CSE 331

Loops & Recursion

Kevin Zatloukal

Administrivia

- Posted *updated* notes on testing
  - 0-1-many used for loops also
  - count number of times through the loop
- Posted notes on today's topic…

Checking Correctness with Loop Invariants

```
{{ P }}
{{ Inv: I }}
while (cond) {
  S
}
{{ Q }}
```

Formally, invariant split this into three Hoare triples:

1. {{ P }} {{ I }} — I **holds initially**
2. {{ I and cond }} **S** {{ I }} — **S preserves** I
3. {{ I and not cond }} {{ Q }} — Q **holds when loop exits**

Recall: Example Loop Correctness

- **Recursive function to calculate** $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : ℕ

- **This loop claims to calculate it as well**

```
{{ }}
let i: number = 0;
let s: number = 0;
{{ Inv: s = sum-to(i) }}
while (i != n) {
  i = i + 1;
  s = s + i;
}
{{ s = sum-to(n) }}
```

Example Loop Correctness

- **Recursive function to calculate** $1 + 2 + \dots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : ℕ

- **This loop claims to calculate it as well**

```
{{ }}
let i: number = 0;
let s: number = 0;
{{ i = 0 and s = 0 }}
{{ Inv: s = sum-to(i) }}
while (i != n) {
  ...
```

sum-to(i)
= sum-to(0)   **since** i = 0
= 0   **def of** sum-to
= s

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : ℕ

- This loop claims to calculate it as well

```
{{ Inv: s = sum-to(i) }}
while (i != n) {
  {{ s = sum-to(i) and i ≠ n }}
  i = i + 1;
  s = s + i;
  {{ s = sum-to(i) }}
}
```

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : ℕ

- This loop claims to calculate it as well

```
{{ Inv: s = sum-to(i) }}
while (i != n) {
  {{ s = sum-to(i) and i ≠ n }}
  i = i + 1;
  {{ s = sum-to(i–1) and i–1 ≠ n }}
  s = s + i;
  {{ s = sum-to(i) }}
}
```

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : ℕ

- This loop claims to calculate it as well

```
{{ Inv: s = sum-to(i) }}
while (i != n) {
  {{ s = sum-to(i) and i ≠ n }}
  i = i + 1;
  {{ s = sum-to(i–1) and i–1 ≠ n }}
  s = s + i;
  {{ s – i = sum-to(i–1) and i–1 ≠ n }}
  {{ s = sum-to(i) }}
}
```

s = i + sum-to(i-1)   **since** s – i = sum-to(i-1)
= sum-to(i)   **def of** sum-to

Example Loop Correctness

- **Recursive function to calculate $1 + 2 + \ldots + n$**

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : ℕ

- **This loop claims to calculate it as well**

{{ **Inv**: s = sum-to(i) }}

```
while (i != n) {
  i = i + 1;
  s = s + i;
}
```

{{ s = sum-to(i) and i = n }}
{{ s = sum-to(n) }}

sum-to(n)
= sum-to(i)   **since** i = n
= s   **since** s = sum-to(i)

Loops & Recursion

Loops and Recursion

- To check a loop, we need a loop invariant

- Where does this come from?
  - part of the algorithm idea / design

    see 421 for more discussion
  - Inv and the progress step **formalize** the algorithm idea

    most programmers can easily formalize an English description

    (very tricky loops are the exception to this)

- Today, we'll focus on converting *recursion* into a loop
  - HW6 will fit these patterns
  - (more loops later)

Example Loop Correctness

- Recursive function to calculate n^2 without multiplying

$$\begin{aligned} &\textbf{func}\ \text{square}(0) && := 0 \\ &\qquad\ \ \text{square}(n+1) && := \text{square}(n) + 2n + 1 \qquad\qquad \text{for any } n : \mathbb{N} \end{aligned}$$

- We already proved that this calculates n^2
  - we can implement it directly with recursion

- Let's try writing it with a loop instead…

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : ℕ

- Loop idea for calculating square(n):
  - calculate i = 0, 1, 2, …, n
  - keep track of square(i) in "s" as we go along

i =	0	1	2	…	n
s =	0	1	4	…	n^2

- Formalize that idea in the loop invariant

along with the fact that we make **progress** by advancing i to i+1 each step

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : ℕ

- **Loop implementation**

```
let i: number = 0;
let s: number = 0;
{{ Inv: s = square(i) }}
while (i != n) {
  s = s + i + i + 1;
  i = i + 1;
}
return s;
```

