Good Ideas in Programming Languages

Dmitri Nesteruk

@dnesteruk

What this talk is about

Programming language syntax

Focus on ‘core’ OOP languages
(C++, Java, C#)

And ‘other’ languages
(Kotlin, Rust, MATLAB, etc.)

Thoughts on strictly theoretical language features

How Integral Types Evolved

C/C++

- “Type sizes should be dependent on the platform”
- Used ‘char’ instead of byte for this exact reason ☺
- What is the difference between short, short int, long int, long, int?
- Difference between int and long is between 0 and 4 bytes

Java and C# made things deterministic

- Byte = 8 bits, short = 16 bits, int = 32 bits, long = 64 bits

C++ actually made deterministic types

- <cstdint> int8_t, uint32_t, etc.
- int = signed, uint = unsigned

BTW, float and double are standardized (phew!)

Shorter Integral Type names

Size (# of bytes used for type name) matters

Rust introduced shorter notation: u8, i32, etc.

Also introduced f32, f64

Also used as postfix if type inference is being used
var z = 3.0f32; // instead of 3.f

int/uint (or isize/usize) for machine-native integral type

Should array indices be signed (int)?

- In C++, array x[i] is same as *(x+i)
- In D, array indices are *strictly* unsigned

Other integral types

Rust's notation allows non-byte-aligned vars

- VHDL/Verilog allow precise specification of # of bits
- x : bit_vector(5 downto 0)
- u6 can mean 6-bit unsigned

Can extend this type system for SIMD

- C++ has __int128?
- We can extend it to, e.g., p128 (p = packed)
- Can introduce operator support (x+y)

Base 10 Floating Point

Exemplified by .NET's System.Decimal

Larger precision, smaller range of exponents

More deterministic operations

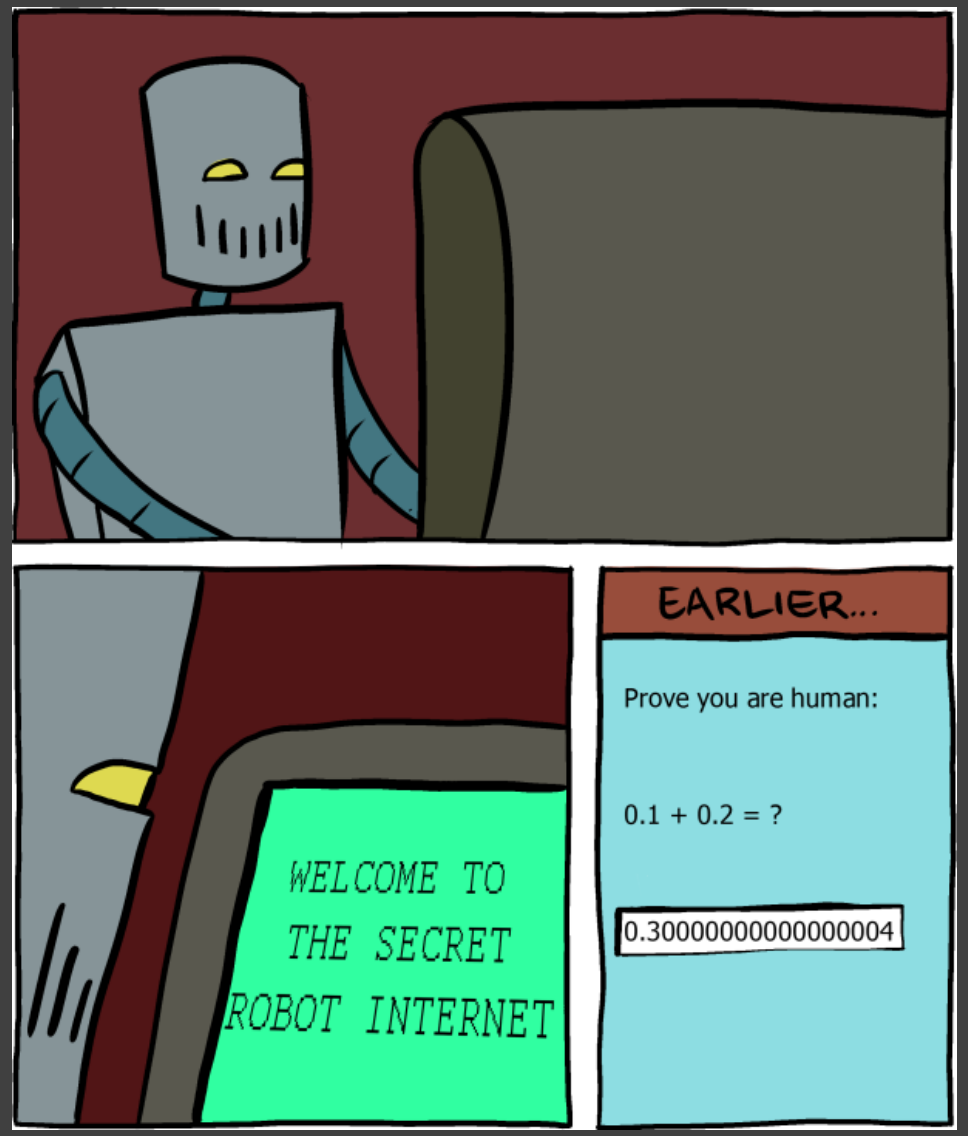
- 0.1+0.2 is finally 0.3
- Textual output represents actual number

