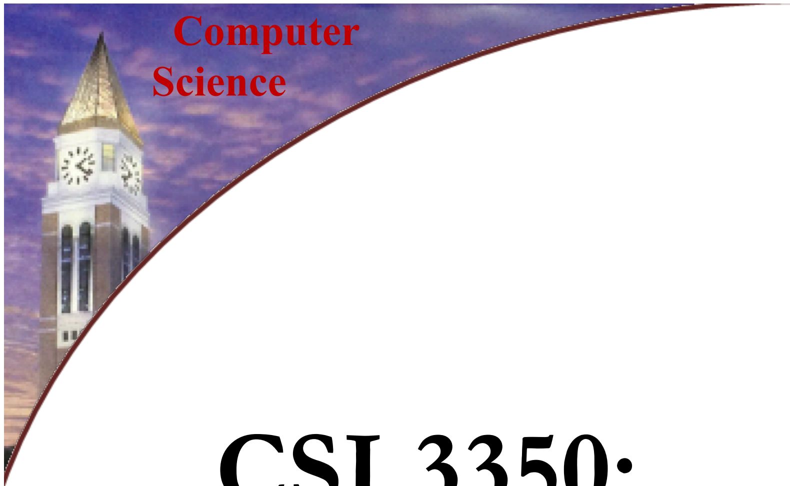
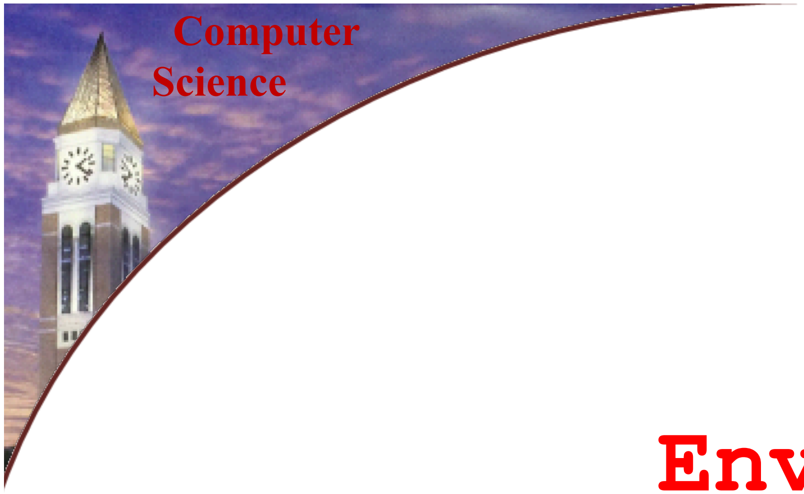


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University



Environment

Think of your **`whole`** **`program`** as a **`novel`**, then the **`Environment`** of your **`program`** is a complete **`introduction`** to all the **`characters`** included in your program.

standing for **names**

so that these **names** and their related **information** can be stored and checked out later !

The Interface For `Environment` ADT

```
Env ::= (empty-env ) | (extend-env var val Env)
```

```
(empty-env)

(extend-env var val Env)

(apply-env search-var Env)
```

Computer Science

Environment

```
(define env
    (extend-env 'n 3
       (extend-env 'm 4
               (empty-env) ) ) )


(apply-env env 'n)
```

Environment

- Data structure-based data representation
- Procedural-based data representation

Read: EOPL 2.1 - 2.3

Data Structure-based Representation

```
(define (apply-env env search-var)
  (if (eqv? (car env) 'empty-env)
      (raise "not found!")
      (if (eqv? (car env) 'extend-env)
          (let (
                (first-var (cadr env))
                (first-val (caddr env))
                (remaining-env (cadddr env))
                )
            (if (eqv? search-var first-var)
                first-val
                (apply-env remaining-env search-var)))
          (raise "invalid environment!"))))
```

Procedural-based Representation

```
Env = Var -> SchemeVal

(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))

(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))

(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

?

Computer Science

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))

(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))

(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

Computer Science

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))

(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))

(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))
```

same

```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding!")))
```

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))
```

same

```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding!")))
```

Computer Science

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))
```

same

```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding!")))
```

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))
```

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))
```

same

```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding!")))
```

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))
```

same

```
(define (extend-env saved-var saved-val saved-env)
  (lambda (search-var)
    (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var))))
```

Computer Science

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))
```

same

```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding!")))
```

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))
```

same

```
(define (extend-env saved-var saved-val saved-env)
 (lambda (search-var)
   (if (eqv? search-var saved-var)
       saved-val
       (apply-env saved-env search-var))))
```

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```scheme
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))
```

same

```scheme
(define (empty-env)
  (lambda (search-var)
    (raise "no binding!")))
```

```scheme
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))
```

same

```scheme
(define (extend-env saved-var saved-val saved-env)
  (lambda (search-var)
    (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var))))
```

```scheme
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

```scheme
(define (apply-env env search-var)
  (env search-var))
```

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))
```

same

```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding!")))
```

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))
```

same

```
(define (extend-env saved-var saved-val saved-env)
  (lambda (search-var)
    (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var))))
```

```
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

same

```
(define (apply-env env search-var)
  (env search-var))
```

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))
```

same

```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding!")))
```

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))
```

same

```
(define (extend-env saved-var saved-val saved-env)
  (lambda (search-var)
    (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var))))
```

```
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

same

```
(define (apply-env env search-var)
  (env search-var))
```

Procedural-based Representation

```
(define (extend-env saved-var saved-val saved-env)
  (lambda (search-var)
    (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var))))
```

Procedural-based Representation

```
(define (extend-env saved-var saved-val saved-env)
  (lambda (search-var)
    (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var))))
```

```
(define (extend-env x y z)
  (lambda (a)
    (if (eqv? a x)
        y
        (apply-env z a))))
```

Procedural-based Representation

