Elimination of Recursion

To understand asymptotic time and space complexity of recursive programs, it is useful to understand how a compiler can translate a recursive program to machine language that corresponds to the RAM model; that is, how recursion is "removed" from a program.

Idea:

All local variables and formal parameters for the procedure are stored on top of a stack. When a recursive call is made, new copies are pushed.
How procedures can be removed:

Step 1: Place a label at the start of each procedure.

Step 2: Place a label directly after each procedure call.

Step 3: Replace each call $P(x_1...x_n)$ that is followed by a label $L$ by code to push the current state and \textit{goto} the label for $P$:

\begin{itemize}
  \item PUSH $L$
  \item PUSH the parameters $x_1...x_n$
  \item PUSH space for local variables declared in $P$
  \item \textbf{goto} to the label associated with $P$
\end{itemize}

Step 4: End a procedure (and replace each return statement) by code that discards from the stack the current state and goes to that return label.
Example: Eliminating Recursion from \( n! \)

Recall the recursive version of factorial:

```plaintext
function FACT(n)
    if \( n \leq 1 \)
        then return 1
    else return \( n \times \text{FACT}(n-1) \)
end
```

**Idea:** FACT has one parameter and no local variables, so the return address and current value of \( n \) are the only items that have to be pushed onto the stack when FACT is called. That is, \( stack[top] \) is (as usual) the next available empty stack frame, \( stack[top-1] \) contains \( n \), and \( stack[top-2] \) contains the return address.

At compile time, we assume that space is created for a special variable \( \text{FactReturnValue} \) that is used to store the most recent value returned by FACT.
The function FACT is converted to:

$L1: \text{ if } stack[\text{top}–1] \leq 1 \text{ then begin}$

$\text{FactReturnValue := 1}$
$\text{POP} \quad \text{— discard } n$
$\text{goto POP} \quad \text{— remove a label and go to it}$
$\text{end}$

$\text{PUSH}(L2) \quad \text{— push the return point (so } n \text{ is now in } stack[\text{top}–2])$
$\text{PUSH}(stack[\text{top}–2]–1) \quad \text{— push } n–1$
$\text{goto } L1$

$L2: \text{ FactReturnValue := FactReturnValue } \ast stack[\text{top}–1]$
$\text{POP} \quad \text{— discard } n$
$\text{goto POP} \quad \text{— remove a label and goto it}$

The call $x := \text{FACT}(n)$ is converted to:

$\text{PUSH}(L3); \text{PUSH}(n)$
$\text{goto } L1$

$L3: x := \text{FactReturnValue}$
Example: Complexity of Recursive $n!$

For $n=1$, $FACT$ takes time that is bounded by some constant $a$. For $n>1$, $FACT$ takes whatever time is consumed by the call to $FACT(n-1)$, plus an additional amount of time to perform the multiplication by $n$ that is bounded by some constant $b$.

Hence, if $T(n)$ denotes the running time of $FACT(n)$, and we let $c$ be the larger of $a$ and $b$, then:

$$T(n) \leq \begin{cases} 
c & \text{if } n \leq 1 \\
T(n-1) + c & \text{otherwise}
\end{cases}$$
Informally, we can see that for all $n \geq 1$:

$$T(n) \leq T(n - 1) + c$$

$$\leq (T(n - 2) + c) + c$$

$$\leq ((T(n - 3) + c) + c) + c$$

$$= c + c + \cdots + c \ (n \ times)$$

$$= cn$$

$$= O(n)$$
(complexity of recursive $n!$ continued)

**Theorem:** For all $n \geq 1$, $T(n) \leq cn$.

**Proof:**

(basis)
For $n=1$, $T(n) \leq c = cn$.

(inductive step)
For $n>1$, assume that $T(i) \leq ci$ for $1 \leq i < n$.
Then $T(n) \leq T(n-1)+c \leq c(n-1)+c = cn$.

**Corollary:** $T(n)$ is $O(n)$.

**Space:** A "smart" compiler would see that this is just tail recursion and translate the program to a simple loop that uses $O(1)$ space. However, if we simply apply mechanical recursion elimination, we end up with a stack that grows to $n$ frames, each using $O(1)$ space, for a total of $O(n)$ space.
Notes about Recursion Elimination

**Functions:** A number of conventions can be used for functions that return a value; for example, as with the $n!$ example, the return value can be placed in a special variable associated with that function.

**Time:** The time required to implement the calls and returns of a given procedure is proportional to the number of formal parameters and local variables, which is typically a constant.

