Review & Recursion with a Number Parameter

Lecture 7 – COMP110 – Spring 2018

Announcements

- PS1 – On Point Functions – Due Sunday at 11:59pm
  - If you have not completed the Walk section of PS1 yet… cancel your weekend plans (but seriously… knock it out before the Dook game)
- WS2 – Practice with Lists & Recursion – Due *next* Friday 2/16 at 11:59pm
- Tutoring – Tomorrow from 12pm - 4pm

1. Fill in the blank with **c**'s type.
Choose **c**'s initial value.

```
let a: string = "1";
let b: number = 2;
let c: ______ = a + b;
```

2. What value of **x** would cause both "BEAT DOOK" and "BISCUITS" to print out?

```
let x: number = await promptNumber("x");

if (x < 55) {
   print("GO UNC");
} else {
   print("BEAT DOOK");
}

if (x >= 55) {
   print("JUMP AROUND");
} else {
   print("BISCUITS");
}
```

3. What is printed when 8 is entered for **x**? What is printed when 10 is entered for **x**?

```
let x: number = await promptNumber("x");

if (x < 10) {
   x = x * 2;
} else {
   x = x / 2;
}

if (x > 10) {
   x = x + 2;
} else {
   x = x - 2;
}

print(x);
```

4. Fill in the blanks.

```
export let main = async () => {

   let value: ___1___ = f("Hello, world");

};

let f = (p: ___2___): number => {
   // Implementation Omitted
};
```

5. Fill in the blanks.

```
export let main = async () => {

   if (g("GDTBATH", 23)) {
         print("GTHD");
   }

};

let g = (a:  __1__ , b:  __2__ ): __3__ => {
   // Implementation Omitted
};
```

6. What is the printed result?

```
export let main = async () => {
    let numbers: List<number> = listify(10, 3, 4);
    let result: number = foo(numbers);
    print(result);
};

let foo = (list: List<number>): number => {
    if (list === null) {
        // Number.MAX_VALUE is a BIG number.
        // It's 1.7976931348623157e+308
        return Number.MAX_VALUE;
    } else {
       let current: number = first(list);
       let fooRest: number = foo(rest(list));
        if (current < fooRest) {
            return current;
        } else {
            return fooRest;
        }
    }
};
```

7. What is the printed result?

```
export let main = async () => {
    let numbers: List<number> = listify(-2, 0, 3);
    let result: number = bar(numbers);
    print(result);
};

let bar = (list: List<number>): number => {
    if (list === null) {
        return 0;
    } else {
        if (first(list) <= 0) {
            return 1 + bar(rest(list));
        } else {
            return bar(rest(list));
        }
    }
};
```

8. What is the printed result?

```
export let main = async () => {
    let numbers: List<number> = listify(1, 2, 3);
    let result: List<number> = baz(numbers);
    print(result);
};

let baz = (list: List<number>): List<number> => {
    if (list === null) {
        return null;
    } else {
        return cons(1 + first(list), baz(rest(list)));
    }
};
```

Designing a Recursive Function (1/3)

Step 1) What is your *absolute* base case?

- What is the *simplest* case you can imagine?
- What argument(s) can the function be called with where an answer can be determined in a single step?
- Example:
  - Function: `count` processes a List to count the number of values it contains
  - Absolute Base Case: The list is empty. The count function returns **0**.

Designing a Recursive Function (2/3)

Step 2) What is *the next simplest case* of the function that can be expressed in terms of the *absolute base case*?

- This is the recursive case. It's like induction in Math. There are simple patterns we will follow.

- Pattern when writing a recursive List function:
  - The *absolute base case* is when the List is *empty*
  - The *next simplest case* is when the List has *one element* in it

- Example:
  - **Absolute base Case:**
    `count(null) = 0`

  - **Recursive Case:**
    `count("John" → null) = 1 + count(rest("John" → null))`

Designing a Recursive Function (3/3)

Step 3) Are there any *special* base cases to handle?

- Can your function conclude a final answer *before* reaching the absolute base case? If so, add a special base case to handle it.

- Example:
  - Function: **`includes`** processes a List to see if a specific value is included in the List.
  - Special Base Case: The specific value is found. Return true.

Designing a Recursive Function

Step 1) What is your *absolute* base case?

Step 2) What is *the next simplest case* of the function that can be expressed in terms of the *absolute base case*?

