Programming Languages and Paradigms
COMP 302

Instructor: Jacob Errington
School of Computer Science
McGill University

Lesson 3: tail-recursion

Last time...

- `let` ... `in` ... expressions: operational and static semantics.
- Scoping, shadowing.

This time...

- Tracing a tail-recursive function.
- Introduction to the world of types.

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

Tracing

Let's trace the evaluation of `sum 5`.

A trace shows every step of the computation until we reach a value, where no more evaluation is possible.

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` $\rightarrow$ subst. 5 into fn

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` $\rightarrow$ subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` $\rightarrow$ eval condition

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` $\rightarrow$ subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` $\rightarrow$ eval condition
3. `if false then 0 else 5 + sum (5 - 1)` $\rightarrow$ choose `else`-branch

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` $\rightarrow$ subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` $\rightarrow$ eval condition
3. `if false then 0 else 5 + sum (5 - 1)` $\rightarrow$ choose `else`-branch
4. `5 + sum 4` $\rightarrow$ recursive call

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` $\rightarrow$ subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` $\rightarrow$ eval condition
3. `if false then 0 else 5 + sum (5 - 1)` $\rightarrow$ choose `else`-branch
4. `5 + sum 4` $\rightarrow$ recursive call
5. `5 + (4 + sum 3)` $\rightarrow$ recursive call

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` $\rightarrow$ subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` $\rightarrow$ eval condition
3. `if false then 0 else 5 + sum (5 - 1)` $\rightarrow$ choose `else`-branch
4. `5 + sum 4` $\rightarrow$ recursive call
5. `5 + (4 + sum 3)` $\rightarrow$ recursive call
6. `5 + (4 + (3 + sum 2))` $\rightarrow$ recursive call

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` $\rightarrow$ subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` $\rightarrow$ eval condition
3. `if false then 0 else 5 + sum (5 - 1)` $\rightarrow$ choose `else`-branch
4. `5 + sum 4` $\rightarrow$ recursive call
5. `5 + (4 + sum 3)` $\rightarrow$ recursive call
6. `5 + (4 + (3 + sum 2))` $\rightarrow$ recursive call
7. `5 + (4 + (3 + (2 + sum 1)))` $\rightarrow$ recursive call

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` $\rightarrow$ subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` $\rightarrow$ eval condition
3. `if false then 0 else 5 + sum (5 - 1)` $\rightarrow$ choose `else`-branch
4. `5 + sum 4` $\rightarrow$ recursive call
5. `5 + (4 + sum 3)` $\rightarrow$ recursive call
6. `5 + (4 + (3 + sum 2))` $\rightarrow$ recursive call
7. `5 + (4 + (3 + (2 + sum 1)))` $\rightarrow$ recursive call
8. `5 + (4 + (3 + (2 + (1 + sum 0))))` $\rightarrow$ return; add

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` $\rightarrow$ subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` $\rightarrow$ eval condition
3. `if false then 0 else 5 + sum (5 - 1)` $\rightarrow$ choose `else`-branch
4. `5 + sum 4` $\rightarrow$ recursive call
5. `5 + (4 + sum 3)` $\rightarrow$ recursive call
6. `5 + (4 + (3 + sum 2))` $\rightarrow$ recursive call
7. `5 + (4 + (3 + (2 + sum 1)))` $\rightarrow$ recursive call
8. `5 + (4 + (3 + (2 + (1 + sum 0))))` $\rightarrow$ return; add
9. `5 + (4 + (3 + (2 + (1 + 0))))` $\rightarrow$ return; add

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` → subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` → eval condition
3. `if false then 0 else 5 + sum (5 - 1)` → choose `else`-branch
4. `5 + sum 4` → recursive call
5. `5 + (4 + sum 3)` → recursive call
6. `5 + (4 + (3 + sum 2))` → recursive call
7. `5 + (4 + (3 + (2 + sum 1)))` → recursive call
8. `5 + (4 + (3 + (2 + (1 + sum 0))))` → return; add
9. `5 + (4 + (3 + (2 + (1 + 0))))` → return; add
10. `5 + (4 + (3 + (2 + 1)))` → return; add

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` → subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` → eval condition
3. `if false then 0 else 5 + sum (5 - 1)` → choose `else`-branch
4. `5 + sum 4` → recursive call
5. `5 + (4 + sum 3)` → recursive call
6. `5 + (4 + (3 + sum 2))` → recursive call
7. `5 + (4 + (3 + (2 + sum 1)))` → recursive call
8. `5 + (4 + (3 + (2 + (1 + sum 0))))` → return; add
9. `5 + (4 + (3 + (2 + (1 + 0))))` → return; add
10. `5 + (4 + (3 + (2 + 1)))` → return; add
11. `5 + (4 + (3 + 3))` → return; add

