Science fonctionnelle

Gianni Mocellin

Straco
www.straco.ch
16.11.2021, 05h00

Introduction *5*

Le noyau *5*

Distributivity **5**
Function[body] 5
body& 6

Orderless **6**

Les expressions **6**

Les atomes **10**

Le calcul **11**
Sum[énumérateur, {énumérateur, 1, 5}] **12**
Product[énumérateur, {énumérateur,1, 5}] **12**

L'évaluateur **13**

Les règles **15**
nom := expression 15
nom[ ] := expression 15
nom1 /: nom2[..., nom1[...], ...] := expression 16
nom1 /: nom2[..., nom1, ...] := expression 16
nom1[nom2[ ], ...] := expression 16
nom1 /: 16
nom[...] [...] := expression 16

Le comparateur **16**

Les listes *17*
idée = List[a,b,c] 17

Table **17**
Table[expr,n] 17
Table[expression, énumérateur] 17
Table[expression, {énumérateur, initiale, finale}] 17
Table[f[n],{n,20}] 18
TableForm[] 18

Array **18**
Array[f, n] 18
Array[f, n, n0] 18
Array[f, n, {a, b}] 18
Array[expression, n] 18
Array[f,20] 19
Array [$list_1$,list2] 19

SparseArray **19**
SparseArray[list] 19
Normal[SparseArray[…]] 19
SparseArray[data,{d_1,d_2,…}] 19
SparseArray[data,dims,val] 20
SparseArray[{{i_,i_}->1},{d,d}] 20

Applying an action *20*

Map **20**
Map[f,expr] 20
f/@expr 20
Map[f] 20
Map (/@) 20
Map[action, idea] (/@) 20
Map[action, idea, level] 22

MapAt[action, idea, position] 22

Apply **23**
Apply (@@) 23

Opérations ***24***

Injection **24**
Inner[fonction1, argument1, argument2, fonction2] 24
Inner[f,list$_1$,list$_2$,g] 24

Ejection **24**
Outer[f, {a, b}, {c, d}] 24

CartesianProduct **24**
CartesianProduct[l_1,l_2] 25

Kronecker **25**
KroneckerProduct[m_1,m_2,…] 25
KroneckerDelta[n_1,n_2,…] 25

Les ensembles ***25***

Subsets **25**
Subsets[*list*] 25

Les unités et quantités ***25***

Entrées **25**
Quantity 26
QuantityArray 26
QuantityDistribution 26

Composants **26**
QuantityMagnitude 26
QuantityUnit 26
UnitDimensions 26
QuantityQ 26
KnownUnitQ 26

Affichage **26**
QuantityForm 26
TargetUnits 26

Conversions **26**
$UnitSystem 27
UnitConvert 27
CompatibleUnitQ 27
CommonUnits 27
UnitSimplify 27

Quantités avec incertitude **27**
Around 27
VectorAround 27
MeanAround 27
AroundReplace 27

Quantités physiques **27**
QuantityVariable 27
QuantityVariableIdentifier 27
QuantityVariablePhysicalQuantity 27

Analyse naturelle **28**
DimensionalCombinations 28
Combinaison possible de quantités 28

QuantityVariableDimensions 28
QuantityVariableCanonicalUnit 28
NonDimensionalizationTransform 28
Unités mixtes 28
MixedUnit 28
MixedMagnitude 28
Unités monétaires 28
CurrencyConvert 28
DateUnit 28
InflationAdjust 28
Unités indépendantes 28
IndependentUnit 28
IndependentPhysicalQuantity 28
IndependentUnitDimension 28
Les arbres 29
CEOing 30

Introduction

Jean et Gianni devant une machine.

JEAN

Qu'est-ce que c'est que cette machine?

GIANNI

C'est une machine à science.

JEAN

Comment fonctionne-t-elle?

GIANNI

On lui donne une idée,

elle l'évalue par rapport à ce qu'elle sait, et,

elle retourne un résultat.

Le noyau

Distributivity

Two binary operations, usually multiplication and addition, in which the operation of an element on a sum of elements is equal to the sum of the individual operation of that element on each element

The first operation is distributive over the second operation.

Function[body]

or

body&

Is a pure (or "anonymous") function.

Function is analogous to λ in LISP or formal logic.

The formal parameters are # (or #1), #2, etc.

When Function[body] or body& is applied to a set of arguments, # (or #1) is replaced by the first argument, #2 by the second, and so on. #0 is replaced by the function itself.

