Fall 2019 Midterm

Instructions:

- The exam will be held from 11:00 am - 12:20 pm. 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 because of the 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 7 questions, a total of 70 pt. Please plan your time accordingly.
- Good luck!!!

Your Scores

Q1	16	
Q2	7	
Q3	10	
Q4	5	
Q5	5	
Q6	12	
Q7	15	
Total	70	

Your Name:

1. (16 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 grammars?
     - × in context free grammars, non-terminals are tokens
     - ✓ a grammar is a way to specify patterns in string
     - ✓ we can use context free grammars to automatically generate a parser for the programming language
     - ✓ we can define different grammars for the same programming language

   - (4 pt) Which of the following is/are true about program paradigms?
     - ✓ we can use imperative, functional and logic programming paradigms to solve a same problem
     - ✓ pure functional programming languages do not need to have loops
     - ✓ side effects make programs hard to verify
     - × domain specific languages are imperative programming languages

   - (4 pt) Which of the following is/are true about ambiguity?
     - × we can always modify a grammar to remove ambiguity
     - × if a grammar is ambiguous, then all the strings generated by the grammar have two or more parse trees
     - × for a non-ambiguous grammar, we only can generate one derivation for a string
     - ✓ we can define operator precedence and associativity through grammars

   - (4 pt) Which of the following is/are true about programming languages?
     - ✓ a programming language should support abstraction, composition and computation
     - ✓ we can implement a programming language given a grammar and a set of operational semantic rules
     - ✓ practical programming languages use mostly context free grammar rules to specify the syntax
     - ✓ we can implement the programming languages in either compilers or interpreters

2. (7 pt) Grammar understanding and analysis: Given the following grammar:

$$S \to (S)$$
$$S \to SS$$
$$S \to \epsilon$$

where the terminals are (,), ϵ and S is the start non-terminal.

(a) (2 pt) Show a leftmost derivation for the string ((())).

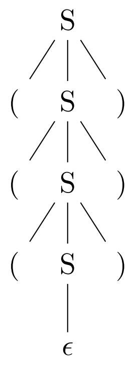
Sol.

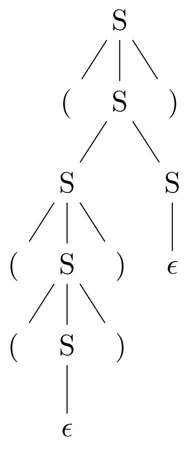
```
S
=> '(' S ')'
=> '(' '(' S ')' ')'
=> '(' '(' '(' S ')' ')' ')'
=> '(' '(' '(' ')' ')' ')'
```

(b) (5 pt) Is this grammar ambiguous? Please justify your answers. If the answer is yes, please modify the grammar to remove the ambiguity.

Sol.

Yes. Because, we can show that the grammar can have two different distinct parse trees for the same string(e.g., ((())))). Two distinct parse tree for the string ((())) is shown below:





Modified non-ambiguous grammar: The ambiguity is removed through enforcing right-associativity.

$$
\begin{aligned}
S &\to AS \\
S &\to \epsilon \\
A &\to (S)
\end{aligned}
$$

3. (10 pt) Grammar construction, strings and parse tree.

   (a) (5 pt) Give a context-free grammar that generates palindromes (strings that read the same forward and backward, including empty strings) consisting of letters a, b and c. Examples of strings: ababa, abccba, acbbca, cbc

   Sol.

$$
\begin{aligned}
S &\to aSa \\
S &\to bSb \\
S &\to cSc \\
S &\to a \\
S &\to b \\
S &\to c \\
S &\to \epsilon
\end{aligned}
$$

   (b) (3 pt) Select your own vocabulary instead of using a, b and c, and generate a *palindrome poem.* The *palindrome poem* should consist of at least 3 sentences, and each sentence should be a palindrome that can be automatically generated from your grammar.

   Sol.

