Final Exam

Instructions:

- The exam will be held from 9:45 am - 11:45 am. The exam has a total of 8 questions (total points: 100). It is close book and close notes and should be finished independently.
- Please write your answers clearly to avoid losing points. For the coding questions, the algorithms and steps are the most important. We will not reduce your points because of the small syntax errors. You are also encouraged to add comments to clarify your code.
- Appendix includes some references for you to use.
- If you run out of space, you always can use the back of the paper to write your answers.
- Good luck!!!

Your Name: ____________

1. (20 pt) Select *all* the correct answers for the following questions.

   (a) (4 pt) Which of the following understandings about semantics is/are correct:

      i. we can prove the correctness of the program using axiomatic semantics
      ii. semantic rules for a programming language should be sound and complete
      iii. any one of the three types of semantics can be used to implement a programming language
      iv. operational semantics define how to compute values for each type of program constructs defined in the grammar

      Sol. i, ii, iv

   (b) (4 pt) Which of the following understandings about type is/are correct:

      i. the dynamic type and the static type of an object can be different
      ii. types define the range of values and operations on the values
      iii. some type systems can reject programs that do not have execution errors
      iv. type checking and type inference both need typing rules

      Sol. i, ii, iii, iv

   (c) (4 pt) Which of the following understandings about logic programming is/are correct:

      i. you can always find the most general unifier for two predicates
      ii. back tracking is used in the logic programming language interpreter to find multiple solutions for a query

iii. Prolog is a domain-specific language

iv. logic programming language is declarative

Sol. ii, iv

(d) (4 pt) Which of the following understandings about functional programming is/are true:

i. lambda calculus is Turing complete

ii. functional programming languages and logic programming languages can solve the same problems

iii. in lambda calculus, there is no side effect in any lambda expression.

iv. lambda encoding uses functions to simulate computation

Sol. i, ii, iii, iv

(e) (4 pt) Which of the following statements about memory management design is/are true?

i. heap and references are the two important design decisions when we design memory management

ii. each program must have its own heap

iii. we can design type systems to improve memory safety

iv. Rust is a language that guarantees memory safety

Sol. i, iii, iv

2. (8 pt) Given:
g: $(\lambda(a)(\lambda(b)(\lambda(c)((a\,b)\ ((a\,b)\ c)))))$
$zero$: $(\lambda(f)(\lambda(x)x))$
one: $(\lambda(f)(\lambda(x)(f\,x)))$.
two: $(\lambda(f)(\lambda(x)(f\ (f\,x))))$.
$three$: $(\lambda(f)(\lambda(x)(f\ (f\ (f\,x)))))$.
$four$: $(\lambda(f)(\lambda(x)(f\ (f\ (f\ (f\,x))))))$.

(a) (3 pt) What is the result of (g one)?

(b) (3 pt) What is the result of (g two)?

(c) (2 pt) What computation does g performs?

Sol.

(a) $(g\ one) =$
$((\lambda(a)(\lambda(b)(\lambda(c)((a\,b)\ ((a\,b)\ c)))))\ one) =$
$(\lambda(b)(\lambda(c)((one\,b)\ ((one\,b)\ c)))) =$
$(\lambda(b)(\lambda(c)((one\,b)\ (((\lambda(f)(\lambda(x)(f\,x)))\ b)\ c)))) =$
$(\lambda(b)(\lambda(c)((one\,b)\ (b\ c)))) =$

$(\lambda(b)(\lambda(c)(((\lambda(f)(\lambda(x)(f\ x)))\ b)\ (b\ c)))) =$
$(\lambda(b)(\lambda(c)(((\lambda(x)(b\ x)))\ (b\ c)))) =$
$(\lambda(b)(\lambda(c)(b\ (b\ c)))) =$
two

(b) $(g\ two) =$
$((\lambda(a)(\lambda(b)(\lambda(c)((a\ b)\ ((a\ b)\ c)))))\ two) =$
$(\lambda(b)(\lambda(c)((two\ b)\ ((two\ b)\ c)))) =$
$(\lambda(b)(\lambda(c)((two\ b)\ (((\lambda(f)(\lambda(x)(f\ (f\ x))))\ b)\ c)))) =$
$(\lambda(b)(\lambda(c)((two\ b)\ (b\ (b\ c))))) =$
$(\lambda(b)(\lambda(c)(((\lambda(f)(\lambda(x)(f\ (f\ x))))\ b)\ (b\ (b\ c))))) =$
$(\lambda(b)(\lambda(c)(((\lambda(x)(b\ (b\ x))))\ (b\ (b\ c))))) =$
$(\lambda(b)(\lambda(c)(b\ (b\ (b\ (b\ c)))))) =$
four