Orderless

Is an attribute that can be assigned to a symbol f to indicate that the elements e_i in expressions of the form $f[e_1,e_2,\ldots]$ should automatically be sorted into canonical order.

This property is accounted for in pattern matching.

Les expressions

JEAN

Comment est-ce qu'on lui donne des idées?

GIANNI

Sous forme d'expressions.

Pour la machine, les idées, scientifiques ou non, s'expriment toutes sous la forme suivante:

expression1 [expression2, expression3, ...]

On peut distinguer la première expression devant les crochets en la nommant

"Tête"

et en nommant le reste entre crochets

"Corps".

Pour la machine, les idées sont donc toujours constituées de deux parties fondamentales: une tête et un corps, ce dernier étant délimité par les deux crochets:

Tête[Corps]

La tête peut elle-même être une expression et le corps peut être composé d'une ou plusieurs expressions.

Le corps peut éventuellement être vide auquel cas on a l'idée suivante:

Tête[ ]

JEAN

Tu peux me donner un exemple concrêt?

GIANNI

L'expression

fonction1[a, b] [ ] [fonction2[1], fonction3[x, y]]

a pour tête

fonction1[a, b] [ ]

et pour corps

[fonction2[1], fonction3[x, y]]

La tête est constituée elle-même expression ayant pour tête

fonction1[a, b]

et pour corps

[ ]

un corps vide en l'occurrence.

Le corps de l'expression de base est quant à lui le suivant

[fonction2[1], fonction3[x, y]]

et il contient lui-même deux éléments qui sont eux-même des expressions, soit

fonction2[1]

et

fonction3[x,y]

et ainsi de suite.

La tête d'une expression peut être extraite par la machine en utilisant la fonction

Head[ ]

et l'un des éléments du corps par la fonction

Part[ ]

L'idée de "*liste*", par exemple, est représentée dans la machine de la manière suivante

List[ ]

En fait, le mot "List" représente une fonction de la machine, une commande qu'on peut donner à la machine.

Si je lui donne l'expression

List[ ]

elle va l'évaluer et me retourner un résultat, en l'occurence,

List[ ]

c'est-à-dire une liste vide qu'on peut aussi représenter par deux accolades de la manière suivante:

{ }

Si j'ai l'idée d'une liste et que je donne à la machine l'expression suivante

List[a, b, c]

elle va l'évaluer et me retourner en sortie l'expression suivante:

List[a,b,c]

représentable également sous la forme

{a,b,c}

JEAN

Comment est-ce que tu appelles les lettres a, b et c qui sont entre les crochets de l'expression d'entrée?

GIANNI

La plupart des scientifiques appellent la tête de l'idée ci-dessus "*fonction*" et ce qu'il y a entre les crochets "*arguments*".

JEAN

C'est très compliqué pour une intelligence humaine et non artificielle.

GIANNI

Pour permettre à l'intelligence humaine de s'y retrouver, la machine nous permet d'attribuer des noms à des expressions par l'intermédiaire du signe égal

"="

Ainsi, on peut commander à la machine de mémoriser l'expression que nous avions tout à l'heure sous un nom que l'on peut retenir facilement.

Par la commande ci-dessous, on peut dire à la machine que l'expression sera connue sous le nom "idée"

idée = fonction1[a, b] [ ] [fonction2[1], fonction3[x, y]]

A partir de ce moment, la machine évaluera les commandes qu'on lui donne en fonction de sa connaissance de l'idée qu'elle aura en mémoire sous le nom de "idée".

Elle répondra donc de la manière suivante aux diverses commandes qu'on lui donne

(les commandes sont en gras et les résultats de l'évaluation par la machine en non gras).

idée

fonction1[a, b] [ ] [fonction2[1], fonction3[x, y]]

Length[idée]

2

Head[idée]

fonction1[a, b][]

Length[Head[idée]]

0

Part[idée, 1]

fonction2[1]

Head[Part[idée, 1]]

fonction2

Part[Part[idée, 1], 1]

1

Part[idée, 2]

fonction3[x, y]

Head[Part[idée, 2]]

fonction3

Part[Part[idée, 2], 1]

x

Part[Part[idée, 2], 2]

y

Les atomes

En descendant dans les niveaux, à un moment ou à un autre, on atteint toujours les idées les plus simples que connait la machine, des idées que l'on peut qualifier de "*atomiques*" ou de "*atomse*", des idées que la machine ne peut plus découper, comme

"a", "x" ou **"1"**

dans l'expression précédente.

