UMass Boston Computer Science

CS450 High Level Languages (section 2)

Compound Data Definitions

Monday, September 25, 2023

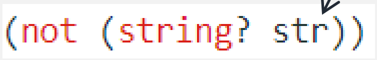
Logistics

- HW 1 in
  - ~~due: Sun 9/24 11:59 pm EST~~
  - Files should not start `big-bang` loop automatically

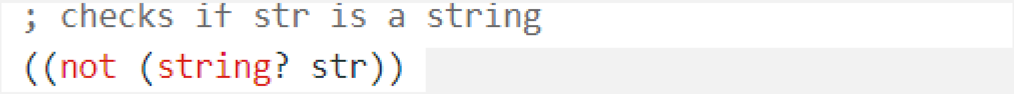
- HW 2 out
  - due: Sun 10/1 11:59 pm EST

- STYLE notes
  - Use comments to explain code if needed, BUT …
    - … the best code needs no comments
  - Redundant comments makes code harder to read
    - More comments ≠ "better"
  - Also, no commented-out code

(not a great variable name)



```
(not (string? str))
```



```
; checks if str is a string
((not (string? str))
```

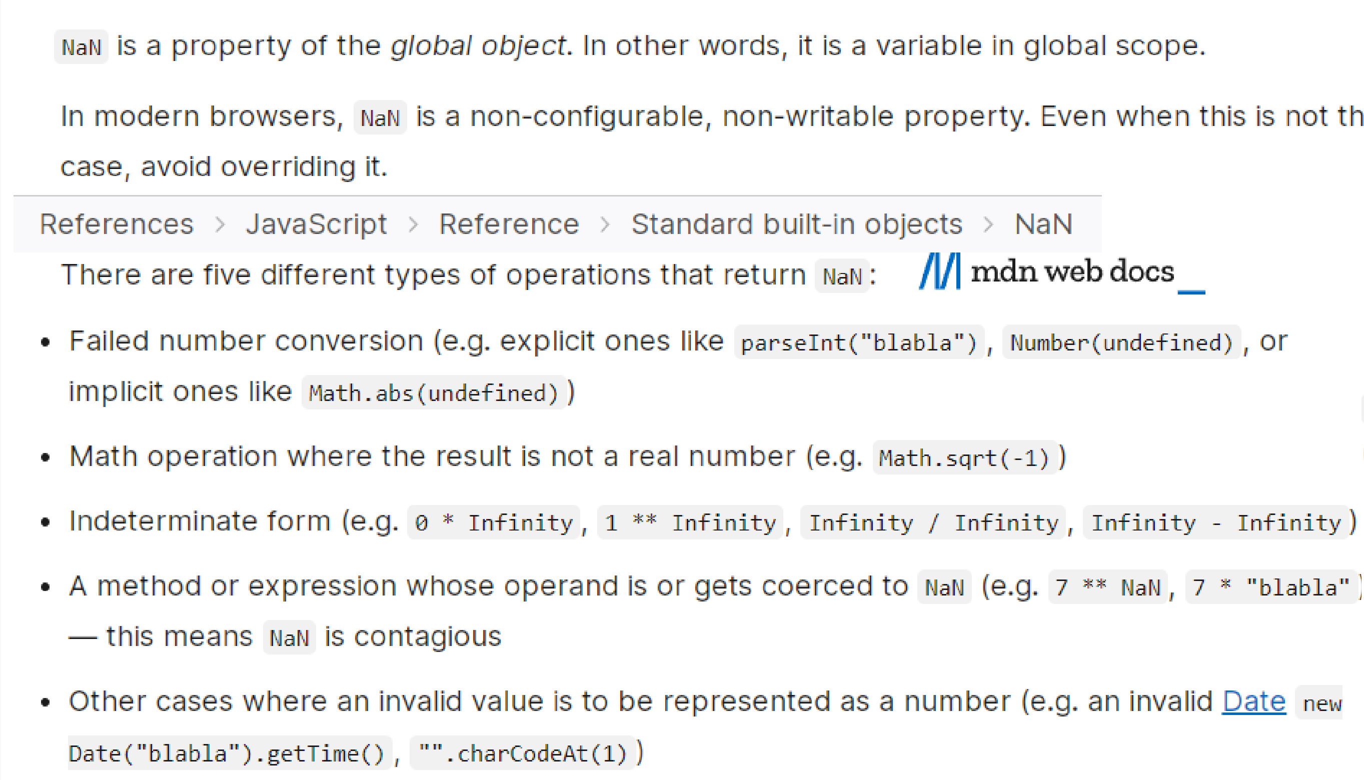
Last Time

Kinds of Data Definitions

- Basic data
  - E.g., **numbers, strings,** etc
- Intervals
  - Data that is from a **range of values,** e.g., [0, 100)
- Enumerations
  - Data that is **one of a list of possible values**, e.g., “green”, “red”, “yellow”
- Itemizations
  - Data value that can be from a **list of possible other data definitions**
  - E.g., **either** a **string** or **number** (Generalizes enumerations)

