CSCI 3155 Programming Languages
Final Exam (Fall 2013)

Monday, December 16, 2013

1:30pm - 4:00pm

WRITE YOUR NAME!
DO NOT TURN THE PAGE UNTIL TOLD TO BEGIN!
READ THE FOLLOWING INSTRUCTIONS CAREFULLY!

Student Information

Full Name: ANSWER KEY

Student ID: ______________________

Instructions

This final exam was designed to test your understanding of the concepts presented in the CSCI 3155 Programming Languages course, as well as your ability to apply those concepts using functional programming techniques.

The exam contains **six (6) problems**, each having several sub-problems. Each problem is labeled with its point value, out of a total of 100 points for the exam. There is **one (1) extra-credit problem** at the end, which is worth an extra 15 points.

You have **2.5 hours** to finish the exam. This should be plenty of time, but please be sure to budget your time wisely. You may wish to skim through each question first, and start with the ones you feel will be easiest. Do not get stuck on any one question!

This is a **closed-book, closed-notes** exam! You may *NOT* use electronic devices of any sort (calculators etc. will not help you on the exam).

Write your answers in the space provided under each question. If you need additional space, use the back of a page. Make sure you have written your full name and student ID in the above space!

▶ Problem 1 – Functional Programming (15 pts)

Scala Preliminaries (2 pts)

In this class, we used the functional programming language Scala for many of the projects.

1. List three reasons why functional languages are well-suited for writing interpreters/compilers:

   1. Good pattern matching for handling ASTs
   2. Recursion
   3. Lots of built-in parsing support (typically)

   } examples

2. What does the command sbt test do?

   Runs the tests (e.g. flatspec unit tests)

Consider the following code:

```
val x = Some(n); x match {
  case(Some(123)) => val x = 234; println(x) // line 3
  case(Some(x)) => println(x)                // line 4
  case(x) => println(x)                      // line 5
}
```

1. If the `println` statement on line 3 occurs, what value of x gets printed? 234
2. Answer the above for lines 4 and 5.

   4 ⇒ n

   5 ⇒ Some(n)

List Processing (5 pts)

In Scala, you can use the "cons" operation `::` to append something to a list.

1. Using only `::` and the empty list `Nil`, build the list `List(1,2,3,4,5)`.

   1 :: 2 :: 3 :: 4 :: 5 :: Nil

2. Assume the above list is stored in variable `x`, and use the `map` function to create a new list consisting of the squares of the elements in `x`. Reminder: this function has the signature `def map[B](f:  (A) => B): List[B]`

   x.map(y ⇒ y*y)

3. Now, use the `foldLeft` function to sum the elements of `x`. Reminder: this function has the signature `def foldLeft[B](z:  B)(f:  (B, A) => B): B`

   x.foldLeft(0) {case(acc, x) ⇒ acc+x}

Recursion (5 pts)

1. Write a recursive list-concatenation function. Remember ~~x.last~~ gets the final element of `x`.

```
def concat(x : List[Int], y : List[Int]) : List[Int] = {
  y match {
    Nil => x
    a :: more => concat(x ++ List(a), more)
  }
}
```

2. Is your function tail recursive? Why or why not?

   yes, because recursive call(s) is only in tail pos.

Higher-order Functions (3 pts)

In Scala, functions are values, and can be passed to other functions.

1. What is the type of the following Scala expression? `((_:Int) + 123)` Int ⇒ Int

2. An anonymous identity function can be written in Scala as `(x => x)`. Call the following function with an anonymous function parameter in order to make it produce a result of 234. For example, `myFunction(x => x)` returns 201, so the identity function doesn't quite work.
   `def myFunction(f : Int=>Int) : Int = { f(200)+1 }`

   myFunction(x ⇒ x + 33)

▶ Problem 2 – Language Syntax (17 pts)

This problem explores programming language *syntax*.

EBNF Grammars (7 pts)

Consider the following grammar:

$$
\begin{array}{rcl}
A & \longrightarrow & A\,x \\
 & | & A\,y\,A \\
 & | & x
\end{array}
$$

The strings "xxyx" and "xx" are contained in the above grammar's language.

