Fall 2019 Final Exam

Instructions:

- Please write your name on the back of the exam when you are done!
- The exam will be held from 9:45 am - 11:45 am. It is close book and close notes, and should be finished independently.
- Please write your answers clearly. If we cannot read your answers, you will lose points. You can always use the back of the paper if you need more space.
- For the coding questions, the algorithms and steps are the most important. We will not reduce your points for small syntax errors. You are also encouraged to add comments to clarify your code.
- There are some references in the Appendix. You are welcomed to check them any time during the exam.
- The exam has a total of 80 pt with 5 extra credits. Please plan your time accordingly.
- Good luck!!!

Your Scores

Q1	20	
Q2	16	
Q3	5	
Q4	6	
Q5	6	
Q6	8	
Q7	6	
Q8	13	
Total	80	
Extra credits Q8	2.5	
Extra credits Haiku	2.5	

1. (20 pt) Multiple choice questions: select *all* the correct answers for the following questions. Mark ✓in front of the correct selections and mark × in front of the incorrect ones.

   - (4 pt) Which of the following statements are/is true about lambda calculus?
     - ✓ there are no recursive functions in lambda calculus
     - × there are no high order functions in lambda calculus
     - × β reduction always returns the same results in dependent of the evaluation order
     - ✓ lambda calculus can simulate all the computations that can be done in Turing machines

   - (4 pt) Which of the following is/are true about types and type systems?
     - ✓ the dynamic type and the static type of an object can be different
     - ✓ the type system of a programming language should be sound
     - ✓ some type systems can reject programs that do not have execution errors
     - ✓ type checking and type inference both need typing rules

   - (4 pt) Which of the following is/are true about memory management of a programming language?
     - × each program must have its own heap
     - ✓ we can design type systems to improve memory safety
     - ✓ RefLang programs can have side effects due to its memory management model
     - ✓ it is easier to design a garbage collector for strongly typed language like Java than weakly type language like C

   - (4 pt) Which of the following is/are true about program semantics?
     - ✓ given the grammar and operational semantics, it is sufficient to implement a programming language
     - ✓ operational semantics define how to compute values for each type of program constructs defined in the grammar
     - × environment is only needed for specifying operational semantics for the typed programming languages
     - ✓ operational semantics also specify heap behaviors of each program construct

   - (4 pt) Which of the following is/are true about logic programming?
     - × you can always find the most general unifies for the two predicates.
     - × imperative, functional and logic programming paradigms are designed to solve different computing problems
     - ✓ the logic programming uses deductive reasoning to solve the problems
     - ✓ the language design for Prolog does not have to consider memory management

2. (16 pt) Given:
   data: $(\lambda(x)\ (\lambda(y)\ (\lambda(z)\ (z\ (x\ y)))))$
   op1: $(\lambda(p)\ (p\ (\lambda(x)\ (\lambda(y)\ x))))$
   op2: $(\lambda(p)\ (p\ (\lambda(x)\ (\lambda(y)\ y))))$
   true: $(\lambda(x)\ (\lambda(y)\ x))$
   false: $(\lambda(x)\ (\lambda(y)\ y))$

   (a) (4 pt) What is the result of $(op1\ ((data\ a)\ b))$?

   Soln:
   $(op1\ ((data\ a)\ b)) =$
   $(op1\ (((\lambda(x)\ (\lambda(y)\ (\lambda(z)\ (z\ (x\ y)))))\ a)\ b)) =$
   $(op1\ ((\lambda(y)\ (\lambda(z)\ (z\ (a\ y))))\ b)) =$
   $(op1\ (\lambda(z)\ (z\ (a\ b)))) =$
   $((\lambda(p)\ (p\ (\lambda(x)\ (\lambda(y)\ x))))\ (\lambda(z)\ (z\ (a\ b)))) =$
   $((\lambda(p)\ (p\ true))\ (\lambda(z)\ (z\ (a\ b)))) =$
   $((\lambda(z)\ (z\ (a\ b)))\ true) =$
   $(true\ (a\ b)) =$
   $((\lambda(x)\ (\lambda(y)\ x))\ (a\ b)) =$
   a

   (b) (4 pt) What is the result of $(op2\ ((data\ a)\ b))$?

