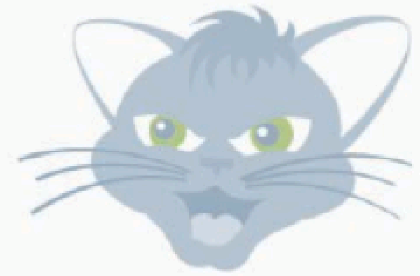
CAT
Programming Language

An Optimization Framework for the MSIL

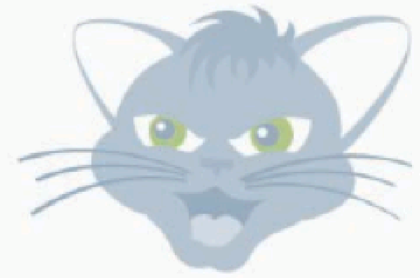
About this Presentation



No silver bullets, but perhaps a fresh view point which may stimulate discussion.

Far too much information for 30 minutes, so brace yourselves, this is going to go quickly.

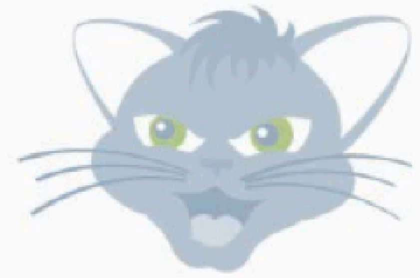
What is Cat?



Cat is a stack-based functional programming language.

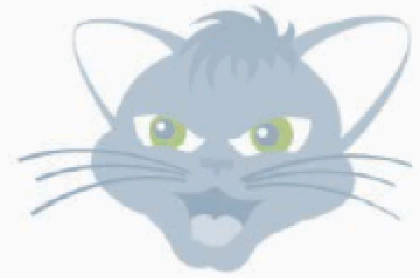
Designed to be an intermediate language for translation and optimization.

Motivation



I designed a multi-paradigm language (Heron). I needed the following:

- an optimizer
- to translate to MSIL, and other targets
- a flexible and powerful macro language

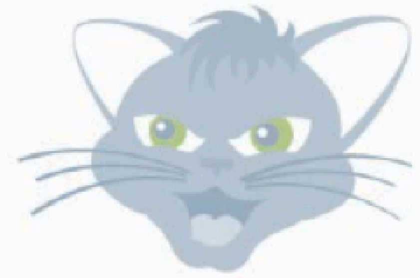


Rejection of Prior Art

- Too low-level
- Miss big opportunities for functional optimizations
- They generally do a poor job.
- Not well suited for parallelization

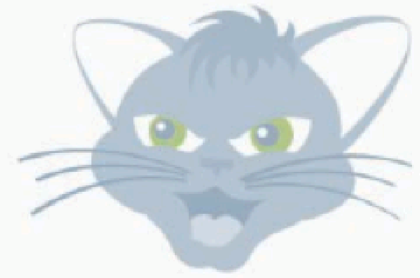
Cat Programming Language

Desirable Features



- Portable
- Easily manipulated
- Easily translated to
- Easily translated from
- Strongly typed
- Easy to code against
- Multi-paradigm

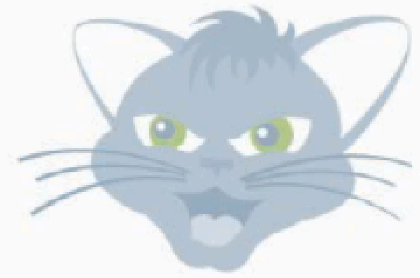
Observations



- Many imperative patterns can benefit from being rewritten as functional ones
- Lack of functional lists in low-level representations make fusion optimizations very hard.
- Pure functions are more easily parallelized.
- It is easier to manipulate code that has no variables or parameters.

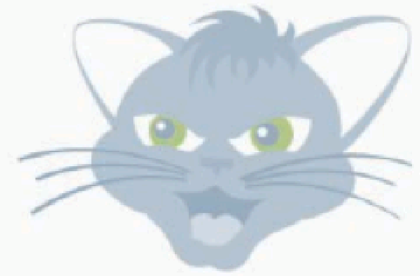
Cat Programming Language

What does Cat offer?



- No variable declarations
- No argument declarations.
- High-level functional primitives
- Static typing
- Pattern rewriting language
- Simple syntax
- Simple semantics

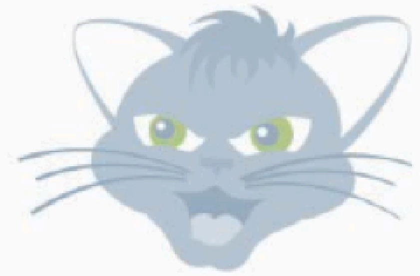
Intended for Human Use



Unlike other low-level languages/ representations Cat is actually intended for human use.

Advantages: easier to investigate, extend, and customize.

Cat Influences



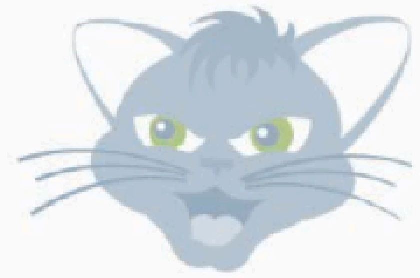
Two major influences: Joy and MSIL.

Joy is a simple language based on the concatenation of functions and combinators. Inspired by Forth and FP.

Countless other languages influenced the design of Cat.

Cat Programming Language

Compositional Language



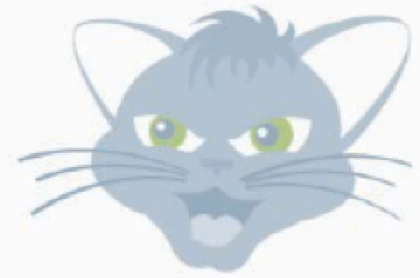
I like to describe Cat as a multi-paradigm compositional language.

Many functional and imperative languages could be viewed as examples of applicative languages.

Some would call Cat a concatenative language.

Cat Programming Language

Function Composition



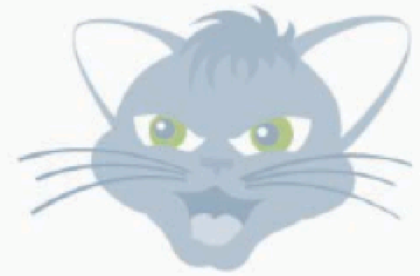
Cat is based on the composition of functions.

```
f g
```

Has the following mathematical meaning:

$g \circ f$

Syntax

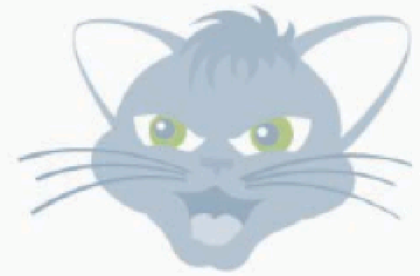


Cat uses postfix notation:

```
{ 1 2 3 * + }
```

Braces indicate scope, and are used for grouping.

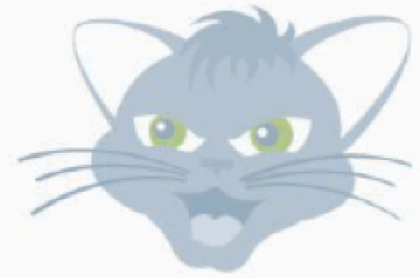
Subdividing a Cat Program



```
{ f g h }
= { f { g h } }
= { { f g } h }
= { { f } { g } { h } }
```

More on why this is important later.