Par définition, les atomes sont des expressions qui n'ont qu'une tête et pas de corps, donc pas non plus de crochets.

La machine connait trois types fondamentaux d'atomes:

- les noms, qui sont des symboles ne contenant que des lettres et des chiffres;

- les textes, que l'on met toujours entre guillemets pour les distinguer des noms, et,

- les nombres qui peuvent être entiers, rationnels ou décimaux.

La tête d'un nom est

Symbol

La tête d'un nombre entier est

Integer

La tête d'un nombre rationnel est

Rational

La tête d'un nombre décimal est

Real

La tête d'un texte est

String

Le calcul

JEAN

Est-ce que la machine est capable de calculer?

GIANNI

Bien sûr.

JEAN

Tu peux me donner un exemple?

GIANNI

La machine connait les deux opérations d'addition et de multiplication, et donc celles de soustraction et de division qui ne sont que des dérivées des deux premières.

La machine sait donc faire la somme de nombres.

Pour faire la somme des nombres allant de 1 à 5, par exemple, on peut lui donner la commande suivante

Sum[énumérateur, {énumérateur, 1, 5}]

et la machine va me retourner son évaluation de cette expression, à savoir

15

L'évaluateur a fait le travail suivant:

1+2=3
3+3=6
6+4=10
10+5=15

La machine sait aussi faire le produit de nombres.

Pour faire le produit des nombres allant de 0 à 5, par exemple, on peut lui donner la commande

Product[énumérateur, {énumérateur,1, 5}]

et la machine va me retourner son évaluation de cette expression, à savoir

120

L'évaluateur a fait le travail suivant:

1*2=2
2*3=6
6*4=24
24*5=120

Si on a une liste d'éléments, on peut facilement calculer la moyenne de ces éléments, que ce soient des noms ou des nombres, peu importe.

On peut par exemple créer une liste constituée de trois éléments et lui donner le nom "liste"

liste = List[a, b, c]

{a,b,c}

Puis demander à la machine la longueur de cette liste en lui donnant la commande

Length[liste]

3

et calculer la moyenne en divisant la somme des éléments par la longueur de la liste.

Sum[liste[[énumérateur]], {énumérateur, 1, Length[liste]}]

a + b + c

Sum[liste[[énumérateur]], {énumérateur, 1, Length[liste]}]
/Length[list]

1/3 (a + b + c)

L'évaluateur

JEAN

Tu m'as dit que tout ce que sait faire la machine c'est d'évaluer des expressions.

C'est donc qu'elle a un évaluateur.

Comment fonctionne t-il?

GIANNI

L'évaluateur de la machine évalue les expressions en leur appliquant des règles de transformation.

En fait la machine possède un noyau contenant de très nombreuses règles de transformation qui lui sont propres.

Les autres sont celles qu'on lui donne nous-même.

C'est cet ensemble de règles de transformation, les siennes et les notres, qui constitue la connaissance de l'évaluateur.

Par exemple, on peut donner une règle de la forme

fonction[expression_] := expression^2
Le signe deux point égal

":="

dit à l'évaluateur de n'évaluer cette règle que lorsqu'on lui demande de l'évaluer et non pas tout de suite.

Ainsi, quand on a donné cette règle à la machine, on peut lui poser la question

fonction[a]

ce à quoi elle va répondre

a^2

ou encore

fonction[a+b]

ce a quoi elle va répondre

(a + b)^2

En fait, l'évaluateur travaille sans arrêt.

Il applique les règles transformation qu'il connait à l'expression qu'on lui donne jusqu'à ce qu'il n'en trouve plus aucune qui puisse s'appliquer, auquel cas il retourne la dernière expression obtenue comme résultat.

Les atomes sont évalués à eux-même, c'est-à-dire qu'aucun changement n'intervient

a
a

Les noms sont évalués à eux-même.

nom
nom

Les noms ayant une expression qui leur est attachée, sont évalués à expression

nom=expression
expression

nom:=expression
expression

Les nombres sont évalués à eux-même

2
2

Les textes sont évalués à eux-même

"texte"
texte

L'évaluation d'une expression est plus complexe.

D'abord la tête et les éléments du corps sont évalués récursivement.

Après l'évaluation de la tête et des éléments du corps, les attributs Flat et Orderless sont pris en compte par l'évaluateur.

Si la tête a l'attribut Orderless, les éléments sont triés dans un ordre standard propre à la machine.

L'attribut Flat assure l'aplatissement des occurrences emboitées de la tête.