Loop invariant says how i and s relate
s holds square(i), whatever i

i starts at 0 and increases to n

Now we can check correctness…

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : ℕ

- **Loop implementation**

```
let i: number = 0;
let s: number = 0;
{{ Inv: s = square(i) }}
while (i != n) {
  s = s + i + i + 1;
  i = i + 1;
}
{{ s = square(i) and i = n }}
{{ s = square(n) }}
return s;
```

square(n)
= square(i) **since** i = n
= s **since** s = square(i)

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : ℕ

- **Loop implementation**

```
{{ }}
let i: number = 0;
let s: number = 0;
{{ i = 0 and s = 0 }}
{{ Inv: s = square(i) }}
while (i != n) {
  s = s + i + i + 1;
  i = i + 1;
}
return s;
```

square(i)
= square(0) **since** i = 0
= 0 **def of** square
= s **since** s = 0

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n : ℕ

- **Loop implementation**

```
{{ Inv: s = square(i) }}
while (i != n) {
  {{ s = square(i) and i ≠ n }}
  s = s + i + i + 1;
  {{ s = square(i+1) }}
  i = i + 1;
  {{ s = square(i) }}
}
return s;
```

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1   for any n : $\mathbb{N}$

- **Loop implementation**

{{ **Inv**: s = square(i) }}
```
while (i != n) {
  {{ s = square(i) and i ≠ n }}
  {{ s + 2i + 1 = square(i+1) }}
  s = s + i + i + 1;
  {{ s = square(i+1) }}
  i = i + 1;
  {{ s = square(i) }}
}
return s;
```

Example Loop Correctness

func square(0) := 0
square(n+1) := square(n) + 2n + 1   for any n : ℕ

- **Loop implementation**

{{ **Inv**: s = square(i) }}
```
while (i != n) {
```
{{ s = square(i) and i ≠ n }}
{{ s + 2i + 1 = square(i+1) }}
```
  s = s + i + i + 1;
```
{{ s = square(i+1) }}
```
  i = i + 1;
```
{{ s = square(i) }}
```
}
return s;
```

s + 2i + 1 = square(i) + 2i + 1   **since** s = square(i)
= square(i+1)   **def of** square

“Bottom Up” Loops on Natural Numbers

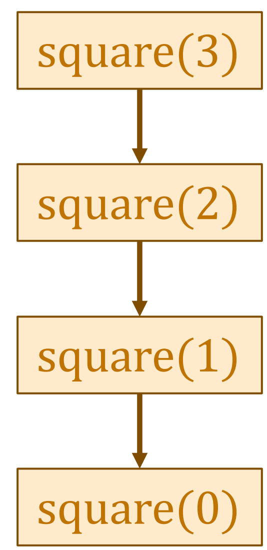
- Previous examples store function value in a variable

  {{ **Inv**: s = sum-to(i) }}

  {{ **Inv**: s = square(i) }}

- Start with $i = 0$ and work up to $i = n$

- Call this a “bottom up” implementation
  - evaluates in the same order as **recursion**
  - from the base case up to the full input



"Bottom Up" Loops on the Natural Numbers

func f(0) := ...
f(n+1) := ... f(n) ... for any n : $\mathbb{N}$

- Can be implemented with a loop like this

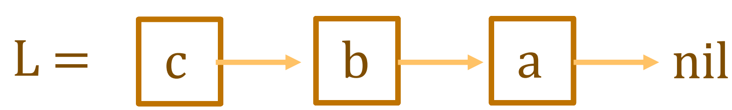
```
const f =(n: number): number => {
  let i: number = 0;
  let s: number = "...";    // = f(0)
  {{ Inv: s = f(i) }}
  while (i != n) {
    s = "... f(i) ..."[f(i) ↦ s]    // = f(i+1)
    i = i + 1;
  }
  return s;
};
```

“Bottom Up” Loops on Lists

- **Works nicely on** $\mathbb{N}$
  - **numbers are built up from** 0 **using** $\mathrm{succ}\ (+1)$
  - **e.g., build** $\mathrm{n} = 3$ up **from** 0