Cat Stacks

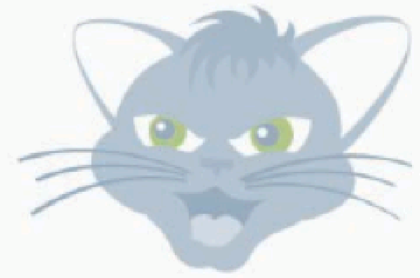


There are two stacks in Cat: the primary stack and the auxiliary stack.

This is like MSIL. The auxiliary stack is essentially the list of locals. The main stack is the evaluation/call stack.

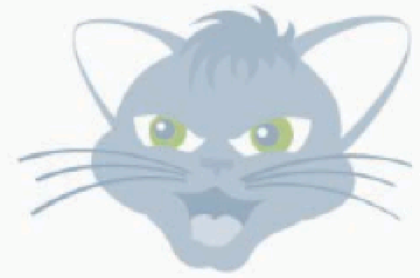
Cat Programming Language

About Cat Programs



A Cat program is the concatenation of zero or more atomic programs and user-defined programs.

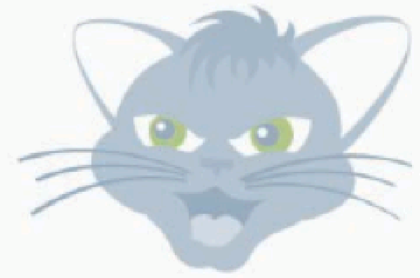
Programs as Functions



Every Cat program can be viewed as a function which maps a pair of stacks onto another pair of stacks.

Only *pure* Cat functions are actual mathematical functions.

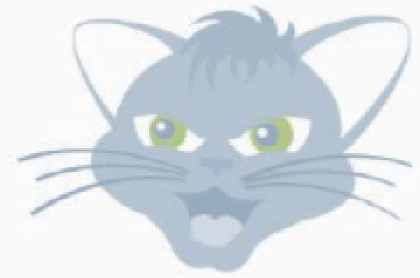
Consumption



The elements on the top of a stack that a Cat program reads, is the consumption of the program.

A Cat program must consume a constant number of elements of fixed type.

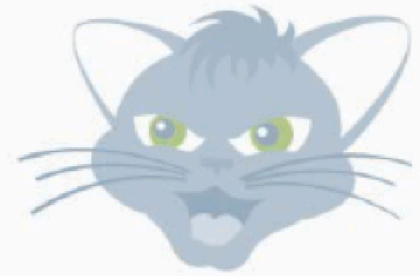
No Peeking



A Cat program can never access elements below the number of values it consumes, by definition.

Cat Programming Language

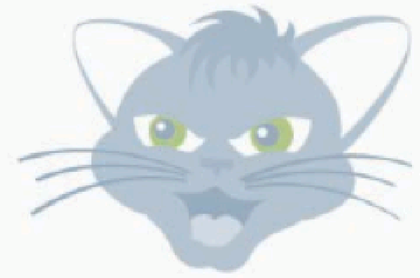
Production



The elements on the top of a stack that a Cat program leaves, is the consumption of the program.

A Cat program must produce a constant number of elements of fixed type.

Cat Programming Language

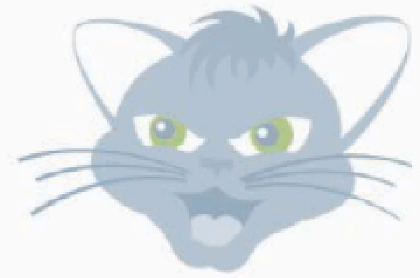


Valence

The valence is the number of elements produced minus the number of element consumed.

Imperative control structures often have a valence of zero.

Cat Programming Language

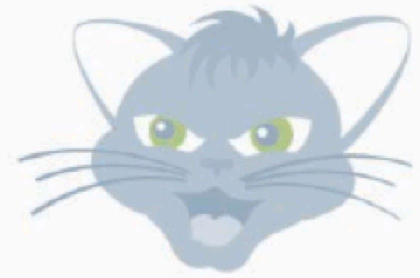


Example

add_int consumes two integers, and produces one integer. It has a valence of -1.

swap consumes two values, and produces two values. It has a valence of 0.

Cat Programming Language

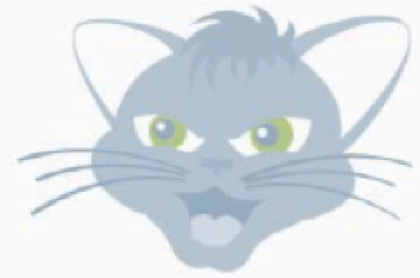


Known Stack State

As a result of consumption and production rules:

At any point in a Cat program, the number and type of elements on the stack is knowable at compile-time.

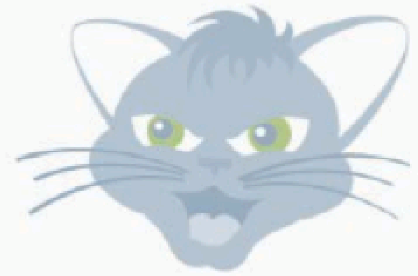
Cat Functions



Most Cat programs only operate on the main stack.

Cat programs that don't consume or produce values on the auxiliary stack are called *functions*. Otherwise they are called *subroutines*.

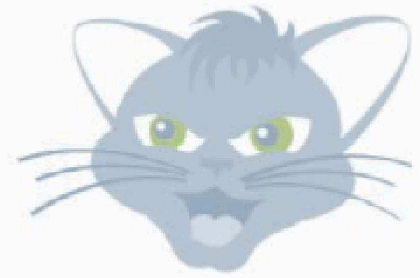
Applicative View of Functions



All Cat functions consume a constant number of stack elements, and produce a constant number of stack elements.

A Cat function can be equated to an imperative / functional (applicative) language function.

Applicative Equivalency



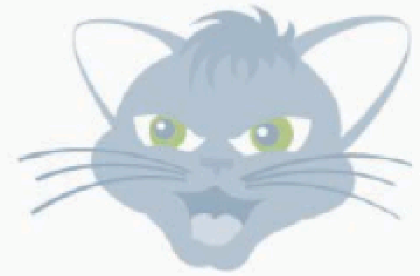
A Cat function can be viewed as a function in an applicative language.

Stack consumption = arguments

Stack production = results

Cat Programming Language

Cat Subroutines

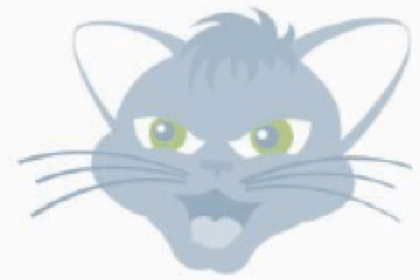


Cat programs which consume arguments or produce results on the auxiliary stack are called subroutines.

Subroutines are usually only used local to a user-defined function.

Cat Programming Language

Atomic Functions

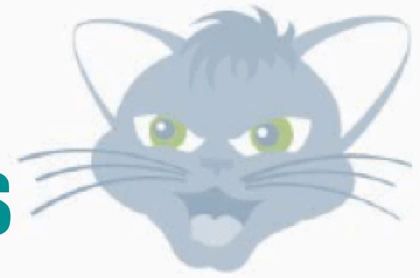


There are several built-in / primitive functions.

These aren't pinned down precisely, nor do I believe should they be.

Ideally Cat primitives are defined by the target, e.g. MSIL.

