Midterm: Solution

Instructions:

- The exam will be held from 11:00 am - 12:15 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 and 1 extra credit question, a total of 77 pt + 5 pt extra credit. Please plan your time accordingly.
- Good luck!!!

1. (16 pt) Multiple choice questions: select *all* the correct answers for the following questions.

   - (4 pt) Which of the following statements are/is true about context free grammars?
     (a) non-terminals are tokens
     (b) the leaf nodes in a parse tree are either terminals or non-terminals
     (c) a context free grammar always has a start symbol
     (d) we can write different grammars for the same programming language

     Sol. c, d

   - (4 pt) Which of the following is/are true about program paradigms?
     (a) we can use imperative, functional and logic programming paradigms to solve a same program
     (b) high order functions is one of the key features of functional programming paradigms
     (c) side effects makes programs hard to verify
     (d) domain specific languages are imperative programming languages

     Sol. a, b, c

   - (4 pt) Which of the following is/are true about ambiguity?
     (a) if a grammar is ambiguous, then all the strings generated by the grammar have two or more parse trees

(b) when there are multiple leftmost derivations, they always generate the same parse trees even for an ambiguous grammar

(c) even for an ambiguous grammar, left-most and right-most derivations can generate the same parse trees

(d) an ambiguous grammar can always be improved to be an unambiguous grammar

Sol. c

- (4 pt) Which of the following is/are true about programming languages?

(a) a programming language is used to express computations

(b) a programming language specification should consist of both grammar and semantic rules

(c) practical programming languages use mostly context free grammar rules to specify the syntax

(d) we can implement the programming languages in either compilers or interpreters

Sol. a, b, c, d

2. (6 pt) Derivations and parse tree. (Use Appendix for grammar if needed)

(a) (2 pt) Write the rightmost derivation for the Varlang program: (let ((x 5))(let ((x 2)) x))

