CSE 331

Tail Recursion

James Wilcox and Kevin Zatloukal

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 the Values in a List

- 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 the Values in a List

Loop

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

```
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;
}
```

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

Recursion

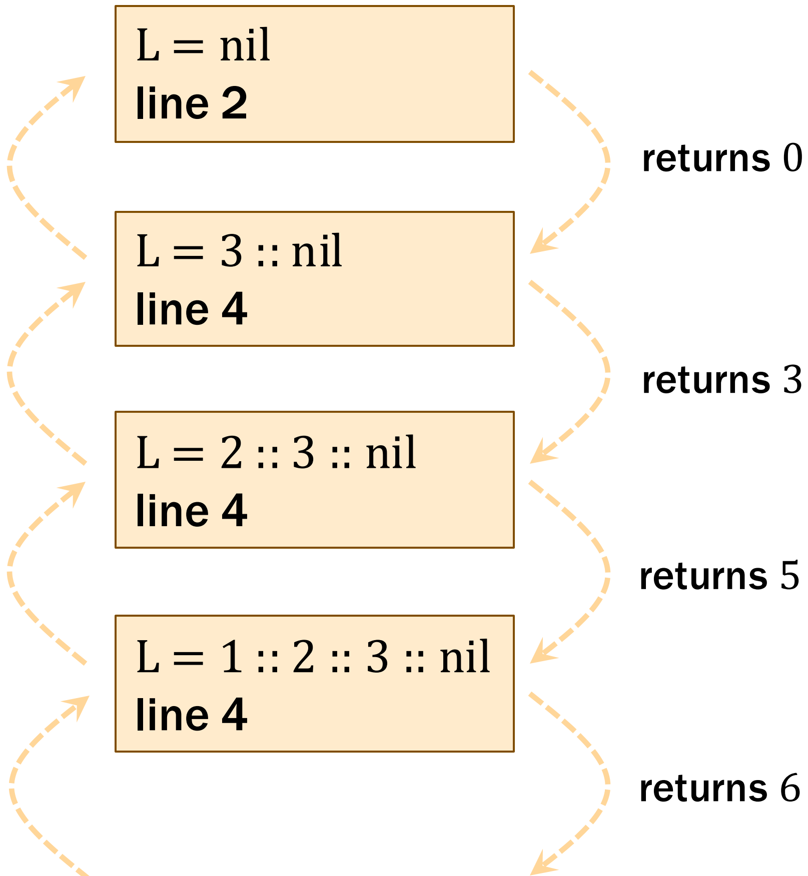
```
const sum = (L: List): bigint => {
  if (L.kind === "nil") {
    return 0n;
  } else {
    return L.hd + sum(L.tl);
  }
}
```

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

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

Recursive Version of Sum

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



… `sum(1 :: 2 :: 3 :: nil)` …

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 this 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, $\text{sum}(L)$ exceeds stack size at $\text{len}(L) = 10{,}000$

- Loops $\gg$ Recursion?

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

Another Sum of the Values in a List

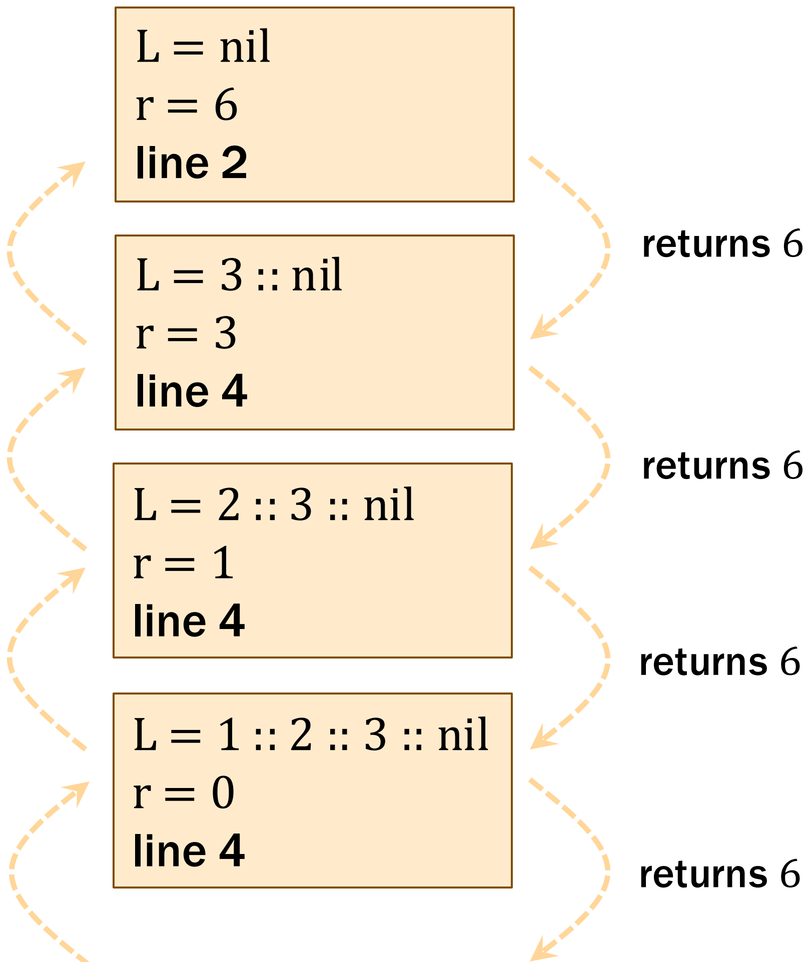
- Saw another summation function in Topic 5

