Principles of Programming languages, spring 2016.

Assignment 3

Data abstraction, partial evaluation, lazy lists, CPS & type inference

Submission instructions:

1. Write answers to each of the programming questions in the corresponding .rkt file.
2. Answers to theoretical questions should be submitted in **ex3.pdf** (Use ex3.docx and "Save as PDF" in word).
3. Submit an archive file named id1_id2.zip (or id1.zip in case of a single student) where id1 and id2 are the IDs of the students responsible for the submission. The zip file should not create any additional directories when inflated.
4. A contract must be included for every implemented procedure.
5. Make sure your .rkt files are in the correct format (see the announcement about "WXME format" in the course web page).
6. Your code is tested automatically, be sure to use exact procedure and file names. Please use the provided submission template. Any appeals that require modifying your submission will not be accepted.

Question 1: Data Abstraction

In order to manipulate sets of elements of the same type, we introduce a Set(T) ADT with the following interface:

```
Signature: make-set(elements)
Purpose: Construct a set with the elements in ‘elements’
Type: [List(T) -> Set(T)]

Signature: set-extend(e, s)
Purpose: Extend set ‘s’ with element ‘e’
Type: [T*Set(T) -> Set(T)]

Signature: set-select(s)
Purpose: select an element from set ‘s’.
Type: [Set(T) -> T]
Pre-conditions: not(empty?(s))

Signature: set-remove-el(e,s)
Purpose: Remove 'e' from set ‘s’. If 'e' is not in 's', do nothing.
Type: [T*Set(T) -> Set(T)]
Pre-conditions: not(empty?(s))

Signature: set-remove(s)
Purpose: Remove any element from set ‘s’.
Type: [Set(T) -> Set(T)]
Pre-conditions: not(empty?(s))

Signature: set->list(s)
Purpose: Turns set ‘s’ into a list with the same elements.
Type: [Set(T) -> List(T)]

Signature: set?(s)
Purpose: Set identifier
Type: [T -> Boolean]

Signature: set-empty?(s)
Purpose: Tests whether 's' is empty: Return true if and only if the set
‘s’ is empty.
Type: [Set(T) -> Boolean]

Signature: set-equal?(s1, s2)
Purpose: Return true if and only if ‘s1’ 's2' are the same set.
Type: [Set(T) -> Boolean]

Empty-set primitive value
```

a. **Client procedure**:
   1. Below is the implementation presented in example 3.8, in chapter 3.2.4. The given implementation breaks the "abstraction barrier" of the Set ADT. Point to the points where the ADT barrier is broken, and correct the implementation, using the above interface. **Write your implementation, including the contract in the file q1a.rkt**.

```
; Signature: permutations(s)
; Type: [Set(T) -> List(List(T))]
; Purpose: Compute all permutations of the elements of in 's'
(define permutations
  (lambda (s)
    (if (set-empty? s)
        (list empty)
        (flatmap
         (lambda (x)
           (map (lambda (p) (cons x p))
                (permutations (set-remove-el x s))))
         s))
    ))
```

2. Using the Set(T) ADT write a client side procedure, power-set, which computes the power set of a given set. Make sure not to break the Set(T) ADT abstraction barrier. **Write your implementation including the relevant contracts in q1a.rkt**.

b. **Numeral Set implementation:** *Gödel encoding* is a method that encodes a number sequence as a numeral value. The inverse decoding method computes the encoded sequence. A sequence of numbers **(e_1,e_2,…,e_n)** is encoded as the multiplication of the powers of the first **n** primes, in the following manner:

$$enc(e_1,e_2,\ldots,e_n)=2^{e1}\cdot 3^{e2}\cdot\ldots\cdot p_n^{en}$$

Use Gödel encoding, or any other number encoding of your choice, for implementing the Set(Number) ADT. Notice that for the implementation, the contracts should be written not from the client view as above, but from the implementation view, i.e., you cannot use the Set ADT to specify the type of a n ADT procedure). **Write your implementation including the relevant contracts in q1b.rkt.**