Stack Manipulation Functions



pop – Remove top item of stack

dup – Duplicate top item of stack

dupd – Duplicate second item on stack

swap – Swap top items of stack

swapd – Swap second and third items of stack

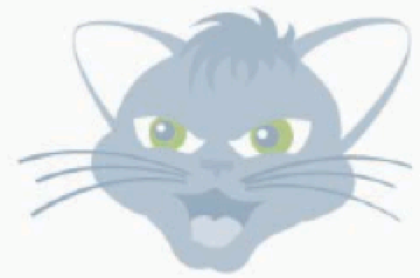
rot3left – place third item on top of stack

rot3right – insert top item in third position

rot4left – place fourth item on top of stack

rot4right – insert top item in fourth position

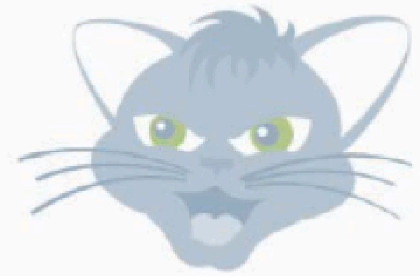
Arithmetic Functions



add – Add top two elements of stack
sub - Subtract top element from second element
mul – Multiply top two elements of stack
div – Divide top element from second element
mod – Modulo top element from second element
divmod – Perform both divide and modulo
inc – Add one to top element
dec – Subtract one from top element
max – Returns the maximum of two values
min – Returns the minimum of two values
minmax – Orders the two elements in ascending order

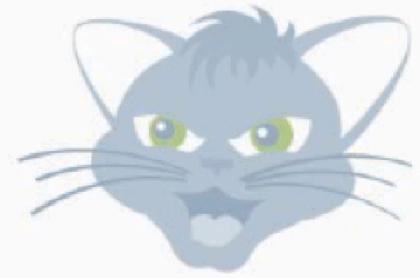
Cat Programming Language

Symbols are Valid Names



```
define + { add }
define - { sub }
define * { mul }
define / { div }
define % { mod }
define /% { divmod }
```

List Functions



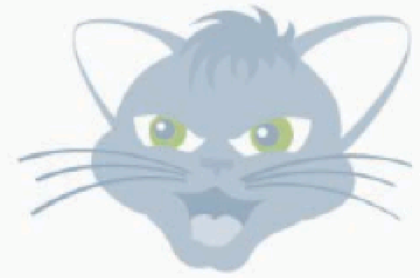
Like most functional languages the standard list manipulation functions are present.

- cons
- uncons
- head
- tail
- rev
- count

etc.

Cat Programming Language

Literals

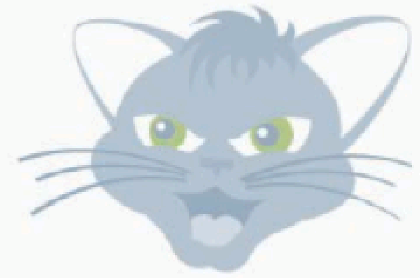


Literals are programs which push values onto the main stack. This has interesting consequences when we talk about lists.

```
42 "Hello" 3.14
```

Cat Programming Language

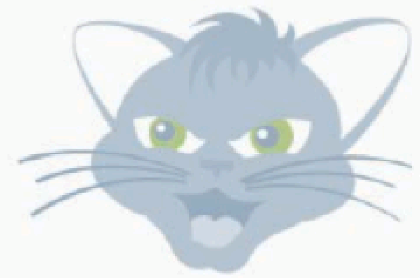
Scope and Namespace



Curly braces used to express scope of name bindings (i.e. definitions) and transforms. A corresponding open and close brace delimits a *block*.

Namespaces are simply named blocks.

Anonymous Functions

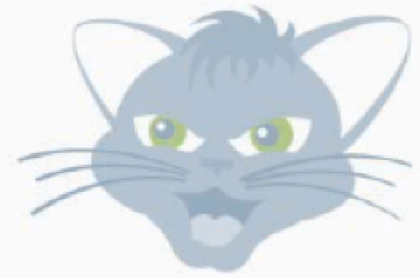


Anonymous functions (a.k.a. quotations), are expressed using square braces. They are pushed onto the stack as data, and can be executed using the i combinator.

```
41 [1 +] i
 = 42
```

Cat Programming Language

Lists



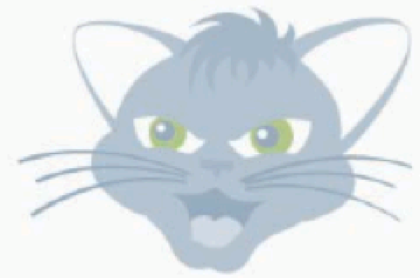
There is no syntax for lists as literals, but you can construct them from anonymous functions:

```
[1 2 3] eval
```

or the shortcut

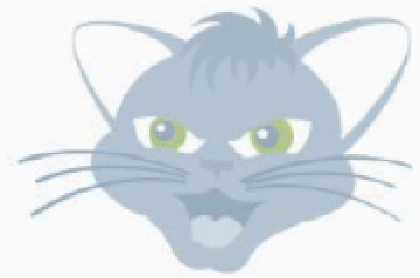
```
[1 2 3] $
```

Eval



Remember [1 2 3] is an anonymous function which consists of the composition of the functions: 1, 2, and 3.

The eval function executes a function, and places produced elements in a list.



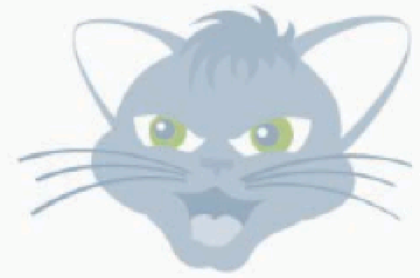
Named Functions

Named functions are defined as follows:

```
define f { 6 7 * }
```

Calling f would leave the value 42 on the stack.

Control Flow



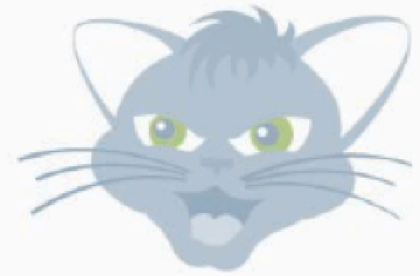
Choices can be made using the if function.

```
{ true [42] [13] if }
```

Looping is also achieved with functions, such as while.

Cat Programming Language

While

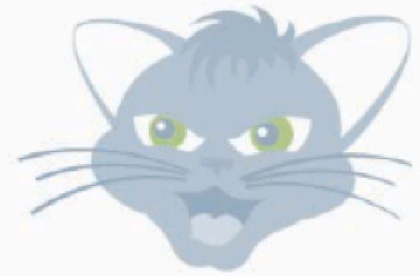


While takes two functions as parameters.

The first function is the loop body. It must have a valence of zero.

The second function is the loop invariant. It must have a net production of precisely a single boolean value.

Cat Programming Language



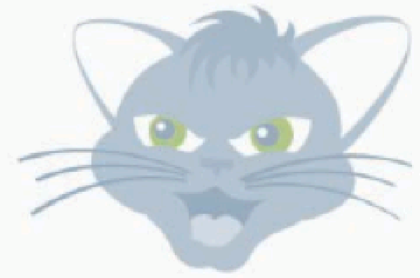
While Example

Factorial function

```
define factorial {
  dup [dec *] [[1 >] dip] while pop
}
```

There are much better implementations of factorial.

Comprehensions

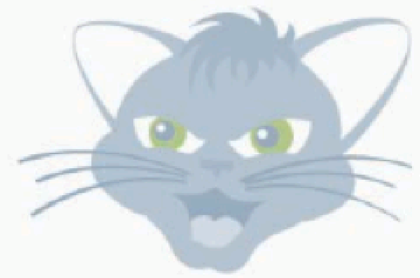


Lists can be generated using comprehensions.

n – Output a list of the first N natural numbers

init – Takes a starting value, and a function which is applied to each successive value generated.

