Generic Types and Building Lists Recursively

Lecture 14

What are the differences between these two classes?

```
class NodeNumber {
   data: number = 0;
   next: NodeNumber = null;
}

class NodeString {
   data: string = "";
   next: NodeString = null;
}
```

Generic Types

- When working with **collections** of data, like lists and arrays, you often want capabilities (functions/algorithms) that work regardless of the collection's data type
  - i.e. does a list of strings include a specific string? does a list of numbers include a specific number?

- Creating a data structure class and related functions per data type leads to a *lot* of repetition in code.

- **Generic types** offer a solution and enable you to parameterize your data types at a class or function level.

Generic Classes

- When 2 classes differ only by the types of their properties you should use a generic class instead.

```
class NodeNumber {
   data: number = 0;
   next: NodeNumber = null;
}
```

```
class NodeString {
   data: string = "";
   next: NodeString = null;
}
```

Generic Classes

- **Step 1:** Designate a class as being generically typed and update any recursively typed properties

- Place a "diamond" **<>**, with a name placeholder in it like **<T>**, after the class name:

```
class Node<T> {
   data: number = 0;
   next: Node<T> = null;
}
```

- You can read this as "Class Node is generic for any type T"
- The use of the capital letter **T**(ype), is only a convention. We could place another letter, like **U**, or even a word here, like **TYPE .**

Generic Classes have Generic Properties

- **Step 2:** Change the relevant properties to be generically typed.

```
class Node<T> {
   data: T;
   next: Node<T> = null;
}
```

- Once a generic class is defined, you can use concrete types such as:
  - `Node<number>` - where T is number, thus the `data` property's type is number
  - `Node<string>` - where T is string, thus the `data` property's type is string
- **Big idea**: using only one generic class definition for Node, you can work with Node objects that hold data of any type!
- Note: you cannot assign a default value to a generic property. Think about why not.

Follow-Along: Define a generic Node class.

- In 00-generics-app.ts, let's define the following class together:

```
export class Node<T> {
  data: T;
  next: Node<T> = null;
}
```

Generic Classes - Constructing Objects

- **Step 3:** Constructing objects of generic types.

```
// Explicit Typing
let a: Node<string> = new Node<string>();
a.data = "hello, world";

// Type Inference
let b = new Node<number>();
b.data = 110;
```

You can use the concrete types anywhere you could otherwise use a class name. For example, declaring variables and constructing objects.

Note, however, the concrete types Node<string> and Node<number> are not the same type! For example, trying to assign `a.next = b;` in the code above will error.

What's different about these two functions?

```
export let consString = (data: string, next: Node<string>): Node<string> => {
    let n = new Node<string>();
    n.data = data;
    n.next = next;
    return n;
};

export let consNumber = (data: number, next: Node<number>): Node<number> => {
    let n = new Node<number>();
    n.data = data;
    n.next = next;
    return n;
};
```

Generic Functions (1 / 4)

- Do we *really* need to duplicate the logic of functions like **cons** for every concrete type of **Node**?

- Good news! No, we do not thanks to **generically typed functions**.

- Rule of Thumb: When 2+ functions differ *only* in parameter types or return type, you can replace them with a single generic function.

Generic Functions (2 / 4)

- **Step 1:** Designate a function as a function that is "Generic for any Type"

- Place a "diamond" **<>**, with a name in it like **<T>**, before the parameter list:

```
let cons = <T> (data: string, next: Node<string>): Node<string> => {
    // ...
};
```

- As with classes, the use of the capital letter **T**(ype), is only a convention. Additionally, the choice of using T in the class definition and then again here are independent decisions. It's good style for generic functions to use the same generic type name conventions as the generic classes they operate on, though.

Generic Functions (3 / 4)

- **Step 2:** Identify the types that changed between your otherwise identical functions.

```
let consString = (data: string, next: Node<string>): Node<string> => {
```

```
let consNumber = (data: number, next: Node<number>): Node<number> => {
```

- Replace the types that changed with the generic type T:

```
let cons = <T> (data: T, next: Node<T>): Node<T> => {
```

- Finally, inside the function definition body, replace specific type with generic.

