Final Exam Review

Announcements

- Quiz 9

- Check your Rainbow grades
  - Exams 1-2, Quiz 1-8, HW 1-5 are in
  - We'll have HW 6 and Quiz 9 by tomorrow and we'll have HW 7 as soon as possible, likely by Wednesday
- Let me know if you see any problems

Announcements

- Final exam
  - Thursday December 15th 6:30pm to 9:30pm
  - Darrin 308 and 318. Assigned seating zones.
  - 6 pages crib sheet
  - Instructors and TAs office hours Monday through Thursday. No Mentor office hours.

  - I will post an announcement on Submitty

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

- 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
  - 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, 16, 17)
  - Syntax and semantics
  - Free and bound variables
  - Substitution
  - Rules of the Lambda calculus
    - Alpha-conversion
    - Beta-reduction
  - Normal forms
  - Reduction strategies
  - Fixed-point combinator and recursion

Topics

- Haskell (Lectures 19, 20, and 21)
  - Basic syntax
  - Algebraic data types
  - Pattern matching
  - Lazy evaluation
  - Types and type inference
  - Basics of type classes
  - Maybe and List monads

Topics

- Data abstraction and types (Lecture 22, 23)
  - Types and type systems
  - Type equivalence
  - Types in C

Topics

- Parameter passing mechanisms (Lecture 24_Part1)
  - Call by value
  - Call by reference
  - Call by value-result
  - Call by name
  - Call by sharing

Topics

- Object-oriented languages and polymorphism (Lecture 24_Part2)
  - Subtype polymorphism
  - Parametric polymorphism
    - Explicit parametric polymorphism
    - Implicit parametric polymorphism

Topics

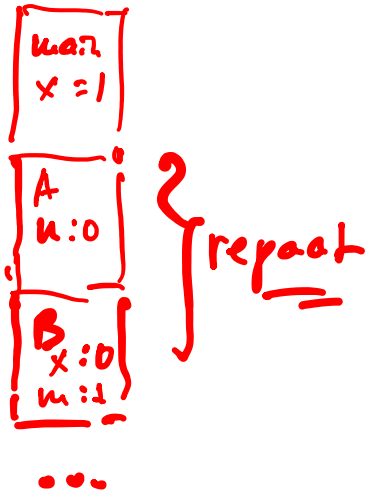
- Concurrency in Java (Lecture 25)
  - Threads and tasks
  - Synchronized blocks
  - Concurrency errors
    - Data races
    - Atomicity violations

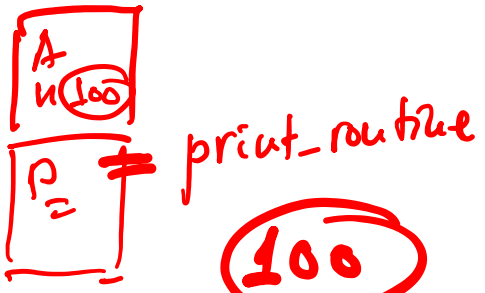
Practice Problems

Dyn. with shallow binding: 100 (binds to closest frame of B's x)

Dyn. with deep binding: 101 (binds to x in main)

```
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 */
```





shallow binding: 100

deep binding: 101

Practice Problems

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 with equal?

(equal? (flatten lis1) (flatten lis2))