Revisiting Factorial



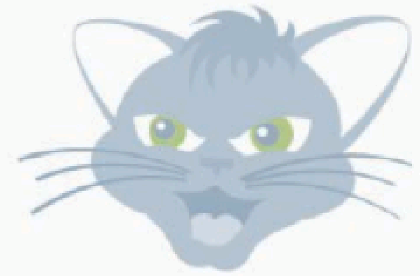
The easier way to create a factorial function is as follows:

```
define factorial {
  n [*] reduce
}
```

Doesn't this create overhead of manual list? Not for a clever implementation.

Cat Programming Language

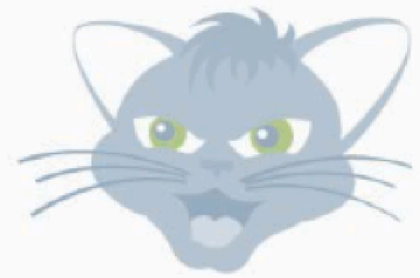
Selective Laziness



Cat does not dictate laziness, but some functions can be implemented to use generator functions instead of allocating lists in memory.

Cat Programming Language

Pure Functions

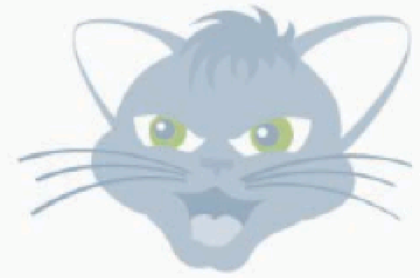


A pure function can be replaced with any other function generating the same result (referentially transparent).

```
{ 1 2 + } -> { 3 }
```

This is key for optimization.

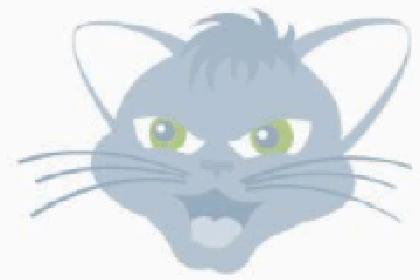
Impure Functions



Cat functions that have side-effects have a special type notation and are strictly restricted.

Most atomic programs, which take function parameters, don't accept impure functions as parameters.

Impure Functions



Impure functions are crucial to a practical language. Some common examples:

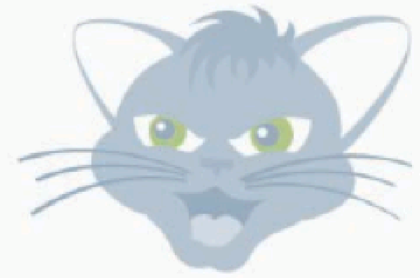
read – Read from standard input

write – Write to standard output

rnd – Generate a random number

time – Get the current time

Impure Limitations



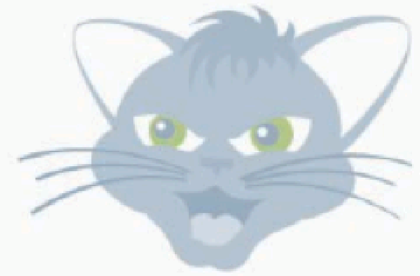
They can't be rewritten easily, order is important.

In Cat it is easy to identify pure functions, and to rewrite them.

Too bad for impure code (which is not as common as one might think)

Cat Programming Language

Ignoring Impurities



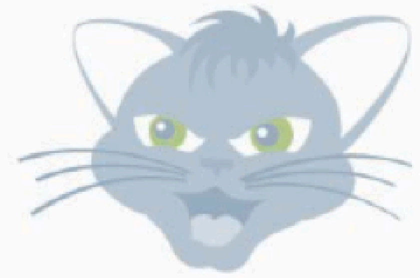
If g is impure, we can work around it:

```
define f { f0 f1 g f2 f3 }
```

We can rewrite it as:

```
define a { f0 f1 }
define b { f2 f3 }
define f { a g b }
```

Type Annotation



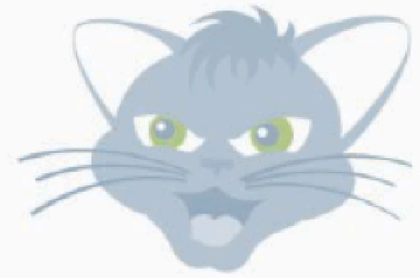
Cat supports type annotation.

```
define f : (int int)->(int)
```

Indicates that f consumes two int values from the stack and produces a single int value onto the stack.

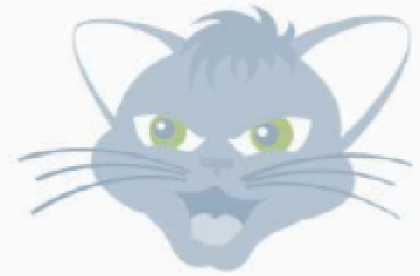
Cat Programming Language

Type Annotations



- They are very useful: they catch design bugs. To a certain degree they document what happens.
- They are optional, the type can be inferred.

Cat Programming Language



Labeled Type Annotations

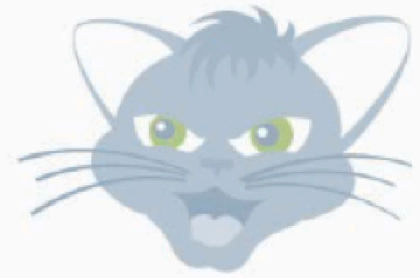
Type annotations may have labels for arguments.

```
define f : (x:int y:int) -> (x)
```

Only one interpretation: f = pop.

Cat Programming Language

Applicative Equivalency



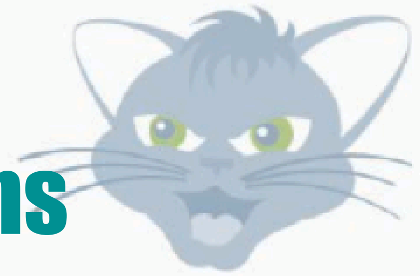
In some applicative-language X:

```
R f(X0 x0, X1 x1)
```

In Cat

```
define f : (x0:X0 x1:X1)->(R)
```

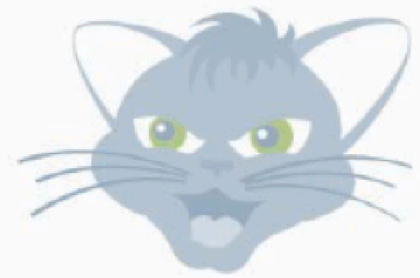
List Functions with Annotations



```
define cons : (a:any x:list)->(y:list)
define uncons : (x:list)->(a:any y:list)
define head : (x:list)->(car:any)
define tail : (x:list)->(cdr:list)
define first : (x:list)->(x car:any)
define rest : (x:list)->(x cdr:list)
define nil : ()->(x:list)
define unit : (a:any)->(x:list)
define pair : (a:any b:any)->(x:list)
define cat : (x:list y:list)->(z:list)
```

Cat Programming Language

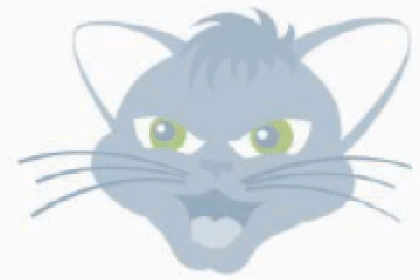
List Indexing Operations



Setting a value in a list, is a functional operation which constructs a new list.

```
define get_at : (x:list n:int)
  -> (x n a:any)

define set_at : (x:list n:int a:any)
  -> (y:list)

define [] { get_at }

define []= { set_at }
```

