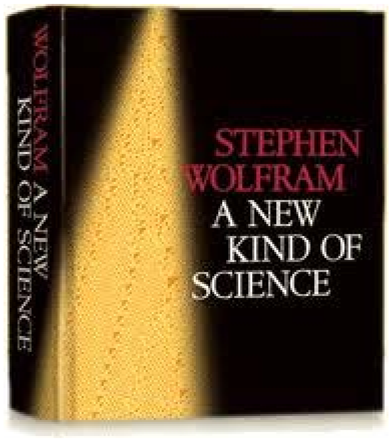
A New Kind of Language



Victor Adamchik

http://www.cs.cmu.edu/~adamchik

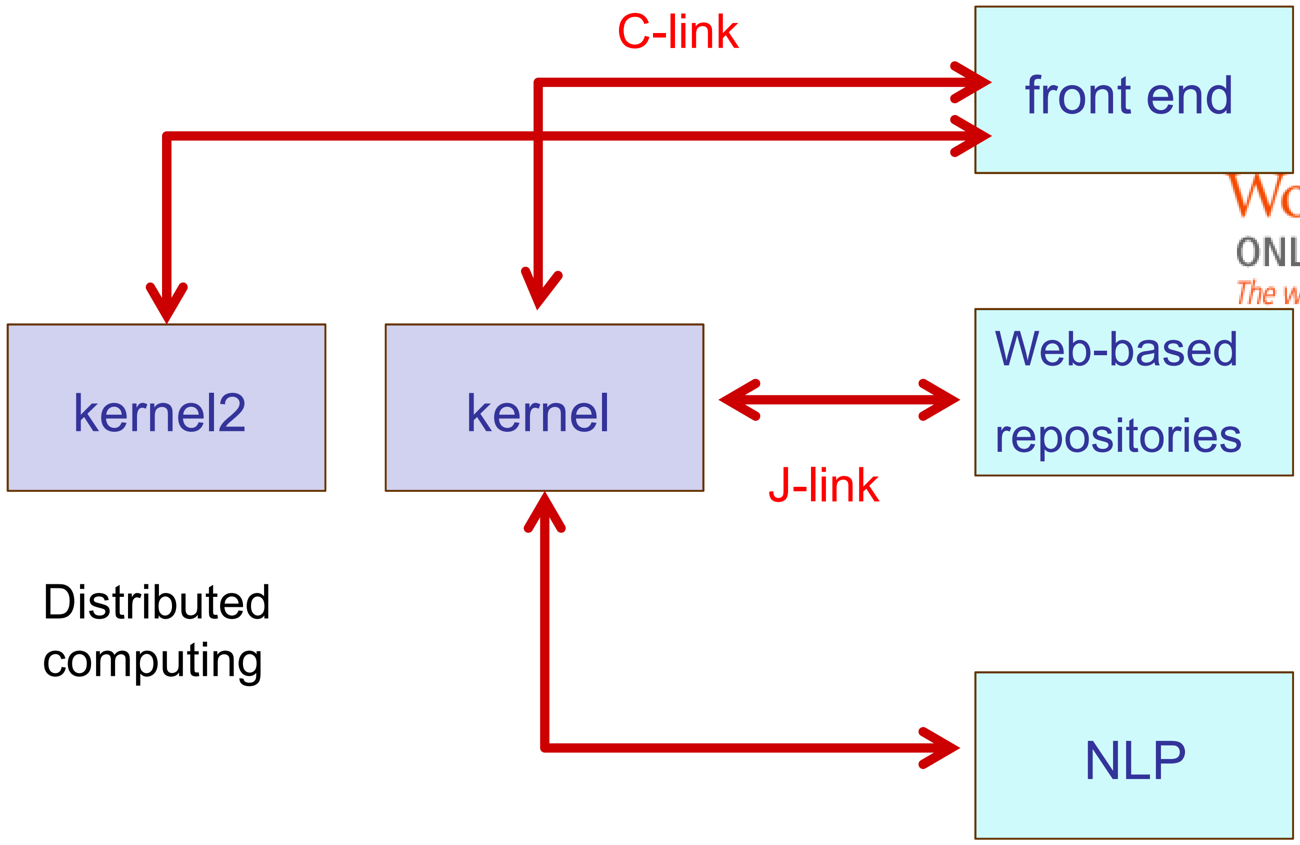
Presentation plan

Language + Libraries + Compiler

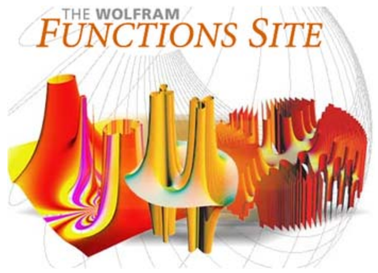
In WL they are all bundled together…

- Architecture
- Expressions
- Evaluator
- Pattern Matching

Architecture



Wolfram Mathematica
ONLINE INTEGRATOR
The world's only full-power integration solver

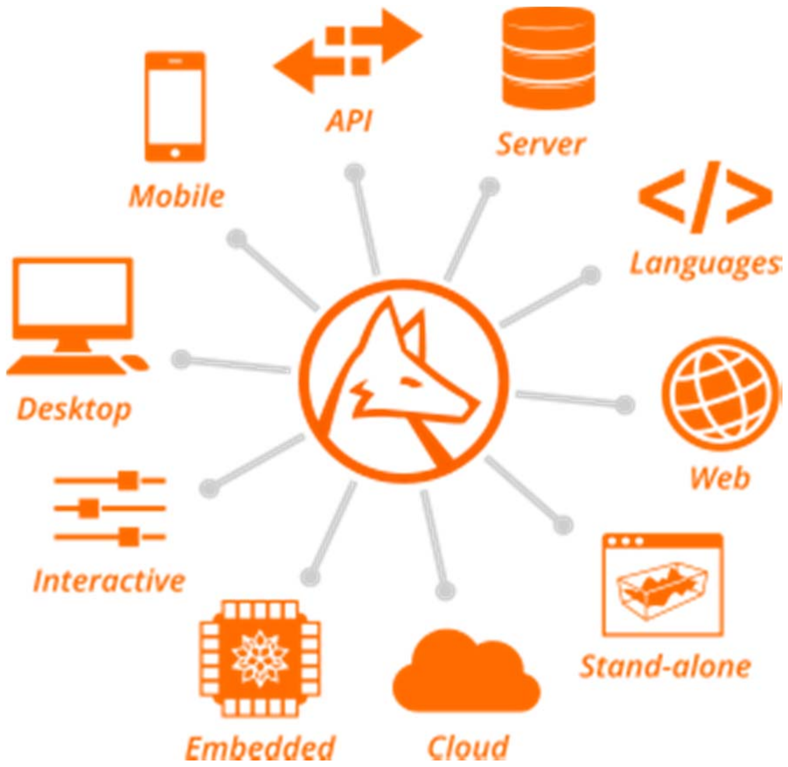


WolframAlpha computational knowledge engine

Distributed computing

The kernel and the front end are separate programs, and each can be used without of the other.

TWL - another kind of language, a knowledge-based language



Create your program in the Wolfram Language

and deploy it everywhere.

Everything is an expression

The basic data objects used in *Mathematica are called "expressions".*

Expressions can be classified as either "atomic" expressions or "compound" expressions.

The exact internal structure of an expression depends on whether the expression is a normal expression, a symbol, a number, or a string.

Expression Structure

A Lisp-like structure - a list (head arg1 … argn)

In WL head[arg1, ..., argn], like Plus[2, 2]

Each element can be accessed directly via Part (via BFS ordering), head is at index [[0]]

Function definition is a pattern-based.

```
f[x_] := x^2;

f[x_Integer] := x^3;
```

How expressions are evaluated?

WL is a term rewriting system - whenever an expression is entered, it is evaluated by using rewrite rules.

