Call Trees

```
fun sum_list nil = 0
| sum_list (x::xs) = x + sum_list xs
```

has a **linear** call-tree

```
sum_list ([2,1])
  |
sum_list ([1])
  |
sum_list (nil)
```

```
fun fib 0 = 0
| fib 1 = 1
| fib n = fib (n-1) + fib (n-2)
```

has a **non-linear (branching)** call-tree

```
fib (3)
  /
  /
  /
  /
fib (2) fib (1)
  /     /
  /
  /
fib (0) fib (1)
```
fun reverse nil = nil
| reverse (x::xs) = reverse xs @ [x]
val L = [1,2,3];
reverse(L);

Environment during recursion: (see p. 67)

..added in reverse(nil)

xs  nil
x  3

..added in reverse([3])

xs  [3]
x  2

..added in reverse([2,3])

xs  [2,3]
x  1

..added in reverse([1,2,3])

L  [1,2,3]
...  .top level environment
Running Time

fun reverse nil = nil
| reverse (x::xs) = (reverse xs) @ [x]

Consider calling `reverse` on a list of length $n$

- it makes $n$ calls to `append`
- which takes time $1, 2, \ldots n-2, n-1, n$

the running time is thus **quadratic.**
Performance Test

We need generator of large data:

```haskell
fun from i j = 
  if i > j then nil 
  else i :: from (i+1) j
```

Execute reverse L where L is the value of (from 1 n)

<table>
<thead>
<tr>
<th>n</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>2 seconds</td>
</tr>
<tr>
<td>20,000</td>
<td>7 seconds</td>
</tr>
<tr>
<td>40,000</td>
<td>34 seconds</td>
</tr>
<tr>
<td>100,000</td>
<td>very slow</td>
</tr>
</tbody>
</table>

When testing `sum_list`, we rather want

```haskell
fun ones 0 = nil 
|    ones n = 1 :: ones (n−1)
```
Assessment

```haskell
fun reverse nil = nil
| reverse (x :: xs) = (reverse xs) @ [x]
```

Why must we call `append`?

- `::` only allows us to add items in front of list
- `reverse` does non-trivial computation only when going up the tree

We might consider doing computation when going down the tree
Passing Results Down In Call Tree

Recall that list reversal is special case of foldl

```haskell
fun foldl f e nil = e
| foldl f e (x::xs) = foldl f (f(x,e)) xs

fun my_reverse xs = foldl op:: nil xs;
```

Specializing foldl wrt op:: yields

```haskell
fun rev_acc e nil = e
| rev_acc e (x::xs) = rev_acc (x::e) xs

fun reverse_acc xs = rev_acc nil xs
```

- e holds “the results so far”
- e is flowing down the tree, informing the recursion at the next level of something that we have accumulated at the current level
Performance Comparison

- Recall that reverse had **quadratic** running time.
- Since reverse_acc uses no append, we expect **linear** running time.

When called on the value of \( \text{from 1 to } n \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>reverse</th>
<th>reverse_acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>2 seconds</td>
<td>instantaneous</td>
</tr>
<tr>
<td>20,000</td>
<td>7 seconds</td>
<td>instantaneous</td>
</tr>
<tr>
<td>100,000</td>
<td>very slow</td>
<td>instantaneous</td>
</tr>
<tr>
<td>1,000,000</td>
<td>infeasible</td>
<td>3 seconds</td>
</tr>
</tbody>
</table>
This function is tail recursive:

- no computation happens after the recursive call
- value of recursive call is the return value
- thus, no variables are referenced after recursive call

This kind of recursion is actually iteration in disguise!
Iterative Reverse

```haskell
fun rev_acc e nil = e
| rev_acc e (x::xs) = rev_acc (x::e) xs
```

can be converted to “pseudo-C (renaming e to acc):

```c
list reverse(xs:list) {
    list acc;
    acc = [];
    while (xs != nil) do {
        acc = hd(xs) :: acc;
        xs = tl(xs);
    }
    return acc;
}
```

- acc holds result
- xs and acc are updated each time through the loop
Tail Recursion versus Non-Tail Recursion

(* version 1: without accumulator *)
fun reverse nil = nil
| reverse (x::xs) = reverse xs @ [x]

(* version 2: with accumulator *)
fun rev_acc e nil = e
| rev_acc e (x::xs) = rev_acc (x::e) xs

x is used after recursion in v.1, but not in v.2
  ➤ for tail-recursive functions, we do thus not need to stack variable bindings for the recursive calls
  ➤ parameter passing can be implemented in the compiler by destructive updates (that is, assignment)!

