CSCI 2041: Tail Recursion and Activation Records

Chris Kauffman

Last Updated:
Fri Sep 21 14:31:59 CDT 2018

Logistics

Reading

- OCaml System Manual: 25.2 (Pervasives Modules)
- Practical OCaml: Ch 3, 9
- Wikipedia: Tail Call

Goals

- Activation Records
- Details of Recursion
- Tail Recursion Optimization

Assignment 1

- Due Wed 9/19 ~~Monday 9/17~~
- Note a few updates announced on Piazza / Changelog
- Questions?

Next Week

- Mon: Review
- Wed: Exam 1
- Fri: Lecture

Function Calls and Activation Records

- Will discuss part of how functions "work"
- Requires notion of where name/bindings are stored
- **Activation Records:** spots in memory where an executing function stores its bindings, **Frame** is slang for activation record
- Often Frames are on the **Function Call Stack**: grows linearly with each function call, last in, first out
  - OCaml uses a function call stack whenever possible as machine architecture is fast at executing stacks
  - Some uses of scopes and functions require something more complex than a function call stack which we may discuss later
- Understanding the function call stack Will elucidate how recursion works
- Allows specification of **tail call optimizations** that may be performed by the compiler

Demo of Function Calls 1

```
1  let dub_sqr x =
2    let sq = x*x in
3    2 * sq
4  ;;
5
6  let mult_ds x y =
7    let xx = dub_sqr x in
8    let yy = dub_sqr y in
9    xx * yy
10 ;;
11
12 let main () =
13   let z = 7 in
14   let w = 2 in
15   let mzw = mult_ds z w in
16   let dzw = dub_sqr z in
17   printf "z: %d  w: %d\n" z w;
18   printf "mzw: %d\n" mzw;
19   printf "dzw: %d\n" dzw;
20 ;;
21
22 main ();;
```

```
Active Functions
| FRAME     | SYMBOL    | VALUE |
|-----------+-----------+-------|
| init      | dub_sqr   | <fun> |
| line:22   | mult_ds   | <fun> |
|           | main      | <fun> |
|-----------+-----------+-------|

main() is called
execute some lines

| FRAME     | SYMBOL    | VALUE |
|-----------+-----------+-------|
| init      | dub_sqr   | <fun> |
| line:22   | mult_ds   | <fun> |
|           | main      | <fun> |
|-----------+-----------+-------|
| main      | z         | 7     |
| line:15   | w         | 2     |
|           | mzw       | ?     |
|           | dzw       | ?     |
|-----------+-----------+-------|
```

Demo of Function Calls 2

```
 1  let dub_sqr x =
 2    let sq = x*x in
 3    2 * sq
 4  ;;
 5
 6  let mult_ds x y =
 7    let xx = dub_sqr x in
 8    let yy = dub_sqr y in
 9    xx * yy
10  ;;
11
12  let main () =
13    let z = 7 in
14    let w = 2 in
15    let mzw = mult_ds z w in
16    let dzw = dub_sqr z in
17    printf "z: %d  w: %d\n" z w;
18    printf "mzw: %d\n" mzw;
19    printf "dzw: %d\n" dzw;
20  ;;
21
22  main ();;
```

main's mzw defined by result from mult_ds call: additional frame

```
| FRAME     | SYMBOL    | VALUE |
|-----------+-----------+-------|
| init      | dub_sqr   | <fun> |
| line:22   | mult_ds   | <fun> |
|           | main      | <fun> |
|-----------+-----------+-------|
| main      | z         | 7     |
| line:15   | w         | 2     |
|           | mzw       | ?     |<-+
|           | dzw       | ?     |  |
|-----------+-----------+-------|  |
| mult_ds   | x         | 7     |--+
| line:7    | y         | 2     |
|           | xx        | ?     |
|           | yy        | ?     |
|-----------+-----------+-------|
```