Itemization Caveats

```
;; A MaybeInt is one of:
(define NaN "Not a Number")
;; or, Integer
;; Interp: represents a number with a possible error case
```



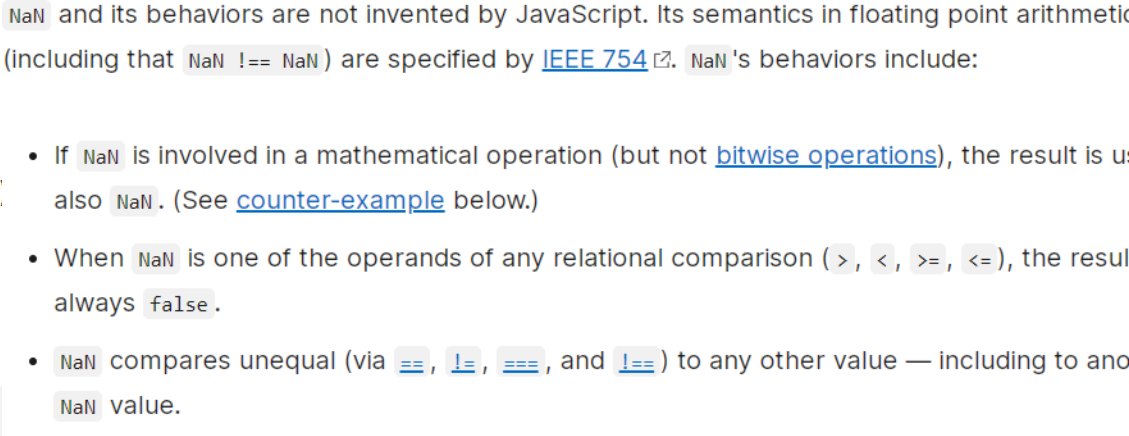
NaN is a property of the *global object*. In other words, it is a variable in global scope.

In modern browsers, NaN is a non-configurable, non-writable property. Even when this is not the case, avoid overriding it.

References › JavaScript › Reference › Standard built-in objects › NaN

There are five different types of operations that return NaN:

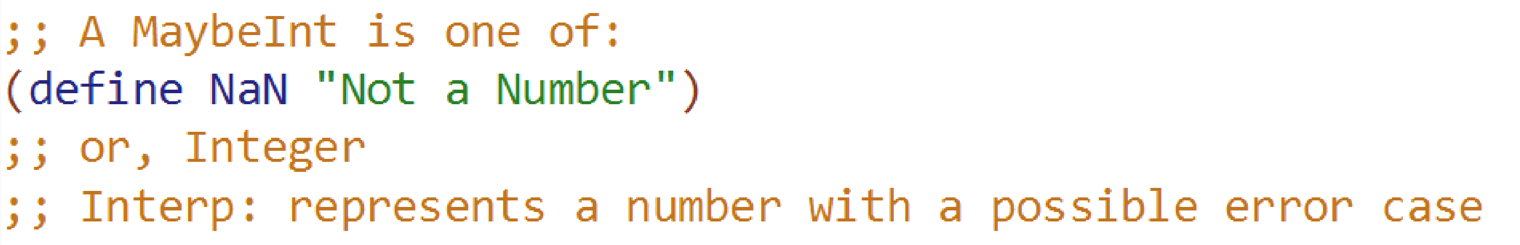
- Failed number conversion (e.g. explicit ones like `parseInt("blabla")`, `Number(undefined)`, or implicit ones like `Math.abs(undefined)`)
- Math operation where the result is not a real number (e.g. `Math.sqrt(-1)`)
- Indeterminate form (e.g. `0 * Infinity`, `1 ** Infinity`, `Infinity / Infinity`, `Infinity - Infinity`)
- A method or expression whose operand is or gets coerced to NaN (e.g. `7 ** NaN`, `7 * "blabla"`) — this means NaN is contagious
- Other cases where an invalid value is to be represented as a number (e.g. an invalid Date `new Date("blabla").getTime()`, `"".charCodeAt(1)`)



