Principles of Programming languages, spring 2015.

Assignment 3.

Abstraction on data and control: The sequence interface, CPS, compound data and ADTs

Submission instructions:

1. Write your answers to each of the programming questions below 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): These include Q1 ; Q2-f, Q2-h ; most of Q3-a, Q4-a(2), Much of Q5
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.
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. Use exact procedure and file names, as your code is tested automatically. Use the provided submission template.

Question 1 - concepts and properties:

Write your answers to the questions below in the file *ex3.pdf*.

a. Give an example for a programming language whose syntax is typeless but its values are typed. Explain how you know that the value domain is typed.

b. Does the type checking mechanism of Java guarantee type safety? Justify your answer with an example.

c. Two students argue about a possible type inference rule for a cond expression in Scheme. Given the Scheme expression:

$\texttt{(cond (}p_1\ e_{11}\ \texttt{...}\ e_{1n}\texttt{)}$
$\qquad\quad\texttt{...}$
$\qquad\quad\texttt{(else}\ e_{m1}\ \texttt{...}\ e_{mn}\texttt{))}$

Joe suggests adding m rules:

For all Scheme variables $\_p_1,\ \_e_{11},\ ...,\ \_e_{1n1}$
Type expressions $\_S_1,\ \_S_{11},\ \_S_{12},\ \ldots,\ \_S_{1n}$, and $\_Tenv$
If: $\_Tenv\ |\text{-}\ p_1{:}\_S_1,\ e_{11}\ |\text{-}\ \_S_{11},\ \ldots,\ \_Tenv\ |\text{-}\ e_{1n}{:}\_S_{1n}$
Then: $\_Tenv\ |\text{-}\ \texttt{(cond (}p_1 \ldots e_{1n}\texttt{)} \ldots \texttt{(else}\ e_{m1} \ldots e_{mn}\texttt{))}\ :\ \_S_{1n}$

and

For all Scheme variables $\_p_1,\ \_p_2,\ \_e_{21},\ ...,\ \_e_{2n}$
Type expressions $\_S_1,\ \_S_2,\ \_S_{21},\ \_S_{22},\ \ldots,\ \_S_{2n}$, and $\_Tenv$
If: $\_Tenv\ |\text{-}\ \_p_1{:}{:}\_S_1$ cannot be proved, and
$\_Tenv\ |\text{-}\ p_2{:}\_S_2,\ e_{21}\ |\text{-}\ \_S_{21},\ \ldots,\ \_Tenv\ |\text{-}\ e_{2n}{:}\_S_{2n}$
Then: $\_Tenv\ |\text{-}\ \texttt{(cond (}p_1 \ldots e_{1n}\texttt{)} \ldots \texttt{(else}\ e_{m1} \ldots e_{mn}\texttt{))}\ :\ \_S_{2n}$

and
...
and

For all Scheme variables $\_p_{m-1},\ \_p_m,\ \_e_{m1},\ ...,\ \_e_{mn}$
Type expressions $\_S_{m-1},\ \_S_m,\ \_S_{m1},\ \_S_{m2},\ \ldots,\ \_S_{mn}$, and $\_Tenv$
If: $\_Tenv\ |\text{-}\ \_p_{m-1}{:}\_S_{m-1}$ cannot be proved, and
$\_Tenv\ |\text{-}\ p_m{:}\_S_m,\ e_{m1}\ |\text{-}\ \_S_{m1},\ \ldots,\ \_Tenv\ |\text{-}\ e_{mn}{:}\_S_{mn}$
Then: $\_Tenv\ |\text{-}\ \texttt{(cond (}p_1 \ldots e_{1n}\texttt{)} \ldots \texttt{(else}\ e_{m1} \ldots e_{mn}\texttt{))}\ :\ \_S_{mn}$

Moe suggests the following rule:

For all Scheme variables _p$_1$, _e$_{11}$, ..., _e$_{1n}$,
…, _p$_m$, _e$_{m1}$, ..., _e$_{mn}$
Type expressions _S$_1$, _S$_{11}$, _S$_{12}$, …, _S$_{1n}$,
..., _S$_m$, _S$_{m1}$, _S$_{m2}$, …, _S$_{mn}$, and _Tenv
If: _Tenv |- _p$_1$:_S$_1$, , e$_{11}$ |- _S$_{11}$, …, _Tenv |- e$_{1n}$:_S$_{1n}$,…,
_Tenv |- p$_m$:_S$_m$, e$_{m1}$ |- S$_{m1}$, …, _Tenv |- e$_{mn}$:_S$_{mn}$
Then: _Tenv |- (cond (p$_1$ … e$_{1n}$) … (else e$_{m1}$ …e$_{mn}$)):
_S$_{1n}$ union _S$_{2n}$ union … _S$_{mn}$

