

UMass Boston Computer Science

CS450 High Level Languages (section 2)

More Kinds of Data Definitions

Wednesday, September 18, 2024

Logistics

- HW 1 out
 - due: Mon 9/23 12pm (noon) EST
- Course web site:
 - See The Design Recipe section
 - Lecture code (see lecture03.rkt) may occasionally be posted

Last
Time

Design Recipe, Step 1: Data Design


Create **Data Definitions**

- Describes the types of data that the program operates on
- Has 4 parts:
 1. **Name**
 2. Description of **all possible values** of the data
 3. **Interpretation** explaining the real world concepts the data represents
 4. **Predicate** returning true if the given value is in the Data Definition

Kinds of Data Definitions

- Basic data
- • Intervals
- Enumerations
- Itemizations

```
template<typename T>
class Array {
    //...
    int size;
    T* array;
    T &operator[](int index) {
        if(index >= size || index < 0)
            throw OUT_OF_RANGE; // #define OUT_OF_RANGE 0x0A
        return array[index];
    }
}
```



Interval Data Definitions

Is this what we want?

It depends (on our application)!
(Data representations are crucial
because they determine what the
rest of the program looks like)

```
;; An AngleD is a number in [0, 360)
;; interp: An angle in degrees
(define (AngleD? deg)
  (and (>= deg 0) (< deg 360)))
```

```
;; An AngleR is a number in [0 2π)
;; interp: An angle in radians
(define (AngleR? r)
  (and (>= r 0) (< r (* 2 pi))))
```

```
;; deg->rad: AngleD -> AngleR
;; Converts the given angle in degrees to radians
```

Function Recipe Steps 1-3:
name, signature, description

```
(define/contract (deg->rad deg)
  (-> AngleD? AngleR?)
  (* deg (/ pi 180)))
```

Step 5: Code

Not allowed by data def!
but should be ok?

```
(check-equal? (deg->rad 0) 0)
(check-equal? (deg->rad 90) (/ pi 2))
(check-equal? (deg->rad 180) pi)
```

Step 4: Examples

```
(check-equal? (deg->rad 360) 0) ; ???
(check-equal? (deg->rad 360) (* 2 pi)) ; ???
```

Step 6: Tests



Kinds of Data Definitions

- Basic data
- Intervals
- • Enumerations
- Itemizations

```
enum season { spring, summer, autumn, winter };
```

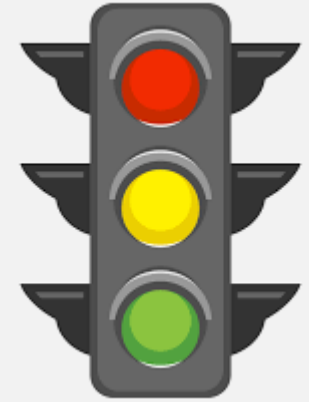


```
enum Colours {  
    RED = 'RED',  
    YELLOW = 'YELLOW',  
    GREEN = 'GREEN'  
}
```

Enumeration Data Definitions

```
;; A TrafficLight is one of:  
;; - RED-LIGHT  
;; - GREEN-LIGHT  
;; - YELLOW-LIGHT  
;; Interpretation: Represents possible colors of a traffic light  
(define RED-LIGHT "RED")  
(define GREEN-LIGHT "GREEN")  
(define YELLOW-LIGHT "YELLOW")
```

NOTE: this is not the only possible data definition.
Is there a better one?



```
(define (red-light? x) (string=? x RED-LIGHT))  
(define (green-light? x) (string=? x GREEN-LIGHT))  
(define (yellow-light? x) (string=? x YELLOW-LIGHT))
```

```
(define (TrafficLight? x)  
  (or (red-light? x)  
      (green-light? x)  
      (yellow-light? x)))
```

