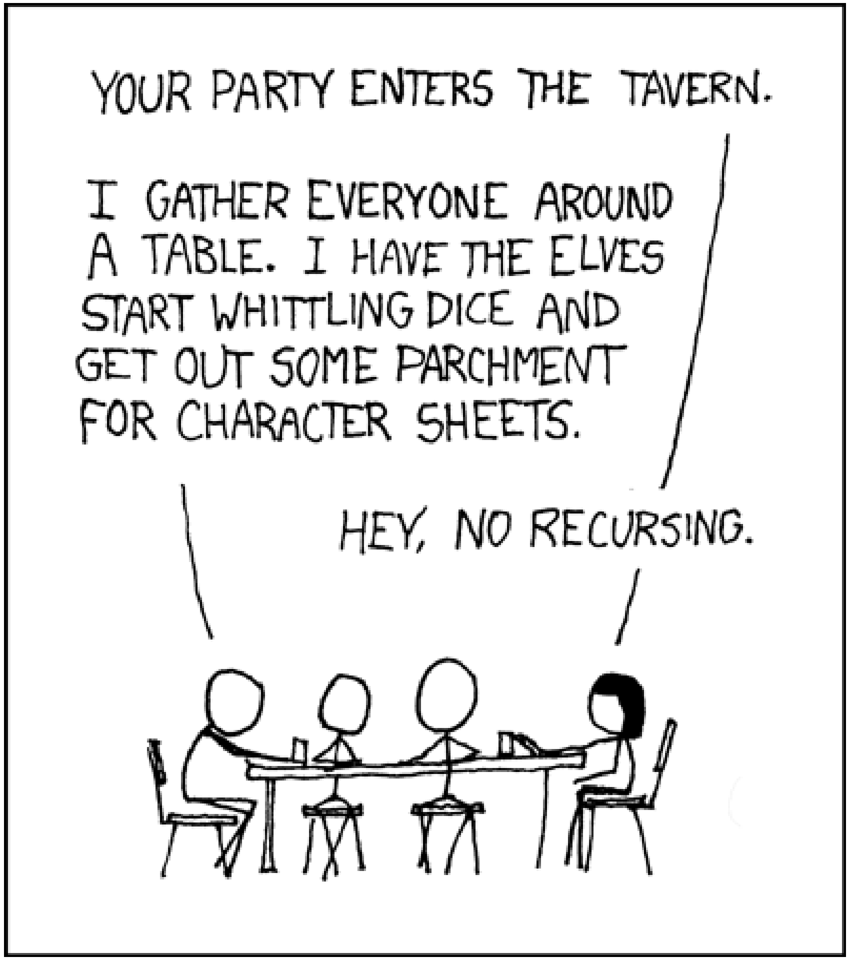
CSE 331
Spring 2025

Tail Recursion I



xkcd #244

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia (05/09)

- HW6 is out!
- note: will quickly wrap up one last Topic 6 example, *then* move on to Topic 7
  - this example is <u>very relevant</u> to your HW :)

Local Variable Mutation & Memory Use

- With only straight-line code & conditionals…
  - it seems like it saves memory
  - but it does not (compiler would fix anyway)

- With loops…
  - it really does save memory

    no improvement in **running time**
  - but loops cannot be used in all cases

    some problems really do require more memory

- When can loops be used and when not?

Sum of List: Recursive Math vs Iterative Code

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

Recursion can be directly translated into code

- **Loop to calculate sum of a list**

$\{\{\ L = L_0\ \}\}$

```
let s: bigint = 0n;
```

$\{\{\ \textbf{Inv}: \text{sum}(L_0) = s + \text{sum}(L)\ \}\}$

```
while (L.kind !== "nil") {
  s = s + L.hd;
  L = L.tl;
}
```

$\{\{\ s = \text{sum}(L_0)\ \}\}$

Sum of List: Recursion vs Loops, in Code

Loop

$\{\{\ L = L_0\ \}\}$

```
let s: bigint = 0n;
```

$\{\{\ \textbf{Inv}: \text{sum}(L_0) = s + \text{sum}(L)\ \}\}$

```
while (L.kind !== "nil") {
  s = s + L.hd;
  L = L.tl;
}
```

$\{\{\ s = \text{sum}(L_0)\ \}\}$

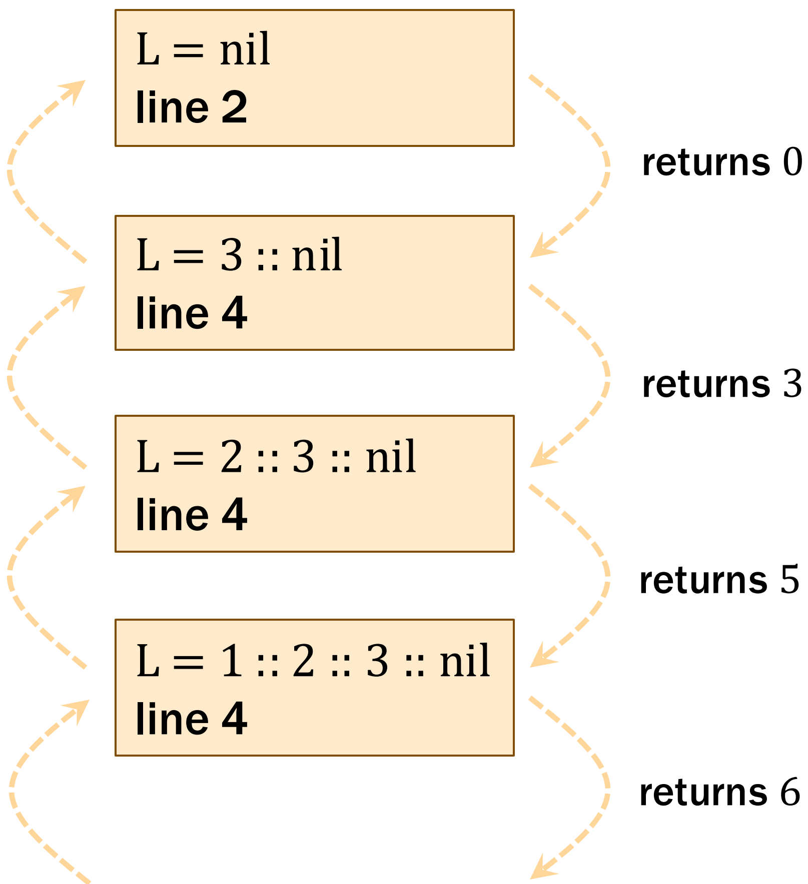
Recursion

```
const sum = (L: List): bigint => {
  if (L.kind === "nil") {
    return 0n;
  } else {
    return L.hd + sum(L.tl);
  }
}
```

Both run in $O(n)$ time where $n = \text{len}(L)$

Loop uses $O(1)$ extra memory, but right does not…

Recursive Version of Sum



… `sum(1 :: 2 :: 3 :: nil)` …

```
const sum = (L: List): bigint => {
1   if (L.kind === "nil") {
2     return 0n;
3   } else {
4     return L.hd + sum(L.tl);
5   }
}
```

List of length 3 takes 4 calls
List of length n takes $n+1$ calls.

Call uses $O(n)$ memory,
where $n = \text{len}(L)$

How much does space efficiency matter?

- In principle, this extra memory usually not a problem
  - $\mathrm{O(n)}$ time is usually the more important constraint

- In practice, sometimes we are memory constrained
  - in the browser, $\mathrm{sum(L)}$ exceeds stack size at $\mathrm{len(L)} = 10{,}000$

- Loops $\gg$ Recursion?

- Nope!
  1. Loops do not <u>always</u> use less memory.
  2. Recursion can solve <u>more problems</u> than loops.
  3. Extra memory use pays for some other benefits.

Another Sum of the Values in a List

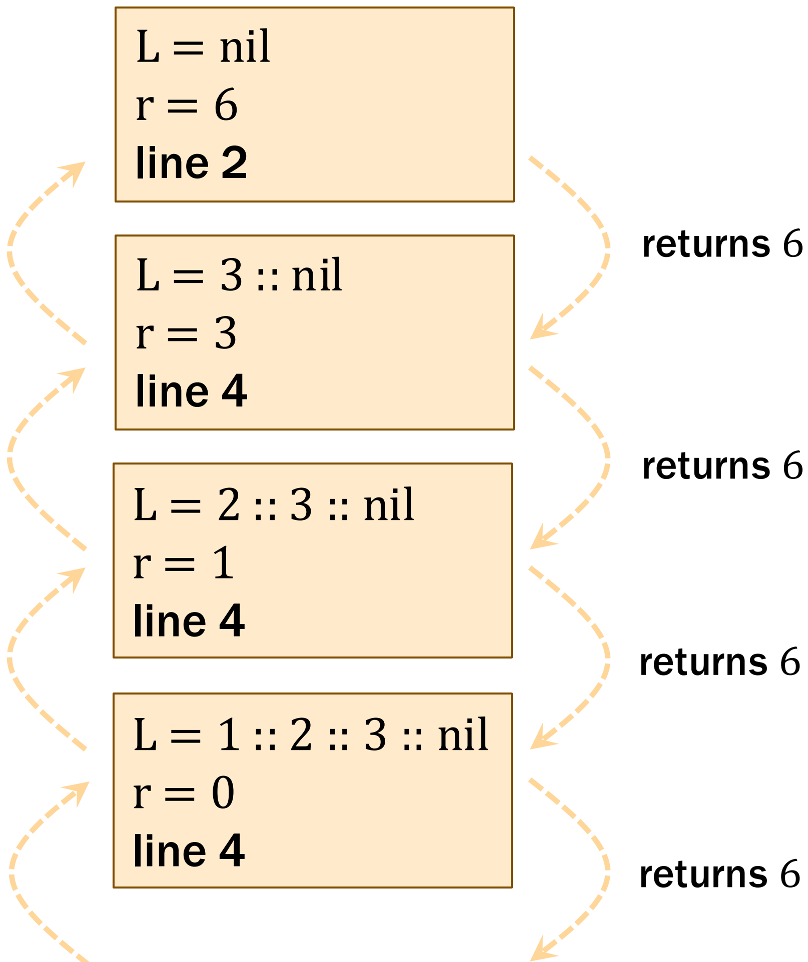
- Saw another summation function in Topic 5 Extras

$$
\begin{aligned}
\text{sum-acc}(\text{nil}, r) &:= r \\
\text{sum-acc}(x :: L, r) &:= \text{sum-acc}(L, x + r)
\end{aligned}
$$

- Translates to the following code

```
const sum_acc = (L: List, r: bigint): bigint => {
  if (L.kind === "nil") {
    return r;
  } else {
    return sum_acc(L.tl, L.hd + r);
  }
}
```

Tail-Recursive Version of Sum




```
const sum_acc =
    (L: List, r: bigint): bigint => {
1   if (L.kind === "nil") {
2     return r;
3   } else {
4     return sum_acc(L.tl, L.hd + r);
5   }
}
```

This is a "tail call" and "tail recursion".

