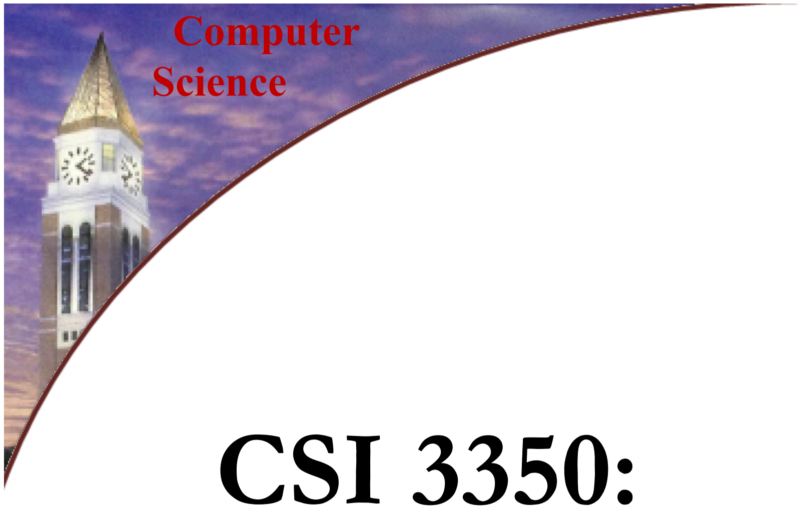
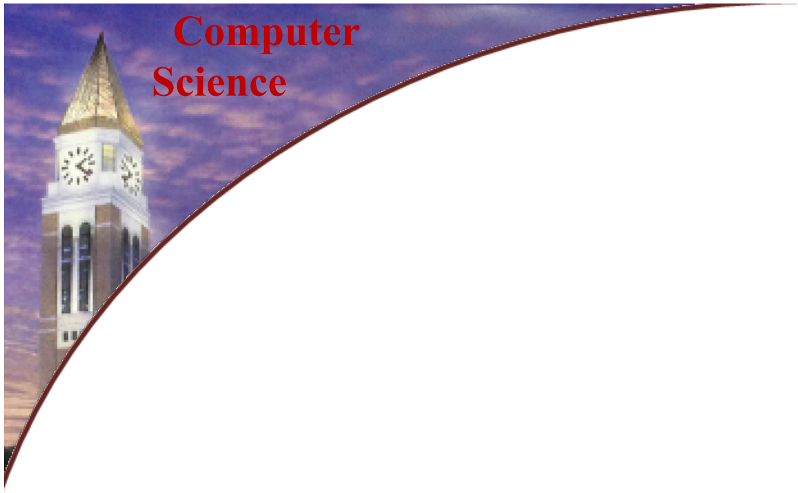


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University



Exam 01

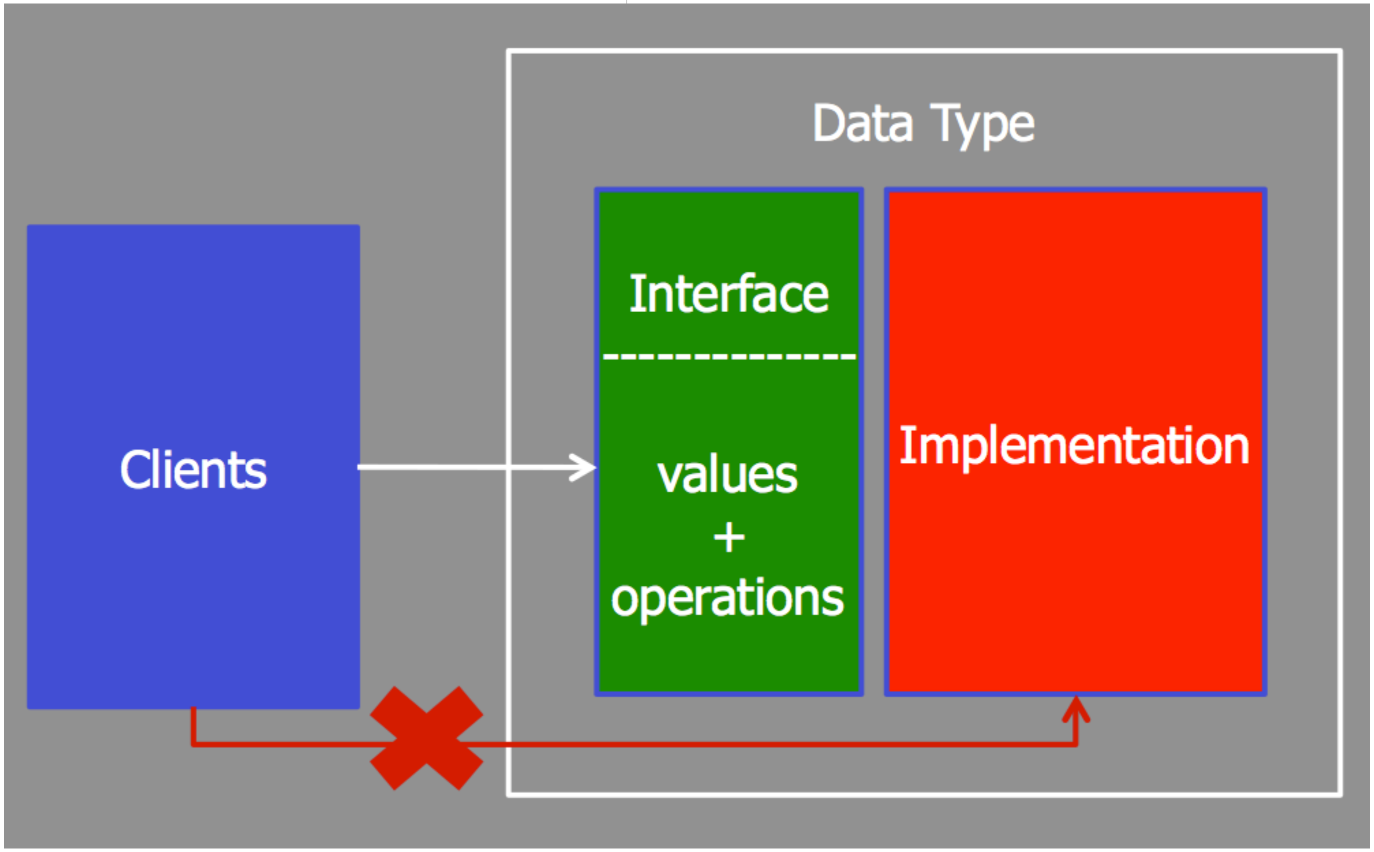
Oct 30 (in class)

(Exam 01 covers HW1~4)

t

Data Interface

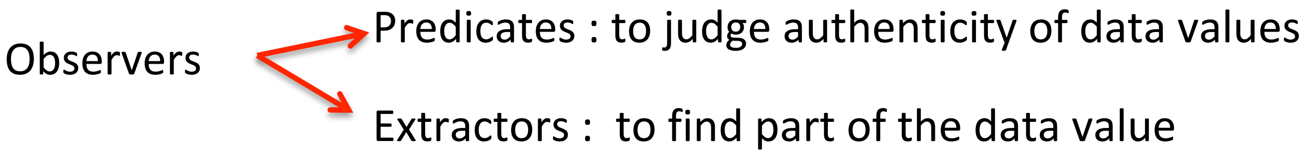
Goal: data implementation independence



Interfaces For Recursive Data Types
(EOPL 2.3 2.4)

A systematic method for defining data interfaces

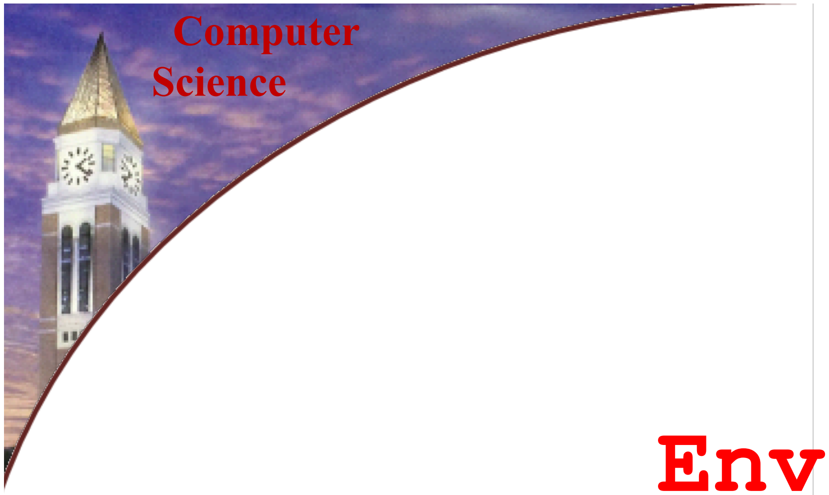
Constructors: to produce data values



Observers

Predicates : to judge authenticity of data values

Extractors : to find part of the data value



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.

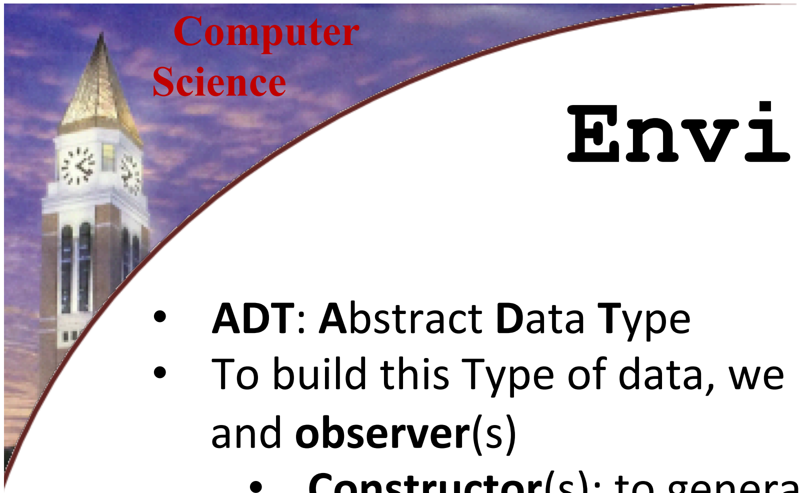
standing for **names**

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

The Interface For Environment ADT

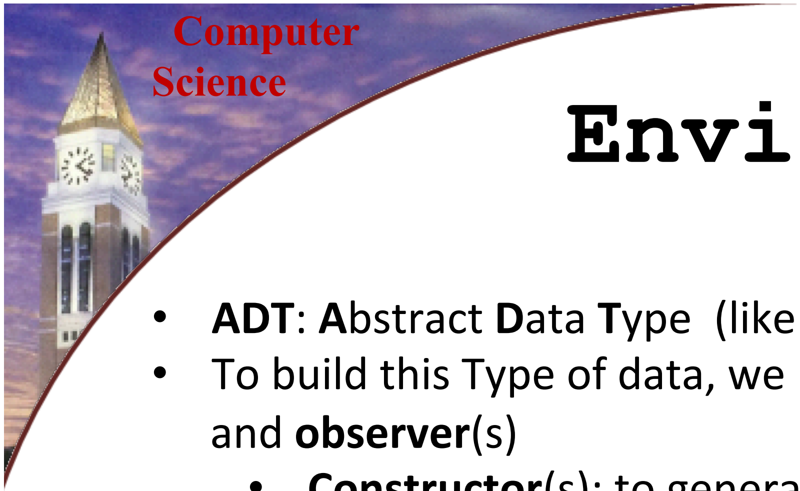
```
Env ::= (empty-env ) | (extend-env var val Env)
```

```
(empty-env)

(extend-env var val Env)

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



Environment ADT

- **ADT**: **A**bstract **D**ata **T**ype
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
  - **Constructor**(s): to generate concrete instances of this Type of data
  - **Observer**: given an instance of this type of data, can we observe it, query it



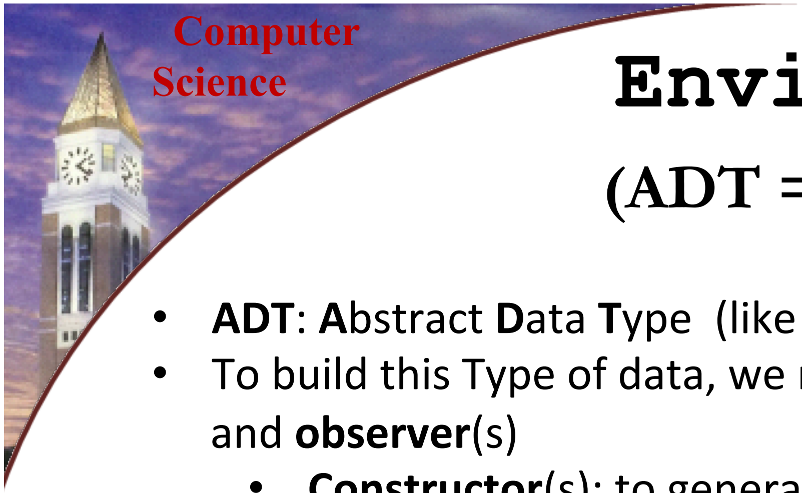
Environment ADT

- **ADT**: **A**bstract **D**ata **T**ype (like ArrayList in Java)
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
  - **Constructor**(s): to generate concrete instances of this Type of data
  - **Observer**: given an instance of this type of data, can we observe it, query it

Environment ADT

(ADT = Abstract Data Type)

- **ADT**: **A**bstract **D**ata **T**ype (like **ArrayList** in Java)
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
  - **Constructor**(s): to generate concrete instances of this Type of data
  - **Observer**: given an instance of this type of data, can we observe it, query it



Environment ADT

(ADT = Abstract Data Type)

- **ADT**: **A**bstract **D**ata **T**ype (like **ArrayList** in Java)
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
  - **Constructor**(s): to generate concrete instances of this Type of data
  - **Observer**: given an instance of this type of data, can we observe it, query it

Environment ADT

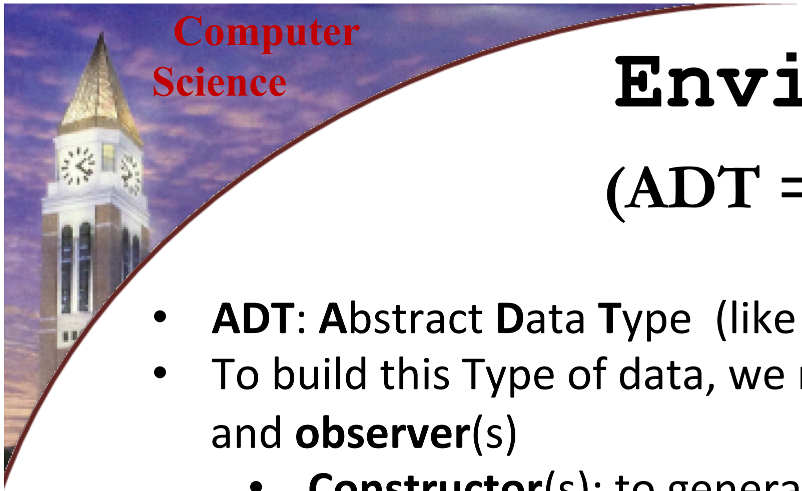
(ADT = Abstract Data Type)

- **ADT**: **A**bstract **D**ata **T**ype (like **ArrayList** in Java)
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
  - **Constructor**(s): to generate concrete instances of this Type of data
    - for **Environment** ADT: 2 constructors
    - for **ArrayList** ADT in Java: 2 constructors
  - **Observer**: given an instance of this type of data, can we observe it, query it

Environment ADT

(ADT = Abstract Data Type)

- **ADT**: **A**bstract **D**ata **T**ype (like **ArrayList** in Java)
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
    - **Constructor**(s): to generate concrete instances of this Type of data
        - for **Environment** ADT: 2 constructors
        - for **ArrayList** ADT in Java: 2 constructors
            - `ArrayList ( )`   ; empty ArrayList
            - `ArrayList(Collection<? extends E> c)`   ; extend ArrayList
    - **Observer**: given an instance of this type of data, can we observe it, query it

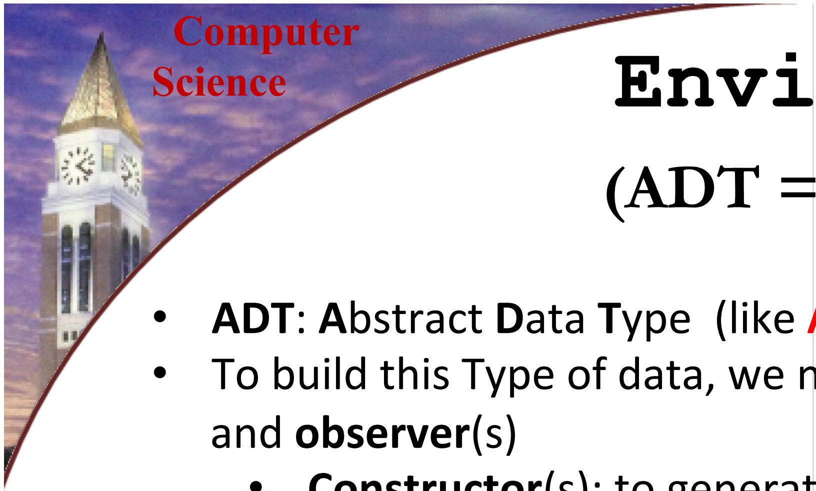


Environment ADT

(ADT = Abstract Data Type)

- **ADT**: **A**bstract **D**ata **T**ype (like **ArrayList** in Java)
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
  - **Constructor**(s): to generate concrete instances of this Type of data
    - for **Environment** ADT: 2 constructors
      - `emtpy-env`
      - `extend-env`
    - for **ArrayList** ADT in Java: 2 constructors
      - `ArrayList ( )`   ; empty ArrayList
      - `ArrayList(Collection<? extends E> c)`   ; extend ArrayList

  - **Observer**: given an instance of this type of data, can we observe it, query it



Environment ADT

(ADT = Abstract Data Type)

- **ADT**: **A**bstract **D**ata **T**ype (like **ArrayList** in Java)
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
  - **Constructor**(s): to generate concrete instances of this Type of data
    - for **Environment** ADT: 2 constructors
      - `emtpy-env`
      - `extend-env`
    - for **ArrayList** ADT in Java: 2 constructors
      - `ArrayList ( )` ; empty ArrayList
      - `ArrayList(Collection<? extends E> c)` ; extend ArrayList
  - **Observer**: given an instance of this type of data, can we observe it, query it
    - for **Environment** ADT: 1 observer
    - for **ArrayList** ADT in Java: many observers, such as

Environment ADT

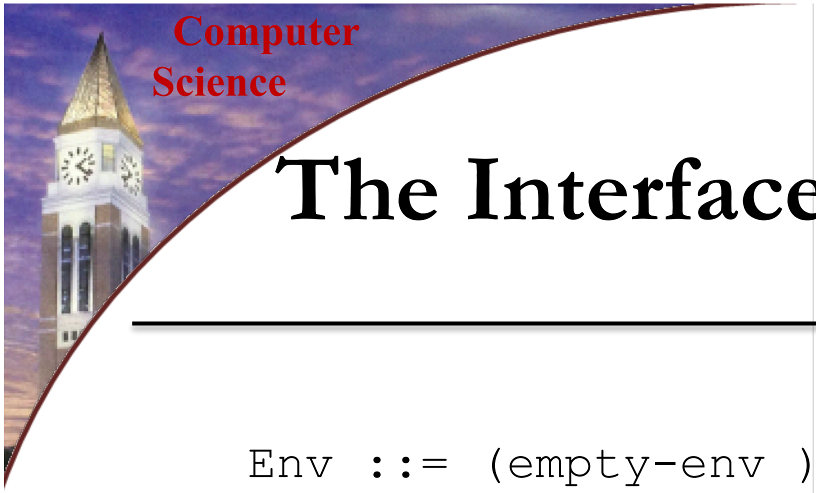
(ADT = Abstract Data Type)

- **ADT**: **A**bstract **D**ata **T**ype (like **ArrayList** in Java)
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
  - **Constructor**(s): to generate concrete instances of this Type of data
    - for **Environment** ADT: 2 constructors
      - `emtpy-env`
      - `extend-env`
    - for **ArrayList** ADT in Java: 2 constructors
      - `ArrayList ( )` ; empty ArrayList
      - `ArrayList(Collection<? extends E> c)` ; extend ArrayList
  - **Observer**: given an instance of this type of data, can we observe it, query it
    - for **Environment** ADT: 1 observer
    - for **ArrayList** ADT in Java: many observers, such as
      - `size()`
      - ......

Environment ADT

(ADT = Abstract Data Type)

- **ADT**: **A**bstract **D**ata **T**ype (like **ArrayList** in Java)
- To build this Type of data, we need to provide our code to give **constructor**(s) and **observer**(s)
  - **Constructor**(s): to generate concrete instances of this Type of data
    - for **Environment** ADT: 2 constructors
      - `emtpy-env`
      - `extend-env`
    - for **ArrayList** ADT in Java: 2 constructors
      - `ArrayList ( )` ; empty ArrayList
      - `ArrayList(Collection<? extends E> c)` ; extend ArrayList
  - **Observer**: given an instance of this type of data, can we observe it, query it
    - for **Environment** ADT: 1 observer
      - `apply-env`
    - for **ArrayList** ADT in Java: many observers, such as
      - `size()`
      - `......`



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)
```