(c) g performs a mutiplication of a given number by 2. e.g. (g three) = 2 * 3 = 6

3. (8 pt) Consider the following lambda expressions.
C: $(\lambda(g)(\lambda(f)\ (\lambda(x)(g\ (f\ x)))))$
B: $(\lambda(f)(\lambda(x)\ (f\ (f\ x))))$

Prove or disprove the following:

(a) (4 pt) $(B\ B) = ((C\ B)\ B)$
(b) (4 pt) $((C\ h_1)((C\ h_2)\ h_3)) = ((C\ ((C\ h_1)\ h_2))\ h_3)$

Sol.

(a) We are going to solve both sides:
$(B\ B) =$
$((\lambda(f)(\lambda(x)\ (f\ (f\ x))))\ B) =$
$(\lambda(x)\ (B\ (B\ x)))$

$((C\ B)B) =$
$(((\lambda(g)(\lambda(f)\ (\lambda(x)(g\ (f\ x)))))\ B)\ B) =$
$((\lambda(f)\ (\lambda(x)(B\ (f\ x))))\ B) =$
$(\lambda(x)(B\ (B\ x)))$

(b) We are going to solve both sides:
$((C\ h_1)((C\ h_2)\ h_3)) =$
$((C\ h_1)(((\lambda(g)(\lambda(f)\ (\lambda(x)(g\ (f\ x)))))\ h_2)\ h_3)) =$
$((C\ h_1)((\lambda(f)\ (\lambda(x)(h_2\ (f\ x))))\ h_3)) =$
$((C\ h_1)(\lambda(x)(h_2\ (h_3\ x)))) =$
$(((\lambda(g)(\lambda(f)\ (\lambda(x)(g\ (f\ x)))))\ h_1)(\lambda(x)(h_2\ (h_3\ x)))) =$
$((\lambda(f)\ (\lambda(x)(h_1\ (f\ x))))\ (\lambda(x)(h_2\ (h_3\ x)))) =$
$(\lambda(x)(h_1\ ((\lambda(x)(h_2\ (h_3\ x))\ x)))) =$

$(\lambda(x)(h_1\ (h_2\ (h_3\ x))))$

$$
\begin{aligned}
&((C\ ((C\ h_1)\ h_2))\ h_3) = \\
&((C\ (((\lambda(g)(\lambda(f)\ (\lambda(x)(g\ (f\ x)))))\ h_1)\ h_2))\ h_3) = \\
&((C\ ((\lambda(f)\ (\lambda(x)(h_1\ (f\ x))))\ h_2))\ h_3) = \\
&((C\ (\lambda(x)(h_1\ (h_2\ x))))\ h_3) = \\
&(((\lambda(g)(\lambda(f)\ (\lambda(x)(g\ (f\ x)))))\ (\lambda(x)(h_1\ (h_2\ x))))\ h_3) = \\
&(((\lambda(f)\ (\lambda(x)((\lambda(x)(h_1\ (h_2\ x)))\ (f\ x))))\ h_3) = \\
&(\lambda(x)((\lambda(x)(h_1\ (h_2\ x)))\ (h_3\ x))) = \\
&(\lambda(x)(h_1\ (h_2\ (h_3\ x))))
\end{aligned}
$$

4. (5 pt) Given two sorted lists of integers L_1 and L_2, write a Prolog program that merges the lists to build a new sorted list.

   Sol.

```
merge([],L,L).
merge(L,[],L).
merge([Head1|Tail1], [Head2|Tail2], L) :-
        Head1 < Head2 -> L = [Head1|R], merge(Tail1,[Head2|Tail2],R) ;
        Head1 > Head2 -> L = [Head2|R], merge([Head1|Tail1],Tail2,R) ;
        L = [Head1,Head2|R], merge(Tail1,Tail2,R).
```

5. (10 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) (5 pt) Write **aunt(Kid, Auntie) :-??**. Aunt is defined as the sister of one's father or mother, or the wife of one's uncle.

(b) (5 pt) Write **descendant(Person, Descendant) :- ??**. You are a descendant if you are the child of a person, or your parent is the child of the person and so forth.

