8. Interfaces & Representation

lecture 08 -- Interfaces & Representation.pdf

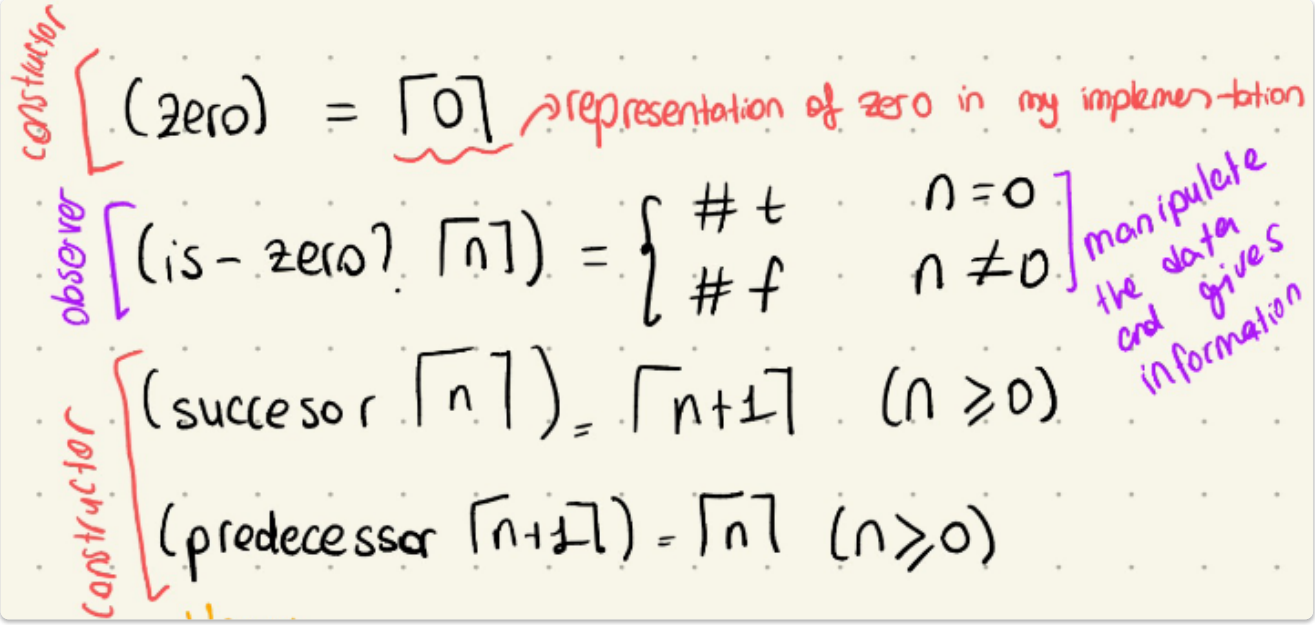
Interface vs Implementation

Interface is like the function definitons and the specific grammar they follow.
The implementation is the inner workings of these functions, how they compute the outputs, the design of the code, the abstracted functions they use, and so on. Basically the implementation is hidden from the user apart from the performance.
TLDR: Interface is everything except the body of the functions, and the body is the implementation.

Representation vs. Value

Natural Numbers

$$\lceil v \rceil \text{ The representation of data } v.$$



In here, we define $\lceil 0 \rceil$ ourselfs. It can be an empty list, or a boolean false. $\lceil . \rceil$ the ceiling symbols are just a layer of abstraction which show us the interface and leave the implementation to us.

Implementing Plus Via Natural Numbers Interface

```
(define plus
        (lambda (x y)
                (if (is-zero? x)
                        y
                        (successor (plus (predecessor x) y))
```

```
                )
        )
)
```

$$(\text{plus } \lceil x \rceil \ \lceil y \rceil) = \lceil x + y \rceil$$

Implementation of Natural Numbers

Unary representation:

$$\lceil 0 \rceil = (\ )$$
$$\lceil n + 1 \rceil = (\#t \quad \lceil n \rceil)$$

Scheme Implementation:

```
(define (zero)
        '()
)
(define (is-zero? n)
        (null? n)
)
(define (successor n)
        (cons #t n)
)
(define (predecessor n)
        (cdr n)
)
```

Another scheme imlementation:

```
(define (zero)
        0
)
(define (is-zero? n)
        (zero? n)
)
(define (successor n)
        (+ n 1)
)
(define (predecessor n)
        (- n 1)

)
```

Representation Strategies

- Data Structure Representation

- Procedural Representation

1. Start with the interface
2. Introduce Implementation

The Environment Interface

Environment is the Function that maps variables to values.

$$\{(\text{var}_1,\ \text{val}_1),\ \ldots\ ,(\text{var}_n,\ \text{val}_n)\}$$
$$f(\text{var}_1) = \text{val}_1$$

The interface:

$$\begin{aligned}(\text{empty-env}) &= \lceil\emptyset\rceil \\ (\text{apply-env}\ \lceil f\rceil\ var) &= f(var) \\ (\text{extend-env}\ var\ v\ \lceil f\rceil) &= \lceil g\rceil \\ \text{where}\ \ g(\text{var1}) &= \begin{cases} v & \text{if var1} = \text{var} \\ f(\text{var1}) & \text{otherwise} \end{cases}\end{aligned}$$

The grammar:

$$\begin{aligned}\text{Env-exp} &::= \{\text{empty-env}\} \\ &::= \{\text{extend-env}\ \ Identifier\ \ Scheme\text{-}value\ Env\text{-}exp\}\end{aligned}$$

Implementation:

$$\begin{aligned}Env &= (\text{empty-env} \mid (\text{extend-env}\ \ Var\ \ SchemeVal\ \ Env)) \\ Var &= Sym\end{aligned}$$

Scheme Implementation:

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

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

(define apply-env
  (lambda (env search-var)
    (cond
      ((eqv? (car env) 'empty-env)
       (display "var not found"))

      ((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
        (display "invalid env")))))
```

Procedural Implementation:

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (display "variable not found"))))

(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)))
```

Usage:

```
(define e
  (extend-env 'd 6
     (extend-env 'y 8
        (extend-env 'x 7
          (extend-env 'y 14
             (empty-env))))))

(e 'd) -> 6
(e 'y) -> 8
(e 'x) -> 7
```