JavaScript Concepts

What does it mean for a language to have first class functions? (functions are values) ➤ can be declared within any scope ➤ can be passed as arguments to a function ➤ can be returned as result of function call

Functions as args • Original reason: simple way to do event handling ➤ E.g., onclick(function() { alert("button clicked!"); }) • Still true today. But many other reasons, including: ➤ performance: asynchronous callbacks ➤ expressiveness: filter, map-reduce, etc

everything in JavaScript is treated as objects

"`use strict`" mode in the first line to prevent you from writing bad codes

var has function scope or global scope, var declared in a function will have function scope, otherwise global scope
let and const both have block scope, but if not in block, then global scope.

A higher-order function is a function that takes one or more functions as arguments, returns a function as its result, or both.

Regardless of where variables are actually declared and initialized, their declarations (but not initializations) are hoisted to the top of the nearest (function or global) scope;thus you may
get an var is undefined for initialization after use

Anonymous Function It is a function that does not have any name associated with it. Normally we use the function keyword before the function name to define a function in JavaScript, however, in anonymous functions in JavaScript, we use only the function keyword without the function name.
function(){} or ()=>{}

CFG and Lambda Calculus

Syntax (grammar) ➤ The symbols used to write a program ➤ E.g., (x + y) is a grammatical expression
Semantics ➤ The actions that occur when a program is executed PL implementation: Syntax -> Semantics
Expressions: e ::= x | λx.e | e1 e2
➤ Functions or λ abstractions: λx.e ➤ This is the same as x => e in JavaScript!
➤ Function application: e1 e2 ➤ This is the same as e1 (e2) in JavaScript!
Application has higher precedence than abstraction
$λx+y.3$: invalid
Function application is left associative
e1 e2 e3 ≝ (e1 e2) e3
λx.λy.λz.e ≝ λx.(λy.(λz.e))
λx.λy.λz.e ≝ λxyz.e
Normal Form is a form that can not be reduced anymore (can reach confluence)
F V (x) = x
F V (λx.e) = F V (e) \ {x}
F V (e1 e2) = F V (e1) ∪ F V (e2)
x is bound in $\lambda x.e$, e is free

Church-Rosser Theorem: "If you reduce to a normal form, it doesn't matter what order you do the reductions." This is known as confluence.
JavaScript's evaluation strategy is call-by-value (ish) : Reduce function, then reduce args, then apply
Haskell's evaluation strategy is call-by-name:evaluate the leftmost function as much as possible then pass in the argument value
• α-renaming or α-conversion
➤ λx.e = λy.e[x:=y] where y∉FV(e)
• β-reduction ➤ (λx.e1) e2 = e1 [x:=e2]
• η-conversion ➤ λx.(e x) = e where x∉FV(e)

Scope & Closure

We create a new environment for everytime we enter either a new block scope or function scope

Haskell

• In haskell, x = 2 is a symbol binding which is immutable.
• Variables in haskell are order independent, so you can call a variable before it is declared.
• Haskell does not do implicit type conversion

```
gt = \x y -> x > y -- \xy.(x > y)
 -- this is a lambda abstraction, equivalent to gt x y = x > y
```

	Compile-time	Run-time checks	Hardware
Pro	No runtime overhead	Permissive	Super fast
Con	Over approximates	Runtime overhead	Catch bugs late

• Def: A language is type safe if no program is allowed to violate its type distinctions
• Haskell and JS are type safe. C is not.
• What language features make it hard to guarantee type safety? A: raw pointer/memory access, casts, etc.
• add :: Integer -> Integer -> Integer
• -> is right associative, thus add :: Integer -> (Integer -> Integer)

```
    reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
               = reverse xs ++ (x:[])
               = (++) (reverse xs) (x:[])
               = ((++) (reverse xs)) (x:[])
               = ((++) (reverse xs)) (((:) x) [])
```

From this parse tree we then have these constraints:

(1) $\tau_0 = [\tau_1] \to \tau_9$

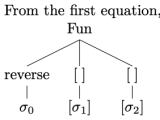
(2) $\tau_2 = \tau_5 \to \tau_7$

(3) $\tau_0 = [\tau_1] \to \tau_5$

(4) $\tau_3 = \tau_1 \to \tau_6$

Answer:

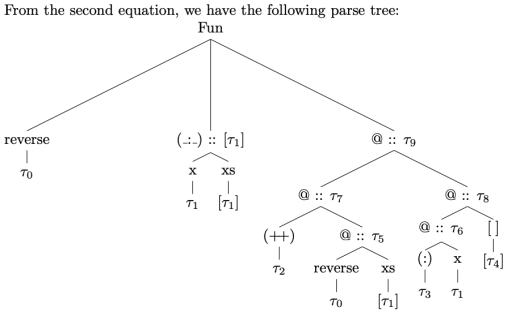
From the first equation, we have the following parse tree:



From this parse tree we then have this single constraint:

(0) $\sigma_0 = [\sigma_1] \to [\sigma_2]$

From the second equation, we have the following parse tree:



Let's break down each expression you've given and explain or compute its result. The expressions T and F represent the boolean values true and false in the lambda calculus, using Church encoding:

- T = λa.λb.a represents true, returning the first argument.
- F = λa.λb.b represents false, returning the second argument.

Next, you're asking for the definitions of AND and OR, as well as the results of applying these operations to T and F. Let's define AND and OR first:

- AND operation in lambda calculus can be defined as λp.λq.p q p, which means it takes two arguments (p and q) and returns q if p is true (T), otherwise returns p itself (which would be F).
- OR operation can be defined as λp.λq.p p q, which takes two arguments (p and q) and returns p if p is true (T), otherwise returns q.

Now, let's evaluate each expression using these definitions:

AND T T should return T, because if both arguments are true, the AND operation is true.
AND T F should return F, because if any argument is false, the AND operation is false.
OR T F should return T, because if any argument is true, the OR operation is true.
OR F F should return F, because if both arguments are false, the OR operation is false.

Finally, you've given a NOT operation defined as λ F T and asked for the result of NOT (AND T F):

(5) $\tau_3 = \tau_1 \to [\tau_1] \to [\tau_1]$

(6) $\tau_6 = [\tau_4] \to \tau_8$

(7) $\tau_7 = \tau_8 \to \tau_9$

Unifying (1) and (3):

(8) $\tau_9 = \tau_5$

Unifying (4) and (5):

(9) $\tau_6 = [\tau_1] \to [\tau_1]$

Unifying (9) and (6):

(10) $\tau_4 = \tau_1$

(11) $\tau_8 = [\tau_1]$

Unifying (11) and (7):

(12) $\tau_7 = [\tau_1] \to \tau_9$

Unifying (2) and (12) and (8):

(13) $\tau_2 = \tau_9 \to [\tau_1] \to \tau_9$

Also we know the type of node $\tau_2 =$ `(++)`

(14) $\tau_2 = [\tau_{10}] \to [\tau_{10}] \to [\tau_{10}]$

Unifying (13) with (14)

(15) $\tau_{10} = \tau_1$

(16) $\tau_9 = [\tau_{10}]$

(17) $\tau_9 = [\tau_1]$

Substituting (17) into (1)

(18) $\tau_0 = [\tau_1] \to [\tau_1]$

Since $\sigma_0 = \tau_0$, unifying (0) and (18):

(19) $\tau_1 = \sigma_1$

(20) $\tau_1 = \sigma_2$

Thus, reverse :: $\tau_0 = [\tau_1] \to [\tau_1]$

- The NOT operation can be more accurately defined as λp.p F T, which switches T to F and vice versa.
- Evaluating NOT (AND T F) would first require computing AND T F, which we've established is F. Applying NOT to F would then yield T, because NOT flips the value.

Let's compute these lambda expressions to confirm the operations.

The computations of the lambda calculus expressions you provided yield the following results, confirming the logical operations as expected:

AND T T returns True, indicating that the logical AND of true and true is true.
AND T F returns False, showing that the logical AND of true and false is false.
OR T F returns True, which means the logical OR of true and false is true.
OR F F returns False, indicating that the logical OR of false and false is false.
NOT (AND T F) returns True, demonstrating that the logical NOT of the AND operation between true and false (which is false) is true.