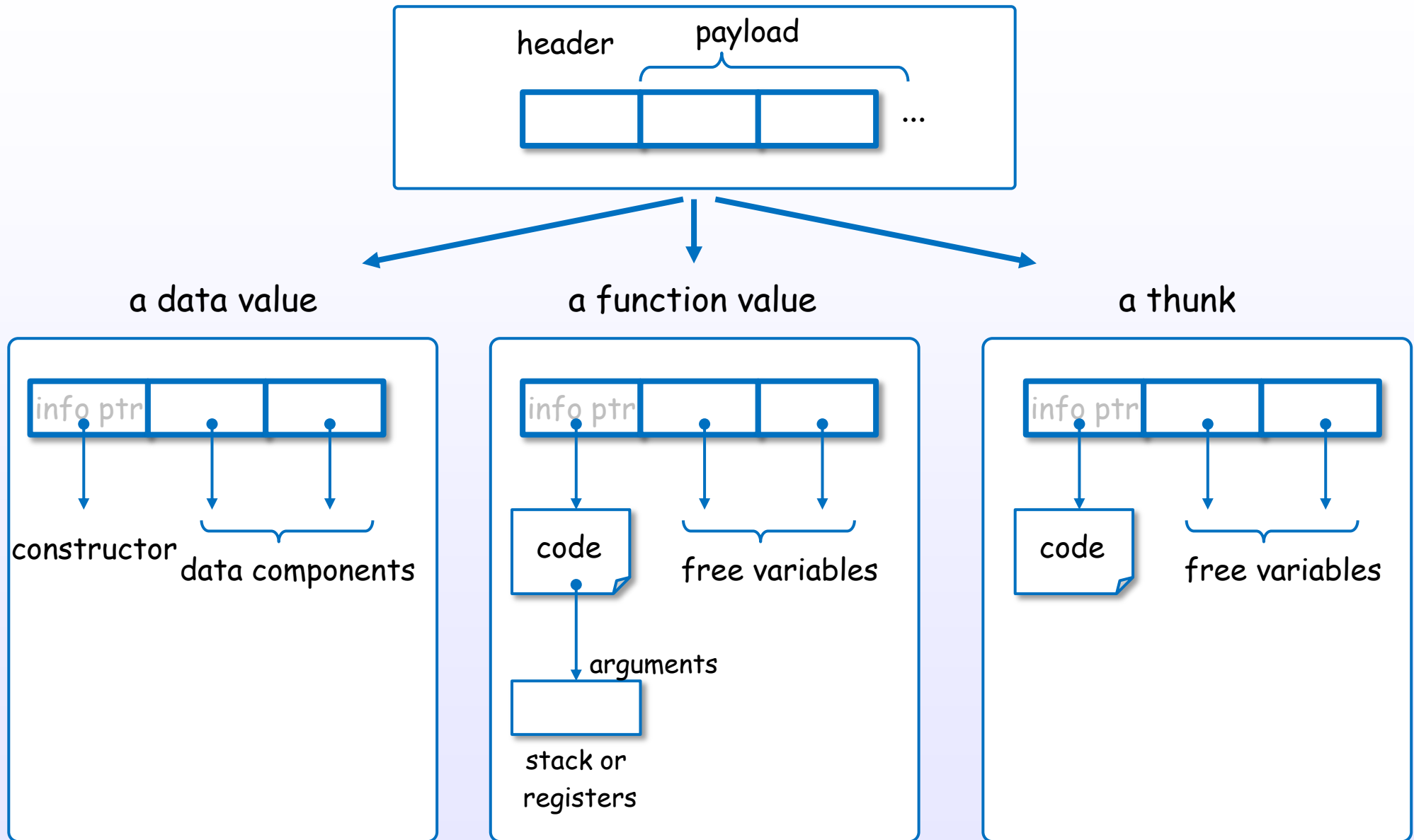
STG-machine is abstraction machine  
which is defined by operational semantics.

STG-machine efficiently performs lazy graph reduction.

# STG-machine



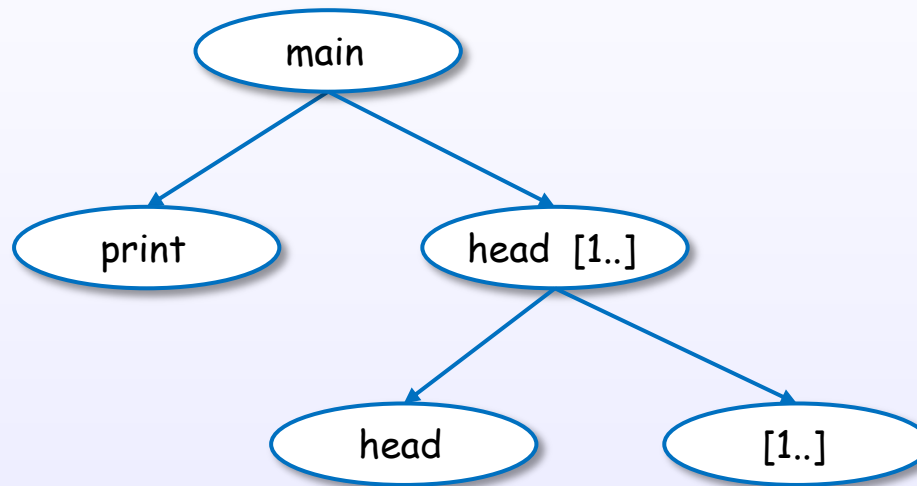
# An unified representation in {heap, stack, static} memory



