Recursion, Loops, Stacks, Tail Calls, and Space Safety
Space Complexity

\[
\text{(define} \ (\text{sum-to} \ n) \\
\text{(cond} \\
\quad [(\text{zero?} \ n) \ 0] \\
\quad [\text{else} \ (+ \ n \ (\text{sum-to} \ (\text{sub1} \ n)))]))
\]

\[
(\text{sum-to} \ 10) \to (+ \ 10 \ (\text{sum-to} \ 9)) \\
\quad \to (+ \ 10 \ (+ \ 9 \ (\text{sum-to} \ 8))) \\
\quad \to (+ \ 10 \ (+ \ 9 \ (+ \ 8 \ (\text{sum-to} \ 7))))
\]

\[(\text{sum-to} \ n) \text{ takes } O(n) \text{ space}\]
Space Complexity

\[
\text{(define \ (sum-to } n \ a) \\
\text{(cond} \\
\text{\quad [(zero? \ n) \ a]} \\
\text{\quad [else \ (sum-to (sub1 \ n) (+ \ n \ a)))])}
\]

\[
\text{(sum-to} \ 10 \ 0) \rightarrow \text{(sum-to} \ 9 \ 10) \\
\rightarrow \text{(sum-to} \ 8 \ 19) \\
\rightarrow \text{(sum-to} \ 7 \ 27)
\]

\[
\text{(sum-to} \ n \ 0) \text{ takes constant space}
\]

Actually, it’s O(log n), but we usually pretend that numbers are represented in constant space
Continuations

In

\[(\texttt{+ 10 (\texttt{+ 9 (\texttt{+ 8 (sum-to 7)}))})\]

the \textit{continuation} of \texttt{(sum-to 7)} is

\[(\texttt{+ 10 (\texttt{+ 9 (\texttt{+ 8 •)})})\]

That is, the \textit{continuation} of an expression is the work remaining after the expression is evaluated.

In particular, the \texttt{sum-to} with \(O(n)\) space complexity creates a continuation of \(O(n)\) size
Stacks

A *stack* is one way to represent a continuation

\[
(+ 10 \ ( + 9 \ ( + 8 \cdot ))) = \\
(+ 10 \cdot) \\
(+ 9 \cdot) \\
(+ 8 \cdot)
\]

Some language implementations use a fixed-size stack to represent continuations

In a high-level language, there is no good reason for this choice, and it creates problems in practice
Space and Local Bindings

(let ([val (make-big-pile-of-data)])
  (+ (f (collapse1 val))
    (g (collapse2 val)))

In this example, val must be retained during the call to f, because it is needed afterward
Space and Local Bindings

(let ([val (make-big-pile-of-data)])
  (+ (f (collapse1 val))
     (g 7))))

In this example, \texttt{val} should \textit{not} be retained during the call to \texttt{f}, because it is not needed by the time that \texttt{f} is called
Languages and Space Complexity

Languages sometimes go wrong in these ways:

• Limiting continuation size to $\ll$ available memory
  
  “stack” usually implies such a limit

• Extending a continuation needlessly
  
  “tail calls” should be handled properly

• Retaining data needlessly
  
  an implementation should be “safe for space”
Continuation Sizes Shouldn’t be Limited
; A list-of-num is either
;   - empty
;   - (cons num list-of-num

(define (sum lon)
 (cond
   [(empty? lon) 0]
   [else (+ (first lon)
         (sum (rest lon)))]))
Data Drives Design

interface IList {
    int sum();
}

class Empty implements IList {
    int sum() { return 0; }
}

class Cons implements IList {
    int sum() {
        return first + rest.sum();
    }
}
Data Drives Design

; A num-tree is either
;  - empty
;  - (node num num-tree num-tree)
(struct node (value left right))

(define (sum-tree t)
  (cond
    [(empty? t) 0]
    [else
      (+ (node-value t)
          (sum-tree (node-left t))
          (sum-tree (node-right t)))]))
Continuation-Limit Workarounds

