Homework: TypeLang

Learning Objectives:

Implement a type system for the most advanced language learned so far: RefLang.

Instructions:

- Total points 51 pt
- Early deadline: Nov 20 (Wed) 2019 at 11:59 PM; Regular deadline: Nov 22 (Fri) 2019 at 11:59 PM (you can continue working on the homework till TA starts to grade the homework)
- Download hw8code.zip from Canvas. Interpreter for Typelang is significantly different compared to previous interpreters:
  - Env in Typelang is generic compared to previous interpreters.
  - Two new files Checker.java and Type.java have been added
  - Type.java defines all the valid types of Typelang.
  - Checker.java defines type checking semantics of all expressions.
  - Typelang.g has changed to add type information in expressions. Please review the changes in file to understand the syntax.
  - Finally Interpreter.java has been changed to add type checking phase before evaluation of Typelang programs.
- Set up the programming project following the instructions in the tutorial from hw2 (similar steps)
- Extend the Typelang interpreter for Q1 - Q6.
- How to submit:
  - Please submit your solutions in one zip file with all the source code files (just zip the complete project's folder).
  - Write your solutions to question 8 in a "hw8.scm" file and store it under your code directory.

```
🟥 hw8code
└─ 🟦 typelang
   └─ 🟩 src
      └─ 🟫 main
         └─ 🟧 java
            └─ 🟥 typelang
               └─ ⬛ examples
```

  - Submit the zip file to Canvas under Assignments, Homework 8.

Questions:

1. (8 pt) Implement type rules for the memory related expressions:

   (a) (4 pt) RefExp: Let a ref expression be (ref e1), where e1 is an expression.

   - if e1's type is ErrorT then (ref e1)'s type should be ErrorT
   - if e1's type is T then (ref e1)'s type should be RefT.
   - otherwise, (ref e1)'s type is ErrorT with message "The Ref expression expect type " + T + "found " + e1's type + " in " + expression.

   Note that you have to add e1's type and expression in the error message. Examples:
   $ (ref : num 45)
   loc:0
   // no explicit error cases
   $ (ref : bool 45)
   Type error: The Ref expression expect type bool found number in (ref 45.0)

   (b) (4 pt) AssignExp: Let a set expression be (set! e1 e2), where e1 and e2 are expressions.

   - if e1's type is ErrorT then (set! e1 e2)'s type should be ErrorT
   - if e1's type is RefT and nestedType of e1 is T then
     - if e2's type is ErrorT then (set! e1 e2)'s type should be ErrorT
     - if e2's type is T, then (set! e1 e2)'s type should be e2's type T
     - otherwise (set! e1 e2)'s type is ErrorT with message "The inner type of the reference type is " + nestedType T + " the rhs type is " + e2's type + " in " + expression
   - otherwise (set! e1 e2)'s type is ErrorT with message "The lhs of the assignment expression expect a reference type found " + e1's type + " in "+ expression.

   Note that you have to add e1's and e2's type and expression in the error message. Examples:
   $ (set! (ref : num 0) #t)
   Type error: The inner type of the reference type is number the rhs type is bool in (set! (ref 0) #t)
   $ (set! (ref : bool #t) (list : num 1 2 3 4 5 6))
   Type error: The inner type of the reference type is bool the rhs type is List<number> in (set! (ref #t) (list 1 2 3 4 5 6))

2. (8 pt) Implement type checking rules for the list expressions

   (a) (4 pt) CdrExp: Let a cdr expression be (cdr e1), where e1 is an expression.