Demo of Function Calls 3

```
1   let dub_sqr x =
2     let sq = x*x in
3     2 * sq
4   ;;
5
6   let mult_ds x y =
7     let xx = dub_sqr x in
8     let yy = dub_sqr y in
9     xx * yy
10  ;;
11
12  let main () =
13    let z = 7 in
14    let w = 2 in
15    let mzw = mult_ds z w in
16    let dzw = dub_sqr z in
17    printf "z: %d  w: %d\n" z w;
18    printf "mzw: %d\n" mzw;
19    printf "dzw: %d\n" dzw;
20  ;;
21
22  main ();;
```

```
mult_ds's xx defined by call to
dub_sqr: additional frame

| FRAME     | SYMBOL    | VALUE  |
|-----------+-----------+--------|
| init      | dub_sqr   | <fun>  |
| line:22   | mult_ds   | <fun>  |
|           | main      | <fun>  |
|-----------+-----------+--------|
| main      | z         | 7      |
| line:15   | w         | 2      |
|           | mzw       | ?      |<-+
|           | dzw       | ?      |  |
|-----------+-----------+--------|  |
| mult_ds   | x         | 7      |--+
| line:7    | y         | 2      |
|           | xx        | ?      |<-+
|           | yy        | ?      |  |
|-----------+-----------+--------|  |
| dub_sq    | x         | 7      |--+
| line:1    | sq        | ?      |
|-----------+-----------+--------|
```

Demo of Function Calls 4

```
 1  let dub_sqr x =
 2    let sq = x*x in
 3    2 * sq
 4  ;;
 5
 6  let mult_ds x y =
 7    let xx = dub_sqr x in
 8    let yy = dub_sqr y in
 9    xx * yy
10  ;;
11
12  let main () =
13    let z = 7 in
14    let w = 2 in
15    let mzw = mult_ds z w in
16    let dzw = dub_sqr z in
17    printf "z: %d  w: %d\n" z w;
18    printf "mzw: %d\n" mzw;
19    printf "dzw: %d\n" dzw;
20  ;;
21
22  main ();;
```

dub_sqr completes, returns value up a frame to mult_ds

```
| FRAME    | SYMBOL   | VALUE |
|----------+----------+-------|
| init     | dub_sqr  | <fun> |
| line:22  | mult_ds  | <fun> |
|          | main     | <fun> |
|----------+----------+-------|
| main     | z        | 7     |
| line:15  | w        | 2     |
|          | mzw      | ?     |<-+
|          | dzw      | ?     |  |
|----------+----------+-------|  |
| mult_ds  | x        | 7     |--+
| line:7   | y        | 2     |
|          | xx       | ?     |<-+
|          | yy       | ?     |  |
|----------+----------+-------|  |98
| dub_sq   | x        | 7     |--+
| line:3   | sq       | 49    |
|----------+----------+-------|
```

Exercise: Demo of Function Calls 5

```
1  let dub_sqr x =
2    let sq = x*x in
3    2 * sq
4  ;;
5
6  let mult_ds x y =
7    let xx = dub_sqr x in
8    let yy = dub_sqr y in
9    xx * yy
10 ;;
11
12 let main () =
13   let z = 7 in
14   let w = 2 in
15   let mzw = mult_ds z w in
16   let dzw = dub_sqr z in
17   printf "z: %d  w: %d\n" z w;
18   printf "mzw: %d\n" mzw;
19   printf "dzw: %d\n" dzw;
20 ;;
21
22 main ();;
```

```
after returning, frame for dub_sq
pops off function call stack,
answer stored in xx

| FRAME    | SYMBOL  | VALUE |
|----------+---------+-------|
| init     | dub_sqr | <fun> |
| line:22  | mult_ds | <fun> |
|          | main    | <fun> |
|----------+---------+-------|
| main     | z       | 7     |
| line:15  | w       | 2     |
|          | mzw     | ?     |<-+
|          | dzw     | ?     |  |
|----------+---------+-------|  |
| mult_ds  | x       | 7     |--+
| line:8   | y       | 2     |
|          | xx      | 98    |
|          | yy      | ?     |
|----------+---------+-------|
```

Show the call stack on next reaching line 3 (in dub_sq)

Answers: Demo of Function Calls 6

```
1  let dub_sqr x =
2    let sq = x*x in
3    2 * sq
4  ;;
5
6  let mult_ds x y =
7    let xx = dub_sqr x in
8    let yy = dub_sqr y in
9    xx * yy
10 ;;
11
12 let main () =
13   let z = 7 in
14   let w = 2 in
15   let mzw = mult_ds z w in
16   let dzw = dub_sqr z in
17   printf "z: %d  w: %d\n" z w;
18   printf "mzw: %d\n" mzw;
19   printf "dzw: %d\n" dzw;
20 ;;
21
22 main ();;
```

dub_sq: x param is 2 this time
return 8 to frame above

```
| FRAME    | SYMBOL   | VALUE |
|----------+----------+-------|
| init     | dub_sqr  | <fun> |
| line:22  | mult_ds  | <fun> |
|          | main     | <fun> |
|----------+----------+-------|
| main     | z        | 7     |
| line:15  | w        | 2     |
|          | mzw      | ?     |<-+
|          | dzw      | ?     |  |
|----------+----------+-------|  |
| mult_ds  | x        | 7     |--+
| line:8   | y        | 2     |
|          | xx       | 98    |
|          | yy       | ?     |<-+
|----------+----------+-------|  |8
| dub_sq   | x        | 2     |--+
| line:3   | sq       | 4     |
|----------+----------+-------|
```

