Principles of Programming Languages 202
Assignment 2

Ben Gindi - 205874142
Shira Segev - 208825349

Question 1: Theoretical Questions

Q1.1

At the examples below, we are relying on the following assumptions:

- Primitive- refers to something that is defined in the language and the interpreter knows how to read. For example: #t, 1, < etc.
- Non-Primitive- refers to something that is NOT defined in the language and the user defines herself. For example: (list 1 2 3), or any expression/ value that is not defined in the language.
- Atomic- does not consist of any other expression.
- Compound expressions- are expressions that contain nested expressions.
- The differences between expression and value- expression is a component in the language of L3, like (+1 2). A value is a result of a computation of an expression.

- Primitive atomic expression example: (in bold)

  Numbers

  ```
  > 3
  3
  > 3.75
  3.75
  ```

  Booleans

  ```
  > #t
  #t
  > #f
  #f
  ```

- Non-primitive atomic expression example: x (defined by the progammer)
- Non-primitive compound expression example: (in bold)

  ```
  > (- 25 5)
  20
  > (number? 3.75)
  #t
  ```

- Primitive atomic value example: (in bold)

  Numbers

  ```
  > 3
  3
  > 3.75
  3.75
  ```

  Booleans

  ```
  > #t
  #t
  > #f
  #f
  ```

- Non-primitive atomic value example: The symbol '`blue` (or `enum BLUE`).
  As mentioned above, the symbol blue (as an example) is not primitive because it doesn't exists in L3, but it is Atomic, because it does not compound of few parts. We also learned, that from the programmer perspective, a closure is an atomic value.
- Non-primitive compound value example: `(list 1 2)`

Q1.2

Special form: Is a compound expression which is not evaluated like regular (application) compound expressions.
Example: `(if (= 3 3) 12 27)` (same for define, for, ext)

Q1.3

Free variable: A variable x occurs free in an expression E if and only if:

- there is a ref to x in E
- x does not bound by any declaration in E.

Example: y occurs free in `(define (x) (+ x y))`

Q1.4

Symbolic-Expression (s-exp): Also called datum, is an hierarchical structures (tree) of values (tokens). It plays in Scheme the same role as the JSON notation in JavaScript - it corresponds to the external notation for values that can be read by the read primitive procedure and manipulated as a value.
Example: The corresponding s-exp of the string `(< (+ 3 4) (/ 10 2))`, will be
⇒ `['<', ['+', '3','4'], ['/', '10', '2']]`

Q1.5

Syntactic abbreviation: As we saw on practical session no. 4:

- `cond` is a syntactic abbreviation of `if`
- `let` is a syntactic abbreviation of `lambda`

This means that when we define the operational semantics of the language, we do not need to define a new computation rule for this (cond \ let) expression type, instead we indicate that this expression is equivalent to a combination of other syntactic constructs that mean the same thing.

Syntactic abbreviation is a solution for decreasing the load on the interpreter, in a way, it makes the language “lighter”, because it consists of less expressions.
Examples:

1.
```
(cond (((> x 3) 4)
((> y 8) 5)
(else 6))
```
⇒
```
(if (> x 3)
4
(if (< y 8)
5
6))
```

2.
```
(let
((a 1) (b 2))
(*a b))
```
⇒
```
(
(lambda (a b)
(* a b))
1 2
)
```

Q1.6

We claim that **there isn't** a program in L3 which cannot be transformed to an equivalent program in L30 under the consideration described at the question.
Explanation- even without the word "list" or list as a literal, we can still create lists using **cons** (It is possible because car & cdr are part of pair, and combining with cons, we can create a list in a recursive way). That way, by changing this only at every program which uses a list, we can still create on.
In addition, since L3 is an extension to L2, while L2 doesn't include the option of list, we could still run any program we wished, so we can also say that L2 is a subset of L30.
For example: `list (0 1 2) => (cons 0 (cons 1 (cons 2 '())))`

Q1.7

PrimOp- Advantage: Given a primitive operator, by checking if it is a PrimOp, we can get its' value **without accessing the environment**.
Closure- Advantage: very efficient because there is **no** need for interpreter modification for **new defined** primitive operations (instead, we will use the existing operations such as lambdas, varRefs ect.).

Q1.8

In class, we implemented map in L3, where the given procedure is applied on the first item of the given list, then on the second item, and so on.
In this paragraph, we would like to examine whether another implementation (applying the procedure in the **opposite order**, as described in the question), be equivalent? and we will do so for all four of the higher order procedures we learned in L3.

- Map- The procedure applications on the list items are **independent**. It is implied since the procedure gets only the list items, and since L3 is functional with no side effects. So the order of the procedure application for the case of map (in the opposite order) **is equivalent**.
- Reduce- The order of the application **matters**. It is implied since in contrast to map, this procedure application gets beside the list item the aggregated value of the previous items.
  For example: `(reduce \ 1 '(1 2 3))` would return $1/6$ for one order, and $3/2$ for the opposite order.
- Filter- Filter receives an array of values, examine each value according to a certain condition, and creates a new array (the returned array) consists of the values which were found suitable. So, since the procedure examine the list items **independently**, the order of the procedure application **does not affect the values** of the returned array, but naturally, **the order** of values in the array will be **opposite as well**.
  For example: `filter(x => x % 2 === 0, [1,2,3,4])` would return `[2,4]` for one order, and `[4,2]` for the opposite order.
- Compose- regarding compose, the opposite order implementation will **not** be equivalent. Compose receives two function arguments f and g and returns a new function as a value. The two functions are activated one after the other, such that the output of the first function will be the input of the second. That is why a different order of arguments will affect the values of each function.
  In addition, compose in known as a non associative procedure (mathematic wise).
  For example:
  ```
  let example = compose(y => y*y, x => x+1);
  example(3); // (3+1)*(3+1)=16 => 3*3+1=10
  ```
  here we will get 16 for one order, and 10 for the opposite order.

Question 2: Programing in L3

Q2.1

Signature: last-element(lst)
Type: [List(T) ➔ T]
Purpose: To return the last element of a given list.
Pre-conditions: The list is not empty
Tests: `(last-element (list 1 3 4)) => 4`

Q2.2

Signature: power(n_1 n_2)
Type: [Number * Number ➔ Number]
Purpose: Given 2 numbers – n_1, n_2 , the procedure returns n_1 to the power of n_2 (${n_1}^{n_2}$).
Pre-conditions: The numbers are not negative
Tests: `(power 2 4) => 16`
`(power 5 3) => 125`

Q2.3

Signature: sum-lst-power (lst, n)
Type: [List<Number> * Number ➔ Number]
Purpose: Given a list and a number N, the procedure returns the sum of all its elements in the power of N.
Pre-conditions: The numbers are not negative (due to using power procedure), and the given parameters are a list and a number N.
Tests: `(sum-lst-power (list 1 4 2) 3) => 1^3+4^3+2^3=73`

Q2.4

Signature: num-from-digits (lst)
Type: [List<Number> ➔ Number]
Purpose: Given a list of digits, the procedure returns the number consisted from these digits.
Pre-conditions: The list is not empty
Tests: `(num-from-digits (list 2 4 6)) => 246`
`(num-from-digits (list 5 7)) => 57`

Q2.5

Signature: is-narcissistic (lst)
Type: [List<Number> ➔ Boolean]
Purpose: The procedure tests if a given number (given as a list of digits) is narcissistic or not.
Pre-conditions: The numbers are not negative (due to using power procedure), and the given parameters are a list and a number N.
Tests: `(is-narcissistic (list 1 5 3)) => #t`
`(is-narcissistic (list 1 2 3)) => #f`