It is necessary to understand the order in which the various parts of an expression are evaluated by term rewriting.

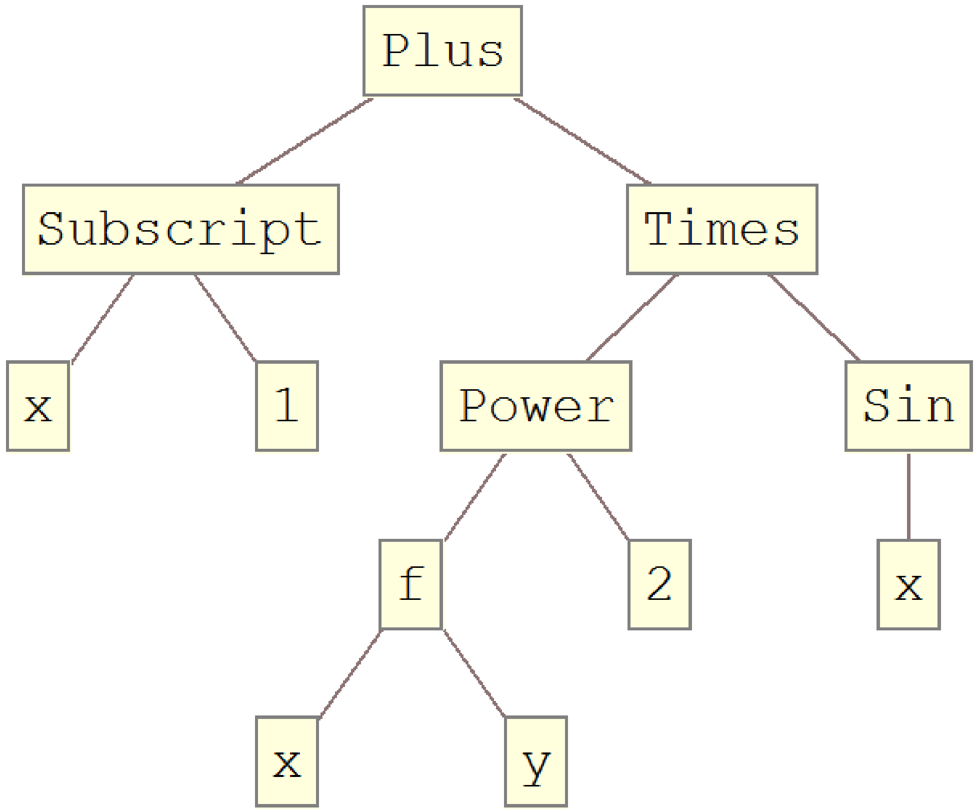
Transformation rules are defined in WL and stored as expressions.

For example, `x=5` is `{HoldPattern[x] :> 5}`

`f[x_,y_]:=x+y` is `{HoldPattern[f[x_,y_]] :> x + y}`

x_1 + Sin[x] * f[x, y]^2

Symbol x might have early defined values, so those rules are stored at OwnValues[x].



The symbols could be heads (parents) of some expressions.