Advice: Try to maximize the number of ADT operations that are implemented in terms of other ADT procedures (and not from scratch).

c. An ADT comprises an interface as well as invariants. Below are two invariants suggested for the Set(T) ADT. For each suggested invariant, write whether it reflects the correctness of the ADT. If not, explain why. Otherwise, provide a proof that the implementation satisfies the invariant. **Write your answer in the file ex3.pdf**.

```
1. set-remove-el(e, set-extend(e, s) ) = s
2. set-extend(n, Empty-set) = make-set(n)
```

d. Consider the ADT Set(Set(Number)), i.e., sets of sets pf numbers, as in the following example:

{ {2,5,1} , {1 56, 9} }

Suggest how to implement such sets using the above number encoding method. **Write the implementation for the** `make-set` **constructor, including the contract in ex3.pdf.**

e. **Procedural implementation:** Replace the above numeral implementation of the Set(T) ADT with a lazy procedural implementation. Make sure to **revise the contract-type, client side**. **Note:** You might need to replace only part of the ADT procedures! **Write your implementation including the relevant contracts in q1e.rkt**.

f. Explain which procedures you re-implemented in part **e** and remained the same. **Write your answer in ex3.pdf**.

Question 2: Partial Evaluation

Below is the function `nth-deriv` (see section 1.5.4 in the course book), which approximates the nth derivative of a given function. `nth-deriv` receives a function of a single argument, f, and a number that indicates the order of the derivative, n.

```
; Signature: nth-deriv(f,n)
; Type: [[Number -> Number]*Number -> [Number -> Number]
(define nth-deriv
 (lambda (f n)
   (if (= n 0)
       f
       (nth-deriv (derive f) (- n 1)))
 ))
```

Partial evaluation is an approach that considers cases where some parameters are known before the rest. It tries to maximize the computations performed on the known parameters, so to minimize the computation left to the final execution of the delayed computation. Currying is a method used to perform partial evaluation, by transforming a multi-parameter function into a higher-order function whose input-parameters are the known original parameters, and its output is a function in the delayed original parameters. However, <u>trivial Currying does not imply partial evaluation</u>.

Partial evaluation of the `nth-deriv` procedure can delay either the function parameter f, or the number parameter n, using the Currying method. Write 2 partial evaluation procedures for `nth-deriv`, using non-naïve Currying.

a. The procedure `c-nth-deriv-f` is a Curried version of `nth-deriv`, in which the number parameter is delayed, i.e., the partial evaluation is performed on `f`. **Implement the procedure and complete its contract in q2.rkt**.

```
; Signature: c-nth-deriv-f (f)
; Type: [[Number -> Number] -> _____________]
(define c-nth-deriv-f
 (lambda (f) …
))
```

b. The procedure `c-nth-deriv-n` is a Curried version of `nth-deriv`, in which the function parameter is delayed, i.e., the partial evaluation is performed on `n`. **Implement the procedure and complete its contract in q2.rkt**.

```
; Signature: c-nth-deriv-n (n)
; Type: [Number -> ___________________________]
(define c-nth-deriv-n
 (lambda (n) …
))
```

Question 3: Lazy Lists

In numerical analysis, Newton's method is a method for finding the roots of a real-valued function, f. Starting with an initial guess, x_0, the method provides an improved approximation, x_1, as follows:

$$x_1 = x_0 - (f(x_0)/f'(x_0))$$

The improve-guess step is repeated until a "close-enough" approximation is obtained. The procedure `function-root`, implements this method, for a real-valued function, `f`, an initial guess, `x0`, and an accuracy parameter, `eps`.

```
; Signature: function-root(f,x0,eps)
; Type: [[Number -> Number] * Number * Number -> Number]
(define function-root
  (lambda (f x0 eps)
    (if (< (abs (f x0)) eps)
        x0
        (let ((x1 (- x0 (/ (f x0) ((derive f 0.001) x0)))))
          (function-root f x1 eps)))))
```

