Metaprogramming Showdown

Lisp vs. Ruby

Tim Crews

Introduction

- **Metaprograms** are *programs about programs*, or (more commonly) *programs that write programs*.
- The appeal:
  - The ability to bend the programming language to our wills, making it the programming language that best expresses our ideas. "Language-Driven Design".
- Lisp is portrayed by practitioners as the ultimate metaprogramming language.
- Metaprogramming is also a big part of Ruby culture.
- So what is the practical difference?

Lisp

From Theory to Practice and Back Again

Theory Begets Practice

- Originated by John McCarthy (in 1958!) as a paper notation to investigate the lambda calculus.
  - Functions as a theoretical basis for mathematics, instead of set theory.
  - The theory of computation – what is computable?
  - Functions as values.
- In the spirit of the times, McCarthy used his notation to describe an `eval` function which executed functions written in the same notation.
- Steve Russell: Hey, I can implement `eval` on our IBM 704!

Lisp's Relationship to Other Languages

- Many “modern” programming language features were pioneered by Lisp decades earlier.
- Smugness:
  - Phillip Greenspun: *Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.*
  - Erik Naggum: *Those who do not know Lisp are doomed to reinvent it.*
- Besides that hopefully tongue-in-cheek smugness, Lisp practitioners are sincerely enthusiastic about the *theoretically* unrivalled power of their language.
- Paul Graham helped foster a Lisp renaissance in the 2000s by popularizing these ideas.

(Most of) Lisp Syntax in One Slide

- `‘(1 2 3)` – A literal list of numbers
- `‘(1 (2 3) 4)` – Nest lists to make trees.
- `(fn a b)` – Unless suppressed, the first position in a list is treated as a function or macro to be invoked. Such a list is called a "form."
- `(+ 1 2)` – Add 1 and 2.
- `(lambda (arg) (+ arg 1))` – An anonymous function that adds one to its argument.
-
```
(defun add_two (a b)
   (+ a b))
(add_two 5 6);   => 11
```

Ruby Aside: Abstract Syntax Trees

```
class Simple

  def add(n1, n2)
    return n1 + n2
  end

end

ParseTree.translate(Simple)
```

```
[:class,
 :Simple,
 [:const, :Object],
 [:defn,
  :add,
  [:scope,
   [:block,
    [:args, :n1, :n2],
    [:return, [:call, [:lvar, :n1],
                      :+, [:array, [:lvar, :n2]]]]]]]]]
```

What does Lisp code look like?

Simple Database

```
(defun make-db (&optional (size 100))
  (make-hash-table :size size))

(defvar *default-db* (make-db))

(defun clear-db (&optional (db *default-db*))
  (clrhash db))

(defmacro db-query (key &optional (db '*default-db*))
  `(gethash ,key ,db))

(defun db-push (key val &optional (db *default-db*))
  (push val (db-query key db)))

(defmacro fact (pred &rest args)
  `(progn (db-push ',pred ',args)
          ',args))
```

The Abstract Syntax Tree of your code is directly represented as a tree data structure.

And that data structure is the first-class data structure of the language.

Collection Functions

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcall fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setq wins obj
                    max  score))))
        (values wins max))))

(defun best (fn lst)
  (if (null lst)
      nil
      (let ((wins (car lst)))
        (dolist (obj (cdr lst))
          (if (funcall fn obj wins)
              (setq wins obj)))
        wins)))

(defun mostn (fn lst)
  (if (null lst)
      (values nil nil)
      (let ((result (list (car lst)))
            (max (funcall fn (car lst))))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (cond ((> score max)
                   (setq max    score
                         result (list obj)))
                  ((= score max)
                   (push obj result)))))
        (values (nreverse result) max))))
```

Lisp Macros

- Because the code itself is represented as a data structure, you can write compile-time code that operates on that data structure.
- The domain and target languages are the same language.
- A macro is a Lisp function, executed at compile-time, that transforms ASTs into new ASTs.
- Macros have the full use of the Lisp language.

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))

(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

Defining a while Loop Construct

- Common Lisp doesn't define a while loop construct.
- We want to make one that could be used like this:

```
(let ((x 1))
  (while1 (< x 10)
    (print “Hello”)
    (setf x (+ x 1))))
```

- Let's try to implement it as a function:

```
(defun while1 (test &rest body)
  (do ()
      ((not (eval test)))
    (eval body)))
```

What's Broken with while1

- The condition `(< x 10)` and the body `(print...)`, `(setf...)` are not quoted.
- They are immediately evaluated one time only at the call site.
- These don't result in forms, so the `evals` make no sense.

```
(defun while1 (test &rest body)
  (do ()
      ((not (eval test)))
    (eval body)))
```

```
(let ((x 1))
  (while1 (< x 10)
    (print “Hello”)
    (setf x (+ x 1))))
```

Fixing the eval Problem

- If we're willing to change the use case, we can fix the eval part of the problem by quoting the arguments and adding a `progn`:

```
(let ((x 1))
  (while1 '(< x 10)
    '(progn
       (print “Hello”)
       (setf x (+ x 1)))))
```

- But this makes the interface to `while1` different and less convenient than native control structures.

What's Still Wrong with while1

- The variable `x` is not in scope in `while1`.
- The lists like `'(< x 10)` are just lists.
- They do nothing to get `x` into the scope of `while1`.

```
(defun while1 (test &rest body)
  (do ()
      ((not (eval test)))
    (eval body)))
```

```
(let ((x 1))
  (while1 '(< x 10)
    '(progn
      (print "Hello")
      (setf x (+ x 1)))))
```

```
*** - EVAL: variable X has no value
The following restarts are available:
USE-VALUE      :R1      Input a value to be used instead of X.
STORE-VALUE    :R2      Input a new value for X.
ABORT          :R3      Abort main loop
Break 1 [12]>
```

Delayed Evaluation with a Lambda

- It could be defined this way:

```
(defun while2 (test body)      ; "&rest" removed
  (do ()
      ((not (funcall test)))
    (funcall body)))
```

- And called this way:

```
(let ((x 1))
  (while2 (lambda () (< x 10))
    (lambda ()
      (print “I am terribly awkward!”)
      (setf x (+ x 1)))))
```

Pros and Cons of while2

- It works:

```
"I am terribly awkward"
"I am terribly awkward"
"I am terribly awkward"
"I am terribly awkward"
"I am terribly awkward"
"I am terribly awkward"
"I am terribly awkward"
"I am terribly awkward"
"I am terribly awkward"
NIL
```

- The lambdas are closures that include `x` in their scope.

```
(let ((x 1))
  (while2 (lambda () (< x 10))
    (lambda ()
      (print "I am terribly awkward!")
      (setf x (+ x 1)))))
```

- But `while2` is still less convenient than native control structures.

Make it a Macro

```
(defmacro while3 (test &rest body)
  `(do ()
       ((not ,test))
     ,@body))
```

- The backquote suppresses immediate evaluation of everything that follows so you can construct the new AST without evaluating it.
- But then the `,` in `,test` reverses that, so it *is* evaluated.
- `,@body` causes body to be spliced into the loop.
- So the key feature provided by this macro is **control over when parameters are evaluated.**

The while3 Macro Expansion

```
(defmacro while3 (test &rest body)
  `(do ()
       ((not ,test))
     ,@body))


(macroexpand-1
  '(while3 (< x 10)
     (print "Hello")
     (setf x (+ x 1))))

(DO NIL ((NOT (< X 10))) (PRINT "Hello") (SETF X (+ X 1)))
```

The while3 macro executed

- With this technique, `while3` is indistinguishable from a built-in loop construct.

