# Femtocode: development of a query language for HEP

Jim Pivarski

Princeton – DIANA

December 12, 2016

Specific motivation: to accelerate data pulls, both

| for the humans | | for the computers. |
|---|---|---|
| *shorter time from concept to code* | and | *shorter time from code to results* |

(By "data pull," I mean extract, transform, and filter data from a collaboration's central dataset (AOD) and download it for further processing. Right now, we're writing C++ modules.)

Specific motivation: to accelerate data pulls, both

| for the humans | | for the computers. |
|---|---|---|
| *shorter time from concept to code* | and | *shorter time from code to results* |

(By "data pull," I mean extract, transform, and filter data from a collaboration's central dataset (AOD) and download it for further processing. Right now, we're writing C++ modules.)

"Chip on my shoulder"

To show that a very high-level, abstract view of computation can nevertheless be fast.

# High-level and speed are not antithetical

Code like

```
bestmuon =
  muons.filter(m => m.iso > 10)
       .maxBy(m => m.pt)
```

does *not* need to create function objects or muon objects at runtime!

It need not be "taken literally."

Another possible execution plan:

1. Start with all `muon.iso` values in one array, all `muon.pt` values in another array, and a "repetition level" to specify where events begin and end.
2. Apply the contents of the filter function to make a mask.
3. Use the mask and repetition level to compact the `muon.pt` into zero or one results per event.

# How fast *should* it be?



**Suppose we want to plot all CMS $t\bar{t}$ data ever collected.**

- About 100 $\text{fb}^{-1}$ (plot) $\times$ 1 nb (plot) is 100 million events.
- Derivation of a plotted quantity could involve $\mathcal{O}(100)$ double-precision values per event.
- That's only 80 GB of data that needs to be evaluated; anything else is overhead.

*Single* computer, *single* thread, treat all data as an array:

load from disk: 21 minutes (Numpy), 22 minutes (ROOT)

CPU execution: 70 seconds (no optimization), 13 seconds (`-O3`)

GPU execution: 12 seconds to copy to GPU, 5 seconds to compute

(Probably close to "theoretical optimum." Distributed processing and multi-user cache can help.)

# Role of language

Directly filling histograms from the collaboration-wide repository, eliminating private skims, would fundamentally improve analysis workflows. Analysis could become "interactive."

But... how can a language help?

# Role of language

Directly filling histograms from the collaboration-wide repository, eliminating private skims, would fundamentally improve analysis workflows. Analysis could become "interactive."

But... how can a language help?

- Writing vectorized, one-array-at-a-time algorithms is unnatural, distracting from analysis.
- Physicists *should* be thinking about one event at a time and muons as objects.

Transforming human concepts into physical execution is exactly what programming languages are for.

# What's wrong with the languages we have?



As I said in my June 20 talk, C++ and Python are too *expressive* to permit these kinds of optimizations.

Automatic vectorization (in general) is an unsolved problem in computer science. But building a vectorizable language is relatively easy: they're just less capable. Typically, query languages like SQL are vectorized.

Business intelligence with SQL is typically "interactive," and big data projects are pushing $\mathcal{O}$(second) response times to petabyte scales: Ibis, Impala, Kudu, Drill, ... others? (Google paper).

**However, SQL and its relatives are not sufficient for us.**

**Example query:**

"Momentum of the track with $|\eta| < 2.4$ that has the most hits."

```
Track *best = NULL;
for (int i = 0;  i < tracks.size();  i++) {
  if (fabs(tracks[i]->eta) < 2.4)
    if (best == NULL ||
        tracks[i]->hits.size() > best->hits.size())
      best = tracks[i];
}
if (best != NULL)
  return best->pt;
else
  return 0.0;
```

# Nested query in SQL

**Example query:**

"Momentum of the track with $|\eta| < 2.4$ that has the most hits."

```
WITH hit_stats AS (
  SELECT hit.track_id, COUNT(*) AS hit_count FROM hit
    GROUP BY hit.track_id),
 track_sorted AS (
    SELECT track.*,
    ROW_NUMBER() OVER (
     PARTITION BY track.event_id
     ORDER BY hit_stats.hit_count DESC)
  track_ordinal FROM track INNER JOIN hit_stats
    ON hit_stats.track_id = track.id
    WHERE ABS(track.eta) < 2.4)
 SELECT * FROM event INNER JOIN track_sorted
   ON track_sorted.event_id = event.id
WHERE
  track_sorted.track_ordinal = 1
```