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

- 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);
  }
}
```

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    }
}
```

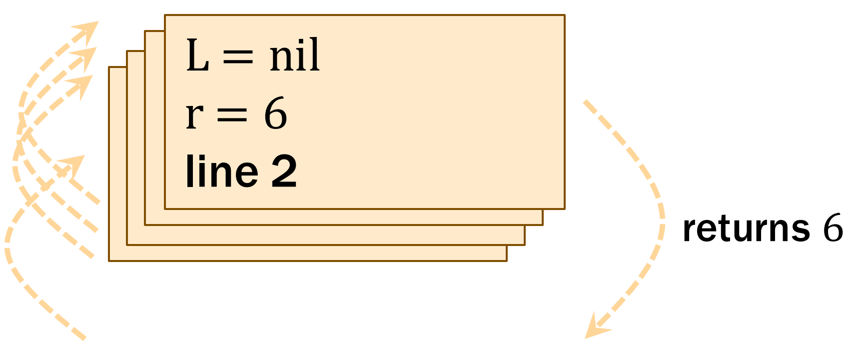
This is a "tail call" and "tail recursion".

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

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

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   }
}
```



… **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**

Loops vs Tail Recursion

- 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
- Chrome added tail-call optimization… then dropped it!
  - loops / tail-call optimization have downsides (more later…)
  - it no longer does this automatically

    you must manually convert to a loop if you require O(1) memory

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

```
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

Example 1: Loop to Tail Recursion

```
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:**

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

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

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

Example 1: Loop to Tail Recursion

```
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: Max Value in a List

```
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: Max Value in a List

```
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: Loop to Tail Recursion

```
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-acc(nil, s) := h(s)

max-acc(x :: L, s) := max-acc(L, g(s, x))

$h(s) \rightarrow s$

$g(s, x) \rightarrow x$ **if** $x > s$

s **if** $x \leq s$

max-func(L) := max-acc(L, f(L))

$f(L) \rightarrow L.hd$ **if** $L \neq nil$

Example 2: Loop to Tail Recursion

```
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 $x \leq s$

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

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

Example 2: Loop to Tail Recursion

```
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

- 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)

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** $2 \le 4$)

= 4

- 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

More Loops vs Tail Recursion

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

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

Example 2: Max Value in a List

```
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!

Example 2: Max Value in a List

```
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
how 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**

Ordinary Loops vs Tail Recursion

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

- Can solve <u>exactly</u> 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…

Recall: Faster Sum

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

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

- **Both versions are recursive and $O(n)$ time**
  - **<u>second</u> version is tail recursive**

- **Saw that $\text{sum-acc}(S, r) = \text{sum}(S) + r$**
  - **proved this by structural induction**
  - **tells us that $\text{sum}(S) = \text{sum-acc}(S, 0)$**

Tail Recursion to a Loop

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

- **Could implement** sum-acc **as recursively:**

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

```
r = S.hd + r;
S = S.tl;
```

  – now want to restart at the top with new values for `S` and `r`

Tail Recursion to a Loop

sum-acc(nil, r) := r

sum-acc(x :: L, r) := sum-acc(L, x + r)

- **Could implement** sum-acc **as recursively:**