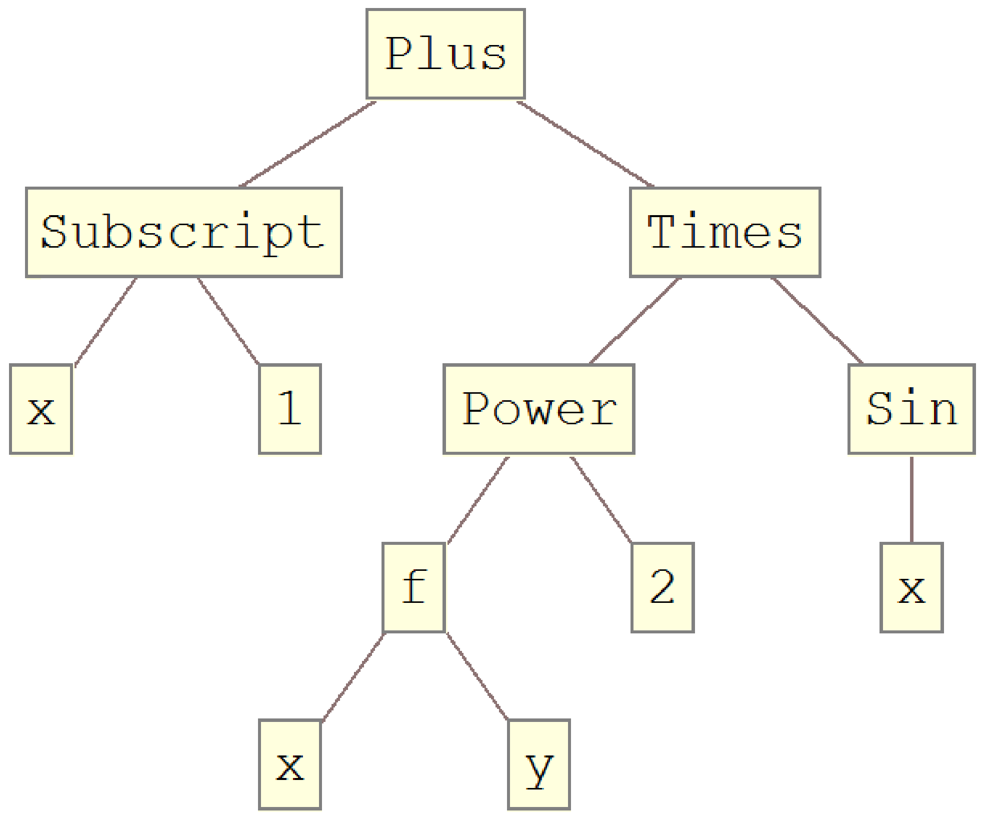
```
g[x_]:=x^2;

g = h;

g[5]
```

Definition for g had no chance to execute.

So g[5] returns h[5].

x_1 + Sin[x] * f[x, y]^2

The parent (head) might have early defined rules,
f[x_, y_] := x + y
so those rules are stored at DownValues[f]. This rule is only applied when encountered with arguments *downwards.*



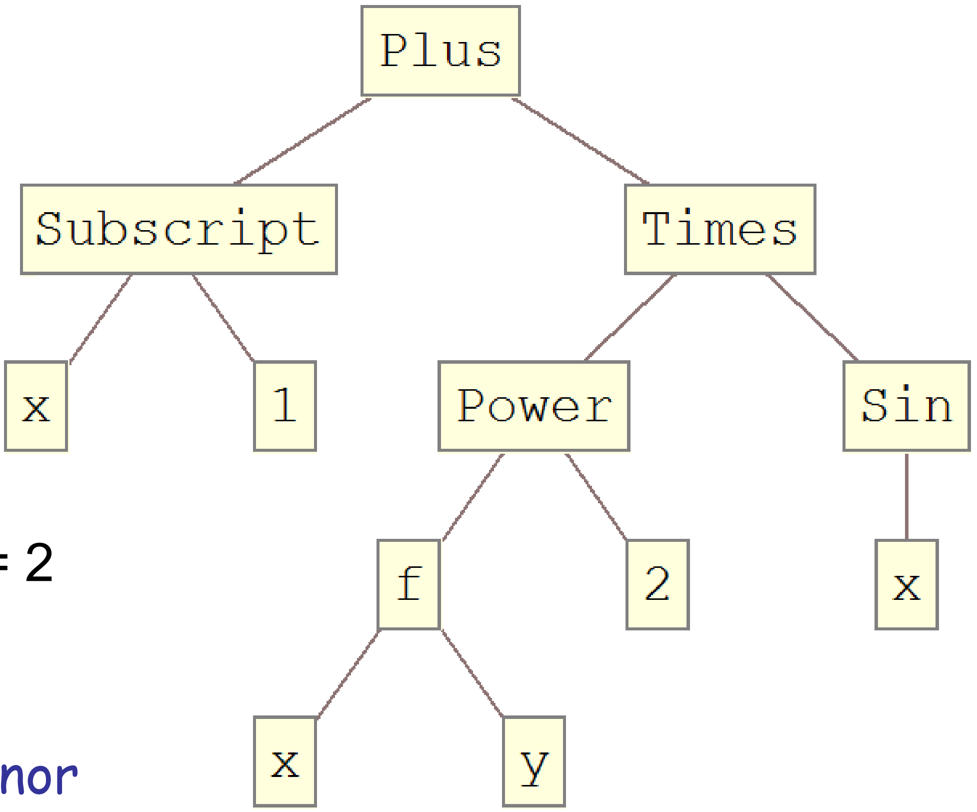
A function might have several rules.

```
{HoldPattern[f[2]] :> 11,
 HoldPattern[f[x_, y_]] :> x + y,
 HoldPattern[f[x_]] :> x^2}
```

