Homework: TypeLang

Instructions:

- Due Date: Nov 18th 2016 (Fri) 6:00 pm
- In this homework, you will learn about types and typechecking by writing well-typed programs and by extending the Typelang interpreter to add type checking rules for some of the expressions.
- Download and install a *fresh* copy of Typelang interpreter from lecture notes. 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.
- For Q1 write your answers in HW8.scm and save it as src/typelang/examples/HW8.scm
- Extend the Typelang interpreter for Q2 - Q6.
- Export your homework solution as archive file (File->Export->General->archive). Attach the archive file. Please provide the complete project as submission not just the src directory.

Questions:

1. (25 pt) The goal of the question is to help you review and deep the understandings of the concepts you learn in the Typelang lecture.
   - (10 pt) Please list all the concepts you have learned about types. For each of the concept, provide one or two sentences to explain your understandings on what are they.
   - (10 pt) In formal discussions about Type, we do not discuss the value produced by type checking rules, but the implementation typechecking rules produce a type ErrorT that encapsules a, hopefully helpful, message to the programmer. Write down all conditions under which rules in Checker.java produces a type ErrorT as follows.

Term	Condition
Program	When the declared type of the define decl doesnt match with expressions type.

   - (5 pt) How is the initial type environment constructed in Checker.java different from the initial environment constructed in Evaluator.java? Brief answers please.

Sol

- concepts:
  - type
  - type system
  - type inference
  - type checking
- ErrorT, see table 1
- In checker.java empty environment is extended to map ”read” and ”eval” with their corresponding types while in Evaluator.java empty environment is extended to map ”read”, ”eval” and ”require” variables with their corresponding values.

2. (40 points) In this question you will extend the typechecking rules of TypeLang to add rules for checking correctness of list related expressions. Implement the typechecking rules of the following expressions to realize the correctness criteria that follows.

   (10 points)

   (a) (10 points)
   CarExp: Let a car expression be (car e1), where e1 is an expression.
   - if e1s type is ErrorT then (car e1)s type should be ErrorT
   - if e1s type is PairT then (car e1)s type should be the type of the first element of the pair
   - otherwise, (car e1)s type is ErrorT with message ”The car expect an expression of type Pair, found”+ e1s type+ ”in” + expression

   Note that you have to add e1s type and expression in the error message. Some examples appear below.
   $ (car ”Hello”)
   Type error: The car expect an expression of type Pair, found string in (car ”Hello”)
   $ (car (car ”Hello”))
   Type error: The car expect an expression of type Pair, found string in (car ”Hello”)

   (b) (10 points)
   CdrExp: Let a cdr expression be (cdr e1), where e1 is an expression.
   - if e1s type is ErrorT then (cdr e1)s type should be ErrorT
   - if e1s 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”+ e1s type+ ”in” + expression

   Note that you have to add e1s type and expression in the error message. Some examples appear below.
   $ (cdr ”Hello”)
   Type error: The cdr expect an expression of type Pair, found string in (cdr ”Hello”)
   $ (car (cdr ”Hello”))
   Type error: The cdr expect an expression of type Pair, found string in (cdr ”Hello”)

Term	Condition
Program	When the declared type of the define decl doesnt match with expressions type.

AssignExp	if `lhs_exp` evaluates to ErrorT
	if `rhs_exp` evaluates to ErrorT
	if `rhs_exp`'s type doesnt match with `lhs_exp`'s nested type
	if `lhs_exp` evaluates to type other than RefT