only these 3 interface functions needed for `Environment` ADT

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)
```

only these 3 interface functions needed for `Environment` ADT

but these 3 functions can be implemented in 2 completely different ways:

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)
```

only these 3 interface functions needed for `Environment` ADT

but these 3 functions can be implemented in 2 completely different ways:

- **data structure-based**
- **procedural-based**

Environment

EOPL 2.1 - 2.3

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

both follow the definition of Environment (`Env`) below

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

Environment

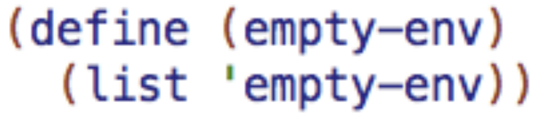
EOPL 2.1 - 2.3

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

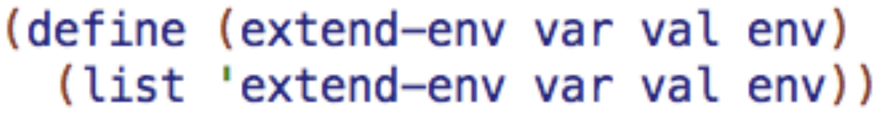
both follow the definition of Environment (**Env**) below

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

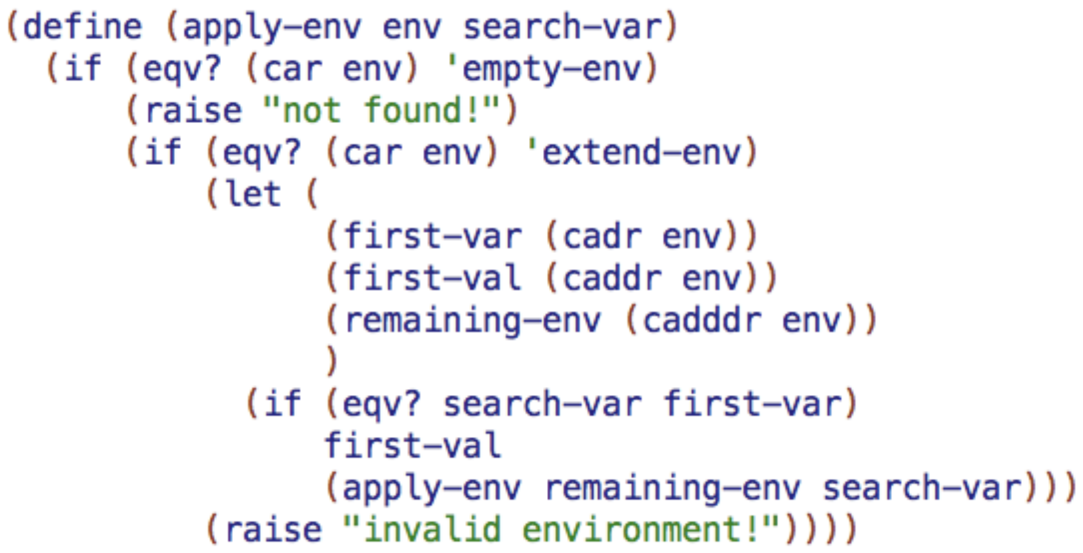
Data Structure-based Representation of Environment



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



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

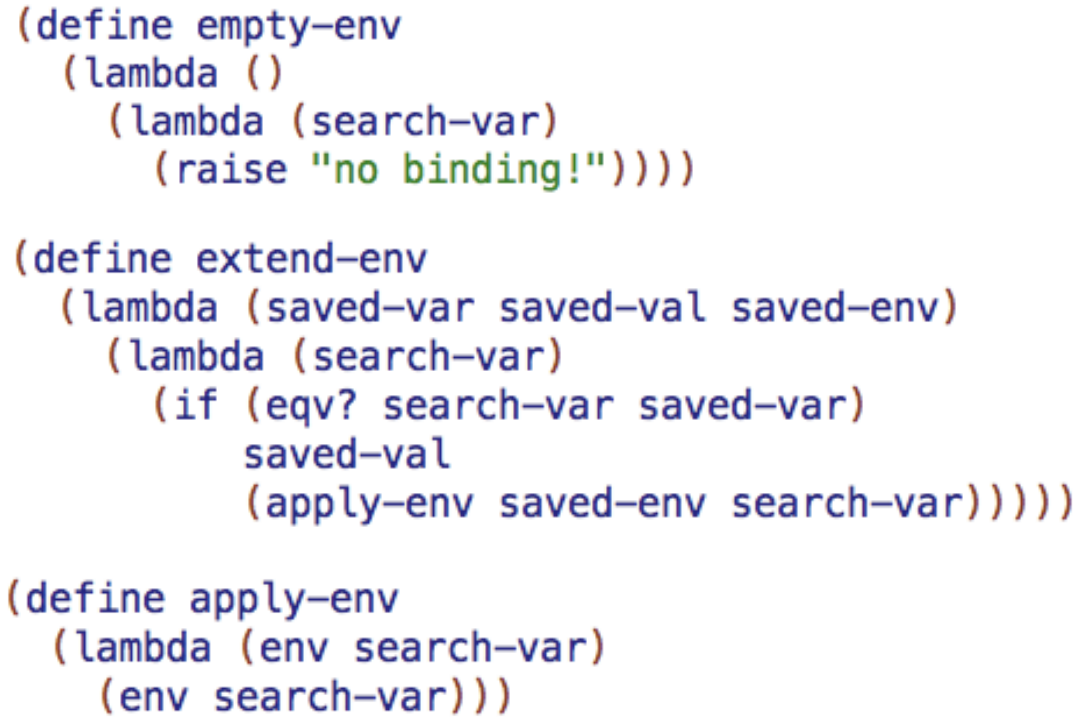


```
(define (apply-env env search-var)
  (if (eqv? (car env) 'empty-env)
      (raise "not found!")
      (if (eqv? (car env) 'extend-env)
          (let (
                (first-var (cadr env))
                (first-val (caddr env))
                (remaining-env (cadddr env))
                )
  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

Procedural-based Representation of Environment

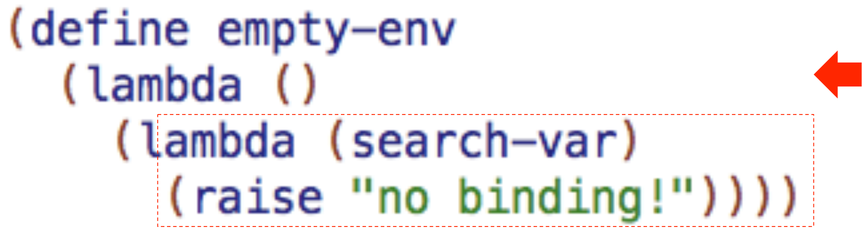
`Env = Var -> SchemeVal`

(`Env` is a procedure (function) whose input is a variable name, such as **m**
whose output is a SchemeVal, such as **5**)



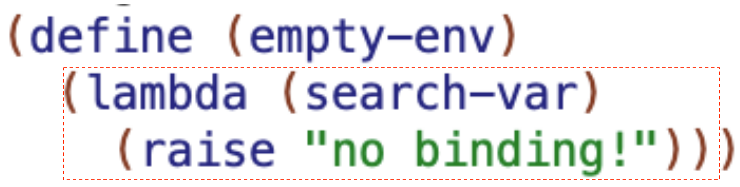
```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))

(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 of Environment

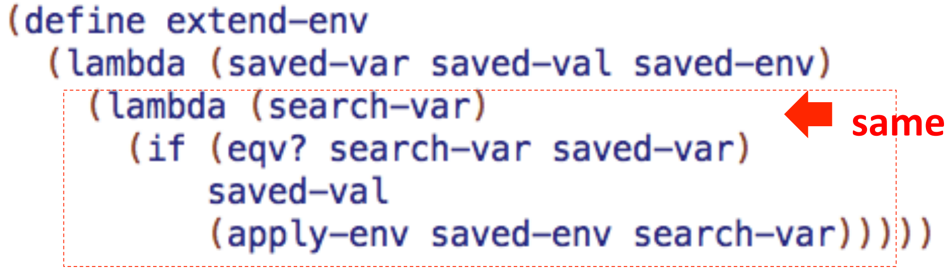


```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))
```

same

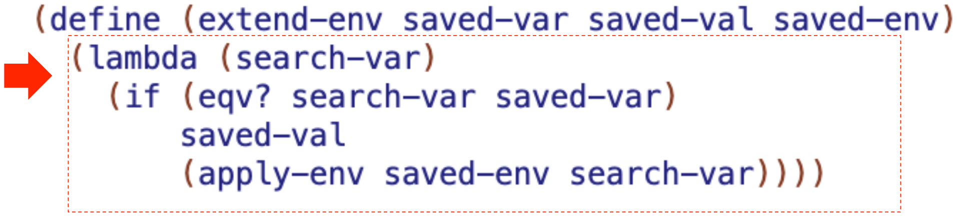


```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding!")))
```

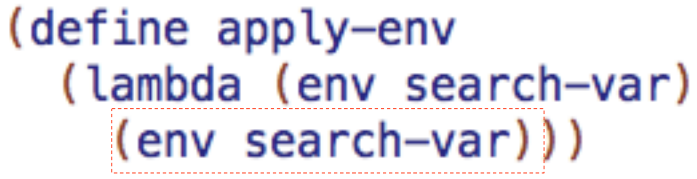


```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
          saved-val
          (apply-env saved-env search-var)))))
```

same

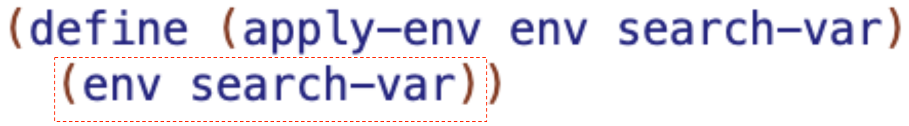


```
(define (extend-env saved-var saved-val saved-env)
  (lambda (search-var)
    (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var))))
```



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

same

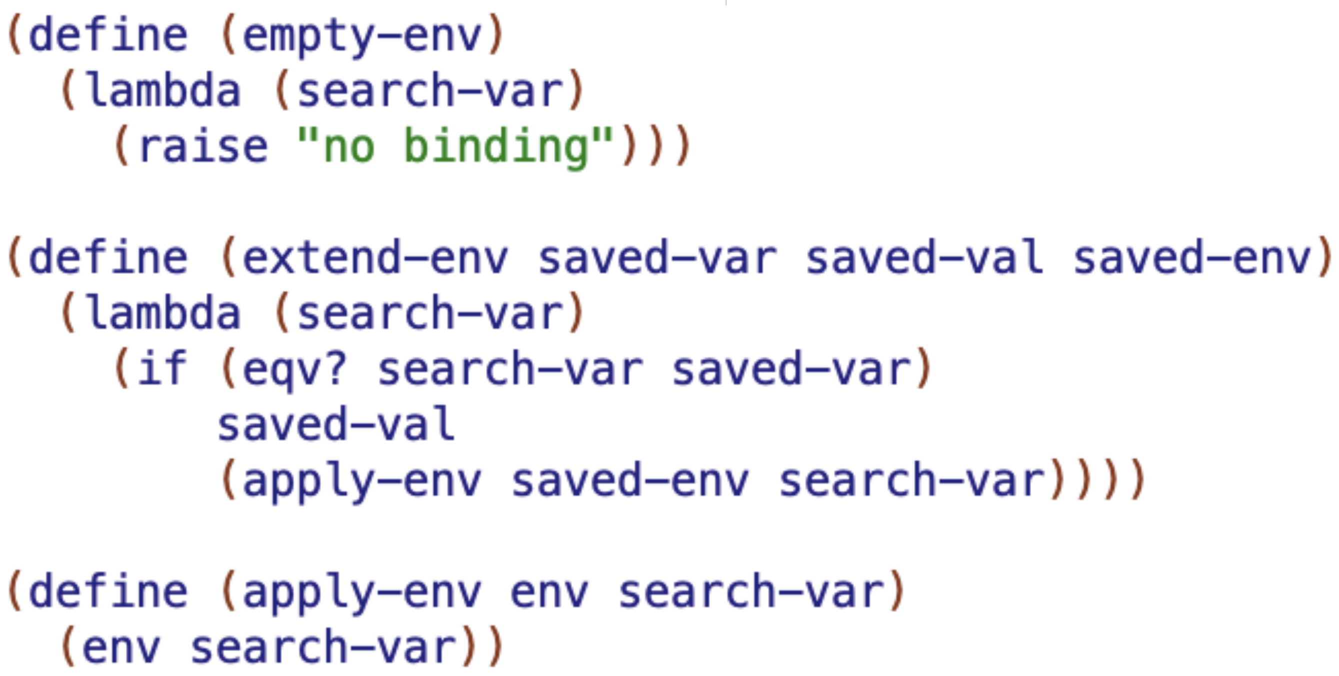


```
(define (apply-env env search-var)
  (env search-var))
```

Procedural-based Representation of Environment

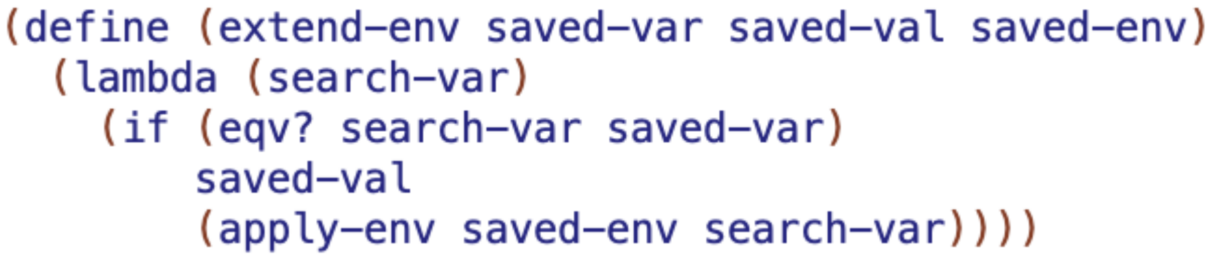
`Env = Var -> SchemeVal`

(`Env` is a procedure whose input is a variable name, such as m
whose output is a SchemeVal, such as 5)

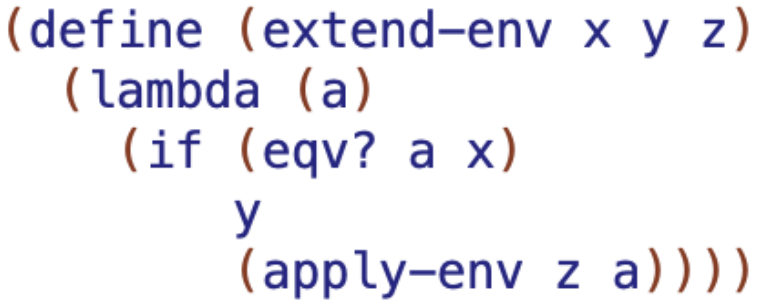


```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding")))

(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 env search-var)
  (env search-var))
```