The method is not always guaranteed to converge. For some functions and some initial guesses, it may enter an infinite loop. For example, try to apply the method above in the following two scenarios:

1. $f(x) = x^3 + x^2$, with the initial guess `x0` = -0.7 and accuracy `eps` = 0.001
2. $g(x) = x^3 - 2x + 2$, with the initial guess `x0` = 0 and accuracy `eps` = 0.001

a. Implement Newton's method for finding roots of real-valued functions using lazy-lists. Write the procedure `lazy-function-root`, which receives a real-valued function, `f`, an initial guess, `x0`, and an accuracy parameter, `eps`, as parameters. The procedure returns a lazy list of the computed guesses. A partial contract is given below. **Implement the procedure and complete its contract in q3.rkt**.

```
Signature: lazy-function-root(f,x0)
Type: [[Number -> Number] * Number * Number -> Lzl]
```

b. Point out an advantage of using a lazy-list based implementation over the implementation provided above. **Write your answer in ex3.pdf**.

Question 4: Continuation Passing Style

a. Recursive to iterative CPS transformation:

Recall the implementation of the procedure `filter`, presented in class, which creates a recursive process:

```
; Signature: filter(pred, seq)
; Purpose: return a list of all elements in 'seq' that satisfy 'pred'
; Type: [[T-> Boolean]*List(T) -> List(T)]
(define filter
  (lambda (pred seq)
    (cond ((empty? seq) empty)
          ((pred (car seq)) (cons (car seq) (filter pred (cdr seq))))
          (else (filter pred (cdr seq))))))
```

The filtering predicate `pred` can be primitive or user-defined. Accordingly, there are two CPS-equivalent iterative variants, `filter1$` and `filter2$`. `filter1$` is given a CPS user-defined predicate `pred$`, while `filter2$` is given a primitive `pred` argument.

1. **User-defined predicate** `pred$`**:** Use the recursive to iterative CPS transformation in order to transform the procedure `filter` into a CPS-equivalent iterative procedure `filter1$`. **Write your implementation including the contract in q4.rkt**.

2. Prove that `filter1$` is CPS-equivalent to `filter`. **Write your proof in ex3.pdf**.

3. **Primitive predicate** `pred`**:** Use the recursive to iterative CPS transformation in order to transform the procedure `filter` into a CPS-equivalent iterative procedure `filter2$`. Note that the continuation in `filter2$` should be applied to the value of the predicate application. **Write `filter2$`, with its contract in q4.rkt**.

b. A CPS service with multiple future continuations :

An *association-list* is a list of key-value pairs. The natural operation on an association list is retrieval of the value for a given key. But what should be the result if a requested key is not in the list? This can be determined by client-callers that request for a key-value, and determine how to continue in case of success or failure.

1. Write a value-retrieval CPS-procedure get-value for association lists:

```
Signature: get-value$(assoc-list, key, success, fail)
Purpose: Find the value of 'key'. If 'key' is found, then apply the
         continuation 'success' to the pair (key . val). Otherwise,
         apply the continuation 'fail'.
Type: [List(Pair(Symbol,T))*Symbol*[Pair(Symbol,T1)->T2] *
        [Empty->T3]] -> T2 union T3)
Examples:   > (get-value$ '((a . 3) (b . 4)) 'b (lambda(x)x) (lambda()#f))
             (b . 4)
```

Write your implementation including the contract in q4.rkt.