Compare the suggestions of Joe and Moe:

1. Explain which suggestion guarantees type safety and why?

2. Which suggestion (if any) includes redundant conditions?

3. Is there a `cond` expression whose evaluation does not cause a runtime error and still cannot be proved to have a type? What does it say about the type-inference system?

d. Lazy-lists:

1. Lazy lists, as presented in the lecture notes are constructed using the List constructor cons, which receives a closure as its 2nd argument. For example:

```
(define ones (cons 1 (lambda () ones)))
```

Joe suggests introducing a better constructor that receives directly the head and tail of the lazy-list:

```
(define cons-lzl
  (lambda (h t)
    (cons h (lambda () t))))
```

Is that a good suggestion? Explain.

2. Consider the definition of the ones lazy list above. Can you prove that it has a type, using the type inference system presented in chapter 2? If not, suggest an extension that will enable that.

e. CPS style:

1. One of the uses of writing procedures in the CPS style is the automatic transformation of a procedure that creates a recursive process, into an iterative procedure – one that creates an iterative process.

   For example, we presented the transformation of the `fib` recursive procedure, that computes the factorial of an integer, into a CPS style iterative procedure `fib$`, which is CPS-equivalent to `fib`.

   But, in fact, we do not need this transformation to `fib$`, since we already presented (in chapter 1), a nice iterative procedure `fib-iter` which computes the Fibonacci numbers. Compare the iterative procedures `fib$` and `fib-iter`.

2. Consider the `scale-tree` procedure for multiplying the leaves of an integer valued tree (only leaf values) by some factor.

   If you are asked to provide an iterative `scale-tree` – which approach would you suggest using: Finding a `scale-tree-iter` or using a transformation to `scale-tree$`?

3. What are the advantages and disadvantages of transforming a recursive procedure f into a CPS-equivalent iterative procedure?

f. Definition within scope: Consider the following 2 versions of implementing a procedure that checks whether a number is odd:

```
(define odd1?
  (lambda (n)
    (letrec ((even?
               (lambda (n)
                 (if (= n 0)
                     #t
                     (odd1? (- n 1))))))
      (if (= n 0)
          #f
          (even? (- n 1))))))

(define odd2?
  (letrec ((even?
             (lambda (n)
               (if (= n 0)
                   #t
                   (odd2? (- n 1))))))
    (lambda (n)
      (if (= n 0)
          #f
          (even? (- n 1))))))
```

Explain which version is preferred, based on the criterion of minimizing the number of closures that are created during procedure application. For example, how many closures are created by each procedure when computing (odd1? 4) and (odd2? 4)? Explain.

Question 2 – Sequence Operations

The function `or-map` takes a closure *f* and a list *lst* as parameters, and returns #t if one of the elements in the mapping of *f* to *lst* is not #f, and #f otherwise.

For example:

```
(or-map even? (list 1 2 3 4 5 6))
> #t
(or-map number? (list #t #t #f 'a 'b 'c (lambda (f) (f 15))))
> #f
```

a. Implement the function or-map without sequence operations, and complete its associated contract in ex3.rkt.

b. Implement the function `or-map-seq-filter` using only the sequence operations `filter` and `map`, and complete its associated contract in ex3.rkt.

c. Implement the function `or-map-seq-acc` using only the sequence operations `accumulate` and `map`, and complete its associated contract in ex3.rkt.

d. Write a partially evaluated version: `or-map-c-f` of `or-map` (using Currying) for one of the functions in the previous three tasks, and complete its contract in the file ex3.rkt. This function Curries out the function (not the list).

e. Write a partially evaluated version: `or-map-c-lst` of `or-map` (using Currying) for one of the functions in the previous three tasks, and complete its contract in the file ex3.rkt. This function Curries out the list (not the function).

f. Which version did you rewrite as `or-map-c-f`? Explain in what ways the function `or-map-c-f` performs partial evaluation, and in what ways it saves on computation? Why is your implementation of `or-map-c` better than the original? Put your answers in ex3.pdf.

g. Implement the procedure `or-map-tree` which takes a function f and a nested list of lists (of unknown depth – i.e. a tree), and returns #t if the application of f on one of the leaves of the tree is not #f, and #f otherwise.

