The following function will take more or less the same amount of time to complete \textit{regardless of the length of Xs}:

\begin{verbatim}
fun {EndOfList Xs}
    case Xs of nil then true
    else false end
end
\end{verbatim}

- It will likely take different amount of time to complete on different computers! Therefore we \textit{roughly} say that the case statement takes some constant time, $k$, to complete. On one computer $k$ could be 0.2ms, and on another 0.3ms.
- The same is not true for the following recursive function. Its runtime seems to vary with the length of the input variable Xs. Somehow its runtime is \textit{not constant}. This is because one of the statements inside the case statement is a recursive call.

\begin{verbatim}
fun {Len Xs}
    case Xs of nil then 0
             [] X|Xr then 1+{Len Xr} end
end
\end{verbatim}
### Execution times for kernel instructions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$k$</td>
</tr>
<tr>
<td>$\langle x \rangle_1 = \langle x \rangle_2$</td>
<td>$k$</td>
</tr>
<tr>
<td>$\langle x \rangle = \langle y \rangle$</td>
<td>$k$</td>
</tr>
<tr>
<td>$\langle s \rangle_1 &lt; s \rangle_2$</td>
<td>$T(\langle s \rangle_1) + T(\langle s \rangle_2)$</td>
</tr>
<tr>
<td>local $\langle x \rangle$ in $\langle s \rangle$ end</td>
<td>$k + T(\langle s \rangle)$</td>
</tr>
<tr>
<td>proc ${ \langle x \rangle \langle y \rangle_1 \ldots \langle y \rangle_n }$ end</td>
<td>$k$</td>
</tr>
<tr>
<td>if $\langle x \rangle$ then $\langle s \rangle_1$ then $\langle s \rangle_2$ end</td>
<td>$k + \max(T(\langle s \rangle_1), T(\langle s \rangle_2))$</td>
</tr>
<tr>
<td>case $\langle x \rangle$ of $\langle p \rangle$ then $\langle s \rangle_1$ else $\langle s \rangle_2$ end</td>
<td>$k + \max(T(\langle s \rangle_1), T(\langle s \rangle_2))$</td>
</tr>
<tr>
<td>${ \langle x \rangle \langle y \rangle_1 \ldots \langle y \rangle_n }$</td>
<td>$T_x(\text{size}(I({y_1 \ldots y_n})))$</td>
</tr>
</tbody>
</table>
Tutorial 8 Exercise 1

fun {Reverse Xs} 
  case Xs of nil then  
    nil  
  [] X|Xr then  
    {Append {Reverse Xr} [X]}  
  end  
end

What does Reverse do?
{Reverse [a b c]} evaluates to:

{Append {Reverse [b c]} [a]}
{Append {Append {Reverse [c]} [b]} [a]}
{Append {Append {Append {Reverse nil} [c]} [b]} [a]}
{Append {Append {Append nil [c]} [b]} [a]}
{Append {Append [c] [b]} [a]}
{Append [c b] [a]}
[c b a]
Kernel Translation of Reverse

fun \{\text{Reverse } Xs\}
  \text{case } Xs \text{ of } \text{nil then }
  \text{nil }
  [] X|Xr \text{ then }
  \{\text{Append } \{\text{Reverse } Xr\} [X]\}
end
end

\textit{translates to kernel language:}

proc \{\text{Reverse } Xs \ R\}
  \text{case } Xs \text{ of } \text{nil then }
  \text{R=}\text{nil }
  [] X|Xr \text{ then }
  \text{local } T \text{ in }
  \{\text{Reverse } Xr \ T\}
  \{\text{Append } T [X] \ R\}
end
end
end
proc {Reverse Xs R}
    case Xs of nil then R=nil
    [] X|Xr then
        local T in {Reverse Xr T} {Append T [X] R} end
    end
end

- The input function I is, \( I(\{Xs\})=Xs \), that means the runtime of Reverse depends on \( Xs \)
- The size function size is, \( size(Xs)=n \) where \( n \) is the length of \( Xs \)

First, we need to figure out the asymptotic time complexity for Append,
- \( I_{Append}(\{Xs \ Ys\})=Xs \)
- \( size_{Append}(\{Xs\})=n \) where \( n \) is the length of \( Xs \)
- \( T_{Append}(\ size(I( \{Xs,Ys\} ))) = k_{Append} + n \)

Now we can figure out the asymptotic time complexity for Reverse
- \( T(n) = k_{case} + max( \ T(R=nil), T(\ local \ T \ in \ {Reverse Xr T}{Append T [X] R} \ end)) \)
- \( T(n) = k_{case} + T(\ local \ T \ in \ {Reverse Xr T}{Append T [X] R} \ end) \)
- \( T(n) = k_{case} + k_{local} + T(\ {Reverse Xr T} {Append T [X] R}) \)
- \( T(n) = k_{case} + k_{local} + T(\ {Reverse Xr T} \ ) + T(\ {Append T [X] R} \ ) \)
- \( T(n) = k_{case} + k_{local} + T_{Reverse}(n-1) + T_{Append}(n) \)
- \( T(n) = k_{case} + k_{local} + k_{Append}+n + T_{Reverse}(n-1) \)
- \( T(n) = k_{all} + n + T_{Reverse}(n-1) \)