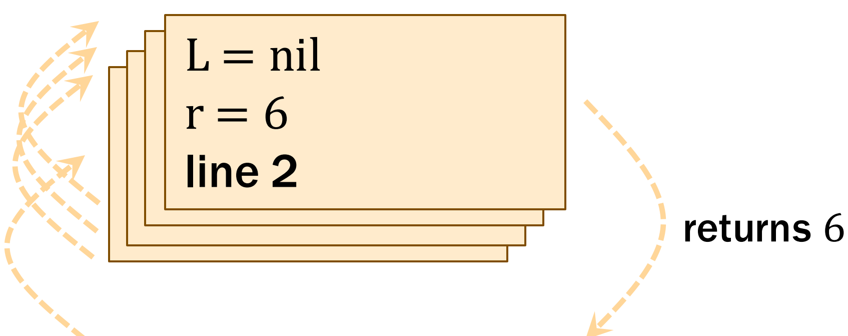
Same return value means no need to remember where we were.

No need to keep stack old frames!
Tail call optimization reuses them...

… `sum_acc(1 :: 2 :: 3 :: nil, 0)` …

Tail-Recursive Version of Sum, Optimized

```
const sum_acc =
   (L: List, r: bigint): bigint => {
1   if (L.kind === "nil") {
2     return r;
3   } else {
4     return sum_acc(L.tl, L.hd + r);
5   }
}
```



… **sum_acc**(1 :: 2 :: 3 :: nil, 0) …

Tail call optimization reuses stack frames so only O(1) memory

What does this look like? A loop!

`sum_acc` calculates the *same values* in the *same order* as the loop

Tail-Call Optimization

- Tail-call optimization turns tail recursion into a loop
- Functional languages implement tail-call optimization
  - standard feature of such languages
  - you don't write loops; you write tail recursive functions
- More on JS & tail-calls in a moment! But first…

Think-Pair-Share: Leaf Me Alone

Is this function tail-recursive?

```
type Tree =
{ kind: "leaf", value: bigint } |
{ kind: "branch", left: Tree, right: Tree };

const f = (node: Tree): bigint => {
  if (node.kind === "leaf") {
    return node.value;
  } else {
    return f(node.left) + f(node.right);
  }
}
```

No! The last thing we do is add!

sli.do #cse331

Think-Pair-Share: Tail Me Later

Is this function tail-recursive?

```
const g = (a: List<bigint>, b: List<bigint>): boolean => {
  if (a === nil && b === nil) {
    return true;
  }
   if (a === nil || b === nil) {
    return false;
  }
  if (a.hd !== b.hd) {
    return false;
  }
  return g(a.tl, b.tl);
}
```

sli.do #cse331

Yes! The last thing we do is return!

Think-Pair-Share: Be Mean or Be Square

Is this function tail-recursive?

```
const h =
  (a: List<number>, acc: number): number => {

  if (a === nil) {
    return Math.sqrt(acc);
  }
  return h(
        a.tl,
      acc + Math.pow(a.hd, 2)
  );
}
```

sli.do #cse331

Yes! The last thing we do is return!

Aside: Tail-Call Optimization & JavaScript

- technically, JavaScript's spec since ~ 2015 (TC39 v6) *says* it should have tail-call optimization (TCO), but…
  - Chrome added tail-call optimization… then undid it!*
  - other major browsers (e.g. Firefox) *never* implemented it!
  - one reason: loops / tail-call optimization have downsides (more later today …)
- in 2025,
  - Safari's engine (WebKit) supports TCO, as do derivative runtimes (e.g. Bun, which uses JavaScriptCore)
  - Chrome has put forward a (mostly-inactive) proposal for opt-in (explicit) TCO; it has a long and hotly debated history
  - Firefox does not have TCO
- tl;dr: you probably can't rely on it for browser apps

Loops vs Tail Recursion

Ordinary Loops ≤ Tail Recursion (with tail-call optimization)

- Tail recursion can solve all problems loop can
  - any loop can be translated to tail recursion
  - both use O(1) memory with tail-call optimization
- Translation is simple and important to understand
- Tells us that Ordinary Loops ≪ Recursion
  - correspond to the *special* case of tail recursion

Loop to Tail Recursion (1/2)

```
const myLoop = (R: List): T => {
  let s = f(R);
  while (R.kind !== "nil") {
    s = g(s, R.hd);
    R = R.tl;
  }
  return h(s);
};
```

{{ Inv: $\text{my-acc}(R_0, s_0) = \text{my-acc}(R, s)$ }}

- **Tail-recursive function that does same calculation:**

my-acc(nil, s)	:= h(s)	after loop
my-acc(x :: L, s)	:= my-acc(L, g(s, x))	loop body

my-func(L)	:= my-acc(L, f(L))	before loop

Loop to Tail Recursion (2/2)

```
const myLoop = (R: List): T => {
  let s = f(R);
  {{ Inv: my-acc(R0, s0) = my-acc(R, s) }}
  while (R.kind !== "nil") {
    s = g(s, R.hd);
    R = R.tl;
  }
  return h(s);
};
```

Inv formalizes the fact that we loop on tail recursion

recursive cases (tail calls)

base cases

- **Tail-recursive function that does same calculation:**

$\text{my-acc}(\text{nil}, s) \quad := h(s)$ — after loop

$\text{my-acc}(x :: L, s) \quad := \text{my-acc}(L, g(s, x))$ — loop body

$\text{my-func}(L) \quad := \text{my-acc}(L, f(L))$ — before loop

Example 1: Iterative Sum to Tail Recursion (1/2)

```
const sumLoop = (R: List): bigint => {
  let s = 0;
  while (R.kind !== "nil") {
    s = s + R.hd;
    R = R.tl;
  }
  return s;
};
```

- **Tail-recursive function that does same calculation:**

$$\text{sum-acc}(\text{nil}, s) \quad := h(s) \qquad\qquad h(s) \to s$$

$$\text{sum-acc}(x :: L, s) \quad := \text{my-acc}(L, g(s, x)) \qquad\qquad g(s, x) \to s + x$$

$$\text{sum-func}(L) \quad := \text{my-acc}(L, f(L)) \qquad\qquad f(L) \to 0$$

Example 1: Iterative Sum to Tail Recursion (2/2)

```
const sumLoop = (R: List): bigint => {
  let s = 0;
  while (R.kind !== "nil") {
    s = s + R.hd;
    R = R.tl;
  }
  return s;
};
```

{{ Inv: $\text{sum-acc}(R_0, s_0) = \text{sum-acc}(R, s)$ }}

- **Tail-recursive function that does same calculation:**

$$\text{sum-acc}(\text{nil}, s) := s$$
$$\text{sum-acc}(x :: L, s) := \text{sum-acc}(L, s + x)$$

$$\text{sum-func}(L) := \text{sum-acc}(L, 0)$$

Example 2: Iterative Max Value in a List (1/2)

```
const maxLoop = (R: List): bigint => {
  if (R.kind === "nil") throw …
  let s = R.hd;
  R = R.tl;
  while (R.kind !== "nil") {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

maxLoop(1 :: 3 :: 4 :: 2 :: nil)

Iteration	R	s

Example 2: Iterative Max Value in a List (2/2)

```
const maxLoop = (R: List): bigint => {
  if (R.kind === "nil") throw …
  let s = R.hd;
  R = R.tl;
  while (R.kind !== "nil") {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

maxLoop(1 :: 3 :: 4 :: 2 :: nil)

Iteration	R	s
0	3 :: 4 :: 2 :: nil	1
1	4 :: 2 :: nil	3
2	2 :: nil	4
3	nil	4

Example 2: Tail-Recursive Max Value in a List (1/3)

```
const maxLoop = (R: List): bigint => {
  if (R.kind === "nil") throw …
  let s = R.hd;
  R = R.tl;
  while (R.kind !== "nil") {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

$$\text{max-acc}(\text{nil}, s) := h(s) \qquad h(s) \to s$$

$$\text{max-acc}(x :: L, s) := \text{max-acc}(L, g(s, x)) \qquad g(s, x) \to \begin{cases} x & \textbf{if } x > s \\ s & \textbf{if } x \le s \end{cases}$$

$$\text{max-func}(L) := \text{max-acc}(L, f(L)) \qquad f(L) \to L.\text{hd} \ \textbf{if } L \ne \text{nil}$$

Example 2: Tail-Recursive Max Value in a List (2/3)

```
const maxLoop = (R: List): bigint => {
  if (R.kind === "nil") throw …
  let s = R.hd;
  R = R.tl;
  while (R.kind !== "nil") {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

{{ Inv: $\text{max-acc}(R_0, s_0) = \text{max-acc}(R, s)$ }}

$\text{max-acc}(\text{nil}, s) := s$

$\text{max-acc}(x :: L, s) := \text{max-acc}(L, x)$ if $x > s$

$\text{max-acc}(x :: L, s) := \text{max-acc}(L, s)$ if $x \le s$

$\text{max-func}(\text{nil}) := \text{undefined}$

$\text{max-func}(x :: L) := \text{max-acc}(L, x)$

Example 2: Tail-Recursive Max Value in a List (3/3)

```
const maxLoop = (R: List): bigint => {
  if (R.kind === "nil") throw …
  let s = R.hd;
  R = R.tl;
  while (R.kind !== "nil") {
    if (R.hd > s)
      s = R.hd;
    R = R.tl;
  }
  return s;
};
```

max-func(1 :: 3 :: 4 :: 2 :: nil)

max-func(1 :: 3 :: 4 :: 2 :: nil)	
= max-acc(3 :: 4 :: 2 :: nil, 1)	**def of …**
= max-acc(4 :: 2 :: nil, 3)	(**since** $3 > 1$)
= max-acc(2 :: nil, 4)	(**since** $4 > 3$)
= max-acc(nil, 4)	(**since** $2 \leq 4$)
= 4	

max-acc(nil, s)	:= s	
max-acc(x :: L, s)	:= max-acc(L, x)	if $x > s$
max-acc(x :: L, s)	:= max-acc(L, s)	if $x \leq s$

max-func(nil)	:= undefined
max-func(x ::L)	:= max-acc(L, x)

Loops vs Tail Recursion in Math

- Tail recursion gives nicer notation for loop operation

maxLoop(1 :: 3 :: 4 :: 2 :: nil)

Iteration	R	s
0	3 :: 4 :: 2 :: nil	1
1	4 :: 2 :: nil	3
2	2 :: nil	4
3	nil	4

max-func(1 :: 3 :: 4 :: 2 :: nil)

$$
\begin{aligned}
&\text{max-func}(1 :: 3 :: 4 :: 2 :: \text{nil}) \\
&= \text{max-acc}(3 :: 4 :: 2 :: \text{nil}, 1) && \textbf{def of ...} \\
&= \text{max-acc}(4 :: 2 :: \text{nil}, 3) && (\textbf{since } 3 > 1) \\
&= \text{max-acc}(2 :: \text{nil}, 4) && (\textbf{since } 4 > 3) \\
&= \text{max-acc}(\text{nil}, 4) && (\textbf{since } 2 \le 4) \\
&= 4
\end{aligned}
$$

- Loops are hard to describe with math
  - math never mutates anything, so loops are not a good fit
  - tail recursive notation shows loop operation in calculation block

Loops vs Tail Recursion as a Tradeoff

- Ordinary oops use less memory than (non-tail) recursion

- This is a tradeoff
  - save memory at the loss of information...