   Soln:
   $(op2\ ((data\ a)\ b)) =$
   $(op2\ (((\lambda(x)\ (\lambda(y)\ (\lambda(z)\ (z\ (x\ y)))))\ a)\ b)) =$
   $(op2\ ((\lambda(y)\ (\lambda(z)\ (z\ (a\ y))))\ b)) =$
   $(op2\ (\lambda(z)\ (z\ (a\ b)))) =$
   $((\lambda(p)\ (p\ (\lambda(x)\ (\lambda(y)\ y))))\ (\lambda(z)\ (z\ (a\ b)))) =$
   $((\lambda(p)\ (p\ false))\ (\lambda(z)\ (z\ (a\ b)))) =$
   $((\lambda(z)\ (z\ (a\ b)))\ false) =$
   $(false\ (a\ b)) =$
   $((\lambda(x)\ (\lambda(y)\ y))\ (a\ b)) =$
   b

   (c) (4 pt) What computation do $op1$ and $op2$ perform?

   Soln:
   $op1$ extracts the first in a pair and $op2$ extracts the second in a pair.

   (d) (4 pt) given $x : (data\ (data\ true\ false)(data\ true\ (data\ false\ true)))$, write an expression, using $op1$, $op2$ and x, that evaluates to *false*.

   Soln:
   $(op2\ (op1\ x))$

3. (5 pt) Write a TypeLang function, *AddThenPow(x, y, z)*, where x, y, and z have the type num, and the function returns $(x + y)^z$. Note that Typelang supports +, -, *, /, but no other default math functions.

   Soln:

```
(define power : (num num -> num)
        (lambda (x:num y:num)
                ( if (= y 0) 1 (* x power (x (- y 1))) )
        )
)

(define AddThenPow : (num num num -> num)
        (lambda (x: num y: num z: num)
                (power (+ x y) z)
        )
)
```

4. (6 pt) Write the results for the following RefLang programs.

   (a) (3 pt) (let ((a (ref 4))) (let ((b (deref a))) (let ((c b)) (let ((d (ref c))) (deref d)))))

   Soln:
   4

   (b) (3 pt) (let ((a (ref 4))) (let ((b (deref a))) (let ((c (ref (set! a 6)))) (let ((d (ref 5))) (let ((a d)) (+ b (- (deref a) (deref c))))))))

   Soln:
   3

5. (6 pt) Given the following definition for a node of a linked list as well as the head node.

```
(define pairNode (lambda (fst snd) (lambda (op) (if op fst snd))))
(define node (lambda (x) (pairNode x (ref (list)))))
(define head (node 1))
```

   Write a Reflang program that traverses the linked list and print values of each node:
   (define print (lambda (head) (...))

   Soln:

```
(define getFst (lambda (p) (p #t)))
(define getSnd (lambda (p) (p #f)))
```

```
(define print
       (lambda (head)
              ( if (null? (deref (getSnd head)))
                     (getFst head)
                     (cons (getFst head) (print (deref (getSnd head))))
              )
       )
 )
```

6. (8 pt) Given the following Prolog facts:

```
male(james1).
male(james2).
male(charles1).
male(charles2).

female(catherine).
female(elizabeth).
female(claudia).
female(fay).

%% parent(child, parent).
parent(charles1, james1).
parent(elizabeth, james1).
parent(elizabeth, claudia).
parent(charles1, claudia).
parent(charles2, charles1).
parent(catherine, charles1).
parent(james2, charles1).
parent(charles2, fay).
parent(catherine, fay).
parent(james2, fay).

%% married(A, B): A is married to B
married(james1, claudia).
married(claudia, james1).
married(charles1, fay).
married(fay, charles1).
```

Write the following rules:

(a) (4 pt) Write **aunt(Child, Aunt) :-** . An aunt is the sister of the child's father or mother, or she is the wife of the child's uncle.

(b) (4 pt) Write **descendant(Person, Descendant) :-** . You are a descendant of X if you are a child of X, or your parent is a descendant of X.