```
(let ((x 1))
  (while3 (< x 10)
    (print “Hello”)
    (setf x (+ x 1))))
```

```
"Hello"
"Hello"
"Hello"
"Hello"
"Hello"
"Hello"
"Hello"
"Hello"
"Hello"
NIL ;
```

No closure is involved or needed. The macro was expanded in-place inside the `(let...)` form.

Turing-Complete – On Purpose!

- This macro can do everything at compile-time that Lisp can do at run-time.
- You don't have to learn a second language to metaprogram.
- You don't have to jump between two different programming paradigms.
- Nevertheless, two levels of program do co-exist, which can be disorienting.

```
(defconstant *segs* 20)
(defconstant *du*   (/ 1.0 *segs*))
(defconstant *pts*  (make-array (list (1+ *segs*) 2)))

(defmacro genbez (x0 y0 x1 y1 x2 y2 x3 y3)
  (with-gensyms (gx0 gx1 gy0 gy1 gx3 gy3)
    `(let ((,gx0 ,x0) (,gy0 ,y0)
           (,gx1 ,x1) (,gy1 ,y1)
           (,gx3 ,x3) (,gy3 ,y3))
       (let ((cx (* (- ,gx1 ,gx0) 3))
             (cy (* (- ,gy1 ,gy0) 3))
             (px (* (- ,x2 ,gx1) 3))
             (py (* (- ,y2 ,gy1) 3)))
         (let ((bx (- px cx))
               (by (- py cy))
               (ax (- ,gx3 px ,gx0))
               (ay (- ,gy3 py ,gy0)))
           (setf (aref *pts* 0 0) ,gx0
                 (aref *pts* 0 1) ,gy0)
           ,@(map1-n #'(lambda (n)
                         (let* ((u (* n *du*))
                                (u^2 (* u u))
                                (u^3 (expt u 3)))
                           `(setf (aref *pts* ,n 0)
                                  (+ (* ax ,u^3)
                                     (* bx ,u^2)
                                     (* cx ,u)
                                     ,gx0)
                                  (aref *pts* ,n 1)
                                  (+ (* ay ,u^3)
                                     (* by ,u^2)
                                     (* cy ,u)
                                     ,gy0))))
                     (1- *segs*))
           (setf (aref *pts* *segs* 0) ,gx3
                 (aref *pts* *segs* 1) ,gy3))))))
```

Figure 13.5: Macro for generating Bezier curves.

Beyond Functional Programming

- All functional programming languages treat functions as data.
- But Lisp treats the functions' *code* as data.
- No gimmicks, back-doors, or language hacks.
- Just as we can create libraries of functions that operate on specialized data structures, we can create libraries of macros that transform code structures.

From Practice Back to Theory

- In a Lisp program, all functionality is represented by forms.
- The Lisp language allows you to create any new form you want, that can do anything that Lisp can do – which means anything that is computable.
- There are no priviliged constructs out of reach of the Lisp programmer.

(((((The Down-side)))))

- How can a language with an awesome hook like that not have hit the mainstream after all this time?
- Except for macros, all other Lisp language features can be (and have been) copied by other languages.
- The lack of syntax is very hard to live with.
- But that's what makes macros possible.
- It is the essential defining feature that makes a Lisp.

- A cultural issue:
  - "The Lisp Curse" -- Maybe it is *too* fluid.

Ruby Metaprogramming

Is Ruby "an Acceptable Lisp?"

Creating a New “while” in Ruby

- Of course, Ruby already has a while loop:

```
x = 1
while x < 10 do
    puts "Hello"
    x += 1
end
```

- Mission: Define a replacement for `while` using a function.
- Ruby faces similar issues to Lisp.

Blocks are Awesome

- Blocks provide an elegant solution for the loop body:

```
def while2 (bool_test)
  loop do
    break if not bool_test
    yield
  end
end
```

- This thoroughly addresses one major use of macros in Lisp: Establishing a context in which code can run.
- Ruby code uses this a *lot*, without resorting to metaprogramming.

What is not broken about while2

- Parsing the comparison expression as an argument to `while2` without parentheses works just fine:

```
y = 2
while2 y < 5 do
  puts y
  y += 1
end
```

```
def while2 (bool_test)
  loop do
    break if not bool_test
    yield
  end
end
```

- No syntax error on the while2 call. Thanks Ruby!

What is not broken about while2

- The scope inside "the loop" is the same scope as outside "the loop".