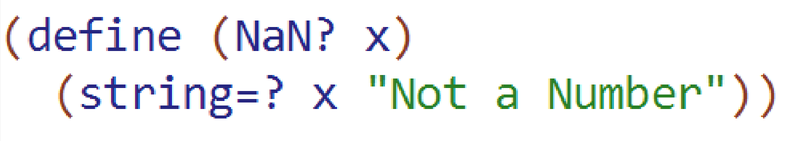
NaN and its behaviors are not invented by JavaScript. Its semantics in floating point arithmetic (including that `NaN !== NaN`) are specified by IEEE 754. NaN's behaviors include:

- If NaN is involved in a mathematical operation (but not bitwise operations), the result is u also NaN. (See counter-example below.)
- When NaN is one of the operands of any relational comparison (`>`, `<`, `>=`, `<=`), the resul always `false`.
- NaN compares unequal (via `==`, `!=`, `===`, and `!==`) to any other value — including to ano NaN value.

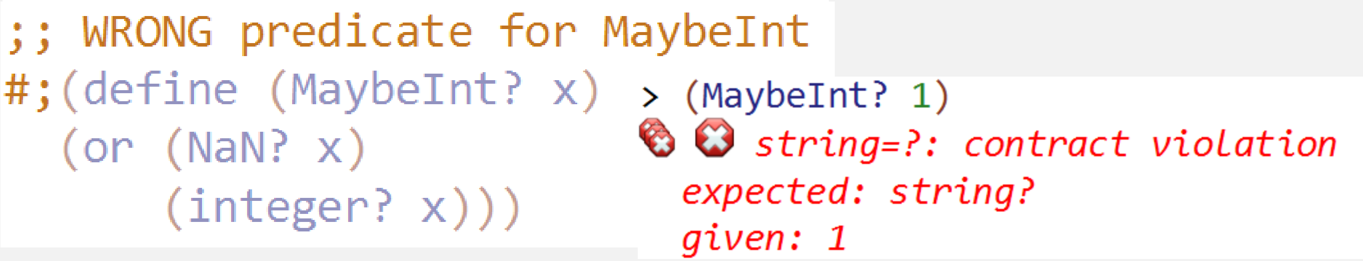
Itemization Caveats



```
;; A MaybeInt is one of:
(define NaN "Not a Number")
;; or, Integer
;; Interp: represents a number with a possible error case
```

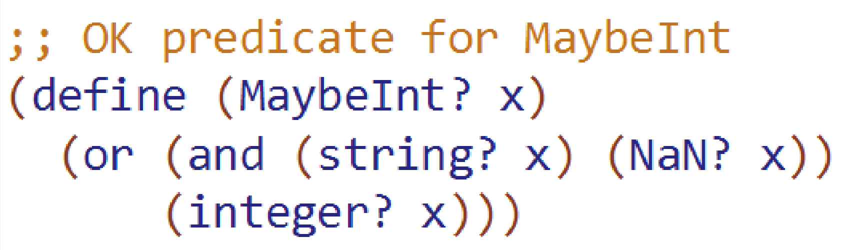


```
(define (NaN? x)
  (string=? x "Not a Number"))
```