“Pausing” Iterative Max Value in a List (1/2)

```
const maxLoop = (R: List): bigint => {
1 if (R.kind === "nil") throw …
2 let s = R.hd;
3 R = R.tl;
4 while (R.kind !== "nil") {
5   if (R.hd > s)
6     s = R.hd;
7   R = R.tl;
8 }
9 return s;
};
```

Suppose we are at line 5
with $R = 4 :: 2 :: \text{nil}$ **and** $s = 3$

Could have started out with…

$R = 1 :: 3 :: 4 :: 2 :: \text{nil}$

$R = 3 :: 4 :: 2 :: \text{nil}$

$R = 0 :: 1 :: 3 :: 3 :: 1 :: 1 :: 1 :: 0 :: 4 :: 2 :: \text{nil}$

…

Could have been one of infinitely many lists!

“Pausing” Iterative Max Value in a List (2/2)

```
const maxLoop = (R: List): bigint => {
1 if (R.kind === "nil") throw …
2 let s = R.hd;
3 R = R.tl;
4 while (R.kind !== "nil") {
5   if (R.hd > s)
6     s = R.hd;
7   R = R.tl;
8 }
9 return s;
};
```

Suppose we are at line 4
with $R = 4 :: 2 :: \mathrm{nil}$ **and** $s = 3$

Could have been one of infinitely many lists!

Is there a situation where knowing **<u>how</u> we got to a line is important?**

It matters when debugging!

Loop saves memory at the cost of harder debugging.

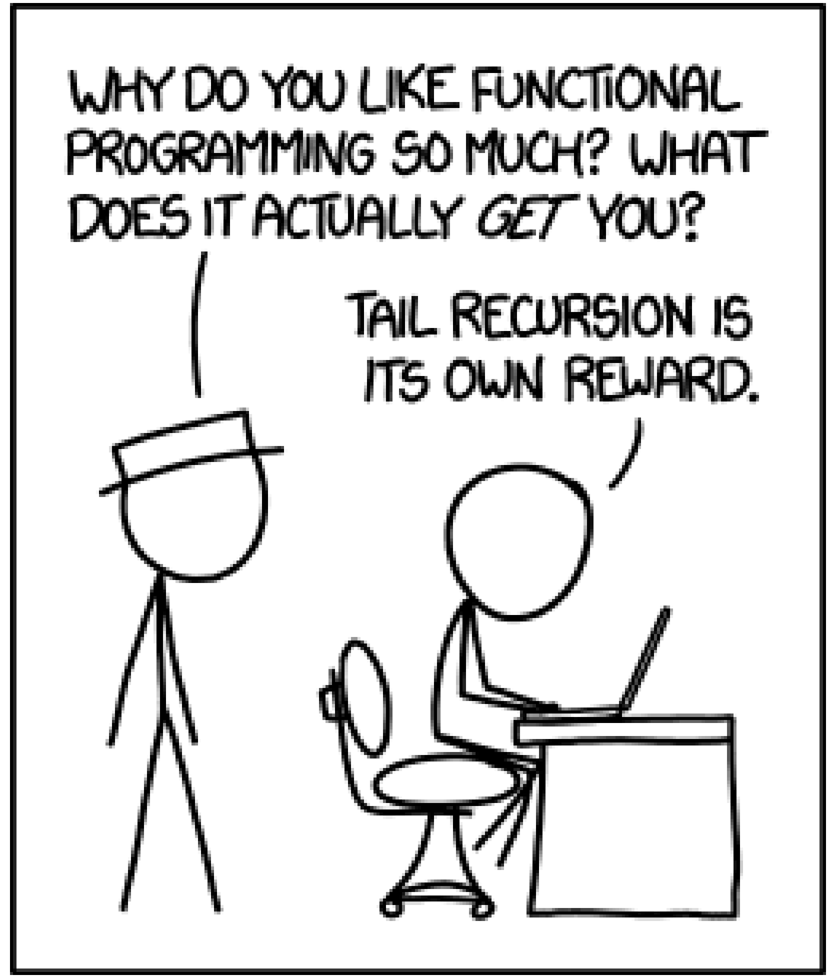
This is why (I think) Chrome removed the optimization.

Key Takeaways

- **Any loop can be translated to tail recursion**
  - **they describe the same *calculation***
    - tail recursive version *is a* loop (with tail call optimization)
  - **tail recursive notation is also useful for analyzing the loop**
- **Ordinary loops are strictly *less powerful* than recursion**
  - **not all recursive functions can be written as tail recursion**
  - **many problems cannot be solved in O(1) memory**
    - e.g., tree traversals *require* extra space
    - many (most?) list operations require extra space
- **Ordinary loops save memory but are harder to debug**
  - **information thrown away tells you how you got there**

CSE 331
Spring 2025

Tail Recursion II



xkcd #1270

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia (05/12)

- New: math conventions page
  - nothing should come as a surprise
  - Work-in-progress – please give us feedback!

Recall: Key Takeaways

- **Any loop can be translated to tail recursion**
  - **they describe the same *calculation***

    tail recursive version *is a* loop (with tail call optimization)
  - **tail recursive notation is also useful for analyzing the loop**

- **Ordinary loops are strictly *less powerful* than recursion**
  - **"ordinary loops" being loops with constant memory**
  - **not all recursive functions can be written as tail recursion**
  - **many problems cannot be solved in O(1) memory**

    e.g., tree traversals *require* extra space

    many (most?) list operations require extra space

- **Ordinary loops save memory but are harder to debug**
  - **information thrown away tells you how you got there**

Recall: Loop to Tail Recursion

```
const myLoop = (R: List): T => {
  let s = f(R);
  {{ Inv: my-acc(R0, s0) = my-acc(R, s) }}
  while (R.kind !== "nil") {
    s = g(s, R.hd);
    R = R.tl;
  }
  return h(s);
};
```

Inv formalizes the fact that we loop on tail recursion

recursive cases (tail calls)

base cases

- **Tail-recursive function that does same calculation:**

my-acc(nil, s)	:= h(s)	after loop
my-acc(x :: L, s)	:= my-acc(L, g(s, x))	loop body
my-func(L)	:= my-acc(L, f(L))	before loop

Ordinary Loop & Recursion Equivalence

Ordinary Loops ≈ Tail Recursion (with tail-call optimization)

- Can solve exactly the same problems
  - can translate any loop to tail recursion
  - can translate any tail recursive function to an ordinary loop
- Translation is simple and important to understand
  - do this if your recursion runs out of stack space in Chrome
- Let's look at an example...

Faster Len

$$\begin{aligned} &\text{len}(\text{nil}) && := 0 \\ &\text{len}(x :: L) && := 1 + \text{len}(L) \end{aligned}$$

$$\begin{aligned} &\text{len-acc}(\text{nil}, r) && := r \\ &\text{len-acc}(x :: L, r) && := \text{len-acc}(L, r + 1) \end{aligned}$$

- **Both versions are recursive and $O(n)$ time**
  - <u>**second**</u> **version is tail recursive**
- **Can show that** $\text{len-acc}(S, r) = \text{len}(S) + r$
  - **proved by structural induction**
  - **tells us that** $\text{len-acc}(S, 0) = \text{len}(S)$

Translating Faster Len to a Loop

$$\text{len-acc}(\text{nil}, r) := r$$
$$\text{len-acc}(x :: L, r) := \text{len-acc}(L, r + 1)$$

- **Could implement** len-acc **with a loop as:**

```
const len_acc = (S: List, r: bigint): bigint => {
  {{ Inv: len-acc(S_0, r_0) = len-acc(S, r) }}
  while (S.kind !== "nil") {
    r = r + 1;
    S = S.tl;
  }
  return r;
};
```

recursive cases (tail calls)

base cases

  - **clear that the invariant holds initially**

Proving len_acc Correct (1/4)

$$\begin{aligned} \text{len-acc}(\text{nil}, r) &:= r \\ \text{len-acc}(x :: L, r) &:= \text{len-acc}(L, r + 1) \end{aligned}$$

- **Could implement** len-acc **with a loop as:**

```
const len_acc = (S: List, r: bigint): bigint => {
  {{ Inv: len-acc(S_0, r_0) = len-acc(S, r) }}
  while (S.kind !== "nil") {
    r = r + 1;
    S = S.tl;
  }
  {{ len-acc(S_0, r_0) = len-acc(S, r) and S = nil }}
  {{ len-acc(S_0, r_0) = r }}
  return r;
};
```

$$\begin{aligned} \text{len-acc}(S_0, r_0) &= \text{len-acc}(S, r) \\ &= \text{len-acc}(\text{nil}, r) && \textbf{since } S = \text{nil} \\ &= r && \textbf{def of } \text{len-acc} \end{aligned}$$

Proving len_acc Correct (2/4)

len-acc(nil, r) := r

len-acc(x :: L, r) := len-acc(L, r + 1)

- **Could implement** len-acc **with a loop as:**

```
const len_acc = (S: List, r: bigint): bigint => {
  {{ Inv: len-acc(S_0, r_0) = len-acc(S, r) }}
  while (S.kind !== "nil") {
    {{ len-acc(S_0, r_0) = len-acc(S, r) and S = S.hd :: S.tl }}
    r = r + 1;
    S = S.tl;
    {{ len-acc(S_0, r_0) = len-acc(S, r) }}
  }
  return r;
};
```

Proving len_acc Correct (3/4)

$$\text{len-acc}(\text{nil}, r) := r$$
$$\text{len-acc}(x :: L, r) := \text{len-acc}(L, r + 1)$$

- **Could implement** len-acc **with a loop as:**

```
const len_acc = (S: List, r: bigint): bigint => {
  {{ Inv: len-acc(S_0, r_0) = len-acc(S, r) }}
  while (S.kind !== "nil") {
    {{ len-acc(S_0, r_0) = len-acc(S, r) and S = S.hd :: S.tl }}
    {{ len-acc(S_0, r_0) = len-acc(S.tl, r + 1) }}
    r = r + 1;
    S = S.tl;
    {{ len-acc(S_0, r_0) = len-acc(S, r) }}
  }
  return r;
};
```

Proving len_acc Correct (4/4)

len-acc(nil, r) := r
len-acc(x :: L, r) := len-acc(L, r + 1)

