PPL 162 - Assignment 2
Syntax, Semantics and the Sequence Interface

Submission instructions:

1. Each task has clear instruction indicating in which file you answers and code must be written.
2. Submit an archive file named ***id1_id2.zip*** (which contains three files task2.rkt, task5.rkt and ex2.pdf) where *id1* and *id2* are the IDs of the students responsible for the submission (or ***id1.zip*** if there is one student in the group).
3. A contract must be included for every implemented procedure.
4. Make sure the .rkt file is in the correct format (see the announcement about "WXME format" in the course web page).
5. Use exact procedure and file names, as your code will be tested automatically.

Task 1: Applicative and Normal Order

(Provide your answer in ex2.pdf)

1.1 Evaluate the following expression according to the applicative evaluation algorithm:

```
( ( (lambda (x)                                  ;1
        (lambda (y)                              ;2
            (+ x y) ) ) 5) 4)                    ;3
```

1.2 In one of the steps of the applicative evaluation algorithms above, the value passed to the function applicative-eval is not exactly of the type "Expression". Show this value and explain what part of it is not an expression.

1.3 Evaluate the following expression according to the normal evaluation algorithm:

```
(( (lambda (x)                                   ;1
         (lambda (y)                             ;2
            (cons x y)))                         ;3
          (+ 1 2))                               ;4
   0)                                            ;5
```

1.4 Answer for each question "Correct" / "Incorrect". If the answer is "Incorrect" give a counter-example.

a. If the computation of an expression in applicative order terminates, then the computation of this expression in normal eval will also terminate.

b. If the computation of an expression in normal order terminates, then the computation of this expression in applicative eval will also terminate.

c. The computation of an expression in normal order can cause side effects that will NOT occur when the same expression is computed in applicative eval.

d. The computation of an expression in applicative order can cause side effects that will NOT occur when the same expression is computed in normal eval.

Write your answers in *ex2.pdf*.

Task 2: Types

We studied in class the "Type Language" used to describe the types of values and expressions in our programming language. The concrete syntax of the Type Language is the following BNF:

```
Type -> 'Void' | Non-void
Non-void -> Atomic | Composite | Type-variable
Atomic -> 'Number' | 'Boolean' | 'Symbol'
Composite -> Procedure | Pair | List
Procedure -> '[' Tuple '->' Type ']'
Tuple -> (Non-void '*')* Non-void | 'Empty'
Pair -> 'Pair' '(' Type ',' Type ')'
List -> 'List' '(' Type ')' | 'List'
Type-variable -> --a symbol starting with an upper case letter--
```

For each type described by an expression in the Type Language, we must have in the Programming Language a set of functions that allow the programmer to create and manipulate values of the type. These functions constitute the Type functional interface and include:

1. Value constructors
2. Type predicates
3. Equality predicates
4. Operations (including accessors)

2.1 We want to extend the Type Language and the Programming Language with a new primitive type called "**Date**" which denotes the set of legal dates in the calendar. Should the new type expression "**Date**" be an atomic or a composite type expression? Explain.

Show the required modifications to the Type Language BNF to support the new **Date** type.
What should be the contracts of the value constructor, type predicate and accessors functions that manipulate values of type **Date** in the Programming Language? Write 5 function contracts according to the DbC style (Signature, Type, Purpose, Test).

2.2 A friend claims that we do not need to add a new primitive type **Date** to the Type Language and to the Programming Language because we can already represent Date values using values of type List(Number) in the language. He also claims we can implement the **Date** functional interface, so that these functions also do not need to be primitive functions in the language.

Explain what would be the differences between a primitive type Date and its functional interface - implemented as primitive functions on one side, and a programmer defined type with programmer defined functions to implement the Date type functional interface.

Write your answers in *ex2.pdf*.

Task 3: The Sequence Interface and ADT

In this question, we will implement an Abstract Data Type (ADT) called "Relation" using the Sequence Interface of Scheme. The Relation ADT implements a restricted form of "relational algebra" which is the basis of relational databases - see https://en.wikipedia.org/wiki/Relational_algebra for a short introduction to the topic.

Recall that the Sequence Interface is an abstract functional interface over sequences which includes **map**, **filter**, **foldr**. We can also use the Racket functions **andmap** and **ormap**.

To define Relation values, we first define:

- **Row** - a single row in a relation, which contains a single record for that relation.
- **Schema** - describes the names of the attributes in a relation. Each row in the relation has a value for each of the attributes of the schema of the relation.

We define a Relation as the combination of a Schema and a list of Rows and define the types of Table and Database on top of these objects:

- **Schema**: ordered sequence of symbols without repetition. Each symbol is used as the name of an attribute.
- **Row**: ordered sequence of atomic values.
- **Relation**: a composite value which contains a Schema and a set of Rows. Every row in the list must be consistent with the Schema of the relation - that is, the row must have the same number of values as there are attributes in the Schema.
- **Table**: a named relation - it contains a name and relation.
- **Database**: a collection of tables. All tables must have distinct names within a database, and in addition, for simplification, we impose that the schemas of all the table relations are disjoint (that is, there is no attribute name that appears in two different tables in a single database).