128-bit number

- 96-bit mantissa
- 32-bit sign+exponent

Operators exist for integral-decimal ops but not float-decimal

Special literal required (1.5m or M)



Arrays

Arrays in OOP languages *suck*

- C++ is ruinous for 2D
- C#: `int [,] x = new int[2,2] { { 1,2 }, {3, 4} };`
- MATLAB: `x = [1 2; 3 4]`

MATLAB (CAS, short for MATrix LABoratory) does them best

- *Everything* is treated as an array
- Scalars are 1x1 arrays
- Dynamically expandable in size & dimensions
- Composable: `Z = [A B]`
- Built-in operators: `X'` transposes, `X^-1` inverts
- N-D matrix is 1D addressable

MATLAB Array Operators

Elementwise operators

```
.+  .-  .*  ./  .^
```

Scalar-matrix product

```
c * A
```

Matrix product

```
A * B
```

Miscellanea

- `A \ B` solves the system Ax = B

Matrix Product

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

Hadamard Product

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \odot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae & bf \\ cg & dh \end{pmatrix}$$

Implicit product operator

Implicit *

3x // same as 3*x

- Needs to be disambiguated with
  - Type decorators (3f32)
  - Units of measure decorators

Prime mark strictly for transpose

- `X' * Y`
- does not compute `X'`, just generate different * code

Sensible Operators

:= is definition, <- and -> assignment, = equality only

- =:= binding operator
- =8= with FP tolerance (maybe requires front-end)

< and > are strictly ordering operators (no template/generic stuff)

^ is exponentiation (alternatively **, XOR is xor)

Multi-level bracketing for readability

```
x = {[(2+3)-4]/2};
```

- Too bad [] has been hijacked for arrays ☹
- Typical math notation is x_{ij}

Need to reconcile with other types of brackets

Height matters too!

Built-In Array/Matrix Support

Whitespace-significant initialization

```
X = [1 2; 3 4]; // instead of {{1,2},{3,4}
X = [ 1 2
      3 4 ]; // also works, kind of
```

Sensible dot/cross product

```
X * Y; // matrix product, assuming dimensions match
X .* Y; // elementwise (Hadamard) product
```

Array-bound operations should use SIMD and parallelism by default instead of forcing us to decorate `for` loops

Masks for definitely-zero elements (reduces # of ops) (JAI)

```
Y = mtx([1 1; 0 1], mask);
```

Static *and* dynamic sizes

Implicit Lambda Arguments and Lambda Braces

Items.Select(x => Process(x)); // x=>x is even worse

Items.Aggregate((k,v) => k + v);

Implicit single-argument name (e.g., it):
Items.Select(Process(it)); // it = first argument to lambda (Kotlin)

Implicit names for multiple arguments:
Items.Aggregate($1 + $2); // should that be $0?

Loss of distinction between function and expression mitigated with brace equivalence
Items.Aggregate{$1 + $2}; // argument is a function

Allows for DSL construction

Lambda Maturation Cycle (JAI)

```
                                { ... } anonymous block
      (x:i32) -> float { ... } anonymous function
f :=  (x:i32) -> float { ... } variable
f :=  (x:i32) -> float { ... } member or global
```

Sensible tuples

```
fn sum_and_product(x,y) -> (sum,product)
{
  return (x+y, x*y); // type inference here
}
let s,p = sum_and_product(2,3);
let z = sum_and_product(4,5);
print(z.sum + ", " + z.2); // simpler ordinal indexing
```

We are getting them in C#!

Pattern-matching

- case (int x, _) =>

Data Class (Kotlin)

```
data class Person(var name:String, var age:Int)
```

Properties & constructor

`equals()/hashCode()`

`toString()`

Components/destructuring:

```
val (name, age) = john
```

Copy function:

```
val john = Person("John", 23)
val jane = john.copy(name = "Jane")
```

Enum-as-Class

What is the difference between

```
enum Foo { Bar }
```

and

```
struct Bar; ?
```

Member grouping (Tlön)

```
class Foo
{
  group names
  {
    first, middle?, last : string
  }
}
```

Foo has members first, middle, and last

Those same members also exposed as a `names` enumeration

Can be emulated with array-backed properties

SOA/AOS (JAI)

```
struct Foo SOA
{
  int x;
  bool b;
}
Foo foos[128];
```

Spatial locality: better to store `int[128]` followed by `bool[128]`

Non-Keyboard Symbols

High-DPI (4K+) displays mean all symbols are readable (more or less)