Soln:

```
father(Child, Dad) :- male(Dad), parent(Child, Dad).
mother(Child, Mom) :- female(Mom), parent(Child, Mom).
brother(Sibling, Bro) :- male(Bro), father(Sibling, Father),
father(Bro, Father), Bro \= Sibling, mother(Sibling, Mother), mother(Bro, Mother).
sister(Sibling, Sis) :- female(Sis), father(Sibling, Father), father(Sis, Father),
    ↪ Sis \= Sibling, mother(Sibling, Mother), mother(Sis, Mother).

aunt(Child, Aunt) :- female(Aunt), parent(Child, Parent), sister(Parent, Aunt).
aunt(Child, Aunt) :- female(Aunt), parent(Child, Person), brother(Person, Brother),
    ↪ married(Aunt,Brother).

descendent(Person, Descendent) :- parent(Descendent, Person).
descendent(Person, Descendent) :- parent(Descendent, Someone), descendent(Person,
    ↪ Someone).
```

7. (6 pt) Given two lists of integers L1 and L2, write a Prolog program P(L1,L2,L) where L contains common integers of the two lists. Note that you can use the Prolog library function *not(X)* to negate X, but you cannot use other Prolog library functions or functions we have explained in the previous homework and lectures. See the following example output.

   ?-p([1,2], [1], L).
   L = [1].
   ?-p([1,2], [3,4,5], L).
   L = [ ].

Soln:

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).

p([],_,[]).
p([X|L],K,M):- not(member(X,K)), p(L,K,M).
p([X|L],K,[X|M]):- member(X,K), p(L,K,M).
```

8. (12+2 pt) In Typelang some of the top-level elements also changed to provide the functionality of type checking, e.g., you have observed the syntactic changes made in the grammar. Lets review an example to get the idea.

   The syntax for let expression in the grammar file was as follows:
   letexp ::= (**let** ((*identifier exp*)$^+$) exp)

But to enable type checking the syntax changed as following:
letexp ::= (**let** ((*identifier* : *T exp*)$^+$) exp)

Previously you have implemented an encrypted let (lete for let encrypted), which is similar to let, but takes an additional parameter key and a dec expression that is similar to VarExp. All values are stored by encrypting them with key, and read by decrypting them with key. The example log is as follows:

```
$ (lete 42 ((x 1)) x)
43
$ (lete 42 ((x 1)) (dec 42 x))
1
$ (lete 10 ((y 12)) y)
22
$ (lete 10 ((y 12)) (dec 10 y))
12
```

(a) (3 pt) Write a Antler grammar rule to support typed lete expression. A reference Antler grammar rule can be found in Appendix.

Soln:

```
leteexp returns [LeteExp ast]
locals [ArrayList<String> names, ArrayList<Type> types, ArrayList<Exp>
   ↪ value_exps]
@init { $names = new ArrayList<String>(); $types=new ArrayList<Type>();
   ↪ $value_exps = new ArrayList<Exp>(); } :
'(' 'lete' key=exp
'(' ( '(' id=Identifier ':' t=type e=exp ')' { $names.add($id.text);$types.add(
   ↪ $t.ast); $value_exps.add($e.ast); } )+ ')'
body=exp
')' { $ast = new LetExp($names, $types, $value_exps, $body.ast, $key.ast); }
;
```

(b) (3 pt + 2pt) Describe the type checking rule for lete expression. (2 extra credit if you state it in the formal type inference rule)

Soln:

TYPE OF LETEEXP

$$\frac{\begin{array}{c} \texttt{tenv} \vdash \ e_i : t_i \ , \ \forall i \ \in \texttt{0...n} \\ \texttt{tenv} \vdash \ k : num \\ \texttt{tenv}_0 = (\texttt{ExtendEnv}\ var_0 ,\ t_0 ,\ tenv)\ \dots \\ \texttt{tenv}_n = (\texttt{ExtendEnv}\ var_n ,\ t_n ,\ tenv_{n-1}) \\ tenv_n \vdash e_{body} : t \end{array}}{\texttt{tenv} \vdash (\texttt{LeteExp k},\ var_0 ,\ \dots ,\ var_n ,\ t_0 ,\ \dots ,\ t_n ,\ e_0 ,\ \dots ,\ e_n ,\ e_{body}) \ : \ \texttt{t}}$$

(c) (6 pt) Implement the type checker for lete expression. For error types use the type `ErrorT`, for numerics use the type `NumT`. All these types extend the class `Type`. A reference typechecker can be found in Appendix.

Soln:

```
public Type visit(LeteExp 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());
        Exp key = e.key();

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

        if (!(keyType instanceof NumT)) {
                return new ErrorT("The key should have num type, found " +
                    ↪ keyType.tostring() + " in " + ts.visit(e, null));
        }