The code in **relational-algebra.rkt** contains the skeleton for the answer to this part.

The concrete representation of a database using the implementation of the Relation, Table and Database ADTs in this file is the following. Note that we use quoted symbols inside rows to represent symbolic values.

```
(define DB1
  '(
    (Students
     (S_StudentId S_Name S_Course)
     (0 'Avi 'CS330)
     (1 'Yosi 'CS142)
     (2 'Sarah 'CS630))
    (Projects
     (P_ProjectId P_Course P_Title)
     (0 'CS330 'Intro)
     (1 'CS330 'PPL)
     (2 'CS142 'SPL)
     (3 'CS630 'Compilation))
    (Grades
     (G_StudentId G_ProjectId G_Grade)
     (0 0 10)
     (1 2 99)
     (2 3 60))))
```

3.1 Type Predicates

Define the following type predicates in the Relational Algebra ADT using the Sequence Interface.

schema?(x): verify that an object is a valid Schema (no repeated values).
relation?(x): verify that an object is a valid Relation:
- The relation schema is valid
- The rows are all consistent with the schema (same length as the schema)

table?(x): verify that an object is a valid Table
database?(x): verify that an object is a valid Database
- Every table in the database is valid
- There are no tables in the database that have the same name
- The schema of all the table relations are pairwise disjoint

Note: in pure Relational Algebra, there cannot be repeated rows in a relation - that is, the following relation is not well formed:

```
'((a b c)
  (1 2 3)
  (1 2 3))
```

We do NOT need to check this constraint in this assignment.

Requirements:

- Define the contract for each function.
- Use ONLY the Sequence Interface methods to implement these functions over lists. In particular, use **map, filter, foldr, andmap and ormap** - do NOT write recursive functions to iterate over the lists.

3.2 Functional Row Accessors

Consider the code in Relational-Algebra.rkt that defines the functions row-accessors and row-accessor. Complete the Type and the Test parts of their contract.

3.3 Relation Operations

Implement the following operations over Relations:

1. **project**
2. **select**
3. **product**
4. **join**
5. **union**

The contract of each function is provided in **relational-algebra.rkt**.

Requirements:

- Use ONLY the Sequence Interface methods to implement these functions over lists. In particular, use **map, filter, foldr, andmap and ormap** - do NOT write recursive functions to iterate over the lists.

Task 4: Abstract Syntax and Interpreters

In this question, we will develop an interpreter for a query language that allows computations over Relations, similar to SQL.

The code of the interpreter is organized in 3 modules:

- **relational-algebra.rkt**: this provides the Relation, Table and Database ADTs (as developed in Task 3). It implements the "primitive" functions of the SQL language - the datatypes that implement Relation values, and operations to manipulate Relations.
- **sql-asp.rkt**: this provides the Abstract Syntax and Parser for the SQL language.
- **sql-interpreter.rkt**: this provides the interpreter for the SQL language. This module uses **sql-asp.rkt** to manipulate SQL expressions, and **relational-algebra.rkt** to manipulate Relation values.

The following BNF describes the concrete syntax of our SQL language:

```
<query> -> '(' 'select' '(' <attribute>+ ')' 'from' <relation-exp>
           ['where' <predicate>] ')' []optional - appears 0 or 1 times
<attribute> -> symbol
<relation-exp> -> <table-var> | <composite-relation>
<table-var> -> symbol
<composite-relation> -> '(' <relation-exp> 'product' <relation-exp> ')' |
          '(' <relation-exp> 'join' <relation-exp> 'on' <predicate> ')'
<predicate> -> '(' <operand> <operator> <operand> ')'
<operand> -> <attribute> | <primitive>
<primitive> -> boolean | number | <quoted-symbol>
<quoted-symbol> -> '(' 'quote' symbol ')'
<boolean> -> '#t' | '#f'
<op> -> '=' | 'eq?' | '>' | '<'
```

The following are example expressions in this language:

```
(select (S_StudentId S_Name) from Students where (S_StudentId = 0))

(select (S_StudentId S_Name) from Students where (S_Name = 'Yosi))

(select (S_StudentId G_Grade) from (Students product Grades)
        where (G_Grade < 90))

(select (S_StudentId G_Grade)
        from (Students join Grades on (S_StudentId = G_StudentId))
        where (G_Grade > 56))

(select (S_StudentId S_Course G_Grade)
        from (Students join (Projects join Grades
                                       on (P_ProjectId = G_ProjectId))
                        on (S_StudentId = G_StudentId)))
```

Note that relation expressions can be embedded in a recursive manner - predicates cannot.

