ECE 220 Computer Systems & Programming

Lecture 12 – Recursion

ECE ILLINOIS

ILLINOIS

Recursion

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

- Similar to recurrence function in mathematics.
- Like iteration -- can be used interchangeably;
  sometimes recursion results in a simpler solution.
- Must have at least 1 **base case** (terminal case) that ends the recursive process.

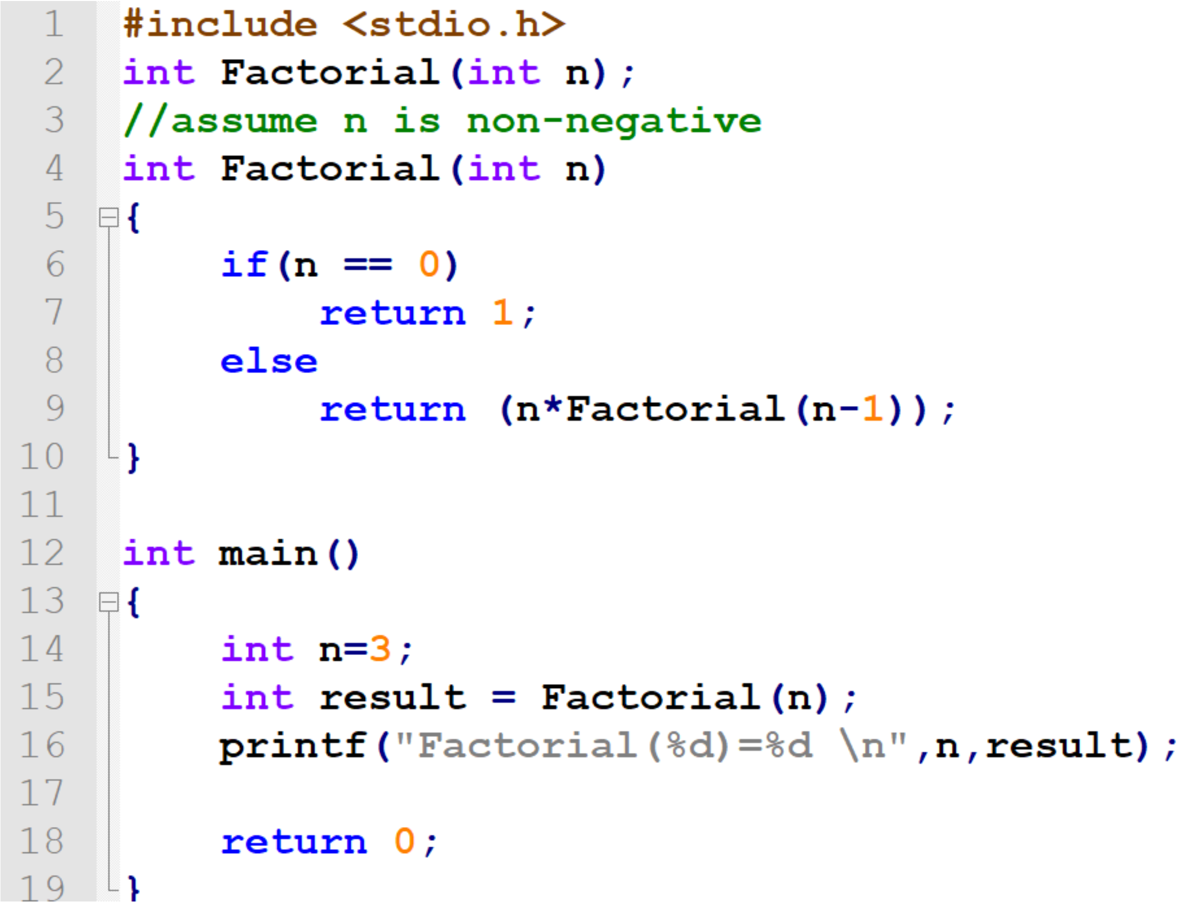
Example: n!

Factorial:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

$$n! = \begin{cases} n \cdot (n-1)! & , n > 0 \\ 1 & , n = 0 \end{cases}$$

```
int Factorial(int n)
{

     if
      Return  ….



 else

 return

}
```