Procedural-based Representation of Environment

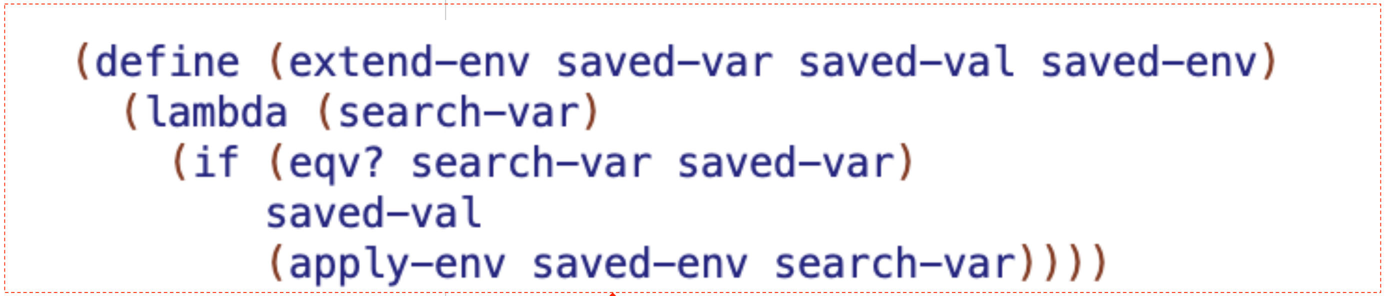


```
(define (extend-env saved-var saved-val saved-env)
  (lambda (search-var)
    (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var))))
```



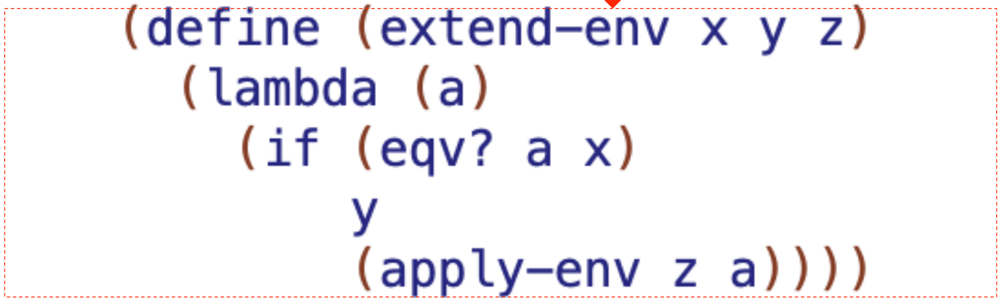
```
(define (extend-env x y z)
  (lambda (a)
    (if (eqv? a x)
        y
        (apply-env z a))))
```

Procedural-based Representation of Environment



```
(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

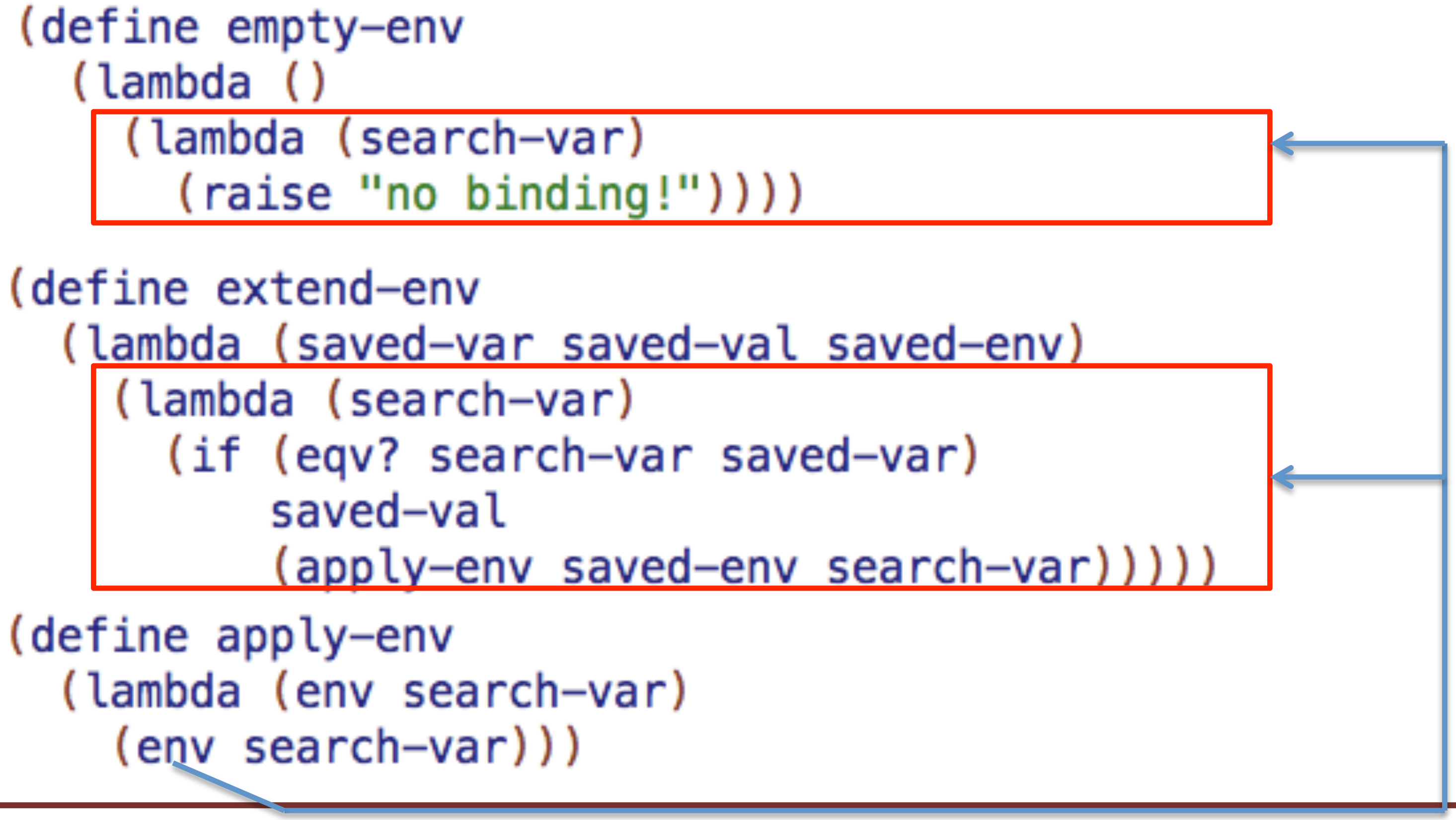


```
(define (extend-env x y z)
  (lambda (a)
    (if (eqv? a x)
        y
        (apply-env z a))))
```

Procedural-based Representation

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

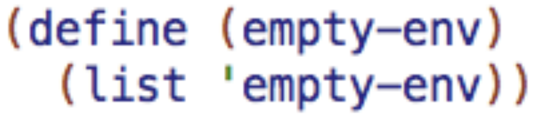
(`Env` is a procedure whose input is a var, such as m
whose output is a SchemeVal, such as 5)



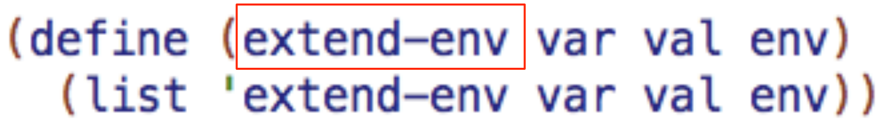
```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (raise "no binding!"))))

(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 Representation of Environment

How to use Environment ADT concretely ?



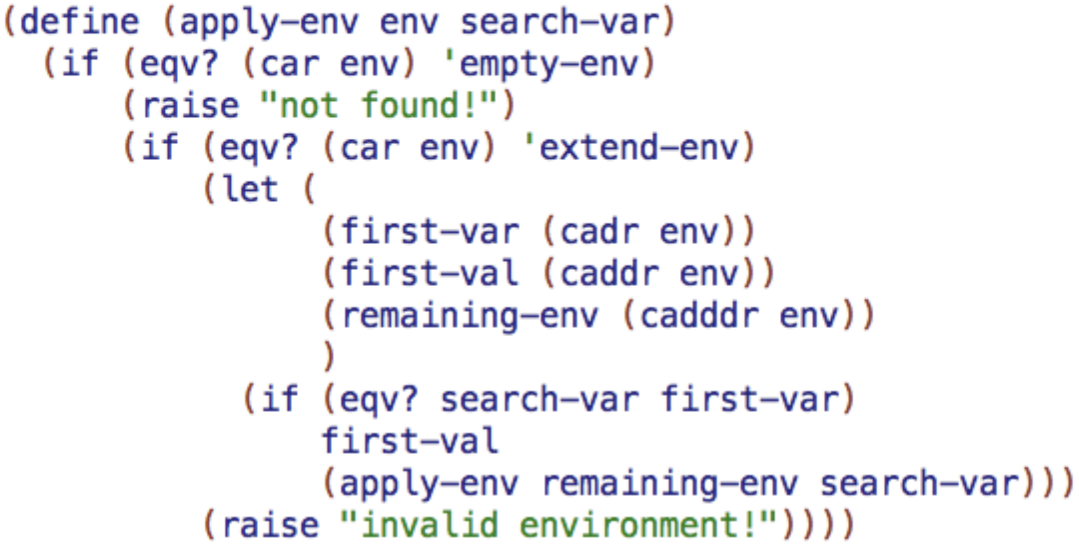
```
(define (empty-env)
  (list 'empty-env))
```



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

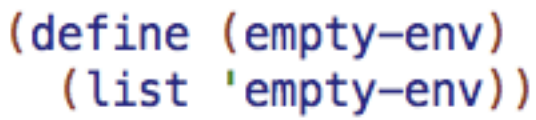
How to call `extend-env` ?

(extend-env ‘m 3 (empty-env))

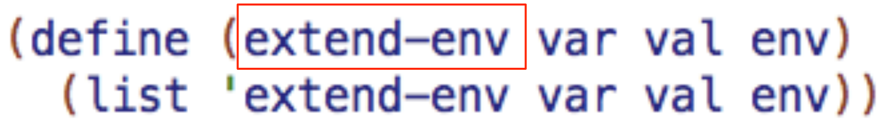


```
(define (apply-env env search-var)
  (if (eqv? (car env) 'empty-env)
      (raise "not found!")
      (if (eqv? (car env) 'extend-env)
          (let (
                (first-var (cadr env))
                (first-val (caddr env))
                (remaining-env (cadddr env))
                )
  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

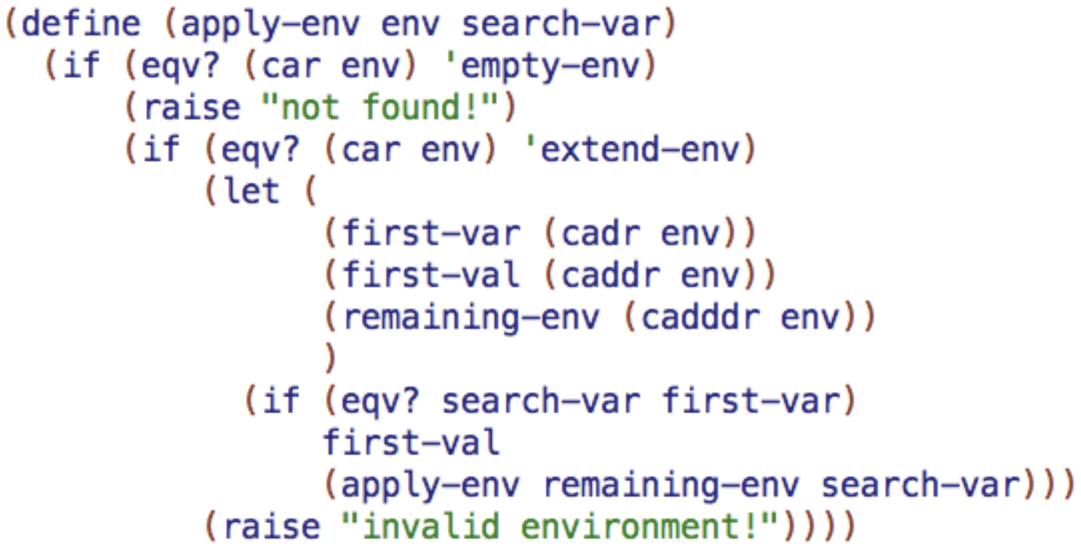
Data Structure-based Representation of Environment



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



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



```
(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!"))))
```

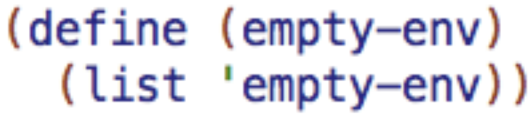
How to use Environment ADT concretely ?

How to call `extend-env` ?

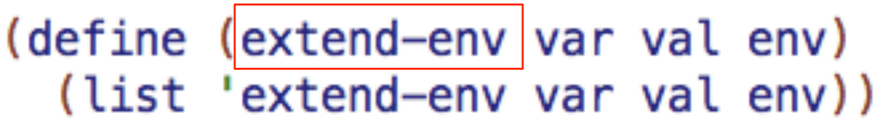
(extend-env ‘m 3 (empty-env))

Computer Science

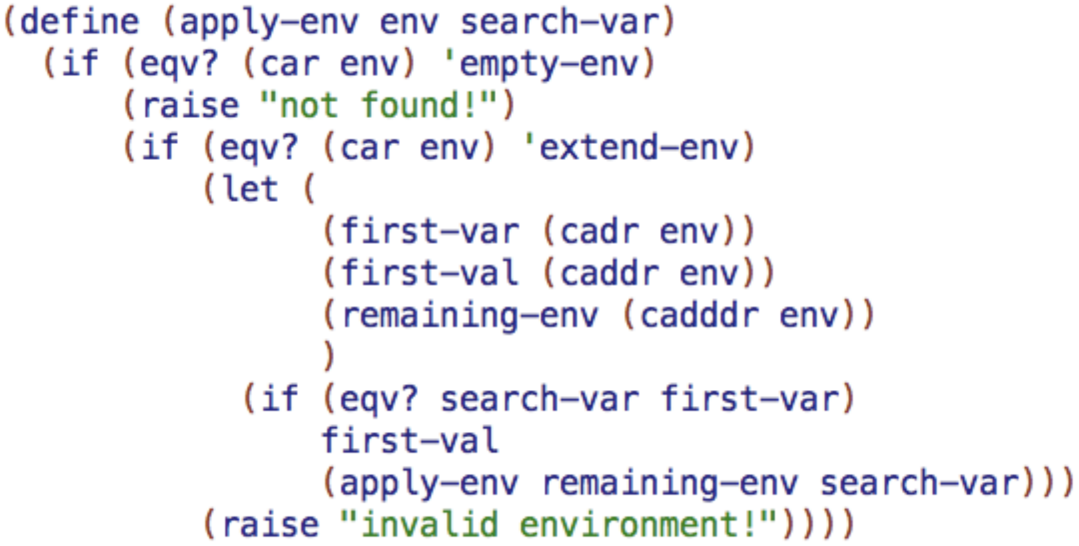
Data Structure-based Representation of Environment



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



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



```
(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!"))))
```

How to use Environment ADT concretely ?

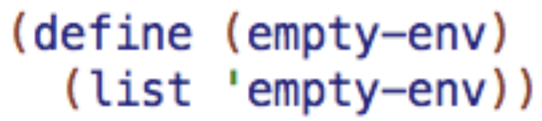
How to call `extend-env` ?

(extend-env ‘m 3 (empty-env))

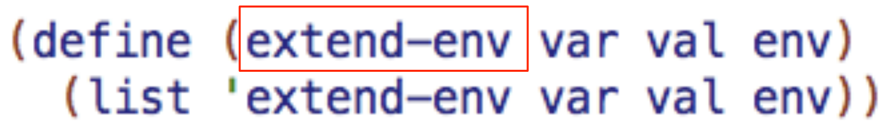
↓

‘(extend m 3 (empty-env))

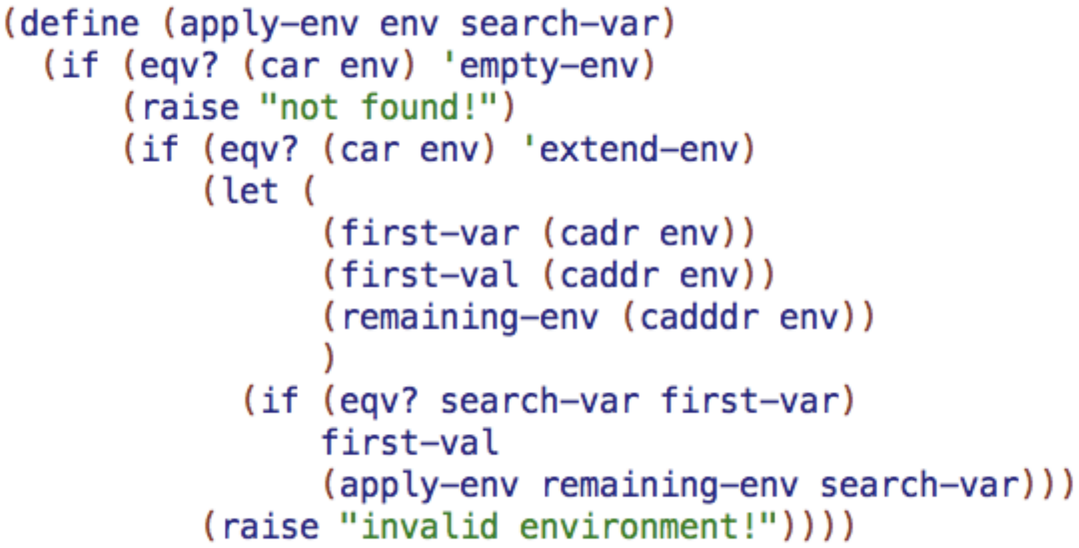
Data Structure-based Representation of Environment



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



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