- **Could implement** len-acc **with a loop as:**

```
const len_acc = (S: List, r: bigint): bigint => {
  {{ Inv: len-acc(S0, r0) = len-acc(S, r) }}
  while (S.kind !== "nil") {
    {{ len-acc(S0, r0) = len-acc(S, r) and S = S.hd :: S.tl }}
    {{ len-acc(S0, r0) = len-acc(S.tl, r + 1) }}
    r = r + 1;
    S = S.tl;
  }
  return r;
};
```

len-acc(S_0, r_0)
= len-acc(S, r)
= len-acc(S.hd :: S.tl, r) **since** S = S.hd :: S.tl
= len-acc(S.tl, r + 1) **def of** len-acc

Generallizing Tail Recursion to a Loop (1/2)

$$\text{sum-acc}(\text{nil}, r) := r$$
$$\text{sum-acc}(x :: L, r) := \text{sum-acc}(L, x + r)$$

- **Two types of rules in the definition**
  - **base case: calculate an answer from the argument**
  - **recursive case: recurses with new arguments**

    tail recursion requires that we return whatever that call returns

Generallizing Tail Recursion to a Loop (2/2)

$f(\ldots\ p_1\ \ldots,\ r)\ :=\ \ldots$

…

$f(\ldots\ p_n\ \ldots,\ r)\ :=\ \ldots$

base cases

$f(\ldots\ q_1\ \ldots,\ r)\ :=\ f(\ldots)$

…

$f(\ldots\ q_n\ \ldots,\ r)\ :=\ f(\ldots)$

recursive cases (tail calls *only*)

- **Tail-recursive function becomes a loop:**

```
// Inv: f(args0) = f(args)
while (args /* match some q pattern */) {
  args = /* right-side of appropriate q pattern */;
}
return /* right-side of appropriate p pattern */;
```

Rewriting the Invariant (1/3)

```
// Inv: sum-acc(S0, r0) = sum-acc(S, r)
while (S.kind !== "nil") {
  r = S.hd + r;
  S = S.tl;
}
return r;
```

- This is the most direct invariant
  - says answer with current arguments is the original answer
  - shows that this implements sum-acc but not sum

- Can be rewritten to show it implements sum
  - use the relationship we proved between sum-acc and sum

Rewriting the Invariant (2/3)

```
// Inv: sum-acc(S0, r0) = sum-acc(S, r)
```

- Can be rewritten using $\text{sum-acc}(S, r) = \text{sum}(S) + r$

```
// Inv: sum(S0) + r0 = sum(S) + r
```

- Can use the fact that we set the initial value of r

```
let r = 0;
// Inv: sum(S0) = sum(S) + r
```

Rewriting the Invariant (3/3)

$$\begin{aligned} \mathrm{sum}(\mathrm{nil}) &:= 0 \\ \mathrm{sum}(x :: L) &:= x + \mathrm{sum}(L) \end{aligned}$$

- **Final version of the loop:**

```
let r = 0;
// Inv: sum(S_0) = sum(S) + r
while (S.kind !== "nil") {
  r = S.hd + r;
  S = S.tl;
}
return r;
```

- **Erased all evidence of our tail recursive version ;)**
  - **will practice this on the homework**

Worked Example: Last Element (1/4)

$$\begin{aligned} &\text{last}(\text{nil}) && := \text{undefined} \\ &\text{last}(x :: \text{nil}) && := x \\ &\text{last}(x :: y :: L) && := \text{last}(y :: L) \end{aligned}$$

- **Returns the last element of the list**
  - **only defined if the list is non-empty**

    otherwise, there is no last element

- **This is already tail recursive**
  - **so we can translate it into a loop…**

Worked Example: Last Element (2/4)

last(nil)	:= undefined
last(x :: nil)	:= x
last(x :: y :: L)	:= last(y :: L)] tail recursive case

- **Translate to a loop:**

```
// @param S a non-empty list
const last = (S: List) => bigint {
  // Inv: last(S_0) = last(S)
  while (args /* match some recursive pattern */) {
    args = /* right-side of recursive pattern */;
  }
  return /* right-side of base case pattern */;
};
```

Worked Example: Last Element (3/4)

last(nil)	:= undefined	base cases
last(x :: nil)	:= x	
last(x :: y :: L)	:= last(y :: L)	tail recursive case

- **Translate to a loop:**

```
// @param S a non-empty list
const last = (S: List) => bigint {
  // Inv: last(S_0) = last(S)
  while (S.kind !== "nil" && S.tl.kind !== "nil") {
    S = S.tl;
  }
  return /* right-side of base case pattern */;
};
```

Worked Example: Last Element (4/4)

last(nil)	:= undefined	base cases
last(x :: nil)	:= x	
last(x :: y :: L)	:= last(y :: L)	tail recursive case

- **Translate to a loop:**

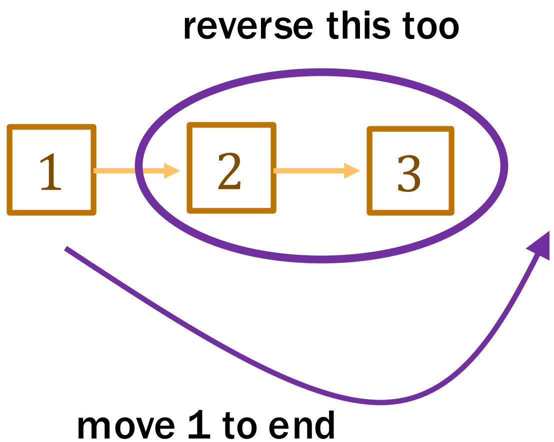
```
// @param S a non-empty list
const last = (S: List) => bigint {
  // Inv: last(S0) = last(S)
  while (S.kind !== "nil" && S.tl.kind !== "nil") {
    S = S.tl;
  }
  if (S.kind === "nil")
    throw new Error("no last element!");
  return S.hd;
};
```

Reversing a List

- **Mathematical definition of** $\text{rev}(S)$

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) + [x] \end{aligned}$$

  – **note that** rev **uses** concat $(+)$ **as a helper function**



Reversing a List (Slowly)

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$

- This correctly reverses a list but is slow
  - concat takes $\Theta(n)$ time, where n is length of L
  - n calls to concat takes $\Theta(n^2)$ time

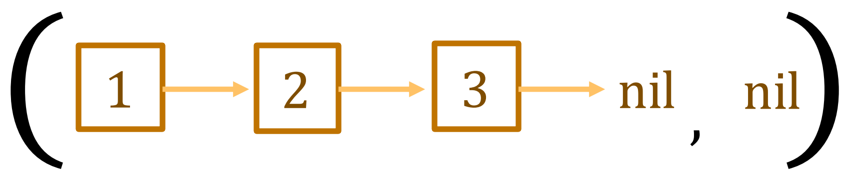
- Can we do this faster?
  - yes, but we need a helper function

Tracing Through Faster List Reversal (1/4)

- **Helper function** rev-acc(S, R) **for any** S, R : List

rev-acc(nil, R) := R

rev-acc(x :: L, R) := rev-acc(L, x :: R)

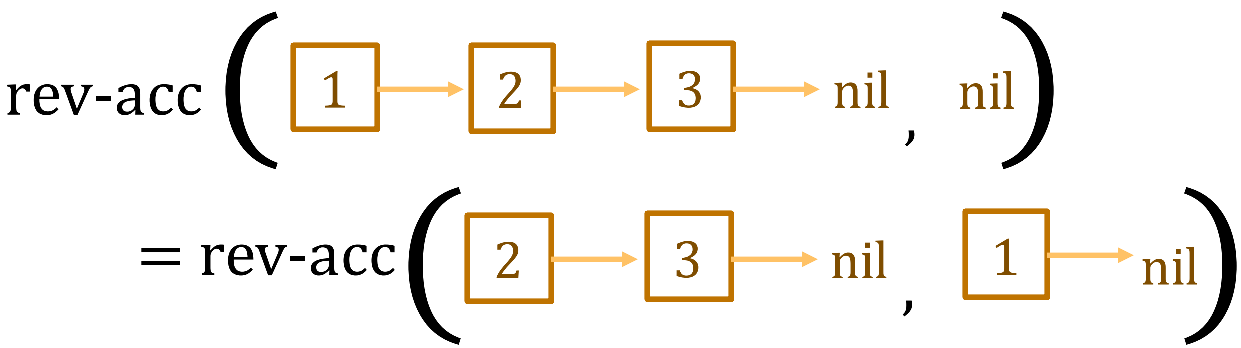
rev-acc 

Tracing Through Faster List Reversal (2/4)

- **Helper function** rev-acc(S, R) **for any** S, R : List

$$\text{rev-acc}(\text{nil}, R) := R$$

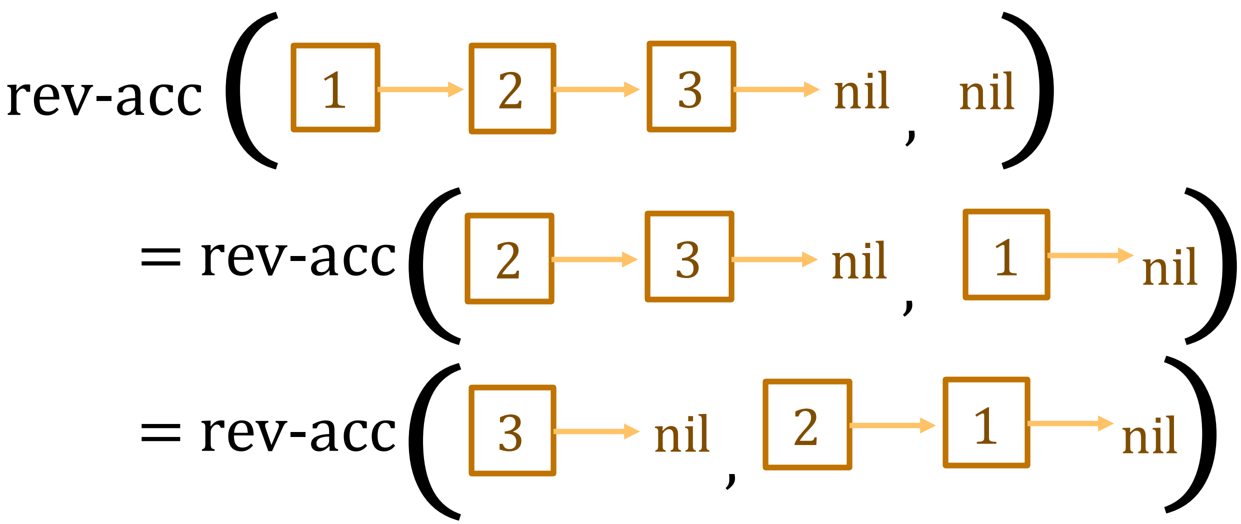
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$



Tracing Through Faster List Reversal (3/4)

