CSE 331
Software Design & Implementation

Spring 2025

Section 7 – Tail Recursion

Administrivia

- HW 7 written released tonight, due Wed. May 20th

Postfix vs Infix Notation

When writing mathematical expressions, there are 2 different ways that we can express a math operation (ex: x * y + z)

Infix Notation - operators are placed between the operands they act upon

x * y + z

By order of operation, here we know to multiply x and y first, then add z to the product

Postfix Notation - operator comes after the operands

x y * z +

By order of operation, we take the first 2 operands (x and y) and apply the operator (*). We then take that product and the next operand (z) and apply the operator (+)

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

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

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 */;
```

Quick check in!

```
const fakehash = (L: List): bigint => {
  let s = 1;
  let r = 17;
  while (L.kind !== "nil") {
    r = r + s * L.hd;
     s = s*31;
    L = L.tl;
  }
  return r;
};
```

- What would the declaration of the accumulator function fakehash-acc look like?
  - What should the arguments of fakehash-acc be?
  - What does fakehash-acc return?

const fakehash-acc(**???**) : **???** => {...}

Quick check in!

```
const fakehash = (L: List): bigint => {
  let s = 1n;
  let r = 17n;
  while (L.kind !== "nil") {
    r = r + s * L.hd;
     s = s*31n;
    L = L.tl;
  }
  return r;
};
```

- L is the list from the original function fakehash
- s helps compute the value stored into r
- r is the return value

const fakehash-acc(**L: List**, **s: bigint**, **r: bigint**) : **bigint** => {...}

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:

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

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 => { ... };
```

What variables do we need to initialize within the function? What should those initial values be?

- c = 1
- s = 0

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 => { ... };
```

What is the base case? What should the while condition be?

- Base case: L.kind === “nil”
- while (L.kind !== “nil”)

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 => { ... };
```

Look at the method definition. Which variables are modified inside the loop? How are they modified?

- s = s + c * digits.hd
- c = base * c
- digits = digits.tl

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 => { ... };
```

Now put it all together! 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**.

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

```
const value = (digits: List<number>, base: number): number => {
    let c: number = 1;
    let s: number = 0;
    // Inv: value-acc(digits_0, base, 1, 0) = value-acc(digits,
base, c, s)
    while (digits.kind !== "nil") {
        s = s + c * digits.hd;
        c = base * c;
        digits = digits.tl;
    }
    return s;
};
```

Question 2

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

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

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

1) **Define:** Let P(ds) to be the claim that value-acc(ds, b, c, s) = s + c * value(ds, b) for all integers b, c, s. We will prove that this holds for all lists ds by structural induction
2) **Base Case:**

   value-acc(nil, b, c, s) = s   def of value-acc

   = s + c * 0

   = s + c * value(nil, b)   def of value
3) **IH:** Suppose P(xs) holds for some List<Z>, xs

Question 2

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

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

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

4) Inductive Step: Prove P(d::xs) holds

value-acc(d::xs, b, c, s)	= value-acc(xs, b, b*c, s+c * d)	def value-acc
	= s + c * d + b * c * value(xs, b)	IH
	= s + c(d + b * value(xs, b))	
	= s + c * value(d::xs, b)	def of value

5) Conclusion: P(ds) holds for all lists by Structural Induction

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

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)
```

We can see that

$\text{value}(\text{digits}_0, \text{base}) = 0 + 1 * \text{value}(\text{digits}_0, \text{base})$

$= \text{value-acc}(\text{digits}_0, \text{base}, 1, 0)$ by previous fact

$= \text{value-acc}(\text{digits}, \text{base}, c, s)$ Inv

$= s + c * \text{value}(\text{digits}, \text{base})$ by previous fact

So we get the invariant, $\text{value}(\text{digits}_0, \text{base}) = s + c * \text{value}(\text{digits}, \text{base})$

