Practice Exam

CSE 331 2025 Summer

See the last page for some math definitions you may need.

Question 1

Which line of the following function would ***not*** type check in TypeScript? Fill in the blank with the line number or “n/a” if all lines would type check.

```
1    const f = (a: bigint, b: boolean): string | bigint => {
2         if (b) {
3             return a ** a;
4         } else {
5             return a ** 2n;
6         }
7    }
8    const g: bigint = f(2n, true);
```

Answer	

Question 2

Determine the most appropriate type of request method for the described route functionality. Answer “G” for GET and “P” for POST.

Route description	Answer G/P
A route that resets a user’s password.	
A route to retrieve a list of all buildings on the UW campus.	
A route that accepts a longitude and latitude in the query parameters and sends back a list of the 5 closest pizzerias to that location.	
A route that sets a user’s status to “Going” for an event on Comfier (the hot new networking app) and sends back a list of all guests who have RSVP’d.	

Question 3

Suppose a developer is debugging their client-server application, but they forgot to start their server. If they check the Network tab in the Chrome Developer Console, which status code (a / b / c / d / e) would they expect to see for their attempted requests?

a. 200 b. 400 c. 404 d. 500 e. 504

Answer	

Question 4

Consider these functions that exist within a React component. Note: these functions are intentionally out of the usual order and do not follow 331 naming conventions. Assume the route "/api/add" is a defined POST request which accepts the record `{key: string, value: string}` in the body of the request, and sends back a record of type `{saved: boolean}`.

```
A = (data: unknown): void => {
    if (!isRecord(data) || data.saved === undefined) {
        throw new Error();
    }
    __Blank 1__
}

B = (key: string, value: string): void => {
    fetch("/api/add", {
        method: "POST",
        body: JSON.stringify({key: key, value: value});
        headers: {"Content-Type": "application/json"}
    })
        .then(__Blank 2__)
        .catch((e) => console.error(e));
}

C = (res: Response): void => {
    if (res.status === __Blank 3__)
        res.__Blank 4__
           .then(__Blank 5__)
           .catch((e) => console.error(e));
    else
        res.__Blank 6__
           .then((e) => console.error(`bad status ${res.status}: ${e}`);
           .catch((e) => console.error(e));
}
```

Fill in the answer box for each blank with the correct option to complete the sequence of operations related to making a fetch request. You will not need every option.

Blank	Answer (one of a - h)
Blank 1	
Blank 2	
Blank 3	
Blank 4	
Blank 5	
Blank 6	

a. A
b. B
c. C
d. alert(“success!”)
e. 200
f. 400
g. text()
h. json()
i. console.error(...)
j. JSON.stringify()

Question 5

Below are methods of a React component with a single text box where users can enter their name. In React, when we give users an input area, we need to make sure there is a corresponding state that mirrors what they see. Answer below with the line of code needed to fill in the blank and successfully complete state mirroring.

```
constructor (props) {
    super(props);
    this.state = {text: “”}
}

render = () => {
    return <input type=”text” __Blank 1__
                  onChange={ __Blank 2__ }></input>;
}

onChange = (evt: ChangeEvent<HTMLInputElement>) => {
    __Blank 3__
}
```

Blank 1	
Blank 2	
Blank 3	

Question 6

Consider this math definition:

$$
\begin{array}{lll}
 & f: \mathbb{N}, \mathbb{S}^* \to \mathbb{S}^* \cup \mathbb{Z} & \\
f(0, s) & := & s \\
f(1, s) & := & 1 \\
f(n+2, s) & := & n + 2 + f(n+1, s)
\end{array}
$$

In the Answer box below, select the TypeScript function (a / b / c / d) that is a straight-from-the-spec implementation of this math definition.

a.

```
const f = (x: bigint, s: string) : string | bigint => {
    if (x === 0n) {
        return s;
    } else if (x === 1n) {
        return x;
    } else {
         return x + 2n + f(x + 1n, s);
    }
}
```

b.

```
const f = (x: bigint, s: string) : string | bigint => {
    if (x === 0n) {
        return s;
    } else if (x === 1n) {
        return x;
    } else {
        return x + f(x - 1n, s);
    }
}
```

c.

```
const f = (x: bigint, s: string) : string | bigint => {
    if (x === 0n) {
        return s;
    } else {
        return x * (x + 1n) / 2n;
    }
}
```

d.

```
const f = (x: bigint, s: string) : [string, bigint] => {
    if (x === 0n) {
        return s;
    } else if (x === 1n) {
        return x;
    } else {
        return x + 1n + f(x - 1n, s);
    }
}
```

Answer	

Question 7

The following proof attempts to prove the claim $\text{at}(\text{get-positives}(\text{a} :: \text{L}), 0) \geq \text{at}(\text{a} :: \text{L}, 0)$. See the last page for these math definitions.