```
(define (extend-env saved-var saved-val saved-env)
  (lambda (search-var)
    (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var))))
```

same

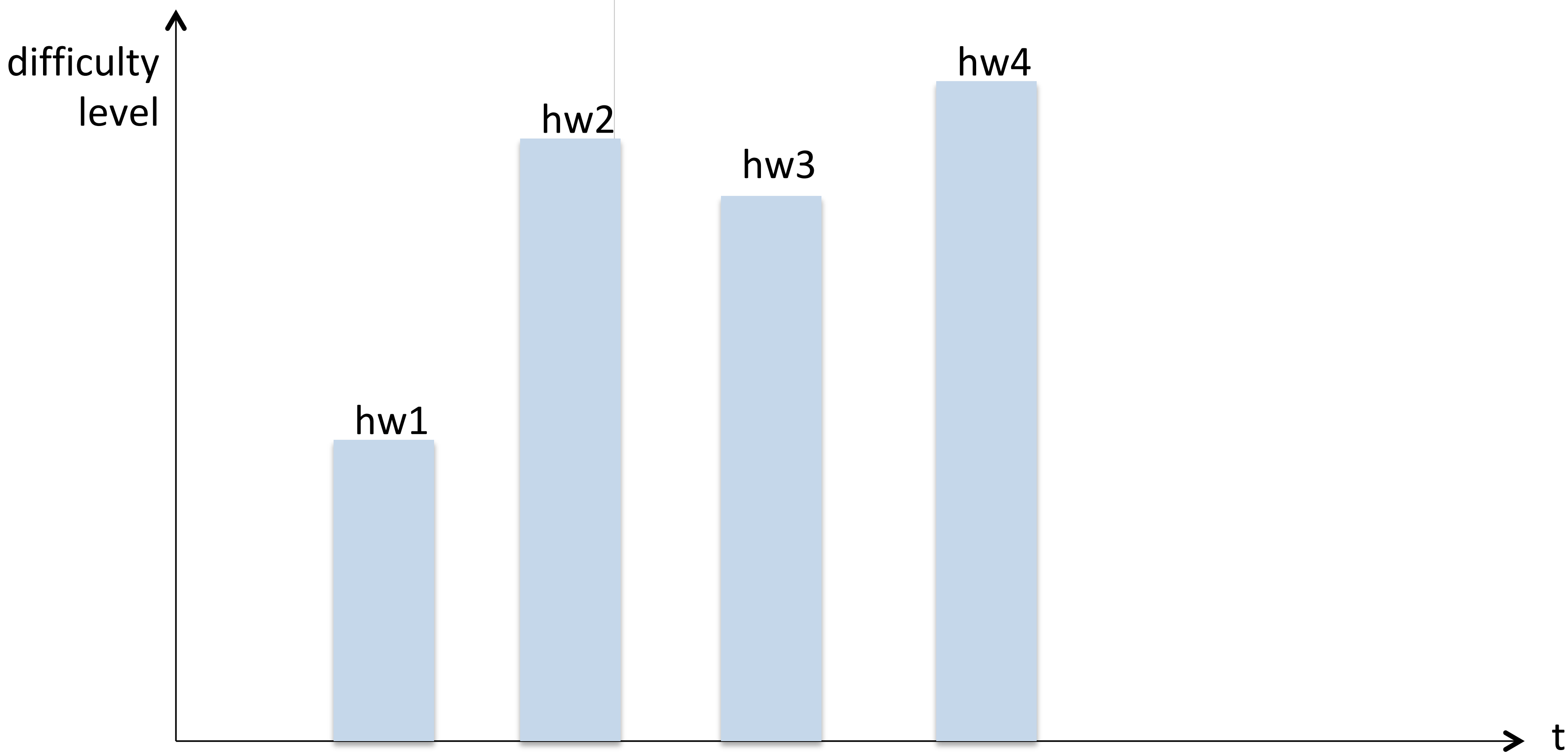
```
(define (extend-env x y z)
  (lambda (a)
    (if (eqv? a x)
        y
        (apply-env z a))))
```

Exam 01

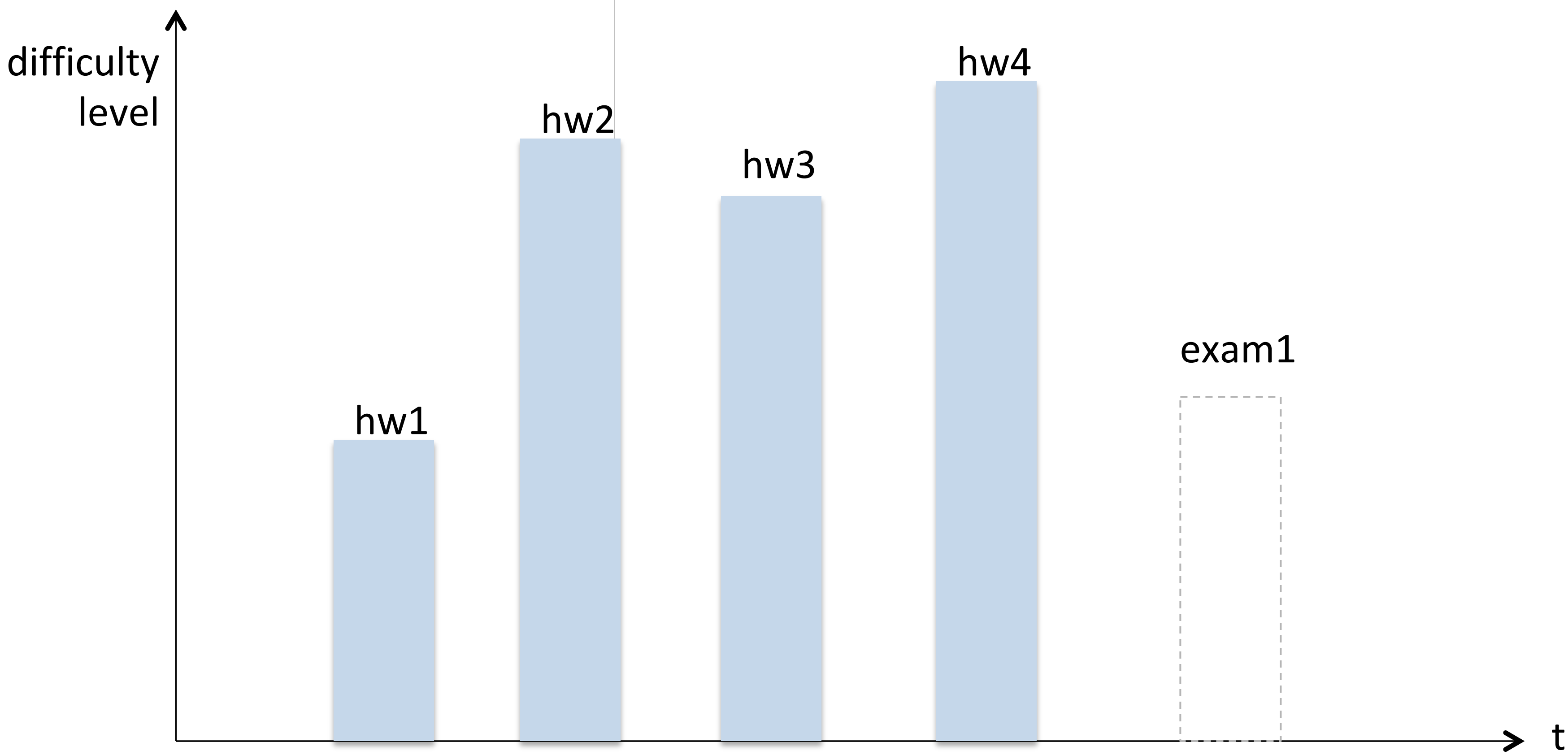