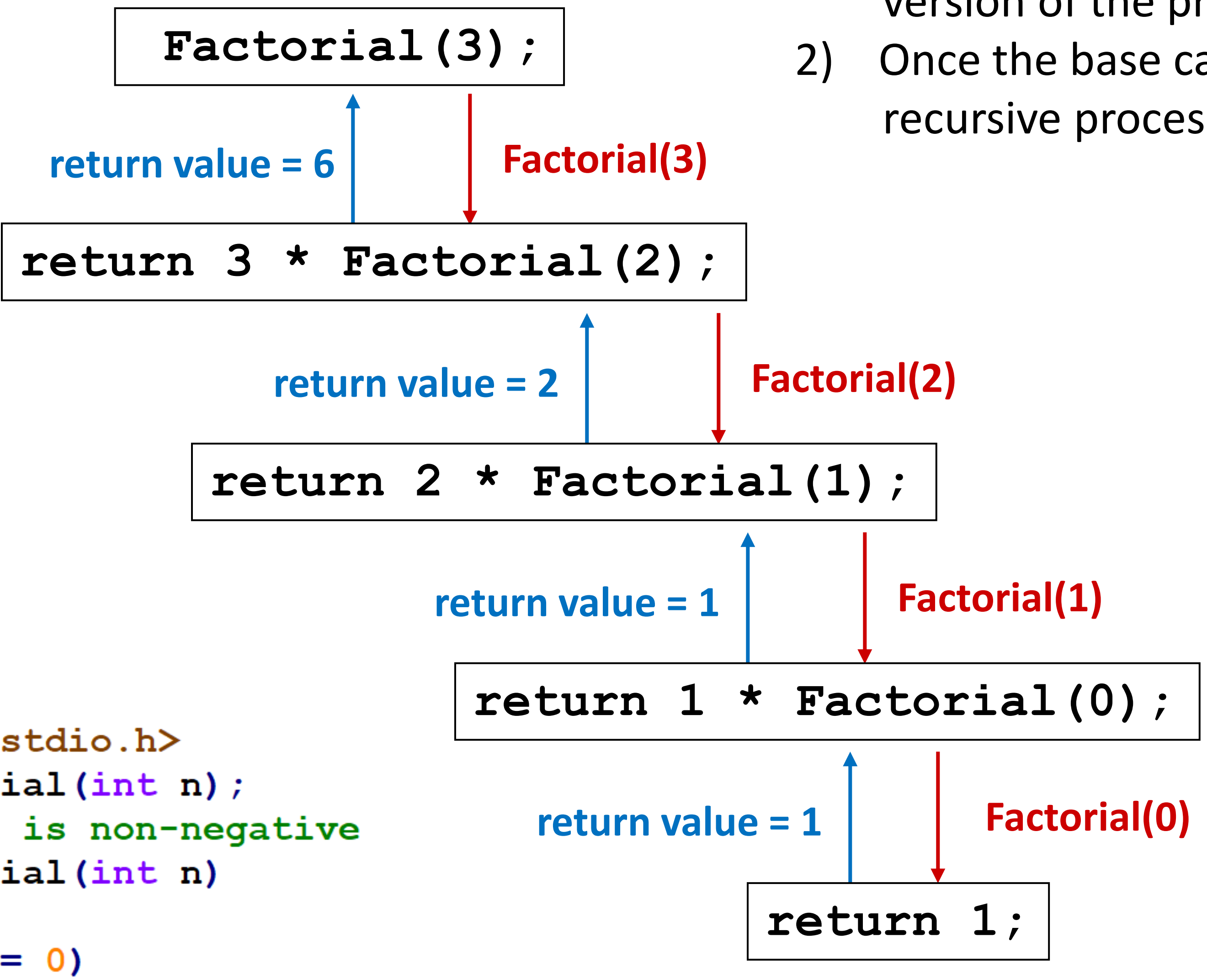


```
#include <stdio.h>
int Factorial(int n);
//assume n is non-negative
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return (n*Factorial(n-1));
}

int main()
{
    int n=3;
    int result = Factorial(n);
    printf("Factorial(%d)=%d \n",n,result);

    return 0;
}
```

Executing Factorial

Observation:

1) Each invocation solves a smaller version of the problem;
2) Once the base case is reached, recursive process stops.



Factorial(3);

return value = 6

Factorial(3)

return 3 * Factorial(2);

return value = 2

Factorial(2)

return 2 * Factorial(1);

return value = 1

Factorial(1)

return 1 * Factorial(0);

return value = 1

Factorial(0)

return 1;

```c
#include <stdio.h>
int Factorial(int n);
//assume n is non-negative
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return (n*Factorial(n-1));
}
```

