Programming languages and particle physics

Jim Pivarski

Princeton University – IRIS-HEP

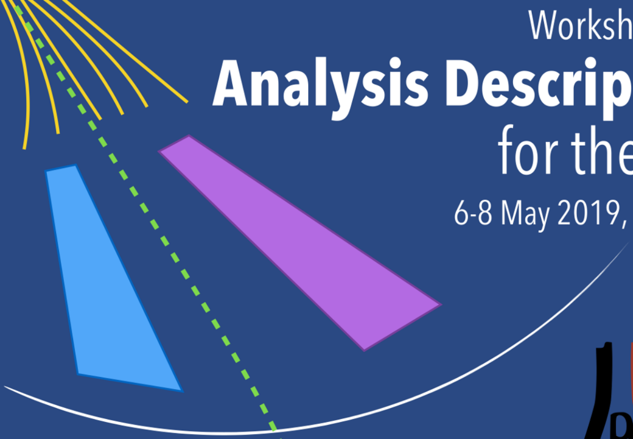
May 8, 2019

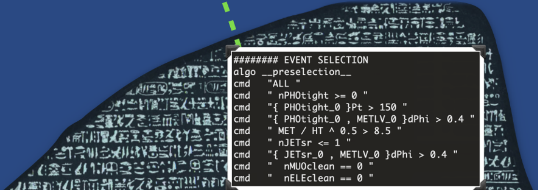
Workshop on

Analysis Description Languages for the LHC

6-8 May 2019, Fermilab LPC

https://indico.cern.ch/event/769263/





```
######## EVENT SELECTION
algo __preselection__
cmd   "ALL "
cmd   " nPHOtight >= 0 "
cmd   "{ PHOtight_0 }Pt > 150 "
cmd   "{ PHOtight_0 , METLV_0 }dPhi > 0.4 "
cmd   " MET / HT ^ 0.5 > 8.5 "
cmd   " nJETsr <= 1 "
cmd   "{ JETsr_0 , METLV_0 }dPhi > 0.4 "
cmd   "  nMUOclean == 0 "
cmd   "  nELEclean == 0 "
```

An analysis description language (ADL) is a human readable declarative language that unambiguously describes the contents of an analysis in a standard way, independent of any computing framework.

Adopting ADLs would bring numerous benefits for the LHC experimental and phenomenological communities, ranging from analysis preservation beyond the lifetimes of experiments or analysis software to facilitating the abstraction, design, visualization, validation, combination, reproduction, interpretation and overall communication of the

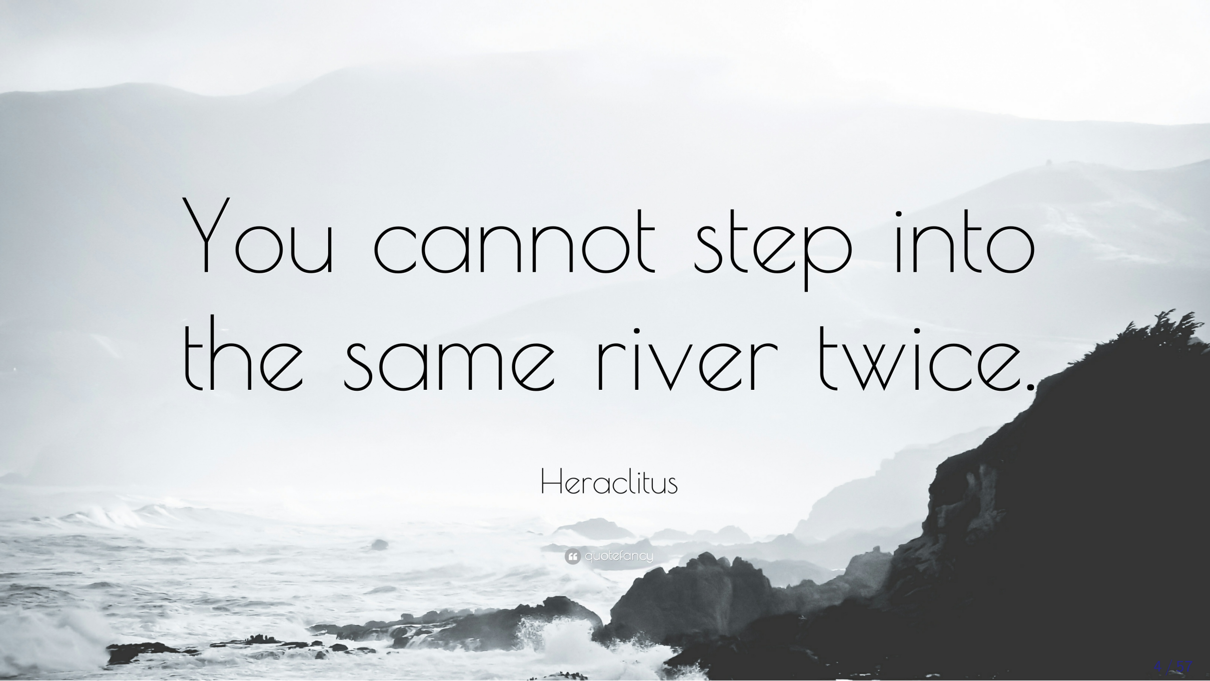
But that just ended a few minutes ago.