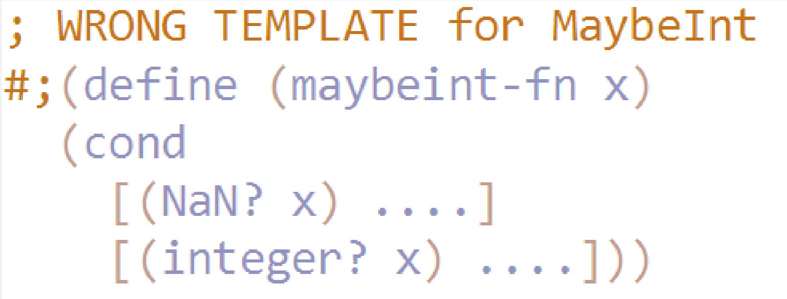


```
;; WRONG predicate for MaybeInt
#;(define (MaybeInt? x)
    (or (NaN? x)
        (integer? x)))
```

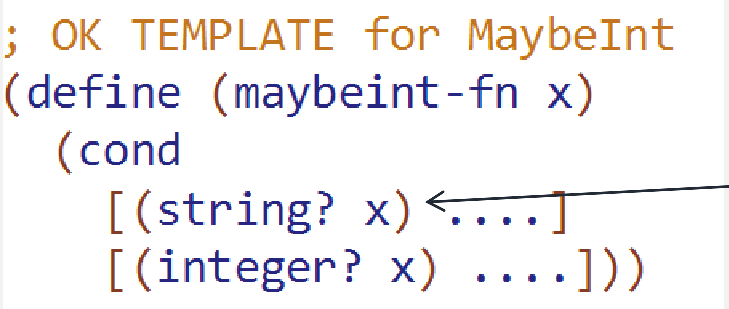
```
> (MaybeInt? 1)
string=?: contract violation
expected: string?
given: 1
```



```
;; OK predicate for MaybeInt
(define (MaybeInt? x)
  (or (and (string? x) (NaN? x))
      (integer? x)))
```



```
; WRONG TEMPLATE for MaybeInt
#;(define (maybeint-fn x)
    (cond
      [(NaN? x) ....]
      [(integer? x) ....]))
```



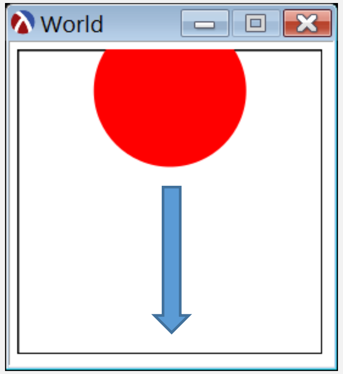
```
; OK TEMPLATE for MaybeInt
(define (maybeint-fn x)
  (cond
    [(string? x) ....]
    [(integer? x) ....]))
```

Inside the function, **we only need to <u>distinguish between valid input cases</u>**

Last Time

Falling Ball Example

```
;; A WorldState is a Non-negative Integer
;; Interp: Represents the y Coordinate of the center of a
;;         ball in a `big-bang` animation.
```



What if the **ball can also move side-to-side?**

`WorldState` would need <u>two</u> **pieces of data:**
the `x` *and* `y` **coordinates**

```
;; A WorldState is an Integer ...
;; ... and another Integer???
```

We need a way to create **compound data**
i.e., **a new data definition that <u>combines</u> values from other data defs**