CallExp	if operator evaluates to ErrorT if operator evaluates to type other than FuncT if number of actuals does not match with number of formals if any of the actual parameters evaluates to ErrorT if any of the actual parameters is not assignable to declared type of corresponding formal
CarExp	if expression insided CarExp evaluates to ErrorT if expression insided CarExp evaluates to type other than PairT
CdrExp	if expression insided CdrExp evaluates to ErrorT if expression insided CdrExp evaluates to type other than PairT
ConsExp	if fst expression of ConsExp evaluates to ErrorT if snd expression of ConsExp evaluates to type other than PairT
DerefExp	if location expression of DerefExp evaluates to ErrorT if location expression of DerefExp evaluates to type other than RefT
FreeExp	if location expression of FreeExp evaluates to ErrorT if location expression of FreeExp evaluates to type other than RefT
IfExp	if condType expression of IfExp evaluates to ErrorT if condType expression of IfExp evaluates to type other than BoolT if thentype expression of IfExp evaluates to ErrorT if elsetype expression of IfExp evaluates to ErrorT if type of thentype expression and type of elsetype expression do not match
LambdaExp	When the number of actuals and the number of formals are not equal
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 - 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), does not match with type of corresponding ei
LetrecExp	Same as LetExp
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 - if type of any expression ei, where ei is an expression of an element of list, is not assignable to T
NullExp	Let a null expression be (null? e1), where e1 is an expression. - if e1s type is ErrorT - if e1s type is not ListT
RefExp	Let a ref expression be (ref: T e1), where e1 is an expression. - if e1s type is ErrorT - if e1s type is not assignable to T
VarExp	When the variable has not been declared
BinaryComparator	Let a BinaryComparator be (binary operator e1 e2), where e1 and e2 are expressions. - if e1s type is ErrorT - if e2s type is ErrorT - if e1s type is not NumT - if e2s type is not NumT
CompoundArithExp	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 - if type of any expression ei, where ei is an expression of element in list at position i, is not NumT
VarExp	When the variable has not been declared
AddExp	same as CompoundArithExp
SubExp	same as CompoundArithExp
MultExp	same as CompoundArithExp
DivExp	same as CompoundArithExp
EqualExp	When the variable has not been declared
GreaterExp	When the variable has not been declared

Table 1: ErrorT

(c) (10 points)
ConsExp: Let a cons expression be (cons e1 e2), where e1 and e2 are expressions.

- if either e1s or e2's type is ErrorT then (cons e1 e2)s type should be ErrorT
- if e1s type is T1 and e2's type is T2 then (cons e1 e2)s type is PairT with T1 and T2 being elements of PairT.

Note that you have to add e1s type and expression in the error message. Some examples appear below.
$ (cons #t (list : num 1 2 3 4 5 6 #t))
Type error: The 6 expression should have type number found bool in (list 1.0 2.0 3.0 4.0 5.0 6.0 #t)
$ (cons (car (cdr "Hello")) "testing")
Type error: The cdr expect an expression of type Pair, found string in (cdr "Hello")

(d) (10 points)
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 eis 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 1.0 2.0 3.0 4.0 5.0 6.0 7.0)
$ (list : num 1 2 3 4 5 #t 6 7 8)
Type error: The 5 expression should have type number found bool in (list 1.0 2.0 3.0 4.0 5.0 #t 6.0 7.0 8.0)

Sol

- carexp

```
    public Type visit(CarExp e, Env<Type> env) {
      Exp exp = e.arg();
      Type type = (Type)exp.accept(this, env);
      if (type instanceof ErrorT) { return type; }

      if (type instanceof PairT) {
        PairT pt = (PairT)type;
        return pt.fst();
      }

```

```
        return new ErrorT("The car expect an expression of type Pair, found "
        + type.tostring() + " in " + ts.visit(e, null));
      }
```

- cdrexp

```
      public Type visit(CdrExp e, Env<Type> env) {
        Exp exp = e.arg();
        Type type = (Type)exp.accept(this, env);
        if (type instanceof ErrorT) { return type; }

        if (type instanceof PairT) {
          PairT pt = (PairT)type;
          return pt.snd();
        }

        return new ErrorT("The cdr expect an expression of type Pair, found "
        + type.tostring() + " in " + ts.visit(e, null));
      }
```

- consexp

```
      public Type visit(ConsExp e, Env<Type> env) {
        Exp fst = e.fst();
        Exp snd = e.snd();

        Type t1 = (Type)fst.accept(this, env);
        if (t1 instanceof ErrorT) { return t1; }

        Type t2 = (Type)snd.accept(this, env);
        if (t2 instanceof ErrorT) { return t2; }

        return new PairT(t1, t2);
      }
```

- listexp

```
      public Type visit(ListExp e, Env<Type> env) {
        List<Exp> elems = e.elems();
        Type type = e.type();

        int index = 0;
        for (Exp elem : elems) {
          Type elemType = (Type)elem.accept(this, env);
          if (elemType instanceof ErrorT) { return elemType; }

          if (!assignable(type, elemType)) {
            return new ErrorT("The " + index +
            " expression should have type " + type.tostring() +
            " found " + elemType.tostring() + " in " +
            ts.visit(e, null));
          }
          index++;
        }
        return new ListT(type);
      }
```

3. (50 points)In this question you will extend the typechecking rules of TypeLang to add rules for checking correctness of memory related expressions. Implement the typechecking rules of following expressions to realize the correctness criteria that follows.