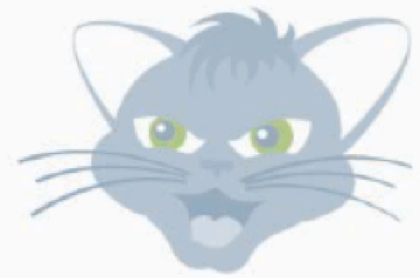
Impure Type Annotations



There are special type annotations for impure functions:

```
define read_int : () ~> (x:int)
define write_int : (x:int) ~> (x)
```

Type Inference

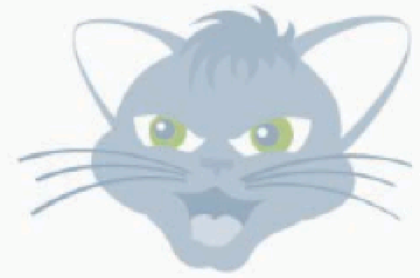


Every function consumes a number of stack elements (Cn) and produces a number of stack elements (Pn).

```
define f { g h }
Cn(f) = Cn(g) + Max(Cn(h) - Pn(g), 0)
Pn(f) = Pn(h) + Max(Pn(g) - Cn(h), 0)
```

This can be extended to two stacks.

Auxiliary Stack Notations

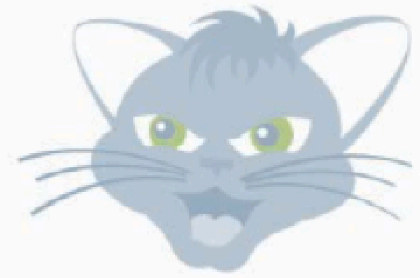


There is a theoretical notation for auxiliary stack (local stack) annotation.

Impractical for programmer use; imagine hundreds of locals.

Used only for discussion, not for annotation.

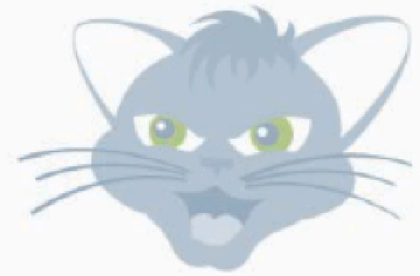
Primitive Subroutines



There are only two, but they have a special characteristic: embedded numerical constants.

```
load#N
store#N
```

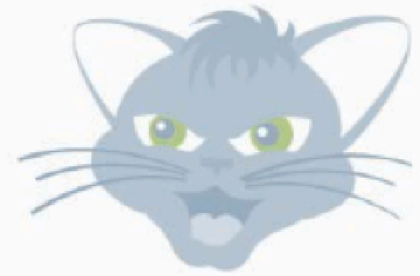
Load Subroutine Types



```
define load#0 :
  ()(x0:any) -> (x0)(x0)

define load#1 :
  ()(x1:any x0:any) -> (x1)(x1 x0)

define load#2 :
  ()(x2:any x1:any x0:any) -> (x2)(x2
 x1 x0)
```

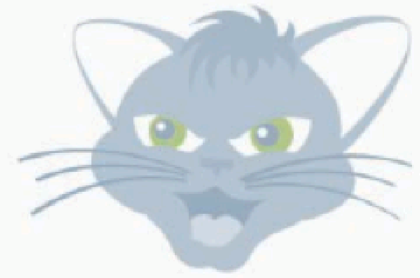
Cat Programming Language

Store Subroutines



```
define store#0 :
  (x:T)(x0:T) -> ()(x)

define store#1 :
  (x:T)(x1:T x0:any) -> ()(x x0)

define store#2 :
  (x:T)(x2:T x1:any x0:any) -> ()(x
 x1 x0)
```

Cat Programming Language



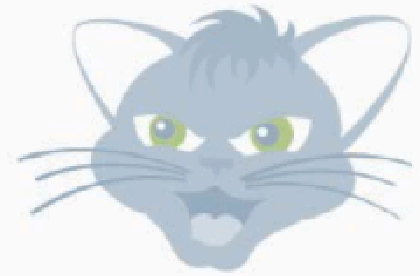
Imperative to Functional

Consider the following code:

```
{
  int x, y;
  x = 6;
  y = x * 7;
}
```

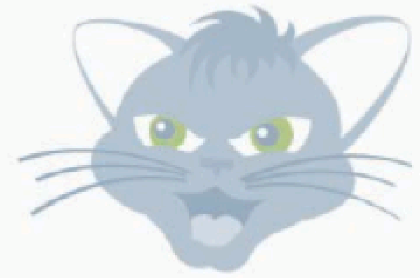
Cat Programming Language

Statements as Functions



```
{
  local_int // ()()->()(int)
  local_int // ()()->()(int)
  6         // ()()->(6)()
  stloc#1   // (a:any)(x1:any x0:any)->()(a x0)
  ldoc#1    // ()(x1:any x0:any)->(x0)(x1 x0)
  7         // ()()->(7)()
  *         // (int int)()->(int)
  stloc#0   // (a:any)(x1:any x0:any)->()(x1 a)
  pop_local // ()(any)->()()
  pop_local // ()(any)->()()
}
```

Combinators

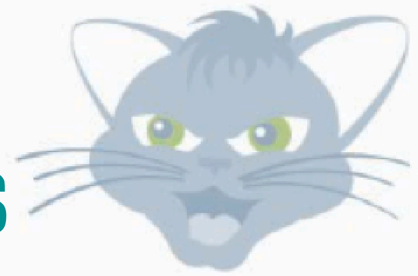


Combinators in Cat are pure functions which take pure functions (*but not subroutines*) as operators and evaluate them.

Expressing what they do is easy, expressing their type is a little trickier.

Cat Programming Language

Combinator Type Expressions



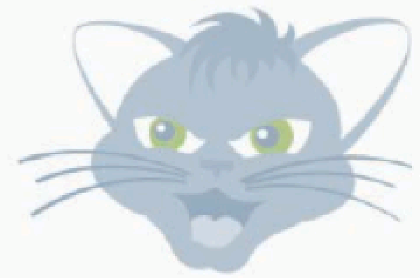
Types of combinators are complex expressions:

```
define f : ((A:any*)->(B:any*)) -> ()
```

In this case, f takes any program and ignores it. Short form is:

```
define f : (A->B) -> ()
```

I Combinator



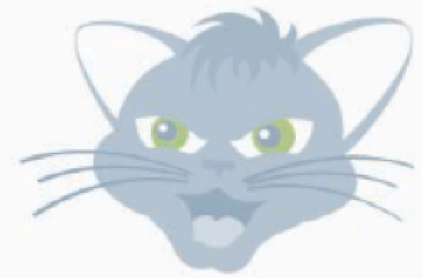
The i combinator takes a single function and executes it.

```
define i : (A f:A->B) -> (B)
```

The type of i is determined by the function parameter.

Cat Programming Language

If Combinator



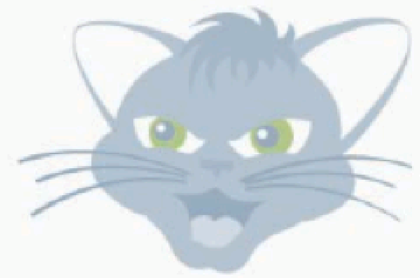
The if function is also a combinator:

```
define if :
  (x:A cond:bool f:A->B g:A->B) -> B
```

Note that if can not have an ambiguous type.