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
```

:type find
(Eq a) =>
a -> [(a,b)] -> b

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) $T \subseteq S \subseteq A$

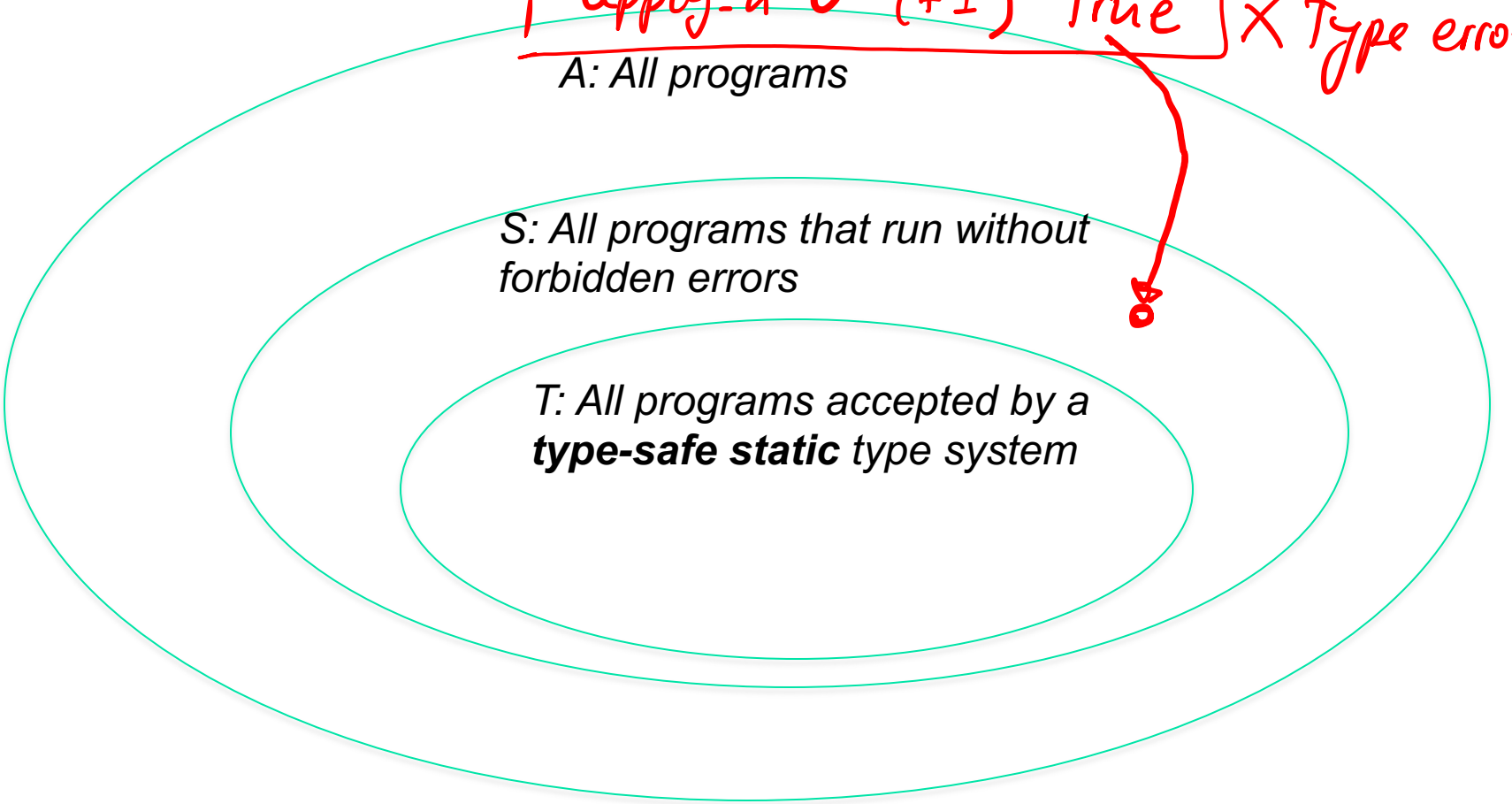
Cont.

```
apply_n n f x =
  if (n == 0) then x else (apply_n (n-1) f (f x))
```

```
Eq a, Num a => a -> (b -> b) -> b -> b
               n       f       x   return
```

