CSE 331
Software Design & Implementation

Winter 2025

Section 7 – Tail Recursion

Administrivia

- HW 7 written released tonight, due Wed. Feb 26

Loops vs Tail Recursion

- Tail-call optimization turns tail recursion into a loop

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 Loops ≪ Recursion
  - correspond to the *special* case of tail recursion

Loop to Tail Recursion

Translate loop to tail recursive helper function and main function:

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

my-func(R) := my-acc(R, f(R))

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

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

1. Loop body → recursive case of accumulator function
2. After loop body → base case of accumulator function
3. Before loop body → variable set up

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);
};
```

- **Final result:** tail-recursive function that does same calculation:

  my-func(R) := my-acc(R, **f**(R)) — Main func to call

  my-acc(nil, s) := **h**(s) — Helper accumulator func

  my-acc(x :: L, s) := my-acc(L, **g**(s, x))

Tail Recursion to Loop

f(… p_1 …, r)	:= …		
…			base cases
f(… p_n …, r)	:= …		
f(… q_1 …, r)	:= f(…)		
…			recursive cases
f(… q_n …, r)	:= f(…)		

- **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

Digit representations: List<Z>

Example, 120 in Base-10:

"Big endian": $1 :: 2 :: 0 :: \mathsf{nil}$

- higher order digits at the front

"Little endian": $0 :: 2 :: 1 :: \mathsf{nil}$ ← We're using this one

- higher order digits at the end

Defining value of a base-*b* digit as:

$$\begin{aligned} \mathsf{value}(\mathsf{nil}, b) &:= 0 \\ \mathsf{value}(d :: \mathsf{ds}, b) &:= d + b \cdot \mathsf{value}(\mathsf{ds}) \end{aligned}$$

Question 1

$$\text{value-acc}(\text{nil}, b, c, s) := s$$
$$\text{value-acc}(d :: \text{ds}, b, c, s) := \text{value-acc}(\text{ds}, b, b \cdot c, s + c \cdot d)$$

Write a function that calculates value-acc(digits, b, 1, 0) with a **loop**. Your function should have the following signature:

```
const value = (digits: List<number>, base: number): number => { ... };
```

Be sure to include the invariant of the loop!

Question 1

Write a function that calculates value-acc(digits, b, 1, 0) with a **loop**.

$$\text{value-acc}(\text{nil}, b, c, s) := s$$
$$\text{value-acc}(d :: \text{ds}, b, c, s) := \text{value-acc}(\text{ds}, b, b \cdot c, s + c \cdot d)$$

Question 2

Prove that value-acc(ds, b, c, s) = s + c * value(ds, b)

$$\begin{aligned} \text{value-acc}(\text{nil}, b, c, s) &:= s \\ \text{value-acc}(d :: \text{ds}, b, c, s) &:= \text{value-acc}(\text{ds}, b, b \cdot c, s + c \cdot d) \end{aligned}$$

$$\begin{aligned} \text{value}(\text{nil}, b) &:= 0 \\ \text{value}(d :: \text{ds}, b) &:= d + b \cdot \text{value}(\text{ds}) \end{aligned}$$

Question 2

Prove that value-acc(ds, b, c, s) = s + c * value(ds, b)

$$\begin{aligned} \text{value-acc}(\text{nil}, b, c, s) &:= s \\ \text{value-acc}(d :: \text{ds}, b, c, s) &:= \text{value-acc}(\text{ds}, b, b \cdot c, s + c \cdot d) \end{aligned}$$

$$\begin{aligned} \text{value}(\text{nil}, b) &:= 0 \\ \text{value}(d :: \text{ds}, b) &:= d + b \cdot \text{value}(\text{ds}) \end{aligned}$$

Question 3

Use equation value-acc(ds, b, c, s) = s + c * value(ds, b)
to rewrite the invariant so that it no longer mentions “value-acc”.

```
// Inv: value-acc(digits_0, base, 1, 0) = value-acc(digits, base, c, s)
```

Question 4a

Invariant: value(digits_0, base) = s + c * value(digits, base)

Prove that the invariant holds at the top of the loop

Question 4b

Invariant: value(digits_0, base) = s + c * value(digits, base)

Prove that, when we exit, the function returns value(digits_0, base)

Question 4c

Invariant: value(digits_0, base) = s + c * value(digits, base)

Prove that the invariant holds when we first get to the top of the loop.