# Nested query in Femtocode

**Example query:**

"Momentum of the track with $|\eta| < 2.4$ that has the most hits."

```
tracks.filter(t => abs(t.eta) < 2.4)    # drop tracks
      .maxBy(t => t.hits.size)    # pick one (if any)
      .map(t => t.pt)             # transform it
      .impute(0.0)                # replace "None"
                                  # with a value
```

**Example query:**

"Momentum of the track with $|\eta| < 2.4$ that has the most hits."

```
tracks.filter(abs($1.eta) < 2.4)          # drop tracks
      .maxBy($1.hits.size)                # pick one (if any)
      .map($1.pt)                         # transform it
      .impute(0.0)                        # replace "None"
                                          # with a value
```

Combine the *implementation flexibility* of declarative languages like SQL with the *expressiveness* of a functional language for dealing with nested structure.

Combine the *implementation flexibility* of declarative languages like SQL with the *expressiveness* of a functional language for dealing with nested structure.

**Example use in Python:**

```
h = db.dataset("ttbar-MC")
   .withColumn(varName = "<Femtocode goes here>")
   .filter("<Femtocode using varName>")
   .flatMap("<Femtocode changing nesting level>")
   .Label(hist1 = Bin(100, -5.0, 5.0, "<Femtocode>"),
          hist2 = Bin(20, 0.0, 100.0, "<Femtocode>"),
          hist3 = Bin(314, -pi, pi, "<Femtocode>"))
```

followed by Python analysis on `h["hist1"]`, `h["hist2"]`...

Combine the *implementation flexibility* of declarative languages like SQL with the *expressiveness* of a functional language for dealing with nested structure.

**Example use in Python:**

```
h = db.dataset("ttbar-MC")
  .withColumn(varName = "<Femtocode goes here>")
  .filter("<Femtocode using varName>")
  .flatMap("<Femtocode changing nesting level>")
  .Label(hist1 = Bin(100, -5.0, 5.0, "<Femtocode>"),
         hist2 = Bin(20, 0.0, 100.0, "<Femtocode>"),
         hist3 = Bin(314, -pi, pi, "<Femtocode>"))
```

followed by Python analysis on `h["hist1"]`, `h["hist2"]`...

Python manipulates streams of events and accepts the result, while Femtocode (in quotes) operates on events (maybe remotely).

(Integrates with Histogrammar (`Label`, `Bin`) to aggregate result.)

# Working examples

```
>>> from femtocode.parser import parse
>>> print ast.dump(parse("""
... tracks.filter(abs($1.eta) < 2.4)
...         .maxBy($1.hits.size)
...         .map($1.pt)
...         .impute(0.0)
... """))
```

```
Suite(assignments=[], expression=FcnCall(function=Attribute(
    value=FcnCall(function=Attribute(value=FcnCall(function=
    Attribute(value=FcnCall(function=Attribute(value=Name(id='
    tracks', ctx=Load()), attr='filter', ctx=Load()),
    positional=[Compare(left=FcnCall(function=Name(id='abs',
    ctx=Load()), positional=[Attribute(value=AtArg(num=1), attr
    ='eta', ctx=Load())], names=[], named=[]), ops=[Lt()],
    comparators=[Num(n=2.4)])], names=[], named=[]), attr='
    maxBy', ctx=Load()), positional=[Attribute(value=Attribute(
    value=AtArg(num=1), attr='hits', ctx=Load()), attr='size',
    ctx=Load())], names=[], named=[]), attr='map', ctx=Load()),
     positional=[Attribute(value=AtArg(num=1), attr='pt', ctx=
    Load())], names=[], named=[]), attr='impute', ctx=Load()),
    positional=[Num(n=0.0)], names=[], named=[]))
```

# Working examples

```
>>> propagateTypes("""
... data.map(x => x + y)
... """,
... data=collection(integer), y=integer)
collection(integer)

>>> propagateTypes("""
... data.map(x => x + y)
... """,
... data=collection(integer), y=real)
collection(real)

>>> propagateTypes("""
... data.map(x => x + y)
... """,
... data=collection(integer, fewest=10, most=10),
... y=integer)
collection(integer, fewest=10, most=10)
```

## Propagate intervals of validity:

```
>>> propagateTypes("""
... data.map(x => x + y)
... """,
... data=collection(real(min=3, max=5)),
... y=real(min=100, max=200))
collection(real(min=103.0, max=205.0))
```