1. Write three other strings contained in the language
   xxyxx   xyx   x
2. Describe what an arbitrary string in this language looks like (either use a regular expression, or give a brief written description).
   Groups of x's separated by y's

Now consider the following grammar:

$$
\begin{array}{rcl}
L & \longrightarrow & !\,B\,.\,L \\
 & | & (\,L\,L\,) \\
 & | & B \\
B & \longrightarrow & a\,|\,b\,|\,c\,|\,\cdots
\end{array}
$$

The strings "!x.x" and "!y.!x.y" are contained in the above grammar's language.

1. Write three other strings contained in the language (do not use strings from the below list)
   a   (!x.x !y.y)   etc. . .
2. Indicate which of the following are contained in the language:

`!x.y`	`!x.1`	`!x.x+x`	`(a a)`	`(a a a)`	`!x.!f.(f x)`	`(f !x.x)`

Associativity/Precedence (5 pts)

1. In standard infix notation (i.e. what most calculators use), which of the two operators +, ∗ has the higher precedence? How could the expression "1 + 2 * 3" be parenthesized to indicate this precedence? * higher   1+(2*3)
2. Write a grammar for parsing left-associative addition. For example, "1+2+3" should be parsed as $(1+2)+3$.
   Expr → Expr + Num
   | Num

"Let" Expressions (5 pts)

Write a grammar for "let" expressions. Here are some examples:

```
let x = 123 in x
let y = 2 in (let x = 3 in y)
```

Expr → let Var = Expr in Expr
| Num | (Expr)
| Expr + Expr
⋮

as long as they have these, thats good enough

▶ Problem 3 – Parsing (13 pts)

Consider the following BNF grammar:

$$
\begin{array}{lcl}
Expr & \longrightarrow & Expr + Expr \\
 & | & Expr * Expr \\
 & | & (\, Expr\,) \\
 & | & Num \\
Num & \longrightarrow & 0 \,|\, 1 \,|\, 2 \,|\, \cdots
\end{array}
$$

Lexing (2 pts)

Before constructing a parser based on the above grammar, we will need to *tokenize* the input string. That is, we need to convert the input string into a sequence of strings that correspond to the terminal symbols in the grammar. This procedure is called a *lexer*.

1. Based on the above grammar, what are the possible tokens?

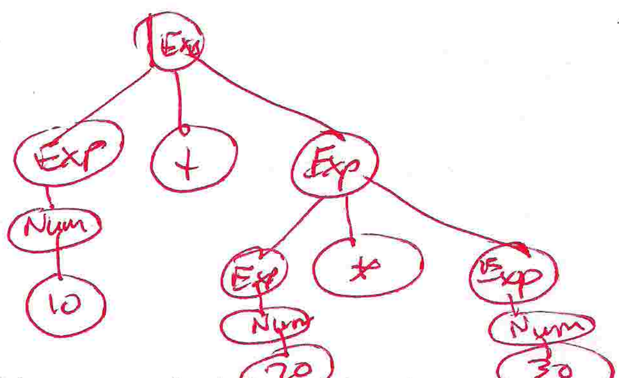
   + * ( ) <n> where this is an integer

2. Write the list of tokens for the input string `"(123 * 234) + (13 * 14)"`

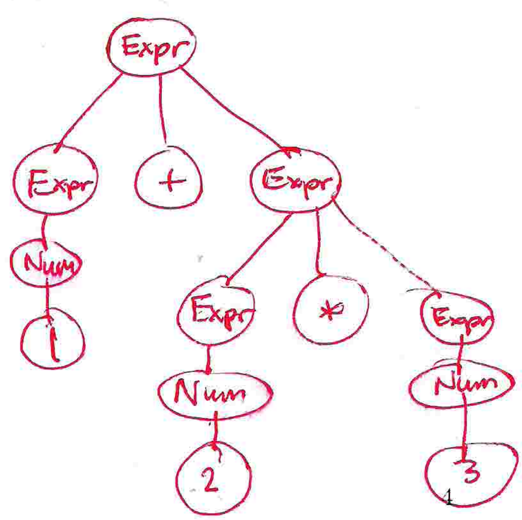
(	123	*	)	+	(	13	*	14	)