   For example:

```
(or-map-tree even? (list 1 (list (list 2 3)) 4 (list 5) 6))
> #t
(or-map-tree number? (list #t (list (list #t #f 'a)) 'b 'c (lambda (f) (f 15))))
> #f
```

   Implement this procedure using the sequence operations `map` *and* `accumulate`.

Complete the function `or-map-tree` and its associated contract in the file ex3.rkt.

h. Does the function `or-map-tree` halts for every legal input? Explain your answer. Your explanation should, at the very least, include a formal intuition as to why `or-map-tree` halts or fails to halt for a given input. Put your answers in ex3.pdf

Question 3 – Lazy Lists

a. Lazy-list ADT

In this task we will define an ADT for lazy lists and implement some user level code with it, as well as prove code invariants.

1. Define the lazy list ADT, using the following procedures:
   - `(make-lzl` *`head`* `(lambda ()` *`lzl`*`)`: A constructor for Lazy Lists.
   - `(head` *`lzl`*`)` : Get the head of a lazy list.
   - `(tail` *`lzl`*`)` : Get the tail of a lazy list (returns a lazy list)
   - `(take` *`lzl n`*`)` : Returns a list with the first n elements of lazy list.
   - `(drop` *`lzl n`*`)` : Returns a lazy list without the 1st n elements in lzl.
   - `(nth` *`lzl n`*`)` : Returns the nth element of lazy list.
   - `(lzl?` *`candidate`*`)` : Evaluates to #t if and only if candidate is a lazy list.
   - `(equal-prefix-lzl?` *`lzl1 lzl2 n`*`)`:
     Evaluates to #t if and only if lzl1 and lzl2 contain the same first n elements.

   For each procedure write its purpose, its type signature (from the client perspective), pre-conditions and post-conditions. Write your solution in the file ex3.pdf.

2. Write five invariants for the Lazy-list ADT. For example:
   `(head (make-lzl a (lambda () lzl)) => a.`
   Write your answers in the file ex3.pdf.

3. Implement the client procedure `lzl-pow`, which takes a lazy list `lzl` and a number `k` and returns the lazy list of the k-powers of the elements of `lzl`.
   For example:

   ```
   (take (lzl-pow (integers-from 2) 2) 5)
   > '(4 9 16 25 36)
   ```

   Write your answers in ex3.rkt.

4. Implement the Lazy-list ADT, and complete their associated contracts and place your answers in ex3.rkt.

5. Prove three of the invariants in entry a.2 for your implementation

Write your answers to (3) and (4) in the file ex3.pdf

b. Apply a function indefinitely

It is often the case that data needs to be read over an open channel, wherein we are given a stream handle, which outputs data either indefinitely or until we reach the end of the stream. Most file transfer protocols have this property, and many session protocols also share it. Therefore, given a stream handle, we would like to read it incrementally into a buffer, perform necessary operations, and continue. This situation is perfect for lazy lists.

Given a function *f* and an initial value *x*, the indefinite application of *f* on *x* is defined to be the sequence: {*x*, *f(x)*, *f(f(x))*, *…*}. This sequence may either be finite or infinite (depending on *f* and *x*).

We wish to implement the indefinite application of a closure to an initial value in a similar way: given a closure *f* and an initial value *x* `(lzl-apply f x)` will generate a lazy list whose first value is *x*, second value is *f(x)* and so on. If at any point *f(f(…f(x)))* returns the symbol *'end_of_lazy_list* then the lazy list should end.

For example:

- `(lzl-apply (lambda (n) (+ n 1)) 0)`
  is equivalent to `(integers-from 0)` as seen in class.

-
  ```
  (lzl-apply
     (lambda (n)
        (if (> n 100)
           'end_of_lazy_list
           (+ n 1))) 0)
  ```
  will generate the lazy list which contains the numbers 0…100

6. Implement `lzl-apply` and its associated contract in the file ex3.rkt. You should use `equal?` In order to test for symbol equality.

7. Use `lzl-apply` to implement the function `group-generator`:
   The nth element of `(group-generator p g)` is $g^n \ mod\ p$. You may use the `modulo` operation (you do not need to use the `expt` operator). Complete the function definition and its associated contract in the file ex3.rkt.

8. Use `lzl-apply` to implement the function `integers-until`:
   The nth element of `(integers-until cond init)` is `init+n`.
   If at any point `cond` <u>***succeeds***</u> on an element – then that is the last element of the lazy list.