   Let, a = ‘Alice’, B = ‘Bob’, C = ‘and’

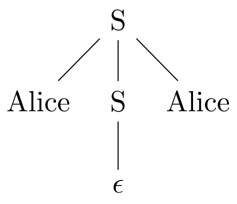
   Generated palindrome poem is below:

   Alice Alice
   Bob Bob
   Bob and Alice Alice and Bob

(c) (2 pt) Select a sentence in your poem and construct a parse tree for it.

Sol.

The parse tree for the sentence "Alice Alice" is below:



4. (5 pt) Bound/free variables, scope:

(a) (3 pt) Mark the free (F) and bound (B) variables in the following program:

```
(define f1

        (lambda (x)

                (if (= 0 x) 0

                        (let

                                ((f2

                                        (lambda (x z) (+ x q) )

                                ))

                        (f2 x 0)

                        )
                )
        )
)
```

Sol.

```
(define f1

        (lambda (x)

                (if (= 0 x) 0
                                  B
                                 (let

                                        ((f2

                                              (lambda (x z) (+ x q) )
                                                           B F
                                        ))

                                 (f2 x 0)
                                   B B
                                 )
                           )
                     )
               )
```

(b) (2 pt) Does this program contain a hole? If so, please list it/them.

Sol.

The definition of x via "(lambda (x)" is not able to be seen in the expression (+ x q). Thus, there is a hole.

5. (5 pt) Given the following semantic rule for a call expression:

Value of CallExp

$$\frac{\begin{array}{c}\texttt{value exp env}_0\texttt{ = (FunVal var}_i\texttt{,for i = 0...k exp}_b\texttt{ env}_0\texttt{)}\\ \texttt{value exp}_i\texttt{ env}_0\texttt{ = v}_i, \texttt{env}_{i+1}\texttt{ = (ExtendEnv var}_i\texttt{ v}_i\texttt{ env}_i\texttt{)}, \text{for i = 0...k}\\ \texttt{value exp}_b\texttt{ env}_{k+1}\texttt{ = v}\end{array}}{\texttt{value (CallExp exp exp}_i\texttt{,for i = 0...k) env}_0\texttt{ = v}}$$

Explain what are the steps happening in the interpreter when evaluating the third program (foo (+ x 10) x) in the context of the following:

$ (define x 10)
$ (define foo (lambda (x y) (+ x y)))
$ (foo (+ x 10) x)

Sol.

(foo (+ x 10) x)
As the expression (foo (+ x 10) x) is called, first, it checks that evaluating (foo (+ x 10) x) results in FunVal. Next, actuals are evaluated in the original environment, i.e, the first expression (+ x 10) is evaluated, since x has a binding 10, the addition expression is evaluated to 20 and second expression is x and evaluated to 10.

Finally, to evaluate the body of function foo, an environment for running the function body fenv is created by first appending the bindings from function value and the initial environment, and then successively adding mapping from formal parameters to actual parameters, e.g., in case of the function foo, the formal parameter x gets the binding 20 and y gets the binding 10 and then function body of foo is evaluated, which, is an addition expression and the final result is the value of this addition between x and y i.e., 30.

6. (12 pt) FuncLang Programming:

   (a) (4 pt) Greatest common divisor (GCD) of two numbers a and b is defined as follows:
   if a > b then (gcd a b) is gcd of a - b and b
   else if a < b then (gcd a b) is gcd of a and b - a
   otherwise, it is a.

   Write a FuncLang program *gcd* that computes the greatest common divisor according the definition above.

   Example scripts:

```
$ (gcd 4 2)
2
$ (gcd 12 15)
3
```

   Sol.

```
(define gcd
        (lambda (a b)
                (if (= a b) a
                        (if (> a b) (gcd (- a b) b)
                                (gcd a (- b a))
                        )
                )
        )
)
```

   (b) (4 pt) Write a function *map*: *map* takes two inputs: a list of numbers *l*, and an operator *op* that takes one operand as input; *map* returns a list where each element is the result of *op* performed on the corresponding member in the list.

