Itskovich, Aleksandr (ai1221)

1) (20 pts) For the following forms apply β-reduction and α-conversion where appropriate. Explain each step.

a) (λx.x)(λx.x)

Step	Explanation
(λx.x)(λx.x) → (λx.x)(λy.y)	α-substitution
(λx.x)(λy.y) → (λy.y)	β-reduction (Identity)

b) (λx.x x) (λx. λy.x x)

Step	Explanation
(λx.x x) (λx. λy.x x) → (λx.x x) (λt. λy.t t)	α-substitution
(λx.x x) (λt. λy.t t) → (λt. λy.t t) (λt. λy.t t)	β-reduction
(λt. λy.t t) (λt. λy.t t) → λy.(λt. λy.t t) (λt. λy.t t)	β-reduction
…	Infinite Recursion

c) ((λx.(x y))(λz.z))

Step	Explanation
((λx.(x y))(λz.z)) → ((λz.z) y)	β-reduction
((λz.z) y) → y	β-reduction

d) (λz.z) (λy.y y) (λx.x a)

Step	Explanation
(λz.z) (λy.y y) (λx.x a) → (λy.y y) (λx.x a)	β-reduction
(λy.y y) (λx.x a) → (λx.x a) (λx.x a)	β-reduction

Itskovich, Aleksandr (ai1221)

(λx.x a) (λx.x a) → (λx.x a) a	β-reduction
(λx.x a) a → a a	β-reduction

e) (λz.z) (λz.z z) (λz.z y)

Step	Explanation
(λz.z) (λz.z z) (λz.z y) → (λz.z) (λt.t t) (λz.z y)	α-substitution
(λz.z) (λt.t t) (λz.z y) → (λz.z) (λt.t t) (λs.s y)	α-substitution
(λz.z) (λt.t t) (λs.s y) → (λt.t t) (λs.s y)	β-reduction
(λt.t t) (λs.s y) → (λs.s y) (λs.s y)	β-reduction
(λs.s y) (λs.s y) → (λs.s y) y	β-reduction
(λs.s y) y → y y	β-reduction

f) (λx.λy.x y y) (λa.a) b

Step	Explanation
(λx.λy.x y y) (λa.a) b → (λx.(λy.x y y)) (λa.a) b	Rewrite for additional clarity
(λx.(λy.x y y)) (λa.a) b → (λy.(λa.a) y y) b	β-reduction
(λy.(λa.a) y y) b → (λy.y y) b	β-reduction
(λa.a) b b → b b	β-reduction

g) (λx.x x) (λy.y x) z

Itskovich, Aleksandr (ai1221)

Step	Explanation
(λx.x x) (λy.y x) z → (λt.t t) (λy.y x) z	α-substitution
(λx.x x) (λy.y x) z → (λy.y x) (λy.y x) z	β-reduction
(λy.y x) (λy.y x) z → ((λy.y x) x) z	β-reduction
((λy.y x) x) z → x x z	β-reduction

h) (λx. (λy. (x y)) y) z

Step	Explanation
(λx. (λy. (x y)) y) z → (λx. (λt. (x t)) y) z	α-substitution
(λx. (λt. (x t)) y) z → (λx.x y) z	β-reduction
(λx.x y) z → z y	β-reduction

i) ((λx.x x) (λy.y)) (λy.y)

Step	Explanation
((λx.x x) (λy.y)) (λy.y) → ((λx.x x) (λy.y)) (λt.t)	α-substitution
((λx.x x) (λy.y)) (λt.t) → ((λy.y) (λy.y)) (λt.t)	β-reduction
(λy.y) (λy.y) (λt.t) → (λy.y) (λt.t)	β-reduction
(λy.y) (λt.t) → (λt.t)	β-reduction (Identity)

j) (((λx. λy.(x y))(λy.y)) w)

Step	Explanation

Itskovich, Aleksandr (ai1221)

(((λx. λy.(x y))(λy.y)) w) → (((λx. λy.(x y))(λt.t)) w)	α-substitution
(((λx. λy.(x y))(λt.t)) w) → ((λy.((λt.t) y)) w)	β-reduction
((λy.((λt.t) y)) w) → ((λt.t) w))	β-reduction
((λt.t) w)) → w	β-reduction (Identity)

2. (20 pts) Consider the following BNF grammar for a language with two infix operators @ and $ and terminals a b c. To receive credit, you must attach a detailed explanation for each of your answers that show the rationale for your answer.

