Recursive List Generation

Lecture 17

Follow-along: Built-in Functions

- Open today's 00-builtins-app.ts
- Declare a variable initialized to a List of strings with listify
- Print the variable
- Print its first and rest

Challenge Question 0: Given the following generic function definitions, what are the inferred types of `xs`, `firstXs`, and `restXs`?

```
let cons = <T> (value: T, list?: Node<T>) => Node<T> {…}
let first = <T> (list: Node<T>) => T {…}
let rest = <T> (list: Node<T>) => Node<T> {…}
```

```
export let main = async () => {
   let xs = cons(1, cons(2, null));
   let firstXs = first(xs);
   let restXs = rest(xs);
};
```

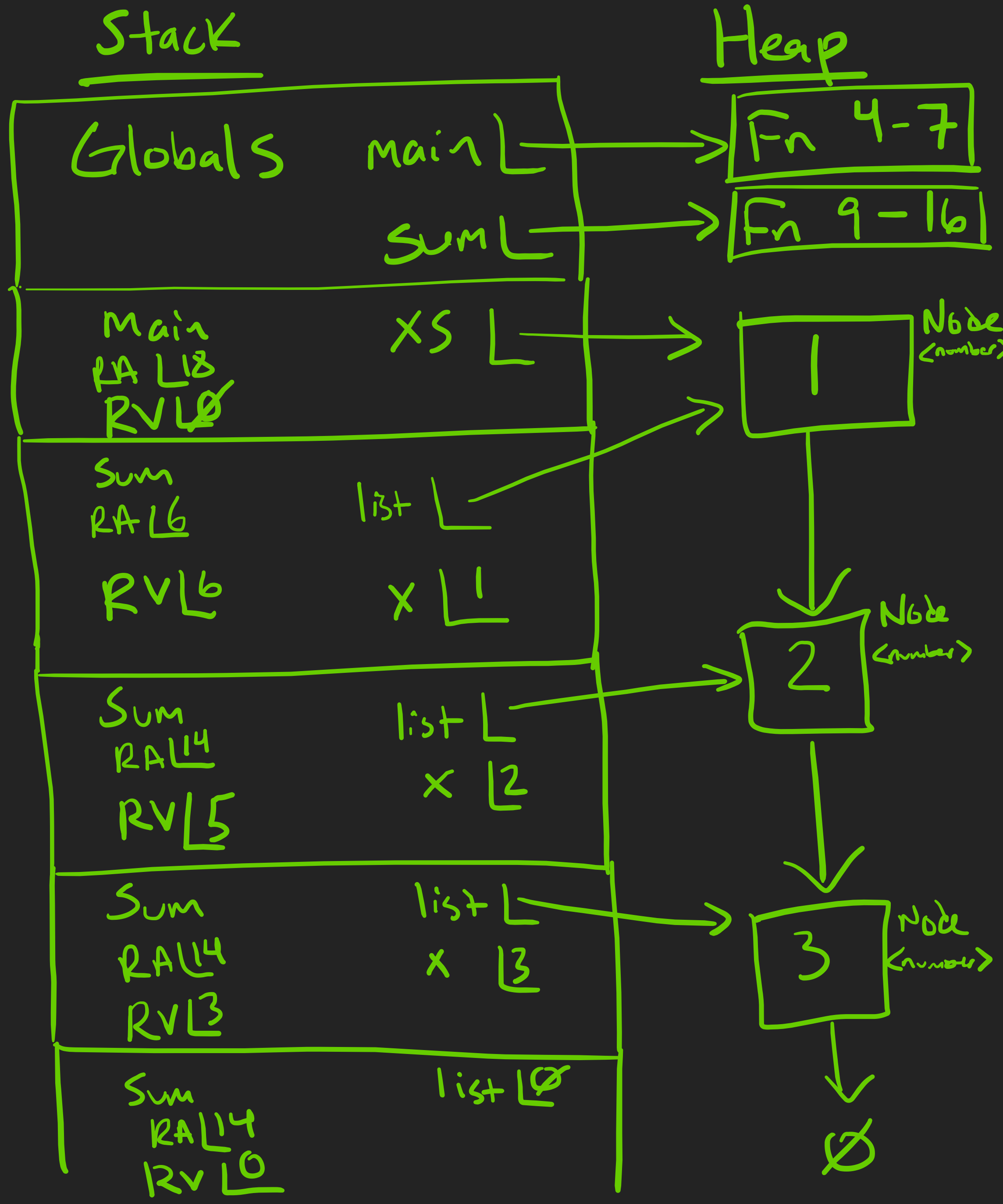
The built-in introcs/list functions

- Each of the following four functions can be imported from the list library:
  ```
  import { cons, first, rest, listify } from "introcs/list";
  ```
- `cons` – construct a list by prepending a value to the front of it
- `first` – read the first value from a list
- `rest` – the list following the first value
- `listify` – given any number of arguments, produce a list in that order
  - e.g. – `listify(1, 2, 3, 4)` produces a list of 1 -> 2 -> 3 -> 4 -> null
- Lists made with these built-ins will display as tables when printed.
  - The head / first Node's value will be bolded.
  - The list will be **`null`** terminated.

```
import { listify, first, rest, Node } from "introcs/list";
import { print } from "introcs";

export let main = async () => {
    let xs = listify(1, 2, 3);
    print(sum(xs));
};

let sum = (list: Node<number>): number => {
    if (list === null) {
        return 0;
    } else {
        let x = first(list);
        return x + sum(rest(list));
    }
};

main();
```

```
1   import { listify, first, rest, Node } from "introcs/list";
2   import { print } from "introcs";
3
4   export let main = async () => {
5       let xs = listify(1, 2, 3);
6       print(sum(xs));
7   };
8
9   let sum = (list: Node<number>): number => {
10      if (list === null) {
11          return 0;
12      } else {
13          let x = first(list);
14          return x + sum(rest(list));
15      }
16  };
17
18  main();
```

Output: 6



```typescript
import { listify, first, rest, Node } from "introcs/list";
import { print } from "introcs";

export let main = async () => {
    let cs = listify("a", "b", "c");
    print(g(cs));
};

let g = (n: Node<string>): string => {
    if (n === null) {
        return "!";
    } else {
        let s = first(n);
        return g(rest(n)) + s;
    }
};

main();
```

CQ1: Draw an environment diagram of the following program and respond on PollEv.com/compunc with the printed output.

```
import { listify, first, rest, Node } from "introcs/list";
import { print } from "introcs";

export let main = async () => {
    let cs = listify("a", "b", "c");
    print(g(cs));
};

let g = (n: Node<string>): string => {
    if (n === null) {
        return "!";
    } else {
        let s = first(n);
        return g(rest(n)) + s;
    }
};

main();
```

Building Lists Recursively

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 01-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**

Hands-on - **range**

- The purpose of the range function is to return a list of nodes starting from the low parameter incrementing by 1 until the high number.
- For example:
  - range(2, 5) returns 2 -> 3 -> 4 -> 5 -> null
- What is the base case?
- What is the simplest case you can imagine after the base case?
- What is the first element you're consing onto the natural recursion at each step?
- Check-in on PollEv.com/compunc when you think you've got it!