The Structure and Beauty of the Mathematica Language

David Leibs
david.leibs@gmail.com

Building Material

"Lisp isn't a language, it's a building material"

Alan Kay

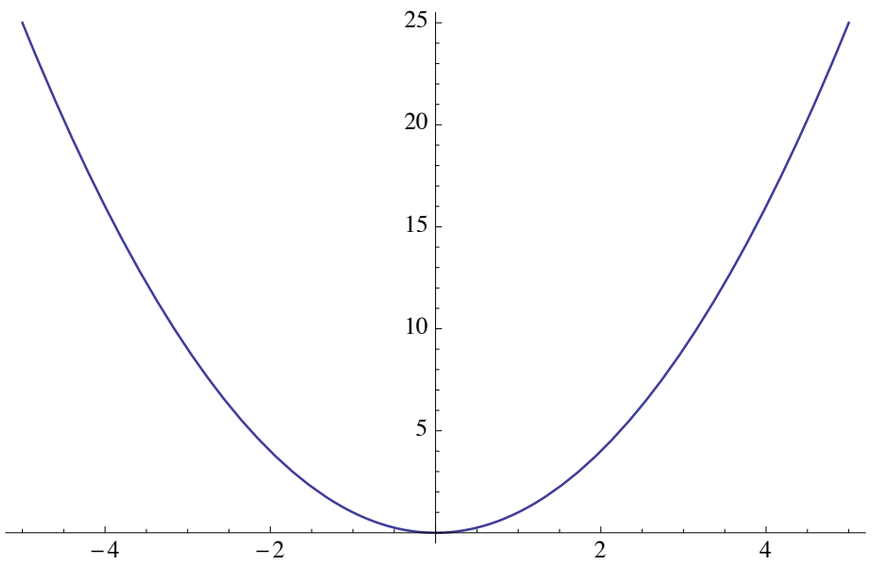
Mathematica is a power tool

- Focus on making the user productive
- Symbolic
- Functional
- Rule Based
- Amazing Library
- “Lispy"

Exploration

In[1] := Plot[x^2, {x, -5, 5}]

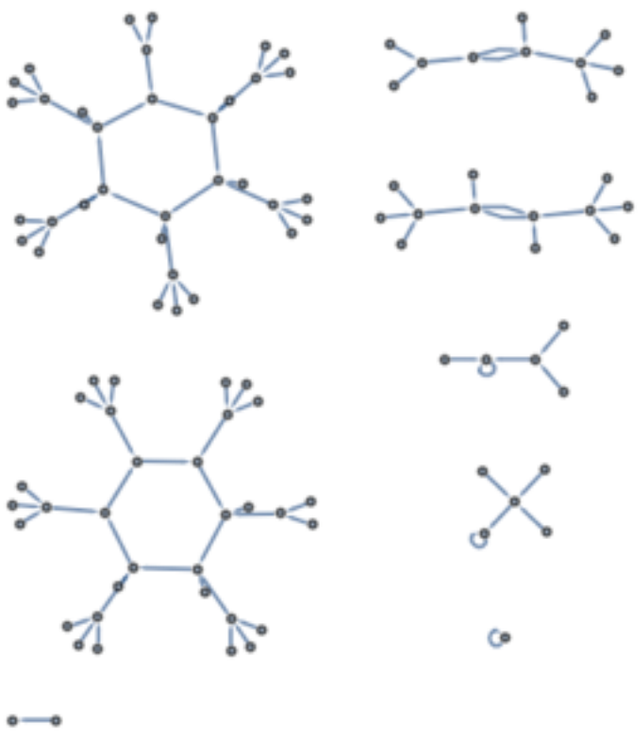
Out[1]:=



Visualization

In[1] := Graph[Table[i -> Mod[i^2, 74], {i, 100}]]

Out[1]:=



Symbolic

In[1] := PDF[NormalDistribution[mean, sd]]

Out[1]=
$$\frac{\text{Exp}\left[-\frac{1}{2}\left(\frac{\#1-\text{mean}}{\text{sd}}\right)^2\right]}{\text{sd}\sqrt{2\pi}}\ \&$$

Symbolic

In[1] := PDF[NormalDistribution[mean, sd]] // CForm

Out[1]:= Function(Exp(-(Power((Slot(1) - mean)/sd,2)/2.))/
(sd*Sqrt(2*Pi)))

Visualize the Gradient

We will again use the Dynamic command to visualize the gradient vector.