2. **Retrieve a value in a list of association lists**:
   Implement two client procedures, as described in the contracts below. The procedures call `get-value$` in order to retrieve values in a **list of association-lists**. **Write your implementation including the contracts in q4.rkt**.

```
Signature: get-first-value(list-assoc-lists, key)
Purpose: Retrieves the value of 'key' in the first association-list
that includes 'key'. If no such value, the Scheme error function is
applied.
Type: [List(Association-list)*Symbol -> T]
Examples:  > (define l1 '((a . 1) (b . 2) (e . 3)))
           > (define l2 '((e . 5) (f . 6)))
           > (get-first-value (list l1 l2), 'e)
           3
           > (get-first-value (list l1 l2), 'g)
           `()

Signature: collect-all-values(list-assoc-lists, key)
Purpose: Returns a list of all values of 'key' in the given association
lists. If no such value, returns the empty list.
Type: [List(Assoc-list)*Symbol -> T]
Examples:  > (define l1 '((a . 1) (b . 2) (e . 3)))
           > (define l2 '((e . 5) (f . 6)))
           > (get-first-value (list l1 l2), 'e)
           `(3 5)
           > (get-first-value (list l1 l2), 'k)
           `()
```

Question 5: Axiomatic Type Inference

Write your answers to all parts of this question in the file ex3.pdf

a. Theory

1. The notion of *substitution* and its *operations* is used both for the construction of operational semantics for Scheme, and for the construction of type inference procedures for Scheme expressions. Compare the two notions: *Substitution* (as in Chapter 2) and *type-substitution* (as in Chapter 5). Note 2 similarities and 2 differences.

2. For each of the typing statements below, mark:
   (i) Whether it is **true** or **false;**
   (ii) Whether it has an instance, which is **true**. That is, whether there exists a type-substitution *s*, such its application to the typing statement *TS*, *TS* ° *s*, is a true typing statement.

   a. `{f:[T1 -> [T2->T1]], x:T1} |- (f x):[T2->T1]`

   b. `{f:[T1 -> [T2->T1]], x:Number} |- (f x):[T2->T1]`

   c. `{f:[T1 -> [Boolean->T1]], x:T1} |- (f x):[T2->T1]`

   d. `{f:[T1 -> [T2->T1]]} |- (f x):[T2->T1]`

   e. `{f:[T1 -> T3], x:T1} |- (f x):[T2->T1]`

3. Find a most general unifier for the following pairs of typing statements, if they are unifiable, or state that they are not unifiable, and explain why.

   a.
   ```
   {f:[T1 -> [T2 -> T1]], x:T1} |- (f x):T1
   {f:T3, x:T4} |- (f x):[T1 -> T2]
   ```

   b.
   ```
   {f:[T1 -> [T2 -> T1]], x:T1} |- (f x):T1
   {f:T3, x:T1} |- (f x):[T1 -> T2]
   ```

   c.
   ```
   {f:[T1 -> [T2 -> T1]], x:T1} |- (f x):Number
   {f:T3, x:Number} |- (f x):[T1 -> T2]
   ```

b. Type inference:

1. Derive a type for the following defined variable (the naïve "partial evaluation" of the map procedure):

```
(define c-map-proc
  (lambda (proc)
    (lambda (lst)
      (if (empty? lst)
          empty
          (cons (proc (car lst))
                ((c-map-proc proc) (cdr lst))))
    )))
```

   For each step indicate:
   - which Rule is used,
   - the base typing-statements,
   - the unifier applied to the typing statements in the rule condition, and the base typing statements
   - whether Monotonicity was used.

2. Assume that the naively Curried previous version is incorrectly written as:

```
(define c-map-proc
  (lambda (proc)
    (lambda (lst)
      (if (empty? lst)
          empty
          (cons (proc (car lst))
                (c-map-proc proc (cdr lst))))
    )))
```

   Show at which step your proof from previous entry fails. There is no need to repeat identical steps. Just write the step where the proof diversify until it ends.