   - if e1's type is ErrorT then (cdr e1)'s type should be ErrorT
   - if e1's type is PairT then (cdr e1)'s type should be the type of the second element of the pair
   - otherwise, (cdr e1)'s type is ErrorT with message "The cdr expect an expression of type Pair, found"+ e1's type+ "in" + expression

   Note that you have to add e1's type and expression in the error message. See some examples below.
   $ (cdr 2)
   Type error: The car expect an expression of type Pair, found number in (cdr 2.0)

```
$ (cdr (cdr 2))
Type error: The cdr expect an expression of type Pair, (cdr (cdr 2))
```

(b) (4 pt) ListExp: Let a list expression be (list : T e1 e2 e3 ... en), where T is type of list and e1, e2, e3 ... en are expressions

- if type of any expression ei, where ei is an expression of element in list at position i, is ErrorT then type of (list : T e1 e2 e3 ... en) is ErrorT.
- if type of any expression ei, where ei is an expression of an element of list, is not T then type of (list : T e1 e2 e3 ... en) is ErrorT with message ”The ” + index + ” expression should have type ” + T + ” found ” + Type of ei + ” in ” + ”expression”. where index is the position of expression in list's expression list.
- else type of (list : T e1 e2 e3 ... en) is ListT.

Note that you have to add ei's type and expression in the error message. Some examples appear below.

```
$ (list : bool 1 2 3 4 5 6 7)
Type error: The 0 expression should have type bool, found number in (list 1 2 3 4 5 6 7 )
$ (list : num 1 2 3 4 5 #t 6 7 8)
Type error: The 5 expression should have type number, found bool in (list 1 2 3 4 5 #t 6 7 8)
```

3. (4 pt) Implement typing rules for the CompoundArithExp expressions.

Let a CompoundArithExp be (ArithExp e1 e2 e3 ... en), where e1, e2, e3... en are expressions.

- if type of any expression ei, where ei is an expression of element in list at position i, is ErrorT then type of (list : T e1 e2 e3 ... en) is ErrorT.
- if type of any expression ei, where ei is an expression of element in list at position i, is not NumT then type of (list : T e1 e2 e3 ... en) is ErrorT with message: ”expected num found ” + ei's type + ” in ” + expression
- else type of (ArithExp e1 e2 e3 ... en) is NumT.

Note that you have to add ei's type and expression in the error message. Some examples appear below.

```
$ (+ #t 6)
Type error: expected num found bool in (+ #t 6 )
$ (+ 5 6 7 #t 56)
Type error: expected num found bool in (+ 5 6 7 #t 56 )
$ (* 45.0 #t)
Type error: expected num found bool in (* 45.0 #t )
$ (/ (list : num 3 4 5 6 7) 45)
Type error: expected num found List<number> in (/ (list 3 4 5 6 7 ) 45 )
```

4. (4 pt) Implement type rules for the comparison expressions:

BinaryComparator: Let a BinaryComparator be (binary operator e1 e2), where e1 and e2 are expressions.

- if e1’s type is ErrorT then (binary operator e1 e2)’s type should be ErrorT
- if e2’s type is ErrorT then (binary operator e1 e2)’s type should be ErrorT
- if e1’s type is not NumT then (binary operator e1 e2)’s type should be ErrorT with message : ”The first argument of a binary expression should be num Type, found ” + e1’s type + ” in ” + expression.
- if e2’s type is not NumT then (binary operator e1 e2)’s type should be ErrorT with message : ”The second argument of a binary expression should be num Type, found ” + e2’s type + ” in ” + expression.
- otherwise (binary operator e1 e2)’s type should be BoolT.

Note that you have to add e1’s and e2’s type and expression in the error message. Some examples appear below.

```
$ (< #t #t)
Type error: The first argument of a binary expression should be num Type, found bool in (< #t #t)
$ (> (list: num 45 45 56 56 67) 67)
Type error: The first argument of a binary expression should be num Type, found List<number> in (> (list 45 45 56 56 67 ) 67)
```

5. (6 pt) Implement type checking rules for the conditions expressions.

   IfExp: Let a IfExp be (if cond then else), where cond, then, else are expressions.