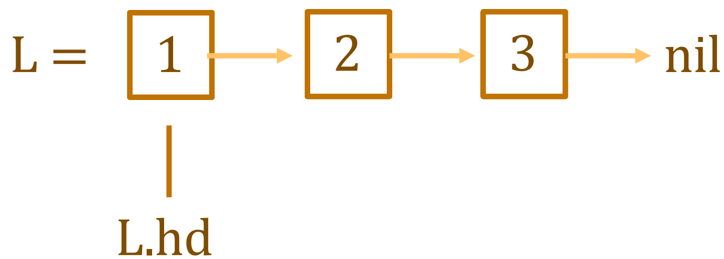
$$\mathrm{n} = \quad 3 \xleftarrow{+1} 2 \xleftarrow{+1} 1 \xleftarrow{+1} 0$$

- **What about** List**?**
  - **lists are built up from** nil **using** cons
  - **e.g., build** $\mathrm{L} = \mathrm{cons}(\mathrm{c}, \mathrm{cons}(\mathrm{b}, \mathrm{cons}(\mathrm{a}, \mathrm{nil})))$ **from** nil**:**



“Bottom Up” Loops on Lists?

- **What about** List**?**
  - **lists are built up from** nil **using** cons
  - **e.g., build** $L = \text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$ **from** nil**:**



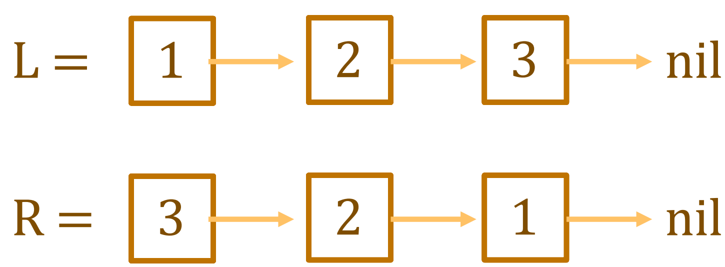
- **First step to build** L **is to build** $\text{cons}(3, \text{nil})$ **from** nil
  - **how do we know what number to put in front of** nil**?**

    3 is all the way at the end of the list!
  - **how can we fix this?**
  - **reverse the list!**

Example “Bottom Up” List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : ℤ and L : List

- **Loop** idea **for calculating** twice(L)**:**
  - **store** rev(L) **in** “R”

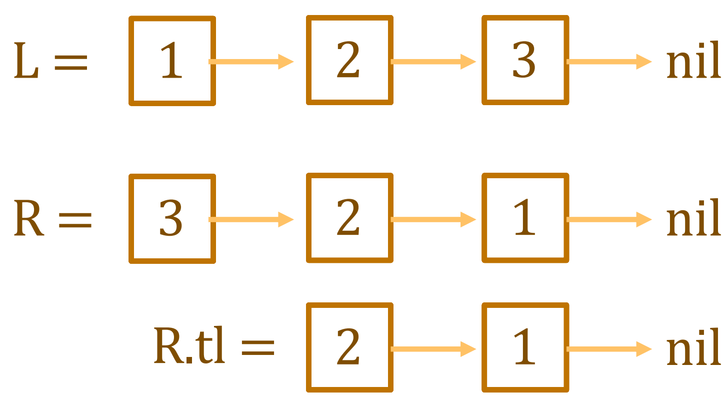


  - **watch what happens as we move** R **forward…**

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : $\mathbb{Z}$ and L : List

- Loop idea for calculating twice(L):
  - store rev(L) in "R"
  - moving forward in R is moving backward in L…

