**3)**

Following is a trace for the input:

(define (fact n)

(if (= n 0) 1 (\* n (fact (- n 1)))))

(fact 3)

We start the tracing process *after* the environment is initialized (i.e. global environment contains primitive functions etc.)

a.

(define (fact n)

(if (= n 0) 1 (\* n (fact (- n 1)))))

1. The main eval method is called on the entire expression, where the predicate definition? returns true, so eval-definition is called
2. eval-definition calls define-variable!; for define-variable!, the first argument is evaluated by calling definition-variable on the expression, which returns the name of the function “fact”
3. The second argument of define-variable! is evaluated by calling eval on the definition-value of the expression.
4. The call of definition-value on the expression recognizes it is a function definition, and so it calls make-lambda on the list of fact parameters (just ‘(n)’) and the body of the function.
5. make-lambda creates a “lambda” list by concatenating “lambda” to the parameters and body.
6. Now after both first arguments of define-variable! (from step 2) were evaluated, define-variable! is called.
7. At the beginning of define-variable!, a local parameter frame is evaluated to first-frame of env, which is evaluated to the car of env.
8. Next, inside define-variable! a procedure “scan” is defined and then called on the frame-variables (which are evaluated to the car of the frame) and frame-values (evaluated to the cdr of the frame).
9. When scan is called, it recursively looks for “fact” in the list of vars of the first frame in the environment (which is initialized with #f, #t, cons etc.). Eventually, since it doesn’t find it, it calls add-binding-to-frame!
10. add-binding-to-frame! cons the function name (“fact”) to the list of vars of the first frame, and its mapping, the constructed “lambda” list to the frame’s values.
11. Then we go back to eval-definition from step 1 and ‘ok is returned to the main eval call and propagated up.

b.

(fact 3)

1. The main eval method is called on the entire expression, where the predicate application? returns true, so apply is called.
2. Before apply is called, its first argument is evaluated: eval is called on the operator of the expression (which is evaluated to “fact”).
3. Upon the call of eval on “fact”, the predicate variable? returns true and so lookup-variable-value is called on “fact”.
4. In lookup-variable-value, env-loop and scan are defined, and then env-loop is called. When it is called, first env is checked that it is not the empty-environment. Since it is not, it’s first frame is taken (like before) and scanned (like before). Since “fact” was added to the first frame as the new first variable, it is immediately found and the corresponding val is returned (the “lambda” list mapped from “fact”).
5. Back to step 2, now the second argument of apply is evaluated: list-of-values is called on the operands of the expression (which are evaluated to the list that contains only the number 3).
6. Upon the call to list-of-values, the first operand which is 3 is evaluated using the main eval method.
7. In the main eval method call on 3, the predicate self-evaluating? returns true, and so the integer 3 is returned.
8. Back in list-of-values, 3 is added to the list of evaluated operands, and since there are no more operands (as identified in the next recursive call), this list is returned.
9. Finally apply is called on the two evaluated arguments.
10. In the apply method, the predicate primitive-procedure? is evaluated on the “lambda” list, which is evaluated to false. Then the predicate compound-procedure? is evaluated,