Need to add an extra step to **Data Design Recipe**

Design Recipe, Step 1: Data Design

Create **Data Definitions**

- Describes the types of data that the program operates on
 - Has 4 parts:
 1. **Name**
 2. Description of **all possible values** of the data
 3. **Interpretation** explaining the real world concepts the data represents
 4. **Predicate** evaluates to true, if the given value is in the data definition
- ➔ • If needed, also define predicates for each **enumeration** or **itemization**
(some languages do this implicitly for you, Racket does not)

Enumeration Data Definitions

cond is only allowed in functions that process enumeration (or itemization) data!

```
;; A TrafficLight is one of:  
;; - RED-LIGHT  
;; - GREEN-LIGHT  
;; - YELLOW-LIGHT  
;; Interpretation: Represents possible colors of a traffic light  
(define RED-LIGHT "RED")  
(define GREEN-LIGHT "GREEN")  
(define YELLOW-LIGHT "YELLOW")
```

The data and function have the same structure!

```
;; next-light: TrafficLight -> TrafficLight  
;; Computes the next light after the given one  
(define (next-light light)  
  (cond  
    [(red-light? light) GREEN-LIGHT]  
    [(green-light? light) YELLOW-LIGHT]  
    [(yellow-light? light) RED-LIGHT]))
```

Function Recipe Steps 1-3:
name, signature, description

Designing data first makes writing function (code) easier!

Step 5: Code

(keep order the same)

```
(check-equal? (next-light RED-LIGHT) GREEN-LIGHT)  
(check-equal? (next-light GREEN-LIGHT) YELLOW-LIGHT)  
(check-equal? (next-light YELLOW-LIGHT) RED-LIGHT)
```

Step 4: Examples

Last
Time

Function Design Recipe

1. **Name**
2. **Signature** – types of the function input(s) and output
3. **Description** – explain (in English prose) the function behavior
4. **Examples** – show (using `rackunit`) the function behavior
5. **Code** – implement the rest of the function (arithmetic)
6. **Tests** – check (using `rackunit`) the function behavior

Function Design Recipe

1. **Name**
2. **Signature** – types of the function input(s) and output
3. **Description** – explain (in English prose) the function behavior
4. **Examples** – show (using `rackunit`) the function behavior
5. **Template** – sketch out the function structure (using input's Data Definition)
6. **Code** – implement the rest of the function (arithmetic)
7. **Tests** – check (using `rackunit`) the function behavior

Enumeration Data Definitions

```
;; A TrafficLight is one of:  
(define RED-LIGHT "RED")  
(define GREEN-LIGHT "GREEN")  
(define YELLOW-LIGHT "YELLOW")  
;; Interpretation: Represents possible colors of a traffic light  
(define (red-light? x) (string=? x RED-LIGHT))  
(define (green-light? x) (string=? x GREEN-LIGHT))  
(define (yellow-light? x) (string=? x YELLOW-LIGHT))
```

A function's template is completely determined by the input's **Data Definition**

```
;; next-light: TrafficLight -> TrafficLight  
;; Computes the next light after the given one
```

```
(define (next-light light)  
  (cond  
    [(red-light? light) ...]  
    [(green-light? light) ...]  
    [(yellow-light? light) ...]))
```

(keep order the same)

Step 5: **Code Template**

Step 6: **Code** (fill in the "..." with arithmetic)

Kinds of Data Definitions

- Basic data
- Intervals
- Enumerations
- • Itemizations

(Generalized enumeration)

Itemization Data Definitions (Generalized enumeration)

2024 tax brackets

Tax rate	Single filers	Married couples filing jointly	Married couples filing separately	Head of household
10%	\$11,600 or less	\$23,200 or less	\$11,600 or less	\$16,550 or less
12%	\$11,601 to \$47,150	\$23,201 to \$94,300	\$11,601 to \$47,150	\$16,551 to \$63,100
22%	\$47,151 to \$100,525	\$94,301 to \$201,050	\$47,151 to \$100,525	\$63,101 to \$100,500
24%	\$100,526 to \$191,950	\$201,051 to \$383,900	\$100,526 to \$191,150	\$100,501 to \$191,150
32%	\$191,951 to \$243,725	\$383,901 to \$487,450	\$191,151 to \$243,725	\$191,151 to \$243,700
35%	\$243,726 to \$609,350	\$487,451 to \$731,200	\$243,276 to \$365,600	\$243,701 to \$609,350
37%	\$609,351 or more	\$731,201 or more	\$365,601 or more	\$609,351 or more

