**Mid-Term Solutions**

Question 1:

a) (λx.x)( λx.x )

=> λx.x -- β-reduction where x = ( λx.x )

b) (λx.x x) (λx. λy.x x)

=> (λx.(x x)) (λx.(λy.x x)) -- Application is left associative

=> ((λx.(λy.x x)) (λx.(λy.x x))) -- β-reduction where x = (λx.(λy.x x))

=> λy. (λx.(λy.x x)) ((λx.(λy.x x)) -- β-reduction where x = (λx.(λy.x x))

=> (λx.(λy.x x))

c) ((λx.(x y))(λz.z))

=> ((λz.z)y) -- β-reduction where x = (λz.z)

=> y -- β-reduction where z = y

d) (λz.z) (λy.y y) (λx.x a)

=> ((λz.z) (λy.y y)) (λx.x a) --application is left associative

=> (λy.y y) (λx.x a) -- β-reduction where z = (λy.y y)

=> (λx.x a) (λx.x a) -- β-reduction where y = (λx.x a)

=> ( λx.x a) a -- β-reduction where x = (λx.x a)

=> (a a) -- β-reduction where x = a

e) (λz.z) (λz.z z) (λz.z y)

=> ((λz.z) (λz.z z)) (λz.z y) --application is left associative

=> ((λz.z) (λc.c c)) (λz.z y) -- α-substitution, z in expression (λz.z z) substituted to c

=> (λc.c c) (λz.z y) -- β-reduction where z=(λc.c c)

=> (λc.c c) (λk.k y) -- α-substitution, z in expression (λz.z y) substituted to k

=> (λk.k y) (λk.k y) -- β-reduction where c= (λk.k y)

=> ((λk.k y)y) -- β-reduction where k= (λk.k y)

=> (y y) -- β-reduction where k=yf) (λx.λy.x y y) (λa.a) b

f) (λx.λy.x y y) (λa.a) b

(λy.( λa.a) y y)b -- β-reduction where x=( λa.a)

=> ( λa.a)b b -- β-reduction where y=b

=> (b b) -- β-reduction where a=b b

g) (λx.x x) (λy.y x) z

=> ((λx.x x) (λy.y c)) z --- α-substitution, x in expression (λy.y x) substituted to c

=> ((λy.y c) (λy.y c)) z -- β-reduction where x=(λy.y c)

=> ((λy.y c)c) z -- β-reduction where y=(λy.y c)

=> (cc)z -- β-reduction where y=c

h) (λx. (λy. (x y)) y) z

=> (λx. (λy. (x y)) t) z --- α-substitution, right most y in expression (λx. (λy. (x y)) y) substituted to t as it is not bound

=> (λy. (z y)) t --- β-reduction where x=z

=> z t --- β-reduction where y=t

i) ((λx.x x) (λy.y)) (λy.y)

* ((λy.y) (λy.y)) (λy.y) --- β-reduction where x=(λy.y)
* ((λy.y) (λy.y)) --- β-reduction where y=(λy.y)
* ((λy.y) (λt.t)) --- α-substitution, right most y in expression (λy.y) substitute to t
* (λt.t) --- β-reduction where y=(λt.t)

j) (((λx. λy.(x y))(λy.y)) w)

* (λy.((λy.y) y))w --- β-reduction where x=(λy.y)
* (λy.(λt.t)y)w --- α-substitution, right most y in expression (λy.((λy.y) y))w substituted to t
* (λt.t)w --- β-reduction where y=w
* w --- β-reduction where t=w

Question 2:

S::= A | S @ A

V ::= S | S $ V

A ::= a | b | c

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

Ans: (a) @ is left associative as the recursion term S is appearing on the beginning of the RHS term. Left recursion specifies left associativity

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

Ans: (b) $ is right associative as the recursion term V appears on the end of the RHS term. Right Recursion specifies right associativity.