   - if cond’s type is ErrorT then (if cond then else)’s type should be ErrorT
   - if cond’s type is not BoolT then (if cond then else)’s type should be ErrorT with message: ”The condition should have boolean type, found ” +cond’s type+ ” in ” + expression
   - if then’s type is ErrorT then (if cond then else)’s type should be ErrorT
   - if else’s type is ErrorT then (if cond then else)’s type should be ErrorT
   - if then’s type and else’s type are typeEqual then (if cond then else)’s type should be then’s type.
   - else (if cond then else)’s type should be ErrorT with message: ”The then and else expressions should have the same ” + ”type, then has type ” + then’s type + ” else has type ” +else’s type+ ” in ” + expression.

   Note that you have to add cond’s, then’s and else’s type and expression in the error message. Some examples appear below.

```
$ (if 5 56 67)
Type error: The condition should have boolean type, found number in (if 5 56 67)
$ (if #t #t 56)
Type error: The then and else expressions should have the same type, then has type bool else has type number in (if #t #t 56)
```

6. (6 pt) Implement type checking rules of the let expressions

   LetExp: Let a let expression be (let ((e1 : T1 V1) (e2: T2 V2)... (en: Tn Vn)) body) where e1, v1, e2, v2 ... en, vn are expressions

- if type of any expression Vi, where Vi is an expression of value_expression of some variable in (let ((e1 : T1 V1) (e2: T2 V2) ... (en: Tn Vn)) body), is ErrorT then type of (let ((e1 : T1 V1) (e2: T2 V2) ... (en: Tn Vn)) body) is ErrorT.
- If type of any expression Vi does not match the type Ti, the type of the LetExp is ErrorT
- Type of (let ((e1 : T1 V1) (e2: T2 V2)... (en: Tn Vn)) body) is the type of the let expression body evaluated in the extended environment (environment containing mapping of declared variable and types)

Note that you have to add ei’s and Vi’s type and expression in the error message. Some examples appear below.

```
$ (let ((x: num 34) (y : num 45) (cond: bool #t)) (if x (+ x y) (/ x y)))
Type error: The condition should have boolean type, found number in (if x (+ x y ) (/ x y ))

$ (let ((x: num #t) (y: bool 8)) x)
Type error: The declared type of the 0 let variable and the actual type mismatch, expect number type, found boolean in (let ((x: num #t) (y: bool 8)))
```

7. (6 pt) Implement type checking rules for the function calls.

   CallExp: Let a call expression be (ef e1 ... en), where ef, e1,... and en are expressions:

   - if the type of ef is ErrorT, return ErrorT
   - if the type of ef is not FuncT, the type of the call expression is ErrorT, reporting the message ”Expect a function type in the call expression, found ”+ef’s type+”in ”+ expression
   - if any one of e1, e2, ...en has ErrorT, the call expression has ErrorT
   - given that ef has FuncT (T1 ... Tn)->Tb, if the actual parameter ei does not have a type Ti, the call expression has ErrorT, reporting the message ”The expected type of the ” + i + ”th actual parameter is ” + Ti + ”, found ” + ei’s type +”in ”+expression
   - otherwise, the type of call expression is Tb

   Some examples appear below.

   ```
   $(define add: (num num num − > num) (lambda (x: num y: num z: num) (+ x (+ y z))))
   $ (add 5 56 #t)
   Type error: The expected type of the third actual parameter is number, found bool in (add 5 56 #t)
   $ (3 4)
   Type error: Expect a function type in the call expression, found number in (3 4)
   ```

8. (9 pt) For all the above typing rules (total 9 of them) you implement, write a typelang program for each type rule to test and demonstrate your type check implementation. (You can use typelang.g in hw8code.zip as a reference for the syntax of TypeLang). For each expression, put in comments which type rules the expression is exercising. For example:

   ```
   $ (list: num 45 45 56 56 67) // test correct types for list expressions
   $ (* 45.0 #t) // test incorrect types for compound arithmetic expressions
   ```