

---

# CSE 331

## Software Design & Implementation

Summer 2024  
Section Quilt – Functional Programming I

# Administrivia

---


- HW Quilt released Thursday evening, due Wednesday (7/10) at **11pm**
  - Please! Start early! HW Quilt and beyond is MUCH more time consuming and challenging than previous homeworks
  - Give yourself time to come to OH and ask questions on Ed
  - Working on the same issue for hours when you're stuck won't help, ask for help!
  - Please look at output of autograder. It will give helpful feedback

# Review – Inductive Data Types

---

- Describe a set by ways of creating an element of the type
  - Each is a “constructor”
  - Second constructor is recursive
  - Can have any number of parameters

Ex: base case                      recursive case

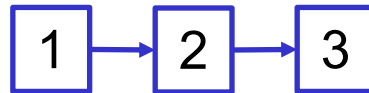
  
type List := nil | cons(x :  $\mathbb{Z}$ , L : List)

nil

cons(3, nil)

cons(2, cons(3, nil))

cons(1, cons(2, cons(3, nil)))



# Review – Structural Recursion


---

- **Inductive types:** builds new values from existing ones
- **Structural recursion:** recurse on smaller parts
  - Call on `n` recurses on `n.val`
  - Guarantees no infinite loops
  - Note: only kind of recursion used for this class

**Ex:** `type List := nil | cons(hd:  $\mathbb{Z}$ , tl: List)`

`func len(nil) := 0`

`len(cons(x, L)) := 1 + len(L)` for any  $x \in \mathbb{Z}$   
and any  $L \in \text{List}$

- Any List is either nil or of the form `cons(x, L)` for some number `x` and List `L`
- Cases of function are exclusive and exhaustive based on 

# Testing

---

```
describe('example', function() {
  it('testBar' function() {
    /* assert statements */
  })
})
```

- Use assertions to compare expected and actual output for each test case
  - `assert.deepStrictEqual(expected, actual);` should be used generally
- Keep your tests simple! Don't want to have to write tests for your tests
- **Note:** Please do not submit commented out test cases to gradescope. The course staff will not count those as valid test cases. It is better to submit failing test cases than commented out test cases.

# Testing – Strict vs Deep

---

Assertion	Failure Condition
<code>assert.strictEqual(expected, actual)</code>	<code>expected !== actual</code>
<code>assert.deepStrictEqual(expected, actual)</code>	values/types of child objects are not equal

```
const v1: Vector = {x: 1, y: 1};  
const v2: Vector = {x: 1, y: 1};
```

← two different objects,  
but same record values

```
it('assert_strict', function() {  
  | assert.strictEqual(v1, v2); ← this will fail  
});
```

```
it('assert_deep_strict', function() {  
  | assert.deepStrictEqual(v1, v2); ← this will pass  
});
```

# Testing – Documenting

---

- Document which subdomain you are testing. A justification: heuristic used, part of code it tests.

**Ex:**

```
describe('example', function() {
```

Name of class being tested



```
  it('testBar' function() {
```

Name of test (can be function being tested)



```
    /* comment describing subdomain being tested */
```

```
    assert...
```

```
  })
```

```
})
```

# Definitions

---

```
type List := nil | cons(hd :  $\mathbb{Z}$ , tl : List)
```

- Len – returns the length of a list:

```
func len(nil)           := 0  
      len(cons(a, L))    := 1 + len(L)   for any  $a : \mathbb{Z}$  and  $L : \text{List}$ 
```

- Sum – returns the sum of the integers in the list:

```
func sum(nil)           := 0  
      sum(cons(a, L))    := a + sum(L)   for any  $a : \mathbb{Z}$  and  $L : \text{List}$ 
```



# Question 1

---

twice takes a list and returns a list of the same length but with every number in the list multiplied by 2

Show the result of applying twice to each list:

nil

\_\_\_\_\_

cons(*a*, nil)

\_\_\_\_\_

cons(*a*, cons(*b*, nil))

\_\_\_\_\_

cons(*a*, cons(*b*, cons(*c*, nil)))

\_\_\_\_\_

...

# Question 1

---

- (b) The previous list of examples is not a formal definition. It does not tell us, for example, what twice does to a list of length 4. More generally, any time we see "...", the definition is probably not formal.

Write a formal definition of twice using recursion.

# Question 1

---

```
func twice(nil)           := nil
      twice(cons(a, L)) := cons(2a, twice(L))   for any a :  $\mathbb{Z}$  and L : List
```

- (c) If we translated this into TypeScript code in the most direct manner (level 0), what heuristic should we use to get a set of subdomains? What specific tests should we use to make sure that everything is correct?