S ::= A | S @ A

V ::= S | S $ V

A ::= a | b | c

a) What is the associativity of the @ operator: (a) left (b) right (c) neither

(a) left.

"S @ A", follows a left recursive rule, meaning that recursion is limited to the left-hand side. Also, because recursion is limited to only one side, the grammar is unambiguous (only one parse tree can be drawn). A left-recursive operator has left associativity.

b) What is the associativity of the $ operator: (a) left (b) right (c) neither

(b) right.

Similar to the answer above, but now the reasoning is flipped. "S $ V" follows a right-recursive rule, meaning that recursion is limited to the right side of the expression. As above, this makes the grammar unambiguous. At any rate, the expression follows a right-recursive rule, and the operator is therefore right associative.

c) Which operator has higher precedence: (a) @ (b) $ (c) neither

(a) @.

In a BNF structure, the later a rule appears in the grammar (i.e. the most nested it is in the grammar structure, the higher its precedence. In the above grammar, the uppermost reference is to V. V references S, and S references A, which references the terminal operators a, b and c. So, even though the expression "S ::= A | S @ A" comes before "V ::= S | S $ V", we still consider the former to be the more nested grammatical expression because it is referenced in the expression corresponding to V.

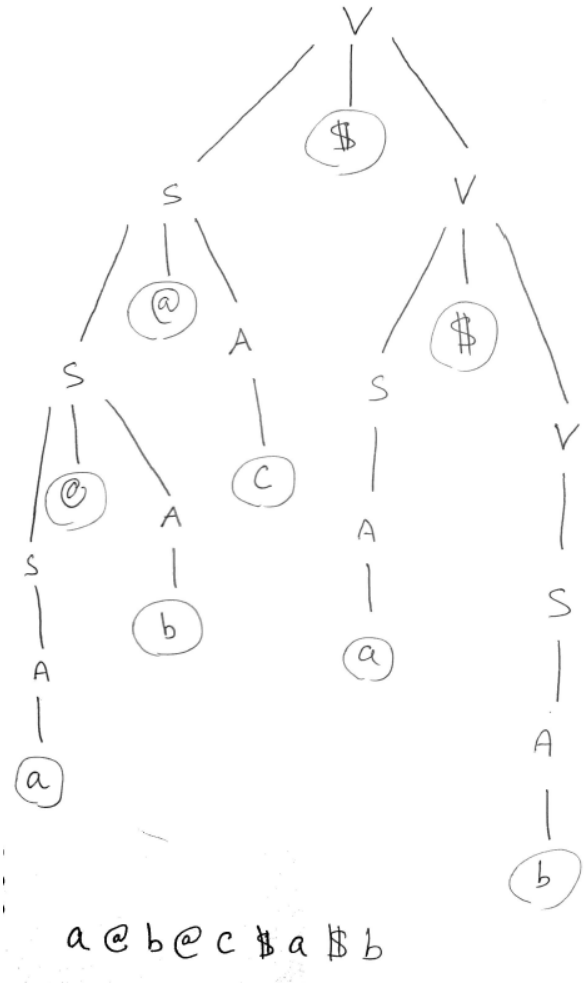
Therefore, V is higher in the parse tree and therefore has less precedence, S is lower in the parse tree and has more precedence, therefore the @ operator has greater precedence than the $ operator.

d) The grammar is: (a) left recursive (b) right recursive (c) both left and right recursive (d) neither left nor right recursive (c)

(c) both left and right recursive

As we saw in parts a) and b), this language has both a left-recursive rule (for S @ A) and a right-recursive rule (for S $ V). Hence the language is both left and right recursive.

e) Draw a parse tree for the following string: a @ b @ c $ a $ b



3. (15 pts) Write programs in Haskell, and Scheme that reverses a simple list of integers using recursion; specifically, in tail recursive form. Do not use higher order functions such as "reverse", comprehensive lists or functions from a library.