```
(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!"))))
```

How to use Environment ADT concretely ?

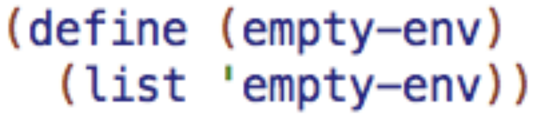
How to call `extend-env` ?

(extend-env ‘m 3 (empty-env))

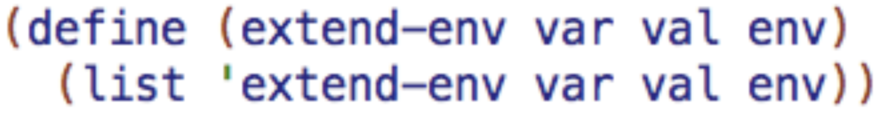
⬇ output

‘(extend-env m 3 (empty-env))

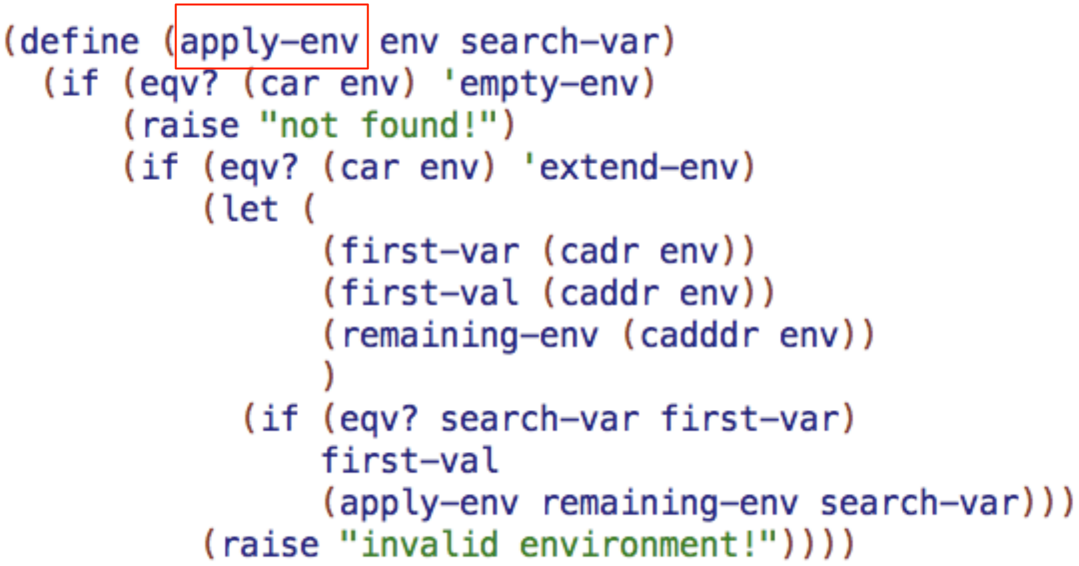
Data Structure-based Representation of Environment



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



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



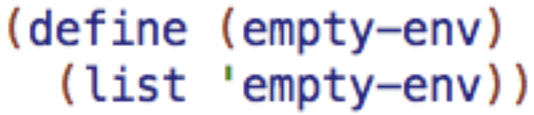
```
(define (apply-env env search-var)
  (if (eqv? (car env) 'empty-env)
      (raise "not found!")
      (if (eqv? (car env) 'extend-env)
          (let (
                (first-var (cadr env))
                (first-val (caddr env))
                (remaining-env (cadddr env))
                )
  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

How to use Environment ADT concretely ?

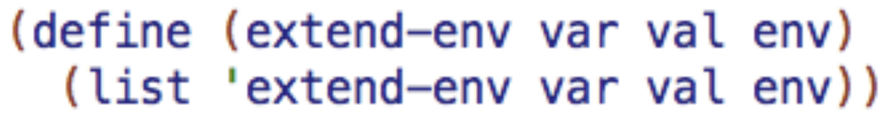
How to call `apply-env` ?

(apply-env (extend-env ‘m 3 (empty-env)) ‘m)

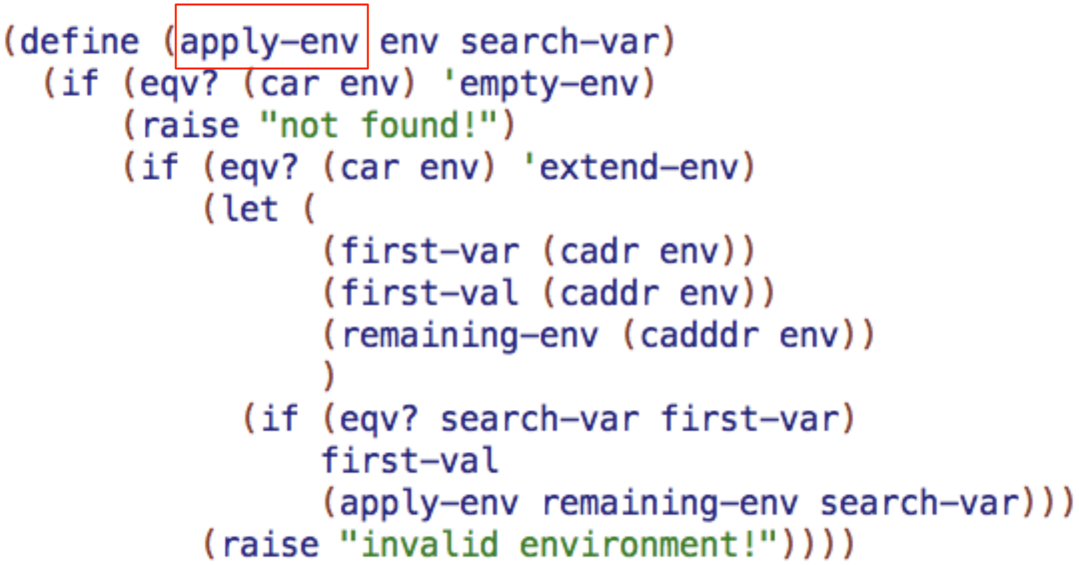
Data Structure-based Representation of Environment



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



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



```
(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!"))))
```

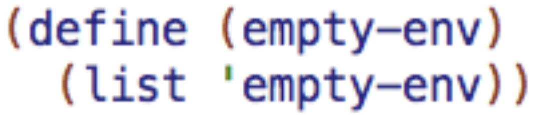
How to use Environment ADT concretely ?

How to call `apply-env` ?

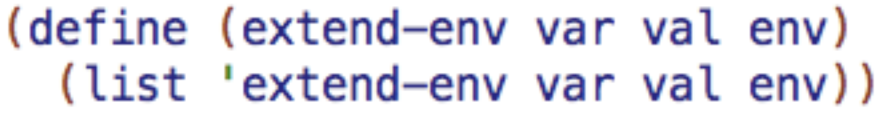
(apply-env (extend-env ‘m 3 (empty-env)) ‘m)

Computer Science

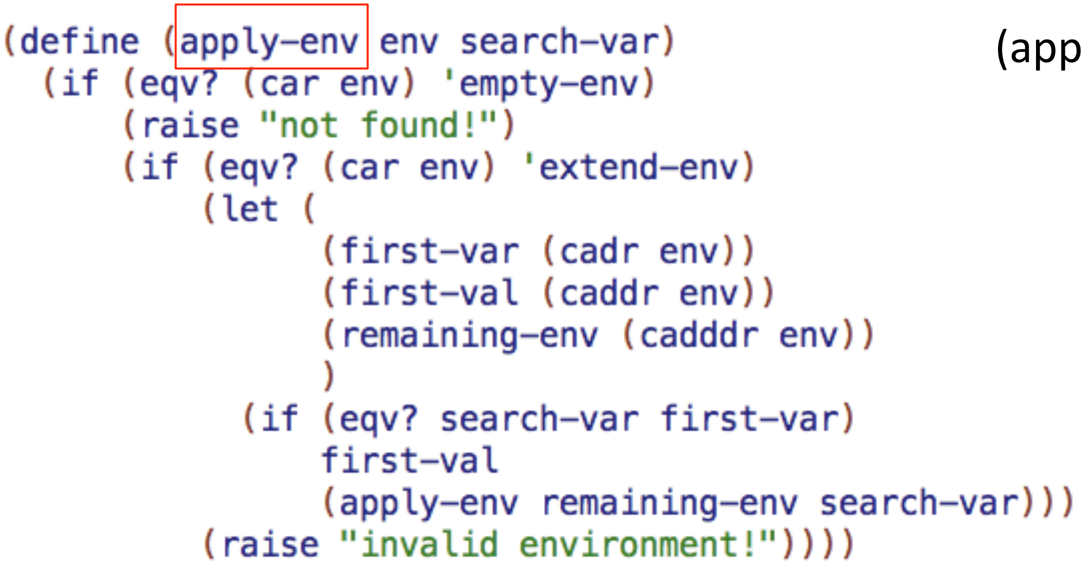
Data Structure-based Representation of Environment



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



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



```
(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!"))))
```

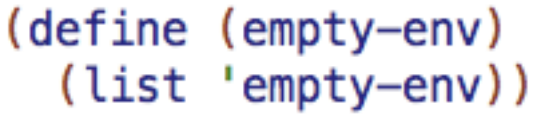
How to use Environment ADT concretely ?

How to call `apply-env` ?

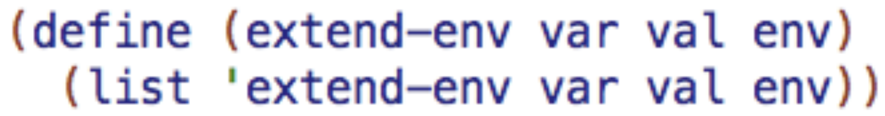
(apply-env (extend-env ‘m 3 (empty-env)) ‘m)

(apply-env ‘(extend-env ‘m 3 (empty-env)) ‘m)

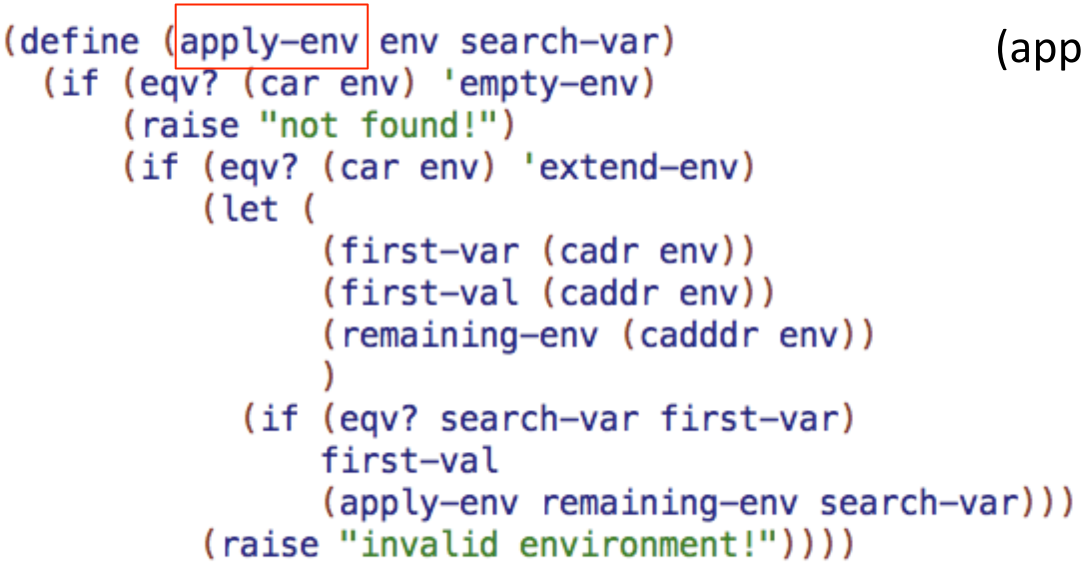
Data Structure-based Representation of Environment



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



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



```
(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!"))))
```

How to use Environment ADT concretely ?

How to call `apply-env` ?

(apply-env (extend-env ‘m 3 (empty-env)) ‘m)

(apply-env ‘(extend-env ‘m 3 (empty-env)) ‘m)

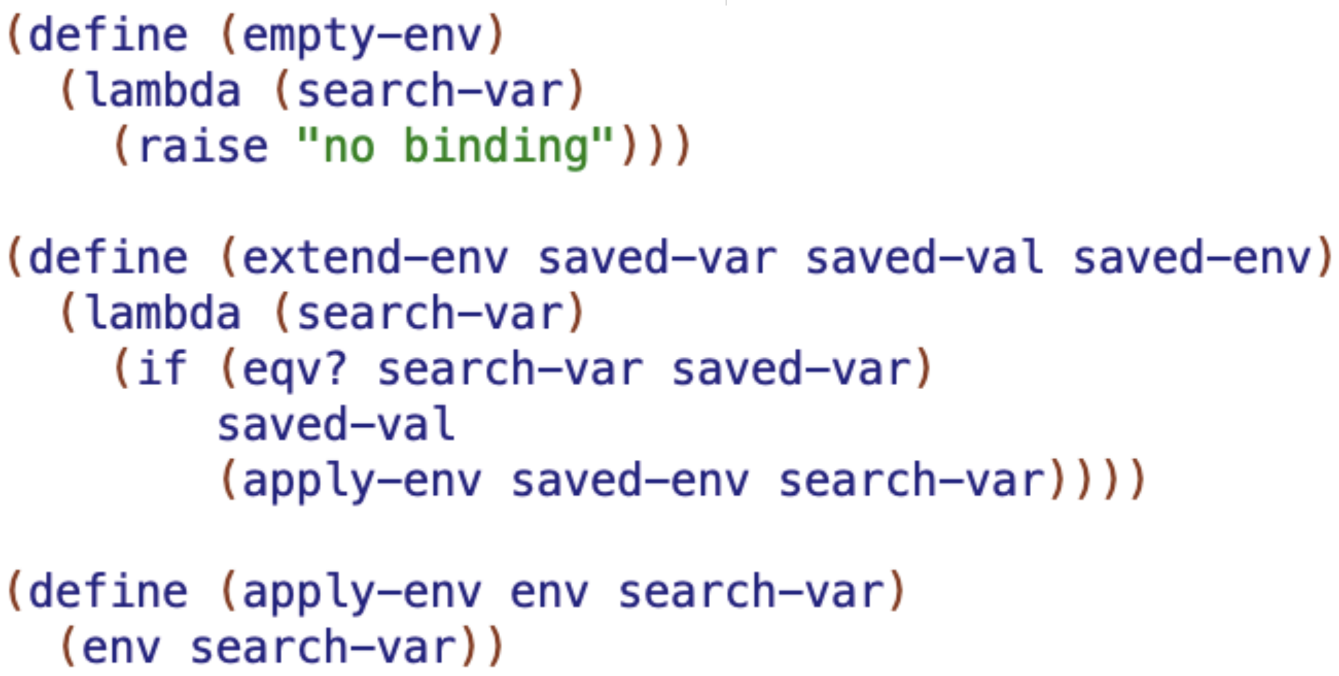
output

3

Procedural-based Representation of Environment

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

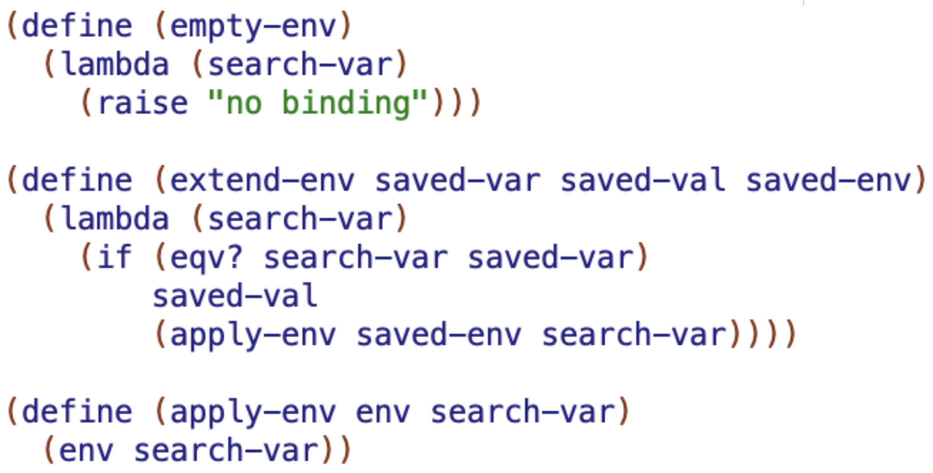
(`Env` is a procedure whose input is a var, such as m
whose output is a SchemeVal, such as 5)



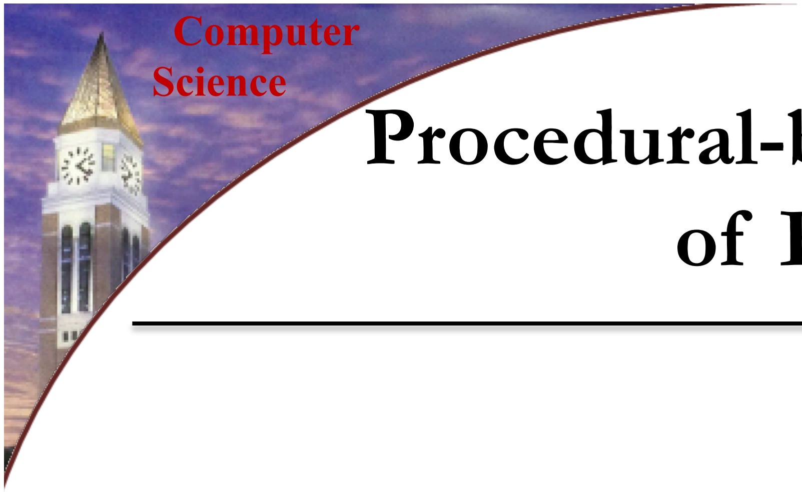
```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding")))

(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 env search-var)
  (env search-var))
```

Procedural-based Representation of Environment

How to use Environment ADT concretely ?