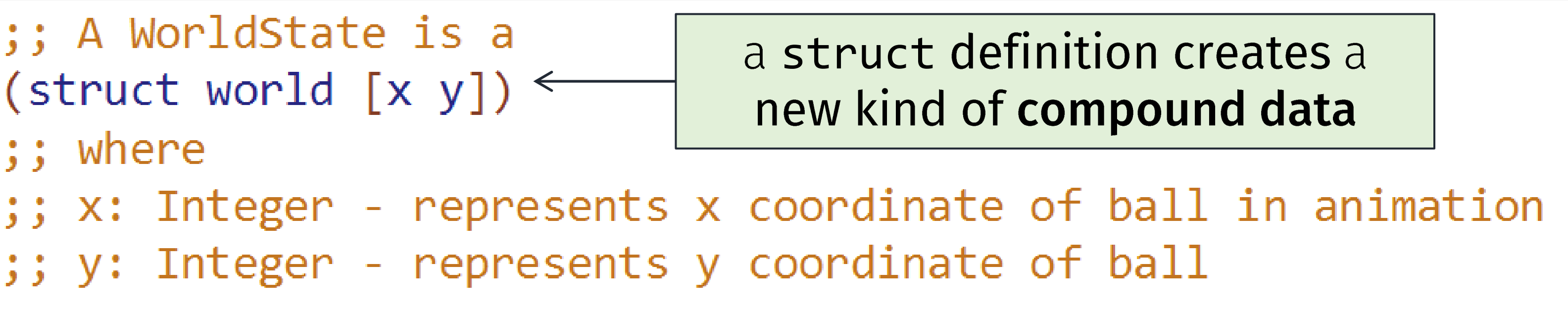
Last Time

Kinds of Data Definitions

- Basic data
  - E.g., **numbers, strings,** etc
- Intervals
  - Data that is from a **range of values,** e.g., [0, 100)
- Enumerations
  - Data that is **one of a list of possible values**, e.g., "green", "red", "yellow"
- Itemizations
  - Data value that can be from a **list of possible other data definitions**
  - E.g., either a **string** or **number** (Generalizes enumerations)
- ➡ **Compound Data**
  - Data that is a **combination of values from other data definitions**

today

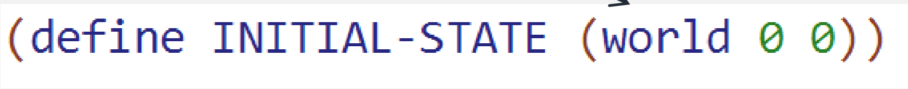
Falling Ball Example



```
;; A WorldState is a
(struct world [x y])
;; where
;; x: Integer - represents x coordinate of ball in animation
;; y: Integer - represents y coordinate of ball
```

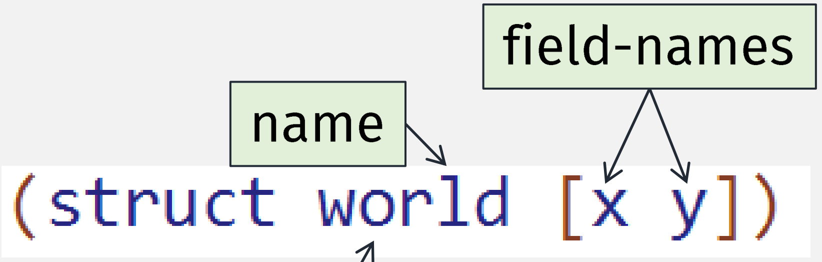
a `struct` definition creates a new kind of **compound data**

Instances of the `struct` are values of that kind of data



```
(define INITIAL-STATE (world 0 0))
```

Parts of a `struct` definition



```
(struct world [x y])
```

Same as name

(Implicitly) defines:

- A **constructor** function → `world`
  - Creates instances of the struct

field-names

name

- **Accessor** functions → `world-x`, `world-y`
  - Get an instance’s field value

name?

- A **predicate** → `world?`
  - Returns true for struct instances

Last Time

Function Design Recipe

1. **Name**

2. **Signature** – <u>types</u> of the function input(s) and output

3. **Description** – <u>explain</u> (in English prose) the function behavior

4. **Examples** – <u>show</u> (using `rackunit`) the function behavior

5. **Code** – <u>implement</u> the rest of the function (arithmetic)