   For example:
   ```
   (integers-until
         (lambda (n)
              (= 0 (modulo n 1027))) 1)
   ```

   Generates a lazy list whose elements are 1…1027 including.

   However: `(integers-until (lambda (n) (+ n 1)) 0)`

Generates the infinite lazy list from 0.

Complete the function `integers-until` and its associated contract in ex3.rkt.

c. Cycle detection algorithm for lazy lists

Given a lazy list, which represents a finite or infinite number of elements, we would like to know whether it contains a cycle, i.e whether we can start at element x and after a finite number of steps return to x.

Floyd's Cycle Detection algorithm allows us to do so efficiently. We initialize two elements: the "tortoise" and the "hare" to be the first and second elements of the lazy list. At each step the tortoise moves one step forward, while the hare moves two steps. If at some point their values are equal – we have reached the same element at different locations – and detected a cycle. You can read more about this algorithm on Wikipedia: http://en.wikipedia.org/wiki/Cycle_detection#Tortoise_and_hare.

Implement `lzl-cyclic?`, a version of Floyd's algorithm for lazy lists. The function takes a lazy list and a bound on the number of steps the tortoise may perform, and either return `#t` (if a cycle has been detected), `#f` (if the list is finite) or `'?` (if the bound has been reached with no conclusion). All comparisons are to be done using `equal?` and your function may not go into an infinite loop, or perform more steps than the provided bound.

For example:

```
- (lzl-cyclic? (group-generator 17 3) 1000)
  > #t
- (lzl-cyclic? (integers-until (lambda (n) (< n 100)) 3) 1000)
  > #f
- (lzl-cyclic? (integers-from 0 1000)
  > '?
```

Implement the function `lzl-cyclic?` and complete its associated contract in the file ex3.rkt.

Question 4 – CPS programming

a. **Recursive to Iterative CPS Transformations**

The following implementation of the procedure `append`, presented in class, generates a recursive computation process:

```
Signature: append(list1, list2)
Purpose: return a list which the arguments lists, from left to
right.
Purpose:   Append list2 to list1.
Type: [List * List -> List]
Example: (append '(1 2) '(3 4) id) should produce '(1 2 3 4)
Tests: (append '() '(3 4)) ==> '(3 4)
       (append '(1 2)
               '(3 4)
               (lambda(x) (map (lambda(x)(+ x 1)) x)))
         ==> '(2 3 4 5)
(define append
   (lambda (x y)
      (if (null? x)
          y
          (cons (car x)
                (append (cdr x) y)))))
```

1. Write a CPS style iterative procedure `append$,` which is CPS-equivalent to `append`. Implement the procedure and complete its associated contract in the file *ex3.rkt*.

2. Prove that `append$` is CPS-equivalent to `append`. That is, for lists $\texttt{lst}_1$ and $\texttt{lst}_2$ and a continuation procedure `cont`,

   ```
   (append$ lst1 lst2 cont)=(cont (append lst1 lst2)).
   ```

   Prove the claim by induction (on the length of the first list), and using the applicative-eval operational semantics. Write your proof in the file *ex3.pdf*.

b. **CPS with multiple continuations**

In the type-inference system, type-expressions are internally represented as lists with prefix notation. For example:

```
> (te-parse '[Number * T1 -> [T2 * T3 -> T3]])
'(-> (* Number T1) (-> (* T2 T3) T3))
```

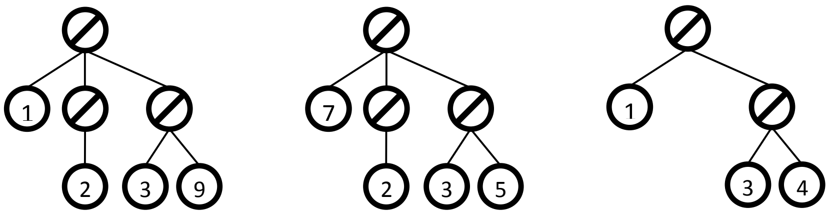
Type expressions are equivalent if they are equivalent up to <u>consistent</u> renaming of type-variables. For example:

Equivalent type-expressions:	*Non-equivalent type-expressions:*
`[Number * T1 -> [T2 * T3 -> T3]]`	`[Number * T1 -> [T2 * T3 -> T3]]`
`[Number * T2 -> [T4 * T5 -> T5]]`	`[Number * T2 -> [T4 * `**`T5`**` -> `**`T7`**`]]`

In the following you are asked to implement two procedures that, combined together will be used to determine equivalence between two given type-expressions:

1. **Structure identity:**

   We regard type-expressions as leaf-valued trees (in which data is stored only in the leaves). Trees may differ from one another both in structure and the data they store. E.g., examine the following leaf-valued trees:



A B C

Trees A and B have different data stored in their leaves. Both A and B have a different structure than C.