いずれも、広義の、“closure” (= code + environment(free variables))

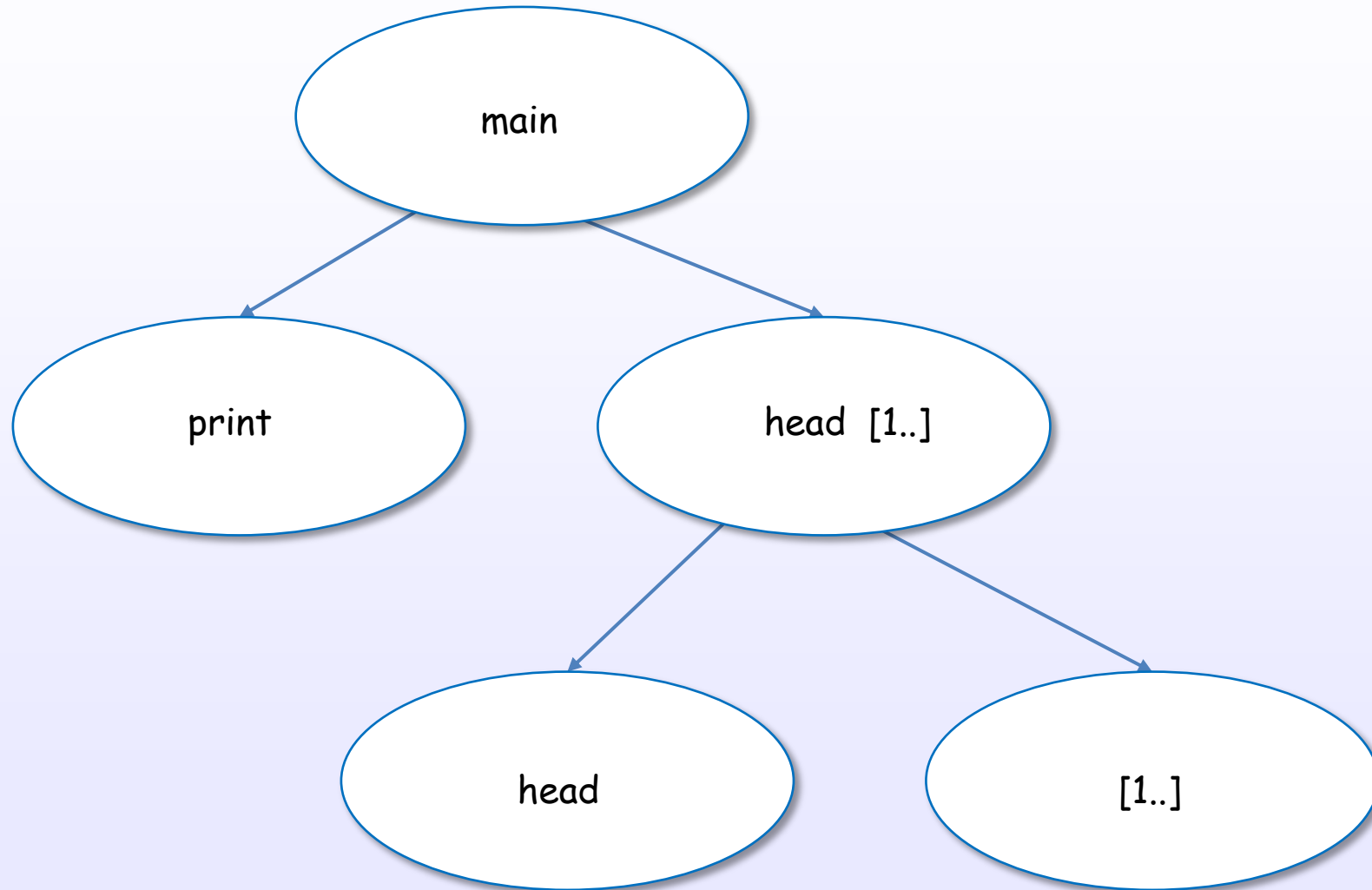
# Mapping the graph to the code

```
main = print (head [1..])
```