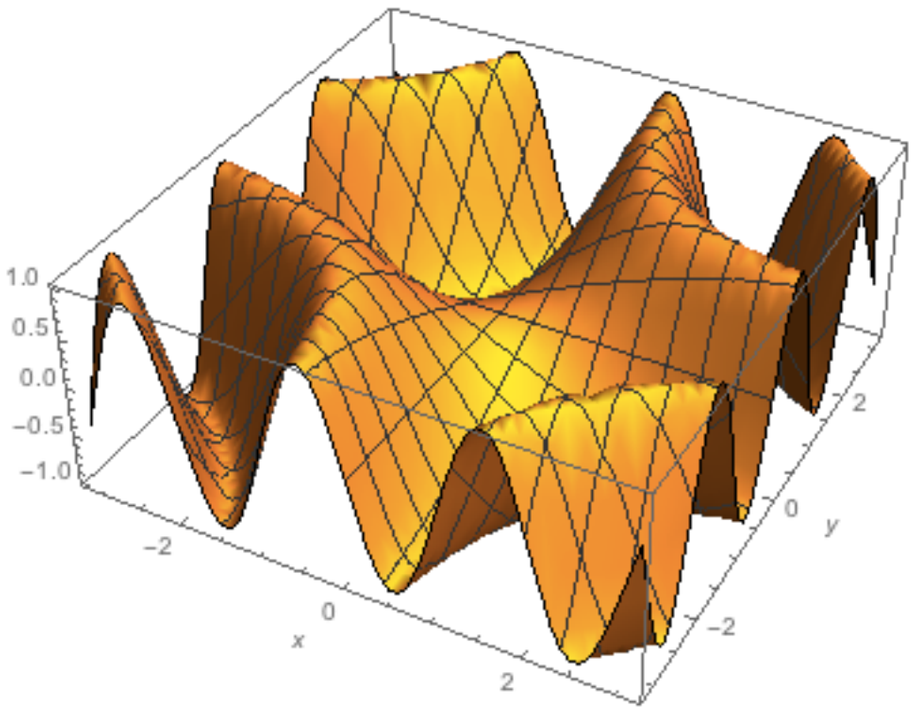
First, define a function and its gradient. Remember that the gradient is vector-valued!

In[2]:=
```
f[x_, y_] := Sin[x y]
```

In[3]:=
```
gradf[x_, y_] := Evaluate[D[f[x, y], {{x, y}}]]
```

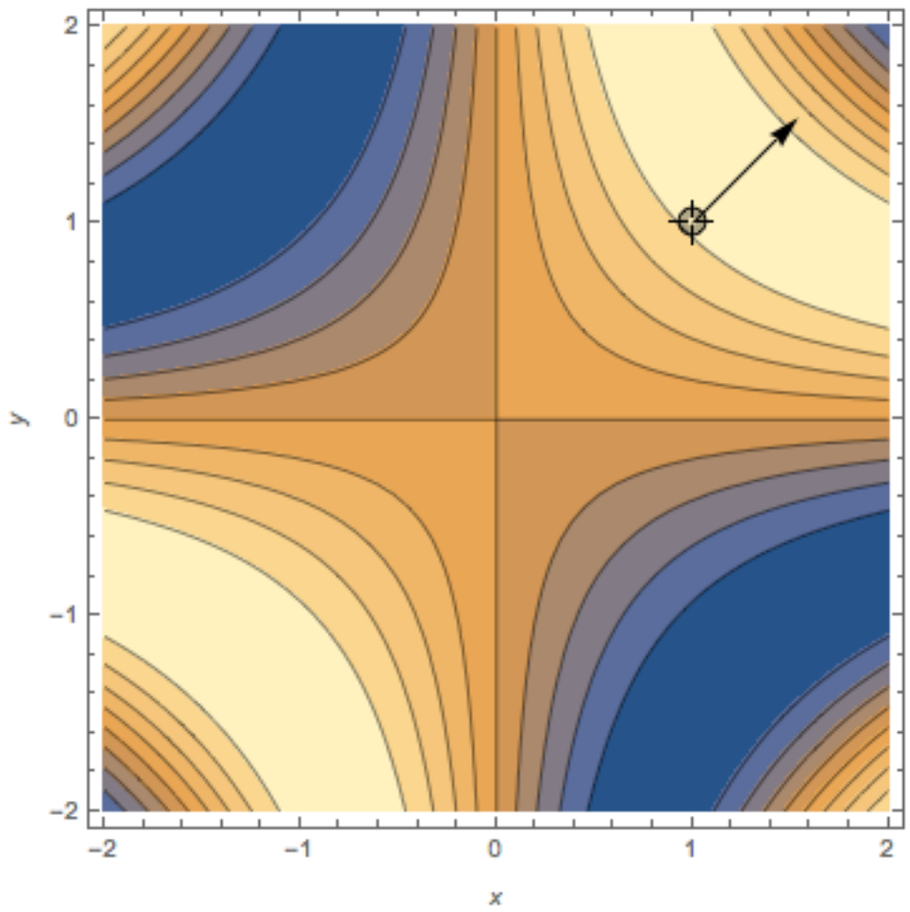
Plot our function.

```
Plot3D[f[x, y], {x, -π, π}, {y, -π, π}, AxesLabel → {x, y}]
```



In[5]:=
```
DynamicModule[
 {basePt = {1, 1},
  plot = ContourPlot[f[x, y], {x, -2, 2}, {y, -2, 2}, FrameLabel → {x, y}]},
 Show[plot,
      Graphics[
       {Dynamic[Arrow[{basePt, basePt + gradf @@ basePt}]],
        Locator[Dynamic[basePt]]}
      ]
 ]
]
```

