# Final Exam Review

# Final Exam is Cumulative

- Programming Language Syntax (Ch. 2.1-2.3.3)
- Logic Programming and Prolog (Ch. 12)
- Scoping (Ch. 3.1-3.3)
- Programming Language Semantics (Sc. Ch. 4.1-4.3)
- Functional programming (Ch. 11)
  - Scheme and Haskell, map/fold questions
- Lambda calculus (Ch. 11 Companion)
- Data abstraction: Types (Ch. 7, 8)
- Control abstraction: Parameter Passing (Ch. 9.1-9.3)
- Object-oriented languages (10.1-10.2)
- Concurrency (13.1). “What can go wrong?” questions
- Dynamic languages (Ch. 14 optional)

- Comparative Programming Languages

# Exam 1 Topics

- Formal languages (Lecture 2 plus chapters)
  - Regular languages
    - Regular expressions
    - DFAs
    - Use of regular languages in programming languages
  - Context-free languages
    - Context-free grammars
    - Derivation, parse, ambiguity
    - Use of CFGs in programming languages
    - Expression grammars, precedence, and associativity

# Exam 1 Topics

- Parsing (Lecture 3 plus chapters)
- LL Parsing (Lectures 3 and 4 plus chapters)
  - Recursive-descent parsing, recursive-descent routines
  - LL(1) grammars
  - LL(1) parsing tables
  - FIRST, FOLLOW, PREDICT
  - LL(1) conflicts

# Exam 1 Topics

- Logic programming concepts (Lecture 5 plus chapters
  - Declarative programming
  - Horn clause, resolution principle
- Prolog (Lectures 5, 6, and 7 plus chapters)
  - Prolog concepts: search tree, rule ordering, unification, backtracking, backward chaining
  - Prolog programming: lists and recursion, arithmetic, backtracking cut, negation-by-failure, generate-and-test

# Exam 1 Topics

- Binding and scoping (Lecture 8 plus reading)
  - Object lifetime
  - Combined view of memory
  - Stack management

  - Scoping (in languages where functions are third-class values)
  - Static and dynamic links
  - Static (lexical) scoping
  - Dynamic scoping

# Exam 1 Topics

- Attribute grammars
  - Attributes
  - Attribute rules
  - Decorated parse trees
  - Bottom-up (i.e., S-attributed) grammars

# Exam 2 Topics

- Scheme (Lectures 12 and 13, plus chapters)
  - S-expression syntax
  - Lists and recursion
  - Shallow and deep recursion
  - Equality
  - Higher-order functions
  - **map, foldl,** and **foldr**
  - Programming with **map, foldl,** and **foldr**
  - Tail recursion

# Exam 2 Topics

- Scheme (Lecture 14, plus chapters)
  - Binding with **let, let*, letrec**
  - Scoping in Scheme
  - Closures and closure bindings

# Exam 2 Topics

- Scoping, revisited (Lecture 15, plus chapters)
  - Static scoping
    - Reference environment
    - Functions as third-class values vs.
    - Functions as first-class values
  - Dynamic scoping
    - With shallow binding
    - With deep binding

# Exam 2 Topics

- Lambda calculus (Lectures 15 and 16)
  - Syntax and semantics
  - Free and bound variables
  - Substitution
  - Rules of the Lambda calculus
    - Alpha-conversion
    - Beta-reduction
  - Normal forms
  - Reduction strategies
    - Normal order
    - Applicative order

# Topics

- Haskell (Lectures 19 and 20)
  - Basic syntax
  - Algebraic data types
  - Pattern matching
  - Lazy evaluation
  - Types and type inference
  - Basics of type classes and Maybe monad

# Topics

- Data abstraction and types (Lecture 21)
  - Types and type systems
  - Type equivalence
  - Types in C

# Topics

- Parameter passing mechanisms (Lecture 22)
  - Call by value
  - Call by reference
  - Call by value-result
  - Call by name

  - Call by sharing

# Topics

- Object-oriented languages and polymorphism (Lecture 23)
  - Subtype polymorphism
  - Parametric polymorphism
    - Explicit parametric polymorphism
    - Implicit parametric polymorphism

# Topics

- Concurrency in Java (Lecture 24)
  - Threads and tasks
  - Synchronized blocks
  - Concurrency errors
    - Data races
    - Atomicity violations

# Practice Problems (Quiz 6)

Dyn. with shallow binding: 100
Dyn. with deep binding: 101

```
x : integer := 1

print_routine(i : integer)
    write_integer(i+x)

procedure A(n : integer, P : procedure)
    if n < 100
      B(n+1, P)
    else
      P(n)

procedure B(m : integer, P : procedure)
    x : integer := 0
    A(m, P)

/* begin of main */
A(0, print_routine)
/* end of main */
```

# Practice Problems (Quiz 6)

Consider the problem of figuring out whether two trees (lists in Scheme) have the same fringe, that is, the same leaves, in the same order, regardless of structure. E.g., ((1 2) 3) and (1 (2 3)) have the same fringe. What is a straight-forward way to solve this problem?