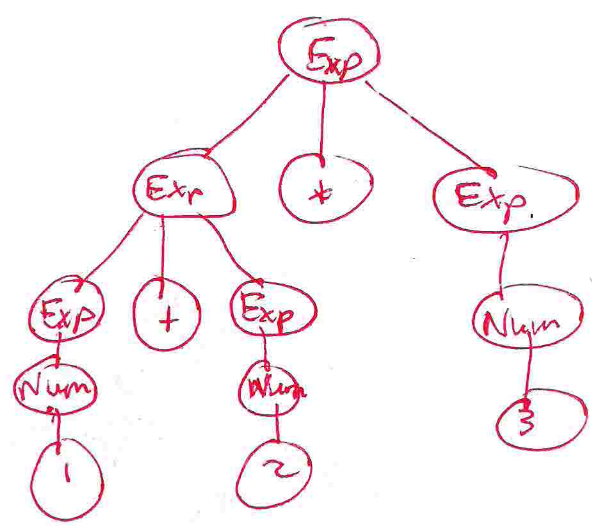
Parse Trees (6 pts)

1. Draw all possible parse trees for the input string `"10 + (20 * 30)"`



2. Draw all possible parse trees for the input string `"(1 + 2 * 3)"`





3. Is the above BNF grammar ambiguous? Why or why not? If so, show how to make it unambiguous.

YES! BECAUSE OF OPERATOR PRECEDENCE ISSUES.

Just ~~the~~ "layer" the grammar so the * forms a new "term" production.

4. The above grammar has left-recursion, since the *Expr* production has another *Expr* on the **left-hand side** of the + and * operators. Show how to eliminate the left recursion.

as long as they have something like this, it's fine

$$\begin{cases} Expr \rightarrow Term + Expr \\ \quad\quad\quad\ \mid Term \\ Term \rightarrow Num * Term \\ \quad\quad\quad\ \mid Num \end{cases}$$

Abstract Syntax Trees (5 pts)

From the parse tree, we can build an Abstract Syntax Tree (AST) that more closely models the program we are processing. Scala's *case classes* are a good way to build ASTs.

```
sealed abstract class Expr
case class BinOpExpr(op:String, e1:Expr, e2:Expr) extends Expr
case class NumExpr(n:Int) extends Expr
```

For the parse tree(s) of "10 + (20 * 30)" which you drew on the previous page, use the above data types to write the corresponding AST. The parameter op should contain the name of the binary operation, e.g. BinOpExpr("+", x, y).

BinOpExpr("+", NumExpr(10),
BinOpExpr("*", NumExpr(20), NumExpr(30)))

▶ Problem 4 – Language Semantics (20 pts)

Basics (5 pts)

1. Define *semantics* in the context of programming languages, and note how it differs from *syntax*.

Semantics = "meaning"
syntax = "structure"

2. Consider the following *let* expression. Which are the free variables? Which are the bound variables?

```
let x = 1+2 in
  let y = x+3 in
    x+y+z
```

z is free
x, y are bound

3. We have seen many different judgements in this class. Define *judgement*. Also, explain what each of the following two judgements mean:

we always have 0 as a nat. number

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{s(n) \in \mathbb{N}}$$

(if n is a nat. num., then so is s(n))

A judgement $\frac{x}{y}$ says "from x we can derive y" (or any number of similar ways of saying this).

Big-step Semantics (6 pts)

We have seen semantics which feature rules such as the following:

$$\frac{e_1 \Downarrow n_1 \in \mathbb{Z} \qquad e_2 \Downarrow n_2 \in \mathbb{Z}}{e_1 + e_2 \Downarrow n_1 + n_2}$$

1. Describe what the above big-step rule means.

If e_1, e_2 eval to ints. n_1, n_2, then $e_1 + e_2$ evals to the sum of the ints, $n_1 + n_2$

Small-step Semantics (6 pts)

We have also seen semantics containing rules such as these:

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v + e_2 \longrightarrow v + e_2'} \qquad \frac{v_1 = n_1 \in \mathbb{Z} \qquad v_2 = n_2 \in \mathbb{Z} \qquad v_3 = n_1 + n_2}{v_1 + v_2 \longrightarrow v_3}$$

1. Describe what the above small-step rules mean.

Reduce $e_1 + e_2$ in steps until we get to $v_1 + v_2$, then just add

2. What is the difference between big-step and small-step semantics? Make sure to explain why they are called "big"- and "small"-step.

Big-step reduce exprs to vals in <u>single</u> step.
Small-step reduce expr → expr → . . . → val.

Scope (3 pts)

1. What is static (lexical) scope?

We can find binding at <u>compile</u> <u>time</u>

2. What is dynamic scope?

~~Binding~~ We could have multiple options for where the variable is bound (determined by actually <u>running</u> the code).

