Programming Assignments – 3

Programming Languages Essentials

PUCSD – January 2016-May 2016

Data Abstraction exercises. We mainly implement the three implementations of the natural number system as described in chapter 2 of the text (3$^{\text{rd}}$ edition). By data abstraction we mean the separation into an *interface*, and an *implementation*. Other programs that need natural numbers use the interface, and are oblivious of the implementation.

A system of natural numbers is defined by three *constructors* (abbreviated by us to: `ctors`): `zero`, `successor`, `predecessor`, and a predicate (abbr: `pred`, and called as *observer* in the text): `is-zero?`. `zero` returns the representation of 0, `successor` returns the next natural number, `predecessor` returns the previous natural number (until 0), and `is-zero?` returns `#t` if a given number is 0, else `#f`.

1. The *Unary representation* and the *Scheme number representation* are already described in the text. Implement them, i.e. the ctors and the pred.
2. *Bignum representation*: Implement the bignum representation of natural numbers.
3. Rewrite the `factorial` procedure using the natural number interface, i.e. just the three ctors and one pred. Execute it for each of the three different representations.

Exercises useful towards language processing. From this point on, please write the signature of the procedure as a header comment to the procedure definition. You can use a Haskell style syntax. Also: the problems may take more time as you go down the list below.

1. The `substitute` procedure in the earlier sheet accepted an old symbol, a new symbol and an expression to transform as its arguments. Generalise this procedure to operate of equal sized lists of old symbols and new symbols.
2. The `split-two-list` procedure: Given a two-list, this returns a list that is formed of two lists: the first list is the list of all the first elements of each two-list, and the second list is the list of all the second elements of each two-list. The order is preserved. For example:

   ```
   > (define a-two-list '((a 1) (b 2) (c 3) (d 4)))
   > (split-two-list a-two-list)
     ((a b c d) 1 2 3 4)
   ```

   Note: You can return `((a b c d) (1 2 3 4))`, but it will be convenient to return in the above form.
3. Write a Scheme procedure, `desugar-let`, that accepts a Scheme expression made up of one or more `let` subexpressions and returns the λ application corresponding to the `let`s. For example given the Scheme expression on the LHS the expression on the RHS should be returned.

   ```
   (define remove-first
     (lambda (sym list-of-syms)
       (if (null? list-of-syms)
           list-of-syms
             (let ((a (car list-of-syms))
                   (b (cdr list-of-syms)))
               (if (eq? sym a)
                   b
                   (cons a
                         (remove-first sym b))
    )))))
   ```

   ```
   (define remove-first
     (lambda (sym list-of-syms)
       (if (null? list-of-syms)
           list-of-syms
           (((lambda (a b)
               (if (eq? sym a)
                   b
                   (cons a (remove-first sym b)))
               )
               (car list-of-syms))
             (cdr list-of-syms))
   )))
   ```

   A restricted form of this exercise is to write a Scheme procedure, `body-preserving-desugar-let`, that *preserves* the body of the `let`. Thus, only the outermost `let` is transformed into a λ application and any inner `let` expressions are preserved as is.
4. The `count-occurrences` procedure: Given a symbol – `sym` – to count and a (possibly nested) list of symbols – `slist`, return the number of occurrences of `sym` in `slist`.

5. The `path` procedure: Given a binary search tree, `bst`, of natural numbers and a number `n`, the `path` procedure searches `bst` for `n`. It returns an empty list if `n` is at the root of `bst`, else it returns a list of symbols – ``‘‘Right’’`` or ``‘‘Left’’`` – that show the path to `n` from the root of `bst`. For example:

```
> (path 17 '(14  (7 () (12 () ()))
                 (26 (20 (17 () ())
                         ())
                     (31 () ()))))

  (Right Left Left)
```

6. Here is a definition of a *free variable*:

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

   (a) E is a variable reference and E is the same as x, or

   (b) E is of the form $(E_1\ E_2)$ an x occurs free in E_1 or E_2, or

   (c) E is of the form (`lambda` (y) E'), where y is different from x, and x occurs free in E'.

   Note that this mimics the structure of the recursive definition of a λ expression.

   (a) Rewrite the definition in the rules-of-inference style.

   (b) Use the definition to define a predicate `free?` that accepts a variable name (i.e. a symbol) and an expression and returns `#t` if the variable occurs free in the expression, and `#f` otherwise.

7. Here is a definition of a *bound variable*

   A variable x occurs bound in an expression E if and only if

   (a) E is of the form $(E_1\ E_2)$ an x occurs bound in E_1 or E_2, or

   (b) E is of the form (`lambda` (y) E'), where x occurs bound in E', or x and y are the same variables and y occurs free in E'.

   No variable occurs bound in an expression made up of just a single variable!

   (a) Rewrite the definition in the rules-of-inference style.

   (b) Use the definition to define a predicate `bound?` that accepts a variable name (i.e. a symbol) and an expression and returns `#t` if the variable occurs bound in the expression, and `#f` otherwise.

8. Write a procedure `free-set` that takes an expression and returns a set (i.e. no duplicates) of free variables in the expression.

9. Write a procedure `bound-set` that takes an expression and returns a set (i.e. no duplicates) of bound variables in the expression.

10. Write a procedure `lexical-address` that takes an expression and returns it with every variable replaced by its lexical address. (Lexical addresses will be done in the class).

11. Write a procedure `un-lexical-address` that takes an lexical addressed expression and a list of formal parameters, and returns an equivalent expression where lexical addresses are substituted by the variable references, or `#f` if no such expression can be returned.

12. Write a procedure `rename` that accepts an expression `exp` and two variables `v1` and `v2`, and returns the expression where every occurrence of `v1` in `exp` is replaced by `v2` if `v1` does not occur free in `exp`, or `#f` otherwise. How is this related to the `substitute` procedure you may have written earlier? How is this related to the α conversion of the λ calculus? Can you write `alpha-convert` using rename?