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

(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 env search-var)
  (env search-var))
```

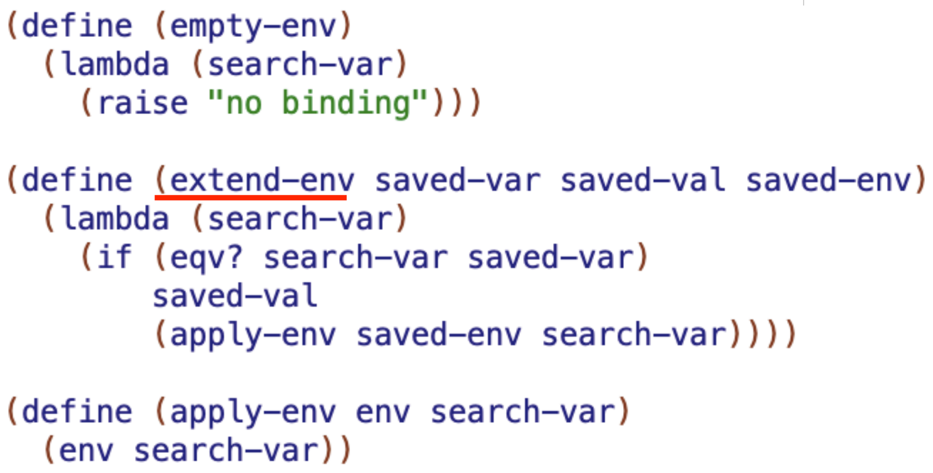


Computer Science

Procedural-based Representation of Environment

How to use Environment ADT concretely ?

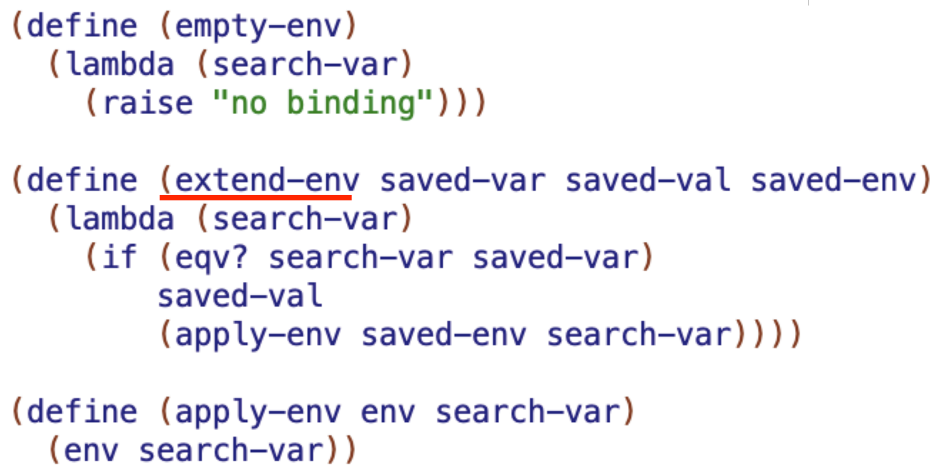
How to call `extend-env` ?



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

(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 env search-var)
  (env search-var))
```

Procedural-based Representation of Environment

How to use Environment ADT concretely ?

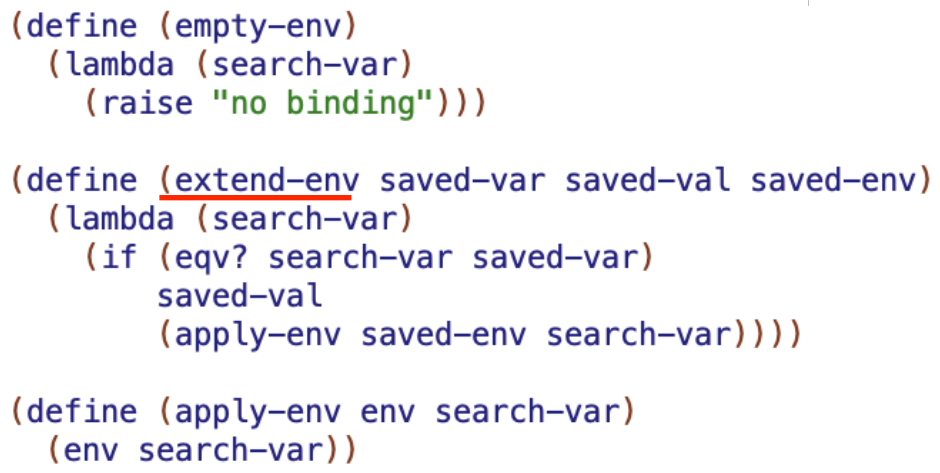


```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding")))

(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 env search-var)
  (env search-var))
```

How to call `extend-env` ?

(extend-env ‘m 3 (empty-env))

Procedural-based Representation of Environment

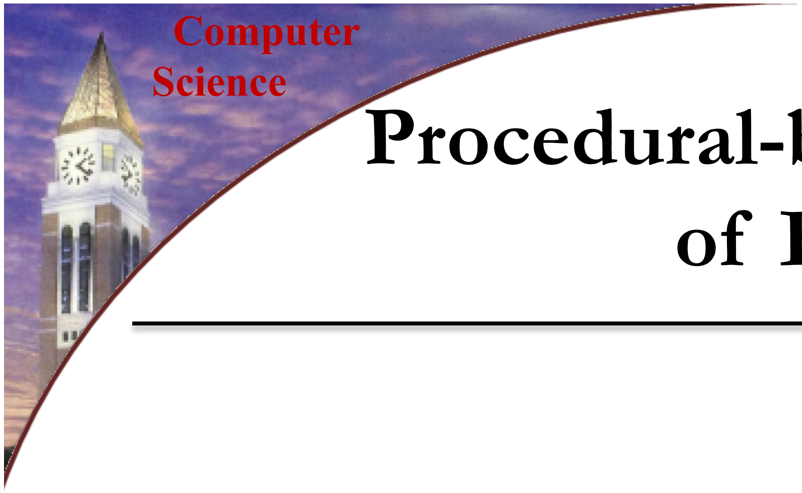


```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding")))

(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 env search-var)
  (env search-var))
```

How to use Environment ADT concretely ?

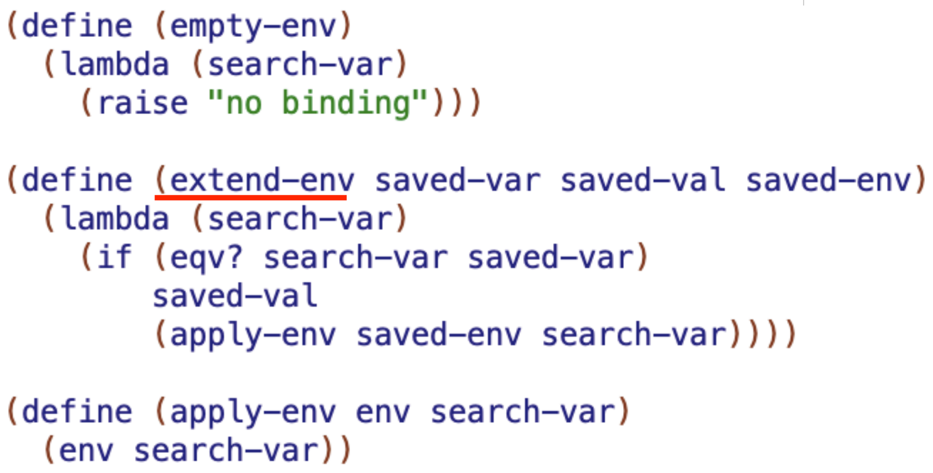
How to call `extend-env` ?

(extend-env ‘m 3 (empty-env))



Procedural-based Representation of Environment

How to use Environment ADT concretely ?



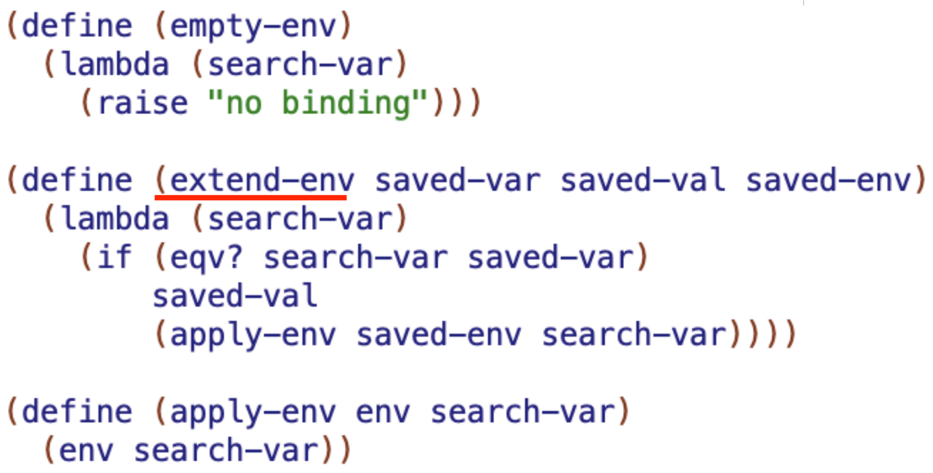
```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding")))

(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 env search-var)
  (env search-var))
```

How to call `extend-env` ?

(extend-env ‘m 3 (empty-env))

(extend-env ‘m 3
  (lambda (search-var) (raise “no binding”)))

Procedural-based Representation of Environment



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

(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 env search-var)
  (env search-var))
```

How to use Environment ADT concretely ?

How to call `extend-env` ?

(extend-env ‘m 3 (empty-env))

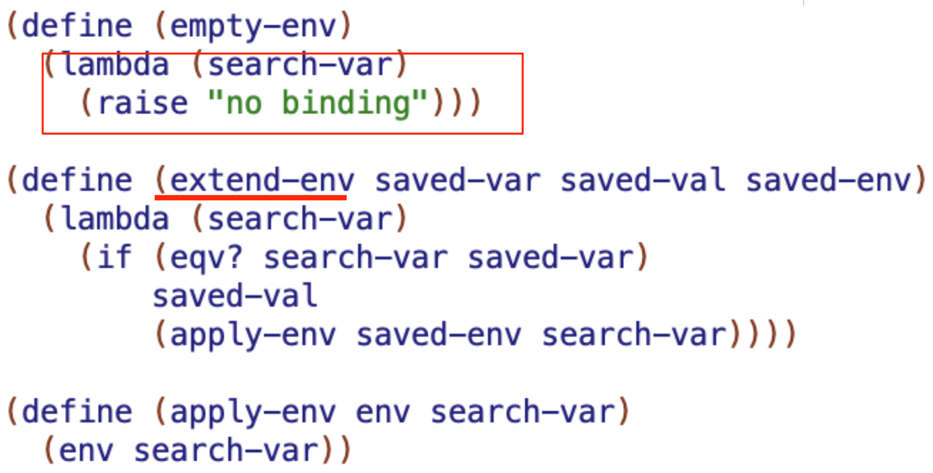
⬇

(extend-env ‘m 3

(lambda (search-var) (raise “no binding”)))

Computer Science

Procedural-based Representation of Environment



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

(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 env search-var)
  (env search-var))
```

How to use Environment ADT concretely ?

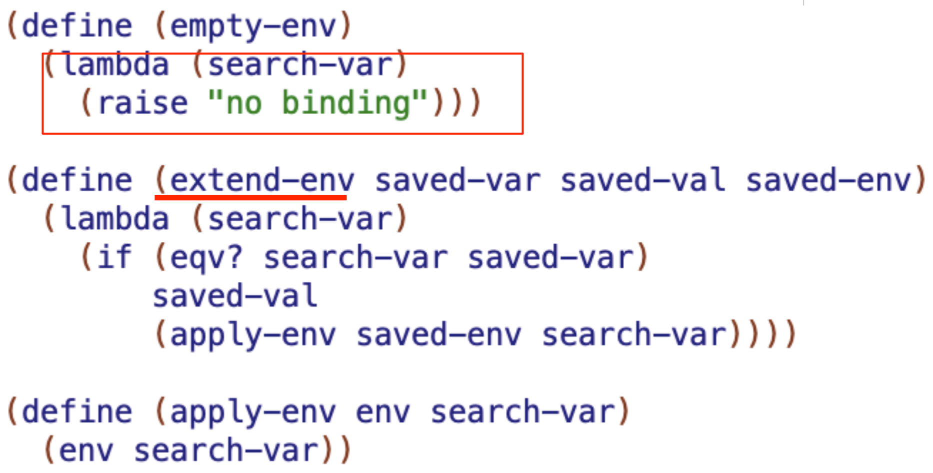
How to call `extend-env` ?

(extend-env ‘m 3 (empty-env))

⬇

(extend-env ‘m 3
(lambda (search-var) (raise “no binding”)))

Procedural-based Representation of Environment



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

(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 env search-var)
  (env search-var))
```

How to use Environment ADT concretely ?

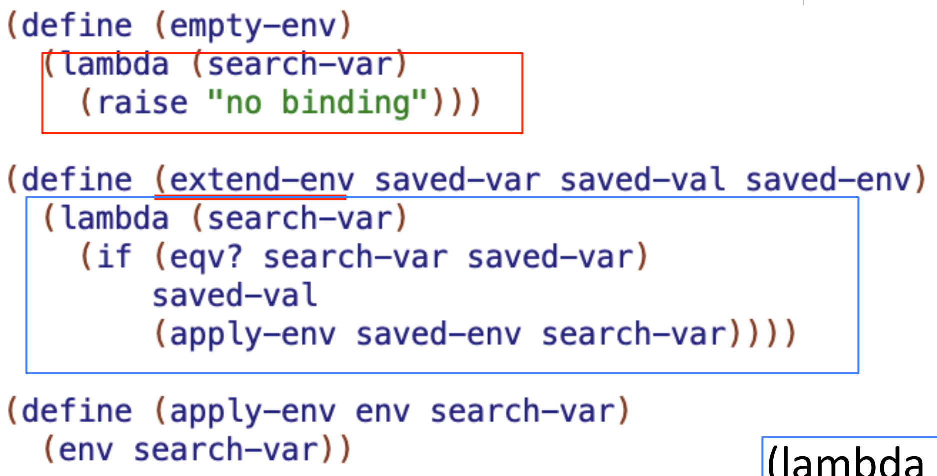
How to call `extend-env` ?

(extend-env ‘m 3 (empty-env))

(extend-env ‘m 3
 (lambda (search-var) (raise “no binding”)))

Procedural-based Representation of Environment

How to use Environment ADT concretely ?



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

(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 env search-var)
  (env search-var))
```

How to call `extend-env` ?

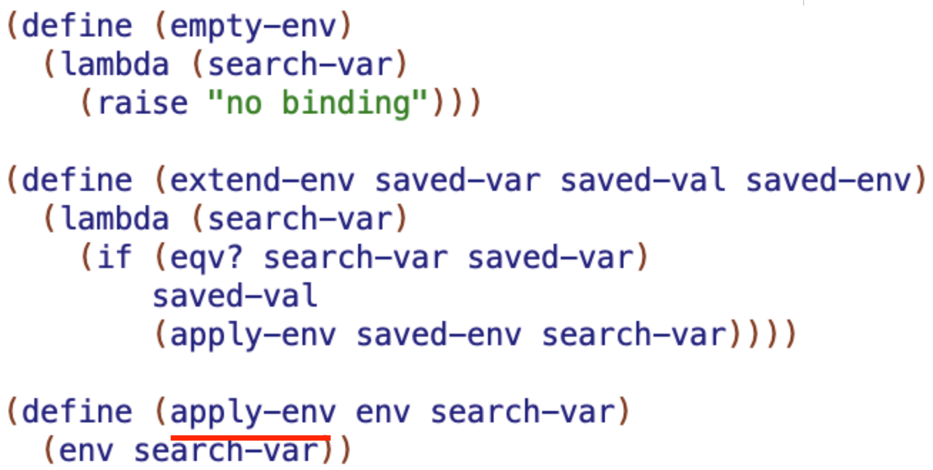
```
(extend-env ‘m 3 (empty-env) )
```

```
(extend-env ‘m 3
 (lambda (search-var) (raise “no binding”)))
```

```
(lambda (search-var)
   (if (eqv? search-var ‘m)
       3
       (apply-env  (lambda (search-var) (raise “no binding”))
                   search-var)))
```

Procedural-based Representation of Environment

How to use Environment ADT concretely ?

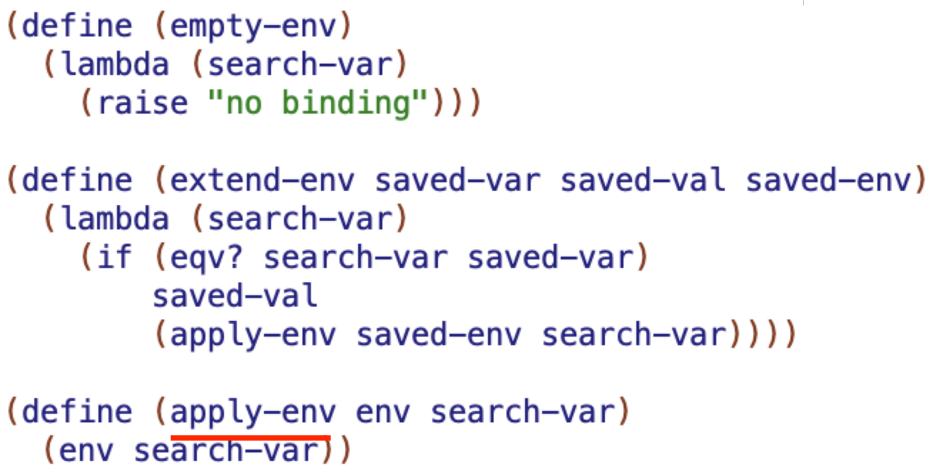


```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding")))

(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 env search-var)
  (env search-var))
```

How to call `apply-env` ?

Procedural-based Representation of Environment

How to use Environment ADT concretely ?