Demo of Function Calls 7

```
1   let dub_sqr x =
2     let sq = x*x in
3     2 * sq
4   ;;
5
6   let mult_ds x y =
7     let xx = dub_sqr x in
8     let yy = dub_sqr y in
9     xx * yy
10  ;;
11
12  let main () =
13    let z = 7 in
14    let w = 2 in
15    let mzw = mult_ds z w in
16    let dzw = dub_sqr z in
17    printf "z: %d  w: %d\n" z w;
18    printf "mzw: %d\n" mzw;
19    printf "dzw: %d\n" dzw;
20  ;;
21
22  main ();;
```

```
answers stored in mult_ds yy
mult_ds now ready to return

| FRAME    | SYMBOL   | VALUE |
|----------+----------+-------|
| init     | dub_sqr  | <fun> |
| line:22  | mult_ds  | <fun> |
|          | main     | <fun> |
|----------+----------+-------|
| main     | z        | 7     |
| line:15  | w        | 2     |
|          | mzw      | ?     |<-+
|          | dzw      | ?     |  |
|----------+----------+-------|  |784
| mult_ds  | x        | 7     |--+
| line:9   | y        | 2     |
|          | xx       | 98    |
|          | yy       | 8     |
|----------+----------+-------|
```

Exercise: Demo of Function Calls 8

```
1   let dub_sqr x =
2     let sq = x*x in
3     2 * sq
4   ;;
5
6   let mult_ds x y =
7     let xx = dub_sqr x in
8     let yy = dub_sqr y in
9     xx * yy
10  ;;
11
12  let main () =
13    let z = 7 in
14    let w = 2 in
15    let mzw = mult_ds z w in
16    let dzw = dub_sqr z in
17    printf "z: %d  w: %d\n" z w;
18    printf "mzw: %d\n" mzw;
19    printf "dzw: %d\n" dzw;
20  ;;
21
22  main ();;
```

```
mult_ds frame pops off stack

| FRAME    | SYMBOL   | VALUE |
|----------+----------+-------|
| init     | dub_sqr  | <fun> |
| line:22  | mult_ds  | <fun> |
|          | main     | <fun> |
|----------+----------+-------|
| main     | z        | 7     |
| line:16  | w        | 2     |
|          | mzw      | 784   |
|          | dzw      | ?     |
|----------+----------+-------|
```

What happens next?

How does the value for dzw get determined?

Answers: Demo of Function Calls 9

```
 1  let dub_sqr x =
 2    let sq = x*x in
 3    2 * sq
 4  ;;
 5
 6  let mult_ds x y =
 7    let xx = dub_sqr x in
 8    let yy = dub_sqr y in
 9    xx * yy
10  ;;
11
12  let main () =
13    let z = 7 in
14    let w = 2 in
15    let mzw = mult_ds z w in
16    let dzw = dub_sqr z in
17    printf "z: %d  w: %d\n" z w;
18    printf "mzw: %d\n" mzw;
19    printf "dzw: %d\n" dzw;
20  ;;
21
22  main ();;
```

```
dub_sq called with param 7
returns 98 to frame above

| FRAME    | SYMBOL   | VALUE |
|----------+----------+-------|
| init     | dub_sqr  | <fun> |
| line:22  | mult_ds  | <fun> |
|          | main     | <fun> |
|----------+----------+-------|
| main     | z        | 7     |
| line:16  | w        | 2     |
|          | mzw      | 784   |
|          | dzw      | ?     |<-+
|----------+----------+-------|  |98
| dub_sq   | x        | 7     |--+
| line:3   | sq       | 49    |
|----------+----------+-------|
```

Demo of Function Calls 10

```
1  let dub_sqr x =
2    let sq = x*x in
3    2 * sq
4  ;;
5
6  let mult_ds x y =
7    let xx = dub_sqr x in
8    let yy = dub_sqr y in
9    xx * yy
10 ;;
11
12 let main () =
13   let z = 7 in
14   let w = 2 in
15   let mzw = mult_ds z w in
16   let dzw = dub_sqr z in
17   printf "z: %d  w: %d\n" z w;
18   printf "mzw: %d\n" mzw;
19   printf "dzw: %d\n" dzw;
20 ;;
21
22 main ();;
```

```
dub_sq frame pops off
main proceeds with printing
```