Flatten, then compare

# Practice Problems (Quiz 7)

```
type Name = String
data Expr = Var Name
          | Val Bool
          | And Expr Expr
          | Or Expr Expr
          | Not Expr
          | Let Name Expr Expr
```

Fill in the type signature of **find**:

-- Looks up variable **n** in binding environment **env**.

-- Returns first binding or throws Exception if no binding of **n**.

-- Ex: **find "x" [("x",True),("x",False),("y",True)]** returns **True**

find :: ____Name -> [(Name,Bool)] -> Bool____

find n env = head [ bool | (var,bool) <- env, var == n ]

# Practice Problems (Quiz 7)

Fill in the Or and Let arms of **eval**:

-- Purpose: evaluates expression **e** in binding environment **env**

-- Returns the boolean value of **e** or throws an Exception

-- Ex.: **eval (Var "x") [("x",True),("x",False)]** returns **True**

eval :: Expr -> [(Name,Bool)] -> Bool

eval e env =

case e of

Var n -> find n env

Val b -> b

And e1 e2 -> (eval e1 env) && (eval e2 env)

Or e1 e2 -> (eval e1 env) || (eval e2 env)

Not e1 -> not (eval e1 env)

Let n e1 e2 -> eval e2 ((n,(eval e1 env)):env)

# Practice Problems

In programming languages types and type checking

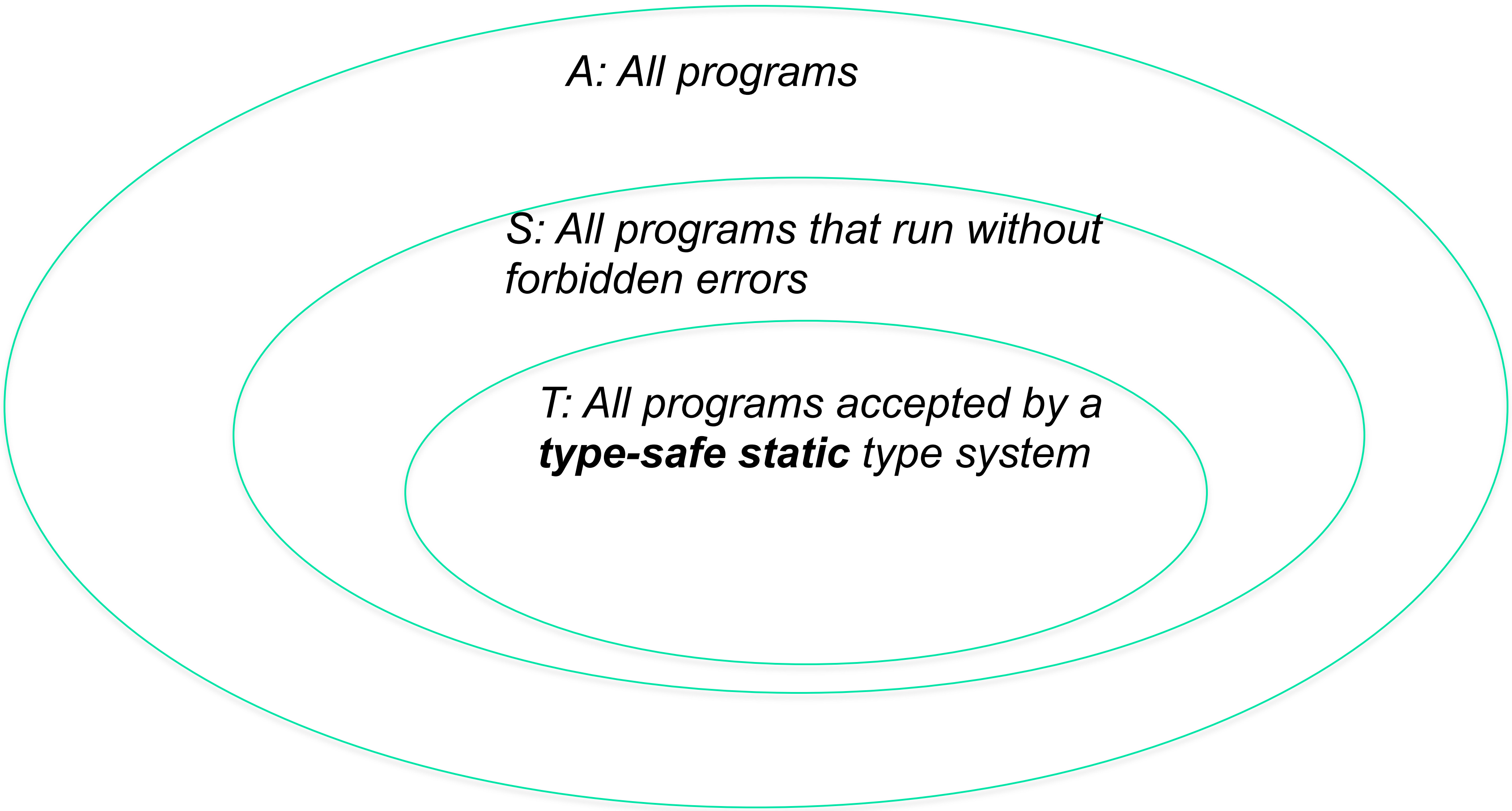
(a) Prevent runtime errors
(b) Abstract data organization and implementation
(c) Document variables and subroutines
(d) All of the above