Cat Programming Language

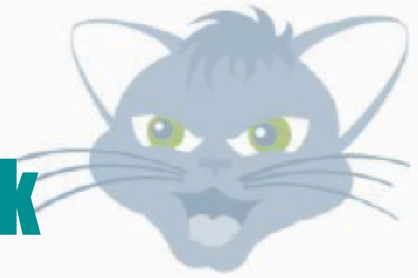
Dip Combinator



The dip combinator takes a function and executes it on the stack, but first removing the top element, and then returning it afterwards.

```
define dip : (x:A C:any f:A->B)->(B C)
```

So common it has a shortcut, the tick symbol (').

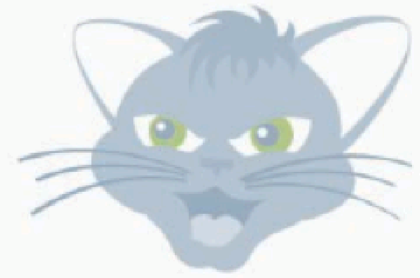


Eliminating the Auxiliary Stack

All pure Cat programs can be rewritten as a pure Cat function by using the dip combinator.

(open problem: doing so using transforms.)

Implementing Dip

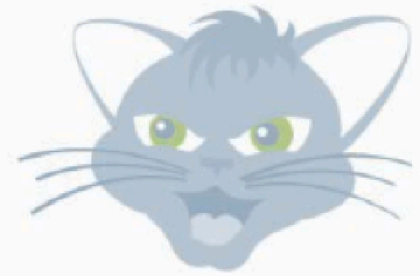


Interestingly: the implementation of dip requires a secondary stack.

```
define dip : (A x:any f:A-B)->(B x)
{
   local_any
   swap stloc#0
  i ldloc#0
  pop_local
}
```

Cat Programming Language

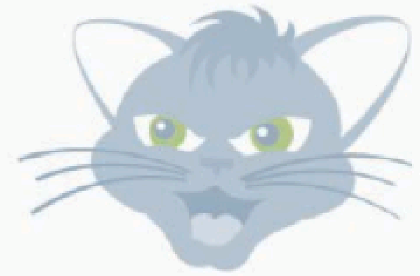
App2 Combinator



Apply function twice, first with y on top, then with x on top.

```
define app2 : (x:A y:A f:A->B)
  -> (A A)
```

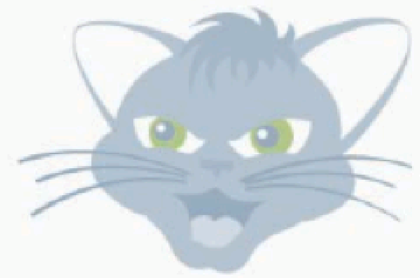
Other Combinators



There are many other combinators, not the least of which are the tail recursion (tailrec) and binary recursion (binrec) combinators.

Cat Programming Language

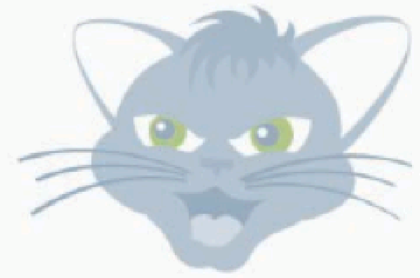
Quick Sort with Binrec



Showing off a bit:

```
define quick_sort {
   [small]  // termination condition
   []       // termination action
   [uncons [>] split]    // default action
   [[swap] dip cons cat] // last action
  binrec
}
```

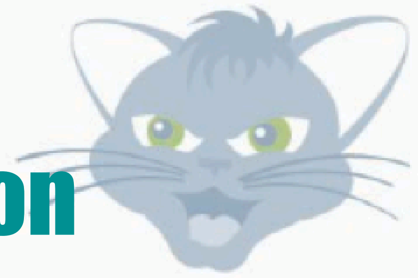
Combinatorial Basis



Any *pure* Cat program can be reduced to a minimal combinatorial basis of two combinators (not quite S and K, but close).

http://tunes.org/~iepos/joy.html#applic

Dynamic Function Construction

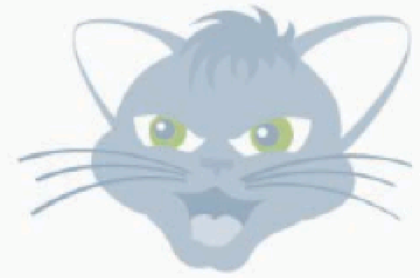


Functions can be concatenated using the cat_fxn operation.

A function which always returns a specific value, can be made using make_fxn.

Cat Programming Language

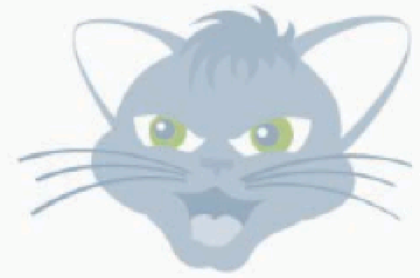
Currying



Binding a function argument with a variable is a simple operation:

```
define curry :
  (a:any f:program)->(program)
{
  [make_fxn] dip cat_fxn
}
```

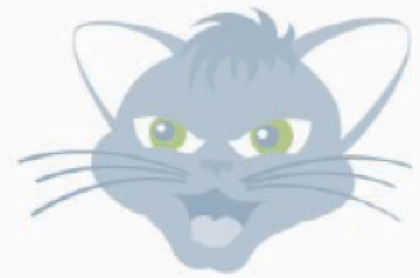
Transformation Engine



Pure Cat functions can be replaced with equivalent pure functions without changing the behavior of the program.

```
{ 1 + } -> { inc }
{ 0 + } -> { }
{ 3 pop } -> { }
{ 1 2 + } -> { 3 }
```

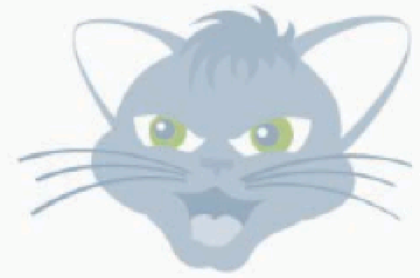
Transforms



A transform expresses a pattern rewriting rule. In effect it is a declarative macro. Conforms to scoping rules.

```
transform { 1 + } -> { inc }
transform { 0 + } -> { }
transform { + } <-> { swap + }
```

Purposes of Transforms

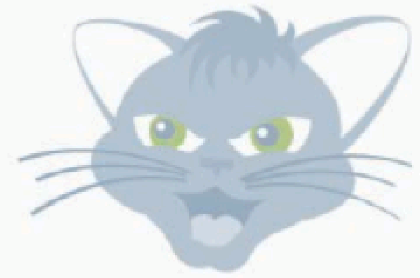


Transforms serve two purposes:

1) Macros (code simplification, syntax rewriting)

2) Expressing optimizations

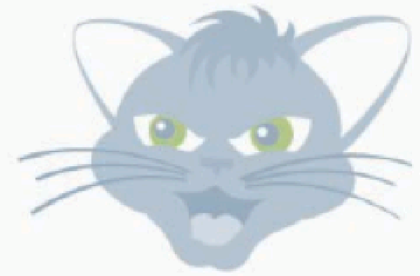
Kinds of Transforms



There are two kinds of transforms, reduction transforms and bi-directional transforms.

Both transforms represent an equivalency. Reduction transforms are always applied if possible.