(This talk is not a summary of the workshop;
come to tomorrow's LPC Physics Forum at 1:30pm.)

But that just ended a few minutes ago.

(This talk is not a summary of the workshop;
come to tomorrow's LPC Physics Forum at 1:30pm.)

Instead, let's take a step back. . .



You cannot step into the same river twice.

Heraclitus

Because, you know, it's different water.

So why do we say it's the same river?

Why do we say it's the same river?

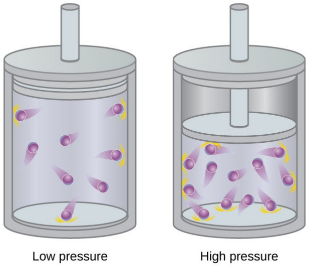
The river is an abstraction.

We associate an enormous number of microscopic states ("molecules here, molecules there") with a single macroscopic state ("the river").

Why do we say it’s the same river?

The river is an abstraction.

We associate an enormous number of microscopic states (“molecules here, molecules there”) with a single macroscopic state (“the river”).



It’s an abstraction like thermodynamics;
it can be exact with the right definitions.

Most of computer science is about abstracting details, too.

```
double bessel_j0(double x) {
    double out;
    if (fabs(x) < 8.0) {
        double y = x*x;
        double ans1 = 57568490574.0 + y*(-13362590354.0 + y*(651619640.7
                      + y*(-11214424.18 + y*(77392.33017 + y*(-184.9052456)))));
        double ans2 = 57568490411.0 + y*(1029532985.0 + y*(9494680.718
                      + y*(59272.64853 + y*(267.8532712 + y*1.0))));
        out = ans1 / ans2;
    }
    else {
        double z = 8.0 / fabs(x);
        double y = z*z;
        double xx = fabs(x) - 0.785398164;
        double ans1 = 1.0 + y*(-0.1098628627e-2 + y*(0.2734510407e-4
                      + y*(-0.2073370639e-5 + y*0.2093887211e-6)));
        double ans2 = -0.1562499995e-1 + y*(0.1430488765e-3
                      + y*(-0.6911147651e-5 + y*(0.7621095161e-6
                      - y*0.934935152e-7)));
        out = sqrt(0.636619772/fabs(x))*(cos(xx)*ans1 - z*sin(xx)*ans2);
    }
    return out;
}
```

Most of computer science is about abstracting details, too.

```
double bessel_j0(double x) {            ← one value goes in
    double out;
    if (fabs(x) < 8.0) {
        double y = x*x;
        double ans1 = 57568490574.0 + y*(-13362590354.0 + y*(651619640.7
                      + y*(-11214424.18 + y*(77392.33017 + y*(-184.9052456)))));
        double ans2 = 57568490411.0 + y*(1029532985.0 + y*(9494680.718
                      + y*(59272.64853 + y*(267.8532712 + y*1.0))));
        out = ans1 / ans2;
    }
    else {
        double z = 8.0 / fabs(x);
        double y = z*z;
        double xx = fabs(x) - 0.785398164;
        double ans1 = 1.0 + y*(-0.1098628627e-2 + y*(0.2734510407e-4
                      + y*(-0.2073370639e-5 + y*0.2093887211e-6)));
        double ans2 = -0.1562499995e-1 + y*(0.1430488765e-3
                      + y*(-0.6911147651e-5 + y*(0.7621095161e-6
                      - y*0.934935152e-7)));
        out = sqrt(0.636619772/fabs(x))*(cos(xx)*ans1 - z*sin(xx)*ans2);
    }
    return out;
}
```