Haskell	reverse :: [a] -> [a] reverse xs = reverseHelper xs []

	reverseHelper :: [a] -> [a] -> [a] reverseHelper [] acc = acc reverseHelper (x:xs) acc = reverseHelper xs (x:acc)

Scheme	(define (reverse lst) (reverseHelper lst ()))
	
	(define (reverseHelper lst acc)
	  (cond
	  ((null? lst) acc)
	  (else
	  (reverseHelper (cdr lst) (cons (car lst) acc))
	)))

4. Given the following grammar:

a) (10 pts) Rewrite the BNF to give + precedence over * and force + to be right associative

<assign> -> <id> = <expr>

<id> -> A | B | C

<expr> -> <expr> * <term> | <term>

<term> -> <factor > + <term> | <factor>

<factor> -> (<expr>) | <id>

5. In your own words what are the key elements of a Functional Programming Language. Compare this with elements of an Imperative Programming Language. What operations/data types are available in each and how do they differ?

A Functional programming language, or languages that are implemented in a function style, have several distinct characteristics. The most important of these is the treatment of **functions as first-class citizens**. This means that function expressions can be assigned to variables, passed as parameters to other functions, or returned from functions. This allows programmers to work with **higher-order functions**, or functions that act on other functions. This allows for constructs like **closures** or **curried functions**, to name just two prominent examples. (A *closure* is a function that defines a state, and returns another function that now has a reference to that state. In JavaScript, for example, this can be a useful way of encapsulating a state instead of maintaining it as an accessible global variable. *Currying* refers to the partial application of a function with multiple parameters; a curried function binds the subset of parameters that were passed to it, and returns a function that takes the remaining number of parameters. This makes it easy to compose functions from other functions, and can be a vital tool in building up functional programs. In a language like Haskell, for example, all functions with more than one parameter are curried under the hood.)

Another important aspect of functional programming languages is that functions should be **pure functions** and have no **side-effects** in evaluation. A pure function is a function that always returns the

same output for the same input, akin to how functions work in mathematics. This makes the behavior of functions **referentially transparent** (meaning they can be replaced with their results without changes to the program) and easier to reason about. A **side-effect** is anything produced by the function during its evaluation which changes the **state** of the environment outside the function. Side-effects include doing things like printing to the console, writing out to a file, or modifying data by reference (for example, by using setters on an object in OOP). Side-effects make programs more difficult to follow, debug and reason about. They also break the mathematical definition of a function that functional languages seek to recreate. Of course, side-effects are sometimes necessary. After all, programs are created to affect the world around them, including things like writing to a file or printing to screen. Haskell, for example, solves this problem by segregating functions that perform side-effects from the rest of the purely-functional program in structures called monads. Other languages, like OCaml or Scala which mix functional programming with object-orientation, allow the programmer to create side-effects as they see fit. The concept of reducing side-effects is related to another important tenet of function programming known as **immutability**. Immutability is the idea that all constructs in a program are **stateless**, or are only created, never mutated. In object-oriented languages, for example, enforcing this principle implies never using setters. If an object's data needs to be altered in some way, a new object is created with the new data instead of modifying the existing object. Aside from being easier to reason about, programs that enforce immutability have other benefits, like being inherently **thread-safe**.