Computation occurs after recursion in v.1, but not in v.2
  ➤ for tail-recursive functions, we do thus not need to stack return addresses; a call can be implemented in the compiler as a goto!
The tail-recursive function

\[
\textbf{fun } f(y_1, \ldots, y_n) =
\]

\[
\quad \ldots
\]

\[
\quad \quad f(<\text{exp}-1>, \ldots, <\text{exp}-n>)
\]

...is roughly equivalent to...

\[
\quad \ldots f(y_1, \ldots, y_n) \{ \]

\[
\quad \quad \textbf{while } \ldots \{ \]

\[
\quad \quad \quad \ldots
\]

\[
\quad \quad \quad \ldots
\]

\[
\quad \quad \quad y_1 = <\text{exp}-1>;
\]

\[
\quad \quad \quad \ldots
\]

\[
\quad \quad \quad y_n = <\text{exp}-n>;
\]

\[
\quad \quad \}
\]

\[
\quad \}
\]
Converting SumList to Tail Recursion

```haskell
fun sum_list nil = 0
| sum_list (x::xs) = x + sum_list xs
```

- The recursive calls are unfolded until we reach the end of the list, from where we then move to the left while summing the results.

```haskell
fun sum_list_acc acc nil = acc
| sum_list_acc acc (x::xs) = sum_list_acc (x+acc) xs
```

- Summation proceeds while moving left to right.
- Top-level call: `sum_list_acc 0 xs`

Performance comparison on the value of `ones`\( n \)

<table>
<thead>
<tr>
<th>( n )</th>
<th><code>sum_list</code></th>
<th><code>sum_list_acc</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>4,000,000</td>
<td>5 seconds</td>
<td>instantaneous</td>
</tr>
<tr>
<td>5,000,000</td>
<td>21 seconds</td>
<td>instantaneous</td>
</tr>
</tbody>
</table>
Tail-Recursive MultList

fun mult_list_acc acc nil = acc
| mult_list_acc acc (x::xs) =
  mult_list_acc (x*acc) xs

Question: what happens if we hit a 0?

fun mult_list_acc_exit acc nil = acc
| mult_list_acc_exit acc (x::xs) =
  if x = 0 then 0 else
  mult_list_acc_exit (x*acc) xs

In C, we might have

```c
int mult_list(xs:list) {
  int acc;
  acc = 1;
  while (xs != nil) do {
    if (hd(xs) = 0) then
      return 0; /* escape */
    else
      acc = hd(xs) * acc;
      xs = tl(xs);
  }
  return acc;
}
```
Making Fibonacci Tail-Recursive

\[
\text{fun } \text{fib} \ 0 = 0 \\
| \quad \text{fib} \ 1 = 1 \\
| \quad \text{fib} \ n = \text{fib} (n-2) + \text{fib} (n-1)
\]

has a branching call-tree, and can be made tail-recursive by using \textit{two} accumulating parameters:

\[
\text{fun } \text{fib}_\text{acc} \ \text{prev} \ \text{curr} \ n = \\
\quad \text{if} \ n = 1 \ \text{then} \ \text{curr} \\
\quad \text{else } \text{fib}_\text{acc} \ \text{curr} \ (\text{prev}+\text{curr}) \ (n-1)
\]

\[
\text{fun } \text{fibonacci}_\text{acc} \ n = \\
\quad \text{if} \ n = 0 \ \text{then} \ 0 \ \text{else } \text{fib}_\text{acc} \ 0 \ 1 \ n
\]

Performance comparison

<table>
<thead>
<tr>
<th>(n)</th>
<th>\text{fib} \</th>
<th>\text{fibonacci}_\text{acc}</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>7 \ seconds</td>
<td>instantaneous</td>
</tr>
<tr>
<td>43</td>
<td>11 \ seconds</td>
<td>instantaneous</td>
</tr>
<tr>
<td>44</td>
<td>17 \ seconds</td>
<td>instantaneous</td>
</tr>
</tbody>
</table>
Correctness of Tail-Recursive Fibonacci

With $F$ the fibonacci function we have

$$F(0) = 0; \quad F(1) = 1; \quad F(n) = F(n-2) + F(n-1)$$

which can be tail-recursively implemented by

```python
fun g(n, prev, curr) = 
  if n = 1 then curr 
  else g(n-1, curr, prev+curr)
```

**Correctness Lemma:** for all $n \geq 1$, $k \geq 0$:

$$g(n, F(k), F(k+1)) = F(n + k)$$

This can be proved by induction in $n$.

- the base case is $n = 1$ which is obvious.
- for the inductive case, $n > 1$,

$$g(n, F(k), F(k+1)) = g(n-1, F(k+1), F(k)+F(k+1)) =$$

$$g(n-1, F(k+1), F(k+2)) = F((n-1)+(k+1)) = F(n+k)$$

Thus $F(n) = g(n, F(0), F(1)) = g(n, 0, 1)$. 
Summary

- A tail-recursive function is one where the function performs **no computation after** the recursive call.
- A good SML compiler will **detect** tail-recursive functions and **implement** them iteratively as loops.
  - There is no need to stack bindings or return addresses.
  - Recursive calls become **gotos**.
  - We can think of arguments as being “assigned to” (destructively update) formal parameters.
- This substantially reduces execution time and space (for stack) overhead.