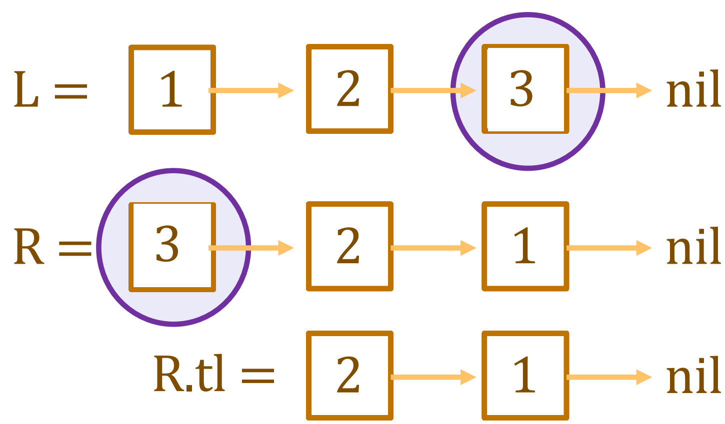


  - as R moves forward, rev(R) remains a **<u>prefix</u>** of L

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : $\mathbb{Z}$ and L : List

- **Loop** idea **for calculating** twice(L)**:**
  - **store** rev(L) **in** "R"
  - **moving forward in** R **is moving backward in** L**…**



  - **value dropped from** R **was** last(L) = 3

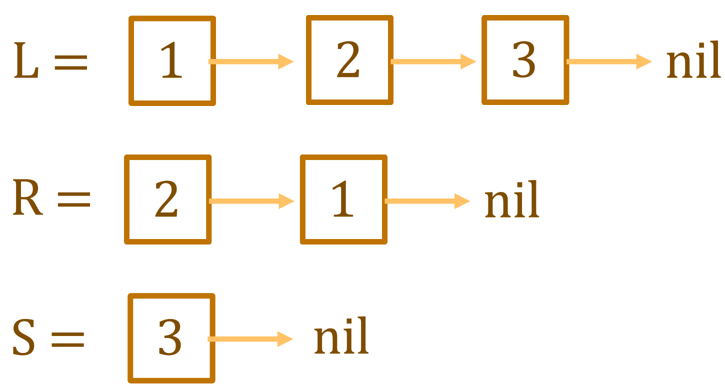
    can use it to build cons(3, nil)

Example “Bottom Up” List Loop

func twice(nil) := nil

twice(cons(x, L)) := cons(2x, twice(L))   for any x : $\mathbb{Z}$ and L : List

- **Loop** idea **for calculating** twice(L)**:**
  - **store** rev(L) **in** “R” **initially. move forward to** R.tl**, etc.**
  - **add items skipped over by** R **to the front of** “S”