## Properly handle shadowed variable “x”:

```
>>> propagateTypes("""
... y = x + -100;
... data.map(x => x + y)
... """,
... data=collection(real(min=3, max=5)),
... x=real(min=100, max=200))
collection(real(min=3.0, max=105.0))
```

# Type system is highly granular

Collections have fewest and most number of elements, and numbers have min and max intervals:

- `real(almost(0), 10)` $\{x|x \in \mathbb{R} \text{ and } 0 < x \leq 10\}$
- `integer(almost(-inf), almost(inf))` $\mathbb{Z}$
- `extended(-inf, inf)` $\mathbb{R} \cup \{-\infty, \infty\}$
- `union(integer, real(0))` $\mathbb{Z} \cup \{x|x \in \mathbb{R} \text{ and } x \geq 0\}$

# Type system is highly granular

Collections have fewest and most number of elements, and numbers have min and max intervals:

- `real(almost(0), 10)` $\{x|x \in \mathbb{R} \text{ and } 0 < x \leq 10\}$
- `integer(almost(-inf), almost(inf))` $\mathbb{Z}$
- `extended(-inf, inf)` $\mathbb{R} \cup \{-\infty, \infty\}$
- `union(integer, real(0))` $\mathbb{Z} \cup \{x|x \in \mathbb{R} \text{ and } x \geq 0\}$

**Why?**

To eliminate the possibility of runtime errors.

With enough information in the type system, the compiler can identify runtime errors before submitting the job, saving the author time and protecting shared resources from waste.

(Someday, they might be cloud-based and cost real money.)

# Working examples

```
>>> propagateTypes("x / y", x=real, y=real)

femtocode.parser.FemtocodeError: Function "/" does not accept
    arguments with the given types:

    /(real,
      real)

    Indeterminate form (0 / 0) is possible; constrain with if-
        else.

Check line:col 1:0 (pos 0):

    x / y
----^
```

# Working examples

```
>>> propagateTypes("x / y", x=real, y=real)

femtocode.parser.FemtocodeError: Function "/" does not accept
    arguments with the given types:

    /(real,
      real)

    Indeterminate form (0 / 0) is possible; constrain with if-
        else.

Check line:col 1:0 (pos 0):

    x / y
----^
```

Applying a constraint changes the type of "y" in the "if" clause to `union(real(max=almost(0)), real(min=almost(0)))`.

```
>>> propagateTypes("if y != 0: x / y else: None",
...      x=real, y=real)
union(null, real)
```

# Working examples

"if", "and", "or", and "not" propagate constraints.

```
>>> propagateTypes("x == 5 and y == 6 and x == y",
... x=real, y=real)

femtocode.parser.FemtocodeError: Function "==" does not accept
    arguments with the given types:

    ==(integer(min=5, max=5),
       integer(min=6, max=6))

    The argument types have no overlap (values can never be
        equal).

Check line:col 1:27 (pos 27):

    x == 5 and y == 6 and x == y
-------------------------------^
```

## Order does not matter.

```
>>> propagateTypes("x == y and x == 5 and y == 6",
... x=real, y=real)

femtocode.parser.FemtocodeError: Function "==" does not accept
    arguments with the given types:

    ==(integer(min=5, max=5),
       integer(min=6, max=6))

    The argument types have no overlap (values can never be
        equal).

Check line:col 1:5 (pos 5):

    x == y and x == 5 and y == 6
---------^
```

## Order does not matter.

```
>>> propagateTypes("x == y and x == 5 and y == 6",
... x=real, y=real)

femtocode.parser.FemtocodeError: Function "==" does not accept
    arguments with the given types:

    ==(integer(min=5, max=5),
       integer(min=6, max=6))

    The argument types have no overlap (values can never be
        equal).

Check line:col 1:5 (pos 5):

    x == y and x == 5 and y == 6
---------^
```

(It's a short step from here to simplifying the algebra with SymPy.)