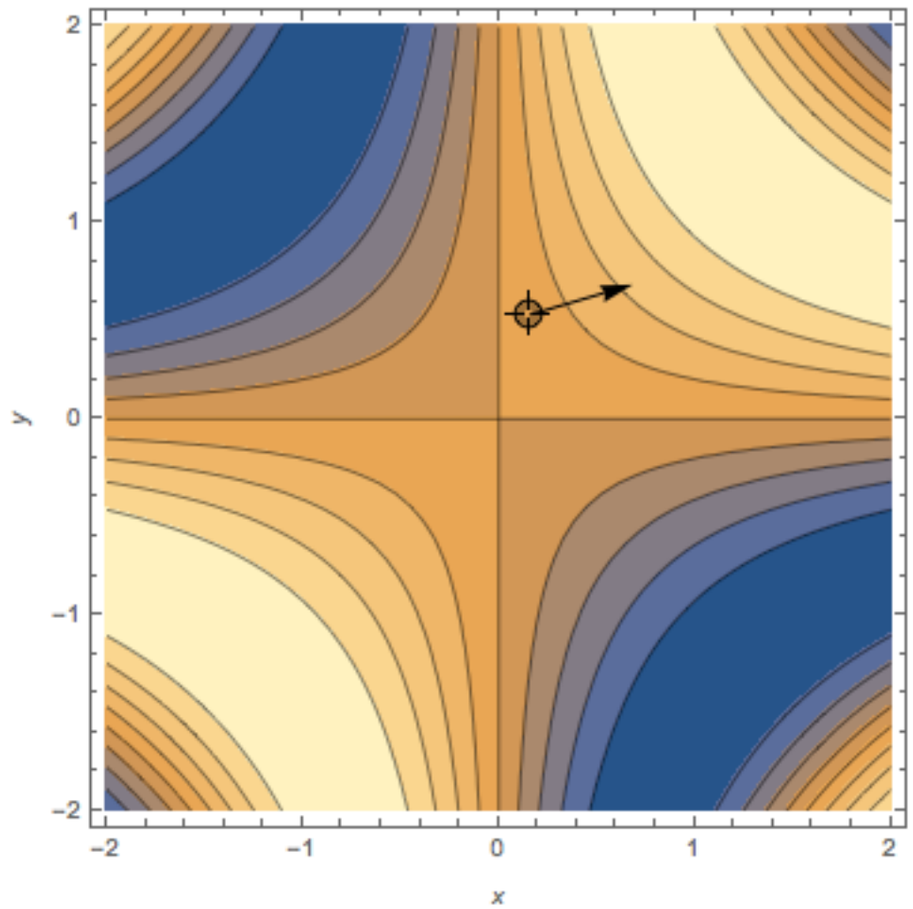
Out[5]=



Notice that the vector always points "up hill", and it gets longer as the hill gets steeper. This is a defining property of the gradient.

In[5]:=
```
DynamicModule[
 {basePt = {1, 1},
  plot = ContourPlot[f[x, y], {x, -2, 2}, {y, -2, 2}, FrameLabel → {x, y}]},
 Show[plot,
      Graphics[
       {Dynamic[Arrow[{basePt, basePt + gradf @@ basePt}]],
        Locator[Dynamic[basePt]]}
      ]
 ]
]
```

