CS550 \ Assignment 5

Group 1

Ariel Stolerman

Bekah Overdorf

Sam Snodgrass

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. Then the main eval method is called on the “lambda” list, where the predicate lambda? is evaluated to true, so make-procedure is called on the lambda-parameters (evaluated to the cadr of the “lambda” list) and the lambda-body (evaluated to the cddr of the “lambda” list).
7. When make-procedure is evaluated with these arguments, it returns a new list with “procedure”, the parameters, the body and the env.
8. Now after both first arguments of define-variable! (from step 2) were evaluated, define-variable! is called.

9. 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.
10. 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).
11. 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!
12. add-binding-to-frame! cons the function name (“fact”) to the list of vars of the first frame, and its mapping, the constructed “procedure” list to the frame’s values.
13. 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? Evaluates to 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? is evaluated to 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 “procedure” 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? is evaluated to 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 “procedure” list, which is evaluated to false. Then the predicate compound-procedure? is evaluated to true (since the list starts with the word “procedure”), and so eval-sequence is called. Before it is called, its arguments are evaluated.
11. The first argument of eval-sequence is evaluated to procedure-body on the procedure object, which is evaluated to the caddr of the procedure.
12. The second argument of eval-sequence is evaluated to extend-environment on the procedure-parameters (evaluated to the cadr of the procedure – the list containing “n”), the arguments (previously evaluated the list containing 3) and procedure-environment (evaluated to the cadddr of the procedure).
13. In the call to extend-environment, the length of vars (the procedure-parameters) and vals (the arguments) is equal, so make-frame is called (evaluated to a new cons cell of the vars and the vals – effectively binding “n” to 3) and cons’ed to the given environment (previously evaluated to the procedure’s environment) as the new first frame in that environment.
14. Now eval-sequence from step 10 is finally called on the procedure-body and the new environment evaluated above. Upon call, last-exp? on the procedure body is evaluated to true (since it contains only one expression with a recursive call), so the main eval method is called on the first-exp of the procedure body (which is the only one) and the given (extended) env.
    The expression evaluated is: “(if (= n 0) 1 (* n (fact (- n 1)))))” (where in the first frame of the env “n” is bound to 3)
15. In the main eval method, the predicate if? is evaluated to true, so eval-if is called.
16. In eval-if, the if-predicate (which is evaluated to (= n 0)) is passed on to the main eval method.
17. The main eval call on (= n 0) – application? Is evaluated to true, so the same trace of calling apply happens, only instead of applying a complex procedure, the primitive procedure “=” is applied (by fetching the procedure = from the environment, where it is bound to the symbol “=”). The expression, which is (= 3 0) is finally evaluated to false.
18. Back in eval-if, the predicate is false, so eval is called on the if-alternative (the else clause) which is “(* n (fact (- n 1)))”.
19. In short, the next is evaluated in a similar manner using recursive calls of eval on the sub-expressions. At the base case of fact, where n is bound to 0, finally the if-consequent in if-eval is evaluated to 1, and propagated up the recursion branch.

20. Finally, back in step 14, the value 6 is received, and propagated up.

4)

Following is an example of a code that provides two different answers depending on which scoping rule is used:

```
(define x 1)
(define (f x) (g 2))
(define (g y) (+ x y))
(f 5)
```

The answer in static scoping: 3

The answer in dynamic scoping: 7