- **Helper function** rev-acc(S, R) **for any** S, R : List

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

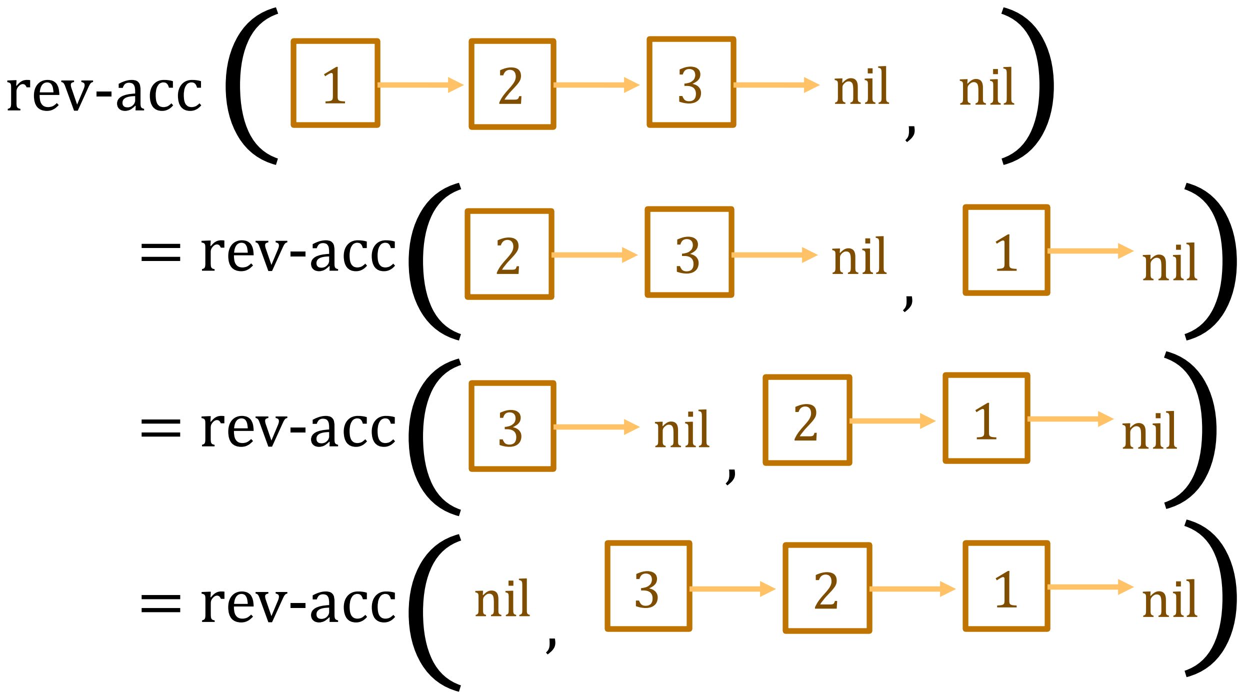


Tracing Through Faster List Reversal (4/4)

- **Helper function** rev-acc(S, R) **for any** S, R : List

$$\text{rev-acc}(\text{nil}, R) := R$$

$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$



Reversing a List Quickly: Proof Setup (1/3)

$$\begin{aligned}
\text{rev}(\text{nil}) &:= \text{nil} \\
\text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x]
\end{aligned}$$

$$\begin{aligned}
\text{rev-acc}(\text{nil}, R) &:= R \\
\text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R)
\end{aligned}$$

- **To show the relationship between** rev **and** rev-acc**, we need a few properties of** concat ($\mathbin{+\!\!+}$)**:**

$A \mathbin{+\!\!+} [] = A$	Identity
$A \mathbin{+\!\!+} (B \mathbin{+\!\!+} C) = (A \mathbin{+\!\!+} B) \mathbin{+\!\!+} C$	Associativity

  - **both are familiar properties for numbers and strings**
  - **these say the same facts hold for lists with "$\mathbin{+\!\!+}$"**

    these and other properties of $\mathbin{+\!\!+}$ are mentioned in the notes on lists

Reversing a List Quickly: Proof Setup (2/3)

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) +\!\!+ [x] \end{aligned}$$

$$\begin{aligned} \text{rev-acc}(\text{nil}, R) &:= R \\ \text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R) \end{aligned}$$

- **The general relationship between the two is this:**

$$\text{rev-acc}(S, R) = \text{rev}(S) +\!\!+ R \qquad \textbf{Lemma}$$

  – **same issue arose with** sum-acc

  there we had: $\text{sum-acc}(S, r) = \text{sum}(S) + r$

  – **need to explain the role of the "accumulator variable" also**

Reversing a List Quickly: Proof Setup (3/3)

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$

$$\begin{aligned} \text{rev-acc}(\text{nil}, R) &:= R \\ \text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R) \end{aligned}$$

- **The general relationship between the two is this:**

$$\text{rev-acc}(S, R) = \text{rev}(S) \mathbin{+\!\!+} R \qquad \textbf{Lemma}$$

- **This shows us that** $\text{rev}(S) = \text{rev-acc}(S, [])$

$$\begin{aligned} \text{rev-acc}(S, []) &= \text{rev}(S) \mathbin{+\!\!+} [] \qquad \textbf{Lemma} \\ &= \text{rev}(S) \end{aligned}$$

Proving Helper Lemma: Base Case (1/2)

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

- **Prove that** $\text{rev-acc}(S, R) = \text{rev}(S) \mathbin{+\!\!\!+} R$
  - **prove by induction on** S **(so** R **remains a variable)**

Base Case (nil):

$$\text{rev-acc}(\text{nil}, R) =$$

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

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

$$\text{rev}(\text{nil}) := \text{nil}$$
$$\text{rev}(x :: L) := \text{rev}(L) \mathbin{+\!\!\!+} [x]$$

Proving Helper Lemma: Base Case (2/2)

$$\begin{aligned} \text{rev-acc}(\text{nil}, R) &:= R \\ \text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R) \end{aligned}$$

- **Prove that** $\text{rev-acc}(S, R) = \text{rev}(S) \mathbin{+\!\!+} R$
  - **prove by induction on** S **(so** R **remains a variable)**

Base Case (nil):

$\text{rev-acc}(\text{nil}, R)$	$= R$	**def of** rev-acc
	$= \text{concat}(\text{nil}, R)$	**def of** concat
	$= \text{concat}(\text{rev}(\text{nil}), R)$	**def of** rev

$$\begin{aligned} \text{concat}(\text{nil}, R) &:= R \\ \text{concat}(x :: L, R) &:= x :: \text{concat}(L, R) \end{aligned}$$

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$

Proving Helper Lemma: Inductive Step (1/2)

$$\begin{aligned} \text{rev-acc}(\text{nil}, R) &:= R \\ \text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R) \end{aligned}$$

- **Prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$

  Inductive Hypothesis: assume that $\text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R)$ **for any** R

  Inductive Step $(x :: L)$**:**

$\text{rev-acc}(x :: L, R) \quad =$

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

$$\begin{aligned} \text{concat}(\text{nil}, R) &:= R \\ \text{concat}(x :: L, R) &:= x :: \text{concat}(L, R) \end{aligned}$$

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$

Proving Helper Lemma: Inductive Step (2/2)

rev-acc(nil, R)	:=	R
rev-acc(x :: L, R)	:=	rev-acc(L, x :: R)

- **Prove that** rev-acc(S, R) = concat(rev(S), R)

  Inductive Hypothesis: **assume that** rev-acc(L, R) = concat(rev(L), R) **for any** R

  Inductive Step (x :: L):

rev-acc(x :: L, R)	= rev-acc(L, x :: R)	**def of** rev-acc
	= concat(rev(L), x :: R)	**Ind. Hyp.**
	= rev(L) # (x :: R)	
	= rev(L) # (x :: concat(nil, R))	**def of** concat
	= rev(L) # ([x] # R)	**def of** concat
	= (rev(L) # [x]) # R	**assoc. of** #
	= concat(rev(L) # [x], R)	
	= concat(rev(x :: L), R)	**def of** rev

concat(nil, R)	:=	R
concat(x :: L, R)	:=	x :: concat(L, R)

rev(nil)	:=	nil
rev(x :: L)	:=	rev(L) # [x]

Implementing rev-acc as a Loop

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

how do we implement this?

- **Tail-recursive function becomes a loop:**

```
// Inv: rev-acc(S₀, R₀) = rev-acc(S, R)
while (S.kind !== "nil") {
  R = cons(S.hd, R);
  S = S.tl;
}
return R;
```

- **Now, use this to calculate** $\text{rev}(S) = \text{rev-acc}(S, \text{nil})$

Tightening the Loop Invariant

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

- **Calculate** $\text{rev}(S)$ **with loop:**

```
const rev = (S: List): List => {
  let R = nil;
  // Inv: rev-acc(S₀, R₀) = rev-acc(S, R)
  while (S.kind !== "nil") {
    R = cons(S.hd, R);
    S = S.tl;
  }
  return R;
}
```

Invariant still mentions rev-acc

Destroy the evidence!

$$\text{rev-acc}(S, R) = \text{rev}(S) \mathbin{+\!\!+} R$$

Tightening the Invariant Some More…

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

- **Calculate** $\text{rev}(S)$ **with loop:**

```
const rev = (S: List): List => {
  let R = nil;
  // Inv: rev(S₀) ++ R₀ = rev(S) ++ R
  while (S.kind !== "nil") {
    R = cons(S.hd, R);
    S = S.tl;
  }
  return R;
}
```

We know $R_0 = []$

And $\text{rev}(S) \mathbin{+\!\!+} [] = \text{rev}(S)$

Finalized Loop Version of rev-acc

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

- **Calculate** $\text{rev}(S)$ **with loop:**

```
const rev = (S: List): List => {
  let R = nil;
  // Inv: rev(S_0) = rev(S) ++ R
  while (S.kind !== "nil") {
    R = cons(S.hd, R);
    S = S.tl;
  }
  return R;
}
```

Options for proving **correctness**:

- Prove relationship btw two recursive functions. Then, implement tail recursion with template.
- Prove loop correct with Floyd logic.

Zooming out on Loops & Recursion

- Ordinary loops are a special case of recursion
  - recursion is more powerful
  - recursion is necessary in many cases (e.g., tree traversals)

    even most list functions *require* extra space

- Likely lingering questions…
  - does this conversion work for *all* list functions?
  - what about functions on other data types?
  - what kinds of problems can neither really solve?