Deforestation

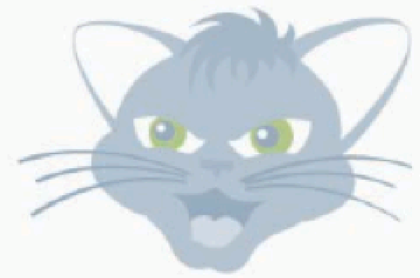


The elimination of intermediate data-structures in functional code is known as deforestation.

There are clear similarities between Cat transforms and deforestation patterns.

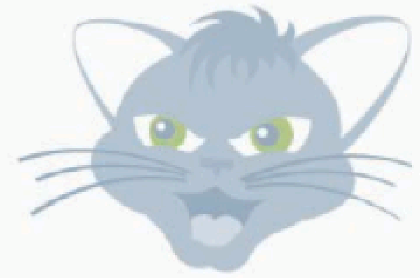
Open problem: can major deforestation heuristics be expressed as Cat transforms?

Associative Transforms



```
{+' +} -> {+ +}
{*' *} -> {* *}
{and' and} -> {and and}
{or' or} -> {or or}
{max' max} -> {max max}
{min' min} -> {min min}
{cat' cat} -> {cat cat}
{merge' merge} -> {merge merge}
```

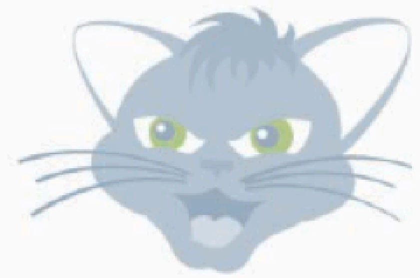
Symmetric Transforms



Certain functions can be reduced to no-ops:

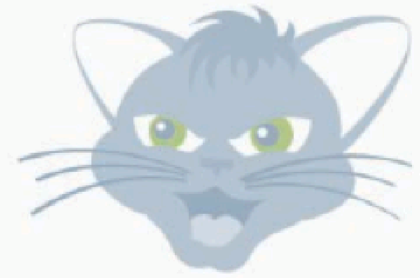
```
{ swap swap } -> {}
{ dup pop } -> {}
{ uncons cons } -> {}
{ zip unzip } -> {}
{ adjacent_difference partial_sum }
  -> {}
```

Homomorphic Transforms



```
{* log} -> {log app2 +}
{and not} -> {not app2 or}
{or not} -> {not app2 and}
{+ double} -> { [double] app2 +}
{square} -> { [square] app2 *}
{max succ} -> { [succ] app2 max}
{cat size} -> { [size] app2 +}
{cat sum} -> { [sum] app2 +}
{cat product} -> { [product] app2 *}
```

Map-Map Transform



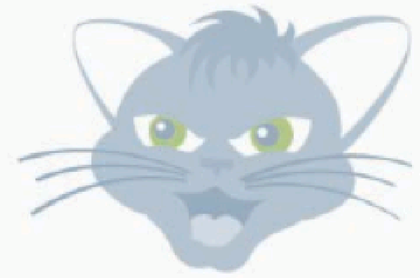
Maps are distributive.

```
transform { f map g map } ->
  { [f g] map }
```

Automatically identifying and rewriting these patterns gives a huge boost to parallel computations.

Cat Programming Language

Filter-Filter Transform



Filters can be combined.

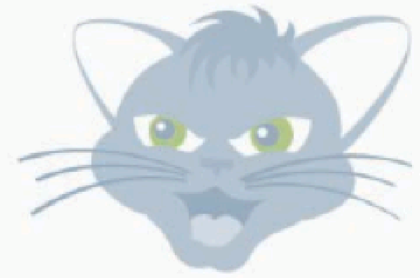
```
transform { f filter g filter } ->
  { [f g and] filter }
```

Filters are also associative.

```
transform { f filter g filter } <->
  { g filter f filter }
```

Cat Programming Language

Map-Filter Transform

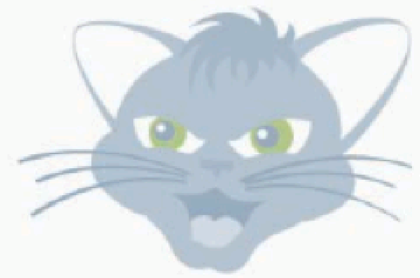


A special operation called "filtermap" combines the effect of mapping and filtering in one operation.

```
transform { [f] map [g] filter }
  -> { [f dup g] filtermap }
```

Cat Programming Language

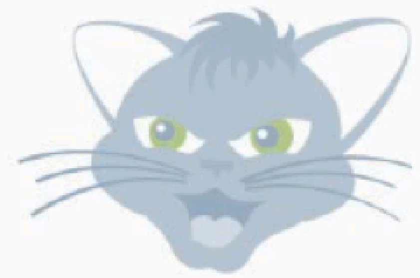
Filtermap Transforms



Filtermap takes a function which consumes a single value, and produces two values (the mapped value, and a boolean).

```
transform {
  [f] filtermap [g] filtermap
} -> {
  [f [pop g][] if] filtermap
}
```

Cat Programming Language

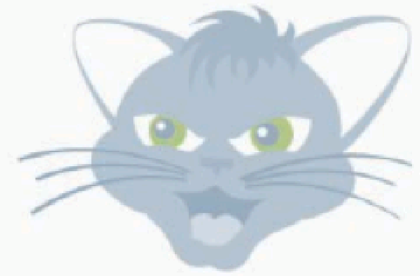


Filter–Reduce Transforms

Filter-reduce operations can be expressed as reduce operations.

```
transform { [f] filter x [g] reduce }
-> { x [f [g] [] if] reduce }
```

Map-Reduce Transform

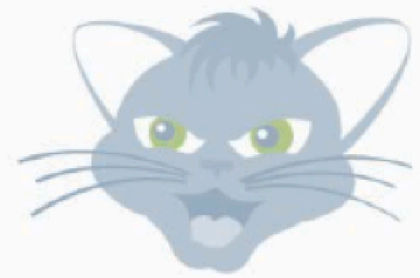


Map-reduce operations can be expressed as reduce operations.

```
transform { [f] map x [g] reduce }
-> { x [f g] reduce }
```

Cat Programming Language

Split-Reduce-Merge

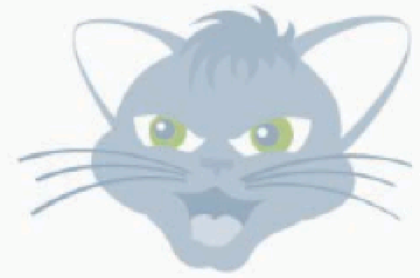


Certain operations when reduced, can be split and then merged.

```
transform { [+] reduce } ->
  { [+] [+] splitreducemerge }
```

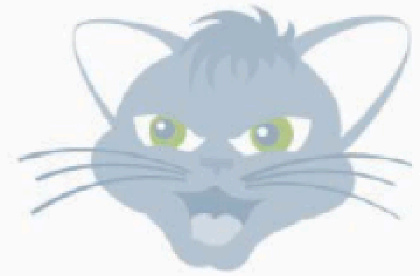
Identifying these is important for parallelization.

Ad Infinitum



The number of reductions which can be expressed as Cat transforms goes on and on. What has been presented here is just the tip of the iceberg.