Functional programming languages also reason about **data structures** and **program evaluation** differently from imperative languages. For example, pure functional languages eschew the use of **loops**, such as for-loop or while-loop, that are facets of imperative languages. Imperative languages say: "do this first, then this second, finally do this third". They define a specific execution order to achieve some effect. Functional programs are created differently; they are essentially function compositions where one function calls another function until the expression is completely simplified. There is no concept of a sequence of steps in which code runs (Haskell introduces this concept in its monads); therefore the most natural approach to evaluation functional programs is through the use of **recursion**. Functions merely call other functions, or themselves, until the computation is complete.

In terms of data structures, imperative languages define them concretely. If a list is declared with ten elements, the appropriate amount of memory is reserved for each element. If a new element needs to be added, the entire list is copied over, or in the case of a linked-list, a new pointer and memory cell are added. But fundamentally, a data-structure is a defined memory-object. In contrast, functional languages define data structures *functionally* or through their behavior. For example, if a functional language wanted to define a set {2, 4, 6, 8}, it would define it as a function that returns *true* for any number divisible by 2 that is greater than 0 or less than 10. An imperative language would just create an object with 4 memory locations and store a number in each location. In other words, functional programs store *behaviors*, imperative programs store *data*. One interesting implication of the functional approach is that it is possible to create **infinite data structures**: because a behavior is stored, not the actual data, we don't need to evaluate anything and save it in memory until the result is actually needed. This leads to the powerful idea of **lazy evaluation**, which is the ability to only evaluate an expression only when a result is needed. So, for example, if an infinite list of odd numbers is defined, it is possible to ask for the first hundred values and the data structure will only evaluate the first hundred results, nothing else. This concept is, of course, not available in imperative languages.

In summary, functional programming languages support **functions as first-class citizens** and support **higher-order functions**. Imperative language don't support functions as first-class citizens. They use **recursion** instead of **loops** when evaluation programs. They don't evaluation programs **sequentially** but as the **simplification of function compositions.** Imperative languages do not. Functional programming languages enforce **pure functions** and programming without **side-effects**, both of which are possible in imperative languages but are not part of the design. Finally, functional languages reason about data structures differently from imperative languages, encoding them as behaviors rather than fixed data, and thereby supporting **infinite collections** and **lazy evaluations**. Both of these are absent from imperative languages.

6) (10 pts) Given the following definition:

```
fibs :: [Integer]

fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

a) What is fibs? What does it do?

Fibs is a list of integers. More accurately, it is an **infinite list of integers** that only calculates the integers when they are needed. I.e., it is **lazily evaluated**. This means that the sequence will not compute the n^{th} element until it is specifically requested by the programmer.

More specifically, the first two elements of the sequence are 1 and 1. The rest of the sequence is determined by adding together (zipWith (+)) the sequence itself with the tail of the sequence itself (i.e. 1+1, 1 + 2, 2 + 3, 3 + 5, …) etc. This ultimately produces elements of the Fibonacci sequence, as shown below:

That is:

fibs = [1, 1, 2, 3, 5, 8 …]

tail fibs = [1, 2, 3, 5, 8, …]

zipWith (+) = [2, 3, 5, 13, …]

b) Explain what the following code returns and why: take 5 fibs

The following code returns the first 5 elements of the Fibonacci sequence: [1, 1, 2, 3, 5]. As mentioned above, the definition of the Fibonacci sequence is defined as the operation by which Fibonacci sequence elements are generated, not the sequence itself. So the expression "take 5 fibs" is essentially saying: "using the Fibonacci generator you defined, generate the first five elements of the Fibonacci sequence."

Also, the tail of the list is always the unevaluated expression that generates the next elements of the sequence. When we calculate the first 5 elements, we store the result as concrete data, along with the expression used to produce the next elements of the sequence. Such is the magic of lazy evaluation.

7) (5 pts) In the following diagram state the scheme lists that are indicated in the box diagrams?

((a) b (c d) e)