FRAME	SYMBOL	VALUE
init	dub_sqr	<fun>
line:22	mult_ds	<fun>
	main	<fun>
main	z	7
line:16	w	2
	mzw	784
	dzw	98
printf	format	"z:.."
line:??	??	7
	??	2

`printf` is like any other function: gets parameters pushed onto stack, eventually returns `unit`

Call Stack Wrap

- All sensible programming languages implement function calls/activation records, mostly like what is shown
- Demo shows a **model** of how to understand function calls/activation records

  > *All models are wrong. Some models are useful.*
  > *– George Box*

- The model is definitely **wrong**
  - details of control transfer / return values are squiggy
  - haven't specified where values are actually stored
  - real CPU's don't track line #'s,
  - haven't dealt with anything except `int` values
- The model is **useful** because it is **accurate**: predicts the behavior of the program without needing above details

Recursive Functions and Activation Records

- Recursive functions work identically to normal functions
- Calling a recursive function creates a frame with local bindings
- Recursing creates another frame, potentially different bindings
- Hitting a base case returns a value, pops a frame off the stack

Recursive Calls Demo 1

```
1  let rec fact n =
2    if n=1 || n=0 then
3      1
4    else
5      let fm1 = fact (n-1) in
6      n*fm1
7  ;;
8
9  let ans = fact 5 in
10 printf "%d\n" ans;;
```

```
(A) Initial call to fact
| FRAME    | SYMBOL | VALUE |
|----------+--------+-------|
| init     | fact   | <fun> |
| line:9   | ans    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 5     |--+
| line:1   | fm1    | ??    |
|----------+--------+-------|
```

```
(B) Recursive case: another frame
| FRAME    | SYMBOL | VALUE |
|----------+--------+-------|
| init     | fact   | <fun> |
| line:9   | ans    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 5     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 4     |--+
| line:1   | fm1    | ??    |
|----------+--------+-------|
```

```
(C) Stacked recursive calls, reached base case
| FRAME    | SYMBOL | VALUE |
|----------+--------+-------|
| init     | fact   | <fun> |
| line:9   | ans    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 5     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 4     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 3     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 2     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |1
| fact     | n      | 1     |--+
| line:3   | fm1    | ??    |
|----------+--------+-------|
```

Recursive Calls Demo 2

```
1   let rec fact n =
2     if n=1 || n=0 then
3       1
4     else
5       let fm1 = fact (n-1) in
6       n*fm1
7   ;;
8
9   let ans = fact 5 in
10  printf "%d\n" ans;;
```

```
(D) Popped lowest frame off, up one level
| FRAME    | SYMBOL | VALUE |
|----------+--------+-------|
| init     | fact   | <fun> |
| line:9   | ans    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 5     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 4     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 3     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |2
| fact     | n      | 2     |--+
| line:5   | fm1    | 1     |
|----------+--------+-------|
```

```
(E) Another frame pops, return answer up
| FRAME    | SYMBOL | VALUE |
|----------+--------+-------|
| init     | fact   | <fun> |
| line:9   | ans    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 5     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |
| fact     | n      | 4     |--+
| line:5   | fm1    | ??    |<-+
|----------+--------+-------|  |6
| fact     | n      | 3     |--+
| line:5   | fm1    | 2     |
|----------+--------+-------|
```

```
(F) Stack "unwound", final answer returning
| FRAME    | SYMBOL | VALUE |
|----------+--------+-------|
| init     | fact   | <fun> |
| line:9   | ans    | ??    |<-+
|----------+--------+-------|  |120
| fact     | n      | 5     |--+
| line:5   | fm1    | 24    |
|----------+--------+-------|
```

