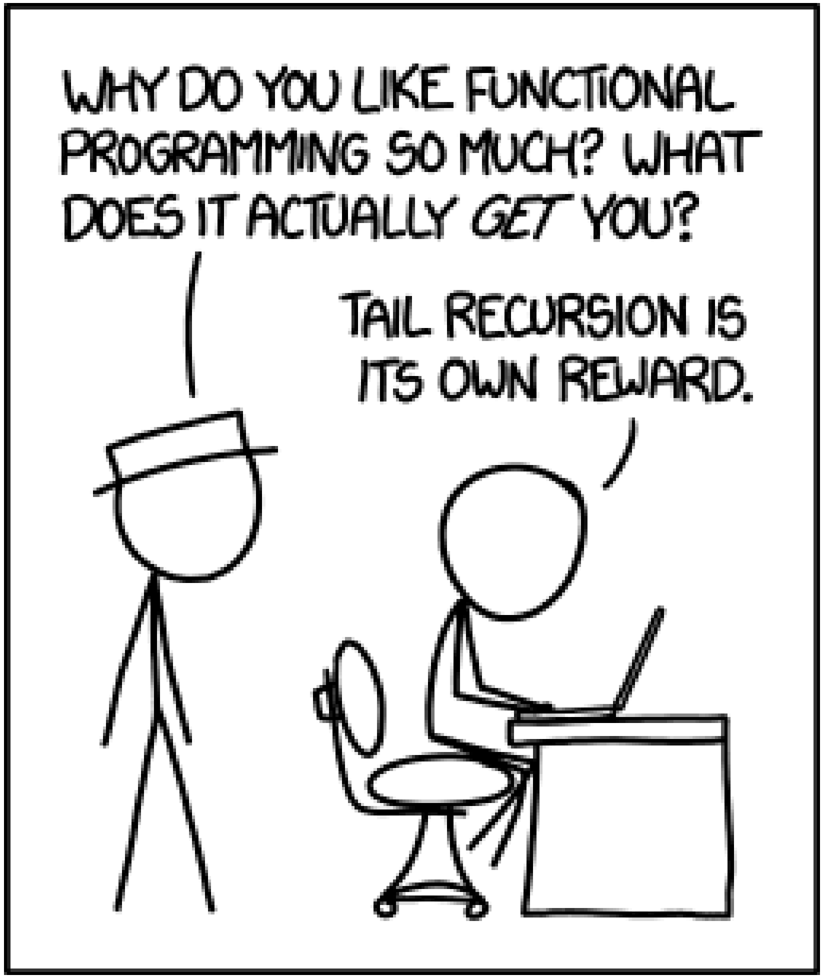
CSE 331
Summer 2025

Tail Recursion

But first, a bit more on mutable ADTs

Jaela Field



xkcd #1270 & Matt

8/8 Agenda

- **Finish** `MutableFastLastList and MutableNumberQueue` **examples**

  Mutable ADT (see Topic 7 slides)

- **New Topic (8): Tail Recursion**

  In less focus than a standard quarter. Additional materials posted if you're interested.

8/8 Agenda

- ✓ **Finish** `MutableFastLastList` and `MutableNumberQueue` **examples**
  - Mutable ADT
- **Tail Recursion**