```
const sum_acc = (S: List, r: bigint): bigint => {
  if (S.kind === "nil") {
    return r;
  } else {
    r = S.hd + r;
    S = S.tl;
    // go to top…
  }
};
```

Tail Recursion to a Loop

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

- **Could implement** sum-acc **as recursively:**

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

  - looks unusual with the return inside the loop...

Tail Recursion to a Loop

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

- **Could implement** sum-acc **as recursively:**

```
const sum_acc = (S: List, r: bigint): bigint => {
  while (S.kind !== "nil") {
    r = S.hd + r;
    S = S.tl;
  }
  return r;
};
```

  - **can be sure this is correct with Floyd Logic**

    but for that we need an **invariant**

Tail Recursion to a Loop

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

- **Could implement** sum-acc **as recursively:**

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

  – **clear that the invariant holds initially**

Tail Recursion to a Loop

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

- **Could implement** sum-acc **as recursively:**

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

$\text{sum-acc}(S_0, r_0)$
$= \text{sum-acc}(S, r)$
$= \text{sum-acc}(\text{nil}, r)$ — since $S = \text{nil}$
$= r$ — def of sum-acc

Tail Recursion to a Loop

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

- **Could implement** sum-acc **as recursively:**

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

Tail Recursion to a Loop

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

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

- **Could implement** sum-acc **as recursively:**

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

Tail Recursion to a Loop

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

- **Could implement** sum-acc **as recursively:**

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

$\text{sum-acc}(S_0, r_0)$
$= \text{sum-acc}(S, r)$
$= \text{sum-acc}(S.hd :: S.tl, r)$ — **since** $S = S.hd :: S.tl$
$= \text{sum-acc}(S.tl, S.hd + r)$ — **def of** sum-acc

Tail Recursion to a Loop

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

- 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

Tail Recursion to a Loop

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

base cases

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

recursive cases

- **Tail-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

```
// 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
- Can be rewritten to not mention sum-acc at all
  - use the relationship we proved between sum-acc and sum

Rewriting the Invariant

```
// 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 know the initial value of r

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

Rewriting the Invariant

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

- 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

Last Element

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

- **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…**

Last Element

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

- **Translate to a loop:**

```
// @param S a non-empty list
const last = (S: List) => bigint {
  // Inv: f(args0) = f(args)
  while (args /* match some recursive pattern */) {
    args = /* right-side of recursive pattern */;
  }
  return /* right-side of base case pattern */;
};
```

Last Element

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

recursive case

- **Translate to a loop:**

```
// @param S a non-empty list
const last = (S: List) => bigint {
  // Inv: last(S0) = last(S)
  while (args /* match some recursive pattern */) {
    args = /* right-side of recursive pattern */;
  }
  return /* right-side of base case pattern */;
};
```

Last Element

last(nil)	:= undefined	base cases
last(x :: nil)	:= x	
last(x :: y :: L)	:= last(y :: L)	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 */;
};
```

Last Element

last(nil) := undefined
last(x :: nil) := x
last(x :: y :: L) := last(y :: L)

base cases (first two lines); recursive case (third line)

- **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;
  }
  if (S.kind === "nil")
    throw new Error("no last element!");
  return S.hd;
};
```

Last Element

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

- *Mechanically* becomes the following 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;
  }
  if (S.kind === "nil")
    throw new Error("no last element!");
  return S.hd;
};
```

Definition of List Reversal

- **Look at some examples…**

L	rev(L)
nil	nil
3 :: nil	3 :: nil
2 :: 3 :: nil	3 :: 2 :: nil
1 :: 2 :: 3 :: nil	3 :: 2 :: 1 :: nil

- **Where does** rev([2, 3]) **show up in** rev([1, 2, 3])**?**
  - **at the beginning, with** 1 :: nil ***after*** **it**

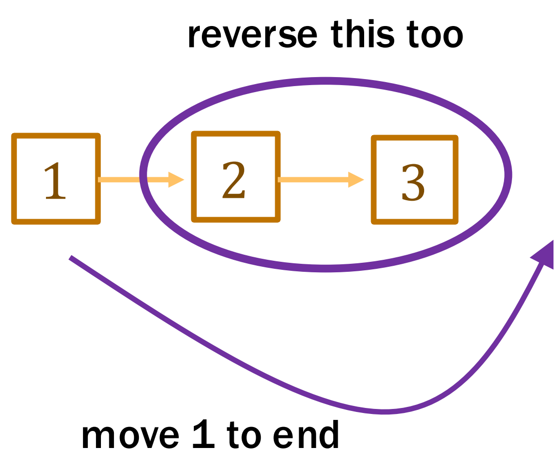
- **Where does** rev([3]) **show up in** rev([2, 3])**?**
  - **at the beginning, with** 2 :: nil ***after*** **it**

