Assignment 2

Eden Abadi: 305554917

Noam Weiler: 203570619

Part 1 – Theoretical questions

1.1 it is a compound expression which is not evaluated like regular (application) compound expressions.
1.2 Atomic expressions are expressions that do not consist of other sub-expressions.
1.3 Compound expressions are expressions that contain nested expressions.
1.4 Primitives get their values from the programming language.
1.5
1.5.1 Atomic and Primitive
1.5.2 Atomic and Primitive
1.5.3 Atomic
1.5.4 Primitive and Compound
1.6 multiple expressions in the body of a procedure expression (lambda form) is useful mainly when those expressions have side effects.
1.7 we call an expression a "syntactic abbreviation" of another expression when implementing a rewrite mechanism which turns all occurrences of a specific syntactic construct into a semantically equivalent syntactic structure.
1.8 ((lambda (x y z) (* (x z) y)) (lambda (x) (+ x 1)) ((lambda (y) (- y 22)) 23) 6)
1.9 It does, we will give a simple example: if run the code (and #f (display 'hi), racket will print #false, meaning it didn't get to the display, although if we run (and #t (display 'hi)) it prints "hi" .
1.10
1.10.1 Yes they are, according to the formal definition: f and g (pure computational functions in the FP paradigm) are equivalent iff whenever f(x) is evaluated to a value, g(x) is evaluated to the same value, if f(x)f(x) throws an exception, so does g(x), and if f(x) does not terminate, so does g(x). We will split to cases:
X is number: both foo and goo will return the value of x + 1 and therefore the conditions preserved.
X is not a number: both functions will fail when they will try to evaluate + x1 – the condition is preserved.
Hence, although "goo" displays text to the screen their result will always be the same for a valid x and will fail for an invalid x, also we can assume display will never fail so the terms are satisfied.
1.10.2 No they are not, that's because goo has side effects - display '(hi-there) for example if we call both functions with the argument 2, foo will return 3 and goo will display "hi there" and return 3.Therefore when considering side effects as differences between functions equivalents those functions foo and goo are not functionally equivalent..

Part 2 Rules of evaluation

2.1

$evaluate\big((define\ x\ 12)\big)[compound\ special\ form]]$

$evaluate(12)[atomic]$

$return\ value: 12$

$add\ the\ binding \ll x >, 12 > to\ the\ GE$

$return\ value: void$

$evaluate\ ((lambda\ (x)\ (+\ x\ \ (+\ (\ /\ x\ 2)\ x)\ )\ x))\ [compound\ non\ special\ form]$

$evaluate\ (lambda\ (x)\ \ (+\ \ x\ \ (+\ \ (/\ x\ 2)\ \ x))\ [compound\ special\ form]$

$return\ value: < clousre\ (x)\ (+\ \ x\ \ (+\ \ (/\ x\ 2)\ x) >$

$evaluate\ \ (x)$ [atomic]

$return\ value: 12\ [GE]$

$replace\ (x)\ \ with\ \ (12): (+\ \ 12\ \ (+\ \ (/\ \ 12\ \ 2)\ 12)$

$evaluate\ \ (+\ \ 12\ \ (+\ \ (/\ \ 12\ \ 2)\ 12)\ [compound\ non\ special\ form]$

$evaluate(+)[atomic]$

$return\ value: < procedure: +>$

$evaluate\ (12)\ [atomic]$

$return\ value: 12$

$evaluate\ (+\ \ (/\ \ 12\ \ 2)\ \ \ 12)[compound\ non\ special\ form]$

$evaluate\ (+)[atomic]$

$return\ value: < procedure: +>$

$evaluate\ (/\ 12\ \ 2)\ [compound\ non\ special\ form]$

$evaluate(/)[atomic]$

$return\ value: < procedure: />$

$evaluate\ (12)[atomic]$

$return\ value: 12$

$evaluate\ (2)[atomic]$

$return\ value: 2$

$return\ value: 6$

$evaluate(12)[atomic]$

$return\ value: 12$

$return\ value: 18$

$return\ value: 30$

$return\ value: 30$

2.2

$evaluate\ (define\ last$

$(lambda\ (l)$

$(if\ (empty?\ (cdr\ l))$

$(car\ l)$

$(last\ (cdr\ l)))))\ [compound\ special\ form]$

$evaluate\ (lambda\ (l)$

$if\big(empty?\ \ (cdr\ \ l)\big)$

$(car\ \ l)$

$\big(last\ \ (cdr\ \ l)\big)))\ [compound\ special\ form]$

$return\ value: < clousre(l)\ \ (if\big(empty?\ \ (cdr\ \ l)\big)\ (car\ \ l)\ (last\ \ (cdr\ \ l))))) >$

$add\ binding \ll last >, < clousre(l)\ \ (if\big(empty?\ \ (cdr\ \ l)\big)\ (car\ \ l)\ (last\ \ (cdr\ \ l)))))$
$> to\ the\ GE$