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

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

Recursion can be directly translated into code

- Loop to calculate sum of a list

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

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

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

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

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

Sum of List: Recursion vs Loops, in Code

Loop

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

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

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

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

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

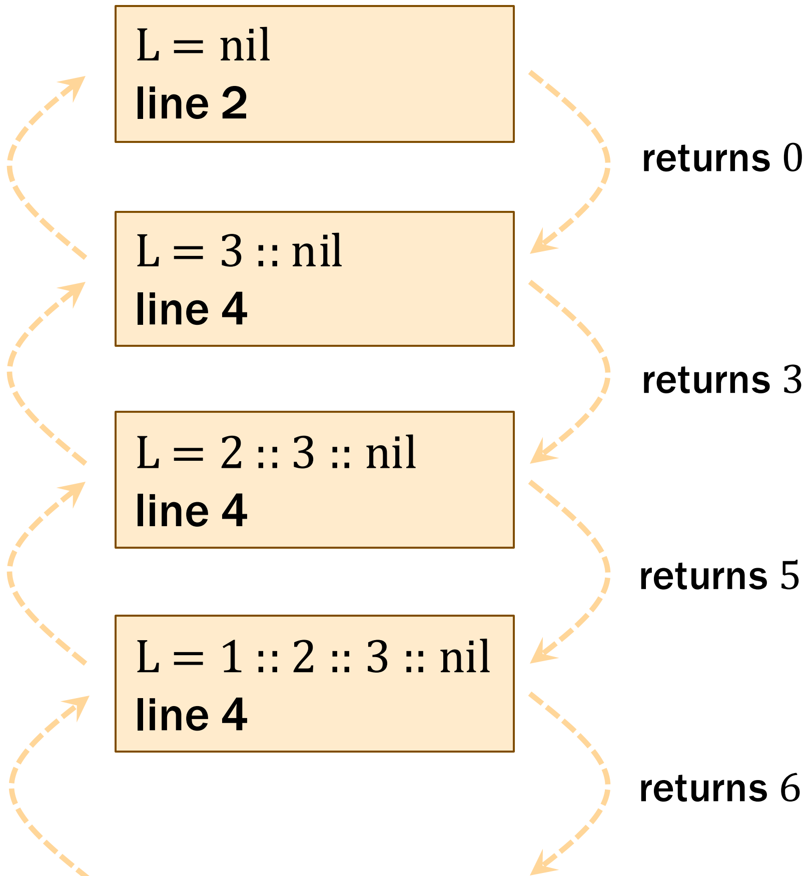
Recursion

```
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 $n = len(L)$

How much does space efficiency matter?

- In principle, this extra memory usually not a problem
  – $O(n)$ 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

- Another summation function

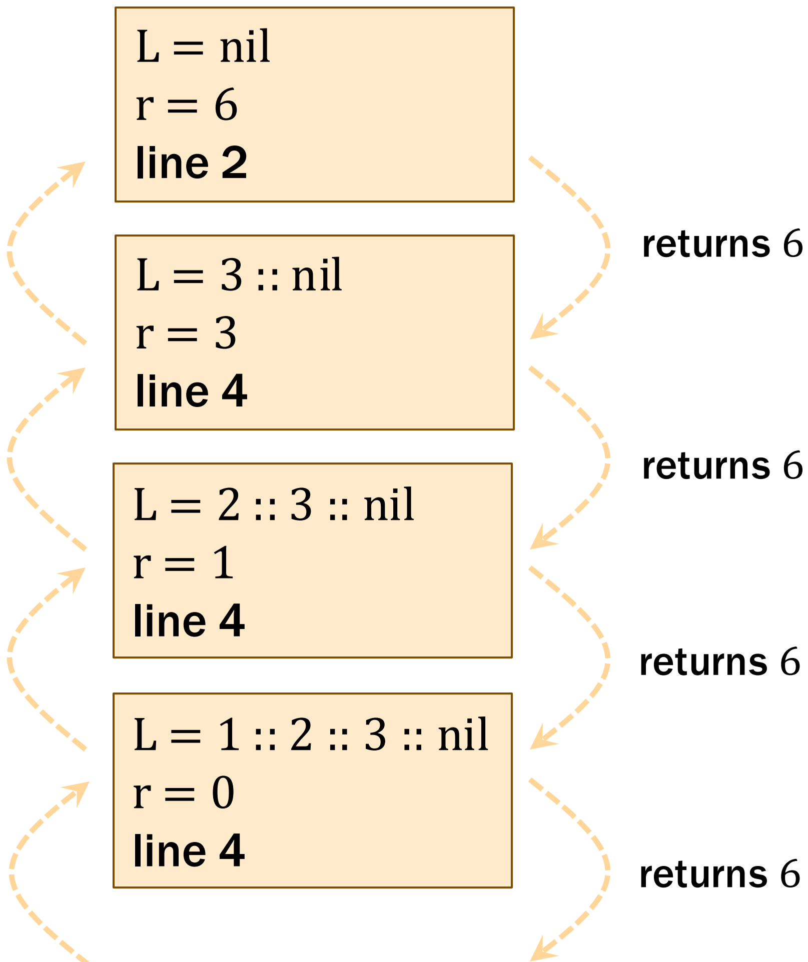
r is an "accumulator variable"

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

- 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



... `sum_acc(1 :: 2 :: 3 :: nil, 0)` ...