Reversing a List

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

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

  - **note that** rev **uses** concat $(\mathbin{+\!\!+})$ **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**

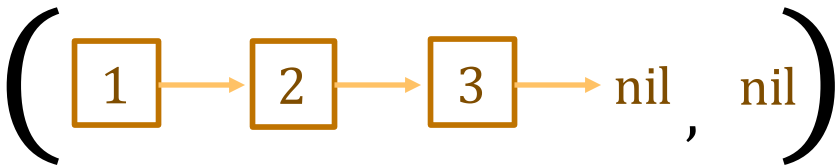
Reversing a List Quickly

- **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



Reversing a List Quickly

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

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

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

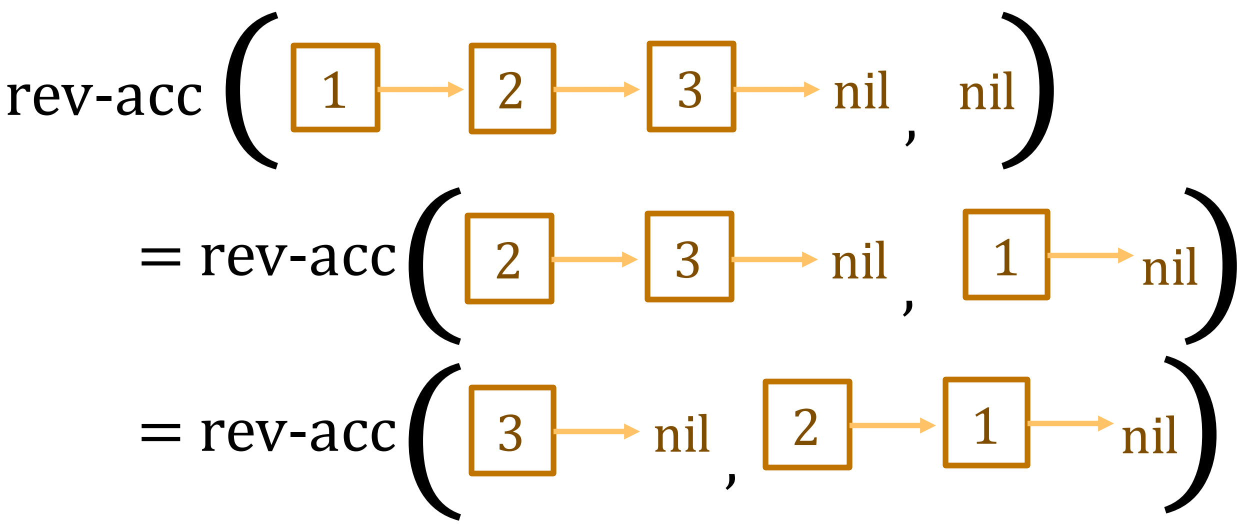
$$\text{rev-acc}\left(\boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3} \rightarrow \text{nil},\ \text{nil}\right)$$

$$= \text{rev-acc}\left(\boxed{2} \rightarrow \boxed{3} \rightarrow \text{nil},\ \boxed{1} \rightarrow \text{nil}\right)$$