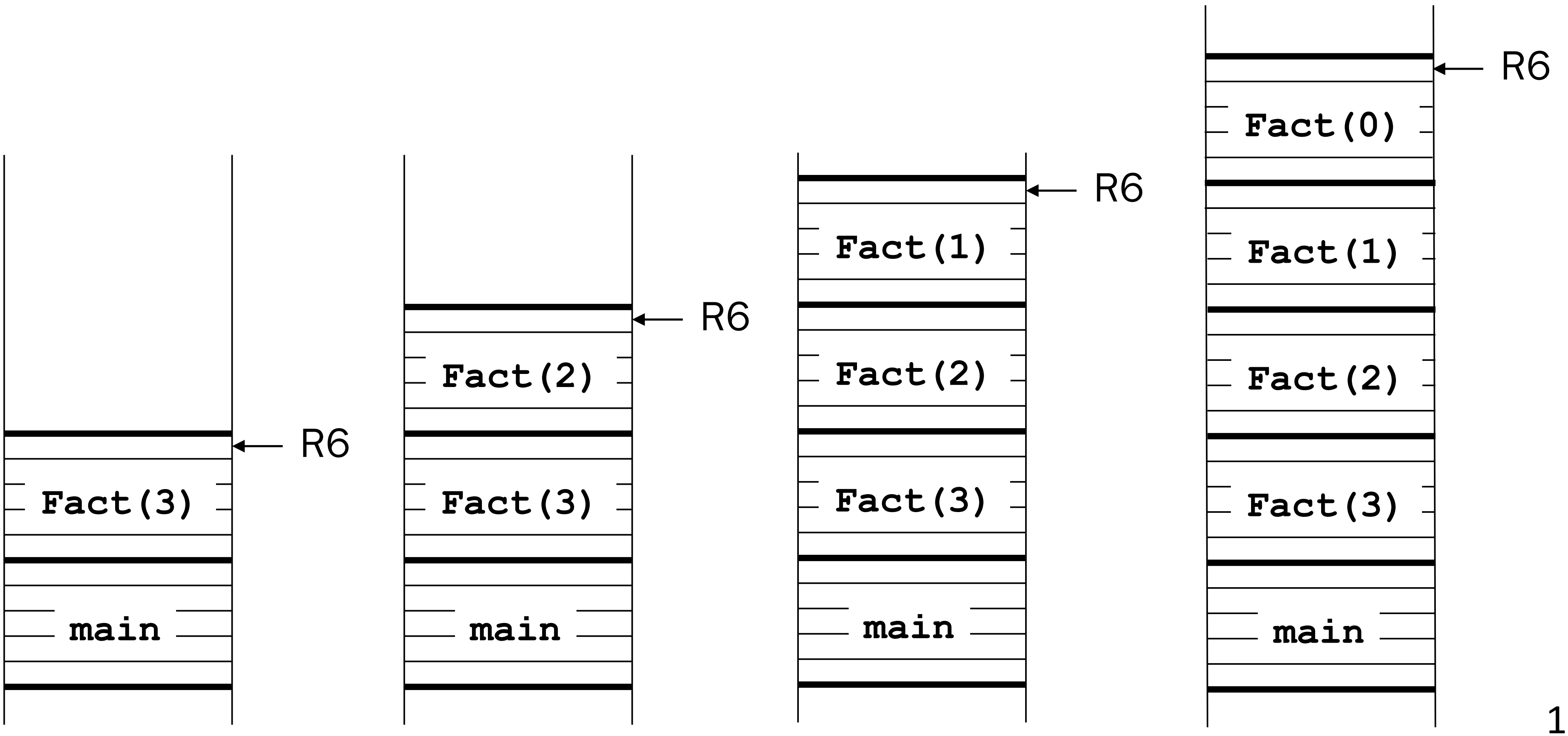
Run-Time Stack During Execution of Factorial

main calls Factorial(3)

Factorial(3) calls Factorial(2)

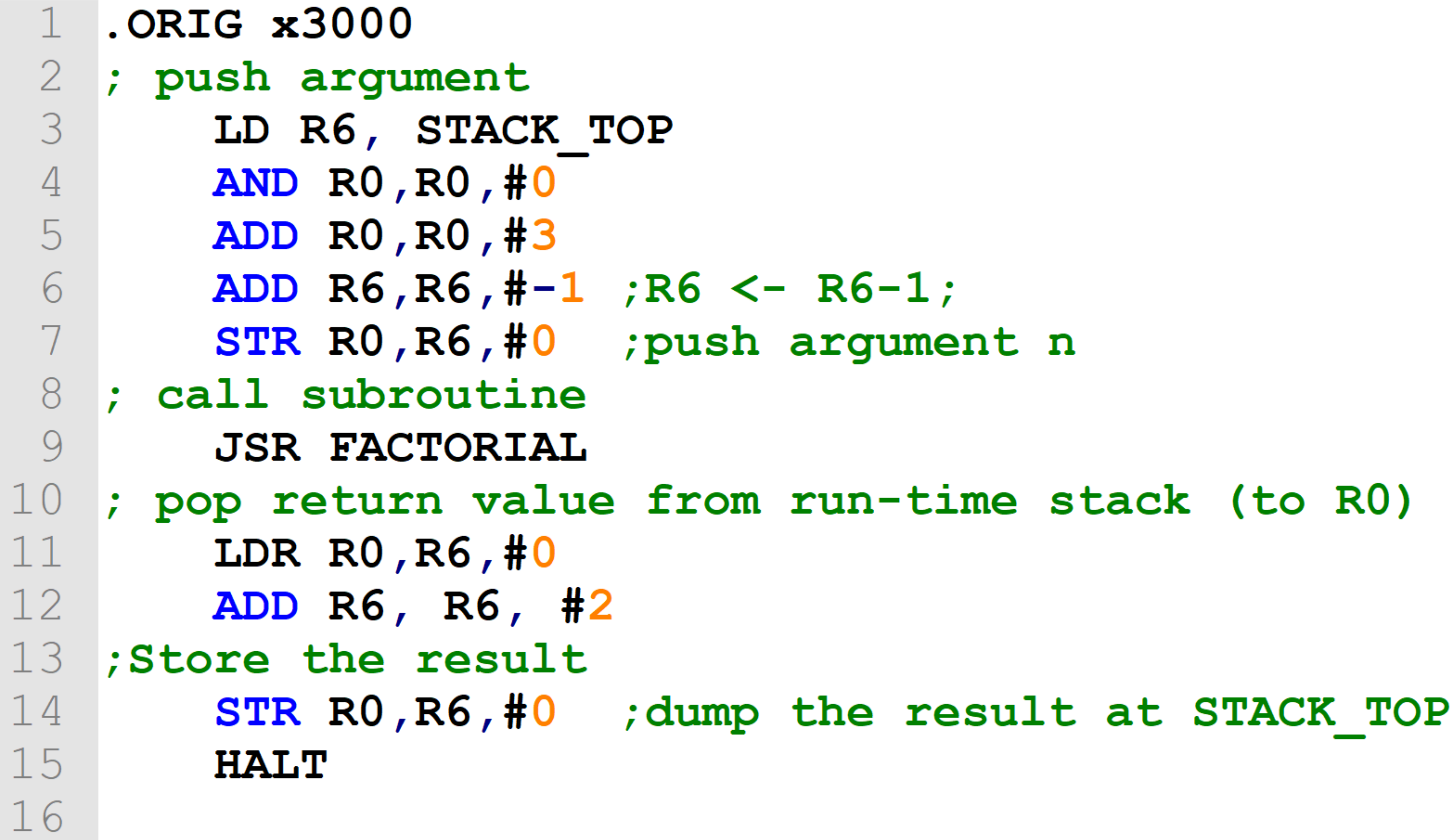
Factorial(2) calls Factorial(1)