Source: Internal Revenue Service

;; A Salary is one of:

```
;; [0, 11600]
;; [11601 47150]
;; [47151 100525]
;; ...
```

;; Interp: Salary in USD,
;; split by 2024 2024 tax bracket

```
(define (10%-bracket? salary)
  (and (>= salary 0) (<= salary 11600)))
(define (12%-bracket? salary)
  (and (>= salary 11601) (<= salary 47150)))
;; ...
```

The data and function have the same structure!

else is fallthrough case

```
;; taxes-owed: Salary -> USD
;; computes federal income tax owed in 2024
(define (taxes-owed salary)
  (cond
    [(10%-bracket? salary) ....]
    [(12%-bracket? salary) ....]
    [else ....]))
```

Some Pre-defined Enumerations

```
; A KeyEvent is one of:  
; - 1String  
; - "left"  
; - "right"  
; - "up"  
; - ...
```

```
; A MouseEvent is one of these Strings:  
; - "button-down"  
; - "button-up"  
; - "drag"  
; - "move"  
; - "enter"  
; - "leave"
```

```
; WorldState KeyEvent -> ...  
(define (handle-key-events w ke)  
  (cond  
    [(= (string-length ke) 1) ...]  
    [(string=? "left" ke) ...]  
    [(string=? "right" ke) ...]  
    [(string=? "up" ke) ...]  
    [(string=? "down" ke) ...]  
    ...))
```

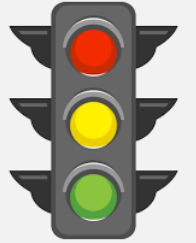
Template

```
;; handle-mouse: WorldState Coordinate Coordinate MouseEvent -> WorldState  
;; Produces the next WorldState  
;; from the given Worldstate, mouse position, and mouse event  
(define (handle-mouse w x y evt)  
  (cond  
    [(string=? evt "button-down") ....]  
    [(string=? evt "button-up") ....]  
    [else ....]))
```

Design Recipe allows combining cases if they are handled the same

```
; A 1String is a String of length 1,  
; including  
; - "\\" (the backslash),  
; - " " (the space bar),  
; - "\t" (tab),  
; - "\r" (return), and  
; - "\b" (backspace).  
; interpretation represents keys on the keyboard
```

In-class exercise: **big-bang** practice



- Create a big-bang traffic light simulator that changes on a mouse click (“button-down” event)

- Data Definition choice?
 - Pros?
 - Cons?

```
;; A TrafficLight is one of:  
(define RED-LIGHT "RED")  
(define GREEN-LIGHT "GREEN")  
(define YELLOW-LIGHT "YELLOW")  
;; Interpretation: Represents possible colors of a traffic light  
(define (red-light? x) (string=? x RED-LIGHT))  
(define (green-light? x) (string=? x GREEN-LIGHT))  
(define (yellow-light? x) (string=? x YELLOW-LIGHT))
```

```
;; A TrafficLight2 is one of:  
(define GREEN-L 0)  
(define YELLOW-L 1)  
(define RED-L 2)  
;; Interp: represents a traffic light state  
(define (red-L? li) (= li RED-L))  
(define (green-L? li) (= li GREEN-L))  
(define (yellow-L? li) (= li YELLOW-L))
```

Submitting

1. File: `in-class-09-18-<Lastname>-<Firstname>.rkt`
2. Join the in-class team: [cs450f24/teams/in-class](https://github.com/cs450f24/teams/in-class)
3. Commit to repo: `cs450f24/in-class-09-18`
 - (May need to `merge/pull` + `rebase` if someone pushes before you)