Oct 30 (in class)

(Exam 01 covers HW1~4)

Exam 01
Oct 30 (in class)
(Exam 01 covers HW1~4)



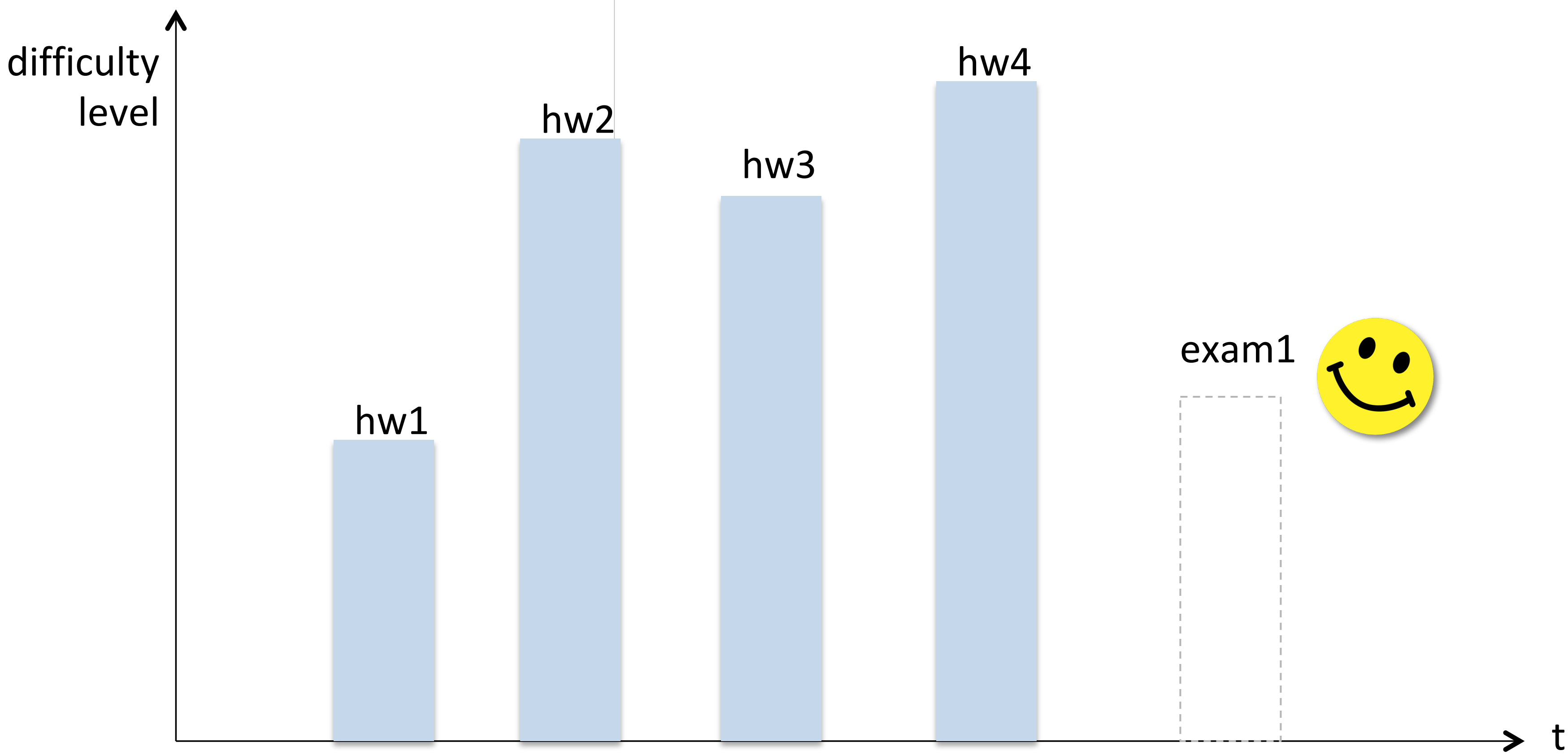
Exam 01
Oct 30 (in class)
(Exam 01 covers HW1~4)



Exam 01

Oct 30 (in class)

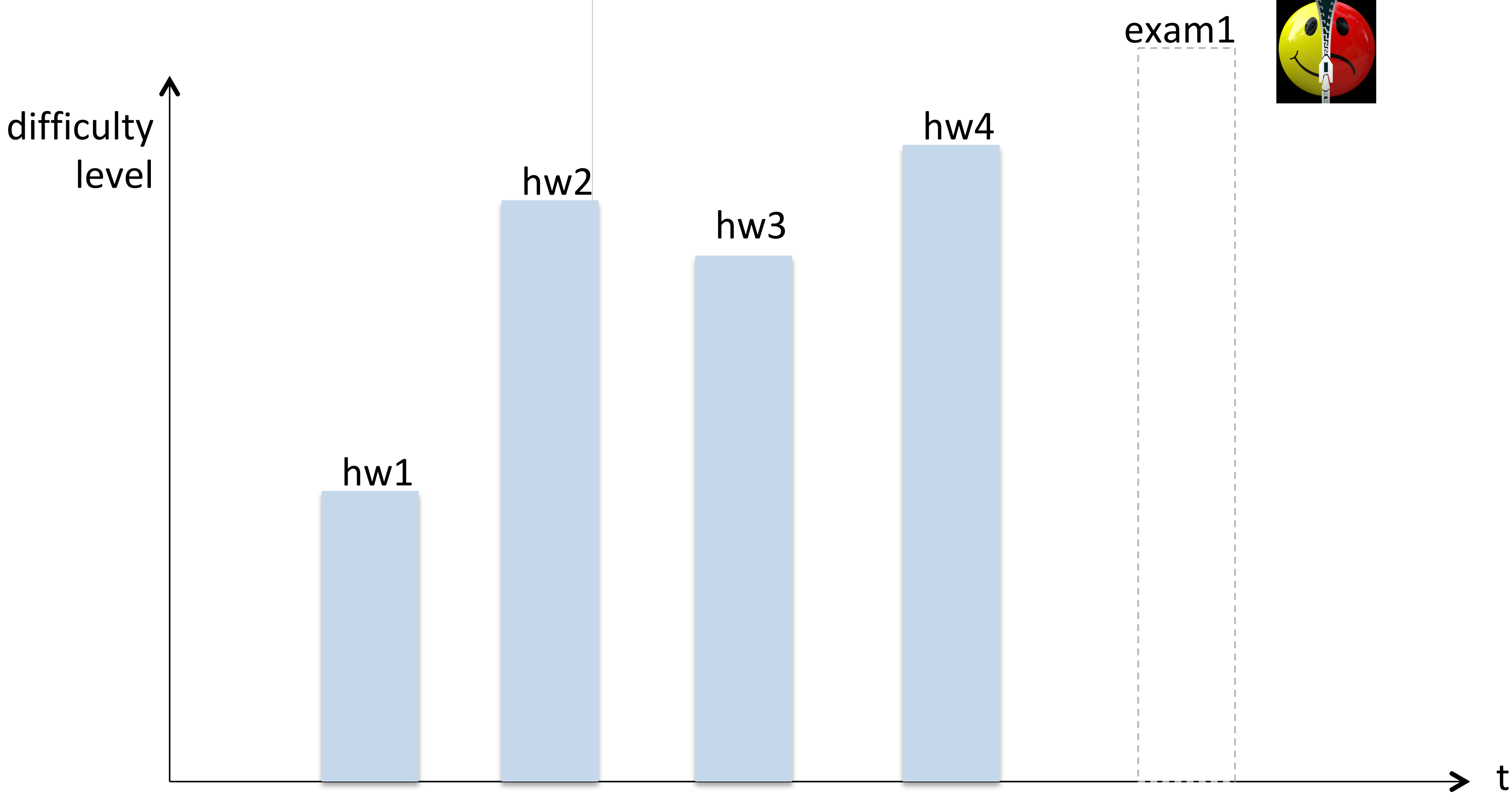
(Exam 01 covers HW1~4)



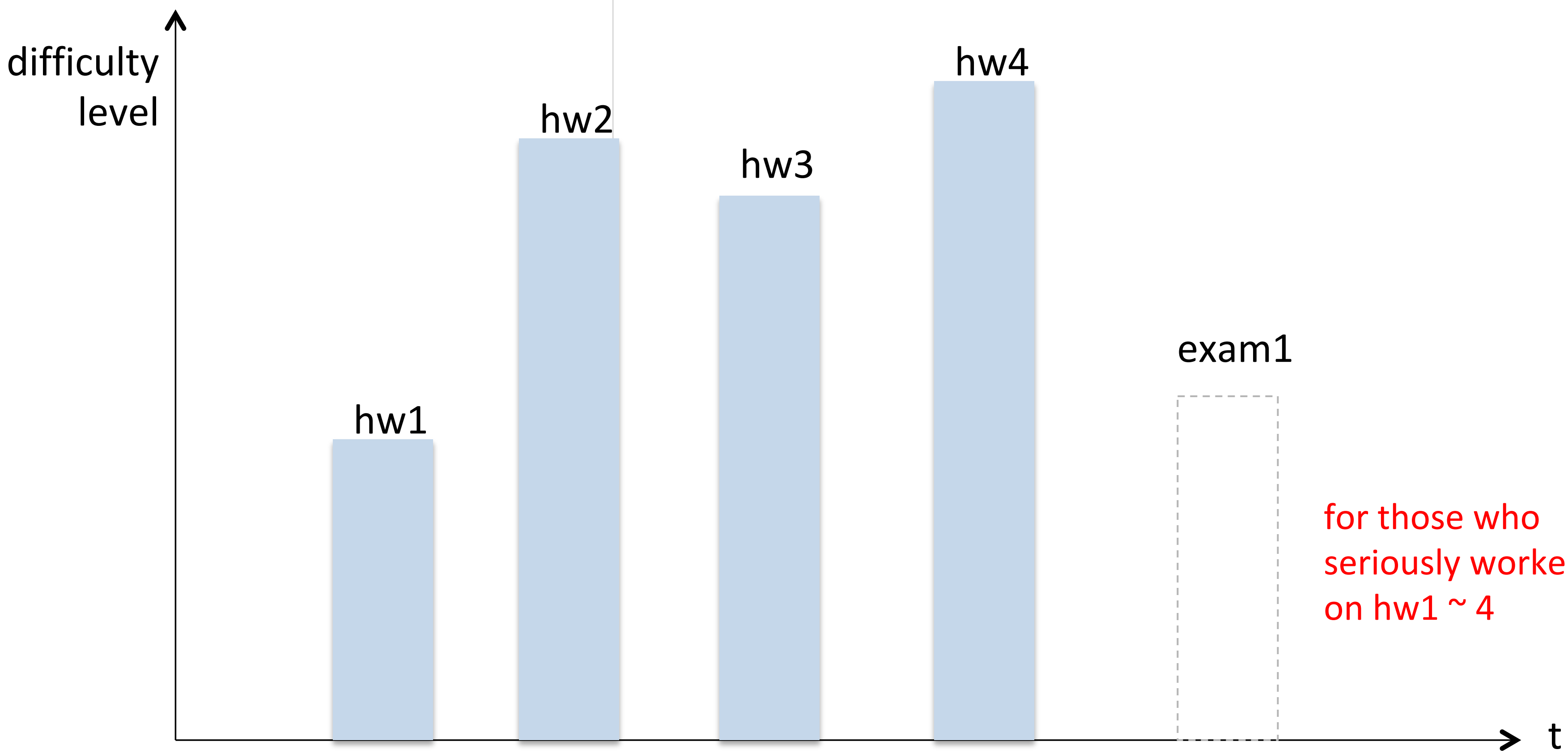