The rules will be applied in order – more specific rules first!

x_1 + Sin[x] * f[x, y]^2

Sometimes you want to assigned rules to a symbol only if it has a special case. For example, $x_1 = 2$. This type of a pattern is handled by UpValues.



In WL this is x /: Subscript[x,1] := 2

The rule is NOT assigned to x nor Subscript.

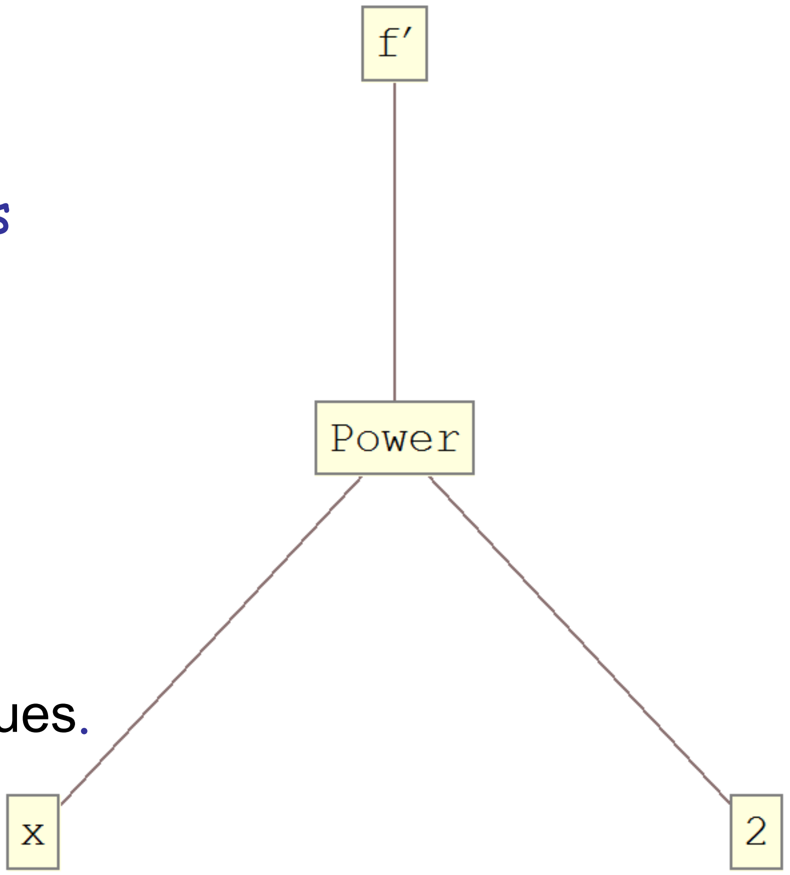
SubValues: Derivative[1][f][x^2]

We cannot assign a rule to f′ = D[f] , it’s not a symbol.

We cannot assign a rule to f′ [Power], it’s math wrong.

This defines neither an OwnValue nor a DownValue …

Such rules are applied through SubValues.



Its behavior is not well documented.