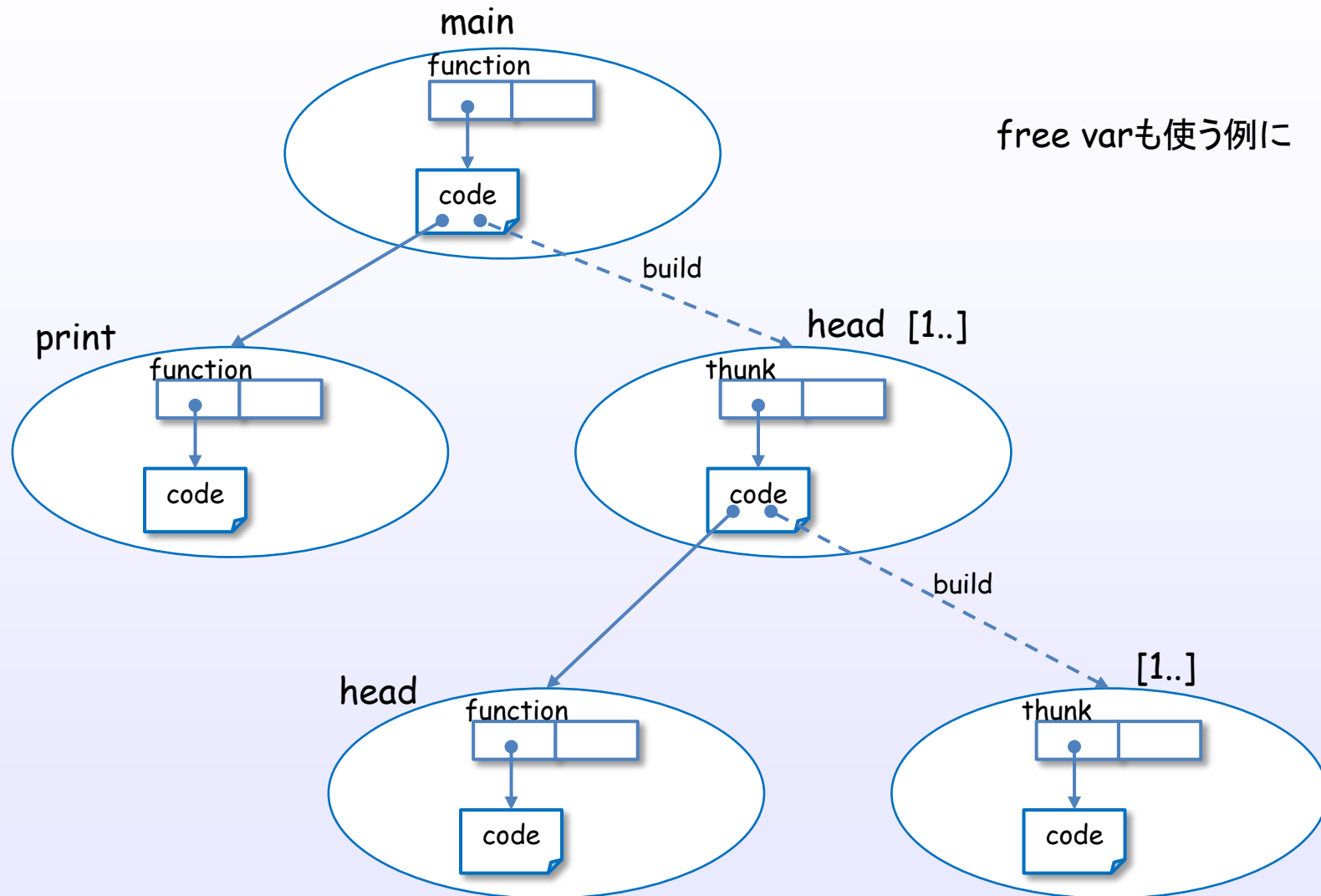
# Mapping the graph to the code

```
main = print (head [1..])
```