We know that \( T(1) = T(\ case \ Xs \ of \ nil \ then \ R=nil) \) which is the same as
\( T(1) = k_{case} + k_{uni} = k_{all} \)

- \( T(n) = k_{all} + n + k_{all} + n-1 + k_{all} + n-2 + k_{all} + n-3 \ldots \) and so on… + \( k_{all} \)
- \( T(n) = (k_{all}+ n) \times n = n*k_{all} + n^2 \)
Big-Oh

Our $T(n)$ function gives us a good time estimate. But we would like to:

- Forget about the constants (because they’re different on different computers/platforms anyway etc)
- Forget about $T(n)$ when $n$ is very small.

This is done by finding another function $f(n)$ which will always be larger than $T(n)$ if it is multiplied with an constant, REGARDLESS of $n$ as long as $n$ is bigger than some value.

In other words, $f(n) \times c_1 = T(n)$ if $n = c_2$, for some constants $c_1$ and $c_2$. We then say that $T(n)$ is of order $f(n)$, or that $T(n)$ has the asymptotic time complexity $O(f(n))$

E.g. $T(n) = n^2 + n + 150$ is of order $n^2$.
Because if we choose $c_1=2$ and $c_2=150$ then for ALL values of $n$, $n^2 \times c_1 = n^2 + n + 150$ as long as $n = c_2$
Complexity of Reverse

Back to our Reverse function, we derived (slide 5):

- $T(n) = (k_{all} + n) \times n$

- So $T(n) = k_{all} \times n + n^2$

- So we know $T(n)$ is of order $n^2$, i.e. $T(n)$ is of $O(n^2)$.

To see that this is true, choose $C_1 = 2 \times k_{all} \times n + n^2$ for all $n=0$
fun {AppAll Xs}
    case Xs
    of nil then nil
    [] X|Xr then {Append X {AppAll Xr}} end
end

{AppAll [ [a b] [c] [d e] ] }
{Append [a b] {AppAll [ [c] [d e] ] } }
{Append [a b] {Append [c] {AppAll [ [d e] ] } } }
{Append [a b] {Append [c] {Append [d e] {AppAll nil} } } }
{Append [a b] {Append [c] {Append [d e] nil } } }
{Append [a b] {Append [c] [d e] } }
{Append [a b] [c d e] }
[a b c d e]
Translation to Kernel Language

\[
\text{proc } \{\text{AppAll } Xs \ R\} \\
\quad \text{case } Xs \\
\quad \quad \text{of } \text{nil } \text{then } R=\text{nil} \\
\quad \quad \quad \text{[]} X|Xr \text{ then} \\
\quad \quad \quad \quad \text{local } T \text{ in} \\
\quad \quad \quad \quad \quad \\{\text{AppAll } Xr \ T\} \\
\quad \quad \quad \quad \quad \\{\text{Append } X \ T \ R\} \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{end} \\
\]

**Tail recursive?** No, the recursive call does not come last!
How do we measure the size of the input arguments?

\( l(Xs) = Xs \), i.e. \( Xs \) is the only input argument we will consider

\( size(Xs) = n \), where \( n \) is the number of lists inside \( Xs \). I.e. we do not care about the length of the sublists since it is given that they are small and consequently constant.

That means that \( T_{\text{Append}}(n) = c_{\text{Append}} \) in this particular example

\[
T(n) = c_{\text{case}} + \max(T(R=\text{nil}), T(\text{local } T \in \{\text{AppAll } Xr T\} \{\text{Append } X T R\} \text{ end}))
\]
\[
T(n) = c_{\text{case}} + T(\text{local } T \in \{\text{AppAll } Xr T\} \{\text{Append } X T R\} \text{ end})
\]
\[
T(n) = c_{\text{case}} + c_{\text{local}} + T(\{\text{AppAll } Xr T\} \{\text{Append } X T R\})
\]
\[
T(n) = c_{\text{case}} + c_{\text{local}} + T(\{\text{AppAll } Xr T\}) + T(\{\text{Append } X T R\})
\]
\[
T(n) = c_{\text{case}} + c_{\text{local}} + T_{\text{AppAll}}(n-1) + T_{\text{Append}}(n)
\]
\[
T(n) = c_{\text{case}} + c_{\text{local}} + c_{\text{Append}} + T_{\text{AppAll}}(n-1)
\]
\[
T(n) = c_{\text{sum}} + T_{\text{AppAll}}(n-1)
\]

We know that \( T(1) = T(\text{case } Xs \text{ of nil then } R=\text{nil}) \) which is the same as

\( T(1) = c_{\text{case}} + c_{\text{uni}} = c_{\text{sum}} \)

\[
T(n) = c_{\text{sum}} + c_{\text{sum}} + T_{\text{AppAll}}(n-2) \ldots \text{ and so on } \ldots c_{\text{sum}}
\]
\[
T(n) = (c_{\text{sum}})^n
\]