Factorial(1) calls Factorial(0)

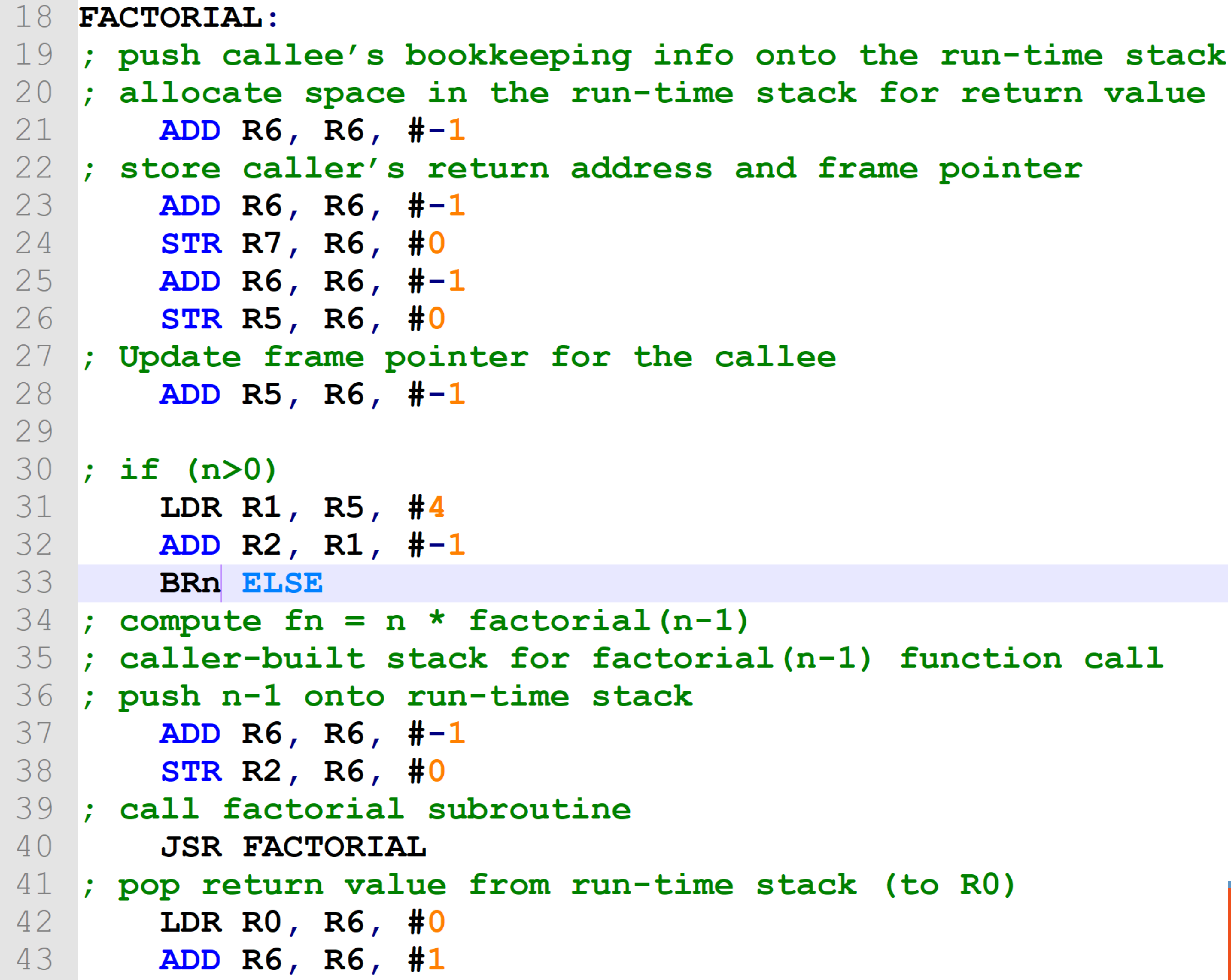


C to LC3 implementation of n!