```
apply_n 0 (+1) True
```

× Type error



Practice Problems

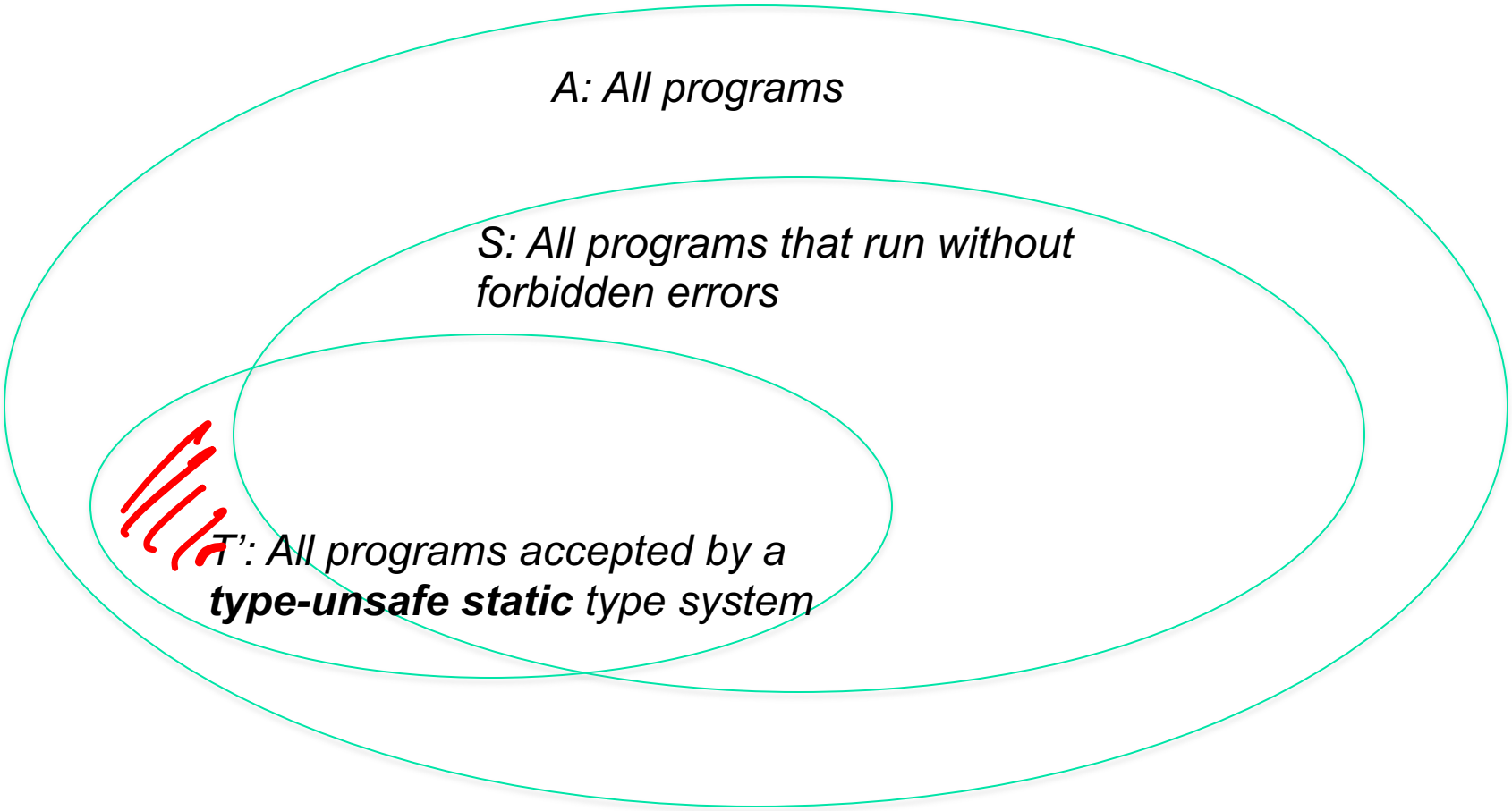
Again, let *S* denote all syntactically valid programs that execute without forbidden errors. Let *T'* denote all programs accepted by certain **type-unsafe** static type system.

$T' \not\subseteq S$ is

(a) true

(b) false

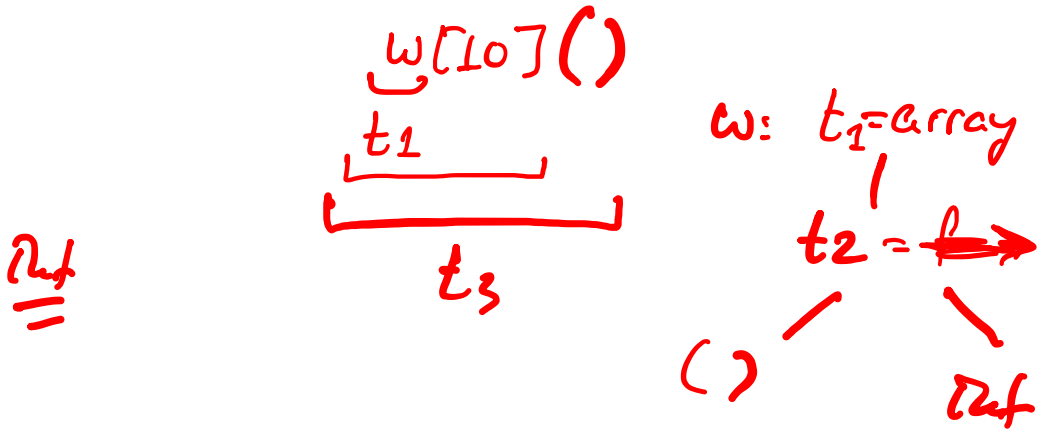
Cont.


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)

$$\begin{aligned}
\texttt{tru} &= \lambda x.\lambda y.\ x \\
\texttt{pair} &= \lambda f.\lambda s.\lambda b.\ b\ f\ s \\
\texttt{fst} &= \lambda p.\ p\ \texttt{tru}
\end{aligned}$$

- Question 1. **`tru v w ->* v`**
- Question 2. **`pair v w ->* λb. b v w`**
- Question 3. **`fst (pair v w) ->* v`**

Practice Problems (Quiz 8)