```
y = 2
while2 y < 5 do
  puts y
  y += 1
end
```

```
def while2 (bool_test)
  loop do
    break if not bool_test
    yield
  end
end
```

- The block that forms the body of the loop is a closure whose context (including `self`) is the same as that in which the function was called.
- There are many metaprogramming tricks that give us exquisite control of the context of the block.

What is broken about while2

- The bool_test is only evaluated once, at the call site:

```
y = 3
while2 y < 5 do
  puts y
  y += 1
end
```

3

4

5

6

7

(keeps incrementing forever...)

```
def while2 (bool_test)
  loop do
    break if not bool_test
    yield
  end
end
```

Adding Delayed Evaluation

- As with the Lisp while2, the condition can be made a proc that can be called.

```
def while3 (bool_test)
  loop do
    break if not bool_test.call
    yield
  end
end
```

while3 Does Work

```
y = 3
while3 lambda {y < 5} do
  puts y
  y += 1
end
```

- As with the Lisp while2 solution, the blocks/lambdas are closures that solve the scope problems.
- As with the Lisp while2 solution, this syntax doesn't look like the native loop syntax.
- So now let's move on to the solution that looks like real Ruby syntax...

The Solution that Looks Like Ruby Syntax

- There isn't one.
- For fine control over evaluation time for arguments, there is no substitute for a macro.
- When a language has concrete syntax, the compiler/intepreter holds a privileged position as the owner of that syntax.

- On the other hand, we were able to accomplish the desired *semantics*, and didn't have to use metaprogramming at all.

Implementing && as a Function

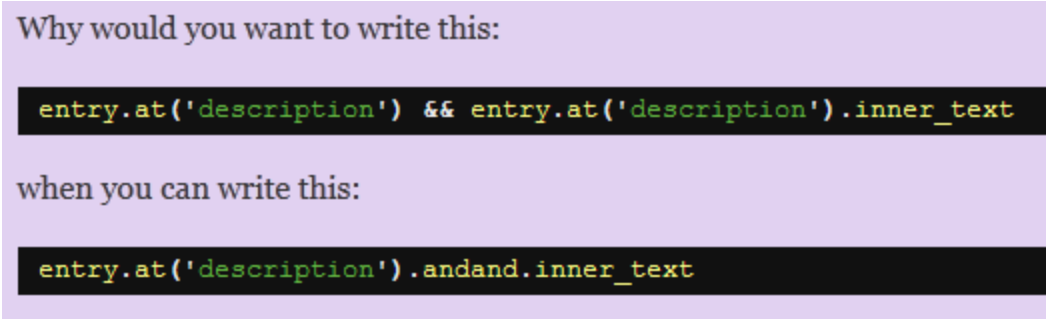
- Mission 2: Implement your own version of && with short-circuit evaluation. Call it "andand".
- Test case:

```
false andand launch_nuclear_missiles
```

- Expected result: Does not launch nuclear missiles.
- This is another demonstration that the compiler/interpreter owns the syntax.
- In Ruby, you cannot define new infix functions.
- Even if you could make an infix function, you would have to use the deferred evaluation trick for the arguments, which affects usability.

Compromising Syntax for Semantics

- There really is a gem named "andand", and it supports short-circuit evaluation *semantics*.
- The compromise is: You have to sacrifice the *syntax*.



- Each step in the chain is a separate function call, and an opportunity to short-circuit the evaluation.

How andand Does it

- It does it without any metaprogramming at all.
- But it monkeypatches Object.
- And it only *mostly* works.
- `nil.andand.bar(blitz())` does return `nil`, but calls `blitz` anyway. (According to its author. I don't get why it would.)