Sol.

```
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(Kid, Auntie) :- female(Auntie), parent(Kid, Parent),
        sister(Parent, Auntie).
aunt(Kid, Auntie) :- female(Auntie), parent(Kid, Person),
        brother(Person, Brother), married(Auntie, Brother).

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

6. (9 pt) Write the results of the following RefLang programs

- Assume semantics from homework 7 and that trying to add anything with the type REF results in ERROR
- Assume that arithmetic and let expressions are evaluated from left to right
- Assume that trying to access a freed location with free or deref results in ERROR

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

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

(c) (3 pt) (let((a (ref 5)) (b (set! a 2)) (c (free a))) (+ (deref a) b))

Sol.

(a) 17

(b) 15

(c) ERROR

7. (15 pt) In this question you will implement a *binary tree* using Reflang. In a binary tree, one node is the root node. Every node other than the root must have a parent node, and has optional left and right child node. If there's no node child associated with this node, it is a leaf node. Each node of the tree can hold a value.

   Specifically in our implementation, each node will have four fields. First field is a number representing the value. The second field is a reference to the parent node, and the last two fields hold the reference to the left and right child. In the case of missing parent or child, the reference is referred to an empty list `(ref (list))`.

   (Hint: in homework7, we implemented a linked list data structure. See below.

```
(define pairNode (lambda (fst snd) (lambda (op) (if op fst snd))))
(define node (lambda (x) (pairNode x (ref (list)))))
(define getFst (lambda (p) (p #t)))
(define getSnd (lambda (p) (p #f)))
```

   The binary tree implementation uses a similar approach except that a node in the linked list only has one successor but a node in a binary tree can have both left and right children.)

   (a) (10 pt) Constructing the data structure:
      i. (4 pt) write a lambda method `node` that accept one numeric argument as the value, and construct the node.
      ii. (2 pt) write a lambda method `value` that accept the node and return the numeric value.
      iii. (4 pt) write a lambda method `left` and `right` that takes a node and return its left or right child node

   (b) (5 pt) write a lambda method `add` that takes three parameters:
      i. first parameter `p` is the parent node that the new node is going to append to.
      ii. second parameter `which` is a boolean variable where `#t` means add to left, and `#f` means add to right.
      iii. a last parameter `c` is the child node that is going to be added.
      iv. The `add` function adds `c` as a left or right child node to the `p` node.

      Following transcripts will help you understand the functions more:

```
(define root (node 1))
(add root #t (node 2))
(add root #f (node 3))
(add (deref (left root)) #f (node 4))
(add (deref (right root)) #t (node 5))
(add (deref (right root)) #f (node 6))
```

Sol.

(a)

```
(define threenode
  (lambda (one two three)
    (lambda (num)
      (if (= num 1) one
        (if (= num 2) two
          (if (= num 3) three #f))))))

;; value, left, right
(define node
  (lambda (x)
    (threenode
      x
      (ref (list))
      (ref (list)))))

(define value (lambda (n) (n 1)))
(define left (lambda (n) (n 2)))
(define right (lambda (n) (n 3)))
```

(b)

```
(define add
  (lambda (p which c)
    (if which
     (set! (left p) c)
    (set! (right p) c))))
```

8. (25 pt) Extending the Typelang to support two string operations ”stradd” and ”strsub”: ”stradd” concatenates two strings, ”strsub” removes the second string from the first string if the second string is a substring of the first string; otherwise, ”strsub” returns the first string. The following transcripts can help you understand the semantics of the two operations.

```
$(define s: string "hello")
$(define t: string "world")
$(stradd s t)
$> "hello world"

$(let ((s: string "hello") (t: string "world")) (stradd s t))
$> "hello world"

$(define a: string "hello")
$(define b: string "he")
(strsub a b)
$> "llo"

$(define c: string "hello")
$(define d: string "world")
(strsub c d)
$> "hello"

$(define c: string "hello")
$(define d: num 15)
(strsub c d)
$> TypeError

$(define c: string "hello")
$(define d: num 15)
(stradd c d)
$> TypeError
```

(a) (5 pt) Modify the grammar to support the two operations. As a reference, we show a grammar rule for addexp in ArithLang below.

```
addexp returns [AddExp ast]
        locals [ArrayList<Exp> list]
        @init { $list = new ArrayList<Exp>(); } :
        '(' '+'
           e=exp { $list.add($e.ast); }
          ( e=exp { $list.add($e.ast); } )+
        ')' { $ast = new AddExp($list); }
        ;
```

Sol

```
exp returns [Exp ast]:
        ...
        | stradd=straddexp {$ast = $stradd.ast; }
        | strsub=strsubexp {$ast = $strsub.ast; }

 straddexp returns [StraddExp ast] :
        '(' 'stradd'
        e1=exp e2=exp
        ')' { $ast = new StraddExp($e1.ast, $e2.ast);}
    ;

 strsubexp returns [StrsubExp ast] :
        '(' 'strsub'
        e1=exp e2=exp
        ')' { $ast = new StrsubExp($e1.ast, $e2.ast);}
    ;
```

(b) (4 pt) Extend the AST implementation. See a reference example below.