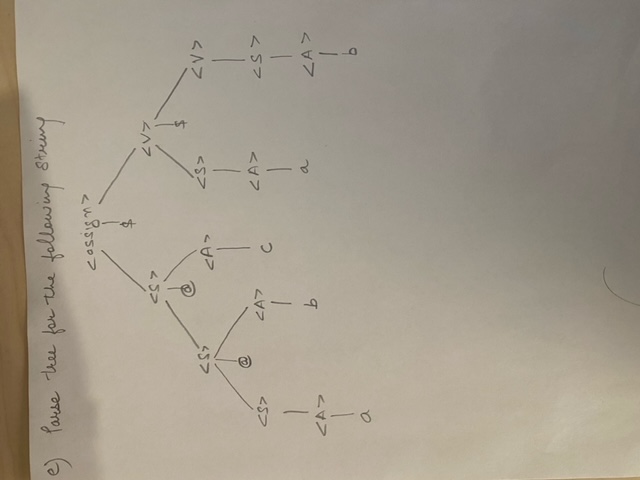
1. Which operator has higher precedence: (a) @ (b) $ (c) neither

Ans: (a) @ has higher precedence than $ as it appears lower in the parse tree (refer image for answer 5)

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

Ans: (c ) The grammar is both left and right recursive as the recursion term appears both at the beginning and end of RHS terms in the grammar sentences (S appears at the beginning while V appears at the end)

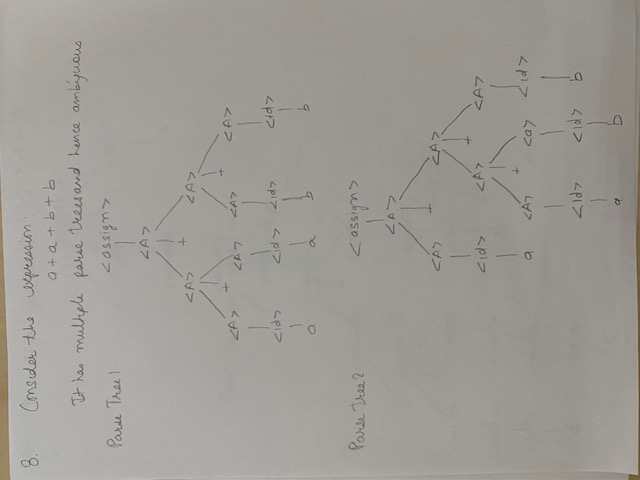
e) Draw a parse tree for the following string: a @ b @ c $ a $ b



3) Code for reversing a list (attached in separate files):

4) The grammar is ambiguous as it allows for multiple parse trees.

For the expression a+a+b+b following two parse tree can be derived.



5a)

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

<id> => A | B | C

<expr> => <expr> \* <term>

| <term>

<term> => <factor> + <term> …. “+” has higher precedence as it appears lower in parse tree and is right

Associative as the recursion term appears on the right hand side

| <factor>

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

5b)

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

<id> => A | B | C

<expr> => <expr> + <term>

| <term>

<term> => <term> \* <factor>

| <factor>

<factor> => ( <expr> )

| <id>| <id>++|<id>-- --- added unary operators

6) In your own words what are the key elements of a Functional Programming Language. Compare this with an Imperative Programming Language. What operations are available in Programming Language and how do they differ from each other.

Functional Programming Language has the following key features:

1. No State. No Memory Calls

Functional programming(FP) consists of functions (approximating mathematical functions) that does not contain state. Provided a consistent set of arguments, the function always returns the same output (no side effects). The output is not stored in the memory as objects and therefore, there is no requirements of memory calls like imperative programming.

1. Low importance of Order of Execution

In Functional programming, the programs are written as an accumulation of functions containing a set of statements. All functions work independent of each other and produce same result regardless of order of execution.

1. Functions are first class citizens

Functions are first class objects that can be bound to variables (similar to strings and integers), passed to other functions, and returned from another function.

1. Modular Programming

In FP Languages, we need to write smaller and independent units, called Pure Functions to support Stateless Programming model.

1. Higher Order Functions

Functional Programming Languages allows for defining higher order function that can be passed both data and a function that acts on that data.

1. Recursion - statelessness rules out the use of imperative constructs like for, while, etc. Instead, recursion is used for iteration.

|  |  |
| --- | --- |
| **Functional Programming** | **Imperative Programming** |
| Functions are Stateless | Operations are performed on objects stored in memory. Assignments statements are used to store the evaluations of complex expressions. These objects in memory are mutable through the course of the program. |
| No Memory Calls | Memory Calls |
| Uses Immutable data | Uses Mutable data |
| Order of execution is not critical. | Order of execution is very important is it is the primary determinant of object state. |
| Functions are akin to mathematical functions and have No-Side Effects | Methods with Side Effects |
| Functions are first-class citizens | Objects are first-class citizens |
| Primary Manipulation Unit is “Function” | Primary Manipulation Unit is Objects |
| Flow is Controlled using Function calls and/or Function Calls with Recursion | Flow is controlled using Loops and Conditional Statements |
| “Recursion” is used for iteration | “Loop” concept (e.g.for, while) is used for iteration |