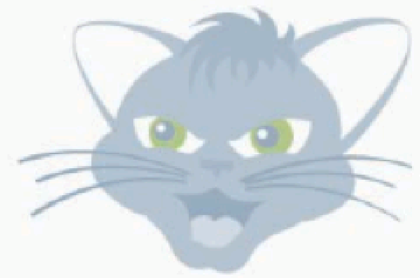
Translating Imperative Code to Cat



Consider the following:

```
list swap_elements(list a, int i, int j)
{
  result = a
  tmp = result[i];
  result[i] = result[j];
  result[j] = tmp
  return result
}
```

Cat Programming Language

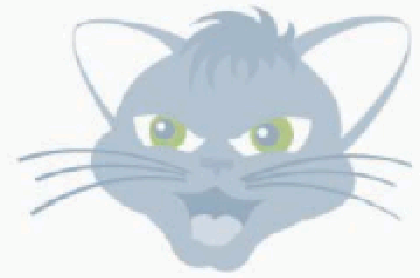


Using Combinators

It is possible, but it's a brain teaser:

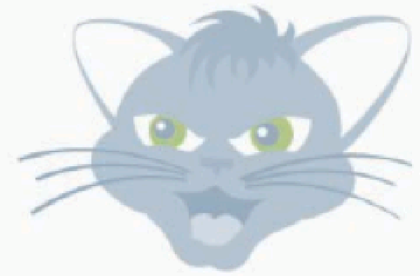
```
define swap_elements
 : (list int int)->(list)
{
  [get_at] app2 // a i j ai=a[i] aj=a[j]
  rot4left      // a j ai aj i
  swap          // a j ai i aj
  [set_at] dip2 // a[j]=ai i aj
  set_at        // a[i] = aj
}
```

Locals are Easier than Dip



- Dip requires an extra stack to implement anyway.
- Imperative code often written with locals.
- Easier for programmers to understand and use variables.
- Machine generating the code (e.g. translation) is *hard*.

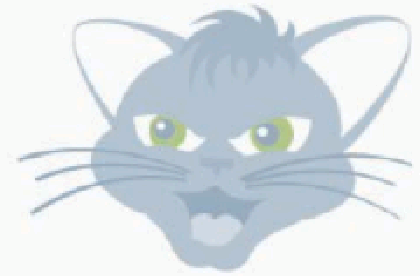
Using Local Stack



You can convert an imperative program to a verbose Cat program, but the mapping is straightforward.

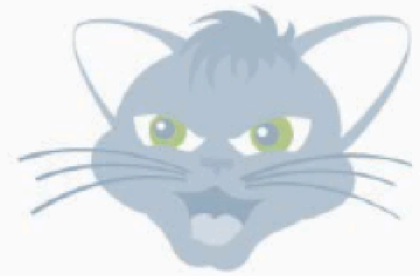
Much of what happens can be expressed using pure functions.

Using Local Stack



```
define swap_elements
{
  // variable and temporary declarations
  local_list  // a
  local_int   // i
  local_int   // j
  local_list  // result
  local_any   // tmp

  // local getters
  define : a? { ldloc#4 }
```

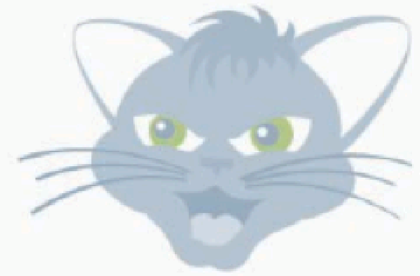
Using Local Stack



```
define : i? { ldloc#3 }
define : j? { ldloc#2 }
define : result? { ldloc#1 }
define : tmp? { ldloc#0 }

// local setters
define : a= { stloc#4 }
define : i= { stloc#3 }
define : j= { stloc#2 }
define : result= { stloc#1 }
define : tmp= { stloc#0 }
```

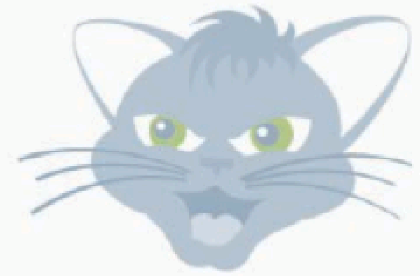
Cat Programming Language

Using Local Stack



```
// initialize argument locals
j= i= a=

// result = a
a? result=

// tmp = result[i]
result? i? [] tmp= pop pop

// result[i] = [j]
```

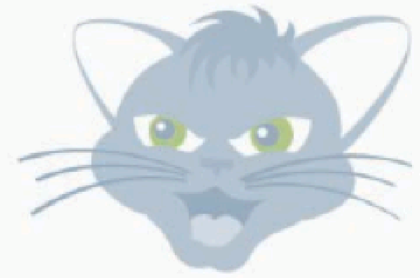
Using Local Stack



```
  result? i? result? j? [] []= result=

  // result[j] = tmp
  result? j? tmp? []= result=

  // return value
  result?

  // clean-up stack
  [poploc] 5 repeat
}
```

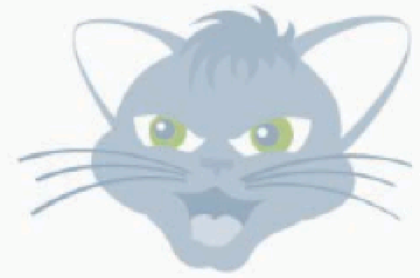
Cat Programming Language

Cat to MSIL



- Arithmetic operations are straightforward
- Returning multiple values has to be faked a bit (use lists or tuples)
- Auxiliary stack maps to list of locals
- Consumed elements have to be mapped to locals.

Emulating Stack Frames



Arguments are the top elements on the main stack. Must be pushed onto auxiliary stack.

Locals are pushed and popped from the auxiliary stack.

Result is pushed on to the main stack.

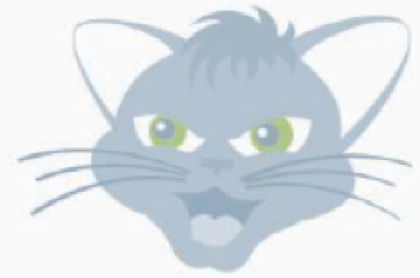
Cat Programming Language

Returning Multiple Values in MSIL

Unfortunately MSIL doesn't support multiple return values.

There are numerous work-arounds, such as returning tuples or lists.

Cat Programming Language



Performance Results

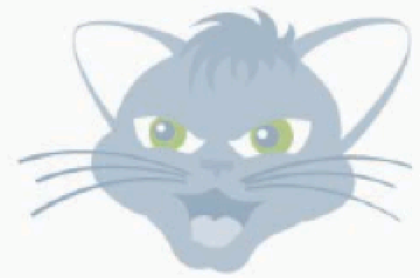
T.B.A.

My question is, who is most interested, and what kinds of results would be most convincing?

Come talk with me, about what you want to see happen with Cat.

Cat Programming Language

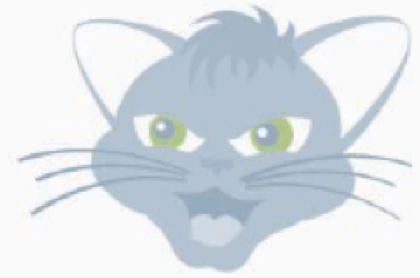
Current Status



Various C# interpreters and MSIL generators for older, deprecated versions of the language.

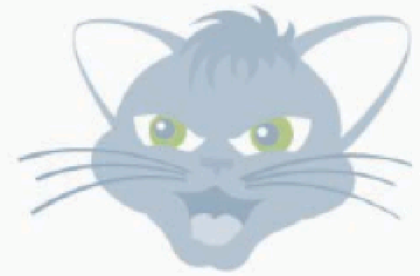
Release of version corresponding to these slides planned for September 1st, 2006.

The Quixotic Dream



The universal translator: any programming language to Cat, and Cat to any programming language.

For More Information



http://www.cdiggins.com/cat