# Mapping the graph to the code

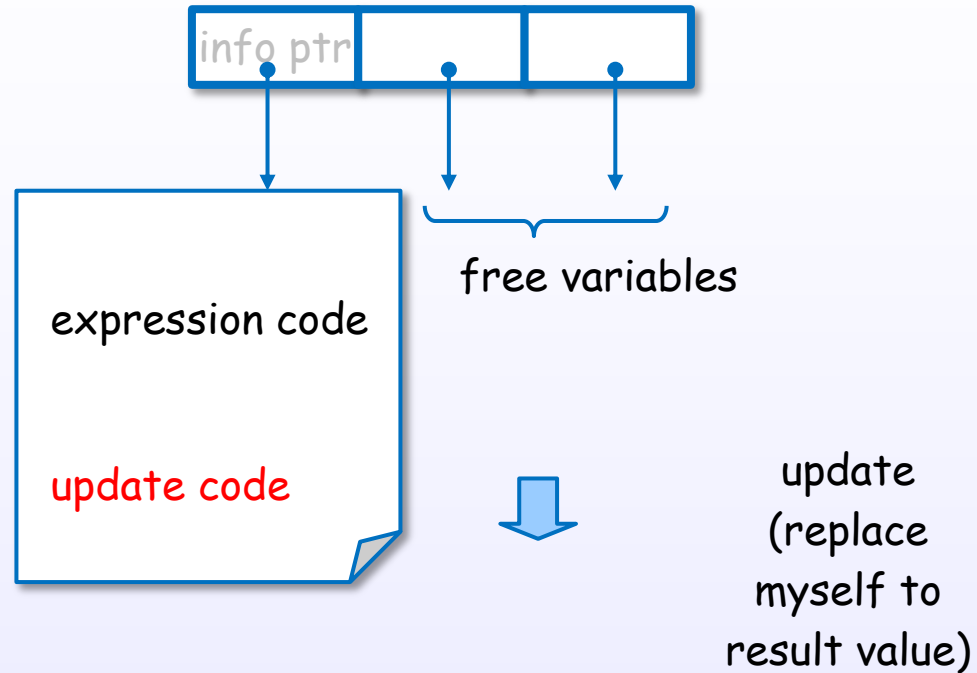
main = print (head [1..])



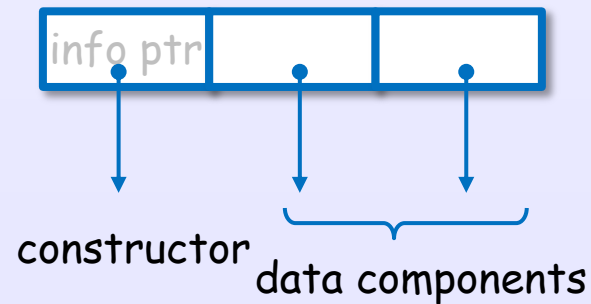


# Self-updating model

a thunk



a data value



## 6. Semantics

# Evaluation layer for GHC's Haskell

Denotational  
semantics

Strict semantics

Non-strict semantics

[Terei]

eva

call

call

call

eva

Operational  
semantics

Strict evaluation

Non-strict evaluation

Eager evaluation

Nondeterministic  
evaluation

Lazy evaluation

...

(Evaluation strategies/  
Reduction strategies)

Call-by-Value

Call-by-Name

Call-by-Need

...

Applicative order reduction

Normal order reduction

...

Rightmost reduction

Innermost reduction

Leftmost reduction

Outermost reduction

no-

non

eva

fun

lazy

sha

Implementation  
techniques

Tree reduction

Lazy graph reduction

...

[Bir

[Haskell/Laziness]

[Bird, Chapter 7] [stephen]

[Hutton, Chapter 8]

[CIS194]

[TAPL, Chapter 3]

[Lazy vs NS]

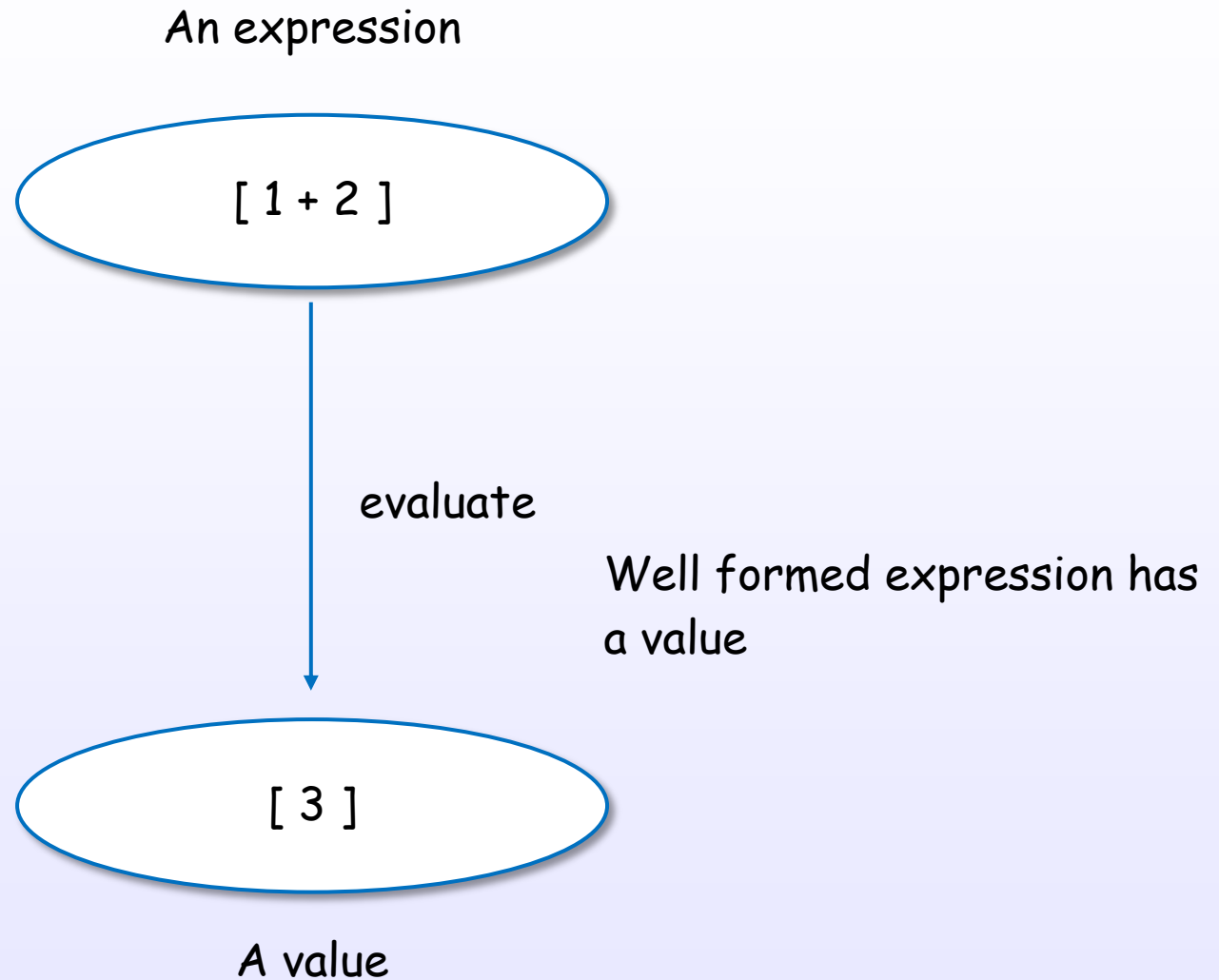
[Lazy Eval]

