Midterm

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 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.
- The exam has a total of 8 questions and 2 extra credit questions (total points: 70 + 10).
- Good luck!!!

1. Multiple Choice Questions: select *all* the correct answers for the following questions.

   (a) (3 pt) Given the following grammar, select the strings that it accepts.
   $real - number \rightarrow signpart.part|signpart$
   $sign \rightarrow +|-$
   $part \rightarrow digit|digitpart$
   $digit \rightarrow 0|1|2|3|4|5|6|7|8|9$

   i. `+32.52`
   ii. `3`
   iii. `-.5`
   iv. `-000.`

   Sol i

   (b) (3 pt) What types of programming paradigms available?

   i. logic programming
   ii. functional programming
   iii. object-oriented programming
   iv. imperative programming

   Sol i,ii,iii,iv

   (c) (3 pt) Which of the following is/are true?

   i. terminals are tokens
   ii. a grammar always has a start symbol
   iii. the nodes in a parse tree are either terminals or non-terminals
   iv. even for an ambiguous grammar, left-most and right most derivations can generate the same parse trees

   Sol i,ii,iii,iv

2. (6 pt) What are the three parts of programming languages? Provide examples to explain each part.

Sol

Atomic computation, composition, abstraction.

For C programming language

(a) the built-in `+,-,*,/` are atomic operators.

(b) the conditional branch (`if` statement), loops (`for, while, do-while`) are composition

(c) functions are abstractions

3. (15 pt) Consider the following grammar:

$S \to S + A|S - A|A$
$A \to A * B|A/B|B$
$B \to 1|2|3|4|5|6|7|8|9$

(a) (2 pt) What is the terminals and non-terminals?

(b) (3 pt) Construct the parse tree for the string `1/3+8-2*4`

(c) (5 pt) Is the grammar ambiguous or not? Justify your answer.

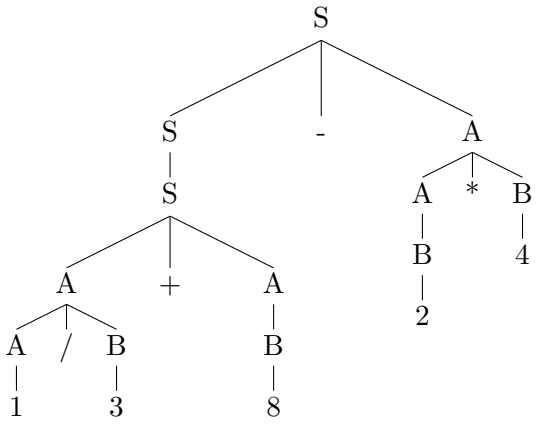
(d) (5 pt) Extend the grammar to support braces, e.g. `1*(2+3)`, where the expression in braces should have the highest priority among all the operators.

(if you need more space, please write on the back of this page.)

Sol

(a)
- terminals: `+,-,*,/`
- non-terminals: `S,A,B`

(b)



(c) Not ambiguous. There's no string that has different parse tree for this grammar. The grammar used associativity and precedence of the operators, which are common techniques to remove ambiguous.

(d) $S \to S + A|S - A|A$
$A \to A * B|A/B|B$
$B \to 1|2|3|4|5|6|7|8|9|(S)$

4. (5 pt) Identify free and bound variables in the following expression:
(let ((addConst (lambda (x y) (+ x y c)))) (addConst 342 z))

Sol

- free variables: `c,z`
- bound variables: `x,y,addConst`

5. (5 pt) Write an Arithlang expression that uses +,-,/,* and is equal to 10.

Sol

```
(* (/ (- (+ 5 10) 11) 2) 5)
```

6. (5 pt) Write a Varlang expression that calculates the area of a triangle($\frac{1}{2} * base * height$) given the base is 10 and the height is 6. You are also restricted to defining a single variable per let expression.

Sol

```
(let ((base 10))
(let ((height 6))
(* (/ 1 2) base height)))
```

7. (10 pt) Write Funclang programs to accomplish the following tasks.

   (a) (3 pt) Construct a global variable `mylist` that holds a list of three pairs, (1,3) (4,2) (5,6).

   (b) (7 pt) Write a function `nth` that takes two arguments `l` and `n`, where `l` is a list of pairs, and `n` is an integer, returns the larger number of the two numbers in the nth pair. If the n is out of range of the list, return -1. Some examples of using `nth` (with the above `mylist` variable):

```
$ (nth mylist 1)
$ 3
$ (nth mylist 2)
$ 4
$ (nth mylist 4)
$ -1
$ (nth mylist 0)
$ -1
$ (nth mylist -4)
$ -1
```

Sol

(a) a

```
(define mylist (list (cons 1 3) (cons 4 2) (cons 5 6)))
```

(b) b

```
(define nth
  (lambda (lst n)
    (if (< n 1) -1
      (if (null? lst) -1
        (if (= n 1)
            (let ((p (car lst)))
              (if (> (car p) (cdr p))
                  (car p) (cdr p)))
          (nth (cdr lst) (- n 1)))))))
```

8. (15 pt) Extend the language to support modular operator (`%`). E.g. `(% 8 3)` returns 2, and `(%8 3 2)` returns 0. Write the grammar for the new operator, and complete the given evaluator class. To save the time, you don't need to write down the class `ModExp` in AST.java. But you can assume that The `ModExp` class is extended from the `CompoundArithExp` class and has the method `public List<Exp> all();` to extract operands.

   (a) (5pt) grammar. You only need to complete the production rule of `modexp`.

   (b) (10 pt) Evaluator

Sol

(a)

```
class Evaluator {
  public Value visit(ModExp e) {
    // write the evaluation of e here
    List<Exp> operands = e.all();
    Exp first = operands.get(0);
    NumVal first_val = (NumVal)first.accept(this);
    double result = first_val.v();
    for (int i=1;i<operands.size();i++) {
      NumVal v = (NumVal)operands.get(i).accept(this);
      result = result % v.v();
    }
    return new NumVal(result);
  }
}
```

(b)

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

9. (Extra Credit: 8 pt) For Question 7: write a function that returns a new list, each element is the multiplication of the two elements in the pair located in a given list, e.g., for the `mylist` defined above, the output is (3, 6, 30)

Sol

```
(define mullist
  (lambda (lst)
    (if (null? lst) lst
      (cons (* (car (car lst)) (cdr (car lst)))
            (mullist (cdr lst))))))
```

10. (Extra Credit: 2 pt) Write a poem on any topics related to programming languages

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*