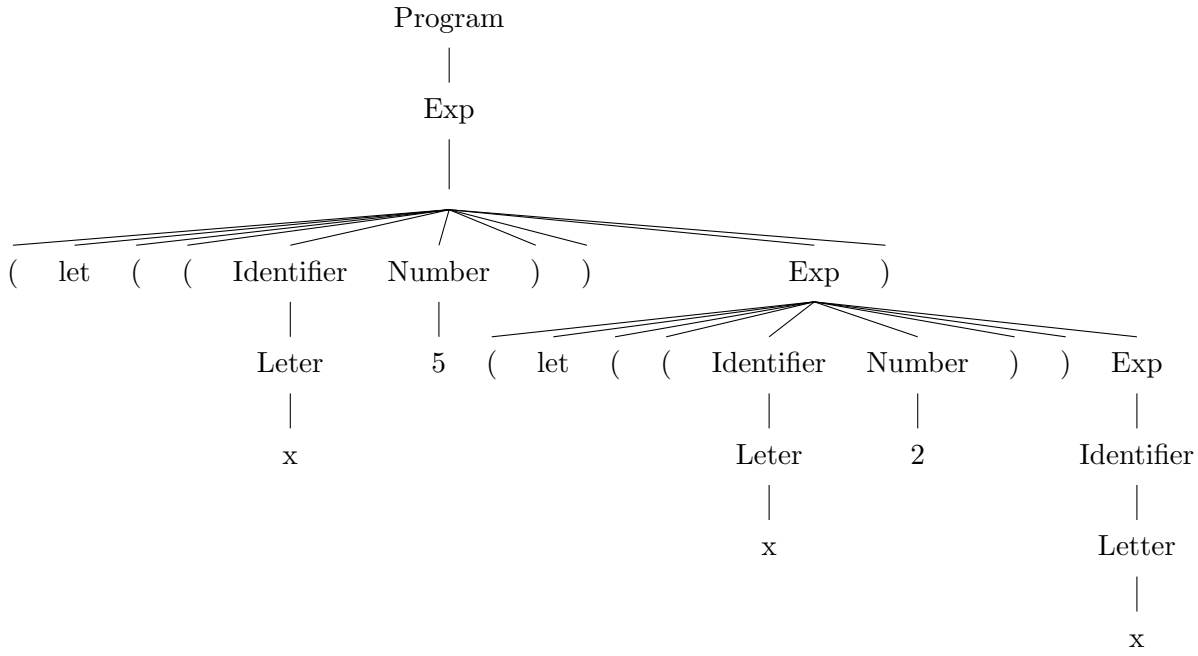
Sol.

```
Program
=> Exp
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' Exp ')'
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' '(' 'let' '(' ('('
   ↪ Identifier Exp ')')+ ')' Exp ')' ')'
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' '(' 'let' '(' ('('
   ↪ Identifier Exp ')')+ ')' Identifier ')' ')'
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' '(' 'let' '(' ('('
   ↪ Identifier Exp ')')+ ')' Letter ')' ')'
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' '(' 'let' '(' ('('
   ↪ Identifier Exp ')')+ ')' x ')' ')'
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' '(' 'let' '(' '('
   ↪ Identifier Exp ')' ')' x ')' ')'
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' '(' 'let' '(' '('
   ↪ Identifier Number ')' ')' x ')' ')'
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' '(' 'let' '(' '('
   ↪ Identifier 2 ')' ')' x ')' ')'
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' '(' 'let' '(' '(' Letter 2
   ↪  ')' ')' x ')' ')'
```

```
=> '(' 'let' '(' ('(' Identifier Exp ')')+ ')' '(' 'let' '(' '(' x 2 ')'
   ↪ ')' x ')' ')'
=> '(' 'let' '(' '(' Identifier Exp ')' ')' '(' 'let' '(' '(' x 2 ')' ')'
   ↪  x ')' ')'
=> '(' 'let' '(' '(' Identifier Number ')' ')' '(' 'let' '(' '(' x 2 ')'
   ↪ ')' x ')' ')'
=> '(' 'let' '(' '(' Identifier 5 ')' ')' '(' 'let' '(' '(' x 2 ')' ')' x
   ↪  ')' ')'
=> '(' 'let' '(' '(' Letter 5 ')' ')' '(' 'let' '(' '(' x 2 ')' ')' x ')'
   ↪  ')'
=> '(' 'let' '(' '(' x 5 ')' ')' '(' 'let' '(' '(' x 2 ')' ')' x ')' ')'
```

(b) (2 pt) Draw the parse tree based on the above derivation

Sol.



(c) (2 pt) What is the value for this Varlang program.

Sol. 2

3. (10 pt) Grammar understanding and analysis:

Given the following grammar:

$$F \rightarrow B\$F|B$$
$$B \rightarrow D\#D|D$$
$$D \rightarrow (F)|x|y$$

where the terminals are $\$, \#, (,), x, y$ and F is the start non-terminal.

(a) (3 pt) Does \$ have a higher precedence than #? Justify your answer.

(b) (3 pt) Are # and \$ left or right associative? Justify your answer.

(c) (2 pt) Can the grammar generate the string $x\$x\#y\#(y\$x)$? If so, show the derivation; if not, justify your answer.

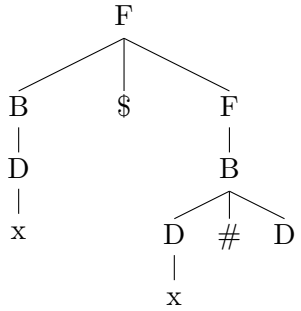
(d) (2 pt) Is this grammar ambiguous? Justify your answer.

Sol.

(a) No, # has a higher precedence because it is lower in the grammar rules, which means it will be evaluated first

(b) \$ is right associative and # is decided by strings using parenthesis

(c) No, because the first y cannot be generated. See the following parse tree where it get stuck:



(d) No, the grammar is not ambiguous. The grammar uses parenthesis to remove ambiguity.