With a limited continuation size, programmers must manage continuations themselves:

```java
int sumTree(Tree n) {
    int a;
    Stack s = new Stack();
    s.push(n);
    while (!s.isEmpty()) {
        n = s.pop();
        if (!n.isEmpty()) {
            a = a+n.getValue();
            s.push(n.getLeft());
            s.push(n.getRight());
        }
    }
    return a;
}
```
Proper Handling of Tail Calls
Tail Recursion

(define (sum-to n a)
  (cond
    [(zero? n) a]
    [else (sum-to (sub1 n) (+ n a))])))

The recursive call to \texttt{sum-to} is in \textit{tail position}

There’s no more work to do in \texttt{sum-to} after the recursive call
(define (sum-to n)
  (cond
    [(zero? n) 0]
    [else (+ n (sum-to (sub1 n)))]))

The recursive call to sum-to is **not** in tail position

There’s more work to do in sum-to after the recursive call
When Tail Recursion Matters

```
(define (run-server socket)
  (define-values (i o) (tcp-accept socket))
  (handle-connection i o)
  (run-sever socket))
```

The server shouldn’t leak memory as it handles connections
When Non-Tail Recursion Is Fine

(define (sum-list l)
  (cond
    [(empty? l) 0]
    [else (+ (first l)
              (sum-list (rest l)))]))

Uses $O(n)$ space for a list of length $n$ — but the list already uses $O(n)$ space
Tail Position

More precisely, *tail position* is relative and inductively defined:

- (lambda (arg ...) tail-expr)
  
i.e., *tail-expr* is in tail position w.r.t. the *lambda* form

- (begin expr ... tail-expr)

- (if expr tail-expr tail-expr)

- (cond [expr expr ... tail-expr] ...)

- (and expr ... tail-expr)

- (or expr ... tail-expr)
Tail Position

More precisely, *tail position* is relative and inductively defined:

- \((\text{lambda} \ (\text{arg} \ ... \ \text{tail-expr})\)\)
  
  i.e., *tail-expr* is in tail position w.r.t. the *lambda* form

- \((\text{begin} \ \text{expr} \ ... \ \text{tail-expr})\)

- \((\text{if} \ \text{expr} \ \text{tail-expr} \ \text{tail-expr})\)

*Proper tail-call handling*:

a function call that is in *tail position* in a function body

has the same continuation as the call to the function

It’s “proper” because it’s consistent with reduction as “ground truth”
Tail Calls

Tail calls need not be immediately recursive:

```
(define (is-even? n)
  (if (zero? n)
      #t
      (is-odd? (sub1 n))))

(define (is-odd? n)
  (if (zero? n)
      #f
      (is-even? (sub1 n))))
```
Tail Calls

Tail calls need not invoke a statically apparent target:

```scheme
(define (check-arg f)
  (lambda (n)
    (unless (number? n) (error "bad"))
    (f n)))

(define (is-even? n)
  (if (zero? n)
      #t
      ((check-arg is-odd?) (sub1 n))))
```
“Improper” Tail Call Handling

```c
int sumTo(int n, int a) {
    if (n == 0)
        return a;
    else
        return sumTo(n-1, n+a);
}
```

```c
sumTo(10, 0)
→ return sumTo(9, 10)
→ return return sumTo(8, 19)
→ return return return return sumTo(7, 27)
```

`sumTo(n, 0)` takes $O(n)$ space

which is bad, although it’s a less severe problem than a fixed-size stack
Tail-Call Workarounds

Languages without tail calls must provide additional syntactic support for tail recursion:

```c
int sumTo(int n) {
    int a = 0;
    while (n != 0) {
        a = a+n;
        n = n-1;
    }
    return a;
}
```
Interlude: Loop Patterns in Racket
Recursion Patterns

A **for** loop is a good pattern for many purposes:

\[
\text{(for/fold ([a 0]) ([i n]) (+ a i))}
\]

instead of

\[
\text{(let () (define (loop a i) (if (= i n) a (loop (+ a i) (- n 1))) (loop 0 0)))}
\]
Loop Variants

Imperative loops:

(for ([i seq])
  (do! i))

List creation:

(for/list ([i seq])
  (make-element i))

Any- and every-checks:

(for/and ([i seq])
  (ok? i))

(for/or ([i seq])
  (ok? i))

Accumulation:

(for/fold ([a 0]) ([i seq])
  (combine a i))
Space Safety
Space Complexity and Local Bindings