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` → subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` → eval condition
3. `if false then 0 else 5 + sum (5 - 1)` → choose `else`-branch
4. `5 + sum 4` → recursive call
5. `5 + (4 + sum 3)` → recursive call
6. `5 + (4 + (3 + sum 2))` → recursive call
7. `5 + (4 + (3 + (2 + sum 1)))` → recursive call
8. `5 + (4 + (3 + (2 + (1 + sum 0))))` → return; add
9. `5 + (4 + (3 + (2 + (1 + 0))))` → return; add
10. `5 + (4 + (3 + (2 + 1)))` → return; add
11. `5 + (4 + (3 + 3))` → return; add
12. `5 + (4 + 6)` → return; add

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` → subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` → eval condition
3. `if false then 0 else 5 + sum (5 - 1)` → choose `else`-branch
4. `5 + sum 4` → recursive call
5. `5 + (4 + sum 3)` → recursive call
6. `5 + (4 + (3 + sum 2))` → recursive call
7. `5 + (4 + (3 + (2 + sum 1)))` → recursive call
8. `5 + (4 + (3 + (2 + (1 + sum 0))))` → return; add
9. `5 + (4 + (3 + (2 + (1 + 0))))` → return; add
10. `5 + (4 + (3 + (2 + 1)))` → return; add
11. `5 + (4 + (3 + 3))` → return; add
12. `5 + (4 + 6)` → return; add
13. `5 + 10` → return; add

Call stack visualization: tracing

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

`sum n` adds up all integers from 0 to n

1. `sum 5` → subst. 5 into fn
2. `if 5 = 0 then 0 else 5 + sum (5 - 1)` → eval condition
3. `if false then 0 else 5 + sum (5 - 1)` → choose `else`-branch
4. `5 + sum 4` → recursive call
5. `5 + (4 + sum 3)` → recursive call
6. `5 + (4 + (3 + sum 2))` → recursive call
7. `5 + (4 + (3 + (2 + sum 1)))` → recursive call
8. `5 + (4 + (3 + (2 + (1 + sum 0))))` → return; add
9. `5 + (4 + (3 + (2 + (1 + 0))))` → return; add
10. `5 + (4 + (3 + (2 + 1)))` → return; add
11. `5 + (4 + (3 + 3))` → return; add
12. `5 + (4 + 6)` → return; add
13. `5 + 10` → return; add
14. `15`

Trick: tail calls

```
1 let rec sum n =
2   if n = 0 then 0 else n + sum (n-1)
```

- Stack growth is a consequence of our code structure: There is something to do *after* the recursive call completes, i.e. adding `n`.
- If there were nothing to do, then the stack frame could be recycled!

Tracing `sum'`

Idea

Before: build up a stack of pending additions *then* do them.

After: add up along the way with a partial sum argument.

Tracing sum'

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body

Tracing sum'

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.

Tracing `sum'`

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.
3. `if false then 0 else sum' (0+5) (5-1)` → choose `else`-branch

Tracing `sum'`

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.
3. `if false then 0 else sum' (0+5) (5-1)` → choose `else`-branch
4. `sum' (0 + 5) (5 - 1)` → add; recursive call

Tracing `sum'`

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.
3. `if false then 0 else sum' (0+5) (5-1)` → choose `else`-branch
4. `sum' (0 + 5) (5 - 1)` → add; recursive call
5. `sum' (5 + 4) (4 - 1)` → add; recursive call

Tracing `sum'`

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.
3. `if false then 0 else sum' (0+5) (5-1)` → choose `else`-branch
4. `sum' (0 + 5) (5 - 1)` → add; recursive call
5. `sum' (5 + 4) (4 - 1)` → add; recursive call
6. `sum' (9 + 3) (3 - 1)` → add; recursive call

Tracing `sum'`

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.
3. `if false then 0 else sum' (0+5) (5-1)` → choose `else`-branch
4. `sum' (0 + 5) (5 - 1)` → add; recursive call
5. `sum' (5 + 4) (4 - 1)` → add; recursive call
6. `sum' (9 + 3) (3 - 1)` → add; recursive call
7. `sum' (12 + 2) (2 - 1)` → add; recursive call

Tracing `sum'`

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.
3. `if false then 0 else sum' (0+5) (5-1)` → choose `else`-branch
4. `sum' (0 + 5) (5 - 1)` → add; recursive call
5. `sum' (5 + 4) (4 - 1)` → add; recursive call
6. `sum' (9 + 3) (3 - 1)` → add; recursive call
7. `sum' (12 + 2) (2 - 1)` → add; recursive call
8. `sum' (14+1) (1-1)` → add; rec. call; subst. args. into body