▶ Problem 5 – Evaluation/Interpretation (15 pts)

In this problem, we investigate how to actually *run* a program represented as an AST.

Writing an Interpreter (5 pts)

1. What is the difference between a compiler and an interpreter? What are some advantages and disadvantages of both?

as long as they say something sensible about a compiler and an interp, it's fine

Compiler - static, produces code (code → code)
Interpreter - "dynamic", produces results (code → outputs)
Compiler - fast code. Interpreter - easy to build, etc.

2. Does type checking generally happen before or after evaluation of the program. Why?

Before. To rule out nonsensical cases before evaluation (simplifies evaluation, etc.)

JavaScript Fragment (5 pts)

In the class, we examined a fragment of JavaScript similar to the following, where *Var* represents identifiers (e.g. `foo`, `bar`, `myVar`, `x`, etc.), and *Num* represents numbers.

$$
\begin{array}{lcl}
Expr & \longrightarrow & \textbf{const}\ Var = Expr; Expr \\
& | & \textbf{if}\ Expr\ \textbf{then}\ Expr\ \textbf{else}\ Expr \\
& | & \textbf{function}\ Var(Var)\ Expr \\
& | & Var(Var) \\
& | & \{Expr\} \\
& | & Expr + Expr \\
& | & Expr < Expr \\
& | & Num
\end{array}
$$

1. How can we evaluate a variable declaration using an environment `env:Map[String,Expr]` which maps variable names to values?

just look it up by name in the map.

2. Can we evaluate a function declaration in a similar way? If so, how?

Similar, but Look up both the function name and param in the map.

Program Transformations (5 pts)

We would like to support multi-argument functions, but we have seen that it is much easier to define semantics for functions of a single variable. The process of converting a multi-argument function into "nested" single-argument functions is called *currying*.

1. Given a function with type $(A, B, C) \to D$, what is the type of the curried equivalent function?

$A \to B \to C \to D$

2. Re-write the following JavaScript function as a curried function *addCurried*:

☆1
```
function add(x,y,z) {
  return x+y+z;
}
```

☆2
```
function addC(x) {
  return function(y) {
    return function(z) {
      return x+y+z
    }
  }
}
```

☆3

3. Adding 1,2,3 using the uncurried original function can be done by calling `add(1,2,3)`. How can we do the same thing using the *curried* function?

   addC(1)(2)(3) ☆4

4. Describe a general procedure for currying all JavaScript functions in a program. That is, given a JavaScript AST corresponding to the above grammar, produce a new JavaScript AST where all functions have *at most one* argument.

   Just replace all function decls. like ☆1 with expanded ☆2. Replace all call sites ☆3 with re-written ☆4.

▶ Problem 6 – Types (20 pts)

Typing Rules (5 pts)

We have seen typing rules of the following form

$$\frac{e_1 : Int \qquad e_2 : Int}{e_1 + e_2 : Int}$$

1. What does it mean for an expression e to be *well-typed*?

There is a typing rule that gives it a type.

2. Write the typing rules for Scala `if` expressions and the < (less than) operator.

$$\frac{e_1 : Boolean \qquad e_2 : Int \qquad e_3 : Int}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : Int} \qquad \frac{e_1 : Int \qquad e_2 : Int}{e_1 < e_2 : Boolean}$$

Type Checking (5 pts)

1. Let's say we have built a language by defining small-step evaluation rules and a set of typing rules. What is a *stuck* expression? What does it mean for this language to be *type-safe*?

Stuck expr – we reach an expr e_s which is not a value, and can't be reduced further.
Type-safe
- Preservation – eval. rules preserve well-typedness
- Progress – we can't get stuck

2. What is the difference between type checking and type inference?

Checking – takes expr and type, returns yes/no
inference – takes expr, returns type

3. Use the typing rules from the above section to derive the type of this Scala expression
`if (1 < 2) 123 else (4+5)`

$$\frac{\dfrac{\dfrac{}{1 : Int} \qquad \dfrac{}{2 : Int}}{(1 < 2) : Boolean} \qquad \dfrac{}{123 : Int} \qquad \dfrac{\dfrac{}{4 : Int} \qquad \dfrac{}{5 : Int}}{(4+5) : Int}}{\text{if } (1<2)\ 123 \text{ else } (4+5) : Int}$$

Subtyping (8 pts)

1. Say we have a class `Person`, and we want a new subclass for representing drivers, i.e. $Driver <: Person$. Define such a subclass in Scala.

   class Driver extends Person {

   }

2. Consider the following code.

```
val a = new Person(); val b = new Driver()
def f(x:Person) = x.name; def g(y:Driver) = y.license
```

   After the above declarations, which of the following will produce a compile error? Circle all correct choices.