# Practice Problems

Let *A* denote all syntactically valid programs. Let *S* denote all syntactically valid programs that execute without forbidden errors. Let *T* denote all programs accepted by certain type-safe static type system. Which one best describes the relation between *T*, *S* and *A*?

(a) $T \subset S \subset A$
(b) $T \subseteq S \subset A$
(c) $T \subset S \subseteq A$
(d) $T \subseteq S \subseteq A$

# Cont.


A: All programs
S: All programs that run without forbidden errors
T: All programs accepted by a **type-safe static** type system

# Practice Problems

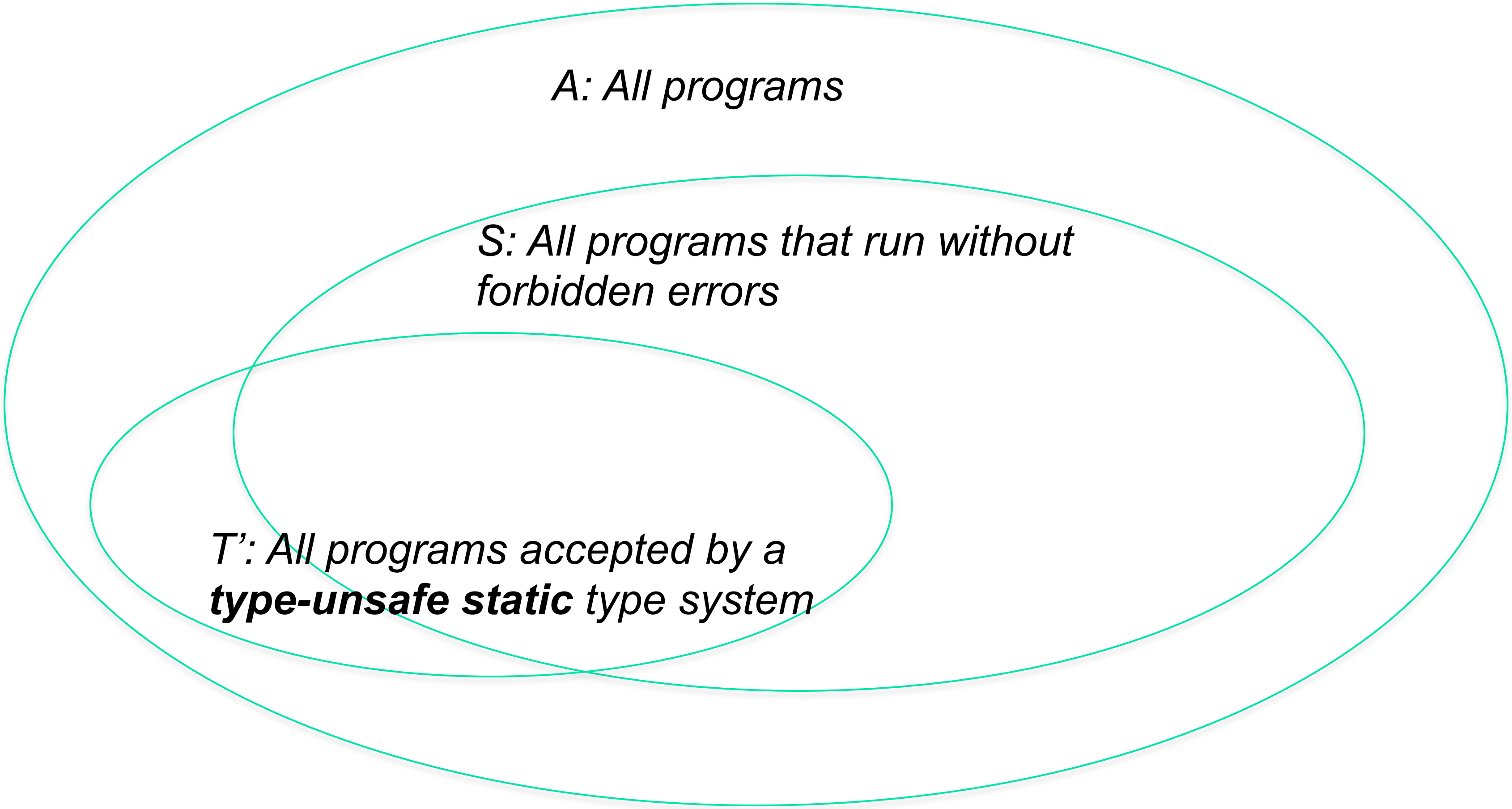
Again, let *S* denote all syntactically valid programs that execute without forbidden errors. Let *T'* denote all programs accepted by certain **type-unsafe** static type system.