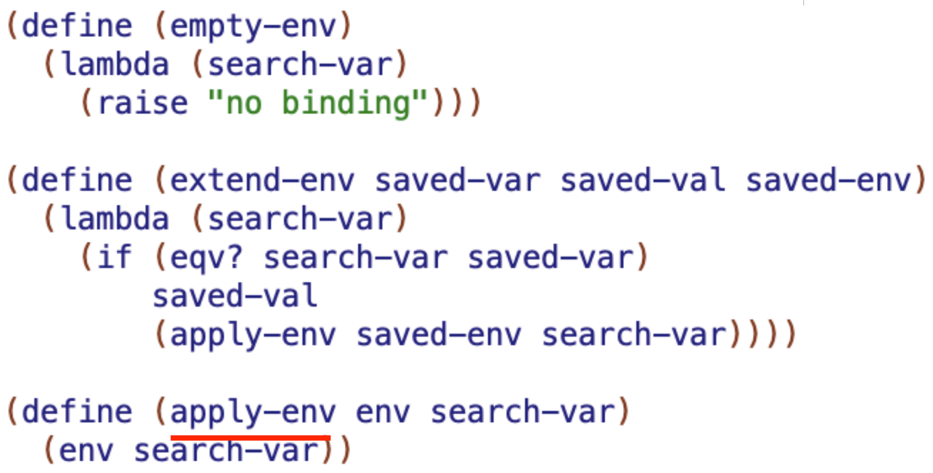
```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding")))

(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 env search-var)
  (env search-var))
```

How to call `apply-env` ?

```
(apply-env
  (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
 ‘m
)
```

Procedural-based Representation of Environment

How to use Environment ADT concretely ?



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

(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 env search-var)
  (env search-var))
```

How to call `apply-env` ?

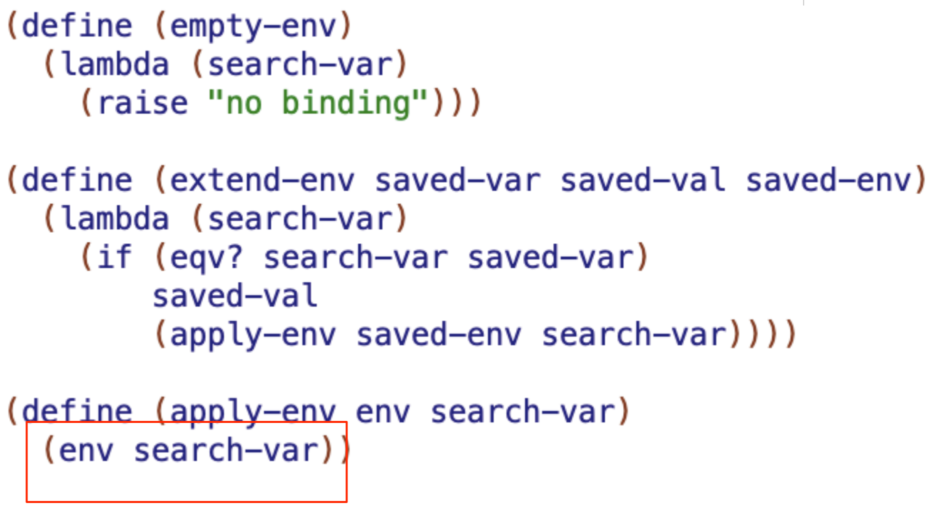
```
(apply-env
  (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
 ‘m
)
```

next

```
( (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
 ‘m
)
```

Procedural-based Representation of Environment

How to use Environment ADT concretely ?



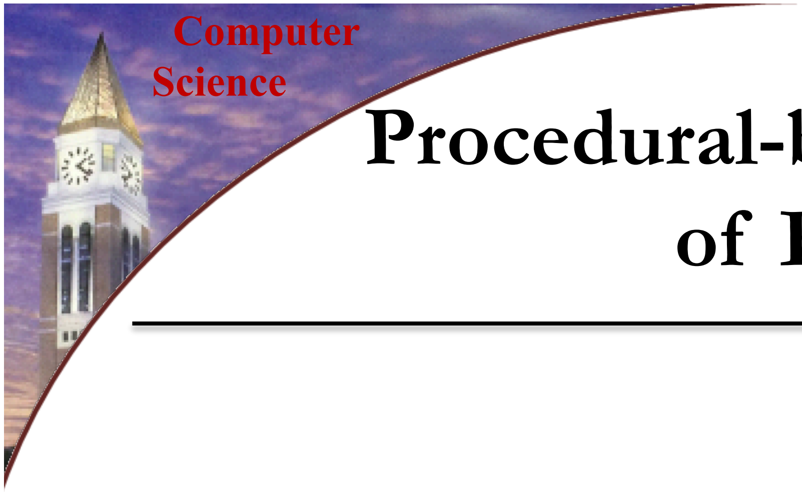
```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding")))

(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 env search-var)
  (env search-var))
```

How to call `apply-env` ?

```
(apply-env
 (extend-env ‘m 3
   (lambda (search-var) (raise “no binding”))
 )
 ‘m
)
```

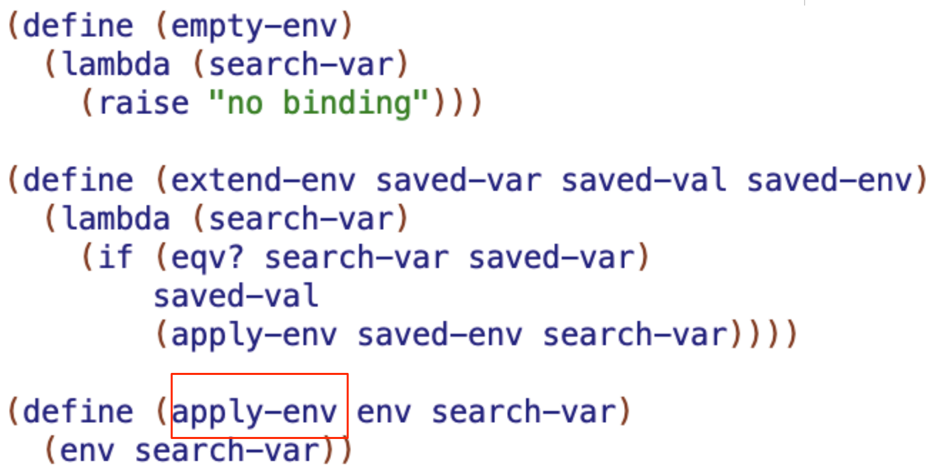
next

```
( (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
  ‘m
)
```



Procedural-based Representation of Environment

How to use Environment ADT concretely ?



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

(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 env search-var)
  (env search-var))
```

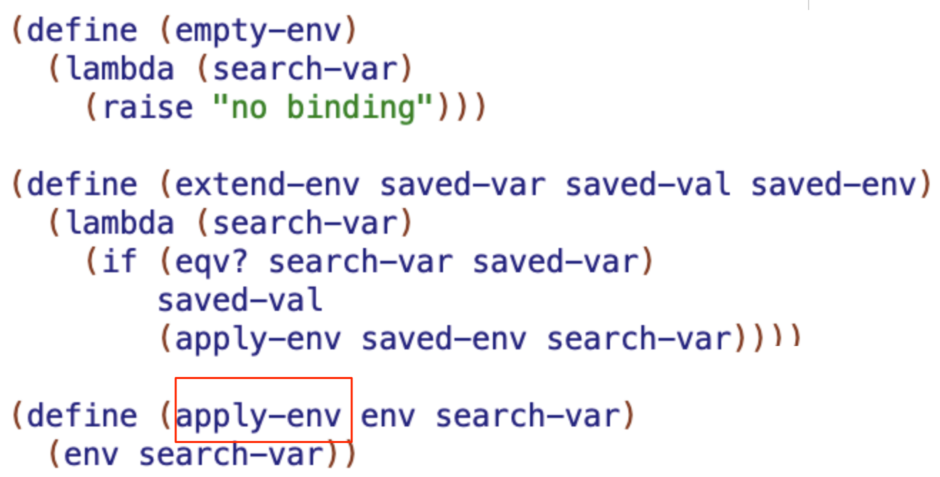
How to call `apply-env` ?

```
(apply-env
  (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
  ‘m
)
```

next

```
( (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
  ‘m
)
```

Procedural-based Representation of Environment



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

(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 env search-var)
  (env search-var))
```

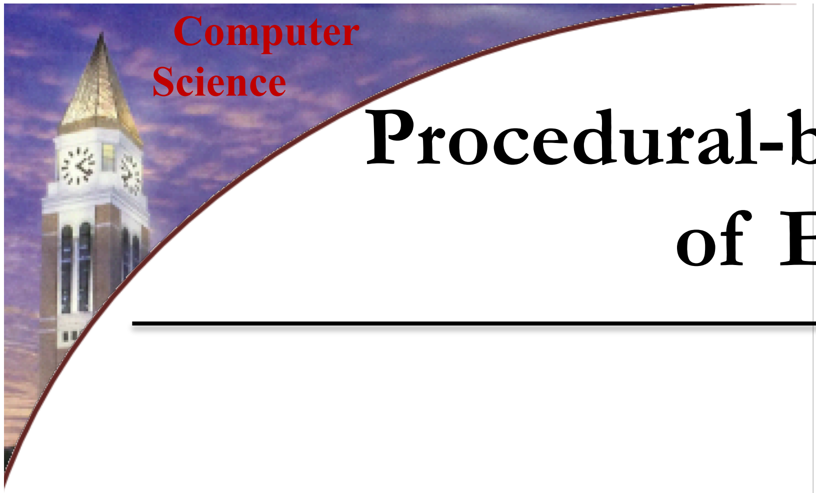
How to use Environment ADT concretely ?

How to call `apply-env` ?

next

```
( (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
 ‘m
)
```

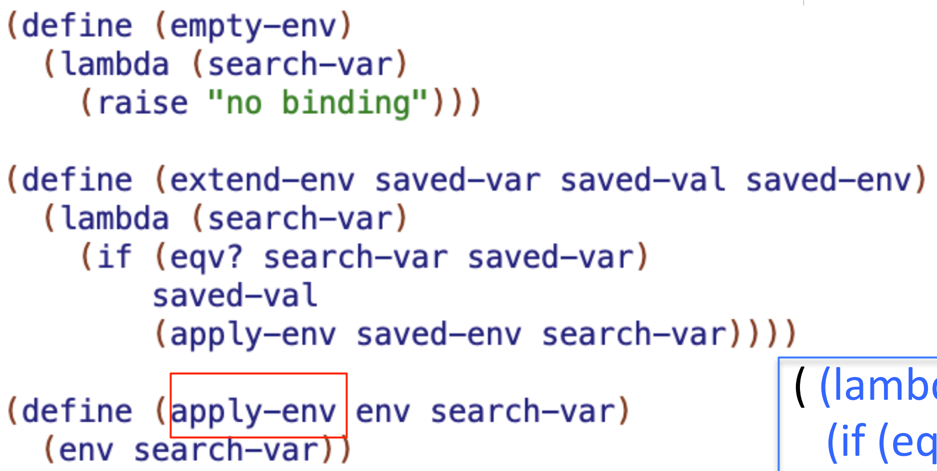
next



Procedural-based Representation of Environment

How to use Environment ADT concretely ?

How to call `apply-env` ?



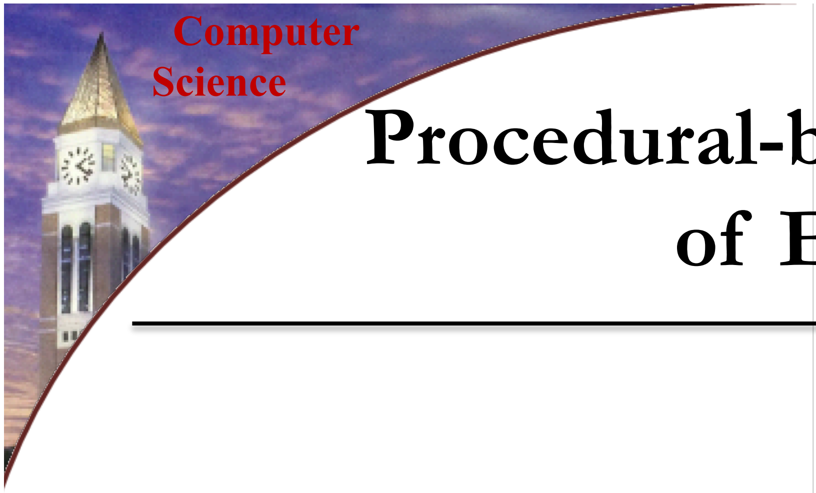
```
(define (empty-env)
  (lambda (search-var)
    (raise "no binding")))

(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 env search-var)
  (env search-var))
```

next

```
( (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
 ‘m
)
```

next

```
( (lambda (search-var)
   (if (eqv? search-var  ‘m)
      3
      (apply-env (lambda (search-var) (raise “no binding”))
                       search-var  )))
 ‘m
)
```



Procedural-based Representation of Environment

How to use Environment ADT concretely ?

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

(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 env search-var)
  (env search-var))
```

How to call `apply-env` ?

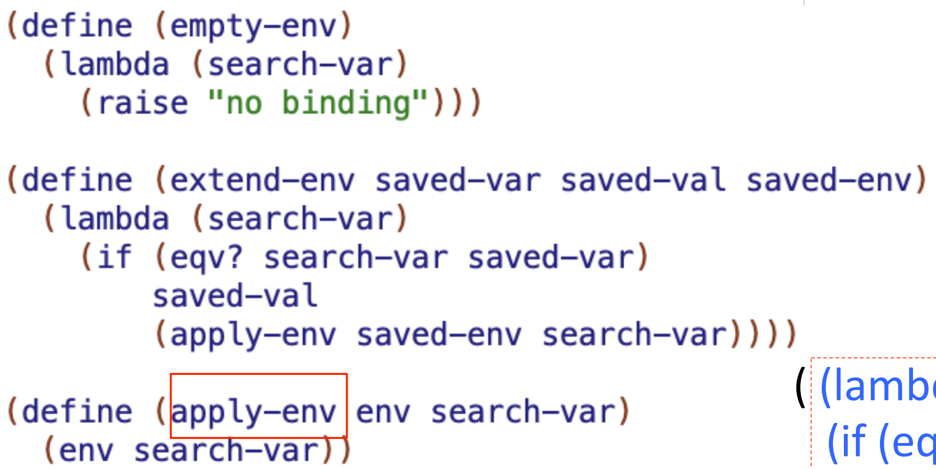
next

```
( (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
  ‘m
)
```

next

```
( (lambda (search-var)
   (if (eqv? search-var  ‘m)
      3
      (apply-env (lambda (search-var) (raise “no binding”))
                          search-var  )))
 ‘m
)
```

Procedural-based Representation of Environment



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

(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 env search-var)
  (env search-var))
```

How to use Environment ADT concretely ?

How to call `apply-env` ?

next

```
( (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
 ‘m
)
```

next

```
( (lambda (search-var)
   (if (eqv? search-var  ‘m)
      3
      (apply-env (lambda (search-var) (raise “no binding”))
                   search-var  )))
 ‘m
)
```

Procedural-based Representation of Environment

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

(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 env search-var)
  (env search-var))
```

How to use Environment ADT concretely ?

How to call `apply-env` ?

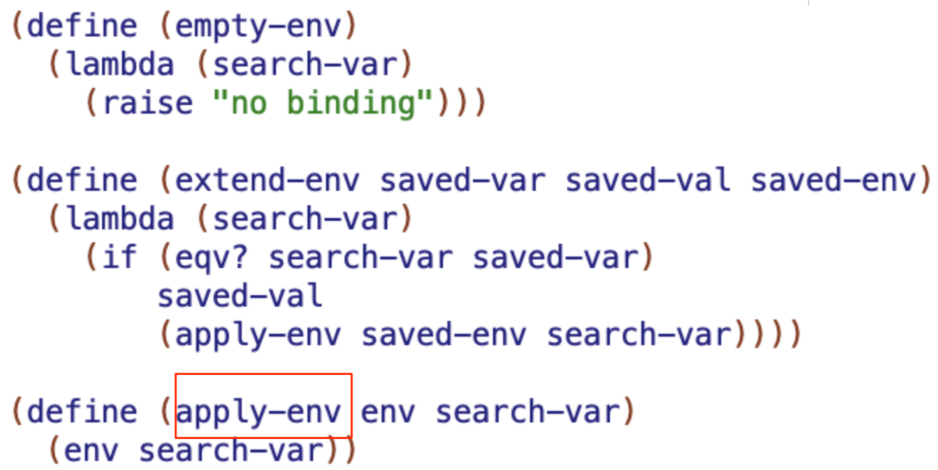
next

```
( (extend-env ‘m 3
     (lambda (search-var) (raise “no binding”))
  )
  ‘m
)
```

next

```
( (lambda (search-var)
    (if (eqv? search-var  ‘m)
      3
        (apply-env (lambda (search-var) (raise “no binding”))
                   search-var  )))
  ‘m
)
```

Procedural-based Representation of Environment



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

(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 env search-var)
  (env search-var))
```

How to use Environment ADT concretely ?

How to call `apply-env` ?

next

```
( (extend-env ‘m 3
    (lambda (search-var) (raise “no binding”))
  )
  ‘m
)
```

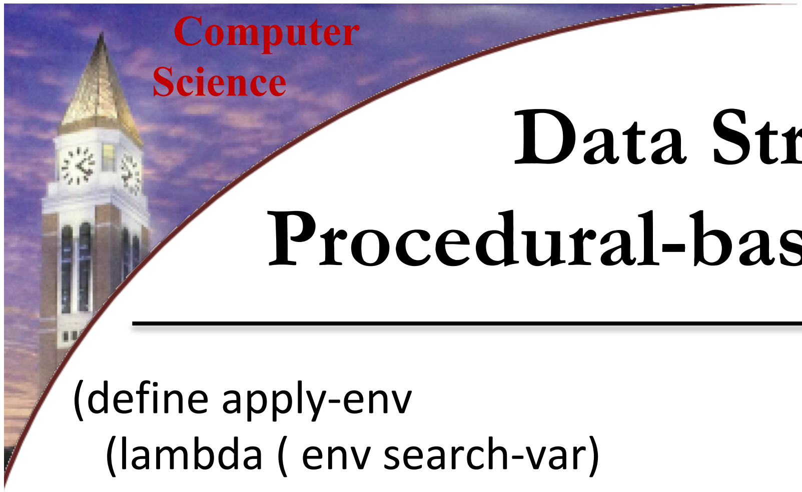
output

3

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!

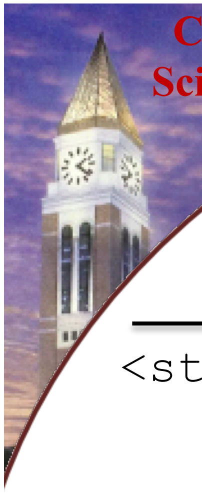
Function makes coding so MUCH simpler!

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!



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!

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-step)
)
```

```
(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-step)
                  #t
                  #f )))
)
```