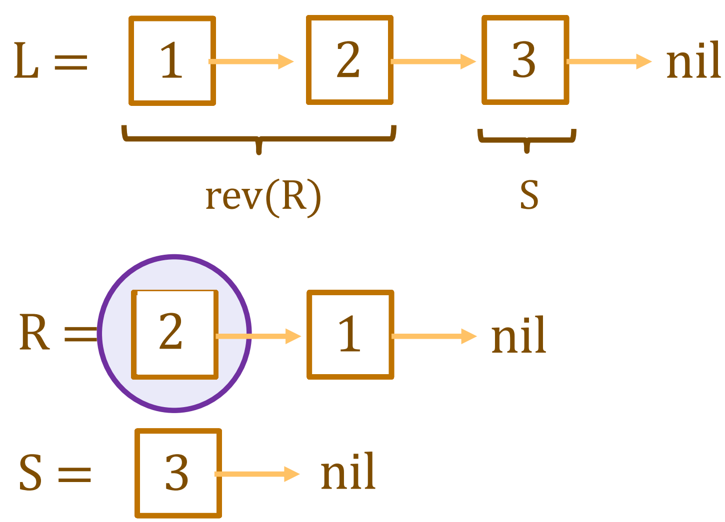


  - **as** R **moves forward,** S **stores a** **<u>suffix</u>** **of** L

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : ℤ and L : List

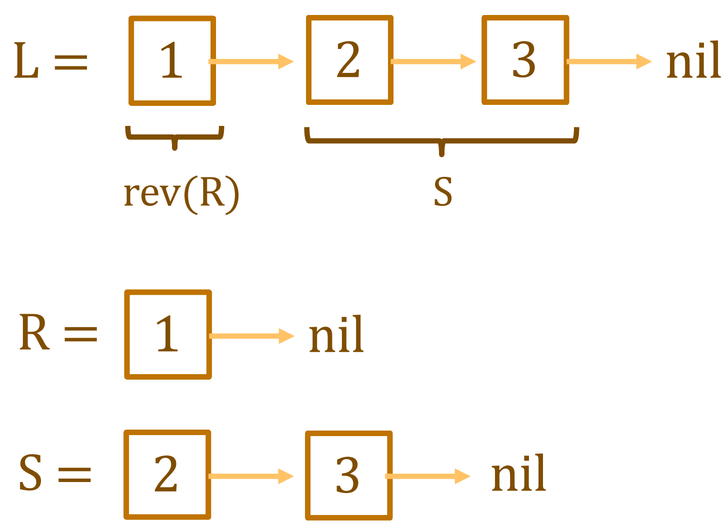
- **Loop** idea **for calculating** twice(L):
  - **store** rev(L) **in** "R" **initially. move forward to** R.tl, **etc.**
  - **add items skipped over by** R **to the front of** "S"



Example "Bottom Up" List Loop

$$
\begin{aligned}
&\textbf{func}\ \mathrm{twice}(\mathrm{nil}) && := \mathrm{nil} \\
&\quad\ \mathrm{twice}(\mathrm{cons}(x, L)) && := \mathrm{cons}(2x, \mathrm{twice}(L)) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\end{aligned}
$$

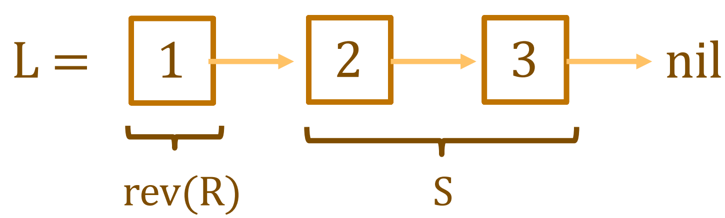
- **Loop** **idea** **for calculating** twice(L)**:**
  - **store** rev(L) **in** "R" **initially. move forward to** R.tl**, etc.**
  - **add items skipped over by** R **to the front of** "S"



Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : ℤ and L : List

- **Loop** idea **for calculating** twice(L)**:**
  - **store** rev(L) **in** "R" **initially. move forward to** R.tl**, etc.**
  - **add items skipped over by** R **to the front of** "S"



- **Formalize that idea in the loop invariant**

L = concat(rev(R), S)

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any $x : \mathbb{Z}$ and L : List

- **Loop** idea **for calculating** twice(L)**:**
  - **store** rev(L) **in** "R" **initially**
  - **add items skipped over by** R **to the front of** "S"
  - **advance by moving** R **forward (shrinking** R**, growing** S**)**
    S is built "bottom up" into the entire list L
  - **calculate** twice(S)**, as we go, in** "T"

- **Formalize that idea in the loop invariant**

$$L = \text{concat}(\text{rev}(R), S) \ \text{ and } \ T = \text{twice}(S)$$

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any $x : \mathbb{Z}$ and L : List

- This loop claims to calculate twice(L)...

```
let R: List = rev(L);
let S: List = nil;
let T: List = nil;
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R !== nil) {
  T = cons(2 * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
}
return T;  // = twice(L)
```

Still need to check this.

Hopefully obvious that it could be wrong.
(Testing length 0, 1, 2, 3 is not enough!)

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : ℤ and L : List

- **This loop claims to calculate** twice(L)

```
...
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R !== nil) {
  T = cons(2 * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
}
{{ L = concat(rev(R), S) and T = twice(S) and R = nil }}
{{ T = twice(L) }}
return T;  // = twice(L)
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any $x : \mathbb{Z}$ and L : List

- **Check that** Inv **is implies the postcondition:**

{{ L = concat(rev(R), S) and T = twice(S) **and R = nil** }}
{{ T = twice(L) }}

L = concat(rev(R), S)
= concat(rev(nil), S)   **since** R = nil
= concat(nil, S)   **def of** rev
= S   **def of** concat

T = twice(S)
= twice(L)   since L = S

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any $x : \mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

{{ }}

```
let R: List = rev(L);
let S: List = nil;
let T: List = nil;
```

{{ R = rev(L) and S = nil and T = nil }}

{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

```
while (R !== nil) {
  T = cons(2 * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : ℤ and L : List