"Bottom Up" Functions on Lists (1/4)

$$\begin{aligned} \text{twice}(\text{nil}) &:= \text{nil} \\ \text{twice}(x :: L) &:= (2x) :: \text{twice}(L) \end{aligned}$$

- **The opposite of "tail recursion" is purely "bottom up"**
  - **tail recursion does the work "top down"**

    all the work is done as we move down the list
  - **this definition is "bottom up"**

    all the work is done as we work back from nil to the full list

"Bottom Up" Functions on Lists (2/4)

twice(nil) := nil
twice(x :: L) := (2x) :: twice(L)

- **Attempt to do this with an accumulator**

twice-acc(nil, R) := R
twice-acc(x :: L, R) := twice-acc(L, (2x) :: R)

  - **this could be implemented with a loop**
  - **but it's incorrect...**

"Bottom Up" Functions on Lists (3/4)

$$\begin{aligned}
\text{twice}(\text{nil}) &:= \text{nil} \\
\text{twice}(x :: L) &:= (2x) :: \text{twice}(L)
\end{aligned}$$

- **Attempt to do this with an accumulator**

$$\begin{aligned}
\text{twice-acc}(\text{nil}, R) &:= R \\
\text{twice-acc}(x :: L, R) &:= \text{twice-acc}(L, (2x) :: R)
\end{aligned}$$

$$\begin{aligned}
&\text{twice}(1 :: 2 :: 3 :: \text{nil}) \\
&\quad = 2 :: \text{twice}(2 :: 3 :: \text{nil}) && \textbf{def of } \text{twice} \\
&\quad = 2 :: 4 :: \text{twice}(3 :: \text{nil}) && \textbf{def of } \text{twice} \\
&\quad = 2 :: 4 :: 6 :: \text{twice}(\text{nil}) && \textbf{def of } \text{twice} \\
&\quad = 2 :: 4 :: 6 :: \text{nil} && \textbf{def of } \text{twice}
\end{aligned}$$

"Bottom Up" Functions on Lists (4/4)

twice(nil) := nil
twice(x :: L) := (2x) :: twice(L)

- **Attempt to do this with an accumulator**

twice-acc(nil, R) := R
twice-acc(x :: L, R) := twice-acc(L, (2x) :: R)

twice(1 :: 2 :: 3 :: nil) = ... 2 :: 4 :: 6 :: nil

twice-acc(1 :: 2 :: 3 :: nil, nil)
= twice-acc(2 :: 3 :: nil, 2 :: nil) **def of** twice-acc
= twice-acc(3 :: nil, 4 :: 2 :: nil) **def of** twice-acc
= twice-acc(nil, 6 :: 4 :: 2 :: nil) **def of** twice-acc
= 6 :: 4 :: 2 :: nil **def of** twice-acc

Clickbait Ending

- Ordinary loops are a special case of recursion
  - recursion is more powerful
  - recursion is necessary in many cases (e.g., tree traversals)

    even most list functions *require* extra space

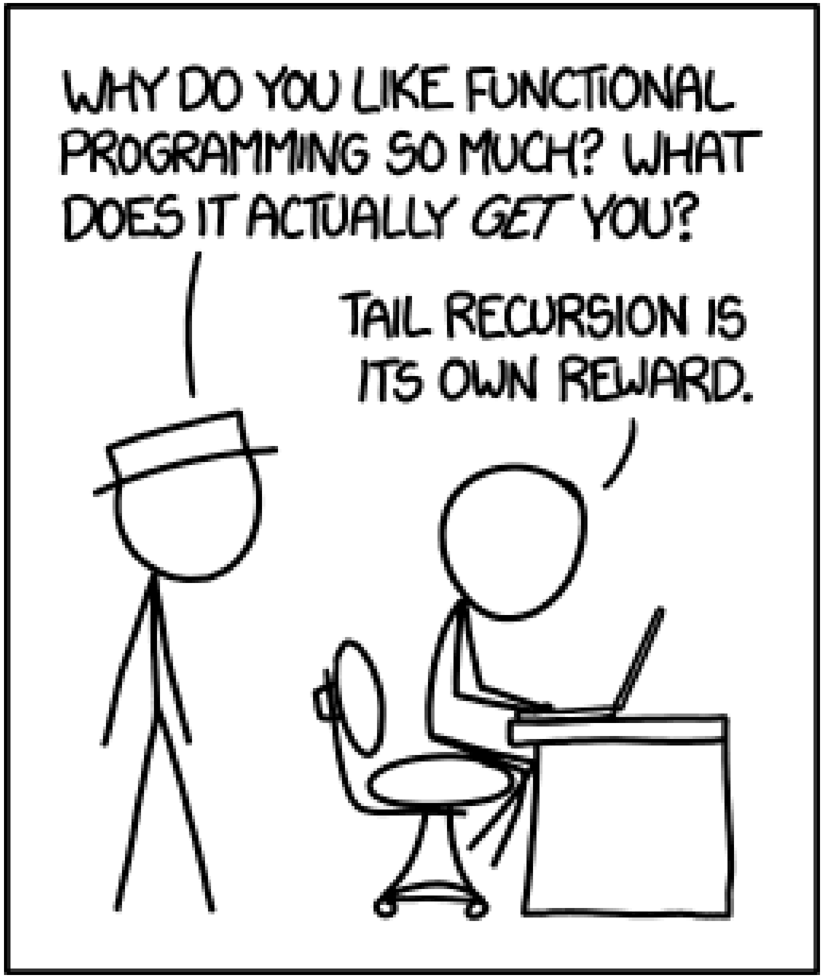
- Likely lingering questions…
  - does this conversion work for *all* list functions?

    by default, no – it seems not all problems are tail-recursive

    but, a tool we learned today *could* fix that problem for us…
  - what about functions on other data types?
  - what kinds of problems can neither really solve?

CSE 331
Spring 2025

Tail & Bottom-up Recursion



xkcd #1270

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia (05/14)

- Reminder: new math conventions page
  - nothing should come as a surprise
  - Work-in-progress – please give us feedback!
- Previous Ed announcement on better (hopefully!) explanation of rev-acc inductive step

Recall: Loops, Recursion, and Cliffhangers

- Ordinary loops are a special case of recursion
  - recursion is more powerful
  - recursion is necessary in many cases (e.g., tree traversals)

    even most list functions *require* extra space

- Likely lingering questions…
  - does this conversion work for *all* list functions?
  - what about functions on other data types?
  - what kinds of problems can neither really solve?

Recall: "Bottom Up" Functions on Lists (1/4)

$$\text{twice}(\text{nil}) := \text{nil}$$
$$\text{twice}(x :: L) := (2x) :: \text{twice}(L)$$

- **The opposite of "tail recursion" is purely "bottom up"**
  - **tail recursion does the work "top down"**

    all the work is done as we move down the list
  - **this definition is "bottom up"**

    all the work is done as we work back from nil to the full list

Recall: "Bottom Up" Functions on Lists (2/4)

twice(nil) := nil
twice(x :: L) := (2x) :: twice(L)

- **Attempt to do this with an accumulator**

twice-acc(nil, R) := R
twice-acc(x :: L, R) := twice-acc(L, (2x) :: R)

  – **this could be implemented with a loop**
  – **but it's incorrect...**

Recall: "Bottom Up" Functions on Lists (3/4)

$$\begin{aligned} \text{twice}(\text{nil}) &:= \text{nil} \\ \text{twice}(x :: L) &:= (2x) :: \text{twice}(L) \end{aligned}$$

- **Attempt to do this with an accumulator**

$$\begin{aligned} \text{twice-acc}(\text{nil}, R) &:= R \\ \text{twice-acc}(x :: L, R) &:= \text{twice-acc}(L, (2x) :: R) \end{aligned}$$

$$\begin{aligned} &\text{twice}(1 :: 2 :: 3 :: \text{nil}) && \\ &\quad = 2 :: \text{twice}(2 :: 3 :: \text{nil}) && \textbf{def of } \text{twice} \\ &\quad = 2 :: 4 :: \text{twice}(3 :: \text{nil}) && \textbf{def of } \text{twice} \\ &\quad = 2 :: 4 :: 6 :: \text{twice}(\text{nil}) && \textbf{def of } \text{twice} \\ &\quad = 2 :: 4 :: 6 :: \text{nil} && \textbf{def of } \text{twice} \end{aligned}$$

Recall: "Bottom Up" Functions on Lists (4/4)

twice(nil) := nil
twice(x :: L) := (2x) :: twice(L)

- **Attempt to do this with an accumulator**

twice-acc(nil, R) := R
twice-acc(x :: L, R) := twice-acc(L, (2x) :: R)

twice(1 :: 2 :: 3 :: nil) = ... 2 :: 4 :: 6 :: nil

twice-acc(1 :: 2 :: 3 :: nil, nil)
= twice-acc(2 :: 3 :: nil, 2 :: nil) **def of** twice-acc
= twice-acc(3 :: nil, 4 :: 2 :: nil) **def of** twice-acc
= twice-acc(nil, 6 :: 4 :: 2 :: nil) **def of** twice-acc
= 6 :: 4 :: 2 :: nil **def of** twice-acc

This Twice Is (not) Right!

$$\text{twice}(\text{nil}) := \text{nil}$$
$$\text{twice}(x :: L) := (2x) :: \text{twice}(L)$$

- **Attempt to do this with an accumulator**

$$\text{twice-acc}(\text{nil}, R) := R$$
$$\text{twice-acc}(x :: L, R) := \text{twice-acc}(L, (2x) :: R)$$

  - **we end up with** $\text{twice-acc}(L, \text{nil}) = \text{rev}(\text{twice}(L))$
  - **we can fix this by reversing the result when we're done**
    we return $\text{rev}(\text{twice-acc}(L, \text{nil}))$
  - **or, we can reverse the list (once) before we recurse**
  - **either lets us use a loop, but neither is** $O(1)$ **memory**

Fixing Twice by "Cheating": Rev Before

twice-acc(nil, R) := R

twice-acc(x :: L, R) := twice-acc(L, (2x) :: R)

twice(L) := twice-acc(rev(L), nil)

```
const twice = (L: List): List => {
    let R = nil;
    let S = rev(L);
    while (S.kind !== "nil") {
      R = cons(2n * S.hd, R);
      S = S.tl;
    }
    return R;  // = twice(L)
}
```

Fixing Twice by “Cheating”: Rev After

twice-acc(nil, R) := R

twice-acc(x :: L, R) := twice-acc(L, (2x) :: R)

twice(L) := rev(twice-acc(L, nil))

```
const twice = (L: List): List => {
    let R = nil;
    while (L.kind !== "nil") {
      R = cons(2n * L.hd, R);
      L = L.tl;
    }
    return rev(R);  // = twice(L)
}
```

Generalizing "The Twice is Right"

- **for any** $g\colon A \to A$ **and** $f\colon \mathrm{List}\langle A\rangle \to \mathrm{List}\langle A\rangle$,

$$f(\mathrm{nil}) := \mathrm{nil}$$
$$f(x :: L) := g(x) :: f(L)$$

we can define

$$\text{f-acc}(\mathrm{nil}, R) := R$$
$$\text{f-acc}(x :: L, R) := \text{f-acc}(L, g(x) :: R)$$

and show that

$$\text{f-acc}(L, R) = \mathrm{rev}(f(L)) +\!\!+ R$$

thus

$$\text{f-acc}(L, \mathrm{nil}) = \mathrm{rev}(f(L)) \qquad \text{("reversing before")}$$
$$f(L) = \mathrm{rev}(\text{f-acc}(L, \mathrm{nil})) \qquad \text{("reversing after"*)}$$

Proving f-acc(ts): Proof Goal

$$\text{f-acc}(\text{nil}, R) := R$$
$$\text{f-acc}(x :: L, R) := \text{f-acc}(L, g(x) :: R)$$