```
public static class CarExp extends Exp {
    private Exp _arg;
    public CarExp(Exp arg){
        _arg = arg;
    }
    public Exp arg() { return _arg; }
    public Object accept(Visitor visitor, Object env) {
        return visitor.visit(this, env);
    }
}
```

Sol

```
public static class StraddExp extends Exp {
    Exp _e1, _e2;
    public StraddExp(Exp e1, Exp e2) {
        _e1 = e1;
        _e2 = e2;
    }
    public Object accept(Visitor visitor, Object env) {
        return visitor.visit(this, env);
    }
    public Exp e1() {return _e1;}
    public Exp e2() {return _e2;}
}
public static class StrsubExp extends Exp {
    Exp _e1, _e2;
    public StrsubExp(Exp e1, Exp e2) {
        _e1 = e1;
        _e2 = e2;
    }
    public Object accept(Visitor visitor, Object env) {
        return visitor.visit(this, env);
    }
    public Exp e1() {return _e1;}
    public Exp e2() {return _e2;}
}
```

(c) (8 pt) Extend the evaluator to support the semantics of ”stradd” and ”strsub” for strings. For string values use the class `StringVal`. See a reference example below.

```
public Value visit(AddExp e, Env<Value> env) {
    List<Exp> operands = e.all();
    int result = 0;
    for(Exp exp: operands) {
        NumVal intermediate = (NumVal) exp.accept(this, env); // Dynamic type-
            checking
        result += intermediate.v(); //Semantics of AddExp in terms of the target
            language.
    }
    return new NumVal(result);
}
```

Sol

```
public Value visit(StraddExp e, Env<Value> env) {
    Exp e1 = e.e1();
    Exp e2 = e.e2();
    StringVal v1 = (StringVal) e1.accept(this, env);
    String s1 = v1.v();
    StringVal v2 = (StringVal) e2.accept(this, env);
    String s2 = v2.v();
    s1 = s1.substring(1, s1.length()-1);
    s2 = s2.substring(1, s2.length()-1);
    String res = "\"" + s1 + " " + s2 + "\"";
    return new StringVal(res);
}
public Value visit(StrsubExp e, Env<Value> env) {
    Exp e1 = e.e1();
    Exp e2 = e.e2();
    StringVal v1 = (StringVal) e1.accept(this, env);
    String s1 = v1.v();
    StringVal v2 = (StringVal) e2.accept(this, env);
    String s2 = v2.v();
    s1 = s1.substring(1, s1.length()-1);
    s2 = s2.substring(1, s2.length()-1);
    s1 = s1.replace(s2, "");
    return new StringVal("\""+s1+"\"");
}
```

(d) (8 pt) Implement the type checker for the two operations. For error types use the type `ErrorT`, for pairs use the type `PairT` and for strings use the type `StringT`. All these types extend the class `Type`. See a reference example of type checker below.

```
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));
}
```

Sol

```
public Type visit(StraddExp e, Env<Type> env) {
    Exp e1 = e.e1();
    Exp e2 = e.e2();
    Type t1 = (Type) e1.accept(this, env);
    Type t2 = (Type) e2.accept(this, env);
    if (t1 instanceof ErrorT) {return t1;}
    if (t2 instanceof ErrorT) {return t2;}
    if (!(t1 instanceof StringT) || !(t2 instanceof StringT)) {
        return new ErrorT("Expected String found " + t1.tostring() + " in " + ts.
            visit(e, null));
    }
    return StringT.getInstance();
}
public Type visit(StrsubExp e, Env<Type> env) {
    Exp e1 = e.e1();
    Exp e2 = e.e2();
    Type t1 = (Type) e1.accept(this, env);
    Type t2 = (Type) e2.accept(this, env);
    if (t1 instanceof ErrorT) {return t1;}
    if (t2 instanceof ErrorT) {return t2;}
    if (!(t1 instanceof StringT) || !(t2 instanceof StringT)) {
        return new ErrorT("Expected String found " + t1.tostring() + " in " + ts.
            visit(e, null));
    }
    return StringT.getInstance();
}
```