Most of computer science is about abstracting details, too.

```
double bessel_j0(double x) {
    double out;
    if (fabs(x) < 8.0) {
        double y = x*x;
        double ans1 = 57568490574.0 + y*(-13362590354.0 + y*(651619640.7
                      + y*(-11214424.18 + y*(77392.33017 + y*(-184.9052456)))));
        double ans2 = 57568490411.0 + y*(1029532985.0 + y*(9494680.718
                      + y*(59272.64853 + y*(267.8532712 + y*1.0))));
        out = ans1 / ans2;
    }
    else {
        double z = 8.0 / fabs(x);
        double y = z*z;
        double xx = fabs(x) - 0.785398164;
        double ans1 = 1.0 + y*(-0.1098628627e-2 + y*(0.2734510407e-4
                      + y*(-0.2073370639e-5 + y*0.2093887211e-6)));
        double ans2 = -0.1562499995e-1 + y*(0.1430488765e-3
                      + y*(-0.6911147651e-5 + y*(0.7621095161e-6
                      - y*0.934935152e-7)));
        out = sqrt(0.636619772/fabs(x))*(cos(xx)*ans1 - z*sin(xx)*ans2);
    }
    return out;
}
```

$\leftarrow$ one value goes in

$\leftarrow$ one value comes out

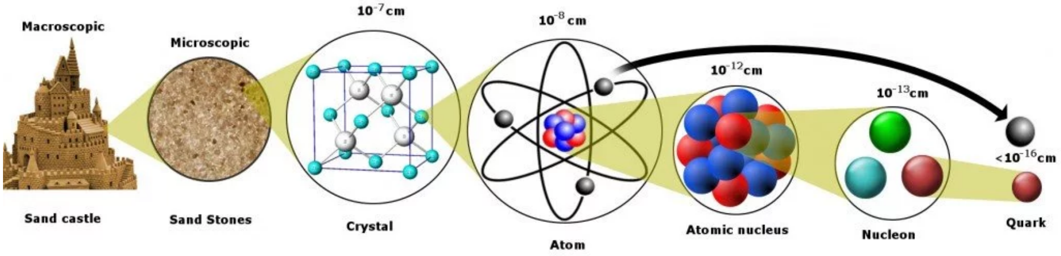
The abstraction is cumulative:

Every function/class/module has an interior and an interface—minimizing

$$\frac{\#\text{external parameters}}{\#\text{internal parameters}}$$

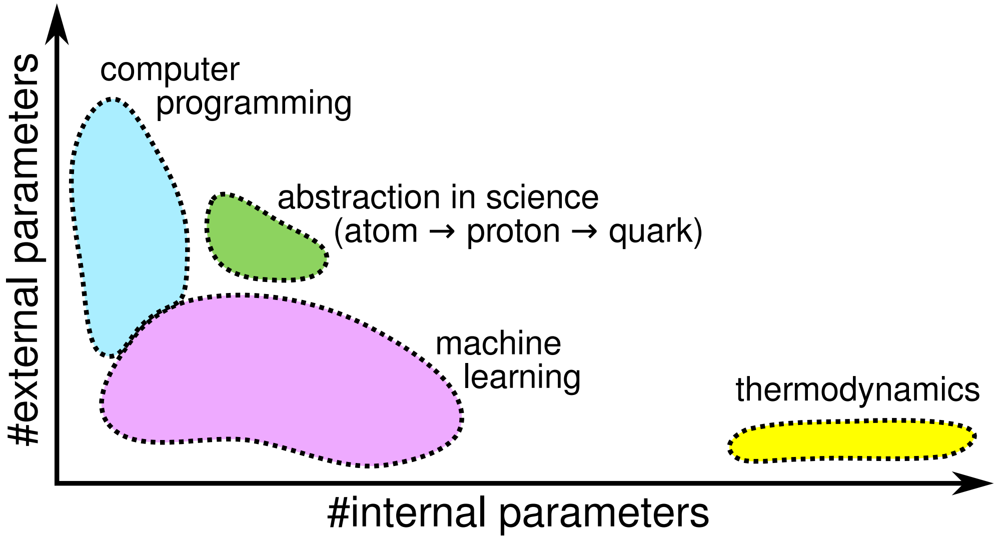
reduces the mental burden on programmers and users.