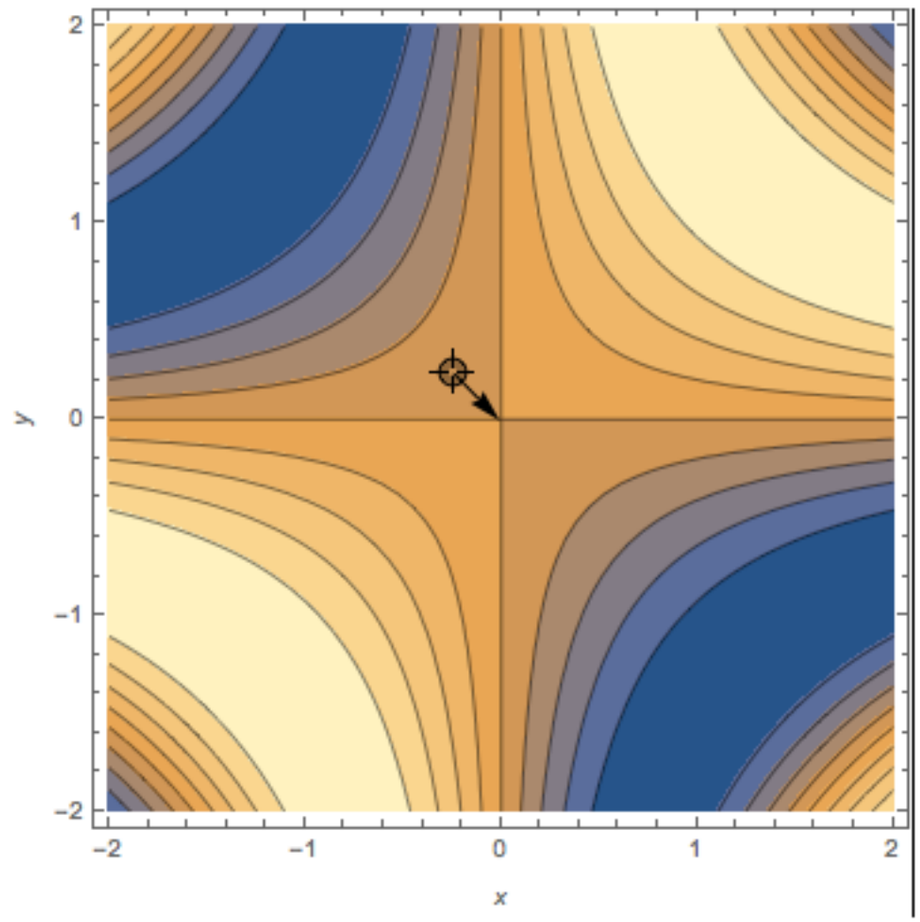
Out[5]=



Notice that the vector always points "up hill", and it gets longer as the hill gets steeper. This is a defining property of the gradient.

In[5]:=
```
DynamicModule[
 {basePt = {1, 1},
  plot = ContourPlot[f[x, y], {x, -2, 2}, {y, -2, 2}, FrameLabel → {x, y}]},
 Show[plot,
      Graphics[
       {Dynamic[Arrow[{basePt, basePt + gradf @@ basePt}]],
        Locator[Dynamic[basePt]]}
       ]
 ]
]
```

Out[5]=



Notice that the vector always points "up hill", and it gets longer as the hill gets steeper. This is a defining property of the gradient.

Mathematica Expressions

In[1] := expression

Out[1]:= result

Pass 1: Simple Evaluation

In[1] := 1

Out[1]:= 1

In[1] := 2

Out[1]:= 2

In[1] := “Hello World!”

Out[1]:= Hello World!

Like a Calculator

In[1] := 3+4

Out[1]:= 7

Like Algebra I

In[1] := 3+4*5

Out[1]:= 23

Function

In[1] := Sqrt[49]

Out[1]:= 7

Lists

In[1] := {3,4,{5,6}}

Out[1]:= {3,4,{5,6}}

APL Like

```
In[1] := {10,20,30} + 5

Out[1]:= {15,25,35}
```

Array Programming

In[1] := {10,20,30} + {5,10,20}

Out[1]:= {15,30,50}

Higher Order Functions

In[1] := Map[Sqrt, {9,16,25,36}]

Out[1]:= {3,4,5,6}

Pure Functions

```
In[1] := Map[#^2&, {3,4,5,6}]

Out[1]:={9,16,25,36}
```

Functions

In[1] := Map[Function[{x},x^2], {3,4,5,6}]

Out[1]:= {9,16,25,36}

Higher Order Functions

In[1] := Nest[# * 1.06&, 100, 10]

Out[1]:= 179.085

Higher Order Functions with history

```
In[1] := NestList[ # * 1.06&, 100, 10]

Out[1]:= {100, 106., 112.36, 119.102, 126.248, 133.823, 141.852,
    150.363, 159.385, 168.948, 179.085}
```

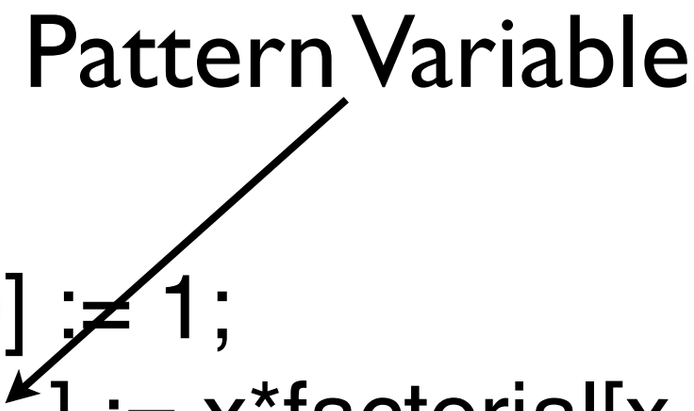
Rule Oriented Style

```
In[1] := factorial[0] := 1;
         factorial[x_] := x*factorial[x - 1];
```

```
In[2] := factorial[5]
```

```
Out[1]:= 120
```

Patterns




```
In[1] := factorial[0] := 1;
        factorial[x_] := x*factorial[x - 1];

In[2] := factorial[5]

Out[1]:= 120
```

In[1] := Map[factorial,{3,4,5}]

Out[1]:= {6, 24, 120}

Imperative Constructs

```
In[1] := L = {};
         For[i = 0, i < 10, i++,
             L = Append[L, i]];
         L

Out[1]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Why isn't Mathematica yelling at me?

In[1] := **X**

Out[1]:= **X**

Is this Heresy?

In[1] := 3+x

Out[1]:= 3+x

Call a Function that does not exist!

In[1] := f[5]

Out[1]:= f[5]

Partial Evaluation

In[1] := f[2+3]

Out[1]:= f[5]

Partial is Good

```
In[1] := Map[f, {3,4,5}]