   Example scripts:

```
$ (define op (lambda (x) (* x x)))
$ (op 2)
$ 4
$ (map (list 0 2 3) op)
$ (0 4 9)

$ (define addSelf (lambda (x) (+ x x)))
$ (addSelf 3)
$ 6
$ (map (list 0 2 3) addSelf)
$ (0 4 6)
```

Sol.

```
(define map
        (lambda (lst op)
                (if (null? lst) (list)
                        (cons (op (car lst )) (map (cdr lst) op))
                )
        )
)
```

(c) (4 pt) Write a FuncLang program to convert a positive integer number to be a list.

Example scripts:

```
$ (ConvertToList 123)
$ (1 2 3)
$ (ConvertToList -1)
$ ()
$ (ConvertToList 3491)
$ (3 4 9 1)
$ (ConvertToList -3491)
$ ()
```

```
(define quotient
        (lambda (a b)
                (if (< a b) 0
                        (+ 1 (quotient (- a b) b))
                )
        )
)
```

```
(define mod
        (lambda (a n)
                (if (= a 0) 0
                        (if (= n 0) 0
                                (- a (* n (quotient a n)))
                        )
                )
        )
)


(define append
        (lambda (lst1 lst2)
                (if (null? lst1) lst2
                        (if (null? lst2) lst1
                                (cons (car lst1) (append (cdr lst1) lst2))
                        )
                )
        )
)

(define ConvertToList
        (lambda (d)
                (if (<= d 0) (list)
                        (append (ConvertToList (quotient d 10)) (list (mod d 10)))
                )
        )
)
```

7. (15 pt) Extend the language to support a *ConverterH* operation on the positive numbers. The operation returns an Hexadecimal value of a given integer. See example scripts below:

```
$ (ConverterH 124)
$ 7C
$ (ConverterH 93)
$ 5D
$ (ConverterH 655)
$ 28F
```

(4 pt) As the first step, please modify the grammar below

```
exp returns [Exp ast]:
| hex=hexexp { $ast = $hex.ast; }

hexexp returns [HexExpr ast]:
```

```
// complete grammar here
       '(' 'ConverterH'
       e=exp
       ')' { $ast = new HexExpr($e.ast); }
       ;
```

(3 pt) Please update the AST implementation

```
public static class HexExpr extends Exp {
// complete AST for HexExpr here
       Exp _value_exp;

       public HexExpr(Exp value_exp) {
              _value_exp = value_exp;
       }

       public Exp value_exp() { return _value_exp; }

       public Object accept(Visitor visitor, Env env) { return visitor.visit(this,
          ↪ env); }
}

public interface Visitor <T> {
//Add a signature for HexExpr AST node.
       public T visit(AST.HexExpr e, Env env); // Additional expressions for midterm
}
```

(3 pt) Then you will extend the value class to support the Hex values.

```
static class HexVal implements Value {
// complete implementation for new type of value HexVal here
       private String _val;
       public HexVal(String v) { _val = v; }
       public String v() { return _val; }
       public String tostring() {
              return "" + _val;
       }
}
```

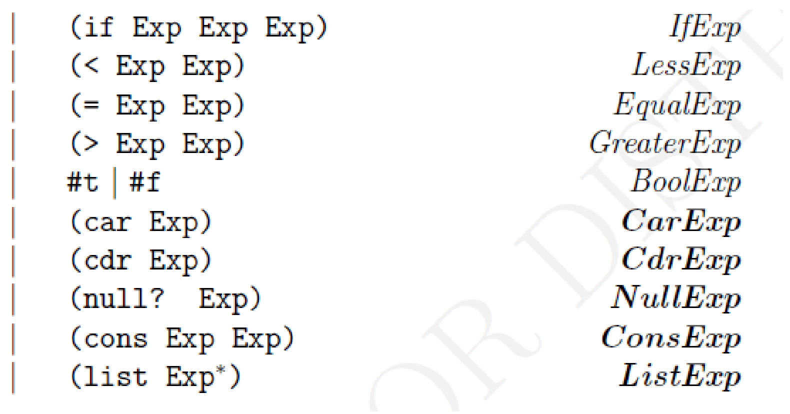
(5 pt) Finally, you can complete the following Evaluator:

```
@Override
public Value visit(HexExpr e, Env env) {
// write the evaluation here
```

```
        // Digits in hexadecimal number system
        char hex[]={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
        int remainder = 1; // store remainder;
        String result = "";

        NumVal value = (NumVal) e.value_exp().accept(this, env); // Dynamic type-
            ↪ checking
        if (value.v() < 0) // Check for positive number
                return new Value.DynamicError(value.v() + " is not a positive number")
                    ↪ ;

        int intermediate = (int) value.v();

        while (intermediate > 0) {
                remainder = intermediate % 16;
                result = hex[remainder] + result;
                intermediate /= 16;
        }
        return new HexVal(result);
}
```

