PPL – Assignment 2

Ahiad Shaiker & Nitzan Guetta

1.1: A special form is an expression which has a special evaluation rule.
For example: Conditionals like if and cond, definitions (define), bindings like let, lambdas, etc.

1.2: Atomic Expressions: expression that it's computation rule for evaluating is to lookup its value in the current state of the computation. Atomic expression doesn't have any sub-expression.
for example: number expressions: 1, string expressions: "abc", boolean expressions: true.

1.3: Compound Expressions: made up of sub-expressions according to the syntax of the language. For example the expression 12 >= 7 is a compound expression made up of 3 sub-expressions, or (12 === 13) ? -1 : 2 is also a compound expression (called a conditional expression).

1.4: Primitives expressions evaluation is built in in the interpreter and which are not explained by the semantics of the language. These include primitive operations (for example, arithmetic operations on number or comparison operators) and primitive literal values (for examples, numbers or boolean values).

1.5.1: Atomic and primitive

1.5.2:Atomic and primitive

1.5.3: Atomic

1.5.4: Compound

1.6: Side effect

1.7: a certain expression and another expression are semantically equivalent

1.8:

```
((lambda (x y z) (* (x z) y))
 (lambda (x) (+ x 1))
 ((lambda (y) (- y 22)) 23)
 6)
```

1.9: Yes. And will return the first argument which is false. It won't evaluate the rest of arguments.
Proof:

scheme behavior for this: (and #f (display 12)) ,
will be: #f .

(and #f (display 12)) =>#f

scheme behavior for this: (and #t (display 12)),
will be: 12 .

(and #t (display 12)) => 12

1.10.1: according to definition of functional equivalence that was shown in class, foo and goo are functionally equivalent.
Foo evaluated to a value and goo is evaluated to the same value.

1.10.2: foo and goo are not goo functionally equivalent according to definition that was shown in class in addition when considering side effects. Goo prints "hi-there" while foo doesn't.

2.1:

Evaluate ((define x 12)) [compound special form]
    Evaluate (12) [atomic]
    Return value: 12
    Add the binding <<x>,12> to GE
Return value: void
Evaluate (((lambda (x) (+ x (+ (/ x 2) x))) x)) [compound special form]
    Evaluate ((lambda (x) (+ x (+ (/ x 2) x))) [compound special form]
    Return value: <Closure,(x), (+ x (+ (/ x 2) x))>
    Evaluate (x) [atomic]
    Return value: 12 (GE)
    Replace x with 12
    Evaluation: ((+ 12 (+ (/ 12 2) 12))) [compound non special form]
        Evaluation(+) [atomic]
        Return value: # <procedure:+>
        Evaluation(12) [atomic]
        Return value: 12
        Evaluation ((+ (/ 12 2) 12)) [compound non special form]
            Evaluation(+) [atomic]
            Return value: # <procedure:+>
            Evaluation ((/ 12 2)) [compound non special form]
                Evaluation(/) [atomic]
                Return value: # <procedure:/>
                Evaluation(12) [atomic]
                Return value: 12
                Evaluation(2) [atomic]
                Return value: 2
            Return value: 6
            Evaluation(12) [atomic]
            Return value: 12
        Return value: 18
    Return value: 30
Return value: 30

2.2:

Evaluate ((define last (lambda (l) (if (empty? (cdr l)) (car l) (last (cdr l)))))) [compound special form]
    Evaluate ((lambda (l) (if (empty? (cdr l)) (car l) (last (cdr l))))) [compound special form]
    Return value: <Closure ,(l), (if (empty? (cdr l)) (car l) (last (cdr l)))>
    Add binding <(last), <Closure ,(l), (if (empty? (cdr l)) (car l) (last (cdr l)))>> to GE
Return value: void

2.3:

Evaluate ((define last (lambda (l) (if (empty? (cdr l)) (car l) (last (cdr l)))))) [compound special form]
    Evaluate ((lambda (l) (if (empty? (cdr l)) (car l) (last (cdr l))))) [compound special form]
    Return value: <Closure ,(l), (if (empty? (cdr l)) (car l) (last (cdr l)))>
    Add binding <(last), <Closure ,(l), (if (empty? (cdr l)) (car l) (last (cdr l)))>> to GE
Return value: void
Evaluate ((last '(1 2)))[compound non special form]
    Evaluate (last) [atomic]
    Return value: <Closure ,(l), (if (empty? (cdr l)) (car l) (last (cdr l)))> [GE]
    Evaluate ('(1 2))
    Return value : '(1 2)
    Replace l with '(1 2): (if (empty? (cdr '(1 2))) (car '(1 2)) (last (cdr '(1 2))))
    Evaluate ((if (empty? (cdr '(1 2))) (car '(1 2)) (last (cdr '(1 2))))) [compound special form]
        Evaluate: ((empty? (cdr '(1 2))) [compound non-special form]
            Evaluate (empty?) [atomic]
            Return value: #<procedure:empty?>
            Evaluate ((cdr '(1 2))) [compound non-special form]
                Evaluate: (cdr) [atomic]
                Return value: #<procedure:cdr>
                Evaluate: ('(1 2)) [compound literal expression]
                Return value: '(1 2)
            Return value: '(2)
        Return value: #f
        Evaluate: ((last (cdr '(1 2)))) [compound non special]
            Evaluate: (last) [atomic]
            Return value: <Closure ,(l), (if (empty? (cdr l)) (car l) (last (cdr l)))> (GE)
            Evaluate ((cdr '(1 2))) [compound non-special form]
                Evaluate: (cdr) [atomic]
                Return value: #<procedure:cdr>
                Evaluate: ('(1 2)) [compound literal expression]
                Return value: '(1 2)
            Return value: '(2)
        Replace l with '(2): (if (empty? (cdr '(2))) (car '(2)) (last (cdr '(2))))
        Evaluate ((if (empty? (cdr '(2))) (car '(2)) (last (cdr '(2))))) [compound special form]
            Evaluate: ((empty? (cdr '(2))) [compound non-special form]
                Evaluate (empty?) [atomic]
                Return value: #<procedure:empty?>
                Evaluate ((cdr '(2))) [compound non-special form]
                    Evaluate: (cdr) [atomic]
                    Return value: #<procedure:cdr>
                    Evaluate: ('(2)) [compound literal expression]
                    Return value: '(2)
                Return value: '()
            Return value: #t
            Evaluate: (car '(2)) [compound non-special form]
                Evaluate: (car) [atomic]
                Return value: #<procedure:car>
                Evaluate: ('(2)) [compound literal expression]
                Return value: '(2)
            Return value: 2
        Return value: 2
    Return value: 2
Return value: 2

3.1:

Binding Instance	Appears first at line	Scope	Line #s of bound occurrences
Fib	1	Universal Scope	4,6
n	1	Lambda Body(1)	2,3,4
y	5	Universal Scope	6

Free occurences: =,+,-.

3.2:

Binding Instance	Appears first at line	Scope	Line #s of bound occurrences
Triple	1	Universal Scope	4
x	1	Lambda Body(1)	3
y	2	Lambda Body(2)	3
z	3	Lambda Body(3)	3

Free occurences: +.

5.1: The added line is marked yellow.

```
/*
;; =============================================================================
;; Scheme Parser
;;
;; L2 extends L1 with support for IfExp and ProcExp
;; L3 extends L2 with support for:
;; - Pair and List datatypes
;; - Compound literal expressions denoted with quote
;; - Primitives: cons, car, cdr, list?
;; - The empty-list literal expression
;; - The Let abbreviation is also supported.

;; <program> ::= (L3 <exp>+) // Program(exps:List(Exp))
;; <exp> ::= <define> | <cexp>            / DefExp | CExp
;; <define> ::= ( define <var> <cexp> )    / DefExp(var:VarDecl, val:CExp)
;; <var> ::= <identifier>                / VarRef(var:string)
;; <cexp> ::= <number>                    / NumExp(val:number)
;;        | <boolean>                  / BoolExp(val:boolean)
;;        | <string>                 / StrExp(val:string)
;;        | ( lambda ( <var>* ) <cexp>+ ) / ProcExp(params:VarDecl[], body:CExp[]))
;;        | ( if <cexp> <cexp> <cexp> )  / IfExp(test: CExp, then: CExp, alt: CExp)
;;        | ( let ( binding* ) <cexp>+ )  / LetExp(bindings:Binding[], body:CExp[]))
;;        | (let* ( binding* ) <cexp>+ )  / LetStarExp(bindings:Binding[], body:CExp[]))
;;        | ( quote <sexp> )           / LitExp(val:SExp)
;;        | ( <cexp> <cexp>* )          / AppExp(operator:CExp, operands:CExp[]))
;; <binding>  ::= ( <var> <cexp> )         / Binding(var:VarDecl, val:Cexp)
;; <prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
;;              | cons | car | cdr | list? | number?
;;              | boolean? | symbol? | string?     ##### L3
;; <num-exp>  ::= a number token
;; <bool-exp> ::= #t | #f
;; <var-ref>  ::= an identifier token
;; <var-decl> ::= an identifier token
;; <sexp>     ::= symbol | number | bool | string | ( <sexp>* )           ##### L3
*/
```