$return\ value: void$

2.3

$evaluate\ (define\ last$

$(lambda\ (l)$

$(if\ (empty?\ (cdr\ l))$

$(car\ l)$

$(last\ (cdr\ l)))))$

$\big(last\ '(1\ 2)\big)\ [compound\ non\ special\ form]$

$evaluate\ (define\ last\ (lambda\ (l)$

$(if\ \ (empty?\ \ (cdr\ l))$

$(car\ \ l)$

$(last\ \ (cdr\ \ l))))\ \ [compound\ special\ form]$

$return\ value: < clousre\ (l)\ \ \left(if\ \ \left(empty?\ \ (cdr\ l)\right)(car\ \ l)\ \ \left(last\ \ (cdr\ \ l)\right)\right) >$

$add\ binding \ll last >,$

$< clousre\ (l)\ \ \left(if\ \ \left(empty?\ \ (cdr\ l)\right)(car\ \ l)\ \ \left(last\ \ (cdr\ \ l)\right)\right)$

$> to\ the\ GE$

$evaluate\ \left(last\ \ '(1\ 2)\right)\ [compound\ non\ special\ form]$

$evaluate\ (last)\ [atomic]$

$return\ value: < clousre\ \ (l)\ \ (if\left(empty?\ \ (cdr\ \ l)\right)(car\ \ l)\ (last\ \ (cdr\ \ l))))) >$

$evaluate\ \ \left('(1\ \ 2)\right)\ [compound\ literal]$

$return\ value:\ '(1\ \ 2)$

$replace\ (l)\ with\ \ '(1\ \ 2)$

$evaluate\ \left(if\ \ \left(empty?\ \ (cdr\ \ '(1\ \ 2)\right)\left(car\ '(1\ \ 2)\right)\ \left(last\ \ (cdr\ \ '(1\ \ 2))\right)\right)\right)$

$[compound\ special\ form]$

$evaluate\ \left(empty?\ \ (cdr\ \ '(1\ \ 2))\right)\ [compound\ non\ special\ form]$

$evaluate\ \ (empty?)\ [atomic]$

$return\ value:\ \ < procedure: empty? >$

$evaluate\ \left(cdr\ '(1\ \ 2)\right)\ [compound\ non\ special\ form]$

$evaluate\ \ (cdr)\ [atomic]$

$return\ value: < procedure: cdr >$

$evaluate\ \left('(1\ \ 2)\right)\ [compound\ literal\ expression]$

$return\ value: '(1\ \ 2)$

$return\ value:\ '(2)$

$return\ value: \#f$

$evaluate\ \ \left(last\ \ (cdr\ \ '(1\ \ 2))\right)[compound\ non\ special\ form]$

$evaluate\ (last)\ [atomic]$

$return\ value:$

$< clousre\ \ (l)\ \ (if\left(empty?\ \ (cdr\ \ l)\right)(car\ \ l)\ (last\ \ (cdr\ \ l)))))>$

$evaluate\ \ \left(cdr\ \ '(1\ \ 2)\right)[compound\ non\ special\ form]$

$evaluate\ \ (cdr)\ [atomic]$

$return\ value: < procedure: cdr >$

$evaluate\ \left('(1\ \ 2)\right)\ [compound\ literal\ expression]$

$return\ value: '(1\ \ 2)$

$return\ value: '(2)$

$replace\ (l)\ with\ '(2)$

$evaluate\ \ (if\ \ \left(empty?\ \ \left(cdr\ \ '(2)\right)\ \left(car\ \ '(2)\right)\ \left(last\ \ \left(cdr\ \ '(2)\right)\right)\right)$

$[compound\ special\ form]$

$evaluate\ \ \left(empty?\ \ \left(cdr\ \ '(2)\right)\right)\ [compound\ non\ special\ form]$

$evaluate\ \ (empty?\,)\ [atomic]$

$return\ value:\ \ < procedure: empty? >$

$evaluate\ \ \left(cdr\ \ '(2)\right)\ [compound\ non\ special\ form]$

$evaluate\ \ (cdr)\ [atomic]$

$return\ value:\ \ < procedure: cdr >$

$evaluate\ \ \left('(2)\right)\ [compound\ literal\ expression]$

$return\ value:\ \ '()$

$return\ value:\ \ \#t$

$evaluate\ \ \left(car\ \ '(2)\right)\ [compound\ non\ special\ form]$

$evaluate\ (car)\ [atomic]$

$return\ value:\ \ < procedure:\ \ car >$

$evaluate\ \ \left('(2)\right)[compound\ literal\ expression]$

$return\ value: '(2)$

$return\ value:\ \ 2$

$return\ value:\ \ 2$

$return\ value:\ \ 2$

$return\ value:\ \ 2$

Part 3 Scopes:

3.1

Free variables occurrences: -, +, =

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

3.2

Free variables occurrences: +

Binding instance	Appears first at line	Scope	Line #s 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

Part 5 BNF:

5.1 We are going to extend the L3 BNF to support let*:

```
 <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>* )
```