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

     (st ‘right-step)
)
```

one liner

```
(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-step)
                  #t
                  #f )))
)
```

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

     (st ‘right-step)
)
```

one liner

```
(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-step)
                  #t
                  #f )))
)
```

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

     (st ‘right-step)
)
```

one liner

```
(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-step)
                  #t
                  #f )))
)
```

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

     (st ‘right-step)
)
```

one liner

```
(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-step)
                  #t
                  #f )))
)
```

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

     (st ‘right-step )
)
```

one liner

```
(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-step)
                  #t
                  #f )))
)
```

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

     (st ‘right-step)
)
```

one liner

```
(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-step)
                  #t
                  #f )))
)
```

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

     (st ‘right-step)
)
```

one liner

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

    (st  ‘extract-size   )
)
```

An Example on `right-step`

Procedural-based implementation

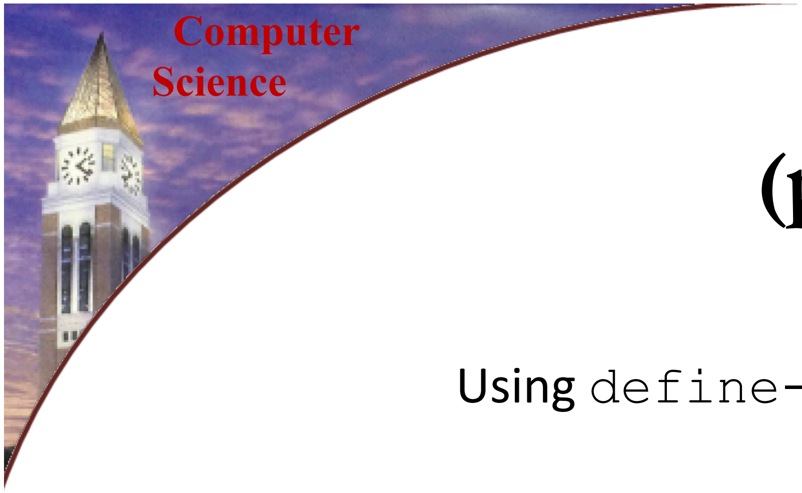
```
(define (right-step n)
        ; constructor
        (lambda (a)
          (if (= a 'extract-size)
              n
              (if (= a 'right-step)
                  #t
                  #f )))
)
```

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

     (st  'right-step)
)
```

⬅ **one liner** ➡

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

     (st  'extract-size   )
)
```

HW 4
(problem 1)

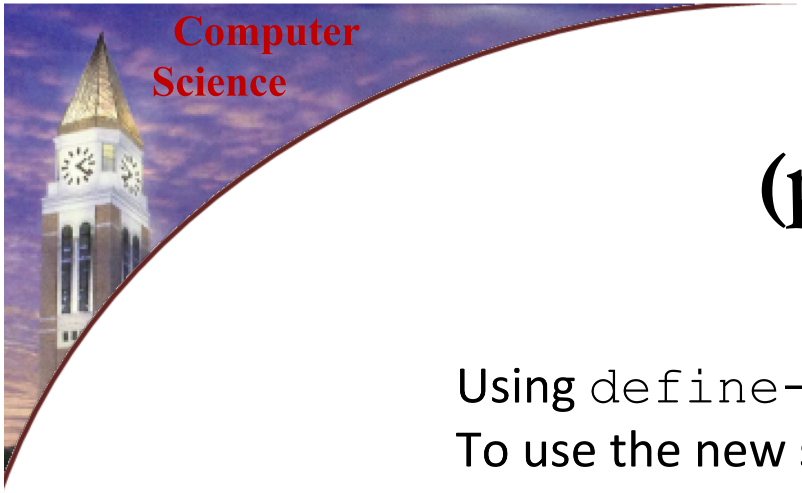


HW 4
(problem 1)

Using `define-syntax-rule` to define new syntax.

HW 4
(problem 1)

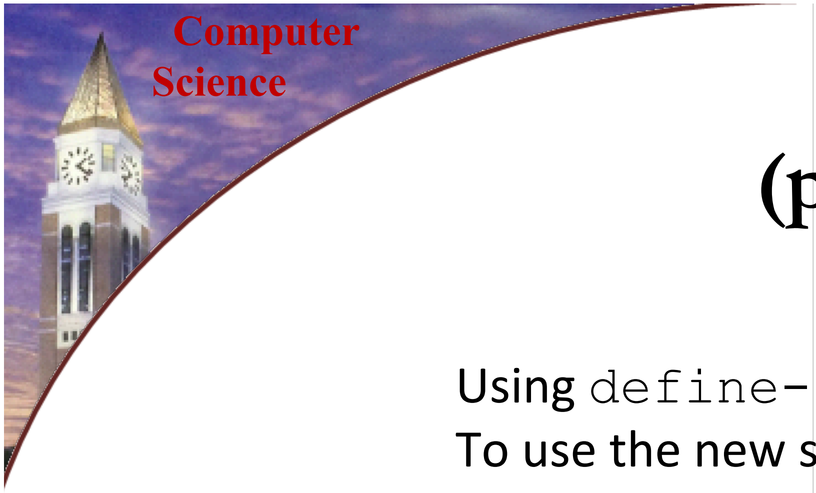
Using `define-syntax-rule` to define new syntax. To use the new syntax just defined, three things happen in order



HW 4
(problem 1)

Using `define-syntax-rule` to define new syntax.
To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating

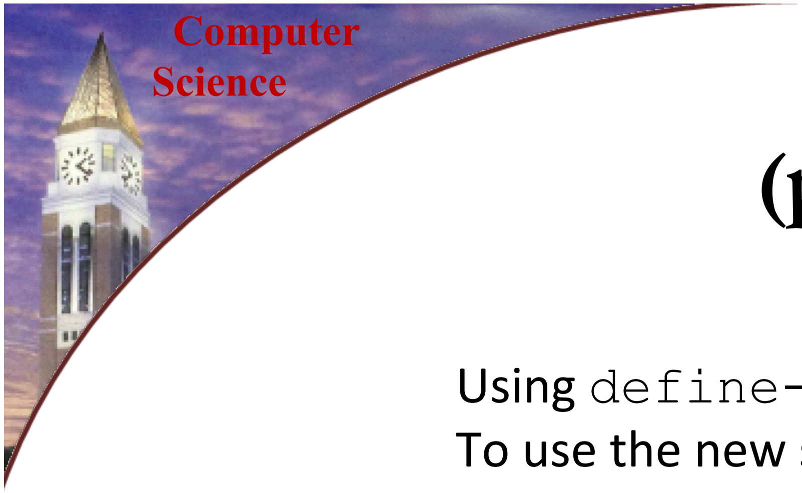


HW 4 (problem 1)

Using `define-syntax-rule` to define new syntax. To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating

```
(define-syntax-rule (for {var <- value-range} yield result)
  (map (lambda (var) result) value-range)
  )
```



HW 4
(problem 1)

Using `define-syntax-rule` to define new syntax.
To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating

```
(define-syntax-rule (for {var <- value-range} yield result)
  (map (lambda (var) result) value-range)
  )
```

HW 4 (problem 1)

Using `define-syntax-rule` to define new syntax.
To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating

use the for syntax

```
(define-syntax-rule (for {var <- value-range} yield result)
  (map (lambda (var) result) value-range)
  )
```

HW 4
(problem 1)

Using `define-syntax-rule` to define new syntax.
To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating

```
(for {a <- ‘(0 1 2 3) } yield (+ a 42)
```

use the for syntax

```
(define-syntax-rule (for {var <- value-range} yield result)
  (map (lambda (var) result) value-range)
  )
```

HW 4
(problem 1)

Using `define-syntax-rule` to define new syntax.
To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating

```
(for {a <- ‘(0 1 2 3) } yield (+ a 42)
```

use the for syntax

```
(define-syntax-rule (for {var <- value-range} yield result)
  (map (lambda (var) result) value-range)
  )
```

HW 4
(problem 1)

Using `define-syntax-rule` to define new syntax. To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating

```
(for {a <- ‘(0 1 2 3) } yield (+ a 42)
```

use the for syntax

```
(define-syntax-rule (for {var <- value-range} yield result)
  (map (lambda (var) result) value-range)
  )
```

HW 4
(problem 1)

Using `define-syntax-rule` to define new syntax.
To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating

```
(for {a <- ‘(0 1 2 3) } yield (+ a 42)
```

use the for syntax

a replaces var
‘(0 1 2 3) replaces value-range
(+ a 42) replaces result

```
(define-syntax-rule (for {var <- value-range} yield result)
  (map (lambda (var) result) value-range)
  )
```

HW 4 (problem 1)

Using `define-syntax-rule` to define new syntax. To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating

```
(for {a <- '(0 1 2 3) } yield (+ a 42)
```

use the for syntax

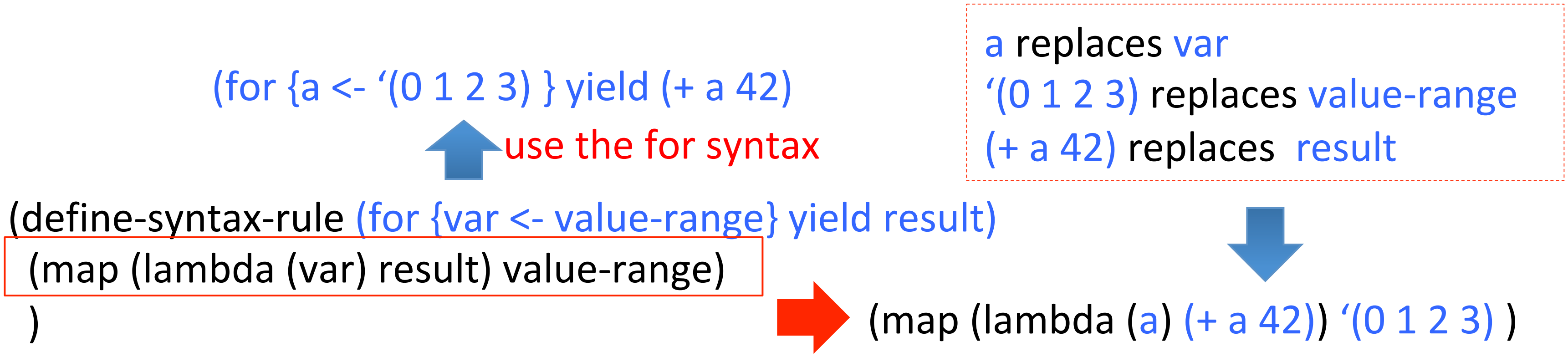
a replaces var
'(0 1 2 3) replaces value-range
(+ a 42) replaces result

```
(define-syntax-rule (for {var <- value-range} yield result)
 (map (lambda (var) result) value-range)
 )
```

HW 4
(problem 1)

Using `define-syntax-rule` to define new syntax.
To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating



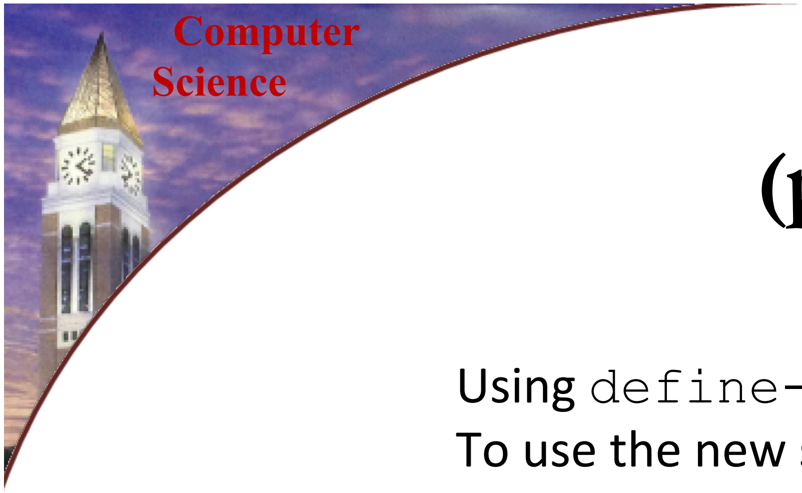
(for {a <- ‘(0 1 2 3) } yield (+ a 42)

use the for syntax

a replaces var
‘(0 1 2 3) replaces value-range
(+ a 42) replaces result

```
(define-syntax-rule (for {var <- value-range} yield result)
 (map (lambda (var) result) value-range)
 )
```

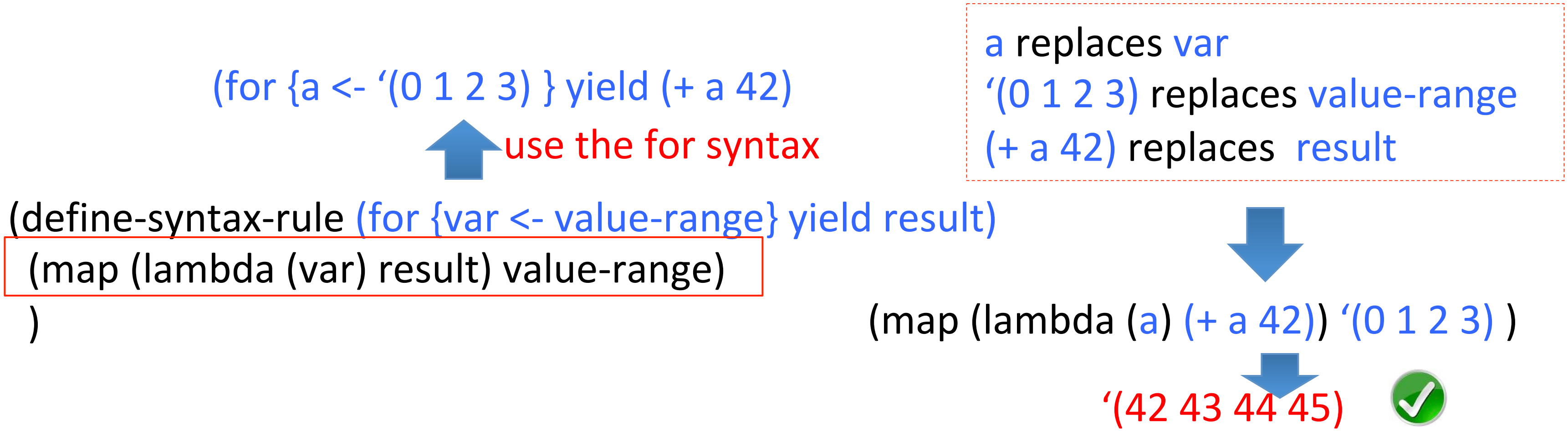
(map (lambda (a) (+ a 42)) ‘(0 1 2 3))



HW 4
(problem 1)

Using `define-syntax-rule` to define new syntax.
To use the new syntax just defined, three things happen in order

- first parsing & matching
- second substituting
- lastly calculating



```
(for {a <- '(0 1 2 3) } yield (+ a 42)
```

use the for syntax

```
(define-syntax-rule (for {var <- value-range} yield result)
  (map (lambda (var) result) value-range)
 )
```

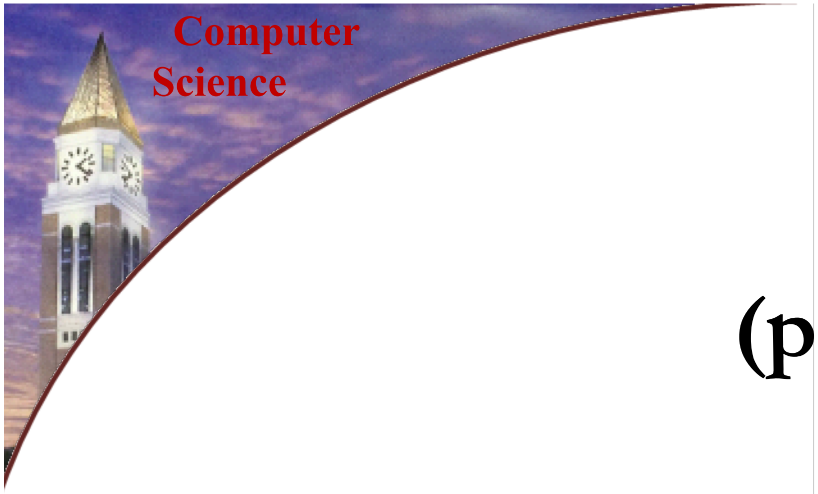
a replaces var
'(0 1 2 3) replaces value-range
(+ a 42) replaces result

```
(map (lambda (a) (+ a 42)) '(0 1 2 3) )
```

'(42 43 44 45) ✓

HW 4 (problem 2)

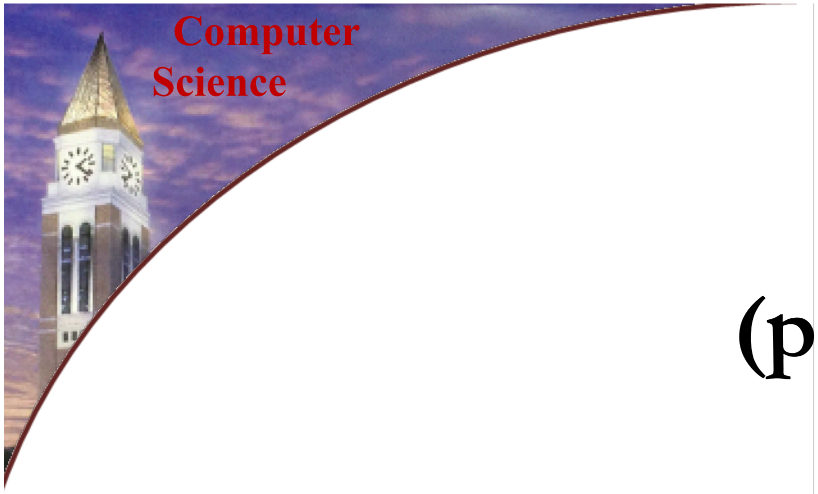
$f(x) = (x + 2)$



HW 4
(problem 2)

f(x) = (x + 2)