Science has layers of abstraction



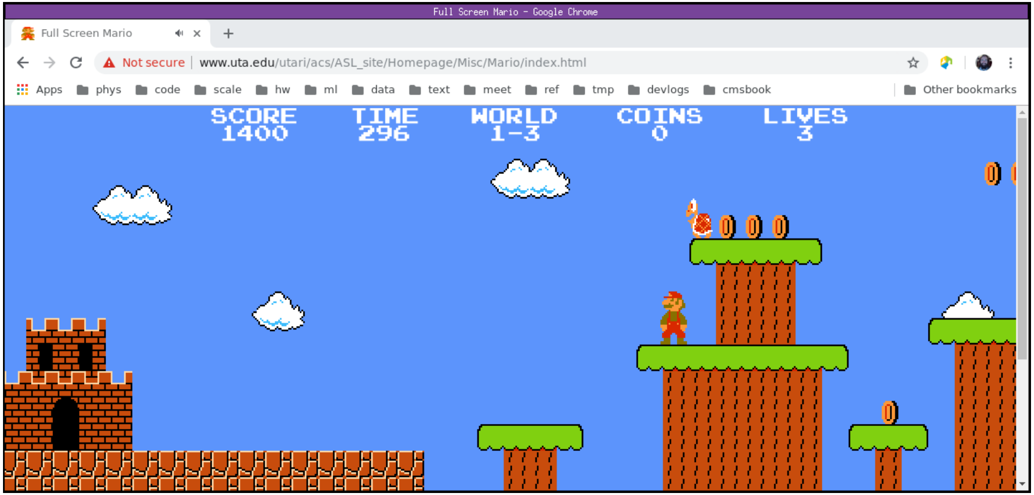
These are approximate, taking advantage of a separation of scales.

(cartoon diagram, not to scale)



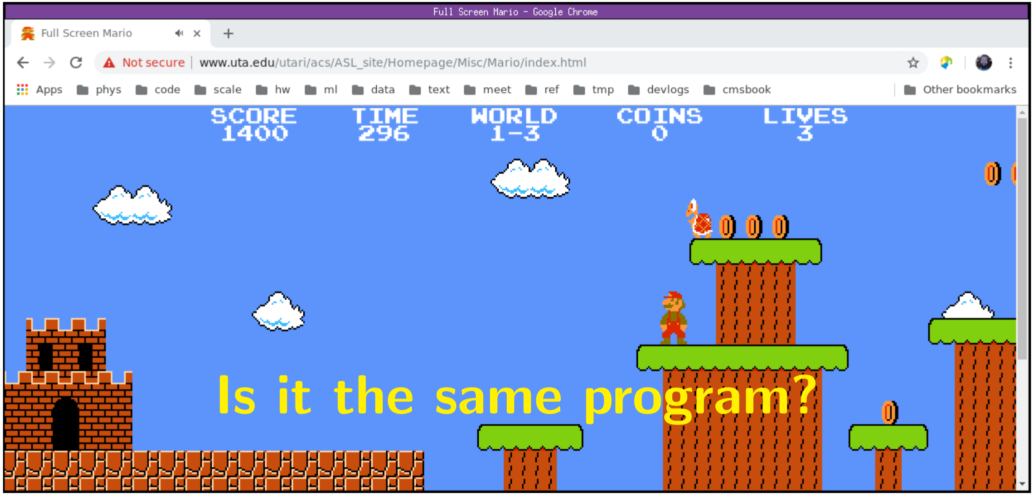
Software interfaces can be exact, despite radical internal differences.