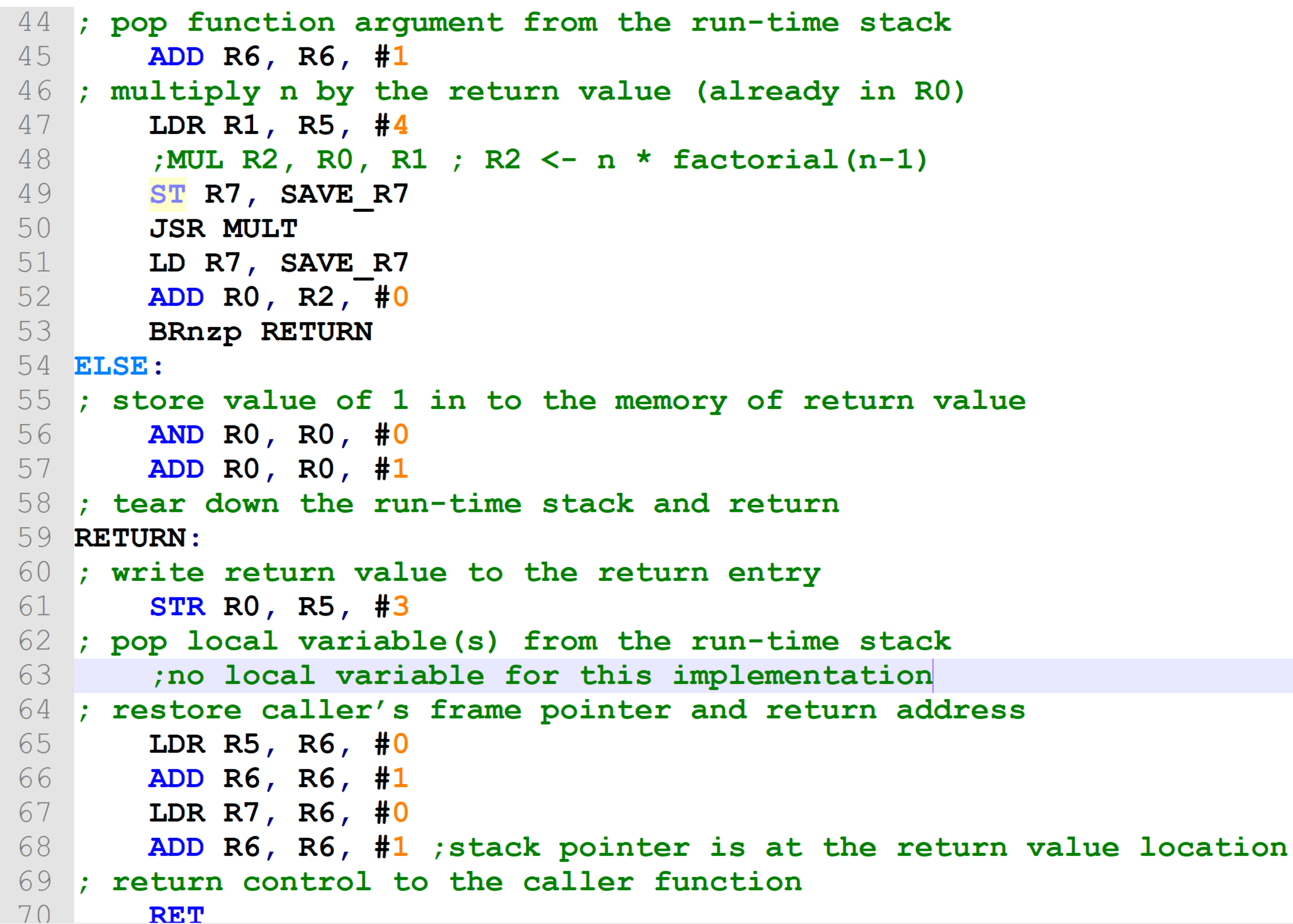
(test case n=3)