```
// Type declarations:
type A = array [1..5] of int
type B = array [1..5] of int
type C = A

// Variable declarations:
a : A
b : B
c : C
```

ANON1

ANON2

Question 4. (1pts) `a,b` and `c` are of equivalent type according to *structural equivalence*. TRUE

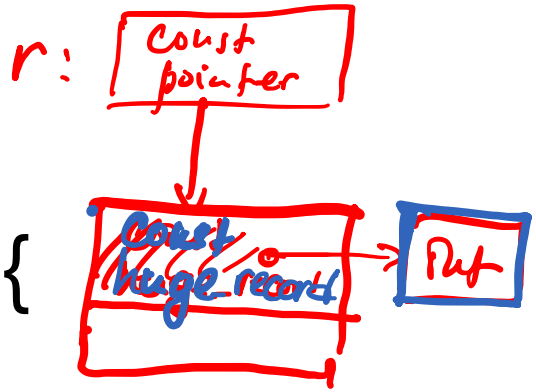
Question 5. (1pts) `a` and `b` are of equivalent type according to *loose name equivalence*. FALSE

Question 6. (1pts) `a` and `c` are of equivalent type according to *loose name equivalence*. TRUE

Question 7. (1pts) `a` and `c` are of equivalent type according to *strict name equivalence*. FALSE

Practice Problems (Quiz 9)