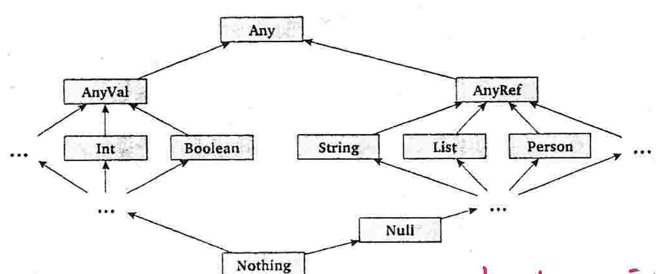
   `f(a)`
   `f(b)`
   `g(a)`
   `g(b)`

3. What is a partial order, and how is it different from a total order?

   partial order
   1. Reflexive $a \le a$
   2. Antisymmetric $a \le b$ and $b \le a \Rightarrow a = b$
   3. Transitive $a \le b$ and $b \le c \Rightarrow a \le c$

   Total order requires these plus $x \le y$ ~~[illegible]~~ or $y \le x$ for all x, y

4. The Subtyping relation <: can define a partial order over types, as seen in the following pictorial representation of Scala's type system (for example, $AnyRef <: Any$). What is special about the `Any` type in Scala? What is special about the `Nothing` type?



   Any - any type below it is a subtype of it
   Nothing - no subtypes (also no values) of Nothing, but ~~subtype~~ it's a subtype of any other type

Polymorphism (2 pts)

Scala allows you to "parameterize" functions and classes using a type variable(s). This is called polymorphism. For example, the polymorphic identity function is `def ident[A](x:A):A = x`, with `A` functioning as the type variable, and classes such as `List[A]` can be used to hold items of any type `A`. Think of a polymorphic class in Scala that allows you to avoid using `null`.

Option [A]

▶ Extra Credit – Various Topics (15 pts extra)

Peano Numbers (5 pts)

Inductive definitions are at the very heart of functional programming, and are generally very useful in computer science. As a simple example, we can define the natural numbers inductively, using only 0 and a function s of type $\mathbb{N} \to \mathbb{N}$.

1. How can we write the natural numbers from 0 to 3 in this way?

   0, s(0), s(s(0)), s(s(s(0)))

2. Define a multiplication function "*" for these inductively-defined natural numbers (hint: it will need to be recursive, and it can be defined in terms of the following addition function):

$$x + 0 = x \qquad x * 0 = 0$$
$$x + s(y) = s(x + y) \qquad x * s(y) = x + (x * y)$$

Function Subtyping (5 pts)

If A, B, C, D are types, there are two constraints that must be satisfied in order to conclude that $(A \to B) <: (C \to D)$. What are they? What does this mean in regards to the inputs and outputs of functions (hint: it is related to covariance/contravariance)?

C <: A and B <: D } covariant with respect to output types

contravariant with respect to input types

Monads (5 pts)

We have seen that a monad has three defining characteristics:

- For each base type t, we can construct the monadic type Mt.
- There is a *return* operation (sometimes called *unit*) with the following type:

$$t \to Mt$$

- There is a *bind* operation with the following type:

$$Mt_1 \to (t_1 \to Mt_2) \to Mt_2$$

We have also seen that Scala's `Option` can function as a monad, so let's call it monad M_o:

1. Using `Int` as a base type t, how would we write the monadic type $M_o t$ in Scala?

   Option[Int]

2. We can write a return operation like this in Scala

```
def myReturn(x : Int) = Some(x)
```

   Based on your above answer, how would you write the type of this function in Scala?

   Int => Option[Int]

3. Finally, write a simple bind operation in Scala by filling in the blanks:

```
def myBind(x : Option[Int],
           f : Int => Option[Int]) : Option[Int] = {
  x match {
    case(None) => None
    case(Some(y)) => f( y )
  }
}
```