The CPS procedure `equal-trees$` receives a pair of leaf-valued trees, `t1` and `t2`, and two continuations: `succ` and `fail` and determines their structure identity as follows:

- If t1 and t2 have the same structure, `equal-trees$` returns a list of corresponding pairs of leaves with different values (or the empty list if there are none).
- Otherwise, `equal-trees$` returns a list with the first conflicting sub-trees in depth-first traversal of the trees.

For example:

```
>  (equal-trees$  '(1  (2)  (3  9))  '(7  (2)  (3  5))  id  id)
'((1.7) (9.5))

> (equal-trees$ '(1 (2) (3 9)) '(1 (2) (3 9)) id id)
'()

> (equal-trees$ '(1 2 (3 9)) '(1 (2) (3 9)) id id)
```

```
'(2 (2))

> (equal-trees$ '(1 2 (3 9)) '(1 (3 4)) id id)
'(() (4))

> (equal-trees$ '(1 (2) ((4 5))) '(1 (#t) ((4 5))) id id)
'(2 #t)
```

Implement the procedure `equal-trees$` and write its contract in *ex3.rkt*.

2. **Consistent renaming**

   Two type expressions with identical structures are not necessarily equivalent. For example:

```
[Number * T1 -> [T2 * T3 -> T3]]
[Number * T4 -> [T4 * T5 -> T5]]

[Boolean * T1 -> [T2 * T3 -> T3]]
[Number  * T2 -> [T4 * T5 -> T5]]
```

   For two type-expressions, t1 and t2, of identical structure to be equivalent, the following must hold:
   - Any type-constructor in one tree is equal to its corresponding type-constructor in the other.
   - Any type variable in t1 is mapped to a single type variable in t2 and vice versa.

   The procedure `is-consistent?` receives a list of pairs of corresponding elements from two type-expressions. Based on this list, it verifies the conditions above to determine if the type-expressions which produced the list are consistently renamed. Implement `is-consistent?` and write its contract in ex3.rkt.

3. **Type-expression equivalence**

   The predicate `te-eq$` receives two type-expressions and returns true if and only if they are equivalent. Use the procedures `equal-trees$` and `is-consistent?` In order to implement `te-eq$`. Write the procedure with its contract in ex3.rkt

Question 5 – The Type-Inference system

For each of the following entries, make changes in the code supplied within the folder with their name (for example, write code answers to 5a in folder 5a/).

Theoretical answers should be placed in *ex3.pdf*.

a. Procedural implementation:
Consider the implementation of the Substitution ADT (module substitution-adt.rkt).
1. Edit the module in order to change the implementation of Substitution into a procedural one, using the lazy approach with type tags.
2. Change accordingly the implementation of the relevant getters and identifiers and correct the Type information (supplier view) in the contracts of the re-written procedures.
3. Is there a need to modify also client procedures sub-apply and sub-combine? Explain your answer.

Use the tests in the module Substitution-adt-tests.rkt to verify your answers.

b. Extending the typed language with let **forms**:
1. Suggest type-constraint equation(s) in order to type `let` expressions of Scheme.
2. Extend the procedure `make-equation-from-se` in module "scheme-expression-to-equation.rkt", so to create the suggested equations for `let` expressions.
   The abstract syntax interface procedures for `let` expressions appear in module ast.rkt.
3. Is there a need to modify also the `solve` procedure in module solve.rkt?

c. Changing the implementation of the Type-Expression ADT:
1. Consider the `match-tvars-in-te$` procedure in module type-expression-adt.rkt.

   - Joe suggests that this procedure can be considered as a client procedure and can be moved to the testing module type-expressions-adt-tests.rkt, where it can be used to test the system results, by comparing a result type expression with an expected one.
   - But Moe claims that this procedure depends on the implementation of Type-Expression, and therefore cannot function as a client procedure.

   Whose claim is right? Joe's or Moe's?

2. In order to justify your answer, change the implementation of say, `make-proc-te`, and see what happens. Submit the change as a part of the type system provided in folder 5c.