Generic Functions (4 / 4)

- **Step 3:** Inside the function definition body, replace specific types with generic.

```
export let cons = <T> (data: T, next: Node<T>): Node<T> => {
    let n = new Node<string>();
    n.data = data;
    n.next = next;
    return n;
};
```

- The programming language will catch the type error above until corrected.

```
export let cons = <T> (data: T, next: Node<T>): Node<T> => {
    let n = new Node<T>();
    n.data = data;
    n.next = next;
    return n;
};
```

Hands-on: Making a Generic **cons** Function

- Open lec10 / 01-generic-functions-app.ts

1. Convert the **cons** function to be a generic function.
   a) Add the diamond T syntax before the parameter list: **let cons = <T> (...**
   b) Replace the specific **string** type with the generic type **T**
      - **string** is replaced with **T**
      - **Node<string>** is replaced with **Node<T>**

2. In the main function, change the specific calls to consString and consNumber to use your the generic cons function.

3. Remove the consString and consNumber functions from the program.

- Check-in on PollEv.com/compunc when your generic `cons` function is working

```
export let cons = <T> (data: T, next: Node<T>): Node<T> => {
   let n = new Node<T>();
   n.data = data;
   n.next = next;
   return n;
};
```

Building Lists Recursively

What do we *know* about **`Lists`**?

- The end of a List is **`null`**
  - Implies: An empty list is `null`.

- The fundamental functions of working with Lists
  1. The ***`cons`*** function constructs Lists by adding a value to the front of a List
  2. The ***`first`*** function returns the first value of a List
  3. The ***`rest`*** function returns a sub-List *without* the first value

Building Lists Recursively

- Previously, our recursive functions took a list as a parameter and then recursively computed *a single value*

- Can we write a function that takes a List as a parameter and then *builds and returns another List*?

- To build a List, we'll process the first value of the List and *cons* it onto the result of processing the rest of the list recursively
  - So far, our recursive functions have processed the `first` value of a List
  - Now we'll use the *cons* function to add a value on to the front of a new List

Follow-Along: Acronymify

- Goal: Given a list of strings (words), return a List of the first letters of each string

- Pseudo-code:
  - Is the list empty? Return an empty list!
  - Else, return a new list with the first letter of the first word consed onto the result of this same process applied recursively to the rest of the list.

- Let's open 02-acronymify-app.ts

```
export let main = () => {
    let words = cons("University", cons("North", cons("Carolina", null)));
    print(toString(words));

    let acronym = acronymify(words);
    print(toString(acronym));
};

let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

acronymify("Michael" → "Jordan" → null);

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
acronymify( "Michael" →  "Jordan" →  null );
          ⬇
     cons( "M",  acronymify( "Jordan" →  null ) );
```

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.

Tracing **acronymify**

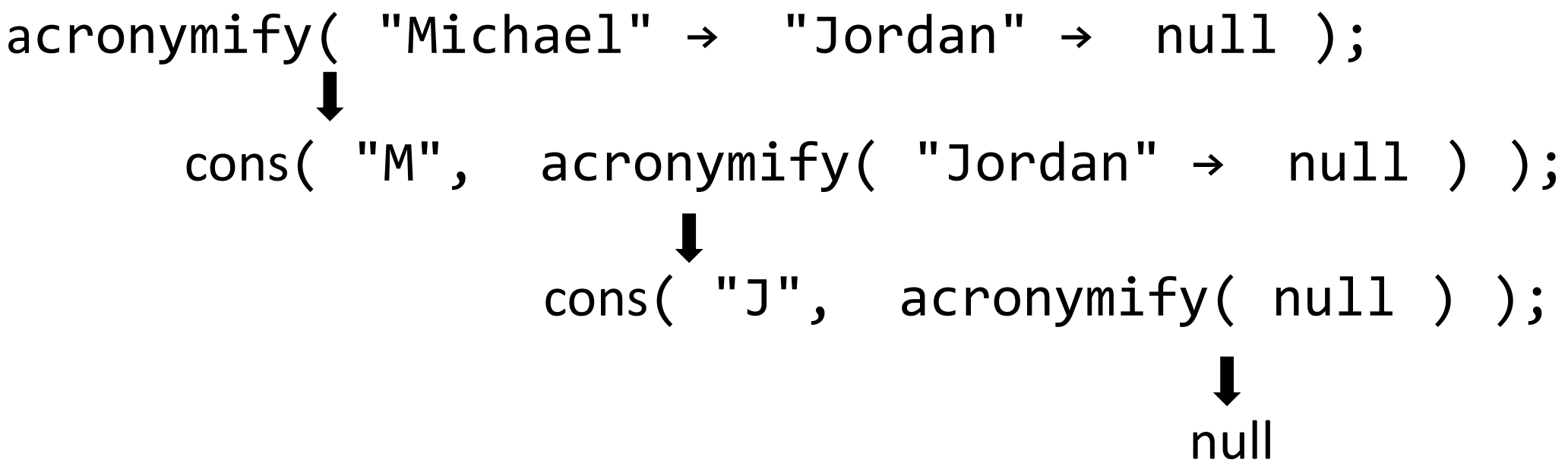
```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
acronymify( "Michael" → "Jordan" → null );
          ⬇
    cons( "M", acronymify( "Jordan" → null ) );
                        ⬇
                 cons( "J", acronymify( null ) );
```