   (a) (10 points)
   NullExp: Let a null expression be (null? e1), where e1 is an expression.

   - if e1s type is ErrorT then (null? e1)s type should be ErrorT
   - if e1s type is ListT then (null? e1)s type should be BoolT.
   - otherwise, (null? e1)s type is ErrorT with message "The null? expect an expression of type List, found " + e1's type + " in " + expression.

   Note that you have to add e1s type and expression in the error message. Some examples appear below.
   $ (null? 5)
   Type error: The null? expect an expression of type List, found number in (null? 5.0)
   $ (null? #t)
   Type error: The null? expect an expression of type List, found bool in (null? #t)

   (b) (10 points)
   RefExp: Let a ref expression be (ref: T e1), where e1 is an expression.

   - if e1s type is ErrorT then (ref: T e1)s type should be ErrorT
   - if e1s type is T then (ref: T e1)s type should be RefT with _nestType T. Note that _nestType is a field in RefT.
   - otherwise, (ref: T 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 e1s type and expression in the error message. Some examples appear below.
   $ (ref : bool 3)
   Type error: The Ref expression expect type bool found number in (ref 3.0)
   $ (ref : num (list : num 1 2 3 4))
   Type error: The Ref expression expect type number found List<number> in (ref (list 1.0 2.0 3.0 4.0))

   (c) (10 points)
   DerefExp: Let a deref expression be (deref e1), where e1 is an expression.

   - if e1s type is ErrorT then (deref e1)s type should be ErrorT
   - if e1s type is RefT then (deref e1)s type should RefT.nestType(). Note that nestType() is method in RefT class.
   - otherwise, (deref e1)s type is ErrorT with message "The dereference expression expect a reference type " + "found " + e1's type + " in " + expression

   Note that you have to add e1s type and expression in the error message. Some examples appear below.
   $ (deref (ref : num 45))
   45

// no explicit error cases

(d) (10 points)
AssignExp: Let a set expression be (set! e1 e2), where e1 and e2 are expressions.

- if e1s type is ErrorT then (set! e1 e2)s type should be ErrorT
- if e1s type is RefT and nestedType of e1 is T then
  - if e2s type is ErrorT then (set! e1 e2)s type should be ErrorT
  - if e2's type is typeEqual To T then (set! e1 e2)s type should be e2's type.
  - 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 e1s and e2's type and expression in the error message. Some examples appear below.
$ (set! (ref : num 0) #t)
Type error: The inner type of the reference type is number the rhs type is bool in (set! (ref 0.0) #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 1.0 2.0 3.0 4.0 5.0 6.0))

(e) (10 points)
FreeExp: Let a free expression be (free e1), where e1 is an expression.

- if e1s type is ErrorT then (free e1)s type should be ErrorT.
- if e1s type is RefT then (free e1)s type should be UnitT.
- otherwise, (free e1)s type is ErrorT with message "The free expression expect a reference type found " + e1's type + " in " + expression.

Note that you have to add e1s type and expression in the error message. Some examples appear below.
$ (free 5)
Type error: The free expression expect a reference type found number in (free 5.0)
$ (free #t)
Type error: The free expression expect a reference type found bool in (free #t)

Sol

- NullExp

```
      public Type visit(NullExp e, Env<Type> env) {
        Exp arg = e.arg();
        Type type = (Type)arg.accept(this, env);
        if (type instanceof ErrorT) { return type; }

        if (type instanceof ListT) { return BoolT.getInstance(); }

        return new ErrorT("The null? expect an expression of type List, found "
        + type.tostring() + " in " + ts.visit(e, null));
      }
```

- RefExp

```
      public Type visit(RefExp e, Env<Type> env) {
        Exp value = e.value_exp();
        Type type = e.type();
        Type expType = (Type)value.accept(this, env);
        if (type instanceof ErrorT) { return type; }

        if (expType.typeEqual(type)) {
          return new RefT(type);
        }

        return new ErrorT("The Ref expression expect type " + type.tostring() +
        " found " + expType.tostring() + " in " + ts.visit(e, null));
      }
```

