Problem Set 1
COMP301 Fall 2019
03.10.2019 17:30 - 18:45

Problem 1[1]: Write inductive definitions of the following sets. Write each definition in all 3 styles (top-down, bottom-up, rules of inference). Using your rules, show the derivation of some sample elements of each set.

(1) $\{3n + 2 | n \in \mathbb{N}\}$
(2) $\{2n + 3m + 1 | n, m \in \mathbb{N}\}$
(3) $\{(n, 2n + 1) | n \in \mathbb{N}\}$
(4) $\{(n, n^2) | n \in \mathbb{N}\}$ Do not mention squaring in your rules! As a hint, remember that $(n + 1)^2 = n^2 + 2n + 1$

Problem 2[2]: Write a derivation from *List-of-Int* to $(-7 \,.\, (3 \,.\, (14 \,.\, ())))$.

Problem 3[3]: If we reversed the order of the tests in the `nth-element`, what would go wrong?
nth-element: $List \times Int \rightarrow SchemeVal$
Usage: `(nth-element lst n)` = the n[th] element of `lst`

```
(define nth-element
  (lambda (lst n)
    (if (null? lst)
      (report-list-too-short n)
      (if (zero? n)
        (car lst)
        (nth-element (cdr lst) (- n 1))))))

(define report-list-too-short
  (lambda (n)
    (eopl:error 'nth-element
      ''List too short by !~s elements.~%'' (+ n 1))))
```

[1] EOPL p.16 Exercise 1.1
[2] EOPL p.16 Exercise 1.4
[3] EOPL p.16 Exercise 1.6

Problem 4[4]: Eliminate the one call to `subst-in-s-exp` in `subst` by replacing it by its definition and simplifying the resulting procedure. The result will be a version of `subst` that does not need `subst-in-s-exp`. This technique is called inlining, and is used by compilers for optimization.

Problem 5[5]: Implement `product`: the expression `(product sos1 sos2)` where `sos1` and `sos2` are each a list of symbols without repetitions, returns a list of 2-lists that represent the Cartesian product of `sos1` and `sos2`. The 2-lists may appear in any order.

```
$ (product (a b c) (x y))
((a x) (a y) (b x) (b y) (c x) (c y))
```

Problem 6[6]: `(up lst)` removes a pair of parantheses from each top level element of `lst`. If a top-level is not a list, it is included in the result as is. The value of `(up (down lst))` is equivalent to `lst`, but `(down (up lst))` is not necessarily `lst`. (`down` appears in exercise 1.17)

```
$ (up ((1 2) (3 4)))
(1 2 3 4)
$ (up ((x (y) z)))
(x (y) z)
```

Exercise 1.17: `(down lst)` wraps parantheses around each top-level element of `lst`.

```
$ (down (1 2 3))
((1) (2) (3))
$ (down ((a) (fine) (idea))
(((a)) ((fine)) ((idea)))
$ (down (a (more (complicated)) object))
((a) ((more (complicated))) (object))
```

[4]EOPL p.22 Exercise 1.12
[5]EOPL p.27 Exercise 1.21
[6]EOPL p.28 Exercise 1.26

Problem 7[7]: Write a procedure path that takes an integer n and a binary search tree *bst* (see EOPL p.10) that contains the integer n, returns a list of lefts and rights showing how to find the node containing n. If n is found at the root, it returns the empty list.

```
$ (path 17 (14
            (7 () (12 () ()))
            (26
                (20
                    (17 () ())
                    ())
                (31 () () ))))
(right left left)
```

Hint: The grammar of BST is:
Binary-search-tree ::= () | (*Int Binary-search-tree Binary-search-tree*)

Problem 8[8]: Write a procedure `g` such that `number-elements` from EOPL page 23 could be defined as:

```
(define number-elements
    (lambda (lst)
        (if (null? lst) ()
            (g (list 0 (car lst)) (number-elements (cdr lst))))))
```

Hint: `number-elements-from` and `number-elements` are given in EOPL page 23.
Usage: `(number-elements-from (v0 v1 v2 ...)  n)`
`= ((n v0) (n+1 v2) (n+2 v2) ...  )`

```
(define number-elements-from
    (lambda (lst n)
        (if (null? lst) ()
            (cons
                (list n (car lst))
                (number-elements-from (cdr lst) (+ n 1))))))

(define number-elements
    (lambda (lst)
        (number-elements-from lst 0)))
```

Don't forget the attendance! Also, download the program from `https://download.racket-lang.org/`

[7]EOPL p.30 Exercise 1.34
[8]EOPL p.30 Exercise 1.36