Le pas suivant de l'évaluateur est gouverné par l'attribut Listable.

Si la tête possède cet attribut et si une liste figure parmi les éléments du corps, la tête List de la liste est échangée avec une tête Thread.

Enfin, les règles sont appliquées.

Les règles applicables sont celles qui sont associée avec les têtes des expressions ou avec les têtes des éléments du corps.

Dans chaque cas, le comparateur intervient pour trouver si une règle est applicable.

Si une règle est applicable, l'expression est remplacée par la droite de la règle, et l'évaluation recommence.

Les règles

Pour restreindre le nombre de règles à appliquer à chaque expression, l'évaluateur considère différents types de règles.

Une règle de la forme

nom := expression

est considérée par l'évaluateur comme appartenant au nom, une valeur propre dite ownvalue.

Une règle de la forme

nom[ ] := expression

est considérée par l'évaluateur comme appartenant a nom, une valeur dite downvalue.

Une règle de la forme

nom1 /: nom2[..., nom1[...], ...] := expression

ou

nom1 /: nom2[..., nom1, ...] := expression

est considérée par l'évaluateur comme appartenant à nom1, une valeur dite upvalue.

Les upvalues sont associées à des noms apparaissant comme arguments de la gauche.

Une règle de la forme

nom1[nom2[ ], ...] := expression

où nom1 est l'une fonction internes N, Default, Attribues ou Format appartient à nom2

est considérée par l'évaluateur comme appartenant à nom2

La déclaration

nom1 /:

n'est pas nécessaire dans ce cas.

Une règle de la forme

nom[...] [...] := expression

est considérée par l'évaluateur comme appartenant au nom nom.

Une telle règle est considérée comme downvalue.

Comme la tête de la gauche n'est pas un nom simple, aucune règle ne peut lui être associée.

Le comparateur

JEAN

Comment la machine fait elle pour savoir qu'elle doit appliquer une règle à une expression.

GIANNI

Tout le système repose sur un comparateur

Les listes

On commence par lui donner comme idée, celle de la liste a, b, c.

idée = List[a,b,c]

La machine me retourne comme résultat

List[a,b,c]

qui démontre qu'il a mémorisé la liste a,b,c sous le nom de "idée".

Table

Table est une commande qui accumule les résultats dans une liste qu'il génère.

Table[expr,n**]**

Generates a list of n copies of expr.

Table[expression, énumérateur]

Génère une liste de copies de l'expression correspondant à l'énumérateur.

Table[expression, {énumérateur, initiale, finale}]

D'abord l'énumérateur est mis à sa valeur initiale.

Puis l'expression est évaluée en utilisant la valeur courante de l'énumérateur s'il apparait dans l'expression.

Pour la répétition suivante, la valeur suivante énumérateur+1 est prise et l'expression est évaluée à nouveau.

Une autre manière de voir est de dire que l'expression est d'une manière ou d'une autre une fonction de l'énumérateur.

Pour chaque valeur de l'énumérateur, on obtient une valeur fonction[énumérateur] de la fonction représentée par l'expression.

Table[f[n],{n,20}]

en supposant qu'il n'y ait pas de conflit de noms de variables, c'est-à-dire que n n'apparaissent pas comme une variable libre dans la définition de f.

On peut également exprimer Table[] en terme de Array[].

Si l'expression à tabuler n'est pas de la forme

f[variable]

où variable est une variable de répétition

TableForm[]

Affiche les éléments d'une liste sous forme de tableau.

Array

Le répétiteur Array prend ce point de vue.

Array[f, n]
Génère une liste de longueur n dont les éléments sont f[n].

Array[f, n, n0]
Génère une liste en utilisant comme origine du compteur n0.

Array[f, n, {a, b}]
Génère une liste utilisant n valeurs comprises entre a et b.

Dans

Array[expression, n]
expression représente une expression qui est appliquée pour chaque valeur du compteur tour à tour.

Il n'est pas nécessaire d'avoir un compteur nommé puisque le nom du paramètre d'une fonction n'a aucune importance en l'occurrence.

La machine saisi simplement les fonctions f [1], f[2], ... and evaluates them.

Array[f,20]

est équivalent à

Array [list$_1$,**list2]**

Génère une liste de longueur n éléments, avec éléments f[i].

f est une fonction, i un indice.

SparseArray

Yields the same sparse array.

Generates a sparse array in which values val_i appear at positions pos_i.

By default, SparseArray takes unspecified elements to be 0.

SparseArray[list] requires that list be a full array, with all parts at a particular level being lists of the same length.