        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);
}
```

9. (Extra credit questions per your request: 2.5 pt) Write a *haiku* about this class. A haiku is a poem in three lines, the first and third lines having five syllables, the second having seven. They're supposed to be profound. Let's see some examples below:

   final is over!
   I am super happy yeah.
   Let's shout in the park!

   Okay, not so profound. Another:

   A +3 poem!
   I will wield it with vigor!
   Sauron is a chump.

   Also not profound. And a third:

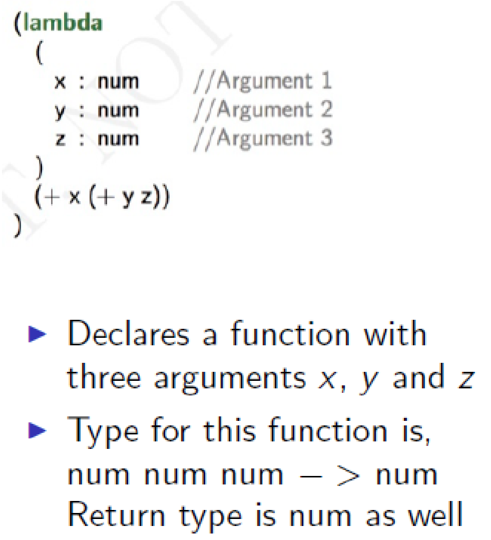
   Sunshine in the morn.
   The snow flakes are dancing wild.
   Olaf sings da da doo.

   You are also welcomed to write it in foreign languages. In this case, please explain the language you used and the meaning of your sentences in English.

   Soln:
   Use the given examples for your reference.

Appendix: Typelang Grammar and Examples

$$
\begin{array}{lll}
\textit{program} & ::= & \textit{definedecl}^{*}\ \textit{exp}? \\
\textit{definedecl} & ::= & (\textbf{define}\ \textit{identifier} : T\ \textit{exp}) \\
\textit{lambdaexp} & ::= & (\textbf{lambda}\ (\{\textit{identifier} : T\}^{*})\ \textit{exp})
\end{array}
$$



```
(lambda
  (
    x : num      //Argument 1
    y : num      //Argument 2
    z : num      //Argument 3
  )
  (+ x (+ y z))
)
```

- Declares a function with three arguments x, y and z
- Type for this function is, num num num $- >$ num
  Return type is num as well

Appendix: Type Checking Rules

(LetExp)

$$
\frac{\begin{array}{c} tenv \vdash e_i : t_i, \forall i \in 0..n \\ tenv_n = (ExtendEnv\ var_n\ t_n\ tenv_{n-1})\ \ldots \\ tenv_0 = (ExtendEnv\ var_0\ t_0\ tenv) \\ tenv_n \vdash e_{body} : t \end{array}}{tenv \vdash (LetExp\ var_0\ \ldots\ var_n\ \ t_0\ \ldots\ t_n\ \ e_0\ \ldots\ e_n\ \ e_{body}) : t}
$$

Appendix: Interpreter Code Examples

1. Editing .g file with type

```
letexp returns [LetExp ast]
locals [ArrayList<String> names, ArrayList<Type> types, ArrayList<Exp> value_exps]
@init { $names = new ArrayList<String>(); $types=new ArrayList<Type>(); $value_exps
    ↪ = new ArrayList<Exp>(); } :
'(' Let
'(' ( '(' id=Identifier ':' t=type e=exp ')' { $names.add($id.text);$types.add($t.
    ↪ ast); $value_exps.add($e.ast); } )+ ')'
body=exp
')' { $ast = new LetExp($names, $types, $value_exps, $body.ast); }
;
```

2. Type implementation

```
public interface Type {
public String tostring();
public boolean typeEqual(Type other);

static class ErrorT implements Type {
String _message;
public ErrorT(String message) { _message = message; }
public String tostring() {
return "Type error: " + _message;
}
public boolean typeEqual(Type other) { return other == this; }
}
...
static class NumT implements Type {
private static final NumT _instance = new NumT();
public static NumT getInstance() { return _instance; }
public String tostring() { return "number"; }
public boolean typeEqual(Type other) { return other.getClass() == this.getClass(); }
}
...
}
```

3. Type Checker

```
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);
}
```