Exam 01

Oct 30 (in class)

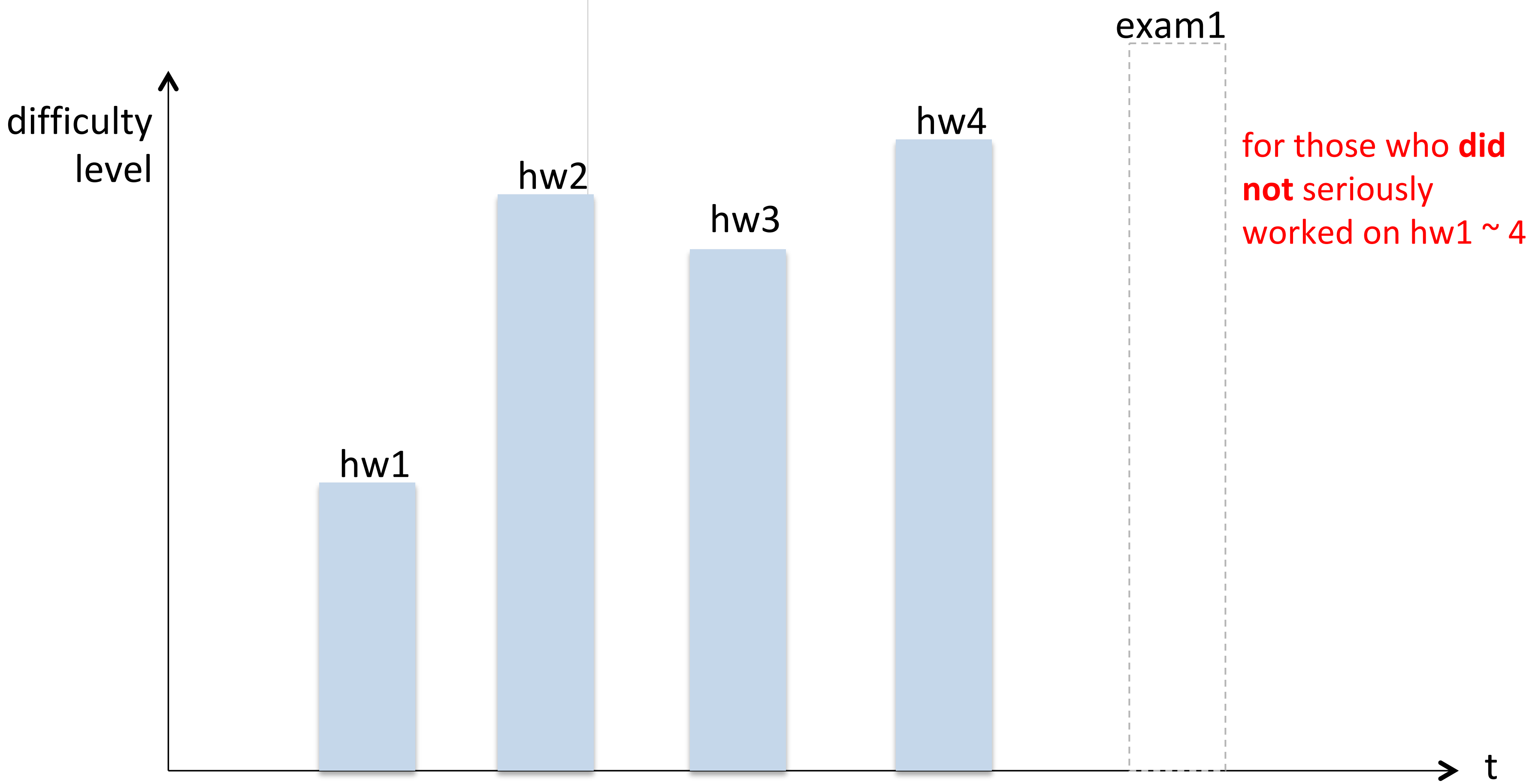
(Exam 01 covers HW1~4)



Exam 01
Oct 30 (in class)
(Exam 01 covers HW1~4)



Exam 01
Oct 30 (in class)
(Exam 01 covers HW1~4)



Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))

(define (extend-env var val env)
  (list 'extend-env var val env))
```

Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))
```

example

```
`(empty-env)
```

```
(define (extend-env var val env)
  (list 'extend-env var val env))
```

Computer Science

Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))
```

example

```
`(empty-env)
```

```
(define (extend-env var val env)
  (list 'extend-env var val env))
```

example

```
`(extend-env x 2 (empty-env))
```

Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))

(define (extend-env var val env)
  (list 'extend-env var val env))
```

example

```
`(empty-env)
```

example

```
`(extend-env x 2 (empty-env))
```

Computer Science

Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))

(define (extend-env var val env)
  (list 'extend-env var val env))
```

example

```
`(empty-env)
```

example

```
`(extend-env x 2 (empty-env))
```

example

```
`(extend-env y 3 (extend-env x 2 (empty-env)))
```

Computer Science

Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))

(define (extend-env var val env)
  (list 'extend-env var val env))
```

example

`(empty-env)

example

`(extend-env x 2 (empty-env))

example

`(extend-env y 3 (extend-env x 2 (empty-env)))

Computer Science

Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))
```

example → `` `(empty-env) ``

```
(define (extend-env var val env)
  (list 'extend-env var val env))
```

example ↓

`` `(extend-env x 2 (empty-env)) ``

example ↓

`` `(extend-env y 3 (extend-env x 2 (empty-env))) ``

Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))

(define (extend-env var val env)
  (list 'extend-env var val env))
```

example

```
`(empty-env)
```

example

```
`(extend-env x 2 (empty-env))
```

example

```
`(extend-env y 3 (extend-env x 2 (empty-env)))
```

Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))
```

example

`(empty-env)

```
(define (extend-env var val env)
  (list 'extend-env var val env))
```

example

`(extend-env x 2 (empty-env))

example

`(extend-env y 3 (extend-env x 2 (empty-env)))

```
Env ::= (empty-env ) | (extend-env var val Env)
```

Data Structure-based Representation

```
(define (empty-env)
  (list 'empty-env))
```

example

`(empty-env)

```
(define (extend-env var val env)
  (list 'extend-env var val env))
```

example

`(extend-env x 2 (empty-env))

example

`(extend-env y 3 (extend-env x 2 (empty-env)))

Env ::= (empty-env) | (extend-env var val **Env**)

Data Structure-based Representation

```
(define (apply-env env search-var)
  (if (eqv? (car env) 'empty-env)
      (raise "not found!")
      (if (eqv? (car env) 'extend-env)
          (let (
                (first-var (cadr env))
                (first-val (caddr env))
                (remaining-env (cadddr env))
                )
            (if (eqv? search-var first-var)
                first-val
                (apply-env remaining-env search-var)))
          (raise "invalid environment!"))))
```

Procedural-based Representation

```
Env = Var -> SchemeVal
```

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))

(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))

(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

?