4. (6 pt) Grammar construction: give a context-free grammar that generates palindromes (strings that read the same forward and backward, including empty strings) consisting of letters a's and b's. Examples of strings: ababa, aba, abba, bab

   Sol. $S \rightarrow aSa|bSb|a|b|\epsilon$

5. (10 pt) Bound/free variables, scope:

   (a) (5 pt) Mark the free and bound variables in the following program:

```
(let ((f (lambda (f x) (+ x (f x)))))
          (let ((g (lambda (f g) (f g 3))))
                    (let ((h (lambda (x) (* 3 x)))) (g f h))))
```

   Sol.

```
(let ((f (lambda (f x) (+ x (f x)))))
                          B  B B
        (let ((g (lambda (f g) (f g 3))))
                               B B
              (let ((h (lambda (x) (* 3 x)))) (g f h))))
                                     B      B B B
```

   (b) (2 pt) Does this program contain a hole? If so, please list it/them.
   Sol. the expression (let ((g (lambda (f g) (f g 3)))) creates a hole for the variable f defined in the parent scope

   (c) (3 pt) Compute the value for this program. (List steps to get partial credit)
   Sol.

   i. The first let assigns (lambda (f x) (+ x (f x)) to f
   ii. The body of the first let we have another let expression which assigns the lambda expression (lambda (f g) (f g 3) to g.
   iii. The body of the second let expression is another let expression which assigns the lambda expression (lambda (x) (* 3 x)) to h
   iv. The body of the third let expression applies the call expression (g f h), where
      - the application of (g f h) applies (f h 3)
      - (f h 3) applies (+ x (f x)), where x=3 and f=h=(lambda (x) (* 3 x)). So the call expression (f x) returns 9.
      - The expression is reduced to (+ 3 9), which is equal to 12.

6. (10 pt) Functional Programming:

   (a) (5 pt) Write a FuncLang program to compute the string compression: you will identify and remove any repetitive letters in a string. For example: given a string "aab", returns "ab"; given a string "1000000", returns "10"; given a string "100100", returns "1010". You can use a list to represent the input string.

   $ define compression (lambda (lst) (...))
   Sol.

```
(define consecutive
  (lambda (ltt lst)
    (if (null? lst)
      (list)
      (if (= (car lst) ltt)
        (consecutive ltt (cdr lst))
        (cons (car lst) (consecutive (car lst) (cdr lst)))))))

(define compression
  (lambda (lst)
    (if (null? lst)
      (list)
      (cons (car lst) (consecutive (car lst) (cdr lst))))))
```

   (b) (5 pt) Write a FuncLang program to compute an integer number from a list of digits from 0-9.

   Example scripts:

   $ ConvertInt(list (1 2 3))
   $ 123
   $ ConvertInt(list (0 2 3))
   $ 23
   $ ConvertInt(list (3 4 9 1))
   $ 3491

   Sol.

```
(define size (lambda (lst) (if (null? lst) 0 (+ 1 (size (cdr lst))))))
(define multp (lambda (n) (if (= n 1) 1 (if (> n 1) (* 10 (multp (- n 1))) 0))))
(define ConvertInt
  (lambda (lst)
    (let ((s (size lst)))
      (if (null? lst)
        0
        (+ (* (multp s) (car lst)) (ConvertInt (cdr lst)))))))
```

7. (20 pt) Implement the languages:

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

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

   Value of CallExp

   Write your evaluator of the `CallExp` to support the semantics.
   Assume the `CallExp` is defined and extended from `Exp` class with the following signature:

   - constructor: `CallExp(Exp operator, List<Exp> operands)`
   - method: `public Exp operator()`
   - method: `List<Exp> operands()`

   Assume the `FunVal` is defined and implemented from `Value` interface with the following signature:

   - constructor: `FunVal(Env env, List<String> formals, Exp body)`
   - method: `public Env env()`
   - method: `public List<String> formals()`
   - method: `public Exp body()`

   Assume you can extend environments by creating a new object using the `ExtendEnv` constructor:

   - `new_env = new ExtendEnv(old_env, name_variable, value);`

   Modify the `CallExp` expression in the Evaluator

```
public class Evaluator implements Visitor<Value> {
  public Value visit(CallExp e, Env env) {
    // write the evaluation of e here
    Object result = e.operator().accept(this, env);
    if(!(result instanceof Value.FunVal))
      return new Value.DynamicError("Operator not a function in call " + ts.
          ↪ visit(e, env));
    Value.FunVal operator = (Value.FunVal) result; //Dynamic checking
    List<Exp> operands = e.operands();

    // Call-by-value semantics
    List<Value> actuals = new ArrayList<Value>(operands.size());
    for(Exp exp : operands)
      actuals.add((Value)exp.accept(this, env));

    List<String> formals = operator.formals();
    if (formals.size()!=actuals.size())
      return new Value.DynamicError("Argument mismatch in call " + ts.visit(e
          ↪ , env));
```

```
      Env fun_env = operator.env();
      for (int index = 0; index < formals.size(); index++)
        fun_env = new ExtendEnv(fun_env, formals.get(index), actuals.get(index)
            ↪ );


      return (Value) operator.body().accept(this, fun_env);
    }
  }
```

(b) (15 pt) Extend the language to support a "substring" operation on the positive numbers. The operation returns a section of numbers as well as the decimal point, if any, specified by an index range. If the range index is out of bound or when a non-positive integer is given, you return -1, indicating an error. See example scripts below:

$ (substr 50010 2 4)
$ 001
$ (substr 50010 1 1)
$ 5
$ (substr (let ((a 50010)) a) 2 4)
$ 001
$ (substr (+ 34 2) 1 (* 1 1 (- 4 2))
$ 34
$ (substr -10 1 2)
$ -1
$ (substr 10 -1 2)
$ -1
$ (substr 10 1 4)
$ -1
$ (substr (/ 10 3) 0 4) # note that 10/3 = 3.333333...
$ 3.33

Assume the `SubStrExpr` is defined (so you do not need to write the definition of this AST node) and extended from `Exp` class with the following signature:

- constructor: `SubStrExpr(Exp src, Exp start, Exp end)`
- method: `public Exp getSrc()`
- method: `public Exp getStart()`
- method: `public Exp getEnd()`

As the first step, please modify the grammar below:

```
exp returns [Exp ast]:
| subs=substrexp { $ast = $subs.ast; }

substrexp returns [SubStrExpr ast]:
// complete grammar here
'(' 'substr'
    str=exp
    s=exp
    e=exp
')' { $ast = new SubStrExpr($str.ast, $s.ast, $e.ast); }
;



;
```

Then you can complete the following Evaluator:

```
@Override
public Value visit(SubStrExpr e, Env env) {
  // write the evaluation here
  NumVal value = (NumVal) e.getStr().accept(this, env);
  NumVal startValue = (NumVal) e.getStart().accept(this, env);
  NumVal endValue = (NumVal) e.getEnd().accept(this, env);

  if (value.v() >= 0) {
    String str = String.valueOf(value.v());
    int start = (int) startValue.v();
    int end = (int) endValue.v();

        if (start >= 0 && start <= end && str.length() > start && end <= str.
            ↪ length())
            return new StringVal(new String(str.substring(start, end)));
  }

  return new StringVal("-1");
}
```

Extra Credit Questions:

1. (5 pt) Improving the compression algorithm in Question 6(a). Write a program to compute the string compression: you will remove any repetitive letters in a string and use *letter(frequency)* to replace them. For example: given a string "aab", the compression returns "a(2)b"; given a string "1000000", the compression returns "10(6)". You can use a list to represent the input string.

   Sol.

```
(define consecutive2
  (lambda (n ltt lst)
    (if (null? lst)
      (if (> n 1)
        (cons "(" (cons n (cons ")" (list))))
        (list))
      (if (= (car lst) ltt)
        (consecutive2 (+ 1 n) ltt (cdr lst))
        (if (> n 1)
          (cons (car lst) (cons "(" (cons n (cons ")" (consecutive2 1 (car lst) (cdr
             ↪ lst))))))
          (cons (car lst) (consecutive2 1 (car lst) (cdr lst))))))))


(define compression2
(lambda (lst)
  (if (null? lst)
    (list)
    (cons (car lst) (consecutive2 1 (car lst) (cdr lst))))))
```

Appendix: Grammar for Funclang

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*

Appendix: Interpreter Code Examples

1. Grammar

```
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);
  }
```