# Working examples

```
>>> viewAsTree("""
... a = x + y;
... b = a + y + z;
... xs.map(x => x + a + a + b).map(y => y + 2)""",
... xs=collection(real), x=real, y=real, z=real)
```

```
Call BuiltinFunction[".map"] has type collection(real)
    Call BuiltinFunction[".map"] has type collection(real)
        Ref xs (frame None) has type collection(real)
        UserFunction has type real
            Call BuiltinFunction["+"] has type real
                Call BuiltinFunction["+"] has type real
                    Call BuiltinFunction["+"] has type real
                        Ref x (frame 2) has type real
                        Call BuiltinFunction["+"] has type real
                            Ref x (frame None) has type real
                            Ref y (frame None) has type real
                    Call BuiltinFunction["+"] has type real
                        Ref x (frame None) has type real
                        Ref y (frame None) has type real
                Call BuiltinFunction["+"] has type real
                    Call BuiltinFunction["+"] has type real
                        Call BuiltinFunction["+"] has type real
                            Ref x (frame None) has type real
                            Ref y (frame None) has type real
                        Ref y (frame None) has type real
                    Ref z (frame None) has type real
    UserFunction has type real
        Call BuiltinFunction["+"] has type real
            Ref y (frame 3) has type real
            Literal 2 has type integer(min=2, max=2)
```

Notice that “a” and “b” do not appear.

# Working examples

```
>>> viewAsStatements("""
... a = x + y;
... b = a + y + z;
... xs.map(x => x + a + a + b).map(y => y + 2)""",
... xs=collection(real), x=real, y=real, z=real)
```

```
tmp_0 := (+ x y)
tmp_1 := (+ xs tmp_0)
tmp_2 := (+ tmp_1 tmp_0)
tmp_3 := (+ tmp_0 y)
tmp_4 := (+ tmp_3 z)
tmp_5 := (+ tmp_2 tmp_4)
tmp_6 := (+ tmp_5 2)
```

Repeated calculations have been rolled into `tmp_*` for execution.

This statement-generation would be even better if it minimized the length of time variables need to stay alive, while maintaining dependency order, so that arrays can be overwritten in-place.

# Status and plans

Done:

- Syntax, parsing, abstract syntax tree.
- Type system, type propagation, type inference for $+$, $-$, $*$, $/$, $//$, $**$, $\%$.
- Type constriction in "if", "and", "or", and "not".
- Prototype for generating statements.

To do:

- Fully generate statements and evaluate on arrays.
- Implement a few more built-in functions for basic usability.
- Focus-group the syntax and scope: will this work for busy physicists?
- Get more feedback from Brian, Philippe, and other experts.
- Work with Jin Chang and Igor Mandrichenko on the server.
- Discuss with ROOT Team about the possibility of this becoming the ROOT 7 TTreeFormula.

# BACKUP

# Features of Femtocode

**Declarative:** order written/order evaluated need not be the same.

**Functional:** map/filter/maxBy instead of explicit for loops.

**Vectorizable:** code appears to act on rows (e.g. events), but automatically translated to operate on columns.

**No unbounded loops:** execution time strictly scales with input data size; not Turing complete.

**No runtime errors:** any compilable query will return some result.

**Statically typed:** stronger type system than most languages is needed to eliminate runtime errors.

**Full type inference:** explicitly writing down types is annoying.

**No recursion:** combining recursion with no unbounded loops is complicated, but big data pulls don't need recursion.

**Pythonic syntax:** familiar to physics users; don't invent new syntax unless absolutely necessary.

## BNF specification of Femtocode syntax

```
body: ';'* suite
suite: (assignment ';'*)* expression ';'*
lvalues: (NAME ',')* NAME [',']
assignment: (lvalues '=' closed_expression
              | fcnndef)
fcnndef: ('def' NAME '(' [paramlist] ')'
            closed_exprsuite)
expression: ifblock | fcndef | or_test
closed_expression: (closed_ifblock | fcndef
                     | or_test ';')
fcndef: '{' [paramlist] '=>' ';'* suite '}'
fcn1def: parameter '=>' expression
paramlist: (parameter ',')* (parameter [','])
parameter: NAME ['=' expression]
exprsuite: (':' expression
             | [':'] '{' ';'* suite '}')
closed_exprsuite: (':' closed_expression
             | [':'] '{' ';'* suite '}')
ifblock: ('if' expression exprsuite
            ('elif' expression exprsuite)*
             'else' exprsuite)
closed_ifblock: ('if' expression exprsuite
            ('elif' expression exprsuite)*
             'else' closed_exprsuite)
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: typecheck (comp_op typecheck)*
comp_op: ('<' | '>' | '==' | '>=' | '<='
             | '!=' | 'in' | 'not' 'in')
typecheck: (arith_expr ['is' arith_expr
             | 'is' 'not' arith_expr])
arith_expr: term (('+' | '-') term)*
term: factor (('*' | '/' | '%' | '//') factor)*
factor: ('+' | '-') factor | power
power: atom trailer* ['**' factor]
atom: ('(' expression ')'
      | ('[' (expression ',')*
             [expression [',']] ']')
       | fcndef '(' [arglist] ')'
      | MULTILINESTRING
       | STRING
       | IMAG_NUMBER
       | FLOAT_NUMBER
       | HEX_NUMBER
       | OCT_NUMBER
       | DEC_NUMBER
      | ATARG
      | NAME)
trailer: ('(' [arglist] ')'
           | '[' subscriptlist ']' | '.' NAME)
subscriptlist: subscript (',' subscript)* [',']
subscript: (expression
           | [expression] ':' [expression]
             [sliceop])
sliceop: ':' [expression]
arglist: (((argument ',')* (argument [',']))
           | fcn1def)
argument: expression | NAME '=' expression
```

