Programming Paradigms

CT331 Week 8 Lecture 1

Finlay Smith

Finlay.smith@universityofgalway.ie

Recap

- **Functional Programming cont.**
  - (list …)
  - (append …)
  - (if expr expr expr)
  - (cond expr expr expr)
- **Recursion**
  - Well defined recursion:
    - Base Case
    - Recursive Case
  - Single vs mutual recursion.
  - General approach to recursive problem solving

Tail Recursion

Advantages of Recursion

- Clearer, simpler, shorter solutions that may be easier to understand
- Program directly reflects the algorithm
- Use with recursive data structures (such as Trees)
- Multiple activations of a function; efficiency considerations

Efficiency Considerations

- Overheads associated with a function call:
  - Space on call stack
  - Space for parameters and local variables
- Time to allocate and release local memory;
  - push and pop from stack

Efficiency Considerations

```
#lang racket

(define (fact num)
  (cond
    [ (= num 1) 1]
    [else (* num (fact (- num 1)))]
   ))
```

Need to keep space for 1 parameter: num

(fact 20) will activate 20 times.

=> (fact 20) will need space for 20 parameters

Efficiency Considerations

Problem:

Multiple activations of a function

Solution:

...

Efficiency Considerations

Problem:

Multiple activations of a function

Solution:

Try rewrite so that you don't have multiple activations of a function and thus can have more efficient solutions.

Tail Recursion

Value of recursive call provides the complete result of the original call. No waiting activations.

In Scheme: if last action of a function is another function call, then the new function call replaces the old one on the call stack. (So no stack growth)

Tail Recursive Factorial

```
(define (tail_fact num)
  (tfact num 1))

(define (tfact num total)
  (cond
    [(= 1 num) total]
    [else (tfact (- num 1) (* num total))]))
```

Helper function

Tail Recursive function.

Note: Helper function serves to initialize `total` to 1

Tail Recursive Factorial

Recursive version

```
#lang racket

(define (fact num)
  (cond
    [ (= num 1) 1]
    [else (* num (fact (- num 1)))]
  ))
```

Function is called with unknown values.

Ie. Cannot multiply until `fact` returns a value

Tail Recursive version

```
(define (tail_fact num)
  (tfact num 1))

(define (tfact num total)
  (cond
    [(= 1 num) total]
    [else (tfact (- num 1) (* num total))]))
```

Function is called with known values.

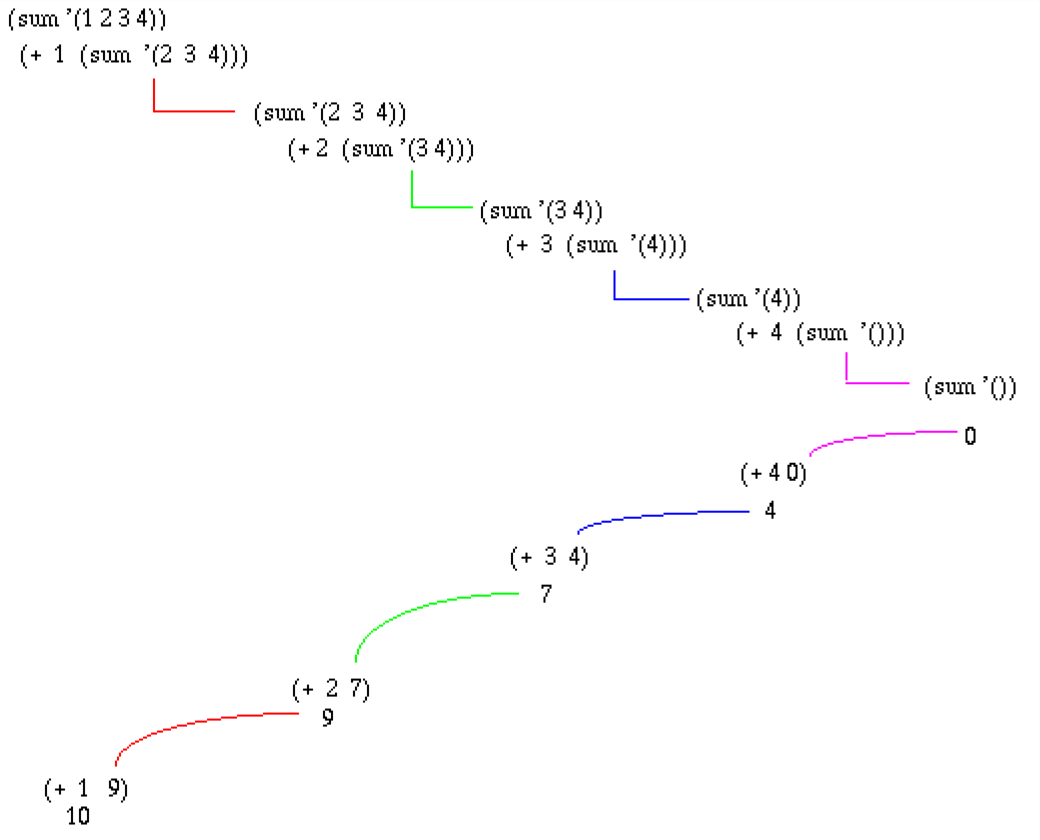
Ie. Is not waiting for any other functions to return.