References : [1]

## 6. Semantics

Bottom

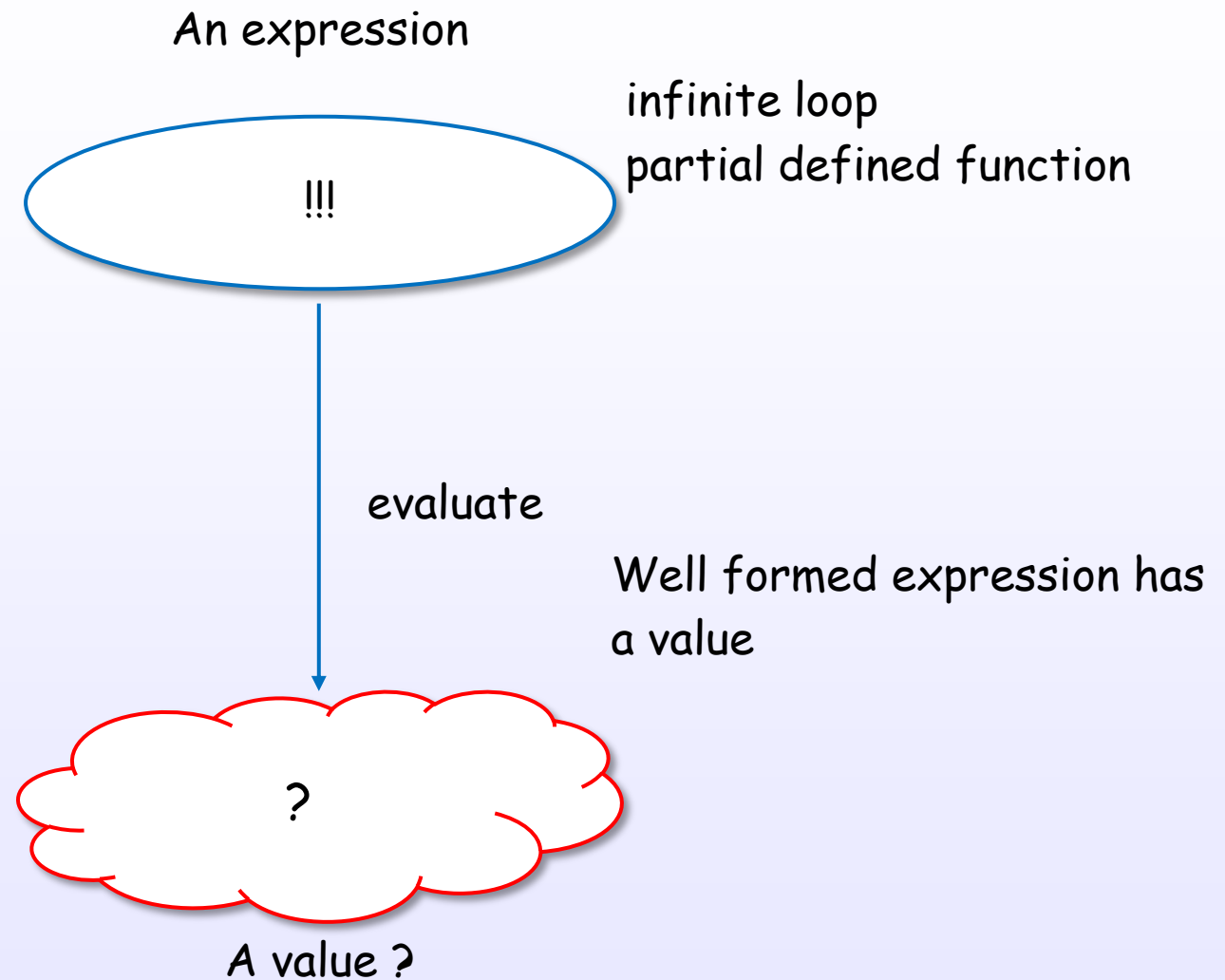
# Well formed expression has a value



[Bird, Chapter 2]

References : [1]

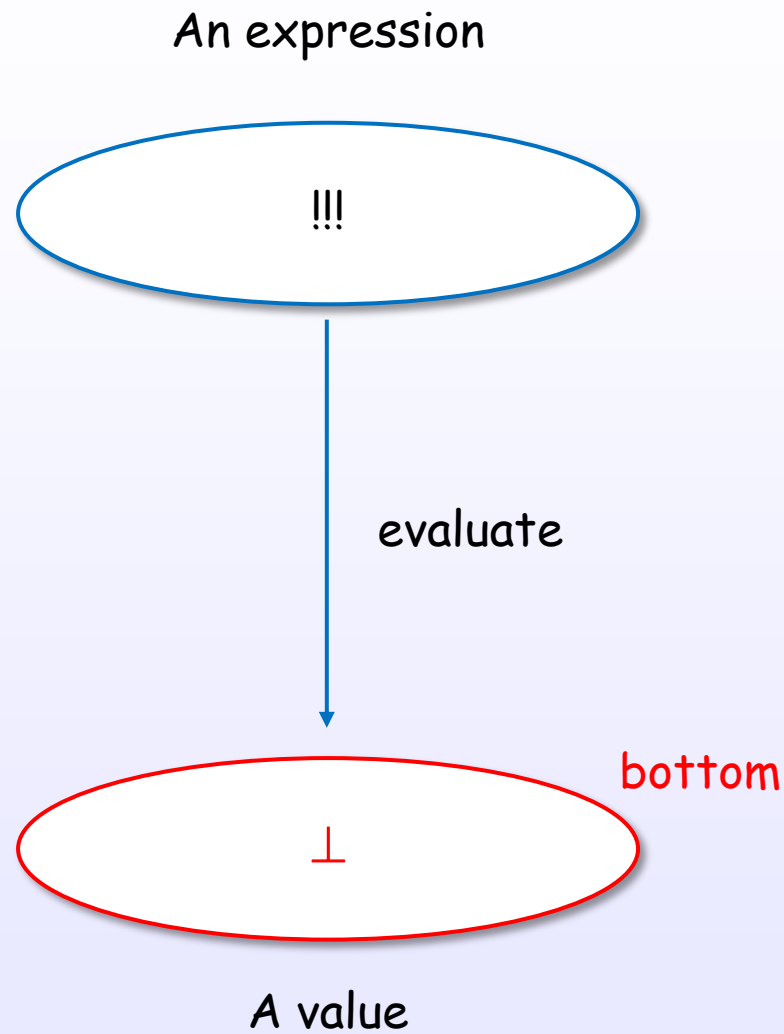
# Well formed expression has a value



[Bird, Chapter 2]

References : [1]