- **Check that** Inv **is true initially:**

{{ R = rev(L) and S = nil and T = nil }}
{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

concat(rev(R), S)	
= concat(rev(rev(L)), S)	**since** R = rev(L)
= concat(L, S)	**Lemma 3**
= concat(L, nil)	**since** S = nil
= L	**Lemma 2**

twice(S)	
= twice(nil)	**since** S = nil
= nil	**def of** twice
= T	**since** T = nil

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any $x : \mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

```
while (R !== nil) {
  {{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}
  T = cons(2 * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
  {{ L = concat(rev(R), S) and T = twice(S) }}
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any $x : \mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

```
while (R !== nil) {
```

{{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}

```
  T = cons(2 * R.hd, T);
  S = cons(R.hd, S);
```

{{ L = concat(rev(R.tl), S) and T = twice(S) }}

```
  R = R.tl;
```

{{ L = concat(rev(R), S) and T = twice(S) }}

```
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : $\mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

```
while (R !== nil) {
```

{{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}

```
T = cons(2 * R.hd, T);
```

{{ L = concat(rev(R.tl), cons(R.hd, S)) and T = twice(S) }}

```
S = cons(R.hd, S);
```

{{ L = concat(rev(R.tl), S) and T = twice(S) }}

```
R = R.tl;
```

{{ L = concat(rev(R), S) and T = twice(S) }}

```
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any $x : \mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

{{ **Inv**: L = concat(rev(R), S) and T = twice(S) }}

```
while (R !== nil) {
```

{{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}

{{ L = concat(rev(R.tl), cons(R.hd, S)) and cons(2·R.hd, T) = twice(cons(R.hd, S)) }}

```
  T = cons(2 * R.hd, T);
```

{{ L = concat(rev(R.tl), cons(R.hd, S)) and T = twice(cons(R.hd, S)) }}

```
  S = cons(R.hd, S);
```

{{ L = concat(rev(R.tl), S) and T = twice(S) }}

```
  R = R.tl;
```

{{ L = concat(rev(R), S) and T = twice(S) }}

```
}
```

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any x : $\mathbb{Z}$ and L : List

- **Check that** Inv **is preserved by the loop body:**

{{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}
{{ L = concat(rev(R.tl), cons(R.hd, S)) and cons(2·R.hd, T) = twice(cons(R.hd, S)) }}

twice(cons(R.hd, S))
  = cons(2 R.hd, twice(S))   **def of** twice
  = cons(2 R.hd, T)   **since** T = twice(S)

Note that R ≠ nil **means** R = cons(R.hd, R.tl)

Example "Bottom Up" List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any $x : \mathbb{Z}$ and L : List

- **Check that Inv is preserved by the loop body:**

{{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}
{{ L = concat(rev(R.tl), cons(R.hd, S)) and cons(2·R.hd, T) = twice(cons(R.hd, S)) }}

L	= concat(rev(R), S)	
	= concat(rev(cons(R.hd, R.tl)), S)	**since** R ≠ nil
	= concat(concat(rev(R.tl), cons(R.hd, nil)), S)	**def of** rev
	= concat(rev(R.tl), concat(cons(R.hd, nil), S))	**Lemma 2**
	= concat(rev(R.tl), cons(R.hd, concat(nil, S)))	**def of** concat
	= concat(rev(R.tl), cons(R.hd, S))	**def of** concat

Example “Bottom Up” List Loop

func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))   for any $x : \mathbb{Z}$ and L : List

- **This loop claims to calculate** twice(L)

```
let R: List = rev(L);
let S: List = nil;
let T: List = nil;
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R !== nil) {
  T = cons(2 * R.hd, T);
  S = cons(R.hd, S);
  R = R.tl;
}
return T;  // = twice(L)
```

“S” is unused! We could remove it.

“S” is useful for proving correctness but it is not needed at run-time. (Example of a “*ghost*” variable.)

"Bottom Up" Loops on Lists

func f(nil) := ...
f(cons(x, L)) := ... f(L) ... for any x : ℤ and L : List

- **Can be implemented with a loop like this**