Question 4a: Back to Floyd Logic

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

Prove that the invariant holds at the top of the loop

```
const value = (digits: List<number>, base: number): number => {
    let c: number = 1;
    let s: number = 0;
    // Inv:value(digits₀, base) = s + c * value(digits, base)
    while (digits.kind !== "nil") {
        s = s + c * digits.hd;
        c = base * c;
        digits = digits.tl;
    }
    return s;
};
```

Question 4a

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

Prove that the invariant holds at the top of the loop

At the top, we see that s=0, c=1, and digits = $digits_0$

s + c * value(digits, base)	= c * value(digits, base)	since s = 0
	= value(digits, base)	since c = 1
	= value($digits_0$, base)	since digits = $digits_0$

Question 4b

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

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

```
const value = (digits: List<number>, base: number): number => {
    let c: number = 1;
    let s: number = 0;
    // Inv:value(digits₀, base) = s + c * value(digits, base)
    while (digits.kind !== "nil") {
        s = s + c * digits.hd;
        c = base * c;
        digits = digits.tl;
    }
    return s;
};
```

Question 4b

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

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

When we exit, we know the invariant is true and digits = nil. We can calculate:

$\text{value}(\text{digits}_0, \text{base})$	$= s + c * \text{value}(\text{digits}, \text{base})$	Inv
	$= s + c * \text{value}(\text{nil}, \text{base})$	since digits = nil
	$= s + c * 0$	def of value
	$= s$	

so we know returning s is correct

Question 4c

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

Prove that the invariant is preserved when we execute the loop body.

```
const value = (digits: List<number>, base: number): number => {
    let c: number = 1;
    let s: number = 0;
    // Inv: value(digits_0, base) = s + c * value(digits, base)
    while (digits.kind !== "nil") {
        s = s + c * digits.hd;
        c = base * c;
        digits = digits.tl;
    }
    return s;
};
```

Question 4c

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

Prove that the invariant is preserved when we execute the loop body.

Reasoning backwards through the loop, we get:

$value(digits_0, base) = (s + c * digits.hd) + (base * c) * value(digits.tl, base)$

We can prove that this follows from the invariant and the fact that digits != nil:

$s + c * digits.hd + base * c * value(digits.tl, base)$

$= s + c\ (digits.hd + base * value(digits.tl, base))$

$= s + c * value(digits.hd::digits.tl, base))$ def of value

$= s + c * value(digits, base)$ since digits != nil

$= value(digits_0, base)$ Inv

Attendance Form

https://tinyurl.com/sp331secBD7

Question 5

The following TypeScript function computes the factorial of a given number using a loop.

```
const factorial = (n: bigint): bigint => {
 let s = 1;
 while (n > 0) {
  s *= n;
  n--;
 }
 return s;
}
```

Define a mathematical definition for a tail-recursive function, factorial-acc, that has identical behavior to the loop body.

Question 5

The following TypeScript function computes the factorial of a given number using a loop.

```
const factorial = (n: bigint): bigint => {
 let s = 1;
 while (n > 0) {
  s *= n;
  n--;
 }
 return s;
}
```

What are the parameters to factorial-acc?

- n - from the original function factorial
- s - defined inside factorial

Question 5

The following TypeScript function computes the factorial of a given number using a loop.

```
const factorial = (n: bigint): bigint => {
 let s = 1;
 while (n > 0) {
  s *= n;
  n--;
 }
 return s;
}
```

What is the base case to factorial-acc?

- n === 0

Question 5

The following TypeScript function computes the factorial of a given number using a loop.

```
const factorial = (n: bigint): bigint => {
 let s = 1;
 while (n > 0) {
  s *= n;
  n--;
 }
 return s;
}
```

Now put it all together!

factorial-acc: (Z, Z) → Z
factorial-acc (0, s) := s
factorial-acc (n+1, s) := factorial-acc(n, s*(n+1))

Question 5

The following TypeScript function computes the factorial of a given number using a loop.

```
const factorial = (n: bigint): bigint => {
 let s = 1;
 while (n > 0) {
  s *= n;
  n--;
 }
 return s;
}
```

Redefine factorial to call factorial-acc:

factorial: $Z \rightarrow Z$

factorial(n) := factorial-acc(n, 1)