# Well formed expression has a value



[Bird, Chapter 2]

References : [1]

# Bottom

[Bird, Chapter 2]

References : [1]



## 6. Semantics

### Non-strict Semantics

# Strictness

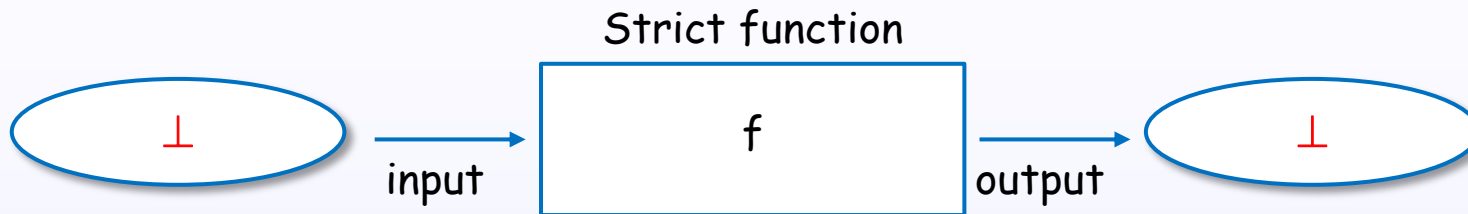
$$f \perp = \perp$$

Strictness is attribution of the function.