## BNF specification of Femtocode syntax that is identical to Python

```
body: ';'* suite
suite: (assignment ';'*)* expression ';'*
lvalues: (NAME ',')* NAME [',']
assignment: (lvalues '=' closed_expression
              | fcnndef)
fcnndef: ('def' NAME '(' [paramlist] ')'
            closed_exprsuite)
expression: ifblock | fcndef | or_test
closed_expression: (closed_ifblock | fcndef
                     | or_test ';')
fcndef: '{' [paramlist] '=>' ';'* suite '}'
fcn1def: parameter '=>' expression
paramlist: (parameter ',')* (parameter [','])
parameter: NAME ['=' expression]
exprsuite: (':' expression
             | [':'] '{' ';'* suite '}')
closed_exprsuite: (':' closed_expression
             | [':'] '{' ';'* suite '}')
ifblock: ('if' expression exprsuite
            ('elif' expression exprsuite)*
             'else' exprsuite)
closed_ifblock: ('if' expression exprsuite
            ('elif' expression exprsuite)*
             'else' closed_exprsuite)
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: typecheck (comp_op typecheck)*
comp_op: ('<' | '>' | '==' | '>=' | '<='
             | '!=' | 'in' | 'not' 'in')
typecheck: (arith_expr ['is' arith_expr
             | 'is' 'not' arith_expr])
arith_expr: term (('+' | '-') term)*
term: factor (('*' | '/' | '%' | '//') factor)*
factor: ('+' | '-') factor | power
power: atom trailer* ['**' factor]
atom: ('(' expression ')'
      | ('[' (expression ',')*
             [expression [',']] ']')
       | fcndef '(' [arglist] ')'
      | MULTILINESTRING
       | STRING
       | IMAG_NUMBER
       | FLOAT_NUMBER
       | HEX_NUMBER
       | OCT_NUMBER
       | DEC_NUMBER
      | ATARG
      | NAME)
trailer: ('(' [arglist] ')'
           | '[' subscriptlist ']' | '.' NAME)
subscriptlist: subscript (',' subscript)* [',']
subscript: (expression
           | [expression] ':' [expression]
             [sliceop])
sliceop: ':' [expression]
arglist: (((argument ',')* (argument [',']))
           | fcn1def)
argument: expression | NAME '=' expression
```

## How is it like Python?

- mathematical expressions and operator precedence:
```
(-b + sqrt(b**2 -
           4*a*c))/(2*a)
```
- slices and 0-indexing:
```
lheweights[::2]
```
- numbers and string literals (favoring Python 3):
```
0xff, 0o77, .3e7, 1j,
"""multi \"line\"
string"""
```
- chained comparisons:
```
0 < x <= 10
```
- keyword arguments:
```
f(arg, some=kwd)
```

## How is it different?

- anonymous functions:
```
{x, y => x + y}
{$1 + $2}
```
- no statements and whitespace independent:
```
if something:
    doIfTrue()
    else:
 doIfFalse()
```
- curly-bracketed blocks with semicolon-separated assignments ending in a single expression.
```
if something
{doIfTrue()} else
{doIfFalse()}
```

# Abstract type system

Space of possible datasets is defined by the type system.

**null:** type with only one value

**boolean:** not the same as integers

**number:** further defined by attributes: min, max, whole
`whole == True` means integer,
`whole == False` means floating point

**string:** charset ("bytes" or "unicode"), fewest, most;
fewest/most constrain the string length

**collection:** fewest, most, ordered
fixed-size arrays/matrices have `fewest == most and ordered == True`

**record:** defined by a dictionary of fields

**union:** tagged union, such as `union(null, string)` for a nullable string type.

# Semantics

Every Femtocode "program" is conceptually a single expression: that which should be computed from the input fields.