Data Structure-based Vs. Procedural-based Data Representation

```
(define apply-env
  (lambda ( env search-var)
   (cond
    (( eqv? (car env) ‘empty-env)
      (report-no-binding-found search-var))
    (( eqv? (car env)  ‘extend-env)
      (let ( (saved-var  (cadr env) )
             (saved-val  (caddr env) )
             (saved-env (cadddr env))
            )
        (if (eqv? search-var saved-var)
             saved-val
             (apply-env saved-env search-var))))
     (else
        (report-invalid-env env) )))
)
```

```
(define apply-env
    (lambda ( env search-var)
        (env search-var) )
)
```

Data Structure-based Vs. Procedural-based Data Representation

```
(define apply-env
  (lambda ( env search-var)
   (cond
    (( eqv? (car env) 'empty-env)
       (report-no-binding-found search-var))
    (( eqv? (car env)  'extend-env)
       (let ( (saved-var  (cadr env) )
              (saved-val  (caddr env) )
              (saved-env (cadddr env))
            )
          (if (eqv? search-var saved-var)
              saved-val
              (apply-env saved-env search-var))))
    (else
       (report-invalid-env env) )))
)
```

```
(define apply-env
   (lambda ( env search-var)
      (env search-var) )
)
```

Because `environment` is implemented as a `procedure (function)`!

Data Structure-based Vs. Procedural-based Data Representation

```
(define apply-env
  (lambda ( env search-var)
   (cond
    (( eqv? (car env) 'empty-env)
      (report-no-binding-found search-var))
    (( eqv? (car env)  'extend-env)
      (let ( (saved-var  (cadr env) )
             (saved-val  (caddr env) )
             (saved-env (cadddr env))
            )
         (if (eqv? search-var saved-var)
             saved-val
             (apply-env saved-env search-var))))
     (else
        (report-invalid-env env) )))
)
```

```
(define apply-env
    (lambda ( env search-var)
       (env search-var) )
)
```

Function is powerful!

Function makes coding so MUCH simpler!

Data Structure-based Vs. Procedural-based Data Representation

```
(define apply-env
  (lambda ( env search-var)
   (cond
    (( eqv? (car env) 'empty-env)
       (report-no-binding-found search-var))
    (( eqv? (car env)  'extend-env)
       (let ( (saved-var  (cadr env) )
               (saved-val  (caddr env) )
               (saved-env (cadddr env))
             )
          (if (eqv? search-var saved-var)
               saved-val
               (apply-env saved-env search-var))))
     (else
        (report-invalid-env env) )))
)
```

```
(define apply-env
    (lambda ( env search-var)
        (env search-var) )
)
```

Function makes coding so MUCH simpler!

Interfaces For Recursive Data Types (EOPL 2.3 2.4)

A systematic method for defining data interfaces

Constructors

Observers → Predicates
→ Extractors

Rule of Thumb

Please read the test file first!

Computer Science

HW04 Problem-1

```
> (for {a-var <- '(0 1 2 3 4)} yield (+ a-var 42) )
> '(42 43 44 45 46)
```

HW04 Problem-1

```
(define-syntax-rule  ( ; new syntax is put here)
    ( ;  interpretation of the new syntax using
      ;    legal Scheme expression is
      ;    here
     )
)
```

Computer Science

HW04 Problem-1

```
(define-syntax-rule  ( ; new syntax is put here)
    ( ;  interpretation of the new syntax using
      ;    legal Scheme expression is
      ;    here                                  .
     )
 )
```

```
(define-syntax-rule (for {a-var <- value-range} yield result)
  ; your code for the meaning of this new syntax
  )
```

HW04 Problem-1

```
(define-syntax-rule  ( ; new syntax is put here)
     ( ;  interpretation of the new syntax using
       ;    legal Scheme expression is
       ;    here
      )
 )
```

a-var	'(0 1 2 3 4)	(+ a-var 42)

```
(define-syntax-rule (for {a-var <- value-range} yield result)
  ; your code for the meaning of this new syntax
  )
```

Computer Science

HW04 Problem-1

```
(define-syntax-rule  ( ; new syntax is put here)
     ( ;  interpretation of the new syntax using
       ;    legal Scheme expression is
       ;    here
      )
  )
```

a-var | `(0 1 2 3 4) | (+ a-var 42)

```
(define-syntax-rule (for {a-var <- value-range} yield result)
  ; your code for the meaning of this new syntax
  )
```

HW04 Problem-1

```
> (for {a-var <- '(0 1 2 3 4)} yield (+ a-var 42) )
> '(42 43 44 45 46)
```

HW04 Problem-1

```
(define-syntax-rule  ( ; new syntax is put here)
     ( ;  interpretation of the new syntax using
       ;    legal Scheme expression is
       ;    here
      )
 )
```

\`(42 43 44 45 46)

a-var **\`(0 1 2 3 4)** **(+ a-var 42)**

```
(define-syntax-rule (for {a-var <- value-range} yield result)
  ; your code for the meaning of this new syntax
  )
```

Computer
Science

HW04 Problem-1

```
(define-syntax-rule  ( ; new syntax is put here)
     ( ;  interpretation of the new syntax using
       ;     legal Scheme expression is
       ;     here
      )
  )
```

`'(42 43 44 45 46)`

`a-var` | `'(0 1 2 3 4)` | `(+ a-var 42)`

```
(define-syntax-rule (for {a-var <- value-range} yield result)
  (map $??some-function??$ value-range)
  )
```

Computer Science

Another Example: `Step` Data Type

rule name

```
<step> ::= <step> <step>          "seq-step"
         | "up" number            "up-step"
         | "down" number          "down-step"
         | "left" number          "left-step"
         | "right" number         "right-step"
```

Another Example: **Step** Data Type

```
<step> ::= <step> <step>          "seq-step"
         | "up" number             "up-step"
         | "down" number           "down-step"
         | "left" number           "left-step"
         | "right" number          "right-step"
```

Computer Science

Another Example: **`Step`** Data Type

```
<step> ::= <step> <step>      "seq-step"
         | "up" number         "up-step"
         | "down" number       "down-step"
         | "left" number       "left-step"
         | "right" number      "right-step"
```

Another Example: **`Step`** Data Type

```
<step> ::= <step> <step>          “seq-step”
         | “up” number             “up-step”
         | “down” number           “down-step”
         | “left” number           “left-step”
         | “right” number          “right-step”
```

Another Example: `Step` Data Type

```
<step> ::= <step> <step>          "seq-step"
        | "up" number              "up-step"
        | "down" number            "down-step"
        | "left" number            "left-step"
        | "right" number           "right-step"
```

Another Example: **Step** Data Type

```
<step> ::= <step> <step>        "seq-step"
        | "up" number            "up-step"
        | "down" number          "down-step"
        | "left" number          "left-step"
        | "right" number         "right-step"
```

Another Example: `Step` Data Type

```
<step> ::= <step> <step>        "seq-step"
        | "up" number           "up-step"
        | "down" number         "down-step"
        | "left" number         "left-step"
        | "right" number        "right-step"
```

`<step>` has **five variants**

So it needs **five constructors!**

Constructors For <step>

```
<step> ::= <step> <step>        "seq-step"
         | "up" number           "up-step"
         | "down" number         "down-step"
         | "left" number         "left-step"
         | "right" number        "right-step"
```

One Constructor for each production rule!