Appendix: Grammar

Program	::=	DefineDecl* Exp?	*Program*
DefineDecl	::=	(define Identifier Exp)	*Define*
Exp	::=		*Expressions*
		Number	*NumExp*
	|	(+ Exp Exp$^+$)	*AddExp*
	|	(- Exp Exp$^+$)	*SubExp*
	|	(* Exp Exp$^+$)	*MultExp*
	|	(/ Exp Exp$^+$)	*DivExp*
	|	Identifier	*VarExp*
	|	(let ((Identifier Exp)$^+$) Exp)	*LetExp*
	|	(Exp Exp$^+$)	***CallExp***
	|	(lambda (Identifier$^+$) Exp)	***LambdaExp***
Number	::=	Digit	*Number*
	|	DigitNotZero Digit$^+$	
Digit	::=	[0-9]	*Digits*
DigitNotZero	::=	[1-9]	*Non-zero Digits*
Identifier	::=	Letter LetterOrDigit*	*Identifier*
Letter	::=	[a-zA-Z$_]	*Letter*
LetterOrDigit	::=	[a-zA-Z0-9$_]	*LetterOrDigit*



|	(if Exp Exp Exp)	*IfExp*
|	(< Exp Exp)	*LessExp*
|	(= Exp Exp)	*EqualExp*
|	(> Exp Exp)	*GreaterExp*
|	#t | #f	*BoolExp*
|	(car Exp)	***CarExp***
|	(cdr Exp)	***CdrExp***
|	(null? Exp)	***NullExp***
|	(cons Exp Exp)	***ConsExp***
|	(list Exp*)	***ListExp***

Appendix: Interpreter Code Examples

1. Editing .g file

```
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); }
;
```

2. Evaluator

```
class Evaluator {

public Value visit(AddExp e) {
List<Exp> operands = e.all();
double result = 0;
for(Exp exp: operands) {
NumVal intermediate = (NumVal) exp.accept(this);
result += intermediate.v();
}
return new NumVal(result);
}
```

3. AST

```
public interface AST {
        public static class LambdaExp extends Exp {
                List<String> _formals;
```

```
            Exp _body;

            public LambdaExp(List<String> formals, Exp body) {
                  _formals = formals;
                  _body = body;
            }

            public List<String> formals() { return _formals; }

            public Exp body() { return _body; }

            public Object accept(Visitor visitor, Env env) {
                  return visitor.visit(this, env);
            }
      }
}
```

4. Value

```
public interface Value {
      static class FunVal implements Value { //New in the funclang
            private Env _env;
            private List<String> _formals;
            private Exp _body;
            public FunVal(Env env, List<String> formals, Exp body) {
                  _env = env;
                  _formals = formals;
                  _body = body;
            }
            public Env env() { return _env; }
            public List<String> formals() { return _formals; }
            public Exp body() { return _body; }
            public String tostring() {
                  String result = "(lambda ( ";
                  for(String formal : _formals)
                        result += formal + " ";
                        result += ") ";
                        result += _body.accept(new Printer.Formatter(), _env);
                        return result + ")";
            }
      }
}
```