1	$\text{at}(\text{a} :: \text{L}, 0) = \text{a}$	def of at
2	$= \text{at}(\text{a} :: \text{get-positives}(\text{L}), 0)$	def of at
3	$= \text{at}(\text{get-positives}(\text{a} :: \text{L}), 0)$	def of get-positives
4	$\leq \text{at}(\text{get-positives}(\text{a} :: \text{L}), 0)$	implication

What is the first line of this proof that has a mistake? If every line is correct, answer n/a.

Answer (line number)	

Which option best describes the mistake that was made on that line?

a. Incorrect citation
b. Incorrect application of definition/rule
c. Combining steps
d. Assuming the conclusion
e. Notation error
f. None of the above

Answer	

Question 8

Consider the following TypeScript function:

```
const g = (x: number, y: number): number => {
    if (x === y)
        throw new Error("x and y must be unique");
    if (x > y)
        return (x - y) * (x + y);
    else
        return (y - x) * (x + y);
}
```

Does the function g satisfy each of the following specifications? Answer Yes (Y) or No (N).

Specification:	**Answer Y/N**

`/**`
` * @param x, must be >= 0`
` * @param y, must be >= 0`
` * @returns z >= 0`
` */`	

`/**`
` * @param x, must be >= 0`
` * @param y, must be >= 0`
` * @throws error if x = y`
` * @returns z > 0`
` */`	

`/**`
` * @param x some number`
` * @param y some number`
` * @requires x and y are not equal`
` * @returns (a - b) * (a + b)`
` *  where a = max(x, y) and b = min(x, y)`
` */`	

Question 9

Consider each of the following functions and answer the corresponding question about testing that function according to the *required 331 testing heuristics*:

```
/** @requires x > 0 */
const zip = (x: bigint): boolean => {
    if (x === 0n)
        return false;
    else if (x === 1n)
        return true;
    else
        return !zip(x - 1n);
}

const zap = (letter: 'a' | 'b' | 'c', count: 1n | 2n): string => {
    if (count === 1n)
        return letter;
    else
        return letter + letter;
}

const zop = (letter: 'a' | 'b' | 'c', x: bigint): string => {
    return zap(letter, (x % 2n) + 1);
}
```

Which of the following are a sufficient minimum set of inputs to test `zip`?

a. -1, 0, 1, 2
b. 0, 1, 2, 3
c. 1, 2, 3
d. 0, 1, 2
e. 1, 2

Answer	

What is the minimum number of tests necessary to test `zap`?

Answer (number of tests)	

What is the minimum number of tests necessary to test `zop`?

Answer (number of tests)	

Question 10

Below is the TypeScript implementation of the function get-positives with incomplete assertions included on some lines. If labeled with "P#" the assertion was produced with forward reasoning and if labeled with "Q#" the assertion was produced with backward reasoning. "Pre" and "Post" are facts we determine given the function spec.

```
/** @returns get-positives(L) */
const getPositives = (L: list<bigint>): List<bigint> => {
    {{ Pre: _____ }}
    let R = nil;
    {{ Inv: get-positives(L0) = R ⧺ get-positives(L) }}
    while (L.kind !== nil) {
        if (L.hd > 0n) {
            R = concat(R, cons(L.hd, nil));
            {{ P1: _____ }}
        }
        {{ Q1: _____ }}
        L = L.tl;
    }
    {{ P2: _____ }}
    {{ Post: _____ }}
    return R;
}
```

For each of the completed assertions below, answer True (T) or False (F) if they are the correct product of forward or backward reasoning to that point in the code.