- **Prove that** $\text{f-acc}(L, R) = \text{rev}(f(L)) + R$
  - **prove by structural induction on** L **(so** R **remains a variable)**
- **Will use prior definitions of concat & rev**
  - **these are very commonly used in recursive list code**

$$f(\text{nil}) := \text{nil}$$
$$f(x :: L) := g(x) :: f(L)$$

Proving f-acc(ts): Base Case (1/2)

$$\begin{aligned} \text{f-acc}(\text{nil}, R) &:= R \\ \text{f-acc}(x :: L, R) &:= \text{f-acc}(L, g(x) :: R) \end{aligned}$$

- **Prove that** $\text{f-acc}(L, R) = \text{rev}(f(L)) \mathbin{+\!\!+} R$

Base Case (nil):

$\text{f-acc}(\text{nil}, R)$

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

Proving f-acc(ts): Base Case (2/2)

$$\begin{aligned} \text{f-acc}(\text{nil}, R) &:= R \\ \text{f-acc}(x :: L, R) &:= \text{f-acc}(L, g(x) :: R) \end{aligned}$$

- **Prove that** $\text{f-acc}(L, R) = \text{rev}(f(L)) +\!\!+ R$

Base Case (nil):

$\text{f-acc}(\text{nil}, R)$	$= R$	**def of** f-acc
	$= \text{concat}(\text{nil}, R)$	**def of** concat
	$= \text{concat}(\text{rev}(\text{nil}), R)$	**def of** rev
	$= \text{concat}(\text{rev}(f(\text{nil})), R)$	**def of** f
	$= \text{rev}(f(L)) +\!\!+ R$	

$$\begin{aligned} f(\text{nil}) &:= \text{nil} \\ f(x :: L) &:= g(x) :: f(L) \end{aligned}$$

$$\begin{aligned} \text{concat}(\text{nil}, R) &:= R \\ \text{concat}(x :: L, R) &:= x :: \text{concat}(L, R) \end{aligned}$$

Proving f-acc(ts): Inductive Step (1/3)

f-acc(nil, R)	:=	R
f-acc(x :: L, R)	:=	f-acc(L, g(x) :: R)

- **Prove that** f-acc(L, R) = rev(f(L)) ++ R

Inductive Hypothesis: **assume that** f-acc(L, R) = rev(f(L)) ++ R **for any** R

Inductive Step (x :: L):

f-acc(x :: L, R)

f(nil)	:=	nil	concat(nil, R)	:=	R
f(x :: L)	:=	g(x) :: f(L)	concat(x :: L, R)	:=	x :: concat(L, R)

Proving f-acc(ts): Inductive Step (2/3)

$$\text{f-acc}(\text{nil}, R) := R$$
$$\text{f-acc}(x :: L, R) := \text{f-acc}(L, g(x) :: R)$$

- **Prove that** $\text{f-acc}(L, R) = \text{rev}(f(L)) \mathbin{+\!\!+} R$

Inductive Hypothesis: assume that $\text{f-acc}(L, R) = \text{rev}(f(L)) \mathbin{+\!\!+} R$ **for any** R

Inductive Step $(x :: L)$**:**

$\text{f-acc}(x :: L, R)$	$= \text{f-acc}(L, g(x) :: R)$	**def of** f-acc
	$= \text{rev}(f(L)) \mathbin{+\!\!+} g(x) :: R$	**Ind. Hyp.**
	$= \text{rev}(f(L)) \mathbin{+\!\!+} g(x) :: \text{nil} \mathbin{+\!\!+} R$	**def of concat (1)**
	$= \text{rev}(f(L)) \mathbin{+\!\!+} [g(x)] \mathbin{+\!\!+} R$	**def of concat (2)**

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

Proving f-acc(ts): Inductive Step (3/3)

$$\begin{aligned} \text{f-acc}(\text{nil}, R) &:= R \\ \text{f-acc}(x :: L, R) &:= \text{f-acc}(L, g(x) :: R) \end{aligned}$$

- **Prove that** $\text{f-acc}(L, R) = \text{rev}(f(L)) \mathbin{+\!\!+} R$

Inductive Hypothesis: assume that $\text{f-acc}(L, R) = \text{rev}(f(L)) \mathbin{+\!\!+} R$ **for any** R

Inductive Step $(x :: L)$**:**

$\text{f-acc}(x :: L, R)$	$= \text{f-acc}(L, g(x) :: R)$	**def of** f-acc
	$= \text{rev}(f(L)) \mathbin{+\!\!+} g(x) :: R$	**Ind. Hyp.**
	$= \text{rev}(f(L)) \mathbin{+\!\!+} g(x) :: \text{nil} \mathbin{+\!\!+} R$	**def of concat (1)**
	$= \text{rev}(f(L)) \mathbin{+\!\!+} [g(x)] \mathbin{+\!\!+} R$	**def of concat (2)**
	$= \text{rev}(g(x) :: f(L)) \mathbin{+\!\!+} R$	**def of** rev
	$= \text{rev}(f(x :: L)) \mathbin{+\!\!+} R$	**def of** f

$f(\text{nil}) := \text{nil}$ $f(x :: L) := g(x) :: f(L)$	$\text{rev}(\text{nil}) := \text{nil}$ $\text{rev}(x :: L) := \text{rev}(L) \mathbin{+\!\!+} [x]$

f-acc(ts) in Code

$$\text{f-acc}(\text{nil}, R) := R$$
$$\text{f-acc}(x :: L, R) := \text{f-acc}(L, g(x) :: R)$$

$$f(L) := \text{rev}(\text{f-acc}(L, \text{nil}))$$

```
const f = (L: List): List => {
    let R = nil;
    {{ Inv: f-acc(L_0, R_0) = f-acc(L, R) }}
    while (L.kind !== "nil") {
      R = cons(g(L.hd), R);
      L = L.tl;
    }
    return rev(R);  // = f(L)
}
```

Proving the Loop f-acc(ts) Correct: Initialization

f-acc(nil, R) := R

f-acc(x :: L, R) := f-acc(L, g(x) :: R)

f(L) := rev(f-acc(L, nil))

```
const f = (L: List): List => {
    let R = nil;
    {{ Inv: f-acc(L0, R0) = f-acc(L, R) }}
    while (L.kind !== "nil") {
      R = cons(g(L.hd), R);
      L = L.tl;
    }
    return rev(R);  // = f(L)
}
```

initialization holds immediately!

Proving the Loop f-acc(ts) Correct: Exit

f-acc(nil, R) := R

f-acc(x :: L, R) := f-acc(L, g(x) :: R)

f(L) := rev(f-acc(L, nil))

```
...
{{ Inv: f-acc(L_0, R_0) = f-acc(L, R) }}
while (L.kind !== "nil") {
  ...
}
{{ f-acc(L_0, R_0) = f-acc(L, R) and L = nil }}
{{ rev(R) = f(L) }}
return rev(R);  // = f(L)
```

rev(R)

= rev(f-acc(nil, R)) — **def of f-acc**

= rev(f-acc(L, R)) — **since** $L = nil$

= rev(f-acc(L_0, R_0)) — **Inv**

= rev(f-acc(L, nil)) — **since** $R_0 = nil$

= f(L) — **def of f**

need to check exit satisfies postcondition

Proving the Loop f-acc(ts) Correct: Body (1/2)

f-acc(nil, R) := R

f-acc(x :: L, R) := f-acc(L, g(x) :: R)

f(L) := rev(f-acc(L, nil))

…

```
let R = nil;
{{ Inv: f-acc(L0, R0) = f-acc(L, R) }}
while (L.kind !== "nil") {
  {{ f-acc(L0, R0) = f-acc(L, R) and L ≠ nil }}
  R = cons(g(L.hd), R);
  L = L.tl;
  {{ f-acc(L0, R0) = f-acc(L, R) }}
```

…

Proving the Loop f-acc(ts) Correct: Body (2/2)

f-acc(nil, R) := R
f-acc(x :: L, R) := f-acc(L, g(x) :: R)

f(L) := rev(f-acc(L, nil))

```
…
let R = nil;
{{ Inv: f-acc(L_0, R_0) = f-acc(L, R) }}
while (L.kind !== "nil") {
  {{ f-acc(L_0, R_0) = f-acc(L, R) and L = L.hd :: L.tl }}
  {{ f-acc(L_0, R_0) = f-acc(L.tl, g(x) :: R) }}
  R = cons(g(L.hd), R);
  L = L.tl;
  {{ f-acc(L_0, R_0) = f-acc(L, R) }}
…
```

need to check …

true by def of f-acc!

Rewriting the f-acc(ts) Invariant

```
let R = nil;
```

{{ **Inv**: $\text{f-acc}(L_0, R_0) = \text{f-acc}(L, R)$ }}

$\text{f-acc}(L, R)$	$= \text{rev}(f(L)) +\!\!+ R$	**by earlier proof**
$\text{f-acc}(L_0, R_0)$	$= \text{rev}(f(L_0)) +\!\!+ R_0$	**by earlier proof**
	$= \text{rev}(f(L_0))$	**as** $R_0 = \text{nil}$

therefore...

{{ **Inv**: $\text{rev}(f(L_0)) = \text{rev}(f(L)) +\!\!+ R$ }}

$\text{f-acc}(L, R) = \text{rev}(f(L)) +\!\!+ R$

f-acc(ts) in Code, Rewritten

f-acc(nil, R) := R

f-acc(x :: L, R) := f-acc(L, g(x) :: R)

f(L) := rev(f-acc(L, nil))

```
const f = (L: List): List => {
    let R = nil;
    {{ Inv: rev(f(L0)) = rev(f(L)) ++ R }}
    while (L.kind !== "nil") {
      R = g(L.hd);
      L = L.tl;
    }
    return rev(R);  // = f(L)
}
```

Taking Stock: Element-wise Processing

- a function like

  f(nil) := nil

  f(x :: L) := g(x) :: f(L)

  can always be written tail-recursively with our "reversal" trick, but it *won't* be O(1) space
- O(n) space is reasonable, since it returns a list
  - loop version is not any better
- is this helpful?
  - yes: can use recursion reasoning while still writing loops
  - no: feels like … overkill?
  - also: bread-and-butter in pure functional languages (e.g. OCaml, Haskell*) – see "map" and "fold"

When is Tail Recursion Natural (or Efficient)?

- there's been a secret hidden pattern for:
  - what's "easy" with tail recursion (aka "loop order", or front-to-back)
  - what's "easy" with bottom-up recursion (aka "natural recursive order", or back-to-front)

- let's compare a few examples we've seen, and contrast to examples that *won't* work

Speed Round: Sum

Do you think this tail-recursive version is correct?

