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. 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.
- The exam has a total of 8 questions and 1 extra credit question. Please plan your time accordingly.
- Good luck!!!

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

   - (4 pt) Given the following grammar, select the strings that it accepts.

$$
\begin{array}{ll}
\textit{stmt} & \rightarrow \textbf{declare id } \textit{optionList} \\
\textit{optionList} & \rightarrow \textit{optionList option} \mid \epsilon \\
\textit{option} & \rightarrow \textit{mode} \mid \textit{scale} \mid \textit{precision} \mid \textit{base} \\
\textit{mode} & \rightarrow \textbf{real} \mid \textbf{complex} \\
\textit{scale} & \rightarrow \textbf{fixed} \mid \textbf{floating} \\
\textit{precision} & \rightarrow \textbf{single} \mid \textbf{double} \\
\textit{base} & \rightarrow \textbf{binary} \mid \textbf{decimal}
\end{array}
$$

     (a) declare foo real
     (b) declare id complex complex
     (c) declare id fixed
     (d) declare id single float

   - (4 pt) Which of the following is/are true about program paradigms?
     (a) functional programming paradigm treats computation as mathematical functions and pure functional programming languages are side-effect free
     (b) the logic programming languages are suitable for programming AI systems
     (c) recursion is a feature that only functional programming paradigm supports
     (d) domain specific languages are imperative programming languages

2. (12 pt) Understand the syntax and semantics of a programming language. Answer the following questions:

   (a) (3 pt) What is syntax? How to formally specify syntax?

   (b) (3 pt) What is semantics? How to formally specify semantics?

   (c) (2 pt) Why do we need formal syntax and semantics when designing and implementing a programming language?

   (d) (2 pt) Why practical programming languages often need natural language descriptions in addition to formal syntax and semantic specifications?

   (e) (2 pt) What makes a good programming language in your opinion?

3. (15 pt) Consider the following grammar:

$$S \to S \cup S | S \cap S | A$$
$$A \to A \times A | A \infty A | B$$
$$B \to \neg B | C$$
$$C \to var | foo | bar | abc | def$$

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

(b) (3 pt) Construct the parse tree for the string $foo \infty \neg bar \cap abc \times var$

(c) (5 pt) Is the grammar ambiguous or not? If not, justify your answer. If yes, eliminate the ambiguity and provide your new grammar below without introducing new terminals.

(d) (5 pt) Extend the grammar to support braces, e.g. $(foo \infty bar) \cap var$, where the expression in braces should have the highest priority among all the operators. Braces are also allowed to enclose a single identifier, e.g. (foo), and is equivalent to the one without braces. The grammar shall still accept the previously accepted programs.

4. (8 pt) Identify free and bound variables in the following expression. Write F (for free variables) or B (for bound variables) under each variables in the description.

```
(let ((foo (lambda (a b) (+ a c))) (c d) (d e)) (foo (+ a b c) (+ d e)))
```

5. (5 pt) Write a Varlang program that has multiple let expressions and also has a hole in the scope. This program needs to use all the arithmetic operators + , −, × and / and aims to calculate 342.

6. (5 pt) Write a FuncLang program that takes a list of lower case characters and returns a list of ASCII values for the characters in the list. You can use the function `ascii` to compute the ASCII value for a character. For example `(ascii 'a')` returns integer 97.

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

   (a) (2 pt) Construct a global variable `mylist` that holds a list of three CONS pairs, (1,3) (4,2) (5,6). Note that the the pair should be *cons pair*, i.e. applying `car` and `cdr` should be able to retrieve the first and second component, respectively.

   (b) (6 pt) write a function `apply-on-nth` that takes three arguments *op*, *lst*, *n*, where *op* is a function, *lst* is a list of pairs, *n* is an integer. The return value should be the result of applying *op* on the *n*-th pair in the list.

   If n is out of range of the list, return -1. You can assume *op* is a function valid to accept two arguments.

   Some examples of using `apply-on-nth` with above *mylist* variable:

```
$ (apply-on-nth + mylist 1)
-> 4                 // 1+3
$ (apply-on-nth - mylist 2)
-> 2 (               // 4-2
$ (apply-on-nth * mylist 8)
-> -1                // third parameter out of range
$ (apply-on-nth / mylist -1)
-> -1               // third parameter out of range
```

   (c) (2 pt) Convert the above FuncLang program into the curried form

8. (15 pt) Extend the language to support switch case. The signature of switch-case:

```
(switch e0
        case e1 body
        case e2 body
        default body)
```

The switch expression will check whether the value of e0 is equal to the following cases from one by one. If equal, value of the corresponding body expression is returned as the result. If no matching found, the value of the body of default is returned. There must be at least one `case` clause and exactly one `default`. Some examples:

```
(define foo
  (lambda (var)
    (switch var
            case 1 (+ var 2)
            case 2 (- var 2)
            case 3 (* var 2)
            case 4 (/ var 2)
            default var)))

(foo 1) ; -> 3
(foo 2) ; -> 0
(foo 3) ; -> 6
(foo 4) ; -> 2
(foo 5) ; -> 5
```

To save time, you are only required to complete the grammar and evaluator class. You can assume `SwitchExp` in defined and extended from `Exp` class, and has the following signature:

- constructor:
  `public SwitchExp(Exp e0, ArrayList<Exp> cases, ArrayList<Exp> bodies, Exp defbody)`
- method: `public Exp e0()`
- method: `public ArrayList<Exp> cases()`
- method: `public ArrayList<Exp> bodies()`
- method: `public Exp defbody()`

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

```
switchexp returns [SwitchExp ast]
locals [ArrayList<Exp> cases=new ArrayList<Exp>(), ArrayList<Exp> bodies=new Ar
    :
// complete grammar here
```

;

(b) (10pt) Evaluator. You need to complete the visit method for `SwitchExp`

```
public class Evaluator implements Visitor<Value> {
    public Value visit(SwitchExp e, Env env) {
    // write the evaluation of e here
```

```
  }
}
```

9. (Extra Credit: 12 pt) Write a FuncLang program called `Shuffle`, which takes an input list and "shuffles" it and returns a list whose members are ordered randomly.

   - (6 pt) Use Random(lst) (it randomly selects an element from the list) to write your program.
   - (6 pt) Use Random(n, m) (it returns a random number between n and m, where it requires $n < m$) to write your program.

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);
  }
```