- DerefExp

```
      public Type visit(DerefExp e, Env<Type> env) {
        Exp exp = e.loc_exp();
        Type type = (Type)exp.accept(this, env);
        if (type instanceof ErrorT) { return type; }

        if (type instanceof RefT) {
          RefT rt = (RefT)type;
          return rt.nestType();
        }

        return new ErrorT("The dereference expression expect a reference type " +
        "found " + type.tostring() + " in " + ts.visit(e, null));
      }
```

- AssignExp

```
      public Type visit(AssignExp e, Env<Type> env) {
        Exp lhs_exp = e.lhs_exp();
        Type lhsType = (Type)lhs_exp.accept(this, env);
        if (lhsType instanceof ErrorT) { return lhsType; }

        if (lhsType instanceof RefT) {
          Exp rhs_exp = e.rhs_exp();
          Type rhsType = (Type)rhs_exp.accept(this, env);
          if (rhsType instanceof ErrorT) { return rhsType; }

          RefT rt = (RefT)lhsType;
          Type nested = rt.nestType();

          if (rhsType.typeEqual(nested)) { return rhsType; }

          return new ErrorT("The inner type of the reference type is " +
            nested.tostring() + " the rhs type is " + rhsType.tostring()
            + " in " + ts.visit(e, null));
        }
        return new ErrorT("The lhs of the assignment expression expect a "
          + "reference type found " + lhsType.tostring() + " in " +
          ts.visit(e, null));
      }
```

- FreeExp

```
      public Type visit(FreeExp e, Env<Type> env) {
        Exp exp = e.value_exp();
        Type type = (Type)exp.accept(this, env);

        if (type instanceof ErrorT) { return type; }

        if (type instanceof RefT) { return UnitT.getInstance(); }

        return new ErrorT("The free expression expect a reference type " +
        "found " + type.tostring() + " in " + ts.visit(e, null));
      }
```

4. (25 points) In this question you will extend the typechecking rules of TypeLang to add rules for checking correctness of comparison expressions. Implement the typechecking semantics of following expressions to realize the correctness criteria that follows.

   (a) (10 points)
   BinaryComparator: Let a BinaryComparator be (binary operator e1 e2), where e1 and e2 are expressions.

   - if e1s type is ErrorT then (binary operator e1 e2)s type should be ErrorT
   - if e2s type is ErrorT then (binary operator e1 e2)s type should be ErrorT
   - if e1s 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 e2s 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 e1s 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 45.0 45.0 56.0 56.0 67.0) 67.0)

   (b) (15 points)
   Look at the inheritance tree of the comparison expressions from AST.java file and write rules for checking correctness of

   i. GreaterExp : Follows semantics of BinaryComparator.

ii. EqualExp : Follows semantics of BinaryComparator.

iii. LessExp : Follows semantics of BinaryComparator.

Sol

- BinaryComparator

```
      private Type visitBinaryComparator(BinaryComparator e, Env<Type> env,
      String printNode) {
        Exp first_exp = e.first_exp();
        Exp second_exp = e.second_exp();

        Type first_type = (Type)first_exp.accept(this, env);
        if (first_type instanceof ErrorT) { return first_type; }

        Type second_type = (Type)second_exp.accept(this, env);
        if (second_type instanceof ErrorT) { return second_type; }

        if (!(first_type instanceof NumT)) {
          return new ErrorT("The first argument of a binary expression "
          + "should be num Type, found " + first_type.tostring() +
          " in " + printNode);
        }

        if (!(second_type instanceof NumT)) {
          return new ErrorT("The second argument of a binary expression "
          + "should be num Type, found " + second_type.tostring() +
          " in " + printNode);
        }

        return BoolT.getInstance();
      }
```

- GreaterExp,EqualExp,LessExp

```
      public Type visit(GreaterExp e, Env<Type> env) {
        return visitBinaryComparator(e, env, ts.visit(e, null));
      }
      public Type visit(EqualExp e, Env<Type> env) {
        return visitBinaryComparator(e, env, ts.visit(e, null));
      }
      public Type visit(LessExp e, Env<Type> env) {
        return visitBinaryComparator(e, env, ts.visit(e, null));
      }
```

5. (30 points) In this question you will extend the typechecking rules of TypeLang to add rules for checking correctness of CompoundArithExp expressions. Implement the typechecking semantics for following expressions to realize the correctness criteria that follows.