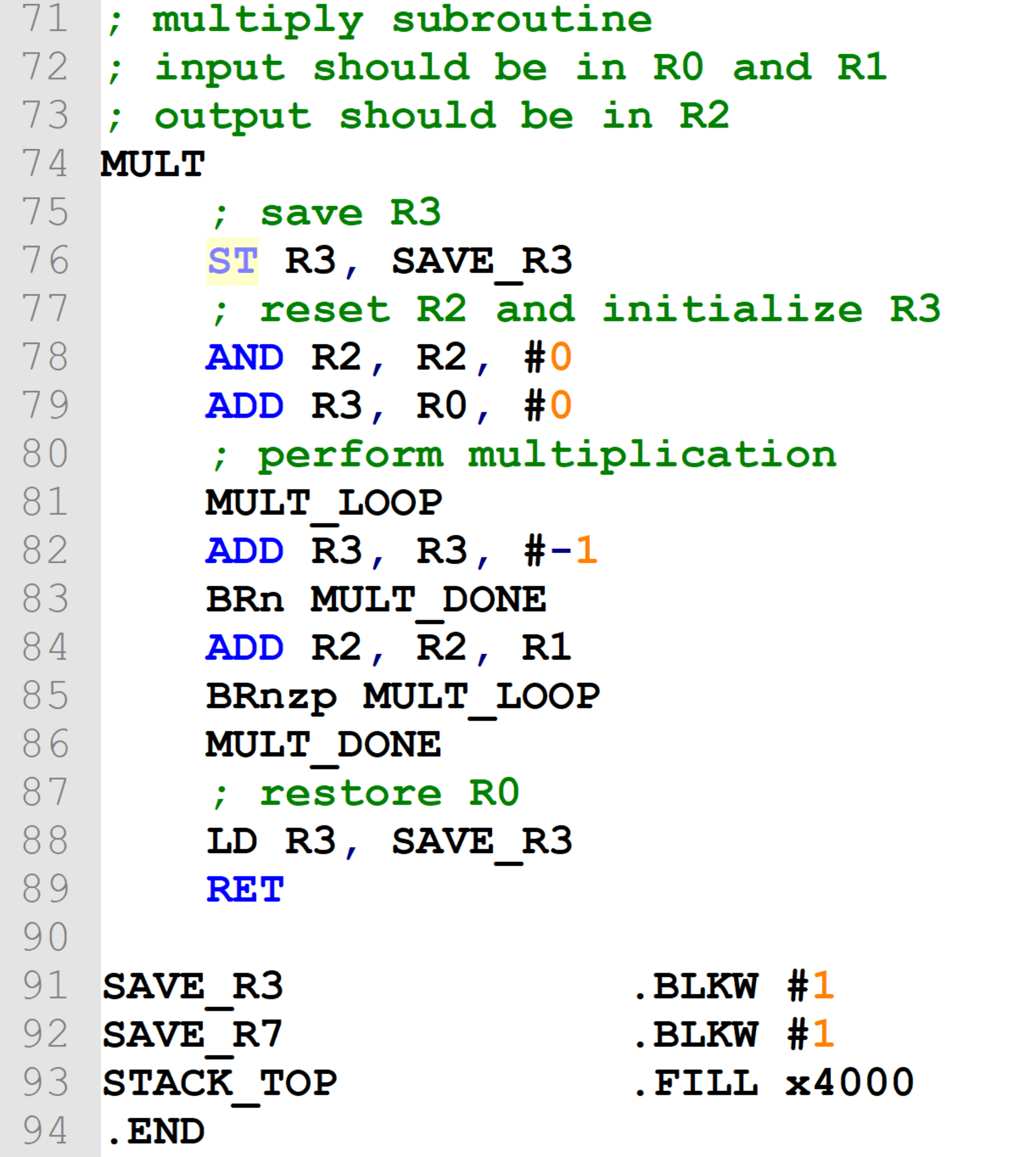
```
.ORIG x3000
; push argument
    LD R6, STACK_TOP
    AND R0,R0,#0
    ADD R0,R0,#3
    ADD R6,R6,#-1 ;R6 <- R6-1;
    STR R0,R6,#0  ;push argument n
; call subroutine
    JSR FACTORIAL
; pop return value from run-time stack (to R0)
    LDR R0,R6,#0
    ADD R6, R6, #2
;Store the result
    STR R0,R6,#0  ;dump the result at STACK_TOP
    HALT
```



```
FACTORIAL:
; push callee's bookkeeping info onto the run-time stack
; allocate space in the run-time stack for return value
    ADD R6, R6, #-1
; store caller's return address and frame pointer
    ADD R6, R6, #-1
    STR R7, R6, #0
    ADD R6, R6, #-1
    STR R5, R6, #0
; Update frame pointer for the callee
    ADD R5, R6, #-1

; if (n>0)
    LDR R1, R5, #4
    ADD R2, R1, #-1
    BRn ELSE
; compute fn = n * factorial(n-1)
; caller-built stack for factorial(n-1) function call
; push n-1 onto run-time stack
    ADD R6, R6, #-1
    STR R2, R6, #0
; call factorial subroutine
    JSR FACTORIAL
; pop return value from run-time stack (to R0)
    LDR R0, R6, #0
    ADD R6, R6, #1
```



```
; pop function argument from the run-time stack
    ADD R6, R6, #1
; multiply n by the return value (already in R0)
    LDR R1, R5, #4
    ;MUL R2, R0, R1 ; R2 <- n * factorial(n-1)
    ST R7, SAVE_R7
    JSR MULT
    LD R7, SAVE_R7
    ADD R0, R2, #0
    BRnzp RETURN
ELSE:
; store value of 1 in to the memory of return value
    AND R0, R0, #0
    ADD R0, R0, #1
; tear down the run-time stack and return
RETURN:
; write return value to the return entry
    STR R0, R5, #3
; pop local variable(s) from the run-time stack
    ;no local variable for this implementation
; restore caller's frame pointer and return address
    LDR R5, R6, #0
    ADD R6, R6, #1
    LDR R7, R6, #0
    ADD R6, R6, #1 ;stack pointer is at the return value location
; return control to the caller function
    RET
```



```
; multiply subroutine
; input should be in R0 and R1
; output should be in R2
MULT
    ; save R3
    ST R3, SAVE_R3
    ; reset R2 and initialize R3
    AND R2, R2, #0
    ADD R3, R0, #0
    ; perform multiplication
    MULT_LOOP
    ADD R3, R3, #-1
    BRn MULT_DONE
    ADD R2, R2, R1
    BRnzp MULT_LOOP
    MULT_DONE
    ; restore R0
    LD R3, SAVE_R3
    RET

SAVE_R3                 .BLKW #1
SAVE_R7                 .BLKW #1
STACK_TOP               .FILL x4000
.END
```