```
def andand (p = nil)
  if self
    if block_given?
      yield(self)
    elsif p
      p.to_proc.call(self)
    else
      self
    end
  else
    if block_given? or p
      self
    else
      MockReturningMe.new(self)
    end
  end
end
```

```
class Object
  include AndAnd::ObjectGoodies
end
```

But What if We Could Modify the AST?

- "If brute force doesn't work, you're not using enough" – Red Green
- There is a `parsetree` gem that generates a Ruby data structure that is the parse tree for your Ruby class or method.
- There is a `rewrite` gem that transforms parse trees.
- There is a `ruby2ruby` gem that takes a parse tree and converts it to a Ruby class or method.
- That sounds just like a Lisp macro!
- This round trip enables some very powerful metaprogramming techniques.

andand Using Rewrite

- andand can be implemented with rewrite so that it doesn't rely on monkeypatching Object.

```
with(andand) do
    # ...
    foo().andand.bar(blitz())
    # ...
end
```

```
# ...
lambda do |__121414053598468__|
    if __121414053598468__.nil?
        nil
    else
        __121414053598468__.bar(blitz())
    end
end.call(foo)
#...
```

- And it implements the desired semantics all the way.

Not for the Faint of Heart

- The rewrite function that accomplished that:
- There's a lot that is recognizable from Lisp macro techniques.
- But we're arguably in the territory of using a different language for metaprogramming.

```
def process_call(exp)
  exp.shift
  receiver_sexp = exp.first
  if matches_andand_invocation(receiver_sexp)
    exp.shift
    mono_parameter = Rewrite.gensym()
    s(:call,
      s(:iter,
        s(:fcall, :lambda),
        s(:dasgn_curr, mono_parameter),
        s(:if,
          s(:call, s(:dvar, mono_parameter), :nil?),
          s(:nil),
          begin
            s(:call,
              s(:dvar, mono_parameter),
              *(exp.map { |inner| process_inner_expr inner })
            )
          ensure
            exp.clear
          end
        )
      ),
      :call,
      s(:array,
        process_inner_expr(receiver_sexp[1])
      )
    )
  else
    begin
      s(:call,
        *(exp.map { |inner| process_inner_expr inner })
      )
    ensure
      exp.clear
    end
  end
end
```

Inherently Limited by Concrete Syntax

- Ruby class or method
  ⇒ parse tree
  ⇒ Ruby class or method
- If the code you want to translate from or to isn't already valid Ruby, parse tree manipulation has nothing to work with.
- So even this brute-force approach will not let us make a real `while` or `&&` replacement.

- The Ruby interpreter is a program written in C.
- So the "brutest force" approach would be to modify MRI to understand your new syntax.

Macros vs. Concrete Syntax

The Heart of the Issue

What's the Best We Could Do, Even Theoretically?

- What if we wanted to expose a programming interface to the compiler/intepreter that allowed new syntax to be defined?
- What language would *it* be written in? How would the functionality of the new syntax be communicated?
- I am almost sure this could not be Ruby.
- I think we will need something that is "turtles all the way down", independent of a particular concrete syntax.
- . . .

But Would We Ever Really Do That?

- Quotes from Paul Graham's essay *Beating the Averages*:
  - Programming languages are not merely technologies, but habits of mind as well.
  - Programming languages vary in power. (Introduces the "Blub" language to avoid alienating any readers.)
  - As long as our hypothetical Blub programmer is looking down the power continuum, he knows he's looking down. Languages less powerful than Blub are obviously less powerful, because they're missing some feature he's used to. But when our hypothetical Blub programmer looks in the other direction, up the power continuum, he doesn't realize he's looking up. What he sees are merely weird languages.

Paul Graham's Most Important Language Feature

- The source code of the Viaweb editor was probably about 20-25% **macros**. Macros are harder to write than ordinary Lisp functions, and it's considered to be bad style to use them when they're not necessary. So every macro in that code is there because it has to be. What that means is that at least 20-25% of the code in this program is doing things that you can't easily do in any other language.

- The resulting software did things our competitors' software couldn't do. Maybe there was some kind of connection.

Lisp and "Little Languages"

- Common Lisp programming practice is to create "little languages".
- Given a problem, imagine the language that would allow you to concisely represent the solution to that problem.
- Use Lisp to create that language.
- Write the solution in that language.
- Repeat until your application is finished.

Write Macros Until No Regularities are Left

- "When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough--often that I'm generating by hand the expansions of some macro that I need to write." - Paul Graham, *Revenge of the Nerds.*

Back to Ruby

Ruby as a Metaprogramming Platform

- Ruby is a very pleasant language to program in.
- Our brains are wired for syntax.
- I believe that Ruby is about as good a metaprogramming platform as is possible, without abandoning concrete syntax.
- The Ruby community has embraced metaprogramming more than any other non-Lisp community that I know of. (Compare to Python.)
- External factors make Ruby a more practical language for everyday use.

Leftover Slides

From an Earlier Version of the Presentation,
Covering C and C++ Metaprogramming

The C Preprocessor

Metaprogramming with Stone Tools

C Preprocessor Language

- `#define` – Defines an object-like or function-like macro:
  - `#define SOME_CONSTANT 1`
  - `#define TIMES_2_PLUS_1(x) ((x)*2+1)`
- `#if, #ifdef, #ifndef` – Conditionals
- `##` – Token concatenation
  - `#define MANGLED_VARIABLE(x) mangled_##x`
  - `MANGLED_VARIABLE(whatever)` => `mangled_whatever`
- # – string-ize (a little more on this later)
- And that's all there is – really!

C Preprocessor – Common Use

- The C preprocessor is almost always used as a simple fill-in-the-blanks template generator.
- Creating a “poor man’s” inline function.