Out[1]:= {f[3], f[4], f[5]}
```

Expression Surgery

In[1] := Apply[factorial, %, 1]

Out[1]:= {6,24,120}

Let's Start Over

In[1] := pass 2

Out[1]:= pass 2

Raw Expressions

In[1] := 1

Out[1]:= 1

Strings

In[1] := “Hello World”

Out[1]:= Hello World

Atoms

In[1] := AtomQ[1]

Out[1]:= True

Symbols

In[1] := **X**

Out[1]:= **X**

Rules Not Slots

In[1] := $x = y$

Out[1]:= y

Evaluation

In[1] := x

Out[1]:= y

In[1] := $y = 1$

Out[1]:= 1

Repeated Evaluation

In[1] := x

Out[1]:= 1

In[1] := $4 + x$

Out[1]:= 5

Quoting

In[1] := Hold[1+2]

Out[1]:= Hold[1+2]

Canonicalization

```
In[1] := FullForm[Hold[1+2]]

Out[1]:= Hold[Plus[1, 2]]
```

Canonicalization

```
In[1] := {1,2,3,Hold[3+4]} //FullForm

Out[1]:= List[1,2,3,Hold[Plus[1, 2]]]
```

Expressions

Atom

or

head[exp0, exp1, ... ,expn]

Heads

In[1] := Head[a + b]

Out[1]:= Plus

Heads

In[1] := Head[Plus]

Out[1]:= Symbol

Evaluation

In[1] := Head[3+4]

Out[1]:= Integer

Holding Evaluation

In[1] := Head[Hold[3+4]]

Out[1]:= Hold

Holding Evaluation

In[1] := Head[First[Hold[3+4]]]

Out[1]:= Integer

Holding Evaluation

In[1] := Head[Unevaluated[3+4]]

Out[1]:= Plus

Juxtaposition is Multiplication

```
In[1] := Map[Head, Unevaluated[3 + 4]]

Out[1]:= 2 Integer
```

The Evaluator

Map[Head, Unevaluated[3 + 4]]
the evaluator sees:
  Map[Head,Plus[3,4]]
and reduces to:
  Plus[Integer,Integer]
but it keeps evaluating using rules it knows and gives us:
  Times[2, Integer]

The Evaluator

e0 [e1, e2, e3, … , en]

eval[e0] [eval[e1], eval[e2], eval[e3], …, eval[en]

...
match to rules and possibly rewrite

Evaluation

In[1] := Plus[3+4,5+6]

Out[1]:= 18

The symbol f has no rule

In[1] := f[3+4, 5+6]

Out[1]:= f[7,11]

Apply lets us change a Head

```
In[1] := Apply[Plus, f[3+4, 5+6]]

Out[1]:= 18
```

Rules: Set

In[1] := x = Random[]

Out[1]:= 0.114681

x has a value

In[1] := {x, x, x}

Out[1]= {0.114681, 0.114681, 0.114681}

Rules: SetDelayed

In[1] := y := Random[]

Rules: SetDelayed

```
In[1] := {y, y, y}

Out[1]:= {0.304997, 0.897893, 0.00343832}
```

Rules: OwnValues

```
In[1] := {OwnValues[x], OwnValues[y]}

Out[1]:= {{HoldPattern[x] :> 0.114681},
          {HoldPattern[y] :> Random[]}}
```

Patterns

In[1] :=

Out[1]:=

Blank

In[1] := Hold[_] //FullForm

Out[1]:= Hold[Blank[]]

Blank with Head

In[1] := Hold[_f] //FullForm

Out[1]:= Hold[Blank[f]]

Matching

- _ Blank
- __ BlankSequence (one or more)
- ___ BlankNullSequence (0 or more)
- | Alternatives
- : Optional
- .. Repeated
- ? PatternTest

Matching

In[1] := MatchQ[5, _Integer]

Out[1]:= True

Replacement Rules

In[1] := 5 x + 19.5 x^2 /. v_Real -> v^2

Out[1]:= 5 x + 380.25 x^2

Rules

```
In[1] := Hold[v_Integer -> v^2] // FullForm

Out[1]:= Hold[Rule[Pattern[v,Blank[Integer]],Power[v,2]]]
```

Destructuring

```
In[1] := f[m] + g[n] /. x_[y_] -> y[x]

Out[1]:= m[f] + n[g]
```

Factorial Again

```
In[1] := factorial[0] := 1;
         factorial[x_] := x*factorial[x - 1];
```

```
In[2] := factorial[5]
```

```
Out[1]:= 120
```

DownValues

```
In[1] := DownValues[factorial]

Out[1]:= {HoldPattern[factorial[0]] :> 1,
    HoldPattern[factorial[x_]] :> x factorial[x - 1]}
```

Factorial Matching Style

```
In[1] := fact[5] //. {fact[0] :> 1, fact[x_] :> x * fact[x - 1]}

Out[1]:= 120
```

Look closer with FixedPointList

```
In[1] := FixedPointList[# /. {fact[0] :> 1, fact[x_] :> x * fact[x - 1]} &,
    fact[5]]