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.

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```



Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
acronymify( "Michael" →  "Jordan" →  null );
          ⬇
     cons( "M",  acronymify( "Jordan" →  null ) );
                       ⬇
                 cons( "J",  null );
                       ⬆
                     null
```

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
acronymify( "Michael" →  "Jordan" →  null );
          ⬇
      cons( "M",  cons( "J", null ) );
                        ⬆
                  cons( "J",  null );
```

Tracing **acronymify**

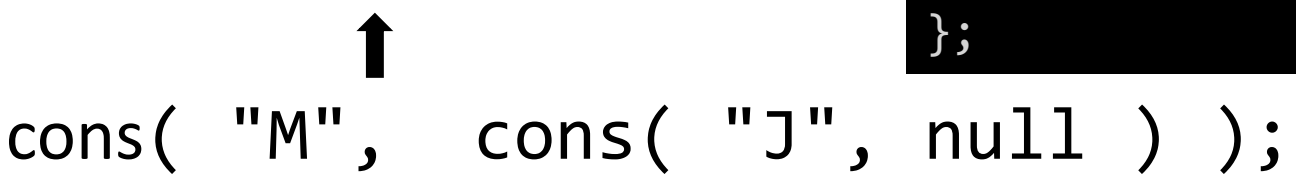
```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
cons( "M",  cons( "J", null ) );
```

```
    cons( "M",  cons( "J", null ) );
```

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```




```
cons( "M",  cons( "J", null ) );
```

Rules of Recursive Functions

1. Test for a base case (empty list)

2. Always change at least one argument when recurring (rest of list)

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**

Giving Instructions to a Bouncer

- **"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."**
  - **This is dense. Let's consider an analogy.**

- Pretend you're the owner of the hottest club in Chapel Hill

- What are the (recursive) instructions would you give to the bouncer?

Bouncer Algorithm

If the entrance line is empty, then your job is done.

Otherwise,

> If the person at the front of the line is 21 or over, then *cons* them to the bar line followed by repeating this same process on the rest of the line.
>
> Else, ignore them and their pleas of desperation. Repeat this process on the rest of the line.

Hands-on - Bouncer

- Open 03-bouncer-app.ts

- Your goal: Add the nested if-then-else case to bounce ages under 21.

1. If age is under 21, return the result of calling the bouncer function recursively on the rest of the list.

2. Otherwise, return the result of consing the current age onto the result of calling the bouncer function recursively on the rest of the list.

- Check-in on PollEv.com/compunc when your agesInBar line only has values >= 21

```
let bouncer = (list: Node<number>): Node<number> => {
    if (list === null) {
        return null;
    } else {
        let age = first(list);
        if (age < 21) {
            return bouncer(rest(list));
        } else {
            return cons(age, bouncer(rest(list)));
        }
    }
};
```

- Notice the difference between these two branches.
- When the first age is under 21, the result is *only* the bouncer function applied to the rest of the line.
- When the first person is 21 or over, the result is *consing* the current person onto the front of a new List, followed by the bouncer function applied to the rest of the line.

Rules of Recursive Functions

1. Test for a base case
   - When recurring on a list: when the list is empty.
   - When recurring on a number: when it is out of some bounds.

2. Always change at least one argument when recurring
   - When recurring on a list: use the rest of the list.
   - When recurring on a number: use arithmetic to bring you closer to base case.

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**