Better comparisons: ≠ ≤ ≥

Products: · × ⊙

Superscript-as-power: x^2+y^3

Logical operations: ∧ ∨

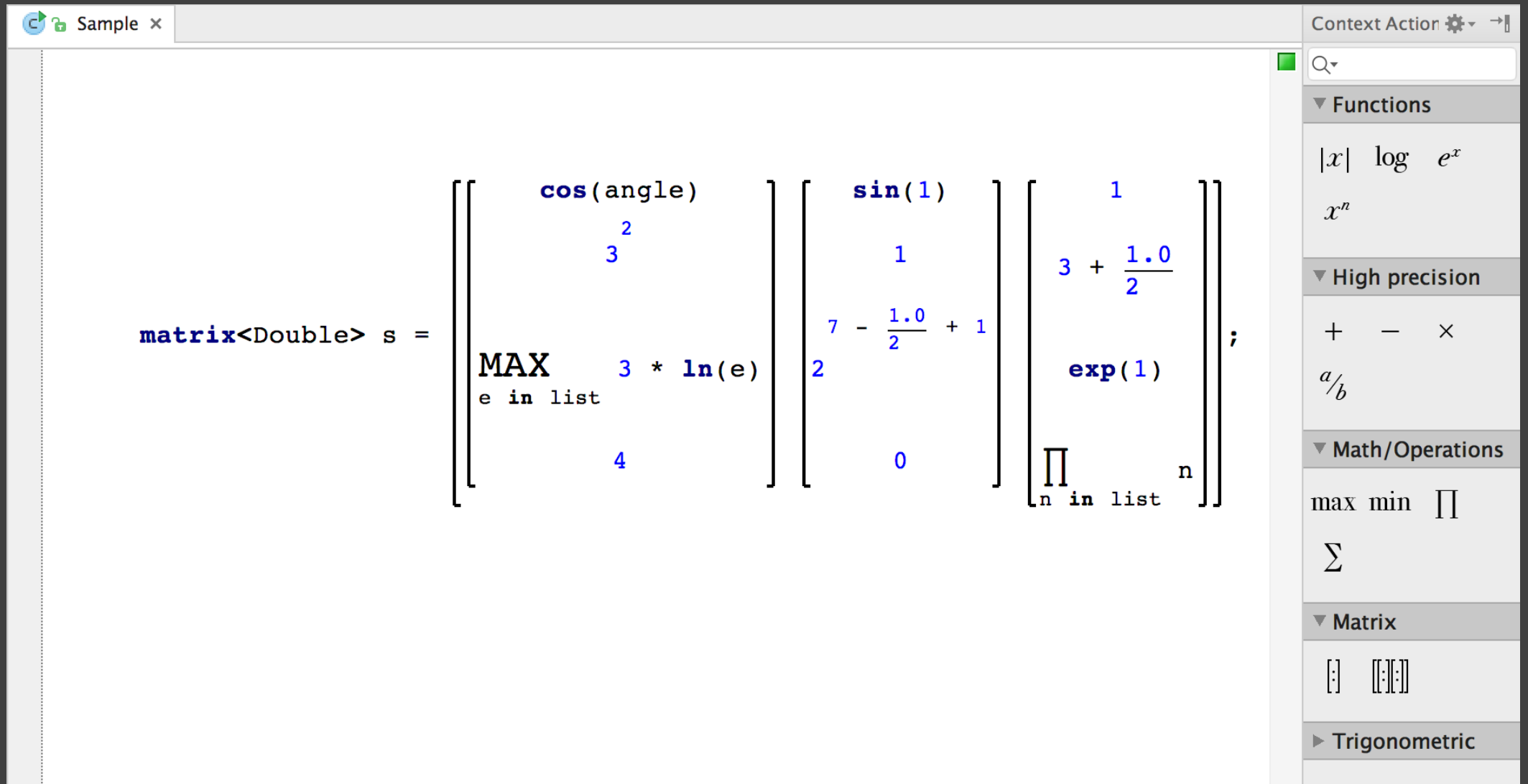
Set Operations

```
x.Contains(y)
!x.Contains(y)
x.Count == 0
x.All(e => y.Contains(e))
x.Remove(e => y.Contains(e))
```

```
y ∈ x or x ∋ y
y ∉ x
x = ∅
y ⊂ x
x \= y

∀x : {x ∈ items | x > 0}
{
  print(x)
}
```

Projectional Editors (MPS)



Conclusions

Plenty of innovations in syntax space

Keyboard restrictions are arbitrary

High-DPI screens render everything well

Plain-text code representation: outdated?

That's it!

Questions? Answers? Hate mail! @dnesteruk

Rust: http://bit.ly/2pPg3T5

Kotlin: http://bit.ly/2sdauS8

D: http://bit.ly/2sF2gmY

MATLAB: http://bit.ly/2sdjpCO

JAI (Jonathan Blow): https://www.youtube.com/user/jblow888

Tlön: https://github.com/nesteruk/tlon