(define (f lon)
  (let ([lon2 (map add1 lon)])
    (+ (length lon2)
      (f (rest lon)))))

With C-like blocks:

(f (list .... n)) → (let ([lst2 (list .... n)]
                          (+ (length lst2)
                             (f (rest (list .... n)))))
                        → (let ([lst2 (list .... n)]
                                (+ n
                                   (f (rest (list .... n)))))
                            → (let ([lst2 (list .... n)]
                                    (+ n
                                       (let ([lst2 (list .... n-1)]
                                             (+ (length lst2)
                                               (f (rest (list .... n-1))))))))

Space complexity would be $O(n^2)$
Space Complexity and Local Bindings

\[
\begin{align*}
\text{(define } & (f \text{ lon)} \\
& (\text{let } ([\text{lon2 (map add1 lon)]}) \\
& (+ (\text{length lon2)} \\
& (f (\text{rest lon})))))
\end{align*}
\]

With substitution:

\[
\begin{align*}
(f (\text{list} \ldots n)) & \rightarrow (\text{let } ([\text{lst2 (list} \ldots n)]) \\
& (+ (\text{length lst2)}) \\
& (f (\text{rest (list} \ldots n))))
\rightarrow (+ (\text{length (list} \ldots n)) \\
& (f (\text{rest (list} \ldots n))))
\rightarrow (+ n \\
& (f (\text{rest (list} \ldots n))))
\rightarrow (+ n \\
& (\text{let } ([\text{lst2 (list} \ldots n-1)]) \\
& (+ (\text{length lst2)}) \\
& (f (\text{rest (list} \ldots n-1)))))))
\end{align*}
\]

Space complexity should be $O(n)$
Space Complexity and Local Bindings

```
(define (g lon)
  (+ (g (rest lon))
      (length lon)))
```

With simple substitution:

```
(g (list .... n)) → (+ (g (rest (list .... n)))
                      (length (list .... n)))
→ (+ (g (list .... n-1))
    (length (list .... n)))
→ (+ (+ (g (rest (list .... n-1)))
     (length (list .... n-1)))
    (length (list .... n)))
→ (+ (+ (g (list .... n-2))
     (length (list .... n-1)))
    (length (list .... n)))
```

Looks like $O(n^2)$, because the sharing of lists isn’t shown
Space Complexity and Local Bindings

\[(\text{define}\ (g\ \text{l}on))\]
\[+(g\ (\text{rest}\ \text{l}on))\]
\[(\text{length}\ \text{l}on))\]\n
With explicit allocation:

\[(g\ (\text{list} \ldots \ n)) \rightarrow (\text{begin})\]
\[(\text{define}\ \text{addr}_n\ (\text{cons}\ n\ \text{empty}))\]
\[(\text{define}\ \text{addr}_{n-1}\ (\text{cons}\ n-1\ \text{addr}_n))\]
\[\ldots\]
\[(g\ \text{addr}_1))\]
\[\rightarrow (\text{begin} \ldots\]
\[+(g\ (\text{rest}\ \text{addr}_1))\]
\[(\text{length}\ \text{addr}_1))\]
\[\rightarrow (\text{begin} \ldots\]
\[+(+(g\ (\text{rest}\ \text{addr}_2))\]
\[(\text{length}\ \text{addr}_2))\]
\[(\text{length}\ \text{addr}_1))\]

Overall size (including definitions) is \(O(n)\)
Space Safety

Reduction semantics with explicit allocation is “ground truth” for Racket

The compiler and run-time system are *safe for space*

i.e., consistent with ground truth, asymptotically
Space Safety and Language Extension

Space safety is particularly important in an extensible language:

```scheme
#lang lazy

(define (list-from n)
  (cons n (list-form (add1 n))))

(define (has-negative? l)
  (if (negative? (car l))
    #t
    (has-negative? (rest l))))

(has-negative? (list-from 0))
```

constant-space behavior depends on not retaining the head of the infinite list
Summary

Functional programming $\Rightarrow$ programming with algebra

- Proper tail-call handling and space safety enable reasoning about complexity via algebra

- Avoiding artificial resource constraints (such as stacks) make reasoning more uniform