Constructors For <step>

```
<step> ::= <step> <step>        "seq-step"
        | "up" number           "up-step"
        | "down" number         "down-step"
        | "left" number         "left-step"
        | "right" number        "right-step"
```

Constructors For <step>

```
<step> ::= <step> <step>        "seq-step"
         | "up" number           "up-step"
         | "down" number         "down-step"
         | "left" number         "left-step"
         | "right" number        "right-step"
```

```
(define (seq-step st-1 st-2)
          ...
 )
```

Constructors For <step>

<step> ::=	<step> <step>	"seq-step"
	| "up" number	"up-step"
	| "down" number	"down-step"
	| "left" number	"left-step"
	| "right" number	"right-step"

```
(define (seq-step st-1 st-2)
        ...
)
```

Constructors For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number             "up-step"
         | "down" number           "down-step"
         | "left" number           "left-step"
         | "right" number          "right-step"
```

```
(define (seq-step st-1 st-2)
        ...

)
```

it returns a seq-step, which is a "sub-type", or **variant**, of step data type.

Constructors For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number              "up-step"
         | "down" number            "down-step"
         | "left" number            "left-step"
         | "right" number           "right-step"
```

```
(define (up-step n)
            ...

 )
```

it returns a `up-step`, which is a **variant**, of `step` data type.

Constructors For <step>

```
<step> ::= <step> <step>       "seq-step"
         | "up" number          "up-step"
         | "down" number        "down-step"
         | "left" number        "left-step"
         | "right" number       "right-step"
```

```
(define (down-step  n)
   ...

)
```

it returns a `down-step`, which is a **variant**, of `step` data type.

Constructors For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number             "up-step"
         | "down" number           "down-step"
         | "left" number           "left-step"
         | "right" number          "right-step"
```

```
(define (left-step  n)
        ...

 )
```

it returns a `left-step`, which is a **variant**, of `step` data type.

Constructors For <step>

```
<step> ::= <step> <step>            "seq-step"
         | "up" number              "up-step"
         | "down" number            "down-step"
         | "left" number            "left-step"
         | "right" number           "right-step"
```

```
(define (left-step  n)
           ...


 )
```

it returns a `left-step`, which is a **variant**, of `step` data type.

Constructors For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number             "up-step"
         | "down" number           "down-step"
         | "left" number           "left-step"
         | "right" number          "right-step"
```

```
(define (right-step  n)
        ...

  )
```

it returns a `right-step`, which is a **variant**, of `step` data type.

Computer
Science

Predicates For `<step>`

Predicates For `<step>`

Purpose -

If an object is created by one of the five constructors of STEP data type, then the predicate function should return true, otherwise false

Predicates For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number            "up-step"
         | "down" number          "down-step"
         | "left" number          "left-step"
         | "right" number         "right-step"
```

```
(define (seq-step? st)
  ...
)
```

Computer Science

Predicates For `<step>`

```
<step> ::= <step> <step>          "seq-step"
         | "up" number            "up-step"
         | "down" number          "down-step"
         | "left" number          "left-step"
         | "right" number         "right-step"
```

if `st` is a legal `seq-step` value then it must be made by the `seq-step` constructor (just mentioned)

```
(define (seq-step? st)
   ...
)
```

Predicates For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number             "up-step"
         | "down" number           "down-step"
         | "left" number           "left-step"
         | "right" number          "right-step"
```

if st is a legal seq-step value then it must be made by the seq-step constructor (just mentioned)

```
(define (seq-step? st)
    ...
)
```

it returns #t if st is a seq-step, a "sub-type" , or **variant**, of step data type; otherwise, it returns #f

Predicates For <step>

```
<step> ::= <step> <step>         "seq-step"
         | "up" number            "up-step"
         | "down" number          "down-step"
         | "left" number          "left-step"
         | "right" number         "right-step"
```

if `st` is a legal `up-step` value then it must be made by the `up-step` constructor

```
(define (up-step? st)
   ...
)
```

it returns `#t` if `st` is a `up-step`, a "sub-type" , or **variant**, of `step` data type; otherwise, it returns `#f`

Computer Science

Predicates For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number            "up-step"
         | "down" number          "down-step"
         | "left" number          "left-step"
         | "right" number         "right-step"
```

if st is a legal down-step value then it must be made by the down-step constructor

```
(define (down-step? st)
   ...
)
```

it returns #t if st is a down-step, a “sub-type” , or **variant**, of step data type; otherwise, it returns #f

Predicates For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number            "up-step"
         | "down" number          "down-step"
         | "left" number          "left-step"
         | "right" number         "right-step"
```

if `st` is a legal `left-step` value then it must be made by the `left-step` constructor

```
(define (left-step? st)
          ...

)
```

it returns `#t` if `st` is a `left-step`, a "sub-type" , or **variant**, of `step` data type; otherwise, it returns `#f`

Predicates For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number            "up-step"
         | "down" number          "down-step"
         | "left" number          "left-step"
         | "right" number         "right-step"
```

if `st` is a legal `right-step` value then it must be made by the `right-step` constructor

```
(define (right-step? st)
   ...
)
```

it returns `#t` if `st` is a `right-step`, a "sub-type" , or **variant**, of `step` data type; otherwise, it returns `#f`

Extractors For `<step>`

Purpose -
If an object is created by one of the five constructors of STEP data type, then the extractor function should be able to return its respective component.

Computer Science

Extractors For <step>

```
<step> ::= <step> <step>          "seq-step"
        | "up" number             "up-step"
        | "down" number           "down-step"
        | "left" number           "left-step"
        | "right" number          "right-step"
```

```
(define (seq-step->st1 st)
              ...

)
```

Extractors For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number             "up-step"
         | "down" number           "down-step"
         | "left" number           "left-step"
         | "right" number          "right-step"
```

```
(define (seq-step->st1 st)
          ...

)
```

```
(seq-step st-1 st-2)
```

Extractors For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number             "up-step"
         | "down" number           "down-step"
         | "left" number           "left-step"
         | "right" number          "right-step"
```

```
(define (seq-step->st1 st)
          ...
)
```

```
(seq-step st-1 st-2)
```

"what is the 1st step in this sequence step?"

Extractors For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number             "up-step"
         | "down" number           "down-step"
         | "left" number           "left-step"
         | "right" number          "right-step"
```

```
(define (seq-step->st2 st)
   ...
)
```

```
(seq-step st-1 st-2)
```

"what is the 2nd step in this sequence step?"