Step 3) Are there any *special* base cases to handle?

Recursion with Numbers

- So far our recursive functions have primarily worked with List data
- Today, we'll design a recursive function that works with numbers
- To explore the difference, we're going to pick a *silly* function to write with an *arbitrary* limitation enforced *only* to challenge us.
- **Challenge: write a function that adds any two numbers… whose implementation can only ever add up by 1 and/or subtract by 1?**

add – Step 1) What is our *absolute base case*?

- What is the simplest case we can imagine when *adding* two numbers?
- How about adding any number and zero?

```
add(13, 0)
```

- What is the expected return value in this case? 13!
- If 13 and 0 had parameter names of *n* and *m*, how would we generically specify this?

```
add(n: number, m: number) => n
```

add – Step 2) What is *the next simplest case* of the function that can be expressed in terms of the *absolute base case*?

- What is the *next simplest case* we can imagine when adding two numbers?

- How about adding any number and 1?

```
add(13, 1)
```

- How can we express this in terms of the absolute base case?

```
add(13, 1) => 1 + add(13, 0)
```

- If 13 and 1 had parameter names of *n* and *m*, how would we generically specify this?

```
add(n, m) => 1 + add(n, m - 1)
```

Translating our recursive design to code:

Base case is when m is 0:

```
add(n, m) => n
```

Recursive case is:

```
add(n, m) => 1 + add(n, m - 1)
```

```
let add = (n: number, m: number): number => {




};
```

Tracing another case…

```
let add = (n: number, m: number): number => {
    if (m === 0) {
        return n;
    } else {
        return 1 + add(n, m - 1);
    }
};
```

add(3, 2)

Tracing another case…

```
let add = (n: number, m: number): number => {
    if (m === 0) {
        return n;
    } else {
        return 1 + add(n, m - 1);
    }
};
```

add(3, 2)

1 + add(3, 1)

Tracing another case…

```
let add = (n: number, m: number): number => {
    if (m === 0) {
        return n;
    } else {
        return 1 + add(n, m - 1);
    }
};
```

add(3, 2)

1 + add(3, 1)

1 + add(3, 0)

Tracing another case…

```
let add = (n: number, m: number): number => {
    if (m === 0) {
        return n;
    } else {
        return 1 + add(n, m - 1);
    }
};
```

add(3, 2)

1 + add(3, 1)

1 + add(3, 0)

3

Tracing another case…

```
let add = (n: number, m: number): number => {
    if (m === 0) {
        return n;
    } else {
        return 1 + add(n, m - 1);
    }
};
```

add(3, 2)

1 + add(3, 1)

1 + 3

Tracing another case…

```
let add = (n: number, m: number): number => {
    if (m === 0) {
        return n;
    } else {
        return 1 + add(n, m - 1);
    }
};
```

add(3, 2)

1 + 4

Tracing another case…

```
let add = (n: number, m: number): number => {
    if (m === 0) {
        return n;
    } else {
        return 1 + add(n, m - 1);
    }
};
```

5

add – Step 3) Are there any special base cases to handle?

- In this example, no.
- There could be, though! Perhaps we would want to handle the special case of negative numbers in a special way.
- We will see more examples of where there are special base cases as we progress through the course.

Hands-on: Design a recursive **subtract** function

- Open 01-sub-app.ts

- Step 1) What is the simplest case (base case) you can imagine?
  - Hint: if **m** is 0, what should you return?

- Step 2) What is *next simplest case* you can imagine? How can you express it in terms of the base case from step 1?
  - Consider when **m** is 1. Try writing out **sub(3, 1)** in terms of **sub(3, 0)**.
  - How can you generalize 3, 1 to be m, n?

- Try changing the arguments you pass to sub. Can you think of any that will cause an infinite recursion error? Check-in on PollEv.com/compunc

```
let sub = (n: number, m: number): number => {
   if (m === 0) {
      return n;
   } else {
      return sub(n, m - 1) - 1;
   }
};
```

Rules of Recursive Functions

1. Test for a base case
   - List Parameter: Is the list empty?
   - **Number Parameter: Is the number equal to 0?**

2. Always change at least one argument when recurring
   - List Parameter: Use the rest of the List when recurring.
   - **Number Parameter: Subtract 1 from the number when recurring.**

3. To build a list, process the first value, and then cons it onto the result of repeating the same process recursively on the rest of the list