The abstract syntax of the language is presented below:

```
<query> :: Components: attributes, relation-exp, predicate
<relation-exp> :: Kinds: <table-var>, <composite-relation>
<attribute> :: symbol
<table-var> :: symbol
<composite-relation> ::  <product> | <join>
<product> :: Components: relation1: <relation-exp>, relation2: <relation-exp>
<join> :: Components: relation1, relation2: <relation-exp>, predicate: <predicate>
<predicate> :: Components: op1, op2: <operand>, operator: <op>
<operand> :: Kinds: <attribute>, <primitive>
<primitive> :: Kinds: <boolean> | <quoted-symbol> | <number>
<op> :: Kinds: a member of '(= eq? < >)
```

4.1 Abstract Syntax

The current version of the BNF and Abstract Syntax in **sql-asp.rkt** supports Composite Relation expressions of type **product** and **join**.

Extend the BNF and the Abstract Syntax to support a new type of Composite Relation expressions called **union**.

Provide:

1. the new BNF rule needed to define **union** expressions
2. the new Abstract Syntax definition for **union**
3. the new functions needed in sql-asp.rkt to define the Abstract Syntax type for **union** expressions
4. the updated functions in the parser **parse-relation** and **parse-union**
5. the updated **relation->concrete** function.

Note: in **sql-asp.rkt**, the function **query->concrete** and the other functions of the type **<x>->concrete** are used to serialize abstract syntax values into their concrete syntax representation. Their function is to operate the inverse operation of parse - that is, **(query->concrete (parse-query q)) === q**.

Draw the Abstract Syntax Tree for the query **q4** defined in file **sql-interpreter.rkt** - add the drawing in ex2.pdf:

```
(select (S_StudentId S_Course G_Grade)
        from (Students join (Projects join Grades
                                       on (P_ProjectId = G_ProjectId))
                           on (S_StudentId = G_StudentId)))
```

4.2 Interpreter

The operational semantics of the SQL is defined in the following algorithm, which is similar to the applicative-eval algorithm defined in p.87 of the course book.

```
def eval-query(query, db):
   let attributes = query-attributes(query)
       relation = eval-relation(query-relation(query), db)
       pred = eval-predicate(query-predicate(query), relation-schema(relation))
     eval-query(query, db) = project(attributes, select(pred, relation))

def eval-relation(relation, db):
   I.  table-var?(relation):
       eval-relation(relation, db) = table-relation(db-table, table-var-table(relation, db))
   II. composite?(relation):
     1. product?(relation):
        eval-relation(relation, db)= product(eval-relation(product-relation1(relation), db)
                                             eval-relation(product-relation2(relation), db))
     2. join?(relation):
        let rel1 = eval-relation(join-relation1(relation), db)
            rel2 = eval-relation(join-relation2(relation), db)
          eval-relation(relation, db) = join(rel1, rel2,
                                        eval-predicate(
                                          join-predicate(relation),
                                          make-schema(append(relation-schema(rel1),
                                                             relation-schema(rel2)))))

def eval-predicate(pred, schema):
  let arg1 = eval-operand(predicate-op1(pred), schema)
      arg2 = eval-operand(predicate-op2(pred), schema)
      operator = eval-operator(predicate-operator(pred))
   eval-predicate(pred db) = (lambda (row) (operator (arg1 row) (arg2 row)))

def eval-operand(op, schema):
  I.  attribute?(op):
      eval-operand(op, schema) = row-accessor(op, schema)
  II. primitive?(op):
      eval-operand(op, schema) = (lambda (row) op)

def eval-operator(op):
  1. op = 'eq?: eval-operator(op) = equal?
  2. op = '<:   eval-operator(op) = <
  3. op = '>:   eval-operator(op) = >
  4. op = '=:   eval-operator(op) = =
```

4.2.1 Is the evaluation order of this interpreter more similar to **applicative-eval** or to **normal-eval**? Explain which lines in the definition of the algorithm make you conclude. (Answer in ex2.pdf).

4.2.2 When evaluating **eval-query(q, db)**, the current interpreter does not check that the **Attribute** names that appear in the query **q** match those that are declared in the schema of the database tables.

Would this checking of attributes be a **syntactic** or a **semantic** process? Explain. (Answer in ex2.pdf).

4.2.3 The current semantics of the SELECT expression is to evaluate:

eval-query(query, db) = project(query.attributes, select(query.pred, query.relation))

That is, we first execute the **select** operation, then we execute the **project** operation.
What would be different if we were to execute instead:

select(query.pred, project(query.attributes, query.relation))

Give an example query that would behave differently using the two different strategies of evaluation.

4.2.4 Implement in sql-interpreter.rkt the missing functions marked with @@:

eval-query and **eval-relation** according to the interpreter algorithm listed above.

4.2.5 Extend the sql-interpreter to support Query expressions that include **union**.

Good luck, Have fun!