6. **Tests** – <u>check</u> (using `rackunit`) the function behavior

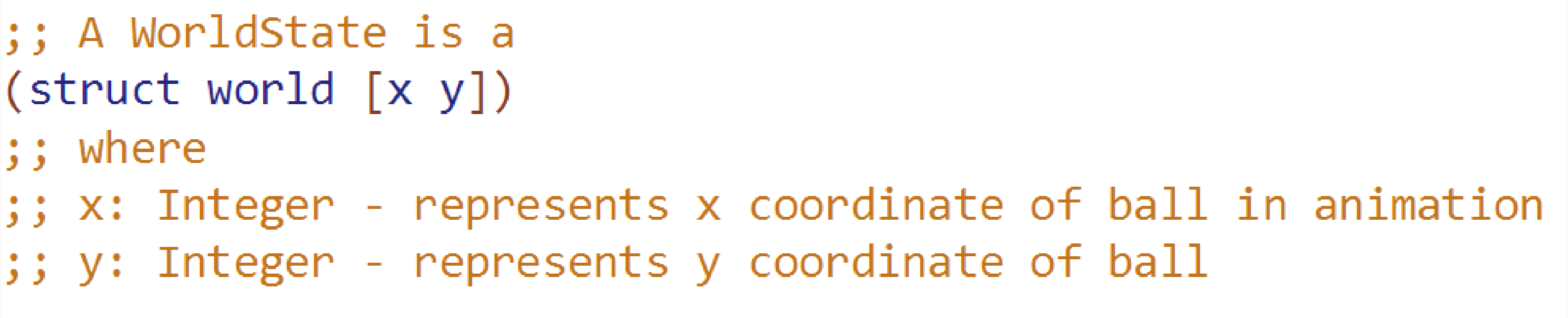
Last Time

Function Design Recipe

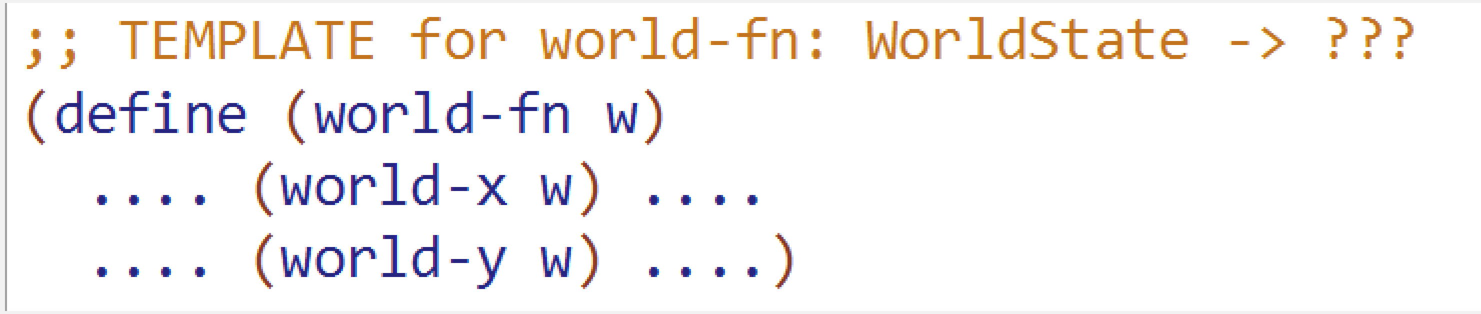
1. **Name**
2. **Signature** – <u>types</u> of the **function input(s)** and **output**
3. **Description** – <u>explain</u> (in **English prose**) the **function behavior**
4. **Examples** – <u>show</u> (using `rackunit`) the **function behavior**
5. **Template** – <u>sketch out</u> the **function structure**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

Template for Compound data

- A **function that consumes compound data** must
  - **extract the individual pieces,** using accessors
  - **combine them,** with arithmetic



```
;; A WorldState is a
(struct world [x y])
;; where
;; x: Integer - represents x coordinate of ball in animation
;; y: Integer - represents y coordinate of ball
```



```
;; TEMPLATE for world-fn: WorldState -> ???
(define (world-fn w)
  .... (world-x w) ....
  .... (world-y w) ....)
```

Code demo

- Moving ball
  - Both x and y coordinate can change
  - With mouse movement
  - (and keyboard directions?)

Check-In Quiz 9/25

on gradescope

(due 1 minute before midnight)