```
#define max(a,b) ((a)>(b)?(a):(b))
```

- This could not be implemented as any single C function because of static typing.
- Macros are not first class citizens at compile-time. They don’t even *exist* at compile time.
- s/compile/run/g
- This is only just barely metaprogramming.

C Preprocessor – Less Common Use

- The preprocessor can produce real functions:
- ```
  #define MAKE_PRINTALL(typ,format)                                  \
      void printall_##typ (                                          \
          typ Collection[],                                          \
          int CollectionSize)                                        \
      {                                                              \
          int counter;                                               \
          for (counter=0; counter<CollectionSize; ++counter)         \
              printf ("%"#format"\n", Collection[counter]);          \
      }

  MAKE_PRINTALL(int,d)
  int numbers[5] = {0,1,2,3,4};
  printall_int(numbers, sizeof(numbers));
  ```
- Source-level debugging grows more inconvenient as macro size increases.
- This example is a strong motivator for C++ templates.
- I still wouldn't call this metaprogramming.
  ```

C Preprocessor Tricks

- Clever tricks based on token concatenation results in something that imitates a general-purpose language.
- These push the technology beyond its original intent.
  - Integer arithmetic
  - Boolean arithmetic
  - Loops and counting

C Preprocessor Tricks

- The C preprocessor language is not Turing-complete, but clever people will find a way.
- Instead of programming functions that follow general rules, explicit examplars can be defined for each possible combination of arguments.
- If you define enough examplars, it can look like the real thing from the programmer's perspective.

C Preprocessor Tricks

```
1  #define LIST_0(x)
2  #define LIST_1(x)              x##1
3  #define LIST_2(x)              LIST_1(x), x##2
4  #define LIST_3(x)              LIST_2(x), x##3
5  #define LIST_4(x)              LIST_3(x), x##4
6  #define LIST_5(x)              LIST_4(x), x##5
7  #define LIST_6(x)              LIST_5(x), x##6
8  #define LIST_7(x)              LIST_6(x), x##7
9  #define LIST_8(x)              LIST_7(x), x##8
10
11 #define LIST(cnt,x)            JOIN(LIST_, cnt)(x)
```

- `int LIST(3, X);` expands to `int X1, X2, X3;`
- It looks like something counted from 1 to 3, but nothing ever did.
- It would be absurd to use this technique in this specific case.
- “Don’t Repeat Yourself” is not an absolute.