DownValues and OwnValues are applied before SubValues.

f [g [x]]

UpValues for g are applied before DownValues for f

This does not mean that UpValues for g will be applied before DownValues g.

All these are not well documented....

But the picture is even more complicated....

Controlling the evaluation

I should point out that the user can (to some extent) control the evaluation process.

I indicate functions that can be used:

Hold, HoldAll, HoldAllComplete, HoldComplete, HoldFirst, HoldRest, HoldPattern, ReleaseHold, SequenceHold, Evaluate, Unevaluated,

Inactive, Activate

It should be clear by now that using Values (4) with Hold (10) attributes creates a million combinations which is impossible to describe and therefore document.

The Evaluation Process

Evaluate symbols with OwnValues

Evaluate the head (with OwnValues)

Evaluate the arguments from left to right.

If the head has Hold-attributes, do not evaluate arguments.

Apply UpValues for arguments

Apply SubValues

Evaluate the head with arguments. (DownValue)

Evaluate the resulting expression

Again, the process is not documented, so I could be wrong...

Pattern Matching

A single blank is used to represent an individual expression, which can be any data object.

```
MatchQ[x^2,_]

MatchQ[x^2,x^_]

MatchQ[x^2,x^_Integer]

MatchQ[x^2,_Power]

MatchQ[x^2,_^_]

MatchQ[x^2,_^2]
```

Alternative Pattern Matching

A | specifies alternative patterns

```
f[a_Real | a_Integer]:= a - 1


MatchQ[x^2,{_}| _^2]
```

Pure Functions

```
square[x_]:= x^2
```

is the same as

```
Function[#^2]
```

is the same as

```
(#^2)&
```

is the same as

```
Function[x, x^2]
```

Thus, `(#^2)&[5]` is **25**

What is `(#^3)&[(#+2)&[3]]` ? 125

What is `(#[[1]]^#[[2]])&[{2,3}]` ?? 8

Conditional Pattern Matching

making it contingent upon meeting certain conditions

```
f[x_?EvenQ]:= x;
f[x_?OddQ]:= x^2;
f[x_]/;Element[x, Reals] && x > 1 := 1/x

MatchQ[2,_?(#>3&)]
MatchQ[2,_Integer ?(#>3&)]


MatchQ[a^b, _^y_/;Head[y] == Symbol]
```

Pattern Matching Complexity

Bubble sort:

```
sort[xs___,x_,y_,ys___]:= sort[xs,y,x,ys]/;x > y
```

We have no idea how efficient Mathematica's pattern matching...

```
bar[a_ * b_, x_] := bar[b, x] /; FreeQ[a, x]
```

The above could be very expensive. For example

`bar[a*b*c*d*e*f*x, x]` has a linear complexity

Higher-order Functions

`Apply, Map, MapThread, Nest,`

`NestList, Fold, FoldList, FixedPoint,`

`FixedPointList, Inner, Outer`

`Fold[#1^#2&, x, {a, b, c, d}]` is $(((x^a)^b)^c)$

`FoldList[#1^#2&,x,{a,b,c,d}]`

gives
$\{x, x^a, (x\wedge a)^b, ((x\wedge a)\wedge b)^c, (((x\wedge a)\wedge b)\wedge c)^d\}$

What is **`Fold[1/(#1+#2)&,x,{1,1,1,1}]`** ?

CF: 1/(1+1/(1+1/(1+x)

Mandelbrot Set

The Mandelbrot set is the set of all complex numbers **c** for which sequence defined by the iteration