**Space:** Each call must allocate new space on the stack for the parameters and local variables. The total space used will be the product of this and the maximum depth of the recursion. This space is "hidden" space that is being used by code added by the compiler, and is in addition to the space that is explicitly used by the program's data structures.
Example: Eliminating Recursion from Towers of Hanoi

Recall the recursive version of Towers of Hanoi:

```plaintext
procedure TOWER(n,x,y,z)
    if n>0 then begin
        TOWER(n–1,x,z,y)
        write “Move ring n from x to y.”
        TOWER(n–1,z,y,x)
    end
```

Idea:

• Four parameters and no local variables.
• The return address will always be in stack[top–5] and the current values of n, x, y, and z will always be in locations stack[top–1] down to stack[top–4].
• These 5 frames move down when new values are being pushed on during the process of simulating a recursive call.

```
+------------------+
|  top             |
|      |    n    |
|  top-1 |      x   |
|  top-2 |      y   |
|  top-3 |      z   |
|  top-4 | return address |
+------------------+
```
The procedure \textit{TOWER} is converted to:

\textbf{L1}: if \texttt{stack[top–1]} is equal to 0 \texttt{goto} \texttt{L3}

\hspace{1em} \texttt{PUSH(L2)} — push the return point
\hspace{1em} \texttt{PUSH(stack[top–4])} — push \texttt{y}, which is now one position lower
\hspace{1em} \texttt{PUSH(stack[top–6])} — push \texttt{z}, which is now two positions lower
\hspace{1em} \texttt{PUSH(stack[top–5])} — push \texttt{x}, which is now three positions lower
\hspace{1em} \texttt{PUSH(stack[top–5]–1)} — push \texttt{n–1}, where \texttt{n} is now four positions lower
\hspace{1em} \texttt{goto L1}

\textbf{L2}: \texttt{write} "Move ring \texttt{stack[top–1]} from \texttt{stack[top–2]} to \texttt{stack[top–3]}." 

\hspace{1em} \texttt{PUSH(L3)} — push the return point
\hspace{1em} \texttt{PUSH(stack[top–3])} — push \texttt{x}, which is now one position lower
\hspace{1em} \texttt{PUSH(stack[top–5])} — push \texttt{y}, which is now two positions lower
\hspace{1em} \texttt{PUSH(stack[top–7])} — push \texttt{z}, which is now three positions lower
\hspace{1em} \texttt{PUSH(stack[top–5]–1)} — push \texttt{n–1}, where \texttt{n} is now four positions lower
\hspace{1em} \texttt{goto L1}

\textbf{L3}: \texttt{POP; POP; POP; POP} — discard \texttt{n, x, y, and z}
\hspace{1em} \texttt{goto POP} — remove and \texttt{goto} the label on top of the stack

The call \textit{TOWER}(n,A,B,C) is converted to:

\hspace{1em} \texttt{PUSH(L4); PUSH(C); PUSH(B); PUSH(A); PUSH(n)}
\hspace{1em} \texttt{goto L1}

\textbf{L4}: 

- 10-
Example: Complexity of TOWER

For \( n=1 \), the two calls for \( n–1 \) do nothing and exactly one move is made. For \( n>1 \), twice whatever the number of moves required for \( n–1 \) are made plus the move made by the write statement. Hence, the number of moves made by TOWER on input \( n \) is:

\[
T(n) = \begin{cases} 
1 & \text{if } n=1 \\
2T(n−1) + 1 & \text{otherwise}
\end{cases}
\]

**Theorem:** For \( n \geq 1 \), \( TOWER(n,x,y,z) \) makes \( 2^n–1 \) moves.

**Proof:**

*(basis)*

For \( n=1 \): \( T(1) = 1 = 2^1–1 \)

*(inductive step)*

Now assume the theorem true for all values in the range 0 to \( n−1 \) (so in particular, \( T(n-1) = 2^{n-1}−1 \)); then:

\[
T(n) = 2T(n−1) + 1 \\
= 2(2^{n−1}−1) + 1 \\
= 2^n − 1
\]
(complexity of TOWER continued)

**Time used by TOWER:**

$O(2^n)$ because it is proportional to the number of moves made.

**Space used by TOWER:**

The configuration of the rings is not explicitly stored (it is implicit in the reordering of $x,y,z$ in the recursive calls). The total space used is dominated by the space for the recursion stack.

Let $c$ denote the space used by a copy of the local variables. Since the space used by the first call is released when it returns, the recurrence relation for the space used by TOWER($n,x,y,z$) is:

$$S(n) = \begin{cases} 
  c & \text{if } n=1 \\
  S(n-1) + c & \text{otherwise}
\end{cases}$$

As we have already seen in the factorial example, the solution to this recurrence relation is $O(n)$. 

- 12-