It is easy to see that \( T(n) \) is of \( O(n) \), (see slide 7)
Lets make it tail-recursive

fun {AppAll Xs}
    case Xs of [X] then X
        [] X1|X2|Xr then
            {AppAll {Append X1 X2}|Xr}
        end
    end

{AppAll [ [a b] [c] [d e] ] }
{AppAll {Append [a b] [c]|[d e]|nil }
{AppAll [ [a b c] [d e] ] }
{AppAll {Append [a b c] [d e]|nil }
{AppAll [ [a b c d e] ] }
[ [a b c d e] ]
Kernel translation

fun {AppAll Xs}
  case Xs of [X] then X
  [] X1|X2|Xr then
    {AppAll {Append X1 X2}|Xr}
  end
end
end

proc {AppAll Xs R}
  case Xs of [X] then R=X
  [] X1|X2|Xr then
    local T in
      {Append X1 X2 T}
      {AppAll T|Xr R}
    end
  end
end
Asymptotic time complexity

Just as before

\( I(Xs) = Xs \), \( Xs \) is the only input argument we will consider

\( \text{size}(Xs) = n \), where \( n \) is the number of lists inside \( Xs \). I.e. we do not care about the length of the sublists since it is given that they are small consequently constant

\[ T_{\text{Append}}(n) = c_{\text{Append}} + n, \] where \( n \) is the size of the first argument to \( \text{Append} \)

\[
T(n) = c_{\text{case}} + \max(T_{\text{case}}(X), \max_{\text{local } T} \{ T_{\text{append}}(X1, X2, T) \})
\]

\[
T(n) = c_{\text{case}} + T_{\text{local}}(T \in \{ \text{append} X1, X2, T \}) \{ \text{appendAll} T | Xr \}
\]

\[
T(n) = c_{\text{case}} + c_{\text{local}} + \max_{T \in \{ \text{append} X1, X2, T \}} \{ T_{\text{append}}(n \cdot c_{\text{sublistSize}}) \} + T_{\text{appendAll}}(n-1)
\]

\[
T(n) = c_{\text{sum}} + n \cdot c_{\text{sublistSize}} + T_{\text{appendAll}}(n-1)
\]

We know that \( T(1) = T_{\text{case}}(Xs \text{ of } [X] \text{ then } R=X) \) which is the same as

\( T(1) = c_{\text{case}} + c_{\text{uni}} = c_{\text{sum}}, \) therefore

\[
T(n) = (c_{\text{sum}} + n \cdot c_{\text{sublistSize}}) \cdot n
\]

\[
T(n) = c_{\text{sum}} \cdot n + c_{\text{sublistSize}} \cdot n^2
\]

In accordance with slide 7, \( T(n) \) is of \( O(n^2) \)
Lets make it tail-recursive and efficient

```plaintext
proc {AppAll Xs R}
  case Xs
    of nil then R=nil
    [] X|Xr then
      local T in
        {Append X T R}
        {AppAll Xr T}
    end
  end
end
```
Asymptotic time complexity

How do we measure the size of the input arguments?

- \( I(Xs) = Xs \), Xs is the only input argument we will consider
- \( \text{size}(Xs) = n \), where n is the number of lists inside Xs. I.e. we do not care about the length of the sublists since it is given that they are small consequently constant.
- That means that \( T_{\text{Append}}(n) = c_{\text{Append}} \) in this particular example

\[
T(n) = c_{\text{case}} + \max(T(R=\text{nil}), T(\text{local } T \text{ in } \{\text{Append } X T R\} \{\text{AppAll } Xr T\} \text{ end})) \\
T(n) = c_{\text{case}} + T(\text{local } T \text{ in } \{\text{Append } X T R\} \{\text{AppAll } Xr T\} \text{ end}) \\
T(n) = c_{\text{case}} + c_{\text{local}} + T(\text{local } T \text{ in } \{\text{Append } X T R\} \{\text{AppAll } Xr T\} \text{ end}) \\
T(n) = c_{\text{case}} + c_{\text{local}} + T_{\text{Append}}(n) + T_{\text{AppAll}}(n-1) \\
T(n) = c_{\text{case}} + c_{\text{local}} + c_{\text{Append}} + T_{\text{AppAll}}(n-1) \\
T(n) = c_{\text{sum}} + T_{\text{AppAll}}(n-1) \\
\]

We know that \( T(1) = T(\text{case } Xs \text{ of } \text{nil then } R=\text{nil}) \) which is the same as \( T(1) = c_{\text{case}} + c_{\text{uni}} = c_{\text{sum}} \), therefore

\[
T(n) = c_{\text{sum}} + c_{\text{sum}} + T_{\text{AppAll}}(n-2) \ldots \text{ and so on } \ldots c_{\text{sum}} \\
T(n) = (c_{\text{sum}}) * n \\
\]

- It is easy to see that \( T(n) \) is of \( O(n) \)
The second lab-assignment has a similar function. Which one?

The helper function for \{HUF \text{Xs}\}, \{\text{Compress Bs Xs}\} had to convert a list of Bytes to a list of Bit sequences! Each byte in Xs is represented by its bit sequence in Bitmap, and Compress had to append/concatenate all those bit sequences to produce the result, just like AppAll!