Reversing a List Quickly

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

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

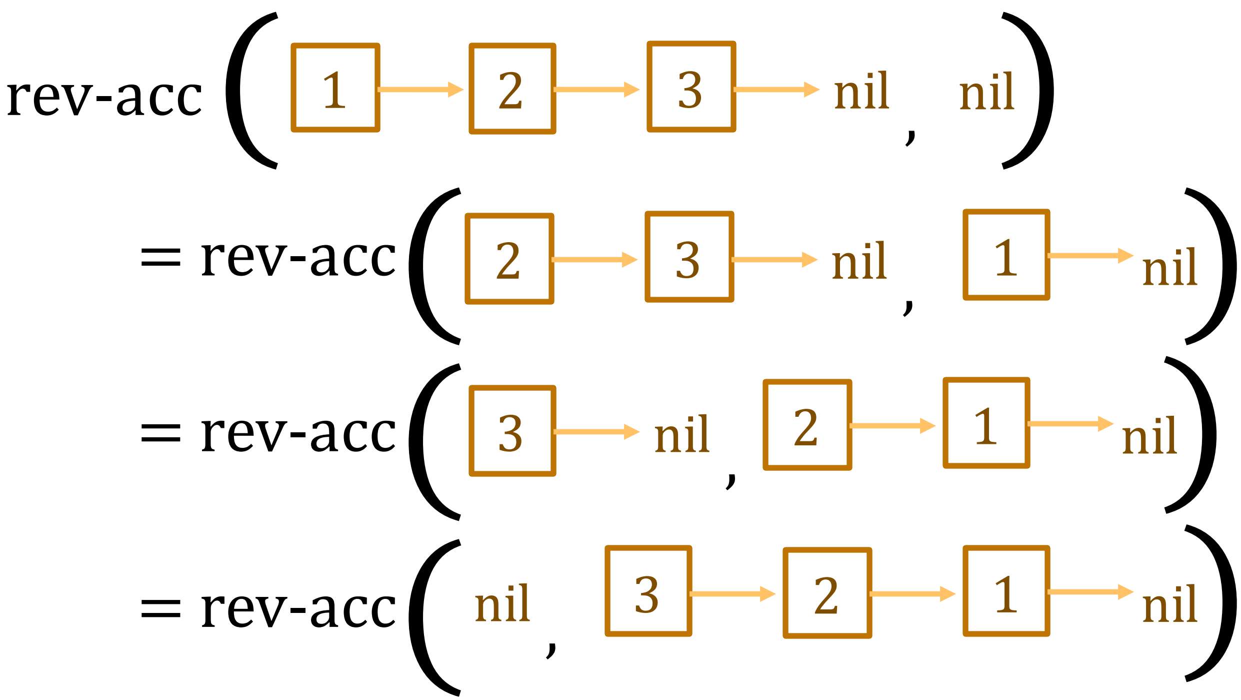


Reversing a List Quickly

- **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

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

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

- **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

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

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

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

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

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

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

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

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

Helper Lemma

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

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

Base Case (nil):

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

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

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

Helper Lemma

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

- **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):

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

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

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

Helper Lemma

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

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

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

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

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

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

$\text{concat}(\text{nil}, R) := R$	$\text{rev}(\text{nil}) := \text{nil}$
$\text{concat}(x :: L, R) := x :: \text{concat}(L, R)$	$\text{rev}(x :: L) := \text{rev}(L) + [x]$

Helper Lemma

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

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

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

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

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

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

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

Helper Lemma

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

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

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

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

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

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

Helper Lemma

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

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

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

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

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

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

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

Helper Lemma

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

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

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

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

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

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

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

Recall: Tail Recursion to a Loop

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

base cases

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

recursive cases

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

```
// Inv: f(args_0) = f(args)
while (args /* match some q pattern */) {
  args = /* right-side of appropriate q pattern */;
}
return /* right-side of appropriate p pattern */;
```

Loop Version of rev-acc

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

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

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** 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!

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

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** 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);
    R = R.tl;
  }
  return R;
}
```

We know $R_0 = []$

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

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(S0) = rev(S) ++ R
  while (S.kind !== "nil") {
    R = cons(S.hd, R);
    R = R.tl;
  }
  return R;
}
```

More On Loops vs 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

- **A lot more that could be said...**
  - **why did sum-acc and rev-acc work?**

    both use associative operations: + and ⧺
  - **many other cases where loops can be used**

    functions defined on natural numbers

    functions defined purely "bottom up" on lists

"Bottom Up" Functions on Lists

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

- **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

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

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 :: twice(2 :: 3 :: nil) **def of** twice
= 2 :: 4 :: twice(3 :: nil) **def of** twice
= 2 :: 4 :: 6 :: twice(nil) **def of** twice
= 2 :: 4 :: 6 :: nil **def of** twice

"Bottom Up" Functions on Lists

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

"Bottom Up" Functions on Lists

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

- **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 $\text{rev}(\text{twice-acc}(L, \text{nil}))$

  – **this lets us use a loop but it's not** $O(1)$ **memory**

More On Loops vs 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

- **A lot more that could be said…**
  - **why did sum-acc and rev-acc work?**

    both use associative operations: + and ⧺
  - **many other cases where loops can be used**

    functions defined on natural numbers

    functions defined purely "bottom up" on lists
  - **but we're out of time**