Exercise: Two Formulations of Summation

- Consider two recursive summing functions shown
- Both use recursion to sum numbers in a given range
- **Naive** compilers will build stack frames in both cases
- However, a major difference between these formulations (?)

```
let rec sum_em_NT i stop =
  if i=stop then
    stop
  else
    let rest = sum_em_NT (i+1) stop in
    i+rest
;;
let sum4 = sum_em_NT 1 4 in ...

| FRAME     | SYMBOL | VALUE |
|-----------+--------+-------|
| sum_em_NT | i      | 1     |
| line:5    | stop   | 4     |
|           | rest   | ??    |<-+
|-----------+--------+-------|  |
| sum_em_NT | i      | 2     |--+
| line:5    | stop   | 4     |
|           | rest   | ??    |<-+
|-----------+--------+-------|  |
| sum_em_NT | i      | 3     |--+
| line:5    | stop   | 4     |
|           | rest   | ??    |<-+
|-----------+--------+-------|  |
                              --+
```

```
let rec sum_em_TR i stop sum =
  if i=stop then
    stop+sum
  else
    sum_em_TR (i+1) stop (i+sum)
;;

let sum4 = sum_em_TR 1 4 0 in ...
| FRAME     | SYMBOL | VALUE |  ^
|-----------+--------+-------|  |
| sum_em_TR | i      | 1     |  |
| line:5    | stop   | 4     |  |
|           | sum    | 0     |  |
|-----------+--------+-------|  |
| sum_em_TR | i      | 2     |  |
| line:5    | stop   | 4     |  ^
|           | sum    | 1     |  |
|-----------+--------+-------|  |
| sum_em_TR | i      | 3     |  |
| line:5    | stop   | 4     |  ^
|           | sum    | 3     |  |
|-----------+--------+-------|  |
                                ^
```

Answers: Two Formulations of Summation

- `sum_NT` recurses, then adds to compute final answer
  - Frames above get answers from frames below, add and return
- `sum_TR` adds, then recurses (no downward dependence)
  - Frames above add, then return answer from frame below

```
let rec sum_em_NT i stop =
  if i=stop then
    stop
  else
    let rest = sum_em_ET (i+1) stop in
    i+rest
;;
let sum4 = sum_em_NT 1 4 in ...
```

```
| FRAME     | SYMBOL | VALUE |
|-----------+--------+-------|
| sum_em_NT | i      | 1     |
| line:5    | stop   | 4     |
|           | rest   | ??    |<-+
|-----------+--------+-------|  |
| sum_em_NT | i      | 2     |--+
| line:5    | stop   | 4     |
|           | rest   | ??    |<-+
|-----------+--------+-------|  |
| sum_em_NT | i      | 3     |--+
| line:5    | stop   | 4     |
|           | rest   | ??    |<-+
|-----------+--------+-------|  |
                              --+
```

```
let rec sum_em_TR i stop sum =
  if i=stop then
    stop+sum
  else
    sum_em_TR (i+1) stop (i+sum)
;;

let sum4 = sum_em_TR 1 4 0 in ...
```

```
| FRAME     | SYMBOL | VALUE |   ^
|-----------+--------+-------|   |
| sum_em_TR | i      | 1     |   |
| line:5    | stop   | 4     |   |
|           | sum    | 0     |   |
|-----------+--------+-------|   |
| sum_em_TR | i      | 2     |   |
| line:5    | stop   | 4     |   ^
|           | sum    | 1     |   |
|-----------+--------+-------|   |
| sum_em_TR | i      | 3     |   |
| line:5    | stop   | 4     |   ^
|           | sum    | 3     |   |
|-----------+--------+-------|   |
                                 ^
                                 |
```

Tail Call Optimization

- **Tail Call:** Return the value of produced by a function call without modification, often the case in recursive functions
- A semi-sophisticated compiler will recognize lack of downward dependence and implement a **tail call optimization**
- Re-use existing Frame for the final function call

```
STANDARD IMPLEMENTATION: linear stack growth
| FRAME      | SYM  | V |  | FRAME      | SYM  | V |  | FRAME      | SYM  | V |
|------------+------+---|  |------------+------+---|  |------------+------+---|
| sum_em_TR  | i    | 1 |  | sum_em_TR  | i    | 1 |  | sum_em_TR  | i    | 1 |
| line:5     | stop | 4 |  | line:5     | stop | 4 |  | line:5     | stop | 4 |
|            | sum  | 0 |  |            | sum  | 0 |  |            | sum  | 0 |
|------------+------+---|  |------------+------+---|  |------------+------+---|
                           | sum_em_TR  | i    | 2 |  | sum_em_TR  | i    | 2 |
                           | line:5     | stop | 4 |  | line:5     | stop | 4 |
                           |            | sum  | 1 |  |            | sum  | 1 |
                           |------------+------+---|  |------------+------+---|
                                                      | sum_em_TR  | i    | 3 |
                                                      | line:5     | stop | 4 |
                                                      |            | sum  | 3 |
                                                      |------------+------+---|
TAIL CALL OPTIMIZATION: constant stack space
| FRAME      | SYM  | V |  | FRAME      | SYM  | V |  | FRAME      | SYM  | V |
|------------+------+---|  |------------+------+---|  |------------+------+---|
| sum_em_TR  | i    | 1 |  | sum_em_TR  | i    | 2 |  | sum_em_TR  | i    | 3 |
| line:5     | stop | 4 |  | line:5     | stop | 4 |  | line:5     | stop | 4 |
|            | sum  | 0 |  |            | sum  | 1 |  |            | sum  | 3 |
|------------+------+---|  |------------+------+---|  |------------+------+---|
```

