**Syntax**

Infinite set **Vars**, of variables:

$u, v, x, y, z, ..., x_1, x_2, ...$

Well-formed $\lambda$ -calculus expressions LExpr is the smallest set of strings matching the following syntax:

$M, N, ...elementofLExpr ::= v\ |(\lambda xM)\ |(MN)$

where x is a random variable and M is an arbitray lambda expression.

a $\lambda$ -Calculus expression is either a variable (v), an abstraction of a variable from an expression ($(\lambda xM)$) or an applciation of one expression to another ((M N) )

**Free/Bound Variables**

A variable occurenc is free in an expression if it is not mentioend in an enclosing abtraction. x ($\lambda$ y• (yz)) The z is free because it is not mentioned in the abstraction (which is $\lambda$ y) The y is bound

A variable can be both free and bound in the same expression

(x($\lambda$ x•(xy))

The first x is a global free variable outside the abstraction The second x is in (xy) is bound

**$\lambda$ -Calculus Moves**

$\alpha$ -renaming M -¿ M' $\beta$ -reduction $\backslash n$ //missed //here

$\alpha$ **-Renaming** ($\lambda$ x • ($\lambda$ y •(x y))) -¿ ($\lambda$ u • $\lambda$ v • (u v))) ($\lambda$ x • (x y)) -/¿ ($\lambda$ y • (y y)) formerly free y has been "captured"

Leaves the meaning of a term unchanged. Same as changing the name of a local variable in a program.

**Substitution**

Subsituting an expression N for all free occurences of X, in another expression M, written M[N/x]

(x ($\lambda$ y • (z y))) [ ($\lambda$ u • u) / z] -¿ (x($\lambda$y •(($\lambda$ u • u)y)))

Y is not free wnyehere so substituing in for that would do nothing.

**Careful Substitution**

When doing general substitution M[N/x], we need to avoid variable capture of free variable sin N by bindings in M:

(x($\lambda$ y•(z y ))) -/¿ (x ($\lambda$ y • ((y x) y)))

If N has free variables which are oing to be inside an ibastraction on those variables in M, then we need to $\alpha$ - rename the abstractions to something else first and then substitute. //here The Golden Rule: //here

**$\beta$ -Reduction**

($\lambda$ x•M) N -¿ M[N/x]

We define an eexpression of the form ($\lambda$ x•M) N as a $\beta$ -redex (reducible expression)

**How this applies to functions**

$f(x) = 2x + 1f(42) = substitute42forxinther.h.s(2x + 1)[42/x]2x42 + 1$

This is basically $\beta$ -reduction

**Normal Form** An expression with no redexes.

Aim: reduce an expression to its normal form (if it exists) i.e. play the game until you cant make anymore moves

((($\lambda$ x • ($\lambda$ y • (y x))) u) v) -¿ (($\lambda$ y • (y u)) v) -¿ (v u) A normal form

(($\lambda$ x • (x x)) u) -¿ ( u u) A normal form

Not all expressions have normal form.

**Normal Form(III)** //here innermost and outermost

**n reduction (eta?))**

**$\lambda$ - Calculus and Computability**

We can use it to encode booleans, numebrs and finctions $\lambda$ - Calculus is Turing-complete

**How Important is Reduction Order?**

The Church-Rosser theorem states: If we can reduce M to $N_1$, by one set of redex choices, and to $N_2$ by another, then there always exists a third value R, to which both $N_1$ and $N_2$ can be reduced.

This third value R may be different but could also be one of $N_1$ or $N_2$

**Normal Forms are Unique**

We reduce M to $N_1$, where $N_1$ is a normal form using some reduction sequence. We reduce M to $N_2$, where $N_2$ is also a normal form using some reduction sequence.

By the Diamond Lemma, there exists an R to which both $N_1$ and $N_2$ can be reduced. But $N_1$ and $N_2$ are normal forms so cant be reduced further. Therefore $N_1$ = R = $N_2$

**Reduction Order**

If we have a chocie of redexes, which should we reduce first?

Always the leftmost-outermost one (Normal Order Reduction)

Haskell uses the Graph Execution model (a variant of NOR) which gives rise to Lazy Evaluation.

**Lambda Abstraction in Haskell**

$\lambda$ becomes ”̈(x -¿ 2 * x + 1) 42

Haskell's default behaviour is to reduce an expression to a notion of a partial/incomplete form. Weak Head Normal Form (WHNF)

A Haskell expression is WHNF when its top-level is either:

A data constructor applied to some or all of its values

or

A function that has not been applied to all of its arguments

Largely implemneted by the way pattern-matching is

**Strictness**

Strict functions always evaluate all their arguments to Normal Form and then produce their own results as a Normal Form.

**Example: isOddl** //here