- Super Mario Bros. entirely rewritten in Javascript by Josh Goldberg.
- Shares none of the original code, but behaves identically.

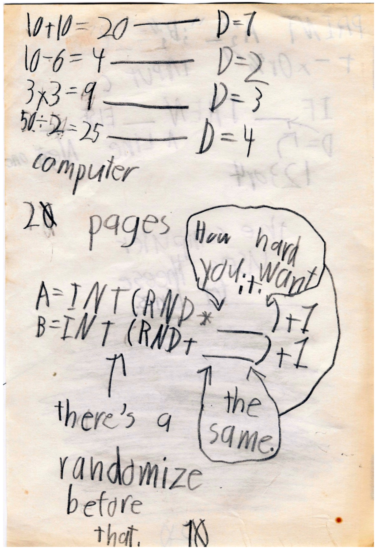


Software interfaces can be exact, despite radical internal differences.

- Super Mario Bros. entirely rewritten in Javascript by Josh Goldberg.
- Shares none of the original code, but behaves identically.


Full Screen Mario - Google Chrome
Full Screen Mario
Not secure
www.uta.edu/utari/acs/ASL_site/Homepage/Misc/Mario/index.html
Apps phys code scale hw ml data text meet ref tmp devlogs cmsbook Other bookmarks
SCORE 1400
TIME 296
WORLD 1-3
COINS 0
LIVES 3
Is it the same program?

As a young programmer, I wasn't satisfied with high-level languages because I wanted to get down to the "real" computer.



10+10=20 —— D=1
10-6=4 —— D=2
3x3=9 —— D=3
50÷2=25 —— D=4
computer

20 pages

How hard you want it

```
A=INT(RND*____)+1
B=INT(RND*____)+1
```

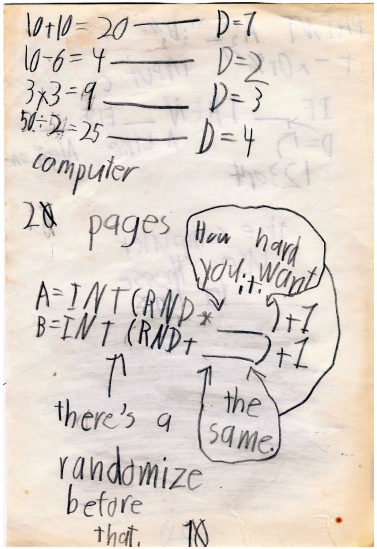
there's a randomize before that.

the same.

10

As a young programmer, I wasn't satisfied with high-level languages because I wanted to get down to the "real" computer.

Which meant Pascal.
Pascal was "real," and BASIC was not.



10+10=20 —— D=1
10-6=4 —— D=2
3x3=9 —— D=3
50÷2=25 —— D=4
computer

20 pages

How hard you want it.

```
A=INT(RND*___)+1
B=INT(RND+___)+1
```

there's a randomize before that.

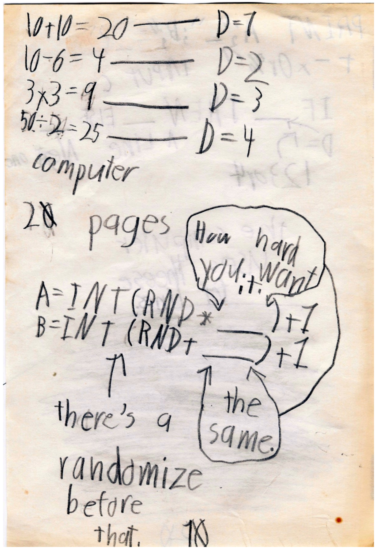
the same.

10

As a young programmer, I wasn't satisfied with high-level languages because I wanted to get down to the "real" computer.

Which meant Pascal.
Pascal was "real," and BASIC was not.

But ultimately, not even assembly code is real in the sense that I'm meaning here.