$T' \not\subseteq S$ is

(a) true

(b) false

# Cont.


A: All programs
S: All programs that run without forbidden errors
T': All programs accepted by a **type-unsafe static** type system

# Practice Problems

**`int w[10]()`** is an invalid declaration in C. Why?

(a) true
(b) false

In C, functions are third-class values. Thus, we cannot pass a function as argument, return a function as a result, or assign a function value to a variable, or structure.

# Practice Problems

`w` in declaration `int (*w[10])()` is

(a) A function
(b) An array
(c) A pointer

# Practice Problems (Quiz 8)

**`A`** is a 3-dimensional array of **`ints`**:

**`int [0..1,0..1,0..1] A.`**

The elements are ordered in memory as:

A[0,0,0],A[1,0,0],A[0,1,0],A[1,1,0],A[0,0,1],
A[1,0,1],A[0,1,1],A[1,1,1]

This is

(a) Column-major order

(b) Row-major order

(c) Neither

# Practice Problems (Quiz 8)

```
typedef struct { int *i; char c; } huge_record;

void const_is_shallow(
        const huge_record* const r) {
  int *x = r->i; // or just *(r->i) = 0;
  *x = 0;
}
```

Is this a compile time error?

No.

# Practice Problems (Quiz 8)

```
c : array [1..2] of integer
m : integer
procedure R(k,j : integer)
    k := k+1
    j := j+2

/* begin main */
c[1] := 1
c[2] := 2
m := 1
R(m, c[m])
write m, c[m]
/* end main */
```

By value: 1, 1

By reference: 2, 2

By value-result: 2, 3

By name: 2, 4

# Practice Problems (Quiz 9)

```
class Account {
    int amount = 0;
    void deposit(int x) {
        amount = amount + x;
    }
}

class DepositTask implements Runnable {
    public void run() {
        synchronized (this) {
            Main.act.deposit(10);
        }
    }
}

public class Main {

    static Account act = new Account();

    public static void main(String arg[]) throws InterruptedException {
        ExecutorService pool = Executors.newCachedThreadPool();

        pool.execute(new DepositTask());
        pool.execute(new DepositTask());
        pool.execute(new DepositTask());

        pool.shutdown();
        pool.awaitTermination(60, TimeUnit.SECONDS);
    }
}
```

Question 1. What are the possible values for **`act.amount`** in the end?

10,20,30 (will accept 0,10,20,30 as well)

Question 2. Field **`amount`** in **`Account`** has

(d) package visibility

# Practice Problems (Quiz 9)

**Question 3.** (2pts) Generic Java class `MyList` is defined as follows:

```
public class MyList<T extends Number> {
   // Type parameter T is bounded by Number
   // MyList uses interface of Number, e.g., intValue(), doubleValue()
   ...
}
```

Is `MyList<Integer> p = new MyList<Integer>();` a valid instantiation of `MyList`? Note: In Java `Integer` is a subclass of `Number`.

(a) Yes

Is `MyList<String> p = new MyList<String>();` a valid instantiation?

(b) No

# Practice Problems (Quiz 9)

Question 4. Consider the C++ code:

```
bar x;
bar y = x;
```

At **`y = x`** C++ calls

(b) copy constructor **`bar::bar(bar&)`**

# Practice Problems (Quiz 9)

Question 5. Consider the C++ code:

```
bar x, y;
y = x;
```

At **`y = x`** C++ calls

(a) assignment operator

**`bar::operator=(bar&)`**

# Practice Problems (Quiz 9)

Question 6. Now suppose this was Java code:

```
bar x, y;
y = x;
```

At `y = x` Java calls

(b) neither