Tracing `sum'`

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.
3. `if false then 0 else sum' (0+5) (5-1)` → choose `else`-branch
4. `sum' (0 + 5) (5 - 1)` → add; recursive call
5. `sum' (5 + 4) (4 - 1)` → add; recursive call
6. `sum' (9 + 3) (3 - 1)` → add; recursive call
7. `sum' (12 + 2) (2 - 1)` → add; recursive call
8. `sum' (14+1) (1-1)` → add; rec. call; subst. args. into body
9. `if 0 = 0 then 15 else sum' (15 + 0) (0 - 1)` → eval. cond.

Tracing `sum'`

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.
3. `if false then 0 else sum' (0+5) (5-1)` → choose `else`-branch
4. `sum' (0 + 5) (5 - 1)` → add; recursive call
5. `sum' (5 + 4) (4 - 1)` → add; recursive call
6. `sum' (9 + 3) (3 - 1)` → add; recursive call
7. `sum' (12 + 2) (2 - 1)` → add; recursive call
8. `sum' (14+1) (1-1)` → add; rec. call; subst. args. into body
9. `if 0 = 0 then 15 else sum' (15 + 0) (0 - 1)` → eval. cond.
10. `if true then 15 else ...` → choose `then`-branch

Tracing sum'

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5` → subst. args. into body
2. `if 5 = 0 then 0 else sum' (0+5) (5-1)` → eval. cond.
3. `if false then 0 else sum' (0+5) (5-1)` → choose `else`-branch
4. `sum' (0 + 5) (5 - 1)` → add; recursive call
5. `sum' (5 + 4) (4 - 1)` → add; recursive call
6. `sum' (9 + 3) (3 - 1)` → add; recursive call
7. `sum' (12 + 2) (2 - 1)` → add; recursive call
8. `sum' (14+1) (1-1)` → add; rec. call; subst. args. into body
9. `if 0 = 0 then 15 else sum' (15 + 0) (0 - 1)` → eval. cond.
10. `if true then 15 else ...` → choose `then`-branch
11. `15`

Where did the call stack go?

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

1. `sum' 0 5`
2. `sum' 5 4`
3. `sum' 9 3`
4. `sum' 12 2`
5. `sum' 14 1`
6. `sum' 15 0`
7. `15`

One stack frame is allocated for the initial call to `sum'`. Recursive calls to `sum'` reuse the same stack frame because *the result of the recursive call is immediately returned.*

Tail calls

The result of the recursive call is immediately returned.
In other words, the recursive call is a tail call.

Recap: tail calls and tail recursion

- Tail call optimization (TCO) recycles the stack frame of the current function.

Recap: tail calls and tail recursion

- Tail call optimization (TCO) recycles the stack frame of the current function.
- Only possible when the last step of evaluating a function is to call another function.

Recap: tail calls and tail recursion

- Tail call optimization (TCO) recycles the stack frame of the current function.
- Only possible when the last step of evaluating a function is to call another function.
- Very important for recursive algorithms! Enables running in *constant stack space.*

Recap: tail calls and tail recursion

- Tail call optimization (TCO) recycles the stack frame of the current function.
- Only possible when the last step of evaluating a function is to call another function.
- Very important for recursive algorithms! Enables running in *constant stack space.*
- An extra parameter is often needed to build up the result, often called an accumulator.

Recap: tail calls and tail recursion

- Tail call optimization (TCO) recycles the stack frame of the current function.
- Only possible when the last step of evaluating a function is to call another function.
- Very important for recursive algorithms! Enables running in *constant stack space.*
- An extra parameter is often needed to build up the result, often called an accumulator.

Recap: tail calls and tail recursion

- Tail call optimization (TCO) recycles the stack frame of the current function.
- Only possible when the last step of evaluating a function is to call another function.
- Very important for recursive algorithms! Enables running in *constant stack space.*
- An extra parameter is often needed to build up the result, often called an accumulator.

```
1 let rec sum' partial_sum n =
2   if n = 0 then partial_sum else
3   sum' (partial_sum + n) (n - 1)
```

`partial_sum` is an accumulator.

- A recursive algorithm that uses only tail calls is called tail-recursive.

Exercise

Rewrite the following function to be tail-recursive.

```
1 let rec factorial (n : int) : int =
2   if n = 0 then 1 else n * factorial (n - 1)
```

Short break before moving on.

The world of types

- Type synonyms
- Tuples
- Alternatives
- `match`

demo