Helpers and Tail Recursion

- Tail recursion often requires extra "auxiliary" parameters
- To avoid extra params in public-facing interfaces, internal tail-recursive helper functions are often used

```
(* Typical implementation of a
   summing function; main interface
   takes start and stop; internal
   recursive helper function tracks
   index i, has parameter for sum
   to allow it to be tail
   recursive *)
let sum_em start stop =
  let rec helper i sum =
    if i=stop then
      stop+sum
    else
      helper (i+1) (i+sum)
  in
  helper start 0
;;
```

```
(* Factorial implementation with
   internal tail-recursive helper
   function; avoids the need to
   pass extra params to main
   function. *)
let factorial n =
  let rec fact i prod =
    if i > n then
      prod
    else
      fact (i+1) (i*prod)
  in
  fact 1 1
;;
```

Exercise: Recognizing Tail-Recursive Functions

- Consider the following 3 recursive definitions of a list min operation A, B, C
- All throw exceptions on empty lists
- Which are tail recursive?

```
let list_min_A list =
  let rec helper curmin lst =
    if lst=[] then
      curmin
    else
      let head = List.hd lst in
      let tail = List.tl lst in
      let tmin = helper curmin tail in
      if head<tmin then
        head
      else
        tmin
  in
  helper (List.hd list) (List.tl list)
;;
```

```
let list_min_B list =
  let rec helper curmin lst =
    if lst=[] then
      curmin
    else
      let head = List.hd lst in
      let tail = List.tl lst in
      let newmin =
        if head<curmin then
          head
        else
          curmin
      in
      helper newmin tail
  in
  helper (List.hd list) (List.tl list)
;;

let rec list_min_C list =
  let head = List.hd list in
  let tail = List.tl list in
  if tail=[] then
    head
  else
    let tmin = list_min_C tail in
    if head<tmin then
      head
    else
      tmin
;;
```

Answers:: Recognizing Tail-Recursive Functions

- Only B is tail recursive: call is done as final step of recursive case
- A,C do selection after recursion

```
let list_min_A list =
  let rec helper curmin lst =
    if lst=[] then
      curmin
    else
      let head = List.hd lst in
      let tail = List.tl lst in
      let tmin = helper curmin tail in
      if head<tmin then (* select after *)
        head            (* recursion *)
      else              (* NOT tail- *)
        tmin            (* recursive *)
  in
  helper (List.hd list) (List.tl list)
;;
```

```
let list_min_B list =
  let rec helper curmin lst =
    if lst=[] then
      curmin
    else
      let head = List.hd lst in
      let tail = List.tl lst in
      let newmin =
        if head<curmin then
          head
        else
          curmin
      in                   (* recurse last *)
      helper newmin tail (* IS tail- *)
  in                       (* recursive *)
  helper (List.hd list) (List.tl list)
;;

let rec list_min_C list =
  let head = List.hd list in
  let tail = List.tl list in
  if tail=[] then
    head
  else                    (* recurse *)
    let tmin = list_min_C tail in
    if head<tmin then     (* then select *)
      head                (* answer *)
    else                  (* NOT tail- *)
      tmin                (* recursive *)
;;
```

Tail Call Wrap

- Tail Call: return a value generated by calling a function without modification
- Tail Call Optimization: Re-use existing stack frame for the next function call
- Can often be done with recursive calls, sometimes in other situations
- Enabled in source code but ultimately **done by the compiler**
- Not all PL/Env support tail call optimizations

PL/Env	TC Opt?	Notes
OCaml	Yes	Cause it rules
SML/NJ	Yes	Most ML dialects support tail call opt
Scheme	Yes	Required by Scheme spec
Common Lisp	Maybe	Some implementations support it
C / gcc	Maybe	Compiler options may do it
Java	No	JVM generally preserves stack frames
Python	No	Not supported