The individual elements of a sparse array cannot themselves be lists.

Functions with attribute Listable are automatically threaded over the individual elements of the ordinary arrays represented by SparseArray objects.

The standard output format for a sparse array indicates the number of nondefault elements and the total dimensions.

Dimensions gives the dimensions of a sparse array.

SparseArray[list**]**

Yields a sparse array version of list.

Normal[**SparseArray**[…]]

gives the ordinary array corresponding to a sparse array object.

SparseArray[data,{d_1,d_2,…}]

yields a sparse array representing a $d_1 \times d_2 \times \ldots$ array.

SparseArray[data,dims,**val]**

yields a sparse array in which unspecified elements are taken to have value val.

SparseArray[{{i_,i_}->1},{d,d}**]**

Gives a d×d identity matrix.

Applying an action

Two important commands that take actions as arguments are Map and Apply.

Map

Map[f,**expr]**

or

f/**@**expr

applies f to each element on the first level in expr.

applies f to parts of expr specified by levelspec.

The default value for levelspec in Map is {1}.

{n}: level n only

{n_1,n_2}: levels n_1 through n_2

Infinity: levels 1 through Infinity

Map[f]

represents an operator form of Map that can be applied to an expression.

Map (/@)

Map[action, idea] (/@)

Map[action, list] maps the action over each member of a list.

This means that this command forms the idea action[mi] for each member mi of the list and returns the list of the results.

However, the second element of the command needs not be a list.

Any idea of the form Head[Body] will do.

Map[action, Head[Body]]

will return

action[Body]

If the body is constituted of several members the action is applied to each member of the body of the idea

Map[action, Head[m1, m2, m3]]

will return

Head[action[m1], f[m2], f[m3]

If the head of the idea is Plus,

Plus[m1, m2, m3]

Map of an action on the idea will return

Plus[action[m1], action[m2], action[m3]

The mapping of an anonymous action is possible.

The command

Map[#[[2]]& List[Plus[a,b], Times[2, x, y], Power[z,2]]

will return

List[b, x, 2]

The infix notation of the command can be used for Map

action /@ idea

action /@ Head[Body]

action /@ List[a,b]

returns

List[action[a], action[b]]

The command

action /@ Plus[a,b]

returns

Plus[action[a], action[b]]

Map[action, idea, level]

The optional third level argument allows to specify the level of the body at which to Map.

The default level is {1}, that is the members of the body.

In a matrix, the members are at level {2} because there are two levels of lists.

If we want to map an action to these members, we can use

Map[action, matrix, {2}]

The command

Map[action, matrix, 2]

would only map the action up to level 2.

The command

Map[action, idea, {-1}]

would map the action at the lowest levels, that is at the level of the atoms of the idea.

MapAt[action, idea, position]

The command Map[] always maps an action at all members of a given level.

The command

MapAt[action, idea, positions]

The position may be a single position or a list of positions.

The list of positions is a list of numbers that describe how to descend down the tree of the idea to that place

MapAt[action, Plus[Times[a,b], Times[c,d],{2,1}]

returns

Plus[Times[c,d], Times[action[c],d]]

Apply

Apply (@@)

Apply[] is a generalization of the usual notion of applying an action to something.

When we speak of applying an action to an idea, we mean to form the idea

action[idea]

and to evaluate it.

If we want to apply an action to several arguments, things get more complicated.

If we have a list

List[a, b, c]

and we want to compute

action[a, b, c]

The command

action[List[a, b, c]

Would be wrong, because it would pass the whole idea with its head to the action.

The command

Apply[action, idea]

does what we want.

It forms the idea

action[a, b, c]

Looked from a different point of view, it replaces the Head of the idea by the action.

If we replace List by action, we get the right idea.

Apply[Times[Plus[a, b, c]]

returns

Times[a, b, c]

average[list] := Apply[Plus, list] / Length[list]

Apply can be written with the infix notation @@

action @@ List[a, b, c]

returns

action[a, b, c]

Opérations

Injection

L'injection utilise deux fonctions pour combiner les éléments de ses deux arguments.

Inner[fonction1, argument1, argument2, fonction2]

fonction1 est la fonction pour la multiplication et fonction2 celle pour l'addition.

Inner[f,$list_1$,$list_2$,g**]**

In Dot f plays the role of multiplication and g of addition.

Ejection

Outer[f, {a, b}, {c, d}]

Forms all possible combinations of the lowest-level elements in each of the lists, and feeds them as arguments to f.