Normal version:

$$\text{sum}(\text{nil}) \quad := 0$$

$$\text{sum}(x :: L) \quad := x + \text{sum}(L)$$

"Tail-Recursive Version":

$$\text{sum-a}(\text{nil}, r) \quad := r$$

$$\text{sum-a}(x :: L, r) \quad := \text{sum-a}(L, x + r)$$

$$\text{sum}(S) \quad \sim \text{sum-a}(S, 0)$$

sli.do #cse331

Yes! Proof left for the reader, but let's trace:

$$\text{sum}([1, 3, 5]) = 1 + \text{sum}([3, 5]) = 1 + 3 + \text{sum}([5]) = 1 + 3 + 5 + \text{sum}([]) = \ldots = 9$$

$$\text{sum-a}([1, 3, 5], 0) = \text{sum-a}([3, 5], 1) = \text{sum-a}([5], 4) = \text{sum-a}([], 9) = 9$$

Speed Round: Sub

Do you think this tail-recursive version is correct?

Normal version:

$$\text{sub}(\text{nil}) \quad := 0$$

$$\text{sub}(x :: L) \quad := x - \text{sub}(L)$$

"Tail-Recursive Version":

$$\text{sub-a}(\text{nil}, r) \quad := r$$

$$\text{sub-a}(x :: L, r) \quad := \text{sub-a}(L, r - x)$$

$$\text{sub}(S) \quad \sim \text{sub-a}(S, 0)$$

sli.do #cse331

No! Let's try a smaller-than-previous example:

$\text{sub}([1, 3]) = 1 - \text{sub}([3]) = 1 - (3 - \text{sub}([])) = 1 - (3 - 0) = -2$

$\text{sub-a}([1, 3], 0) = \text{sub-a}([3], -1) = \text{sub-a}([], -4) = -4$

Speed Round: Sub, But Fixed!!

Do you think this tail-recursive version is correct?

Normal version:

sub(nil) := 0

sub(x :: L) := x - sub(L)

"Tail-Recursive Version":

sub-b(nil, r) := r

sub-b(x :: L, r) := sub-b(L, x - r)

sub(S) ~ sub-b(S, 0)

sli.do #cse331

No! Let's try a smaller-than-previous example:

sub([1, 3]) = 1 - sub([3]) = 1 – (3 - sub([])) = 1 – (3 – 0) = -2

sub-b([1, 3], 0) = sub-b([3], 1) = sub-b([], 2) = 2

Speed Round: Flatten

Do you think this tail-recursive version is correct?

Normal version:

$$\text{flatten}(\text{nil}) \quad := []$$

$$\text{flatten}(x :: L) \quad := x \mathbin{+\!\!+} \text{flatten}(L)$$

"Tail-Recursive Version":

$$\text{flatten-acc}(\text{nil}, r) \quad := r$$

$$\text{flatten-acc}(x :: L, r) \quad := \text{flatten-acc}(L, r \mathbin{+\!\!+} x)$$

$$\text{flatten}(x) \quad \sim \text{flatten-acc}(x, [])$$

sli.do #cse331

Example Flatten:

$$\text{flatten}([[1, 2, 3], [2, 4, 6, 8], [7, 8, 9]]) = [1, 2, 3, 2, 4, 6, 8, 7, 8, 9]$$

Tracing flatten (Bottom-Up Recursion)

$\text{flatten}(\text{nil}) \quad := []$

$\text{flatten}(x :: L) \quad := x +\!\!+ \text{flatten}(L)$

$\text{flatten}_0([[1, 2, 3], [2, 4, 6, 8], [7, 8, 9]])$

$= [1, 2, 3] +\!\!+ \text{flatten}_1([[2, 4, 6, 8], [7, 8, 9]])$

$= [1, 2, 3] +\!\!+ [2, 4, 6, 8] +\!\!+ \text{flatten}_2([[7, 8, 9]])$

$= [1, 2, 3] +\!\!+ [2, 4, 6, 8] +\!\!+ [7, 8, 9] +\!\!+ \text{nil}$

$= [1, 2, 3] +\!\!+ [2, 4, 6, 8] +\!\!+ [7, 8, 9]$ resolve flatten_2

$= [1, 2, 3] +\!\!+ [2, 4, 6, 8, 7, 8, 9]$ resolve flatten_1

$= [1, 2, 3, 2, 4, 6, 8, 7, 8, 9]$ resolve flatten_0

Tracing flatten-acc (Tail Recursion)

$$\begin{aligned} \text{flatten-acc}(\text{nil}, r) &:= r \\ \text{flatten-acc}(x :: L, r) &:= \text{flatten-acc}(L, r \mathbin{+\!\!+} x) \\ \text{flatten}(x) &\sim \text{flatten-acc}(x, []) \end{aligned}$$

$$\text{flatten-acc}([[1, 2, 3], [2, 4, 6, 8], [7, 8, 9]], [])$$

$$= \text{flatten-acc}([[2, 4, 6, 8], [7, 8, 9]], [] \mathbin{+\!\!+} [1, 2, 3])$$

$$= \text{flatten-acc}([[7, 8, 9]], [1, 2, 3] \mathbin{+\!\!+} [2, 4, 6, 8])$$

$$= \text{flatten-acc}([], [1, 2, 3, 2, 4, 6, 8] \mathbin{+\!\!+} [7, 8, 9])$$

$$= [1, 2, 3, 2, 4, 6, 8, 7, 8, 9]$$

A Tale of Two Flattens

"Abstract" Flatten:

$$[1, 2, 3] \mathbin{+\!\!+} [2, 4, 6, 8] \mathbin{+\!\!+} [7, 8, 9]$$

flatten (bottom-up recursion)

$$[1, 2, 3] \mathbin{+\!\!+} ([2, 4, 6, 8] \mathbin{+\!\!+} ([7, 8, 9] \mathbin{+\!\!+} []))$$

flatten-acc (top-down recursion)

$$(([] \mathbin{+\!\!+} [1, 2, 3]) \mathbin{+\!\!+} [2, 4, 6, 8]) \mathbin{+\!\!+} [7, 8, 9]$$

$$[1, 2, 3, 2, 4, 6, 8, 7, 8, 9]$$

same answer! why?

concat is associative: $(a \mathbin{+\!\!+} b) \mathbin{+\!\!+} c = a \mathbin{+\!\!+} (b \mathbin{+\!\!+} c)$

House of the Rising Sum

"Abstract" Sum:

$$1 + 3 + 5$$

sum (bottom-up recursion)

$$1 + (3 + (5 + 0))$$

sum-acc (top-down recursion)

$$((0 + 1) + 3) + 5$$

$$9$$

same answer! why?

addition is *associative*: $(a + b) + c = a + (b + c)$

No Subs, Please

"Abstract" Sub:

$1 - 3$

sub-b is like our "reverse" trick

sub (bottom-up recursion)

$1 - (3 - 0)$

$= -2$

sub-a (top-down recursion)

$(0 - 1) - 3$

$= -4$

sub-b (top-down recursion)

$3 - (1 - 0)$

$= 2$

answer is "reversed"

different answers! why?

subtraction is <u>not</u> associative: $(a - b) - c \neq a - (b - c)$

(*and, the 0's – more of a technicality)

Defining Associativity (Loosely)

- if an operator $\circ$ is left-associative, then

$$a \circ b \circ c = (a \circ b) \circ c$$

- if an operator $\circ$ is right-associative, then

$$a \circ b \circ c = a \circ (b \circ c)$$

- an operator that is both left & right-associative is just “associative”, and thus, we get

$$(a \circ b) \circ c = a \circ (b \circ c)$$

Coming Back to Tail Recursion

- our loop ↔ tail recursion trick works particularly well for all associative operators (and, functions!)
  - also: multiplication, "max of list" examples from earlier

- can apply this elsewhere, e.g.
  - string concatenation
  - set intersection & union
  - standard boolean & bitwise ops (AND, OR, XOR)
  - modular arithmetic
  - function composition (!!)

Okay Buddy, But Does This Get Me a Job?

- common post-123 question:
  "when should I use a loop vs recursion?"
  – one common (imperfect) answer:
    "use the strategy that mirrors your data"

- now have vocabulary for one interesting framing
  – **left-associative** operations lend themselves to top-down recursion (aka loops or tail-recursion)
  – **right-associative** operations lend themselves to bottom-up recursion (aka "natural" recursion)
  – for operations that are both (**associative**), go wild :)

Some Brief Footnotes

- left & right-associativity are programming languages terms
  - very common consideration in compiler & parser implementation
  - in functional programming languages, this conversation generalizes to “foldl” versus “foldr” (with performance implications)

- in math, this is one motivation for studying semigroups
  - though this is probably beyond what you need now (or … ever?)

Bonus: Tail Recursion “modulo cons” (1/3)

- Many very smart programming languages & compilers engineers think about fast tail calls

- Functional languages like OCaml & Haskell* have all sorts of tricks to make tail recursion *very* fast
  - includes some “cheating” with language design
  - some also present in GCC, LLVM (and thus, C, C++, …)

- Common bag of tricks: tail recursion “modulo ___”
  - most famous: “Tail recursion modulo cons” (~1970s)
  - also: tail recursion modulo addition, multiplication, …

Bonus: Tail Recursion "modulo cons" (2/3)

- Discovering Tail Recursion modulo cons yourself is *very* rewarding, so I won't spoil all of it for you
  - it requires some knowledge of what's in the call stack & how function calls work – the stuff in CSE 351
  - but, you *technically* know enough already :)

- Here's a hint:
  - if you know that every function returns either a direct value, a function call, *or* cons on one of the two…
  - can you "shift" the cons to the next function call (plumbing required) to go back to being tail-recursive?

Bonus: Tail Recursion “modulo cons” (3/3)

Google

tail recursion modulo cons

James R. Wilcox
https://jamesrwilcox.com › tail-mod-cons

Tail Recursion Modulo cons

Apr 10, 2014 — **Tail recursion modulo cons**, which allows post-processing the result of the recursive call with a call to cons.

OCaml
https://ocaml.org › manual › tail_mod_cons

The “Tail Modulo Constructor” program transformation

The **tail-mod-cons transformation** preserves the performance of the original, non-tail-recursive version, while a continuation-passing-style transformation incurs ...

ACM Digital Library
https://dl.acm.org › doi

Tail Recursion Modulo Context: An Equational Approach

The tail-recursion modulo cons transformation **can rewrite functions that are not quite tail-recursive** into a tail-recursive form that can be executed ...

Hal-Inria
https://inria.hal.science › document PDF

Tail Modulo Cons

by F Bour · 2021 · Cited by 13 — **Tail-recursion modulo cons** was well-known in the Lisp community as early as the 1970s. For example the REMREC system (Risch, 1973) would ...

Wrapping up Recursion vs Loops

- There is a fundamental tension between:
  - Natural recursive order (bottom-up, aka back-to-front)
  - Natural loop order (front-to-back)
  - Some problems lean towards one or the other; highly related to their **associativity**

- Three ways to bridge this gap:
  - Make the loop serve the recursion

    Bottom-up list loop template calling $\mathrm{rev}(\mathrm{L})$ (and other complex things)
  - Make the recursion serve the loop

    Tail recursion
  - Change the data structure

    that's our next unit :))