Avishai Yaniv 307916023
Alex Abramov 314129438

1.1:

A special form as mentioned in previous assignment, is evaluated with its own rules while a primitive operator has a fixed behavior. For example, whenever we use add operation, the expression is translated and calculated by the following code:

- `proc.op === "+" ? reduce((x, y) => x + y, 0, args)`

And on the other hand, a special form, which includes its unique rules might use addition as an interior calculation, for example:

- `(lambda x y (* (+ x y) 3) 5 2)`

As we can see, the expression (+ x y) refers to the '+' primitive operation, while the whole lambda expression is evaluated in two steps, and in its own rules.

1.2:

The two ways presented in class of primitive operators representation are *VarRef* and *PrimOp*. The former defines primitive operations expressions as VarRefs with 'define', meaning that each operator's value is a Closure. using primitive operation is finding the operator from the environment and then applying the appropriate function. In contrast, the latter defines a special form for the primitive operations which is PrimOp, and each primitive operator's value is just a string which represents the appropriate operation. Using any kind of primitive operation in this case is just a call to a pre-defined code.
Examples:

- Scheme represents primitive operations using *PrimOp* method.
- TypeScript represents primitive operations using *VarRef* method.

1.3:

Equivalent program:

- `(+ 3 5)`

Both Applicative-Order and Norman-Order will apply this addition in the same way.
Unequivalent program:

- `(define h (lambda x ( (display x) (+ x x) ))`
- `(define g (lambda x (display 'hey)))`
- `(g (h 1))`

We distinguish the orders as in Applicative-Order the value (h 1) will be at first evaluated, 1 will be printed, then g will be called with the value 2, whilst Normal-Order will call g with the value (h 1). Since (h 1) will be calculated into 2 iff it is needed, there is no printing shows of 1. In conclusion, the first method prints x and hey while the second prints just hey.

1.4:

The role of *valueToLitExp* function in the substitution model is evaluating arguments into expressions to ensure the result of the substitution is a well-typed AST which can be evaluated.

1.5:

The function *valueToLitExp* is not needed in the environment interpreter since the values are already evaluated and there's no need of the AST and therefore valueToLitExp is not needed at this moment.

Avishai Yaniv 307916023
Alex Abramov 314129438

1.6:

Reasons to switch Applicative-Order to Normal-Order:

1. Normal-Order may prevent infinite loop that Applicative-Order has to calculate as prior calculation in some expressions.
2. Normal-Order may not need divide by zero when if the division expression is not needed for further calculations while Applicative-Order must have the expression evaluated..

An example:

- `(define h (lambda x (+ 1 1)))`
- `(h (/ 3 0))`

Normal-Order returns 2 while Applicative-Order throws an exception of ‘by 0-division’.

1.7:

While Normal-Order might avoid some unwanted calculations, there might be a repetition of expressions evaluation if an expression value is needed. For example:

- `(define h (lambda x (+ x x)))`
- `(h (*2 3))`

h will be called with (*2 3) as x and whenever x is needed the (* 2 3) expression will be evaluated into 6. In our case it happens twice.

1.8:

An approach which had been discussed in class is rewriting let expressions as lambda expressions. Regarding this approach, these let expressions will be evaluated using closures.

Code sample:

```
const rewriteLet = (e: LetExp): AppExp => {
    const vars = map((b) => b.var, e.bindings);
    const vals = map((b) => b.val, e.bindings);
    return makeAppExp(
            makeProcExp(vars, e.body),
            vals);
}
```

Another approach to deal with let expressions in the Environment Model is directly evaluating the expressions. Bindings’ values are calculated and being put in a new frame and the body is evaluated directly as mentioned.

Code sample:

```
// LET: Direct evaluation rule without syntax expansion
// compute the values, extend the env, eval the body.
const evalLet = (exp: LetExp, env: Env): Value | Error => {
    const vals: Value[] = map((v: CExp) => applicativeEval(v, env), map((b: Binding) => b.val, exp.bindings));
    const vars = map((b: Binding) => b.var.var, exp.bindings);
    if (hasNoError(vals)) {
        return evalExps(exp.body, makeExtEnv(vars, vals, env, env));
    } else {
        return Error(getErrorMessages(vals));
    }
}
```