[Bird, Chapter 2]

# Strictness

$$f \perp = \perp$$



Strictness is attribution of the function.

[Bird, Chapter 2]

# Strictness and Non-strictness

Strict

$$f \perp = \perp$$

---

Non-strict

$$f \perp \neq \perp$$

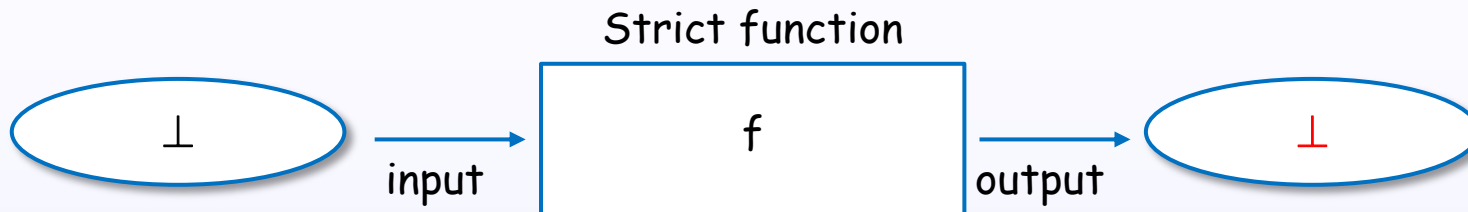
[Bird, Chapter 2]

References : [1]

# Strictness and Non-strictness

Strict

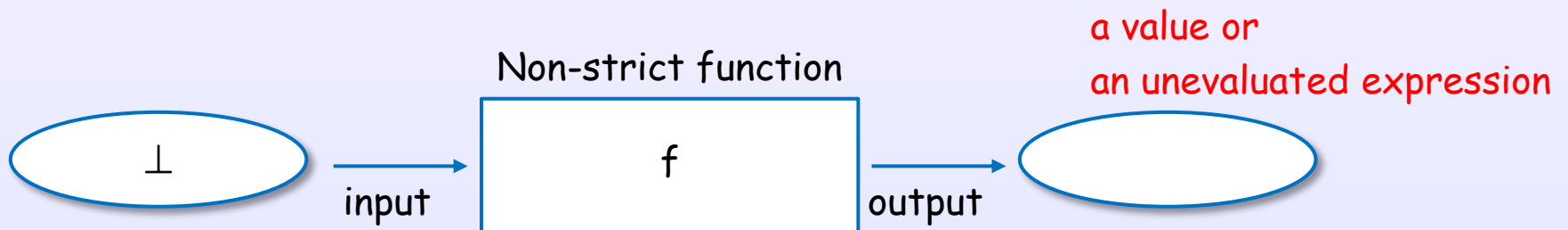
$$f \perp = \perp$$



---

Non-strict

$$f \perp \neq \perp$$



[Bird, Chapter 2]

# Layer

Non-strictness

$$f \perp = \perp$$

Lazy evaluation

*GHC* chosen lazy evaluation to implement non-strict semantics.

Graph reduction

*GHC* chosen graph reduction to implement lazy evaluation.

STG-machine

*GHC* implements graph reduction by STG-machine.

# seq and pseq

$\text{seq } a \ b = \perp, \quad \text{if } a = \perp$   
 $= b, \quad \text{otherwise}$

$\text{pseq } a \ b = \perp, \quad \text{if } a = \perp$   
 $= b, \quad \text{otherwise}$

$\text{seq } a \ \perp = \perp$   
 $\text{seq } \perp \ b = \perp$

$a$  is strict  
 $b$  is strict

$\text{pseq } a \ \perp = \perp$   
 $\text{pseq } \perp \ b \neq \perp$

$a$  is strict  
 $b$  is non-strict

[Runtime Support for Multicore Haskell]

[Snoyman]

## 6. Semantics

Strict analysis



# Strict analysis

## 7. Appendix

## 7. Appendix

### References

# References

- [H1] Haskell 2010 Language Report  
<https://www.haskell.org/definition/haskell2010.pdf>
- [H2] The Glorious Glasgow Haskell Compilation System (GHC user's guide)  
[https://downloads.haskell.org/~ghc/latest/docs/users\\_guide.pdf](https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf)
- [H3] A History of Haskell: Being Lazy With Class  
<http://haskell.cs.yale.edu/wp-content/uploads/2011/02/history.pdf>
- [H4] The implementation of functional programming languages  
<http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/slpj-book-1987.pdf>
- [H5] Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5  
<http://research.microsoft.com/en-us/um/people/simonpj/Papers/spineless-tagless-gmachine.ps.gz>
- [H6] Making a Fast Curry Push/Enter vs Eval/Apply for Higher-order Languages  
<http://research.microsoft.com/en-us/um/people/simonpj/papers/eval-apply>
- [H7] Runtime Support for Multicore Haskell  
<http://community.haskell.org/~simonmar/papers/multicore-ghc.pdf>
- [H8] I know kung fu: learning STG by example  
<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/GeneratedCode>
- [H9] GHC Commentary: The Layout of Heap Objects  
<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects>
- [H10] GHC Commentary: Strict & StrictData  
<https://ghc.haskell.org/trac/ghc/wiki/StrictPragma>