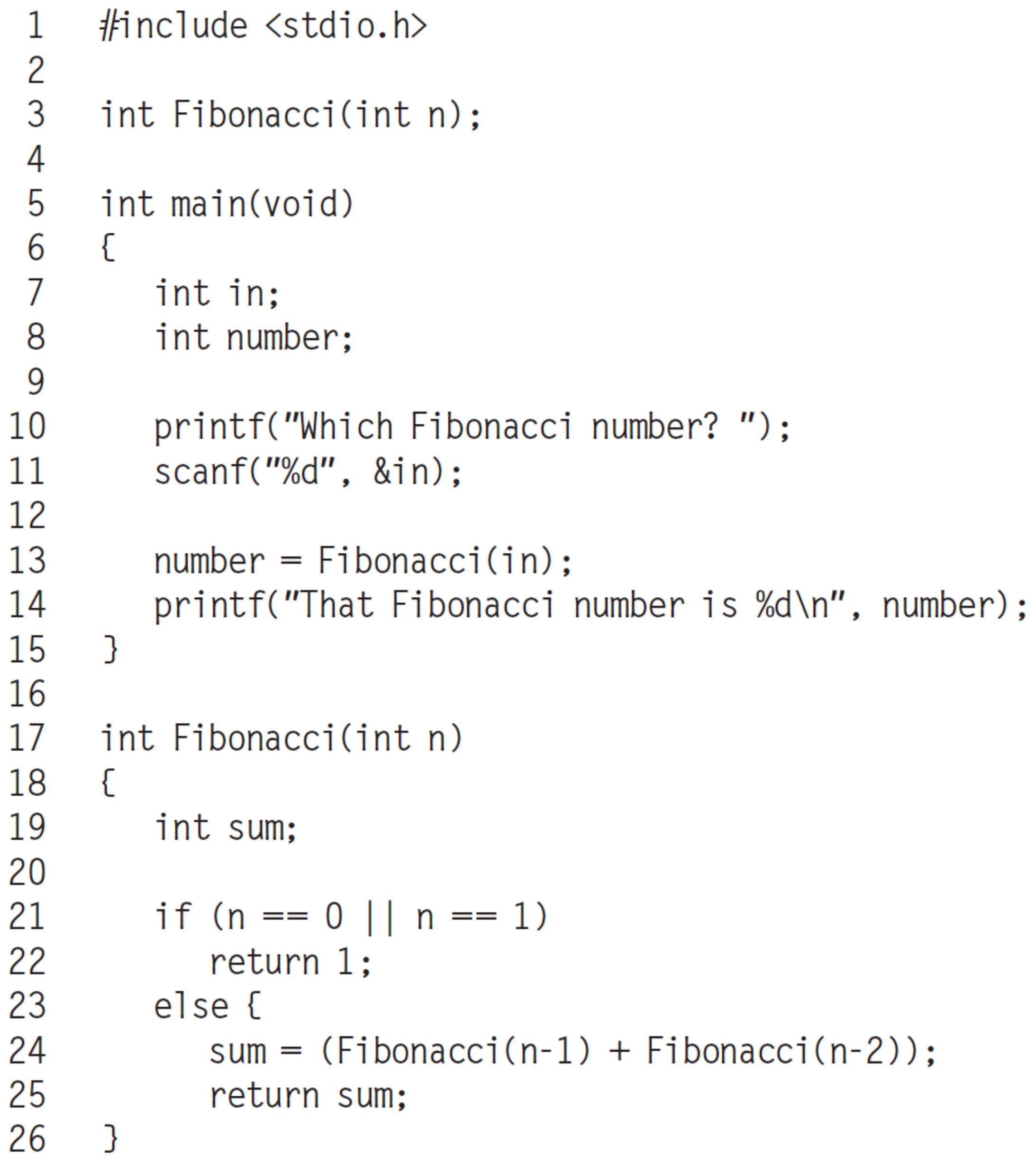
Recursive Binary Search

Fibonacci Series

$$f(n) = f(n-1) + f(n-2)$$

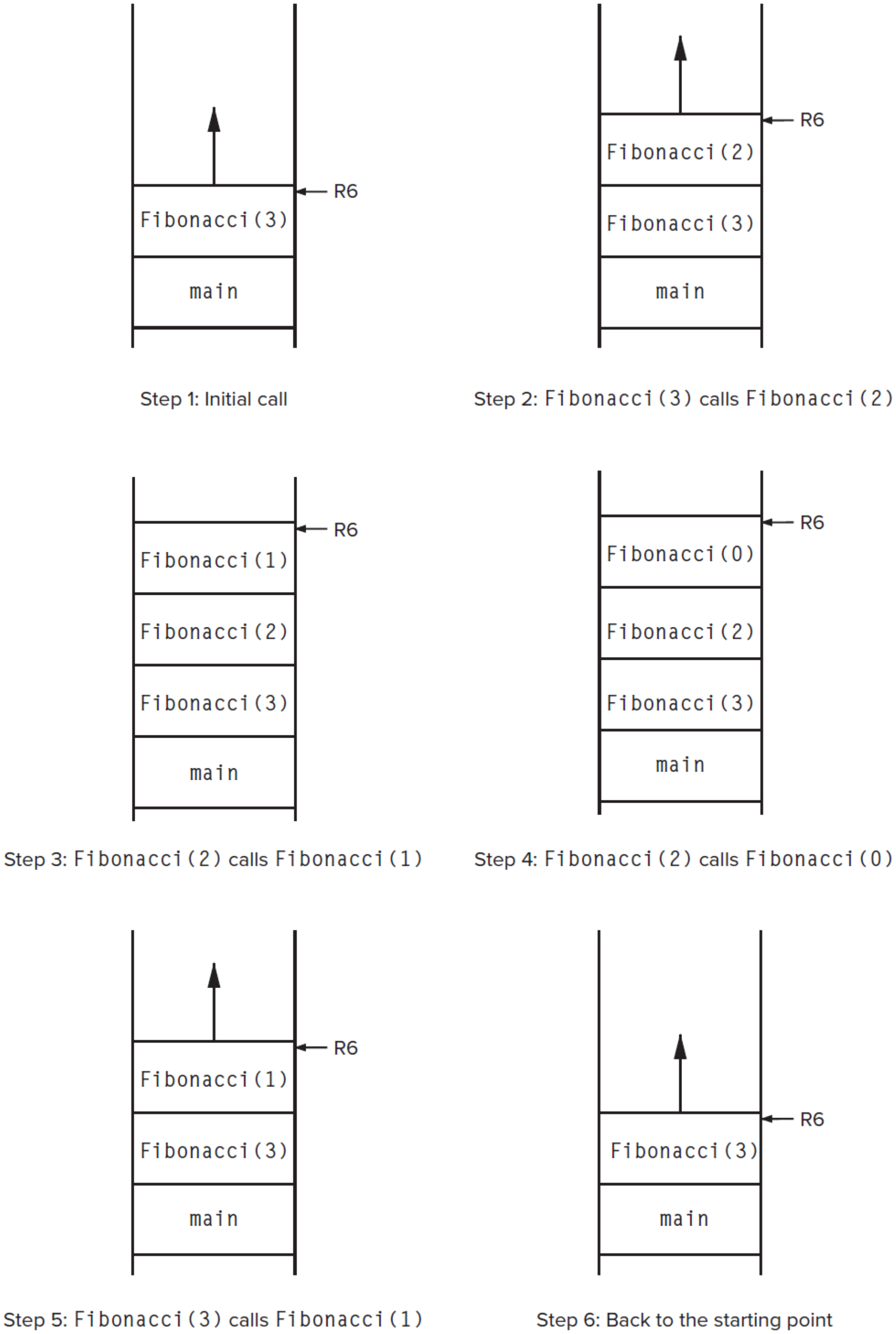
$$f(1) = 1$$

$$f(0) = 1$$



```c
#include <stdio.h>

int Fibonacci(int n);

int main(void)
{
    int in;
    int number;

    printf("Which Fibonacci number? ");
    scanf("%d", &in);

    number = Fibonacci(in);
    printf("That Fibonacci number is %d\n", number);
}

int Fibonacci(int n)
{
    int sum;

    if (n == 0 || n == 1)
        return 1;
    else {
        sum = (Fibonacci(n-1) + Fibonacci(n-2));
        return sum;
}
```

```
#include <stdio.h>

int Fibonacci(int n);

int main(void)
{
    int in;
    int number;

    printf("Which Fibonacci number? ");
    scanf("%d", &in);

    number = Fibonacci(in);
    printf("That Fibonacci number is %d\n", number);
}

int Fibonacci(int n)
{
    int sum;

    if (n == 0 || n == 1)
        return 1;
    else {
        sum = (Fibonacci(n-1) + Fibonacci(n-2));
        return sum;
}
```



Step 1: Initial call

Step 2: Fibonacci(3) calls Fibonacci(2)

Step 3: Fibonacci(2) calls Fibonacci(1)

Step 4: Fibonacci(2) calls Fibonacci(0)

Step 5: Fibonacci(3) calls Fibonacci(1)

Step 6: Back to the starting point

Consider, n=3