**Assignment 2**

**Eden Abadi: 305554917**

**Noam Weiler: *203570619***

**Part 1 – Theoretical questions**

* 1. it is a compound expression which is not evaluated like regular (application) compound expressions.
  2. Atomic expressions are expressions that do not consist of other sub-expressions.
  3. Compound expressions are expressions that contain nested expressions.
  4. Primitives get their values from the programming language.
  5. 1. Primitive
     2. Atomic
     3. None
     4. Compound
  6. multiple expressions in the body of a procedure expression (lambda form) is useful mainly when those expressions have a mutation or printing.
  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.
  8. )(lambda (z) (\* (+ z 1) 1)) 6) ----- not sure!
  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” .
  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). 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 term are satisfied.
      2. No they are not, that’s because goo has side effects - display ‘(hi-there).there for 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

[atomic]

2.2

2.3

**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>\* )