(lambda (x) (+ x 2))

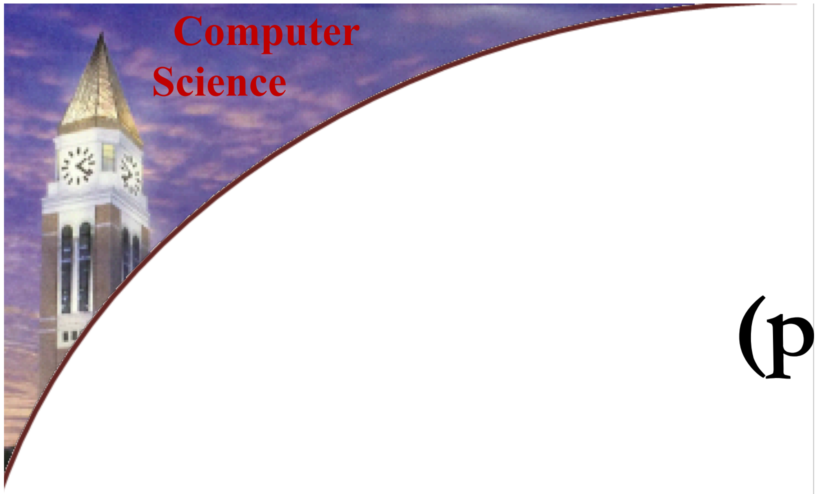


HW 4
(problem 2)

f(x) = (x + 2)

f(3 + 4)

(lambda (x) (+ x 2))

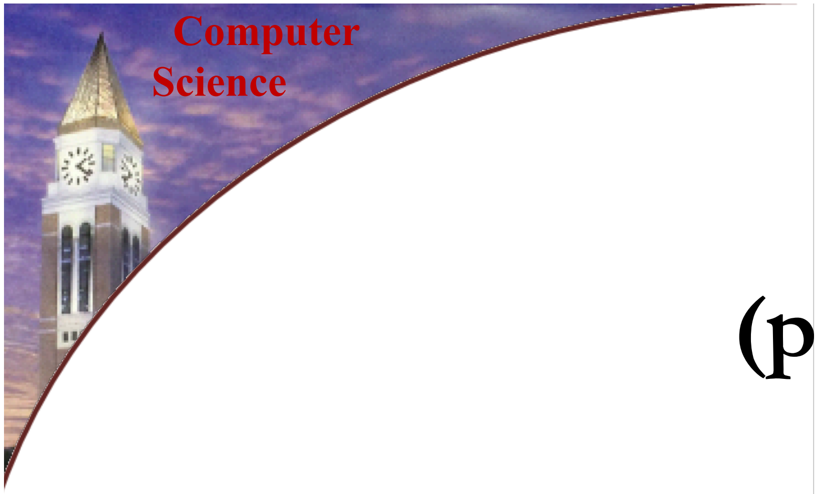


HW 4
(problem 2)

f(x) = (x + 2)

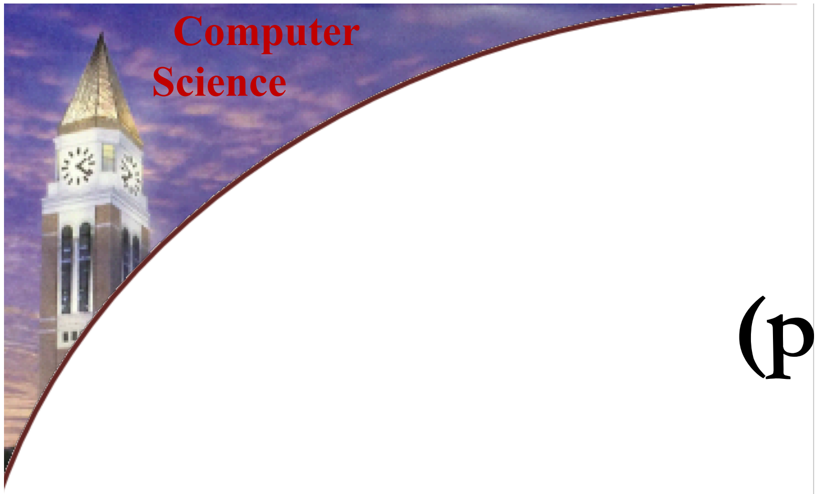
f(3 + 4)

```
(lambda (x) (+ x 2) )

(   (lambda (x) (+ x 2))     (+ 3 4) )
```



HW 4
(problem 2)

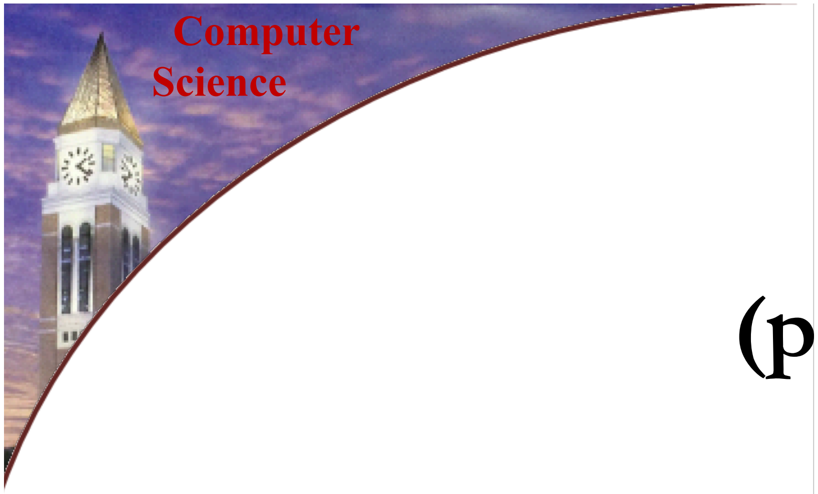
```
(lambda (x) (+ x 2) )

(   (lambda (x) (+ x 2))     (+ 3 4) )
```



HW 4
(problem 2)

(lambda (x) (+ x 2))

$$(\quad (\text{lambda}\ (x)\ \underbrace{(+\ x\ 2)}_{\text{expr 2}})\quad \underbrace{(+\ 3\ 4)}_{\text{expr 1}}\ )$$

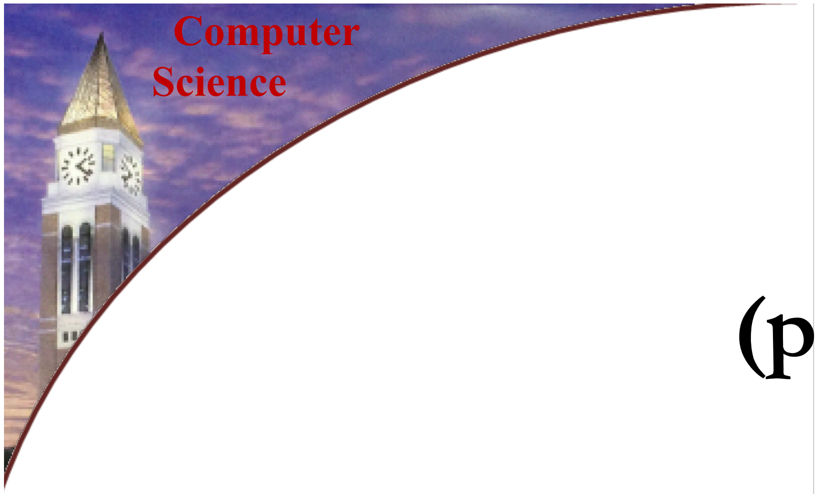


HW 4
(problem 2)

(lambda (x) (+ x 2))

$$(\quad (\text{lambda}\ (x)\ \underbrace{(+\ x\ 2)}_{\text{expr 2}})\quad \underbrace{(+\ 3\ 4)}_{\text{expr 1}}\ )$$

Which expression is run first ?
expr 1 first, then expr 2



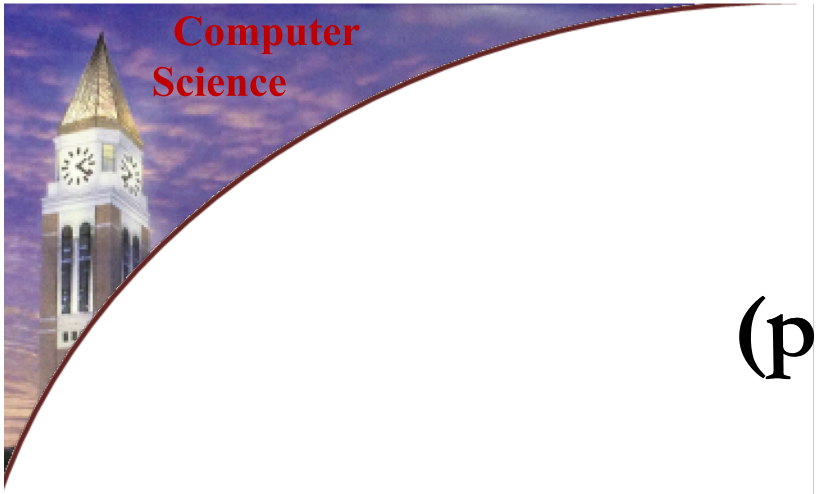
HW 4
(problem 2)

```
(lambda (x) (+ x 2) )

(   (lambda (x) (+ x 2))    (+ 3 4) )
                ‾‾‾‾‾‾‾     ‾‾‾‾‾‾‾
                 expr 2      expr 1
```

Which expression is run first ?

expr 1 first, then expr 2

✓



HW 4 (problem 2)

(lambda (x) (+ x 2))

$$\texttt{(\ \ \ (lambda (x)}\ \underbrace{\texttt{(+ }\mathbf{y}\texttt{ 2)}}_{\text{expr 2}}\texttt{)}\quad \underbrace{\texttt{(+ 3 4)}}_{\text{expr 1}}\texttt{)}$$

Which expression is run first ?
expr 1 first, then expr 2

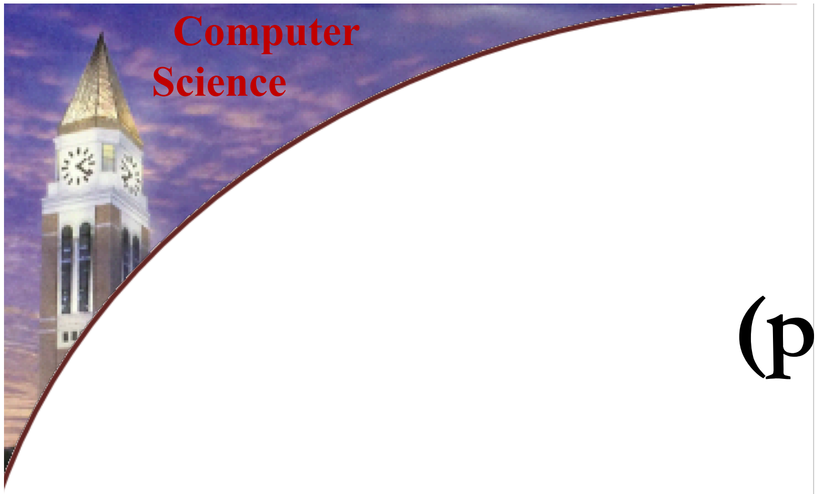
HW 4
(problem 2)

(lambda (x) (+ x 2))

$$\text{((lambda (x) } \underbrace{\text{(+ } \mathbf{y} \text{ 2)}}_{\text{expr 2}} \text{)} \quad \underbrace{\text{(+ 3 4)}}_{\text{expr 1}} \text{)}$$

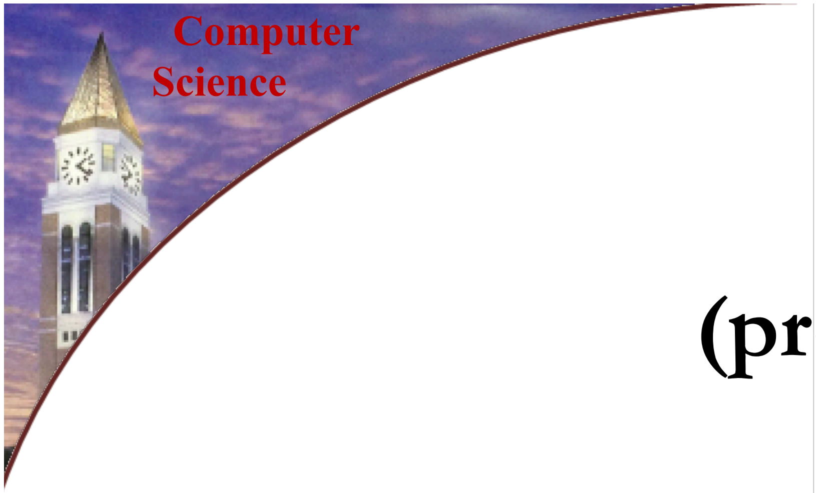
Which expression is run first ?
expr 1 first, then expr 2

✓



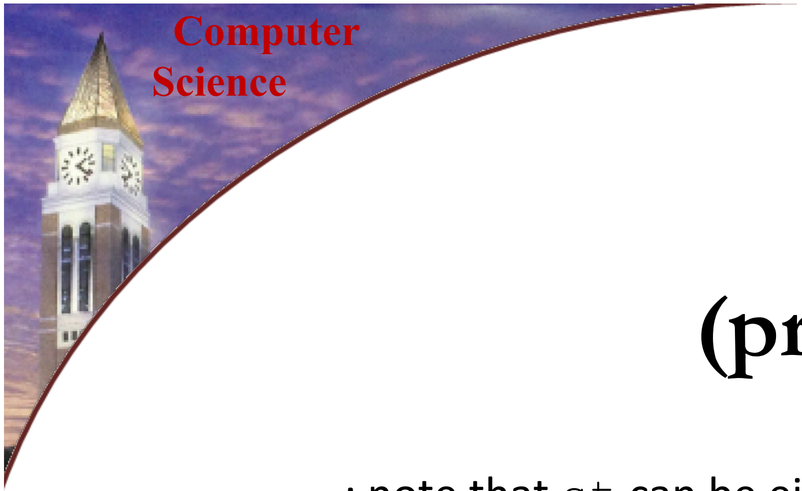
HW 4 (problem 3)

We use black board!

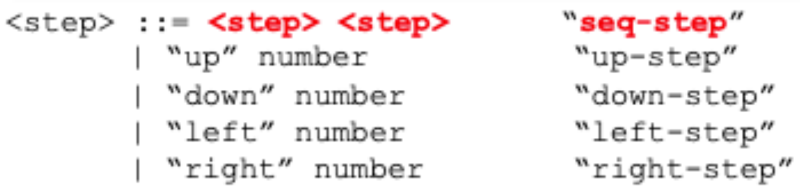


HW 4
(problem 3.b)

```
(define (move start-p st)
      ; your code here
)
```



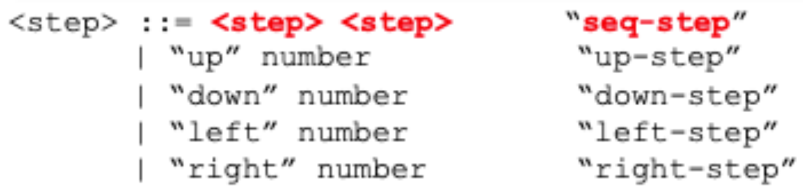
HW 4
(problem 3.b)



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

; note that `st` can be either single-step or `seq-step`

```
(define (move start-p st)
       ; your code here
)
```

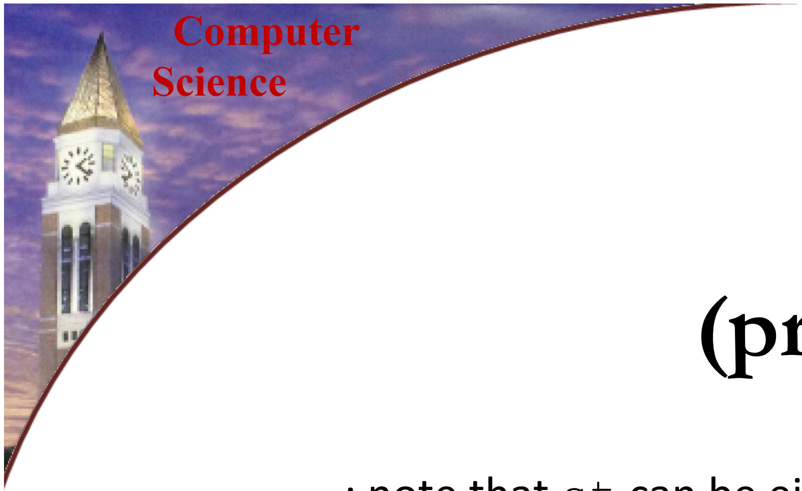


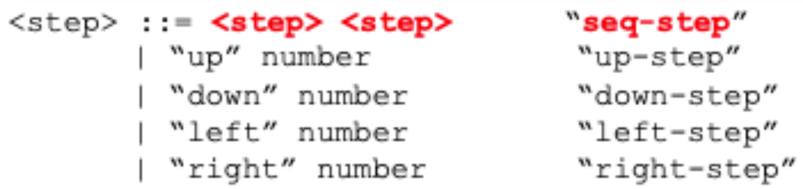
```
<step> ::= <step> <step>        "seq-step"
         | "up" number          "up-step"
         | "down" number        "down-step"
         | "left" number        "left-step"
         | "right" number       "right-step"
```

HW 4
(problem 3.b)

; note that `st` can be either single-step or `seq-step`

```
(define (move start-p st)
      ; base case
      ; recursive case
)
```





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

HW 4
(problem 3.b)

; note that `st` can be either single-step or `seq-step`

```
(define (move start-p st)
       ; base case
       ; for example if st is an up-step, using up-step's
       ;  predicate, then return
       ; ((getx start-p), ((st->n st)+(getY start-p)) )

       ; recursive case
       ;  ??
)
```