Boost.Preprocessor

- The Boost.Preprocessor library elevates this to an art form.
- This sure looks like metaprogramming to me.

```
# define LINEAR_FIB_C(d, p) \
   /* p.n */ BOOST_PP_TUPLE_ELEM(3, 2, p) \
   /**/
#
# define LINEAR_FIB_F(d, p) \
   ( \
      /* p.a1 */ BOOST_PP_TUPLE_ELEM(3, 1, p), \
      /* p.a0 + p.a1 */ BOOST_PP_ADD_D(d, BOOST_PP_TUPLE_ELEM(3, 0, p), BOOST_PP_TUPLE_ELEM(3, 1, p)), \
                        /*          ^^ ^ \
                         * BOOST_PP_ADD() uses BOOST_PP_WHILE().  Therefore we \
                         * pass the recursion depth explicitly to BOOST_PP_ADD_D(). \
                         */ \
      /* p.n - 1 */ BOOST_PP_DEC(BOOST_PP_TUPLE_ELEM(3, 2, p)) \
   ) \
   /**/

int main() {
   printf("linear_fib(10) = %d\n", linear_fib(10));
   printf("LINEAR_FIB(10) = %d\n", LINEAR_FIB(10));
   return 0;
}
```

C Preprocessor

- Explaining how the tricks work:
  - http://www.altdevblogaday.com/2011/07/12/abusing-the-c-preprocessor/
- Boost::Preprocessor:
  - http://www.boostpro.com/mplbook/preprocessor.html

C++ Template Metaprogramming

A Tale of Serendipity

C++ Templates

```
template <typename T>
void printall (T Collection[], int size)
{
    int counter;
    for (counter=0; counter < size; ++counter)
        cout << Collection[counter] << endl;
}
int numbers[5] = {0,1,2,3,4};
printall(numbers, sizeof(numbers));
```

- Each time the compiler encounters a call to this template, it "writes" a new function based on the argument types.
- This addresses many problems raised by the equivalent C macro.
- The compiler has full visibility into this function and its signature. The template participates in the type system.
- The printall function is a real function at run-time.
- Full debugger support.

C++ Templates → Metaprogramming

- But is it metaprogramming? I don't think so. It's still just "filling in the blanks".
- Humorously, in 1994 someone discovered that the template definition subset of C++ was itself Turing complete.
- As a proof of concept, they made a program that never finished compiling, but produced a list of prime numbers in the compiler error message.
- Since then, the process has been formalized and beautified (sort of).
- It now enjoys first-class support in the C++ standard.

An Example Template Metaprogram

```
template <unsigned N>
struct Fibonacci
{
    enum
    {
        value = Fibonacci<N-1>::value + Fibonacci<N-2>::value
    };
};

template <>
struct Fibonacci<1>
{
    enum
    {
        value = 1
    };
};

template <>
struct Fibonacci<0>
{
    enum
    {
        value = 0
    };
};
```

```
int main(void)
{
    std::cout << "Fibonacci(15) = ";
    std::cout << Fibonacci<15>::value;
    std::cout << std::endl;
}
```

Boost.MPL

- The Boost Template Metaprogramming Library uses this technique as a basis for a full pure functional programming language-within-a-language.
  - Conditional statements
  - Sequences and loops
  - Compile-time algorithms
- Strangely, the only “values” supported by this language are C++ *types.*
- For example, the value 0 must be represented as a template class specialized with the value 0.
- If you can get over your distaste for the cumbersome syntax, the result is very powerful.