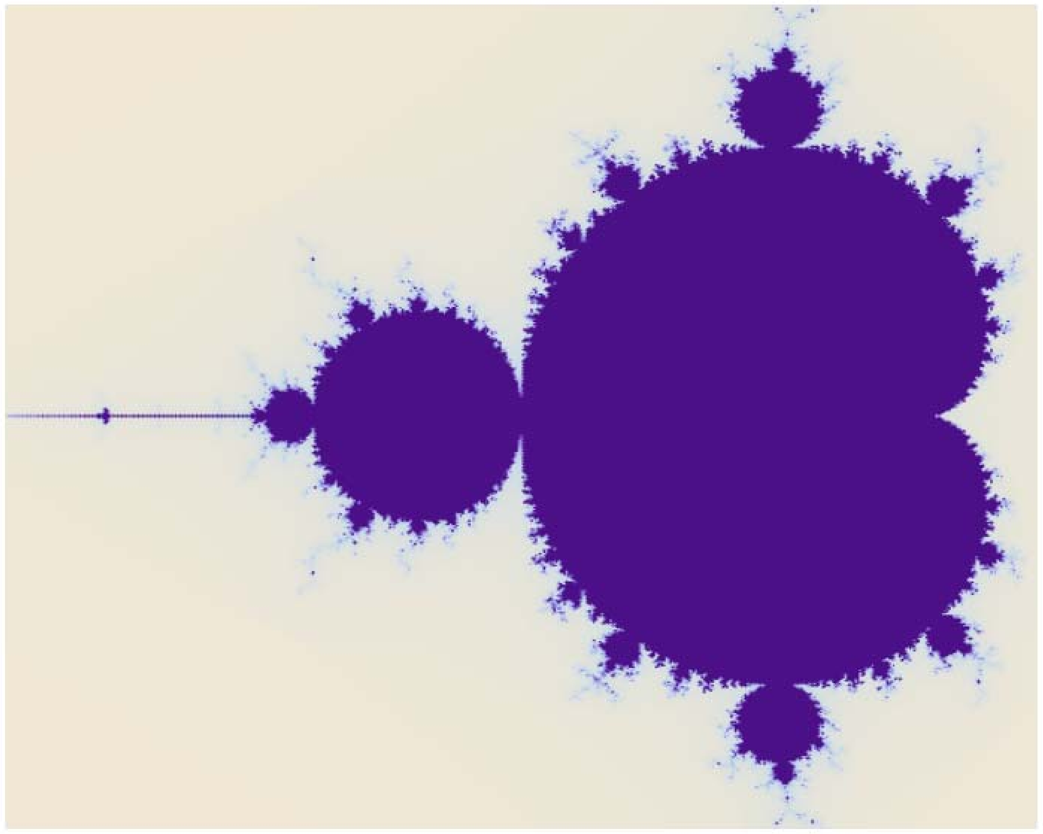
```
f(n+1) = f(n)²+c, f(0) = c
```

remains bounded.

```
FixedPoint[#² + c &, c]

FixedPoint[#² + c &, c,
            SameTest->(Abs[#2-#1] > 10&)]

Mandelbrot[x_]:=
Length[FixedPointList[#² + c &, c, 80,
            SameTest->(Abs[#2-#1] > 10&)]]
```

Mandelbrot Set

```
DensityPlot[-Mandelbrot[x + y I],
{x,-2,0.5},{y,-1,1}, Mesh->False,
Frame->False, AspectRatio->Automatic,
PlotPoints->125];
```



Programming Styles

Procedural Programming

Functional Programming

Rule-Based Programming

Dynamic Programming

Compute n!

Procedural

```
procedural[x_] :=
Module[{prod = 1,ind = 1},
  If[!(IntegerQ[x] && Positive[x]), Return[]];
  While[ind <= x, prod *= ind; ind++];
  Return[prod]
]
```

Recursive

```
rec[1] = 1
rec[x_Integer?Positive] := x rec[x-1]
```

Functional

```
functional[x_Integer?Positive] := Times @@ Range[x]
```

This is the fastest...

Internal Hashing

```
dp[x_Integer?Positive] := dp[x] = Times @@ Range[x]
```

All intermediate results are stored…

Numeric Computations

`N[Pi]` – default machine precision ($MachinePrecision)

`N[Pi, 5000]` – arbitrary precision

Machine numbers work by making direct use of the numerical capabilities of your underlying computer system.

Arbitrary precision computations are based on GMP.

GMP is a free GNU library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers with no practical limit to the precision except the ones implied by the available memory.