10+10=20 —— D=1
10-6=4 —— D=2
3x3=9 —— D=3
50÷2=25 —— D=4
computer

20 pages

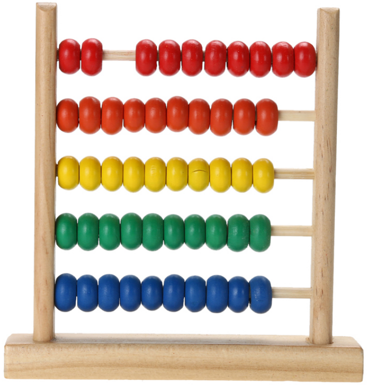
How hard you want it

A=INT(RND* ——)+1
B=INT(RND+ ——)+1

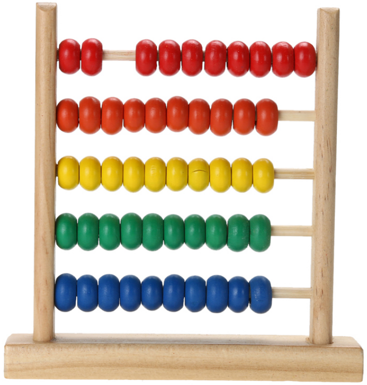
there's a randomize before that. 10

the same

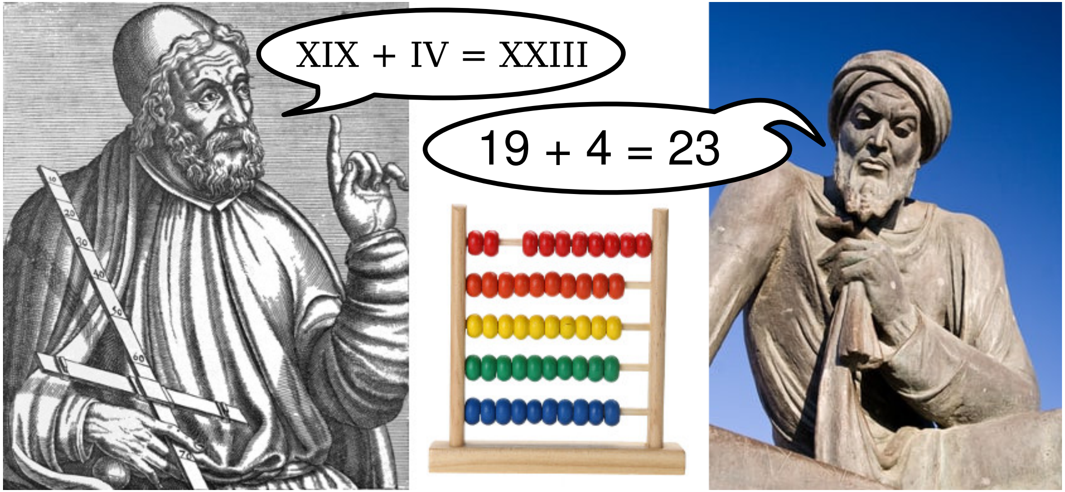
The objectively real part of a computer is a set of physical states.



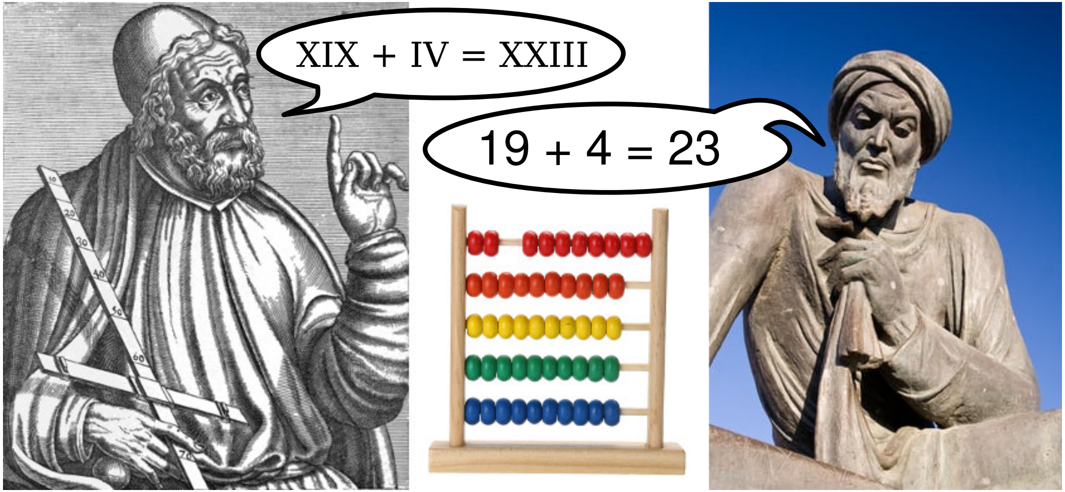
The objectively real part of a computer is a set of physical states *that we interpret* as computations.