   (a) (10 points)
   CompoundArithExp: 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 6.0 )
$ (+ 5 6 7 #t 56)
Type error: expected num found bool in (+ 5.0 6.0 7.0 #t 56.0 )
$ (* 45 #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 3.0 4.0 5.0 6.0 7.0 ) 45.0 )
```

(b) (20 points)
Look at the inheritance tree of the comparison expressions from AST.java file and write rules for checking correctness of

i. AddExp : Follows semantics of BinaryComparator.
ii. SubExp : Follows semantics of BinaryComparator.
iii. MultExp: Follows semantics of BinaryComparator.
iv. DivExp : Follows semantics of BinaryComparator.

Sol

- CompoundArithExp

```java
    private Type visitCompoundArithExp(CompoundArithExp e, Env<Type> env, String
        printNode) {
      List<Exp> operands = e.all();

      for (Exp exp: operands) {
        Type intermediate = (Type) exp.accept(this, env); // Static type-checking
        if (intermediate instanceof ErrorT) { return intermediate; }

        if (!(intermediate instanceof Type.NumT)) {
          return new ErrorT("expected num found " + intermediate.tostring() +
          " in " + printNode);
        }
      }

      return NumT.getInstance();
    }
```

- AddExp, SubExp,MultExp,DivExp

```java
    public Type visit(AddExp e, Env<Type> env) {
      return visitCompoundArithExp(e, env, ts.visit(e, null));
    }
    public Type visit(DivExp e, Env<Type> env) {
      return visitCompoundArithExp(e, env, ts.visit(e, null));
    }
    public Type visit(MultExp e, Env<Type> env) {
      return visitCompoundArithExp(e, env, ts.visit(e, null));
    }
    public Type visit(SubExp e, Env<Type> env) {
      return visitCompoundArithExp(e, env, ts.visit(e, null));
    }
```

6. (30 points) In this question you will extend the typechecking rules of TypeLang to add rules for checking correctness of conditional and let expression. Implement the typechecking rules for following expressions to realize the correctness criteria that follows.

   (a) (15 points)
   IfExp: Let a IfExp be (if cond then else), where cond, then, else are expressions.

   - if conds type is ErrorT then (if cond then else)s type should be ErrorT
   - if conds 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 5.0 56.0 67.0)
   $ (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 56.0)

   (b) (15 points)
   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.
   - Type of (let ((e1 : T1 V1) (e2: T2 V2)... (en: Tn Vn)) body) is type of 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))

Sol

- IfExp

```
      public Type visit(IfExp e, Env<Type> env) {
        Exp cond = e.conditional();
        Type condType = (Type)cond.accept(this, env);
        if (condType instanceof ErrorT) { return condType; }

        if (!(condType instanceof BoolT)) {
          return new ErrorT("The condition should have boolean type, found " +
          condType.tostring() + " in " + ts.visit(e, null));
        }

        Type thentype = (Type)e.then_exp().accept(this, env);
        if (thentype instanceof ErrorT) { return thentype; }

        Type elsetype = (Type)e.else_exp().accept(this, env);
        if (elsetype instanceof ErrorT) { return elsetype; }

        if (thentype.typeEqual(elsetype)) { return thentype; }

        return new ErrorT("The then and else expressions should have the same "
        + "type, then has type " + thentype.tostring() +
        " else has type " + elsetype.tostring() + " in " +
        ts.visit(e, null));
      }
```

- LetExp

```
      public Type visit(LetExp e, Env<Type> env) {
        List<String> names = e.names();
        List<Exp> value_exps = e.value_exps();
        List<Type> types = e.varTypes();
        List<Type> values = new ArrayList<Type>(value_exps.size());

        int i = 0;
        for (Exp exp : value_exps) {
          Type type = (Type)exp.accept(this, env);
          if (type instanceof ErrorT) { return type; }

          Type argType = types.get(i);
          if (!type.typeEqual(argType)) {
            return new ErrorT("The declared type of the " + i +
            " let variable and the actual type mismatch, expect " +
            argType.tostring() + " found " + type.tostring() +
            " in " + ts.visit(e, null));
          }

          values.add(type);
          i++;
        }

        Env<Type> new_env = env;
        for (int index = 0; index < names.size(); index++)
        new_env = new ExtendEnv<Type>(new_env, names.get(index),
        values.get(index));

        return (Type) e.body().accept(this, new_env);
    }
```