## Question 3

---

twice-evens takes a list and returns a list of the same length but with *every other* number (at even indices) in the list multiplied by 2

Show the result of applying twice-evens to each list:

nil

\_\_\_\_\_

cons(*a*, nil)

\_\_\_\_\_

cons(*a*, cons(*b*, nil))

\_\_\_\_\_

cons(*a*, cons(*b*, cons(*c*, nil)))

\_\_\_\_\_

...

# Question 3

---

(b) The previous list of examples is not a formal definition (because of the "...").

Write a formal definition of this function, twice-evens, using recursion.

# Question 3

---

- (c) If we translated this into TypeScript code in the most direct manner what heuristic should we use to get a set of subdomains? What specific tests should we use to make sure that everything is correct?
-

## Question 2

---

```
if (equal(L, cons(1, cons(2, nil)))) {  
  const R = cons(2, cons(4, nil)); // = twice(L)  
  return cons(0, R);              // = twice(cons(0, L))  
}
```

Comments // are the spec, but the code isn't a direct translation of the spec (level 1)

Need to prove it does the same thing as the spec

## Question 2

---

- (a) Using the fact that  $L = \text{cons}(1, \text{cons}(2, \text{nil}))$ , prove by calculation that  $\text{twice}(L) = R$ , where  $R$  is the constant list defined in the code. I.e., prove that

$$\text{twice}(L) = \text{cons}(2, \text{cons}(4, \text{nil}))$$

---

```
func twice(nil)           := nil
      twice(cons(a, L))    := cons(2a, twice(L))   for any  $a : \mathbb{Z}$  and  $L : \text{List}$ 
```



# Question 2

---

```
func twice(nil)           := nil
      twice(cons(a, L)) := cons(2a, twice(L))   for any  $a : \mathbb{Z}$  and  $L : \text{List}$ 
```

(a) twice(L) = cons(2, cons(4, nil))

(b) Using the facts that  $L = \text{cons}(1, \text{cons}(2, \text{nil}))$  and  $R = \text{cons}(2, \text{cons}(4, \text{nil}))$ , prove by calculation that the code above returns the correct value, i.e., prove that

$$\text{twice}(\text{cons}(0, L)) = \text{cons}(0, R)$$

Feel free to cite part (a) in your calculation.

# Question 4

---

```
func twice-evens(nil)           := nil
twice-evens(cons(a, nil))       := cons(2a, nil)           for any  $a : \mathbb{Z}$ 
twice-evens(cons(a, cons(b, L))) := cons(2a, cons(b, twice-evens(L))) for any  $a, b : \mathbb{Z}$  and  $L : \text{List}$ 
```

- (a) Let  $a$  and  $b$  be any integers. Prove by calculation that
- $$\text{len}(\text{twice-evens}(\text{cons}(a, \text{cons}(b, L)))) = 2 + \text{len}(\text{twice-evens}(L))$$

# Question 4

---

Given this code:

```
return 2 + len(twice_evens(L)); // = len(twice-evens(cons(3, cons(4, L))))
```

And the fact we proved in (a):

$$\text{len}(\text{twice-evens}(\text{cons}(a, \text{cons}(b, L)))) = 2 + \text{len}(\text{twice-evens}(L))$$

(b) Explain why the direct proof from part (a) shows that the code is correct according to the specification (written in the comment).

# HW Quilt Reminders/Recommendations:

---

- **No mutation!** Make a new variable for new values you calculate
- Proofs by calculation require explanations/rules for **every** line (except basic algebra, you can say “math” if you want)
- Proofs by calculation can start with the left or right side of the = to prove
- We won't penalize you for more test cases than the minimum required!
- If you get errors that “property \_\_\_\_ does not exist on type \_\_\_\_” it probably means you are missing a `nil` check

# Proof by calculation LaTeX

---

- *Optional*, if you're using LaTeX feel free to use this to align proofs:

```
$$\begin{aligned}
& \textit{first line of proof} && \\\
& \quad \&\! \! \! \quad = \textit{line of proof} && \text{\textit{rule}} \\\
& \quad \&\! \! \! \quad = \textit{line of proof} && \text{\textit{rule}} \\\
\end{aligned}$$
```

- and to align functions:

```
$$\begin{aligned}
\textbf{func} & \quad \&\! \! \! \quad \text{\textit{funcName}}(\textit{case}) \\
& \quad \&\! \! \! \quad := \textit{result} \quad \&\! \! \! \quad \text{\textit{side cond}} \\\
& \quad \&\! \! \! \quad \text{\textit{funcName}}(\textit{case}) \\
& \quad \&\! \! \! \quad := \textit{result} \quad \&\! \! \! \quad \text{\textit{side cond}} \\\
\end{aligned}$$
```