3. Assume that the naively Curried map from entry (b.1) is modified as:

```
(define c-map-proc
  (lambda (proc)
    (lambda (lst)
      (if (empty? lst)
          lst
          (cons (proc (car lst))
                (c-map-proc proc (cdr lst))))
    )))
```

   Explain and show how this change affects the results.

c. Extension of the type-inference system

In this part we extend the axiomatic type inference system to apply an extended Scheme subset. The language is extended with the following **let* special operator**:

```
(let*  ((var₁ val₁)
         …
         (varₙ valₙ))
  (e₁ e₂ … eₘ)))
```

which is equal to n nested let expressions:

```
(let ((var₁ val₁))
   (let ((var₂ val₂))
      …
      (let ((varₙ valₙ))
         (e₁ e₂ … eₘ)
       )…))
```

1. Add a typing rule for `let*` expressions. For example, the typing rule for `let` expressions is:

```
For every: type environment _Tenv,
  variables _v1, _v2, ..., _vn, n>=0,
  expressions _e1, _e2, ..., _en,
  expressions _b1, _b2, ..., _bm, m>=1, and
  type expressions _S1, ...,_Sn, _U1, ...,_Um :
If _Tenv |- _e1:_S1,
   _Tenv |- _e2:_S2,
   _Tenv |- _e3:_S3,
   ...,
   _Tenv |- _en:_Sn,
   _Tenv ° {_v1:_S1, _v2:_S2, ..., _vn:_Sn} |- _b1:_U1,
   _Tenv ° {_v1:_S1, _v2:_S2, ..., _vn:_Sn} |- _b2:_U2,
   ...,
   _Tenv °{_v1:_S1, _v2:_S2, ..., _vn:_Sn} |- _bm:_Um
Then _Tenv |- (let ((_v1 _e1) (_v2 _e2) ... (_vn _en))
                 _b1 _b2 ... _bm) : _Um
```

2. Derive the type of:

```
(let* ((x 1)
       (y (+ x 1)))
  (+ x y))
```

3. Show how the type derivation in entry (2) differ from the type derivation of:

```
(let ((x 1)
      (y (+ x 1)))
  (+ x y))
```

Question 6: Type Inference Using Type-Constraints

For entries (a), (b), (c): write answers in file ex3.pdf

For entry (d): You are required to modify the modules in the type-inference-system according to the instructions in entry (d). The modules are provided in the submission template.

a. Add equations for let* special forms as specified in question 5. Explain how these equations are derived from the inference rule that you have suggested in Question 5.c.1.

b. Derive the type of the following let* expression, using the type-constraints method.

```
(let* ((x 1)
       (y (+ x 1)))
  (+ x y))
```

c. Compare the axiomatic type-inference method with the type-constraint based one. Suggest two advantages and two disadvantages of each method.

d. Type-constraint-based inference system:

1. Consider the module `substitution-adt.rkt`, that defines the substitution ADT. A substitution can be implemented as an association-list (see question 4.b.1 above). This implementation is given by the following make-sub procedure.

```
(define make-sub
  (lambda (variables tes)
    (if (ormap occur? variables tes)
        (error 'make-sub "circular substitution)
        (letrec ((make-assoc-list (lambda (vars tes)
                                    (map cons vars tes))))
          (make-tagged 'sub (make-assoc-list variables tes))))
    ))
```

   i. The selector `sub->expression-of-variable`, is a getter for the type expression of the given variable. It can use the `get-value$` getter written in question 4.b.1. Implement `sub->expression-of-variable` **as a caller to** `get-value$`. Suggest an alternative to the error call, in case of failure (undefined key or variable).

   ii. Adapt other Substitution operations to the association-list implementation, so that all tests in module `substitution-adt-tests.rkt` pass safely.

   iii. Consider the Type-expression operations in module `type-expression-adt.rkt`. Modify all operations that should be changed, so that all tests in module `type-expression-adt-tests.rkt` pass safely.

2. Extend the system to support let special forms. The extension to the Scheme parser module `schme-ast.rkt` is provided in the assignments page on the course website.

3. **Bonus:** Extend the type-inference system to support let* special forms. Note that this extension requires creation of type variables for all nested let* forms.