```
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  }
}
```

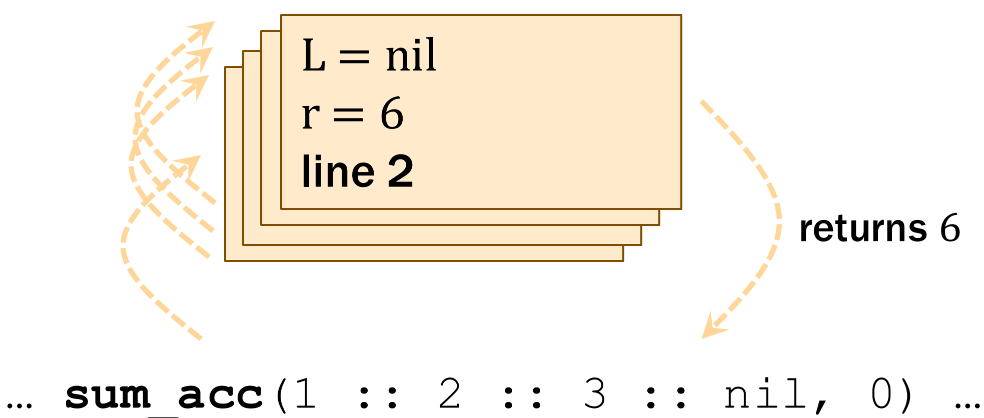
This is a "tail call" and "tail recursion".

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

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

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   }
}
```



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 <u>loop</u>

- 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…

Pause & Ponder: 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!

Pause & Ponder: 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);
}
```

Yes! The last thing we do is return!

Pause & Ponder: 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)
  );
}
```

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 loops 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: my-acc(R_0, s_0) = 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(R_0, s_0) = 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

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:**

sum-acc(nil, s)	:= $h(s)$	$h(s) \to s$
sum-acc(x :: L, s)	:= my-acc(L, $g(s, x)$)	$g(s, x) \to s + x$
sum-func(L)	:= my-acc(L, $f(L)$)	$f(L) \to 0$

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:**

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

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

Loops vs Tail Recursion in Math

- Tail recursion gives nicer notation for loop operation

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

Iteration	R	s
0	3 :: 4 :: 2 :: nil	1
1	4 :: 2 :: nil	4
2	2 :: nil	8
3	nil	10

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

sum-func(1 :: 3 :: 4 :: 2 :: nil)
= sum-acc(1 :: 3 :: 4 :: 2 :: nil, 0)   sum-func
= sum-acc(3 :: 4 :: 2 :: nil, 1)   sum-acc
...
= sum-acc(nil, 10)   sum-acc
= 10   sum-acc

- 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 loops use less memory than (non-tail) recursion

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

Key Takeaways

- **Ordinary loops are a special case of 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**

Zooming out on Loops & Recursion

- 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 List: Twice

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

- **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

This Twice Is (not) Right!

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

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

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

  - **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**

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?
  - pro: can use recursion reasoning while still writing loops
  - con: feels like … overkill?

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)

- Has to do with Associativity
  - Left-associative operations (start on the left, move right) lend themselves to tail recursion (loops)

    e.g. recursive-call(L) :: operation(x)
  - Right-associative operations (start on the right, move left) lend themselves to bottom-up recursion

    e.g. operation(x) :: recursive-call(L)

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"

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
    - ADTs!