Assertion:	**Answer T/F**
{{ Pre: }}	
{{ P1: get-positives(L_0) = R ⧺ (hd :: nil) ⧺ get-positives(L) and L = hd :: tl and L.hd > 0n }}	
{{ Q1: get-positives(L_0.tl) = L.hd :: R ⧺ get-positives(L.tl) }}	
{{ P2: get-positives(L_0) = R ⧺ get-positives(L) and L = nil }}	
{{ Post: R = get-positives(L_0) }}	

Question 11

Below is a TypeScript function that takes an array of integers and mutates the array so each element is the absolute value of the original value at that index.

```
const arrAbs = (A: Array<bigint>): void => {
    let i = -1;
    // Inv: ____________________________
    while (i != A.length - 1) {
        i = i + 1;
        if (A[i] < 0n) {
            A[i] = -A[i];
        }
    }
}
```

Which of the following invariants does this loop maintain?

a. $A[k] = |A[k]|$ for any $0 \leq k < A.length$
b. $A[k] = |A[k]|$ for any $0 \leq k < i$
c. $A[k] = |A[k]|$ for any $0 < k < A.length$
d. $A[k] = |A[k]|$ for any $0 \leq k \leq i$

Answer	

Question 12

The following function takes two arrays of numbers and creates a new array where each index is the product of the values at the same index from the input arrays. The loop already has an invariant, but there are some blanks that need to be filled in to complete the function.

```
/**
 * @requires len(A) = len(B)
 * @returns R such that R[j] = A[j] * B[j] for 0 ≤ j < n
 */
const prod = (A: Array<number>, B: Array<number>): Array<number> => {
    let i = __Blank 1__;
    let R = [];
    // {{ Inv: R[j] = A[j] * B[j] for 0 ≤ j ≤ i }}
    while (i != __Blank 2__) {
        R.push(A[__Blank 3__] * B[__Blank 3__]);
        i = __Blank 4__;
    }
    return R;
```

```
}
```

Select the option for each blank such that the function maintains the loop invariant and guarantees the post condition:

Blank Options	**Answer (of a - e)**
`let i = __Blank 1__;` a. -1 b. 0 c. 1 d. A.length - 1 e. A.length	
`while (i != __Blank 2__)` a. 0 b. A.length - 1 c. A.length d. A.length + B.length - 2 e. A.length + B.length	
`R.push(A[__Blank 3__] * B[__Blank 3__]);` a. i - 1 b. i c. i + 1 d. A.length - i e. A.length - i - 1	
`i = __Blank 4__;` a. j - 1 b. j c. i - 1 d. i e. i + 1	

Question 13

For each of the functions below, indicate whether it is **tail recursive**. Answer “Y” for Yes or “N” for No.

Function:	**Answer Y/N**

`const a = (L: List<bigint>): List<bigint> => {`
`  if (L.kind === “nil”) {`
`  return true;`
`  } else if (L.tl.kind === “nil”) {`
`  return false;`
`  } else {`
`  return a(L.tl.tl);`
`  }`
`}`	

`const b = (x: bigint, S: List<bigint>): => {`
`  if (x === 0n) {`
`  return S;`
`  } else {`
`  return cons(x, b(x - 1n, S));`
`  }`
`}`	

Question 14

In JavaScript, `0 == "0"`, `0 == ""`, and `0 == " "`, all return true, but `"" === " "` returns false. This is evidence of issues with JavaScript’s notion of equality. Specifically, which equality property is broken by the fact that these all result to true besides `"" === " "`?

a. Transitive
b. Reflexive
c. Symmetric

Answer	

Question 15

```
const foo = (y: bigint): bigint => { ... }
const bar = (x: unknown): bigint => {
    if (typeof x === “bigint”) {
        const a = foo(x);
        return a;
    }
    ...
}
```

Which of the following best describes what is happening in the snippet of the function **`bar`**?

a. Type error
b. Defensive programming
c. Type narrowing
d. Representation Exposure

Answer	

Question 16

Refer to the following type and function definitions for the below questions:

```
type XY = {x: number, y: number}
type XYZ = {x: number, y: number, z: number}

const f = (n: XY): XYZ => { … }
const g = (n: XYZ): number => { … }
```

Determine if the TypeScript type checker would allow (A) or disallow (D) each of the following variable assignments.