- when called in a `filter`, it selects events,
- in `map`, it transforms,
- in `flatMap`, it restructures,
- in `withColumn`, it adds a field to the output,
- in Histogrammar quantities, it gets aggregated, probably for plotting.

Most Femtocode snippets are small enough for this to be obvious!

Assignments are provided *as a convenience.*

```
goodVertex = abs(z) < 3;
goodPt = pt > 20;
goodIsolation = iso > 12;
goodVertex and goodPt and goodIsolation
```

Expression ASTs are literally inserted where they are referenced (handling shadowed variable names appropriately, with lexical scope).

This is legal because Femtocode has perfect referential transparency. (All variables are immutable, no side-effects, no exceptions or non-halting functions.)

The same is true of user-defined functions. Moreover, arguments might have different types in different calls.

```
def nonempty(x) {
  x.size > 0
}

nonempty(list) or nonempty(str)
```

When applied to a collection, `nonempty` takes the collection type, when applied to a string, `nonempty` takes the string type.

Types are propagated independently through the function's body with each call. (Thus, it deals with ugly unions transparently.)

(This is what Julia does when you don't provide type annotations.)

So what about recursion? What's its type?

```
def listsum(x) {
  if nonempty(x):
    x[0] + listsum(x[1:])
  else:
    0
}

listsum(list)
```

It can't always be determined, and we want to eliminate infinite loops anyway, so we simply don't allow recursion.

The rule against recursion applies equally to assignment (like a zero-argument function).

```
x = x + 1
```

(If we attempted to interpret the above, we'd have to conclude that `x` is `inf` or `-inf`. Probably not what the user intended.)

Although not logically necessary, we require values and functions to be defined before they are used.

```
goodParticle = goodVertex and goodPt and
    goodIsolation;
goodVertex = abs(z) < 3;
goodPt = pt > 20;
goodIsolation = iso > 12;
```

and

```
def invmass(p4): sqrt(energy(p4)**2 - momentum
    (p4)**2);
def energy(p4): p4[0];
def momentum(p4): sqrt(p4[1]**2 + p4[2]**2 +
    p4[3]**2);
```

are not allowed. The first would likely be a user mistake.

# Second-class functions

In some languages, functions are "first class" (in the sense of "first class citizens") because they can be treated as values, just like numbers or strings. Femtocode is not such a language.

# Second-class functions

In some languages, functions are "first class" (in the sense of "first class citizens") because they can be treated as values, just like numbers or strings. Femtocode is not such a language.

After all assignments have been expanded, anonymous functions and function names can only appear in the arguments of built-in functions that expect them.

```
def goodEta(t): abs(t.eta) < 2.4;
tracks.filter(goodEta)
      .maxBy(t => t.hits.size)
      .map($1.pt)
```

`goodEta`, `t => t.hits.size`, and `$1.pt` can only appear in the appropriate argument slot of functions like `.filter`, `.maxBy`, and `.map`.

And built-in functions never return functions.

# Second-class functions



However, functions can be assigned

```
goodEta = {t => abs(t.eta) < 2.4};
```

and passed as the return value of a user-defined function

```
def cutEta(cut):
  {t => abs(t.eta) < cut};
goodEta = cutEta(2.4);
```

because these constructs are expanded before any types are checked. It gives the user the feeling of freedom when working with functions when they are actually constrained.

The only aspect of first-class functions that the user might miss is the ability to pick a function to call at runtime.

# Why does this matter?

Since unevaluated functions only appear in arguments to `.filter`, `.maxBy`, and `.map`, etc., they are no more powerful than a "for" loop body.

For instance,

```
goodEta = {t => abs(t.eta) < 2.4};
data.filter(goodEta)
```

could be implemented by

```
[t for t in data if abs(t.eta) < 2.4]
```

In C terminology, all functions can be "inlined." That is to say, the exact code needed to execute them is known at compile-time and can be literally inserted if desired.

# Vectorization

It is in general difficult to "vectorize" code: that is, convert code that operates on individual rows of data (events) to instead operate on columns.

- The row-based view is more natural to the data analyst.
- But the column-based implementation is often faster.

