Part I: Theoretical Questions

1.

As can be seen in the provided code, primitive operators are dispatched to the meta-language (TypeScript) based on their name, while special form (if in our example) expressions are not.

```
// @Pre: none of the args is an Error (checked in applyProcedure)
export const applyPrimitive = (proc: PrimOp, args: Value[]): Value | Error =>
    proc.op === "+" ? (allT(isNumber, args) ? reduce((x, y) => x + y, 0, args) : Error("+ expects numbers only")) :
    proc.op === "-" ? minusPrim(args) :
    proc.op === "*" ? (allT(isNumber, args) ? reduce((x, y) => x * y, 1, args) : Error("* expects numbers only")) :
    proc.op === "/" ? divPrim(args) :
    proc.op === ">" ? args[0] > args[1] :
    proc.op === "<" ? args[0] < args[1] :
    proc.op === "=" ? args[0] === args[1] :
    proc.op === "not" ? ! args[0] :
    proc.op === "and" ? isBoolean(args[0]) && isBoolean(args[1]) && args[0] && args[1] :
    proc.op === "or" ? isBoolean(args[0]) && isBoolean(args[1]) && (args[0] || args[1]) :
    proc.op === "eq?" ? eqPrim(args) :
    proc.op === "string=?" ? args[0] === args[1] :
    proc.op === "cons" ? consPrim(args[0], args[1]) :
    proc.op === "car" ? carPrim(args[0]) :
    proc.op === "cdr" ? cdrPrim(args[0]) :
    proc.op === "list" ? listPrim(args) :
    proc.op === "pair?" ? isPairPrim(args[0]) :
    proc.op === "number?" ? typeof(args[0]) === 'number' :
    proc.op === "boolean?" ? typeof(args[0]) === 'boolean' :
    proc.op === "symbol?" ? isSymbolSExp(args[0]) :
    proc.op === "string?" ? isString(args[0]) :
    Error("Bad primitive op " + proc.op);
const evalIf = (exp: IfExp, env: Env): Value | Error => {
    const test = L3applicativeEval(exp.test, env);
    return isError(test) ? test :
        isTrueValue(test) ? L3applicativeEval(exp.then, env) :          //notice how only the then or the alt are evaluated
        L3applicativeEval(exp.alt, env);
};
```

2.

In the regular form , primops are dispatched to the meta-language based on their name , the varref strategy means that primitives are variable references which refer to primitive procedures which are pre-defined in the global environment. That is we first need to initialize the Global Environment with Primitive Values and then we can change the applyPrimitive procedure to a shorter version of the function. The differences that we get is the order of evaluation of parameters in a procedure application - in some cases, the order of evaluation of operands in an application is not specified. This means that we can obtain different outputs if we insert side-effects as part of the operands.

3.

(define loop (lambda (x) (loop x)))

(define g (lambda (x y) y))

(g (loop 0) 1)

normal-eval of the code will return 1 while applicative-eval will get into infinite loop.

(define g (lambda (x y) y)

(g 0 1)

Both normal-eval and applicative-eval will return 1.

4.

The body of a closure is a list of CExp expressions. In the substitution model, our objective is to replace all VarRef occurrences in the body with the corresponding values of the arguments. There is a typing problem with this operation, if we replace VarRef with a value (a number for instance), the resulting body is not a valid AST. To address this discrepancy, we must map the values of the arguments to corresponding expressions. This mapping is performed in our interpreter with the valueToLitExp function.

5.

The function is not needed due to the fact that there is no substitution in the environment interpreter. The model just use varRef and new environments.

6.

A reason to use the normal evaluation instead of applicative is when we evaluate expressions that we don't necessarily need evaluation for the output and their evaluation returns error or entering an infinite loop.

(define loop (lambda (x) (loop x)))

(define g (lambda (x y) y))

(g (loop 0) 1)

In this case the correct output should be 1 and 1 will be the output only using normal-eval (applicative eval will calculate the loop function, thus entering an infinite loop)

7.

A reason is efficiency of the program, that applicative eval does not repeat on the same calculations while the normal eval does.

(define square (lambda (x) (* x x)))

(+ (+ (square 5) (square 5)) square(5))

in the normal eval the square will be calculated 3 times while the applicative eval only once.

8.

Let us examine two different strategies to evaluate a let expression.

In L3 interpreter, let expression is a syntactic abbreviation. The evaluation of the let form involves the creation of closure value and its application to the initial values.

(let [(<var1> <exp1>)
(<var2> <exp2>)
...
(<varn> <expn>)]
body>)

Abbreviates:

((lambda (<var1> ... <varn>) <body>)
<exp1> ... <expn>)

The rewrite process is done in the rewriteLet and rewriteAllLet functions in the rewrite.ts file (which should be referred to from the L3applicativeEval function in the L3-eval.ts file):

```
/*
Purpose: rewrite a single LetExp as a lambda-application form
Signature: rewriteLet(cexp)
Type: [LetExp => AppExp]
*/
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);
}

/*
Purpose: rewrite all occurrences of let in an expression to lambda-applications.
Signature: rewriteAllLet(exp)
Type: [Parsed -> Parsed]
*/
export const rewriteAllLet = (exp: Parsed | Error): Parsed | Error =>
  isError(exp) ? exp :
  isExp(exp) ? rewriteAllLetExp(exp) :
  isProgram(exp) ? makeProgram(map(rewriteAllLetExp, exp.exps)) :
  exp;

const rewriteAllLetExp = (exp: Exp): Exp =>
```

```
    isCExp(exp) ? rewriteAllLetCExp(exp) :
    isDefineExp(exp) ? makeDefineExp(exp.var, rewriteAllLetCExp(exp.val)) :
    exp;

const rewriteAllLetCExp = (exp: CExp): CExp =>
    isAtomicExp(exp) ? exp :
    isLitExp(exp) ? exp :
    isIfExp(exp) ? makeIfExp(rewriteAllLetCExp(exp.test),
                             rewriteAllLetCExp(exp.then),
                             rewriteAllLetCExp(exp.alt)) :
    isAppExp(exp) ? makeAppExp(rewriteAllLetCExp(exp.rator),
                               map(rewriteAllLetCExp, exp.rands)) :
    isProcExp(exp) ? makeProcExp(exp.args, map(rewriteAllLetCExp, exp.body)) :
    isLetExp(exp) ? rewriteAllLetCExp(rewriteLet(exp)) :
    exp;
```

in L4 interpreter, let expression is interpreted directly without the creation of a closure, as can be seen in the evalLet function in the L4-eval.ts file:

```
// 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 = 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));
    } else {
        return Error(getErrorMessages(vals));
    }
}
```