# References

- [B1] Introduction to Functional Programming using Haskell (IFPH 2nd edition)  
<http://www.cs.ox.ac.uk/publications/books/functional/bird-1998.jpg>  
<http://www.pearsonhighered.com/educator/product/Introduction-Functional-Programming/9780134843469.page>
- [B2] Thinking Functionally with Haskell (IFPH 3rd edition)  
<http://www.cs.ox.ac.uk/publications/books/functional/>
- [B3] Programming in Haskell  
<https://www.cs.nott.ac.uk/~gmh/book.html>
- [B4] Real World Haskell  
<http://book.realworldhaskell.org/>
- [B5] Parallel and Concurrent Programming in Haskell  
<http://chimera.labs.oreilly.com/books/12300000000929>
- [B6] Types and Programming Languages (TAPL)  
<https://mitpress.mit.edu/books/types-and-programming-languages>
- [B7] Purely Functional Data Structures  
<http://www.cambridge.org/us/academic/subjects/computer-science/programming-languages-and-applied-logic/purely-functional-data-structures>
- [B8] Algorithms: A Functional Programming Approach  
<http://catalogue.pearsoned.co.uk/catalog/academic/product/0,1144,0201596040,00.html>

# References

- [D1] Laziness  
<http://dev.stephendiehl.com/hask/#laziness>
- [D2] Being Lazy with Class  
<http://www.seas.upenn.edu/~cis194/lectures/06-laziness.html>
- [D3] A Haskell Compiler  
<http://www.scs.stanford.edu/14sp-cs240h/slides/ghc-compiler-slides.html>  
<http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html>
- [D4] Evaluation  
[http://dev.stephendiehl.com/fun/005\\_evaluation.html](http://dev.stephendiehl.com/fun/005_evaluation.html)
- [D5] Incomplete Guide to e Lazy Evaluation (in Haskell)  
<https://hackhands.com/guide-lazy-evaluation-haskell>
- [D6] Laziness  
<https://www.fpcomplete.com/school/starting-with-haskell/introduction-to-haskell/6-laziness>
- [D7] Evaluation on the Haskell Heap  
<http://blog.ezyang.com/2011/04/evaluation-on-the-haskell-heap>
- [D8] Fixing foldl  
<http://www.well-typed.com/blog/2014/04/fixing-foldl>
- [D9] How to force a list  
<https://ro-che.info/articles/2015-05-28-force-list>
- [D10] Evaluation order and state tokens  
<https://www.fpcomplete.com/user/snoyberg/general-haskell/advanced/evaluation-order-and-state-tokens>

# References

- [D11] Reasoning about laziness  
<http://blog.johantibell.com/2011/02/slides-from-my-talk-on-reasoning-about.html>
- [D12] Some History of Functional Programming Languages  
[http://www-fp.cs.st-andrews.ac.uk/tifp/TFP2012/TFP\\_2012/Turner.pdf](http://www-fp.cs.st-andrews.ac.uk/tifp/TFP2012/TFP_2012/Turner.pdf)
- [D13] Why Functional Programming Matters  
<https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf>
- [D14] *GHC* illustrated  
[http://takenobu-hs.github.io/downloads/haskell\\_ghc\\_illustrated.pdf](http://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf)

# References

- [W1] Haskell/Laziness  
<https://en.wikibooks.org/wiki/Haskell/Laziness>
- [W2] Lazy evaluation  
[https://wiki.haskell.org/Lazy\\_evaluation](https://wiki.haskell.org/Lazy_evaluation)
- [W3] Lazy vs. non-strict  
[https://wiki.haskell.org/Lazy\\_vs.\\_non-strict](https://wiki.haskell.org/Lazy_vs._non-strict)
- [W4] Haskell/Denotational semantics  
[https://en.wikibooks.org/wiki/Haskell/Denotational\\_semantics](https://en.wikibooks.org/wiki/Haskell/Denotational_semantics)
- [W5] Haskell/Graph reduction  
[https://en.wikibooks.org/wiki/Haskell/Graph\\_reduction](https://en.wikibooks.org/wiki/Haskell/Graph_reduction)



# References

- [S1] Hackage  
<https://hackage.haskell.org>
- [S2] Hoogle  
<https://www.haskell.org/hoogle>

Lazy,... <sup>!!!</sup>