# Vectorization

It is in general difficult to "vectorize" code: that is, convert code that operates on individual rows of data (events) to instead operate on columns.

- The row-based view is more natural to the data analyst.
- But the column-based implementation is often faster.

This is the difficulty of porting algorithms from CPU to GPU: the GPU is a 32 or 64 lane wide vector machine.

It is also the difficulty of porting pure Python to Numpy. Or "for" loops in R into efficient "lapply."

Even when the CPU is the target, modern CPUs have ∼4 lane wide vector registers and can prefetch memory better when operating on columns (circumventing the primary bottleneck in most calculations).

# Femtocode is vectorizable

Femtocode's restriction on functions allows it to be vectorizable in a way that C++ and Python aren't.

For example, a collection of 1000 events may have 10,000 showers. If the showers' `E2` (one per shower) is an array of length 10,000 and the events' `pedistal` is an array of length 1000, we can't perform element-wise calculations on `E2` and `pedistal`.

However, we can do this:

```
showers.map({s => sqrt(s.E2)}).max - pedistal
```

which translates into:

1. `tmp1 = sqrt(E2)` — 10,000 operations
2. `tmp2 = max(tmp1)` — stream compaction
3. `tmp2 - pedistal` — 1000 operations

# Intermediate representation

Femtocode "compiles" its expressions into a sequence of vector statements and sends them to an execution engine for calculation.

```
{"version": "1.0",
 "dataset": "ttbar-MC",
 "operations": [
   {"filter": [
     {"fcn": ">", "args": ["MET", ["Literal", 20]],
      "type": ["Boolean"], "deps": ["MET"]}]},
   {"withColumn": {"varName": [
     {"fcn": "+", "args": ["a", "b"], "to": "tmp1",
      "type": ["Number", 0, ["almost", "inf"], true],
      "deps": ["a", "b"]},
     {"fcn": "sqrt", "args": ["tmp1"],
      "type": ["Number", 0, ["almost", "inf"], false],
      "deps": ["a", "b", "tmp1"]}]}},
   {"histogrammar": ["Bin", 100, 0, 20, ["varName"],
     "Count", "Count", "Count", "Count"]}
 ]}
```

# Execution engine may reorder operations

Although the execution engine receives a list of operations, each containing a list of assignment statements, it is free to change their order as long as the dependencies (`"deps"`) are satisfied.

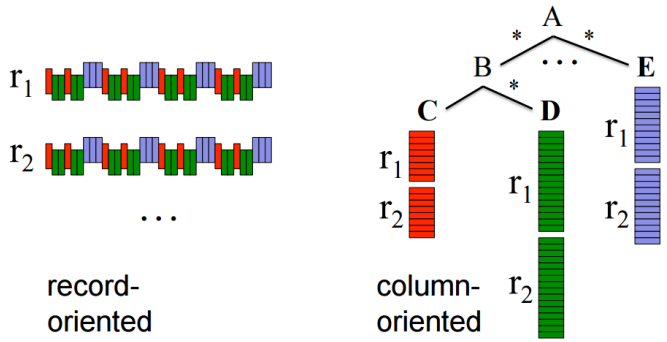
# Execution engine may reorder operations

Although the execution engine receives a list of operations, each containing a list of assignment statements, it is free to change their order as long as the dependencies (`"deps"`) are satisfied.

- Some filters may be more effective than others, depending on the distribution of data.
- One column at a time (Numpy style) may be more effective than JIT-compiling a few operations together (Numexpr style), or vice-versa, depending on register pressure, cache misses, allocation and copy overhead, etc.
- Statements that use the same column as input could be combined to avoid multiple memory fetches.
- The last use of a column may be operated upon in-place.
- If allocations and deallocations can be arranged as a stack, `malloc` alternatives like Obstack may be used.
- Any JIT code must be written in a way that permits vectorization, whether in a CPU, a GPU, or Xeon Phi.

# Data representation

For all conceivable backends, data will be stored in columnar arrays, probably uncompressed.




I've gone back and forth on this, but now I see that Parquet's definition and repetition levels is the most elegant solution:

https://blog.twitter.com/2013/dremel-made-simple-with-parquet

Each group of columns with the same multiplicity would have one "repetition level" array that indicates where collections and subcollections start and end. Only one repetition level array is needed for arbitrarily deep nesting: deeper repetition levels lead to higher integer values in the repetition level array. In principle, subcollections-within-collections could be as deep as $2^{64}$.