Out[1]= {fact[5], 5 fact[4], 20 fact[3], 60 fact[2], 120 fact[1],
 120 fact[0], 120, 120}
```

In[1] := f[x_, y_] := *body*;

In[2] := f[a, b]

ReleaseHold[Hold[*body*] /. {x :> a, y:>b}]

Enough with factorial already!

In[1] :=

```
Clear[factorial];
factorial[0] := 1;
factorial[x_Integer] := Apply[Times, Range[x]]]
factorial[x_Real] := Gamma[x-1];
```

Symbols have Attributes

- HoldFirst
- HoldRest
- HoldAll
- Listable
- Flat
- Orderless
- ...

Attribute: HoldFirst

```
In[1] := SetAttributes[f,HoldFirst];
```

```
In[2] := f[3+4, 5+6]
```

```
Out[1]:= f[3+4, 11]
```

Attribute: HoldRest

```
In[1] := Clear[f];
         SetAttributes[f,HoldRest];
```

```
In[2] := f[3+4, 5+6]
```

```
Out[1]:= f[7, 5+6]
```

Attribute: HoldAll

```
In[1] := Clear[f];
         SetAttributes[f,HoldAll];

In[2] := f[3+4, 5+6]

Out[1]:= f[3+4, 5+6]
```

Control

```
In[1] := SetAttributes[if, HoldRest];
         if[True, thenPart_, elsePart_]   := thenPart;
         if[False, thenPart_, elsePart_] := elsePart;
```

```
In[2] := x = 10;
         if[x < 100, 3+4, 5+6]
Out[1]:= 7

In[2] := if[x > 100, 3+4, 5+6]
Out[1]:= 11
```

Is this a Bug?

In[1] := if[100, 3+4, 5+6]

Out[1]:= if[100, 3+4, 5+6]

Will It Map?

```
In[1] := Map[if[# < 10, # * #, # + #] &, {5, 6, 25, 50, "die"}]

Out[1]:={25, 36, 50, 100, if["die" < 10, "die" "die", "die" + "die"]}
```

If you think you like fexprs

Check out the material by John Shutt on Vau-Calculus at: http://fexpr.blogspot.com.

There is nothing special about Hold

```
In[1] := SetAttributes[hold,HoldAll];
         releasehold[hold[x___]] := x

In[2] := releasehold[hold[3 + 4, 5 + 6]]

Out[1]:= Sequence[7, 11]
```

Sequence Splices

```
In[1] := {releasehold[hold[3 + 4, 5 + 6]]}

Out[1]:= {7, 11}
```

Attribute: Listable

```
In[1] := Clear[f];
         SetAttributes[f,Listable];
```

```
In[2] := f[{a,b,c},{d,e,f}]
```

```
Out[1]:= {f[a, d], f[b, e], f[c, f]}
```

Scoping Constructs

- Function
- With
- Block
- Module

Function: A Very Interesting Head

In[1] := Function[3+4]

Out[1]:= 3 + 4 &

In[1] := Function[3 + 4] // FullForm

Out[1]:= Function[Plus[3,4]]

In[1] := Function[3+4][]

Out[1]:= 7

In[1] := Function[{x,y}, x + y] [3,4]

Out[1]:= 7

In[1] := Map[Sqrt][{4, 9, 25}]

Out[1]:= {2, 3, 5}

Function: A Very Interesting Result

In[1] := Function[{x}, Function[{y}, x*y]] [3]

Out[1]:= Function[{y$}, 3 y$]

Y: A Very Interesting Function

Avert your eyes!

```
In[1] := y := Function[f,
    Function[x,
        f[Function[y, x[x][y]]]][Function[x, f[Function[y, x[x][y]]]]]]
```

Oh No, Factorial Again!

Please don't look at that result!

```
In[1] := factorial = y[Function[f, Function[n, If[ n == 0, 1, n * f[n - 1]]]]]

Out[1]:= Function[n$,  If[n$ == 0, 1, n$ Function[y$, Function[x$,
          Function[f, Function[n, If[n == 0, 1, n f[n - 1]]]]
        [ Function[y$, x$[x$][y$]]]][ Function[x$,
          Function[f, Function[n, If[n == 0, 1, n f[n - 1]]]]
        [Function[y$, x$[x$][y$]]]][y$]][n$ - 1]]]
```

It Computes

In[1] := factorial[5]

Out[1]:= 120

No Names Needed

```
In[1] := y[Function[f, Function[n, If[ n == 0, 1, n * f[n - 1]]]]] [5]

Out[1]:= 120
```

Read about Y

Read about this wonderful function called Y
at Dick Gabriel's website:

http://www.dreamsongs.com/Files/WhyOfY.pdf

With

With[{x = x0, y = y0, z = z0}, expr]

Meta Programming

```
In[1] := strides[dims_] :=  Map[Apply[Times, #] &,
          NestList[Rest, Rest[dims ~Join~ {1}], Length[dims] - 1]];

        indexer[shape_] :=
         With[{syms = Map[Unique[a] &, shape], shp = strides[shape]},
          With[{pat = 1 + (Plus @@ Apply[Times, Partition[Riffle[syms, shp], 2], 1])},
            Function @@ Hold[syms, pat]]];
```

