Final exams
20 points multiple choices
60 points problem solving: reflang, typelang, lambda calculus, logic programming

What is a PL?
- A language to express/communicate/specify computations to computers
- A language in which a developer writes instructions for computers

Programs consist of/ a PL should be able express:
- Atomic instructions
- Composition: e.g., control flow
- Abstraction: e.g., functions, variables

PLs that can work for different programming paradigms
- Imperative: a list of instructions to accomplish a task, low level data structures, iterations
- Functional: declarative (declare a problem as a combination of functions), using functions to specify computations, e.g., Scheme
- Logic: declarative (declare a problem as a set of constraints)

- Functional and logic programming: both good at solving recursion problems (numbers, lists)
- Functional programming: using recursive functions and high order functions to represent computation
- Logic programming: model the facts and relations that form a constraint system, it can be related to real world relations, numbers, graphs

How to specify a PL?
- What is “a PL software”: take in a program and output a value
- Syntax (Grammar): specify the input of a program
- Semantics (Operational semantics, typing rules)

Grammar
Practical programming languages mostly can be specified using context free grammar (CFG)
CFG: (Start, F, R, S)
A derivation is a sequence of applications of production rules to produce a string from the Start symbol. At each step:
(1) Which production rule to select
(2) Which non-terminal symbol to replace
We can generate a parse tree from a derivation. Parse tree is used to generate values. If a parse tree is not unique, there is an ambiguity
Some reasons to cause ambiguity:
- No operator precedence
- Is it left associative or right associative? Not defined in the grammar
- …

1. Read grammars and able to write programs follow the grammar
2. Modify grammar to remove ambiguity
3. Design grammars to specify the patterns of a string

Inference rules
Operational semantics/typing rules
If the conditions above hold, the conclusion bottom can be derived
Operational semantic rules: specify how the value of an expression is computed from its subexpression, how to environment and heap changes as the result of subexpressions
Typing rules: specify how the type of an expression is derived from the types of its subexpressions

1. Understand and implement the inference rules
2. Extra credit if you can write formal inference rules

How to implement a PL?

- Interpreter (extend values, add AST nodes, extend visitor interface, update the heap, implement the type checking rules)

Typical features for a PL?

- Arithmetics (ArithLang)
- Variables (VarLang, DefineLang)
- Functions (FuncLang)
- Memory Management (RefLang)
- Types (TypeLang)
- Data structures (List and pair)
- Control flow (if-then-else)

  ….

(1) Prefix, infix and postfix: see ArithLang syntax
(2) Variable scope: define and use of a variable; for a use, which definition it bounds to? It achieves via “environment”, see VarLang operational semantics, bound/free variable
(3) Functions as first-class objects in PL: **being able to do what everyone else can do.”** supports all the operational properties, being able to be assigned to a variable, passed as a function argument and return from a function (high order function)

Lambda calculus: Turing complete

- Smallest languages
  - func def, func app, name/var
  - single argument functions
  - no name so no recursion
  - high order functions
- Beta reduction

- Evaluation order
- Church encoding: how to simulate data types, arithmetics, boolean computations …

(4) RefLang: decisions with heap memory -- should a developer has direct controls over the heap? And how much control they should have?
Operational semantics will keep track of values, environment, heap
RefLang programming: using functions with pointers to simulate data structures

(5) Typelang: type, type rules, type systems, typed language, static type/dynamic type
Typelang programming and type checking rules

Logic programming

- Write prolog programs for numbers, lists, real-world constraints
- Understand how a Prolog program executes *unification, backtrack, goal directed reasoning*