Extractors For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number            "up-step"
         | "down" number          "down-step"
         | "left" number          "left-step"
         | "right" number         "right-step"
```

```
(define (up-step->n st)
        ...

)
```

(up-step n)

"what is the size of this up step?"

Extractors For <step>

```
<step> ::= <step> <step>        "seq-step"
         | "up" number          "up-step"
         | "down" number        "down-step"
         | "left" number        "left-step"
         | "right" number       "right-step"
```

```
(define (down-step->n st)
        ...

 )
```

```
(down-step n )
```

"what is the size of this down step?"

Extractors For <step>

```
<step> ::= <step> <step>          "seq-step"
         | "up" number            "up-step"
         | "down" number          "down-step"
         | "left" number          "left-step"
         | "right" number         "right-step"
```

```
(define (left-step->n st)
        ...
 )
```

(left-step n)

"what is the size of this left step?"

Extractors For <step>

```
<step> ::= <step> <step>        "seq-step"
         | "up" number          "up-step"
         | "down" number        "down-step"
         | "left" number        "left-step"
         | "right" number       "right-step"
```

```
(define (right-step->n st)
    ...

)
```

`(right-step n )`

"what is the size of this right step?"

An Example on `right-step`

```
(define (right-step n)
    ; constructor

)
```

```
(define (right-step?  st)

    ; predicate
)
```

```
(define (right-step->n  st)

    ; extractor
)
```

An Example on **`right-step`**

Data Structure based implementation

```
(define (right-step n)
    ; constructor
)

(define (right-step?  st)
    ; predicate
)

(define (right-step->n  st)
    ; extractor
)
```

An Example on `right-step`

Data Structure based implementation

```
(define (right-step n)

    ; constructor

)
```

what data structure do you want to use to represent `right-step`?

```
(define (right-step?  st)

    ; predicate

)
```

```
(define (right-step->n  st)

    ; extractor

)
```

An Example on **`right-step`**

Data Structure based implementation

```
(define (right-step n)
    ; constructor                 `( right  n)
)
```

```
(define (right-step?  st)
    ; predicate
)
```

```
(define (right-step->n  st)
    ; extractor
)
```

An Example on **right-step**

Data Structure based implementation

```
(define (right-step n)
      ...
      (list 'right n)          '( right  n)
      ...
)
```

```
(define (right-step?  st)

    ; predicate

)
```

```
(define (right-step->n  st)

    ; extractor

)
```

An Example on `right-step`

Data Structure based implementation

```
(define (right-step n)
      ...
      (list 'right n)          '( right   n)
      ...
)
```

```
(define (right-step?  st)

 ; st must be a list
  ;(car st) must be 'right
  ;(second st) must be a
  ; number
 ...
)
```

```
(define (right-step->n  st)


    ; extractor

)
```

An Example on **`right-step`**

Data Structure based implementation

```
(define (right-step n)
      ...
      (list 'right n)
      ...
)
```

'(right n)

```
(define (right-step?  st)

 ; st must be a list
  ;(car st) must be 'right
  ;(second st) must be a
  ; number
 ...
)
```

```
(define (right-step->n  st)
   ...
   ; (second lst)
   ...
)
```

An Example on `right-step`

Procedural-based implementation

```
(define (right-step n)

  ; constructor

)
```

```
(define (right-step?  st)

    ; predicate

)
```

```
(define (right-step->n  st)

    ; extractor

)
```

Computer Science

An Example on **`right-step`**

Procedural-based implementation

```
(define (right-step n)

      ; constructor
      ; a function is returned to support
            (1)  right-step predicate
            (2)  right-step extractor
)
```

```
(define (right-step?  st)

    ; predicate
)
```

```
(define (right-step->n  st)

   ; extractor
)
```

Computer Science

An Example on `right-step`

Procedural-based implementation

```
(define (right-step n)

      ; constructor
      ; a function is returned to support
            (1)  right-step predicate
            (2)  right-step extractor
)
```

How to use a right-step as a function?

```
(define (right-step?  st)

    ; predicate
)
```

```
(define (right-step->n  st)

   ; extractor
)
```

An Example on `right-step`

Procedural-based implementation

```
(define (right-step n)

      ; constructor
      ; a function is returned to support
            (1)   right-step predicate
            (2)   right-step extractor
)
```

How to use a right-step as a function?

```
(define (right-step?  st)

     ; predicate
)
```

```
(define (right-step->n  st)

    ; extractor
)
```

An Example on `right-step`

Procedural-based implementation

```
(define (right-step n)

      ; constructor
      ; a function is returned to support
            (1)  right-step predicate
            (2)  right-step extractor
)
```

How to use a right-step as a function?

```
(define (right-step?  st)

     (st 'right-p)
)
```

```
(define (right-step->n  st)

   ; extractor
)
```

An Example on `right-step`

Procedural-based implementation

```
(define (right-step n)

      ; constructor
      ; a function is returned to support
            (1)  right-step predicate
            (2)  right-step extractor
)
```

How to use a right-step as a function?

```
(define (right-step?  st)

    (st 'right-p)
)
```

```
(define (right-step->n  st)

   (st 'extract-size)
)
```

Computer Science

An Example on `right-step`

Procedural-based implementation

```
(define (right-step n)

    ; constructor
    ; a function is returned to support
          (1)  right-step predicate
          (2)  right-step extractor
)
```

How to use a right-step as a function?

```
(define (right-step?  st)

    (st 'right-p)
)
```

```
(define (right-step->n  st)

    (st 'extract-size)
)
```

An Example on **`right-step`**

Procedural-based implementation

```
(define (right-step n)
      ; constructor
      (lambda (a)
        (if (= a 'extract-size) ; for extractor
            n
            ... ))
)
```

How to use a right-step as a function?

```
(define (right-step?  st)

     (st 'right-p)
)
```

```
(define (right-step->n  st)

    (st 'extract-size)
)
```

An Example on `right-step`

Procedural-based implementation

```
(define (right-step n)
      ; constructor
      (lambda (a)
        (if (= a 'extract-size)
            n
            (if (= a 'right-p)
                #t
                #f )))
)
```

How to use a right-step as a function?

```
(define (right-step?  st)

     (st 'right-p)
)
```

```
(define (right-step->n  st)

    (st 'extract-size)
)
```