```
In[2] := indexer[{3, 3, 9}]
```

```
Out[1]:= Function[{a$69734, a$69735, a$69736}, 1 + 27 a$69734 + 9 a$69735 + a$69736]
```

Block

In[1] := {x, y, z} = {3, 4, 5};

In[2] := Block[{x = 10, y = 20, z = 30}, {x, y, z}]

Out[1]= {10, 20, 30}

In[3] := {x, y, z}

Out[2]= {3,4,5}

Module

Module[{x,y,...}, expr]
Module[{x = x0, y = y0, …}, expr]

In[1] :=
```
Module[{l = {}, count = 10, i},
  For[i = 0, i < count, i++,
    l = Append[l, i]];
  l]
```

Out[1]:= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Make Y Combinator Pretty

```
In[1] := y := Function[f,
            Module[{g = Function[h, Function[x, f[h[h]][x]]]}, g[g]]];
```

Make Objects in a Functional Way

```
In[1] := makeAccount[name_, start_] :=
          Module[{accname = name, balance = start},
           account[
             Function[accname],
             Function[x,balance = balance + x],
             Function[x,balance = balance - x],
             Function[balance]]];
```

Cool, but what an ugly expression

```
In[1] := acc1 = makeAccount["david", 1000]

Out[1]:= account[
           accname$89163 &,
           Function[x$, balance$89163 = balance$89163 + x$],
           Function[x$, balance$89163 = balance$89163 - x$],
           balance$89163 &]
```

FormatValues

```
In[1] := Format[account[n_Function, _, _, _]]
           := StringJoin["account[", n[], "]"];
```

```
In[2] := acc1
```

```
Out[1]:= account[david]
```

Standard DownValue

```
In[1] := name[account[n_Function, _, _, _]]                := n[];
         deposit[account[_, d_Function, _, _], amt_]       := d[amt];
         withdraw[account[_, _, w_Function, _], amt_]      := w[amt];
         balance[account[_, _, _, b_Function]]             := b[];
```

```
In[2] := balance[acc1]
```

```
Out[1]:= 1000
```

UpValues

```
In[1] := account /: Plus[a : account[___], b_]
          := b + balance[a];
        account /: Plus[a : account[___] , b : account[___]]
          := balance[a] + balance[b];
```

```
In[2] := acc1 + 100
```

```
Out[1]:= 1100
```

```
In[3] := acc1 + acc1
```

```
Out[2]:= 2000
```

Fibonachi

In[1] :=
```
fib[0] = fib[1] = 1;
fib[n_] := fib[n-1] + fib[n-2];
```

In[2] :=
```
Array[fib, 8]
```

Out[1]:=
```
{1, 2, 3, 5, 13, 21, 34}
```

Memoizing Fibonachi

```
In[1] := Clear[fib];
         fib[0] = fib[1] = 1;
         fib[n_] := fib[n] = fib[n-1] + fib[n-2];
```

```
In[2] := fib[100]
```

```
Out[1]:= 573147844013817084101
```

Make Transformation Rules

```
ClearAll[pushR, popR, dupR, swapR, rotR, topR, nextR];
(* stack-to-stack transforms *)
pushR = {{stack___}, datum_} :> {datum, stack};
popR = {{top_, rest___} :> {rest}};
dupR = {{top_, rest___} :> {top, top, rest}};
rotR = {{top_, rest___} :> {rest, top}};
swapR = {{top_, next_, rest___} :> {next, top, rest}};
(* stack-to-value transforms *)
topR = {{top_, rest___} :> top};
nextR = {{top_, next_, rest___} :> next};
```

Microcode for a Forth Machine

```
ClearAll[exec,execAll];
microcode = Dispatch@{
(*BINARIES*)
{stack_,plus}:>With[{r=(stack/.nextR)+(stack/.topR)},
   {stack/.popR/.popR,r}/.pushR],
{stack_,times}:>With[{r=(stack/.nextR)*(stack/.topR)},
   {stack/.popR/.popR,r}/.pushR],
{stack_,minus}:>With[{r=(stack/.nextR)-(stack/.topR)},
   {stack/.popR/.popR,r}/.pushR],
{stack_,div}:>With[{r=(stack/.nextR)/(stack/.topR)},
   {stack/.popR/.popR,r}/.pushR],
{stack_,uminus}:>({stack/.popR,-(stack/.topR)} /. pushR),
(*NULLARIES*)
{stack_,pop}:>(stack/.popR),
{stack_,dup}:>(stack/.dupR),
{stack_,rot}:>(stack/.rotR),
{stack_,swap}:>(stack/.swapR),
(*UNARY-- DEFAULT*)
{stack_,x_}:>({stack,x}/.pushR)};
```

A little functional code

