

# Rogue Context Operator

Abe Pralle  
August 1, 2018

[abepralle.github.io/Presentations/Rogue\\_Context\\_Operator.pdf](https://abepralle.github.io/Presentations/Rogue_Context_Operator.pdf)

# Rogue Language Designers

## Abe Pralle

- Created Rogue language in 2015
- Ad tech developer at AppOnboard
- Indie game developer (Runegate, Plasmaworks)

## Programming Interests

- Games, languages, APIs

## Contact

- [abe.pralle@gmail.com](mailto:abe.pralle@gmail.com)

## Install Rogue

- Repo with installation instructions:  
[github.com/AbePralle/Rogue](https://github.com/AbePralle/Rogue)
- Windows, macOS, Linux



# Rogue Language Designers

## Murphy McCauley

- Frequent collaborator & consultant
- Joined Rogue project in 2016
- Ph.D. student at Berkeley (Computer Science, 2018)
- Programmed "SENSE" packet-level network simulator in Rogue and Python



## Major Contributions to Rogue

Automatic garbage collection, multithreading, two Python extension generators, tuples, method template type inference by parameter types, core Windows compatibility, the *Rose* prototype language, and the book-in-progress "Hacking with Rogue"

# Call Chaining is a Total Hack

## Call Chaining is Great

- Many APIs support call chaining
- Rogue has some nice convenience syntax to support call chaining ("method x()->this")
- Nice to be able to make several calls in one expression

## Call Chaining is a Total Hack

- Call chaining is a hacky way to support calling multiple unrelated methods on a single object context
- Method return values should not be co-opted by a style of method invocation
- Only methods that do not otherwise need to return a value can be adapted for call chaining
- Likewise, useful optional return values are not possible if a method has been written to support call chaining
- Call chaining is a semantic fix for a syntactic problem

# Analogy: Recursion in FORTRAN 77

## Recursive Binary Tree Print in Rogue

```
method display( n:Node )
```

```
  if (n) display( n.left ); println n.value; display( n.right )
```

## Recursive Binary Tree Print in FORTRAN 77

- A function directly calling itself generates a compiler error
- All local variables are like static variables (some compilers)
- Rogue program using F77-style recursion semantics:

```
method display_recurse( n:Node ): display( n )
method display( n:Node )
  if (not n) return
  stack.add( n ); display_recurse( n.left ); n = stack.remove_last
  println n.value; display_recurse( n.right )
```
- Question: can we say that FORTRAN 77 supports recursion?
- Or is this explicit stack-based, pointer-based mess a kludge to simulate the recursion other languages support directly?
- What could a language support directly vs kludgy call chaining?

# Context Operator

## Syntax

- `object.[ method1, method2, property=value, ... ]`
- Operator calls all methods and assigns all properties in sequence and then produces 'object' as expression result

## Old Code With Call Chaining

```
local list = Int32[].add( x ).add( y ).add( z )
println list
list.remove_last
println list.sort( (a,b)=>(a<b) )
```

## New Code With Context Operator

```
local list = Int32[][ add(x), add(y), add(z) ]
println list
println list.[ remove_last, sort( (a,b)=>(a<b) ) ]
```