```
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 9)

```
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 name:
m := m + 1
c[m] := c[m] + 2

By value-result:
(1)
k := k+1 k = 2
j := j (1) +2 j = 3
R(m, c[m])

By value: 1, 1
By ref: 2, 2
By value-res: 2, 3
By name: 2, 4

3
2

By ref

Practice Problems (Quiz 9)

Question 7. 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 8. 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 9. Now suppose this was Java code:

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

At `y = x` Java calls

(b) neither

Practice Problems (Exam 1)

Question 2. (Grammars, ambiguity, and precedence. 20pts) Consider the following *expression grammar* over terminal symbols `a`, `b`, `|`, `*`, `(` and `)`. Note: `|` is a terminal symbol in the language and appears in quotes as `'|'`, so that not to be confused with the rule separator |.

$$
\begin{array}{lcl}
S & \rightarrow & R \\
R & \rightarrow & R \text{ '|' } R \;\mid\; R\,R \;\mid\; R^* \;\mid\; \texttt{a} \;\mid\; \texttt{b} \;\mid\; (\,R\,)
\end{array}
$$

(iv) (8pts) Let's call the language generated by the grammar $\mathcal{L}$. Construct an equivalent unambiguous grammar such that all operations are processed according to the standard conventions of $\mathcal{L}$. Hints: In this question, you do need to add nonterminals and productions as we did in class and in homework, assuming bottom-up interpretation of the tree. Recall that according to standard convention of $\mathcal{L}$ '|' has lowest precedence, * has highest precedence, and all binary operations are left-associative.

$$
\begin{array}{lcl}
S & \rightarrow & R \\
R & \rightarrow & R \text{ '|' } R_1 \mid R_1 \\
R_1 & \rightarrow & R_1\, R_2 \mid R_2 \\
R_2 & \rightarrow & R_2^* \mid a \mid b \mid (R)
\end{array}
$$

Practice Problems (Exam 2)

Question 3. (Attribute grammars over the Lambda calculus, 25 pts). Recall the context-free grammar that defines the syntax of the Lambda calculus:

E	$\rightarrow$	x	*Variable*
E	$\rightarrow$	$(\lambda x.E)$	*Abstraction*
E	$\rightarrow$	$(E\ E)$	*Application*

(i) (10 pts) Write an attribute grammar that associates an integer attribute r with each E, such that r holds the number of reducible expressions in E.

$E \rightarrow x$

$E \rightarrow (\lambda x.\ E_1)$

$E \rightarrow (E_1\ E_2)$

$E.r = 0 \quad E.abs = false$

$E.r = E_1.r \quad E.abs = True$

$tmp = E_1.r + E_2.r$

$E.r = tmp + 1$ if $E_1.abs == True$ else tmp

$E.abs = False$

Practice Problems (Exam 2)

Question 3. (Attribute grammars over the Lambda calculus, 25 pts). Recall the context-free grammar that defines the syntax of the Lambda calculus:

E	$\rightarrow$	x	*Variable*
E	$\rightarrow$	$(\lambda x.E)$	*Abstraction*
E	$\rightarrow$	$(E\ E)$	*Application*

(ii) (15 pts) Write an attribute grammar that removes redundant parentheses. Assume the syntactic convention we used in class: (i) application is left associative and (ii) application has higher precedence than abstraction. As an example, $(\lambda x.\ (x\ (x\ y)))$ will turn into $\lambda x.\ x\ (x\ y)$.

$E \rightarrow x$

$E \rightarrow (\lambda x. E_1)$

$E \rightarrow (E_1\ E_2)$

$E.val = x.val$ (scanner) $E.pri = 2$

$E.val = \lambda x.val \oplus E_1.val$ $E.pri = 0$

$op1 = "(" \oplus E_1.val \oplus ")"$ if $E_1.pri < 1$ else $E_1.val$

$op2 = "(" \oplus E_2.val \oplus ")"$ if $E_2.pri \leq 1$ else $E_2.val$

$E.val = op1 \oplus op2$ $E.pri = 1$

Practice Problems (Exam 2)

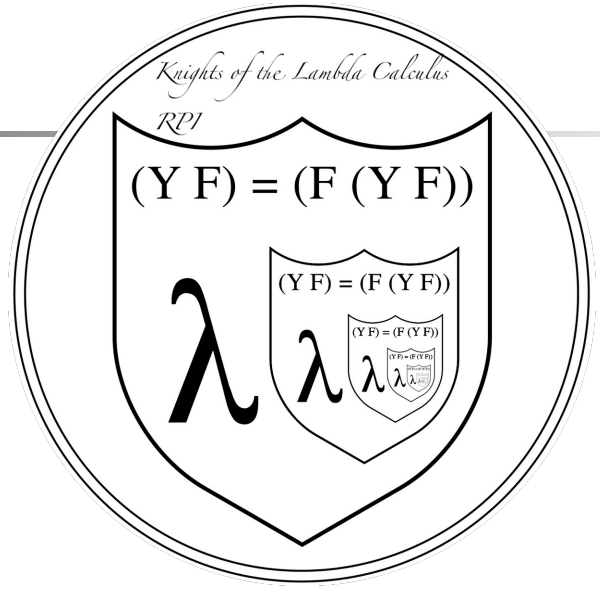
Question 4. (Scheme programming, 15 pts). Consider the following Scheme function.

```
;; requires: (= (length lis) n)
(define (fun lis n)
  (if (zero? n) '()
      (let ((arg (append (cdr lis) (list (car lis)))))
        (cons lis (fun arg (- n 1))))))
```

(fun '(a b c) 3) → ((a b c) (b c a) (c a b))

```
(define (fun-tail lis n acc)
  (if (zero? n) acc
      (let ((arg (append (cdr lis) (list (car lis)))))
        (fun-tail arg (- n 1) (append acc (list lis))))))
```

Thanks!



Modified from: https://en.wikipedia.org/w/index.php?curid=12675230

Source: https://users.cs.northwestern.edu/~robby/logos/

Source: https://www.numi.tech/software-logos/haskell