```
exec = machineState : {stack_, instr_} :> (machineState /. microcode);

execAll = {stack_, {instr_,  instrs___}} :> ({{stack, instr} /. exec, {instrs}});

execute[stack_, instrs_] := First[First[FixedPoint[x ⊂ x /. execAll, {stack, instrs}]]];

execAllTrace[stack_, instrs_] :=
  Module[{history = First /@ FixedPointList[x ⊂ x /. execAll, {stack, instrs}] // Most},
   Grid[Partition[Join[{start}, Riffle[history, instrs]], 2], Frame -> All]];
```

Execute and Observe

In[1] := execAllTrace[{}, {a, b, 3, 4, plus, rot, div, plus}]

Out[1]:=

start	{}
a	{a}
b	{b, a}
3	{3, b, a}
4	{4, 3, b, a}
plus	{7, b, a}
rot	{b, a, 7}
div	$\left\{\frac{a}{b}, 7\right\}$
plus	$\left\{7 + \frac{a}{b}\right\}$

J Interpreter

Stack Matching Rules

This is the set of rules for the J interpreter. J interleaves parsing and execution. The rule here do matching on a 4 element window on the stack. The last rule, move, moves more items from a list when no other rule matches. When a rule matches an action is called.

In[1]:=
```
Clear[parse];
parse[{a_Mark | a_Asgn | a_Lpar, b_Verb, c_Noun, d_, rest___}] := monad[0];
parse[{a_Mark | a_Asgn | a_Lpar | a_Adv | a_Verb | a_Noun, b_Verb, c_Verb, d_Noun, rest___}] := monad[1];
parse[{a_Mark | a_Asgn | a_Lpar | a_Adv | a_Verb | a_Noun, b_Noun, c_Verb, d_Noun, rest___}] := dyad[2];
parse[{a_Mark | a_Asgn | a_Lpar | a_Adv | a_Verb | a_Noun, b_Verb | b_Noun, c_Adv, d_, rest___}] := adverb[3];
parse[{a_Mark | a_Asgn | a_Lpar | a_Adv | a_Verb | a_Noun, b_Verb | b_Noun, c_Conj, d_Verb | d_Noun, rest___}] := conj[4];
parse[{a_Mark | a_Asgn | a_Lpar | a_Adv | a_Verb | a_Noun, b_Verb | b_Noun, c_Verb, d_Verb, rest___}] := fork[5];
parse[{a_Mark | a_Asgn | a_Lpar, b_Conj | b_Adv | b_Verb | b_Noun, c_Conj | c_Adv | c_Verb | c_Noun, d_, rest___}] := bident[6];
parse[{a_Name | a_Noun, b_Asgn, c_Conj | c_Adv | c_Verb | c_Noun, d_, rest___}] := is[7];
parse[{a_Lpar, b_Conj | b_Adv | b_Verb | b_Noun, c_Rpar, d_, rest___}] := paren[8];
parse[l__] := move[9];
```

Evaluation History as Text

```
3 * {5, 5} $ + / \ , 1 + i. {5, 5}
3 * {5, 5} $ + / \ , 1 + << i. {5, 5} >>
3 * {5, 5} $ + / \ , 1 + {{0, 1, 2, 3, 4}, {5, 6, 7, 8, 9}, {10, 11, 12, 13, 14}, {15, 16, 17, 18, 19}, {20, 21, 22, 23, 24}}
3 * {5, 5} $ + / \ , << 1 + {{0, 1, 2, 3, 4}, {5, 6, 7, 8, 9}, {10, 11, 12, 13, 14}, {15, 16, 17, 18, 19}, {20, 21, 22, 23, 24}} >>
3 * {5, 5} $ + / \ , {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}, {16, 17, 18, 19, 20}, {21, 22, 23, 24, 25}}
3 * {5, 5} $ << + / >> \ , {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}, {16, 17, 18, 19, 20}, {21, 22, 23, 24, 25}}
3 * {5, 5} $ Σ \ , {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}, {16, 17, 18, 19, 20}, {21, 22, 23, 24, 25}}
3 * {5, 5} $ << Σ \ >> , {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}, {16, 17, 18, 19, 20}, {21, 22, 23, 24, 25}}
3 * {5, 5} $ →Σ , {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}, {16, 17, 18, 19, 20}, {21, 22, 23, 24, 25}}
3 * {5, 5} $ →Σ << , {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}, {16, 17, 18, 19, 20}, {21, 22, 23, 24, 25}} >>
3 * {5, 5} $ →Σ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25}
3 * {5, 5} $ << →Σ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25} >>
3 * {5, 5} $ {1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, 190, 210, 231, 253, 276, 300, 325}
3 * << {5, 5} $ {1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, 190, 210, 231, 253, 276, 300, 325} >>
3 * {{1, 3, 6, 10, 15}, {21, 28, 36, 45, 55}, {66, 78, 91, 105, 120}, {136, 153, 171, 190, 210}, {231, 253, 276, 300, 325}}
<< 3 * {{1, 3, 6, 10, 15}, {21, 28, 36, 45, 55}, {66, 78, 91, 105, 120}, {136, 153, 171, 190, 210}, {231, 253, 276, 300, 325}} >>
{{3, 9, 18, 30, 45}, {63, 84, 108, 135, 165}, {198, 234, 273, 315, 360}, {408, 459, 513, 570, 630}, {693, 759, 828, 900, 975}}
```