Variable assignment:	**Answer A/D**
`const a1: XY = f({x: 1.2, y: 2.1});`	
`const a2: XYZ = f({x: 1.2, y: 2.1, z: 0});`	
`const a3: number = g({x: 1.2, y: 2.1});`	
`const a4: unknown = g({x: 1.2, y: 2.1, z: 0});`	

Question 17

This question asks about the `TodoList` ADT which represents an *unordered* list of string todo items. There is a class that implements this interface called "`TwoListTodoList`."

```
interface TodoList {
      /**
       * Adds the given item to the list
       * @param item to add to the list
       * @modifes obj
       * @effects adds item to the top of obj
       */
       addItem: (item: string) => void;

      /**
       * Marks the given item as completed if it exists in the list
       * @param item to mark complete
       * @modifies obj
       * @effects entry in obj with name "item" set to complete
       */
      completeItem: (item: string) => void;

      /**
       * @returns list of items that haven't been marked complete
       */
      getIncomplete: () => List<string>;
}

/** @returns an empty Todo List */
const initTodoList (): TodoList => {
      return new TwoListTodoList();
}
```

Label each of the following methods of `TodoList` with the design pattern from the bank below that best describes it. You may need to use certain patterns more than once or not at all.

mutator producer observer factory function singleton

Function	Design Pattern
addItem	
completeItem	

Function	Design Pattern
getIncomplete	
initTodoList	

Question 18

The `TwoListTodoList` class implements the `TodoList` interface from the last question.

```
1  class TwoListTodoList implements TodoList {
2      // AF: obj = this.incomplete ++ this.complete
3      complete: List<string>;
4      incomplete: List<string>;
5
6      constructor () {
7          this.complete = nil;
8          this.incomplete = nil;
9      }
10
11     addItem = (item: string): void => {
12         this.incomplete = cons(item, this.incomplete);
13     }
14
15     completeItem = (item: string): void => {
16         // iterate over list of all items, cons together result
17         let newIncomplete = nil;
18         while (this.incomplete.kind !== “nil”) {
19             if (this.incomplete.hd !== item) {
20                 newIncomplete = cons(temp.hd, newIncomplete);
21             }
22             this.incomplete = this.incomplete.tl;
23         }
24         this.incomplete = newIncomplete;
25         this.complete = cons(item, this.complete);
26     }
27
28     getIncompleteItems = (): List<string> => {
29         return this.incomplete;
30     }
31 }
```

Determine if each statement is true (T) or false (F) regarding the `TodoList` ADT and `TwoListTodoList` representation.

Statement:	**Answer T/F**
`TodoList` is an immutable ADT.	
`TwoListTodoList` properly implements `TodoList`.	
If `TodoList` specified that it maintains the items in their original ordering, `TwoListTodoList` would properly implement `TodoList`	

Question 19

Does TwoListTodoList have representation exposure? If yes, answer with the line number where the rep is exposed, otherwise, answer n/a.

Answer	

Question 20

The spec for the constructor of the `TwoListTodoList` says:

```
// make obj = nil
```

Prove by calculation that the constructor of `TwoListTodoList` is correct.

List Functions

Below is a list of mathematical definitions and properties you may need to reference during the exam.

concat: $(\text{List}\langle A\rangle, \text{List}\langle A\rangle) \to \text{List}\langle A\rangle$

$\text{concat}(\text{nil}, R) := R$

$\text{concat}(x :: L, R) := x :: \text{concat}(L, R)$

Identity of concat: $L \mathbin{++} \text{nil} = L = \text{nil} \mathbin{++} L$ for any list L.

Associativity of concat: $(L \mathbin{++} R) \mathbin{++} S = L \mathbin{++} (R \mathbin{++} S)$ for any lists L, R, S.

at : $(\text{List}\langle A\rangle, \mathbb{N}) \to A$

$\text{at}(\text{nil}, n) := \text{undefined}$

$\text{at}(x :: L, 0) := x$

$\text{at}(x :: L, n+1) := \text{at}(L, n)$

get-positives: $\text{List}\langle A\rangle \to \text{List}\langle A\rangle$

$\text{get-positives}(L) := \text{nil}$ if $L = \text{nil}$

$\text{get-positives}(L) := \text{hd} :: \text{get-positives}(\text{tl})$ if $L \neq \text{nil}$, so $L = \text{hd} :: \text{tl}$, and $\text{hd} > 0$

$\text{get-positives}(L) := \text{get-positives}(\text{tl})$ if $L \neq \text{nil}$, so $L = \text{hd} :: \text{tl}$, and $\text{hd} \leq 0$