Sum versus Running Sum

Recursive sum function, adds up the values in a list.

Each activation of the function waits for “deeper” activations to return before calculating.

Stack must hold all values in order to return sum.



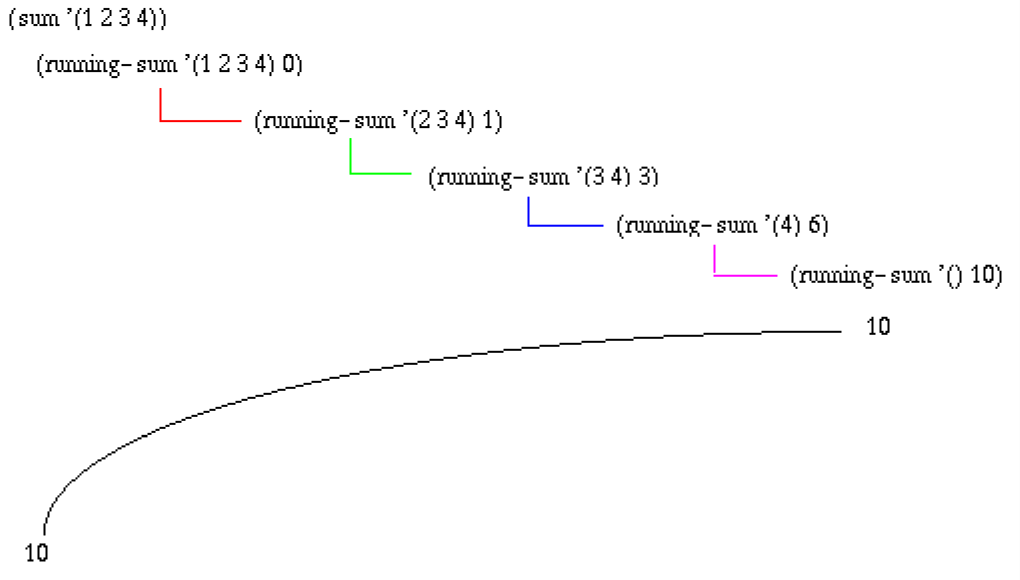
Sum versus Running Sum

Tail Recursive sum function, adds up the values in a list.

Each activation of the function adds `car of list` to `total`

No activations waiting on any other functions.

Stack does not need to hold any values.



Advantages and Disadvantages of Tail Recursion

- Tail recursive procedures only require enough memory space for one active invocation at a time. ✅
- Each invocation disappears upon calling the next.
  - Therefore, more space efficient than other kinds of recursive procedures. ✅
- Often more difficult to create and read than their non tail-recursive counterparts. ❌

Advantages and Disadvantages of Tail Recursion

- If infinite loops are required, then tail recursive procedures are excellent. ✓
- However, if not required, then recursion will not stop due to lack of memory as would happen with non tail-recursive procedures. ✗
- Unless you write a helper function, then have to remember the extra value(s) to pass to function. ✗

General Approach to solving problems recursively:

1. What is the base case?
2. What should the answer be when we are at the base case?
3. How do you reduce to get to this base case ? (often taking the cdr of a list)
4. What other work needs to be done for each function call?
    a. (e.g., creating a new list, etc.)?
5. How can these steps be put together?
6. Is this tail recursive?

Example problem

The built-in function reverse reverses the elements in a list. Write your own version of reverse: write both a tail and non-tail recursive function, e.g.,

```
(reverse_list '(a b c d))
 (d c b a)

(reverse_list '( (a b) c (d e f) g))
 (g (d e f) c (a b))
```

Non tail recursive

```
(define (reverse_list lst)
 (if (empty? lst)
     '()
     (append (reverse_list (cdr lst)) (list (car lst)))
     )
 )
```

Tail recursive

```
(define (rev_list lst)
  (rev_list_tr lst '())
  )

(define (rev_list_tr lst res)
  (if (empty? lst)
      res
      (rev_list_tr (cdr lst) (append (list (car lst)) res))
      ))
```