CartesianProduct

CartesianProduct[l_1,l_2]

Gives the Cartesian product of lists l_1 and l_2.

Kronecker

KroneckerProduct[m_1,m_2,…]

Constructs the Kronecker product of the arrays m_i.

Works on vectors, matrices, or in general, full arrays of any depth.

KroneckerDelta[n_1,n_2,…]

gives the Kronecker delta equal to 1 if all the n_i, are equal, i = i, and 0 otherwise, i ≠ i.

Les ensembles

Subsets

Subsets[*list*]

Gives a list of all possible subsets of list.

Les unités et quantités

La machine sait faire de l'arithmétique non seulement avec des nombres et des symboles mais aussi avec des unités.

Elle sait également faire de l'analyse naturelle et des opérations purement symboliques sur les quantités.

Entrées

Quantity

Une quantité avec une magnitude et une unité.

QuantityArray

Un Array de quantités avec la même unité, avec une unité commune.

QuantityDistribution

Distribution avec unités.

Composants

QuantityMagnitude

Donne la magnitude.

QuantityUnit

Donne l'unité.

UnitDimensions

Donne la dimension associe hors les SI connues

QuantityQ

KnownUnitQ

Affichage

QuantityForm

TargetUnits

Conversions

$UnitSystem

Système par défaut.

UnitConvert

CompatibleUnitQ

CommonUnits

Convertit une liste de Quantities vers une Unit commune

UnitSimplify

Convertit à des Units plus simples.

Quantités avec incertitude

Around

VectorAround

MeanAround

AroundReplace

Quantités physiques

QuantityVariable

Une variable représentant une quantité (m pour masse)

QuantityVariableIdentifier

QuantityVariablePhysicalQuantity

Analyse naturelle

DimensionalCombinations

Combinaison possible de quantités.

QuantityVariableDimensions

QuantityVariableCanonicalUnit

NonDimensionalizationTransform

Unités mixtes

MixedUnit

MixedMagnitude

Unités monétaires

CurrencyConvert

DateUnit

InflationAdjust

Unités indépendantes

IndependentUnit

IndependentPhysicalQuantity

IndependentUnitDimension

Les arbres

Racine-feuilles

Parents-enfants

InOrder means the parent is in between the 2 childs.

	TopDown LeftRight	TopDown RightLeft	BottomUp LeftRight	BottomUp RightLeft
DepthFirst				
BreadthFirst				
LeavesFirst				

I fold over a list and I fold over a tree with a certain traversal order.

TreeMap is also progressively nutriting the tree from the bottom and a TreeFold ends up in the root.

TreeFold a function on a tree: starting at the leaves and applying a function to my subtrees and propagating the value up to the top.

I cannot compute a function until I do not have the input to the function.

The default is DepthFirst.

CEOing

Le temps est utilisé par le scientifique pour construire la réalité en mettant des dérivées temporelles à gauche des équations, une manière de structurer son intuition.

Tout le problème est celui du changement, du mouvement, du transport.

The observer creates in a mathematical, scientific time.

The issue is the reference frame to create things.

The question of energy momentum.

The scientist is building a template, a platonic view, restricting things to a system, getting intuitions, a morphism between systemics and reality, in a space of observation, sensing the reality.

Persistency of mass, of ideas, new mass coming, due to energy momentum.

The plan is to have this function able to eat that kind of information.

What folliation can be created from that data.

When I do sorting what is the traversal order.

Linear list. trees and nodes, and orderings.

Visit a tree.

When I am mapping I am changing things.

Do I refer to the original thing or the changed thing.

I want to map a function to nodes, to a subset of nodes.

Map just maps an f around everything.

Connection with TENSORS.

Evaluation order.

What does it do when you feed it with something it does not want to eat?

Why it did not work? No clue.
WHAT IS THI FUNCTION, ENTITY, EATING?

THIS OMNIVOROUS FUNCTION EATING.

AN IMAGE AND GIVING A GRAPHICS.

ASSUMING I HAVE A POINT CLOUD OF SOMETHING.

Assuming there is a source for the POINTS CLOUD.

Creation of webpages.

And then there is the question of the other way around.

Backtick `.

The PATTERN MATCHER can take something that is embedded inside an existing thing.

Ingesting something into a functions.

Equivalence together 2 things.

Logic programming, theorem prover graph rewriting, unification, combinators, permutations, most general, total ordering, to reach the normal form.

Least computation principle to explore a tree, A DERIVATION TREE.

To feed a function with an object in order to get a result.