```
const f = (L: List): List => {
  let R: List = rev(L);
  let S: List = nil;
  let T: List = ...;    // = f(nil)
  {{ Inv: L = concat(rev(R), S) and T = f(S) }}
  while (R !== nil) {
    T = "... f(L) ..." [f(L) ↦ T]
    S = cons(R.hd, S);
    R = R.tl;
  }
  return T;  // = f(L)
};
```

Tail Recursion

$$\textbf{func } \text{twice}(\text{nil}) := \text{nil}$$
$$\text{twice}(\text{cons}(x, L)) := \text{cons}(2x, \text{twice}(L)) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}$$

- **To calculate** $\text{twice}(\text{cons}(x, L))$:
  - **recursively calculate** $S = \text{twice}(L)$
  - **when that returns, construct and return** $\text{cons}(2x, S)$

- **Not all functions require work *after* recursion:**

$$\textbf{func } \text{rev-acc}(\text{nil}, R) := R \quad \text{for any } R : \text{List}$$
$$\text{rev-acc}(\text{cons}(x, L), R) := \text{rev-acc}(L, \text{cons}(x, R)) \quad \text{for any } x : \mathbb{Z} \text{ and any } L, R : \text{List}$$

  - **such functions are called "tail recursive"**

"Top Down" List Loop

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

- **Tail recursion can be implemented top-down**
  - **no need to reverse the list**

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

Easy to see that Inv holds initially since $S = S_0$ and $R = R_0$

“Top Down” List Loop

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

- **Check that the postcondition holds upon exit:**

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

"Top Down" List Loop

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

- **Check that the postcondition holds upon exit:**

{{ rev-acc(S_0, R_0) = rev-acc(S, R) and S = nil }}
{{ R = rev-acc(S_0, R_0) }}

rev-acc(S_0, R_0)
= rev-acc(S, R)
= rev-acc(nil, R) — **since** S = nil
= R — **def of** rev-acc

"Top Down" List Loop

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

- **Check that** Inv **is preserved by the loop body:**

{{ Inv: rev-acc(S_0, R_0) = rev-acc(S, R) }}

```
while (S !== nil) {
```

{{ rev-acc(S_0, R_0) = rev-acc(S, R) and S ≠ nil }}

```
  R = cons(S.hd, R);
  S = S.tl;
```

{{ rev-acc(S_0, R_0) = rev-acc(S, R) }}

```
}
```

“Top Down” List Loop

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

- **Check that** Inv **is preserved by the loop body:**

{{ Inv: rev-acc(S_0, R_0) = rev-acc(S, R) }}

```
while (S !== nil) {
```

{{ rev-acc(S_0, R_0) = rev-acc(S, R) and S ≠ nil }}

```
R = cons(S.hd, R);
```

{{ rev-acc(S_0, R_0) = rev-acc(S.tl, R) }}

```
S = S.tl;
```

{{ rev-acc(S_0, R_0) = rev-acc(S, R) }}

```
}
```

“Top Down” List Loop

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

- **Check that** Inv **is preserved by the loop body:**

{{ Inv: rev-acc(S_0, R_0) = rev_acc(S, R) }}

```
while (S !== nil) {
```

{{ rev-acc(S_0, R_0) = rev-acc(S, R) and S ≠ nil }}

{{ rev-acc(S_0, R_0) = rev-acc(S.tl, cons(S.hd, R)) }}

```
R = cons(S.hd, R);
```

{{ rev-acc(S_0, R_0) = rev-acc(S.tl, R) }}

```
S = S.tl;
```

{{ rev-acc(S_0, R_0) = rev-acc(S, R) }}

```
}
```

“Top Down” List Loop

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

- **Check that** Inv **is preserved by the loop body:**

{{ rev-acc(S_0, R_0) = rev-acc(S, R) and S ≠ nil }}
{{ rev-acc(S_0, R_0) = rev-acc(S.tl, cons(S.hd, R)) }}

rev-acc(S.tl, cons(S.hd, R))
= rev-acc(cons(S.hd, S.tl), R) — **def of** rev-acc
= rev-acc(S, R) — **since** S ≠ nil
= rev-acc(S_0, R_0) — **since** rev-acc(S, R) = rev-acc(S_0, R_0)

Tail Recursion Elimination

- Most functional languages eliminate tail recursion
  - acts like a loop at run-time
  - true of JavaScript as well

- Alternatives for reducing space usage:
  1. Find a loop that implements it

     check correctness with Floyd logic
  2. Find an equivalent tail-recursive function

     check equivalence with structural induction