Embedded Domain-Specific Languages

- Ruby is well known as an excellent platform for development of embedded domain-specific languages (DSLs).
- C++ with template metaprogramming can give it a run for its money.

```
group      = '(' >> expression >> ')';
factor     = integer | group;
term       = factor >> *(('*' >> factor) | ('/' >> factor));
expression = term >> *(('+' >> term) | ('-' >> term));
```

```
//    Current   Event         Next      Action
//     State                  State
//  +---------+-----------+---------+----------------------+
row < Stopped , play       , Playing , &fsm::start_playback  >,
row < Stopped , open_close , Open    , &fsm::open_drawer     >,
//  +---------+-----------+---------+----------------------+
row < Paused  , play       , Playing , &fsm::resume_playback >,
row < Paused  , stop       , Stopped , &fsm::stop_playback   >,
row < Paused  , open_close , Open    , &fsm::stop_and_open   >,
//  +---------+-----------+---------+----------------------+
```

Drawbacks

- The syntax for C++ template metaprograms is arcane.
- Template metaprograms are arguably written in a very different language from C++ itself, even though it technically *is* C++.
- The skills and knowledge required to do C++ MPL well are different than those required for general C++ programming.
- C++ is a very complex language, and these template features interact with other language features in mysterious ways.
- Thousand-line long error messages!
- Long compile times.

C and C++ Metaprogramming

- There is a distinction between the domain language and the host language.
- The intent isn't to carry out general-purpose computation in the *domain* language – that is just a stunt.
- The intent is to facilitate the production of useful code in the *host* language.
- Metaprograms are usually ugly, or at least awkward. The ugly metaprograms can enable beautiful and expressive regular application programs.
- For these languages metaprogramming is only accidentally possible, but has proved to be very valuable anyway.

C++ Template Metaprogramming

- Boost.MPL
  - http://www.boost.org/doc/libs/1_51_0/libs/mpl/doc/index.html

Notes and References

Lisp Up-Side

- Paul Graham inspired a Lisp renaissance in the 2000s
  - “Beating the Averages” -- http://www.paulgraham.com/avg.html -- In which he claims that Lisp amplifies programmer power in a way that no other language can. And he coins the phrase “Blub language.”
  - "Revenge of the Nerds" -- http://www.paulgraham.com/icad.html -- In which he describes what is historically special about Lisp, and claims that the progress of modern programming languages is towards Lisp – except for concrete syntax.
  - “On Lisp” -- http://www.paulgraham.com/onlisp.html -- A book-length treatment of Lisp programming paradigms, with a very heavy emphasis on metaprogramming and embedded domain-specific languages.

Lisp Down-Side

- Rudolf Winestock – “The Lisp Curse” -- http://www.winestockwebdesign.com/Essays/Lisp_Curse.html

Ruby

- Ruby is an acceptable Lisp
  - Eric Kidd -- http://www.randomhacks.net/articles/2005/12/03/why-ruby-is-an-acceptable-lisp
  - Debashish Ghosh -- http://debasishg.blogspot.com/2007/01/syntax-extensibility-ruby.html
- Ruby is not an acceptable Lisp
  - Reg Braithwaite -- http://weblog.raganwald.com/2007/02/why-ruby-is-not-acceptable.html
  - Steve Yegge – https://sites.google.com/site/steveyegge2/digging-into-ruby-symbols

Ruby and AST Manipulation

- Abstract Syntax Tree manipulation gems
  - https://github.com/seattlerb/parsetree
  - http://rewrite.rubyforge.org/
  - http://docs.seattlerb.org/ruby2ruby/
- Discussion
  - Ruby AST for Fun and Profit -- http://www.igvita.com/2008/12/11/ruby-ast-for-fun-and-profit/
  - Macros, Hygiene, and Call By Name in Ruby -- http://weblog.raganwald.com/2008/06/macros-hygiene-and-call-by-name-in-ruby.html

Ruby vs. Python Metaprogramming

- Dhananjay Nene demonstrates that almost all Ruby metaprogramming techniques have Python equivalents:
  - http://codeblog.dhananjaynene.com/2010/01/dynamically-adding-methods-with-metaprogramming-ruby-and-python/