Programming languages are how we *describe* our interpretations.



Programming languages are how we *describe* our interpretations.



(And some languages are better at it than others.)

Programming languages differ in their degree of abstraction,
but *all* programming languages are for humans, not computers.

Programming languages differ in their degree of abstraction,
but *all* programming languages are for humans, not computers.

Each one re-expresses the programmer's intent in terms of another:

CMSSW configuration	implemented in	Python runtime
Python runtime	implemented in	C source code
C source code	compiled into	machine instructions
machine instructions	built into	logic gates
logic gates	interpreted as	computation.

Programming languages differ in their degree of abstraction,
but *all* programming languages are for humans, not computers.

Each one re-expresses the programmer's intent in terms of another:

CMSSW configuration	implemented in	Python runtime
Python runtime	implemented in	C source code
C source code	compiled into	machine instructions
machine instructions	built into	logic gates
logic gates	interpreted as	computation.

Only the last level actually pushes the abacus beads.

Originally, programming languages *didn't* push the abacus beads.

Ada of Lovelace's algorithm for computing Bernoulli numbers was written for a computer that never ended up being invented, but it was a program.

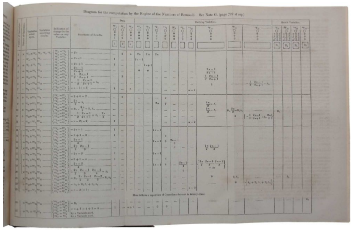
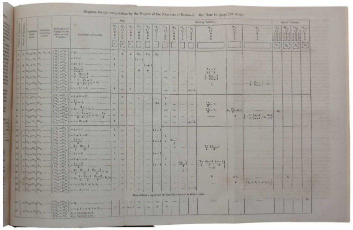


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 et seq.)

[illegible]

Originally, programming languages *didn't* push the abacus beads.

Ada of Lovelace's algorithm for computing Bernoulli numbers was written for a computer that never ended up being invented, but it was a program.



John McCarthy, creator of Lisp: "This EVAL was written and published in the paper and Steve Russel said, 'Look, why don't I program this EVAL?' and I said to him, 'Ho, ho, you're confusing theory with practice—this EVAL is intended for reading, not for computing!' But he went ahead and did it."

Originally, programming languages *didn't* push the abacus beads.

Ada of Lovelace's algorithm for computing Bernoulli numbers was written for a computer that never ended up being invented, but it was a program.

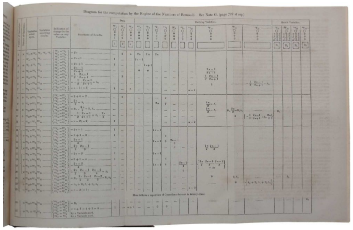


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 et seq.)

[illegible]

John McCarthy, creator of Lisp: "This EVAL was written and published in the paper and Steve Russel said, 'Look, why don't I program this EVAL?' and I said to him, 'Ho, ho, you're confusing theory with practice—this EVAL is intended for reading, not for computing!' But he went ahead and did it."

APL (ancestor of MATLAB, R, and Numpy) was also a notation for describing programs years before it was executable. The book was named *A Programming Language*.

Programmers had to manually translate these notations into instruction codes.

That's why it was called "coding."

Programmers had to manually translate these notations into instruction codes.

That's why it was called "coding."

Von Neumann called assembly language "a waste of a valuable scientific computing instrument—using it for clerical work!"

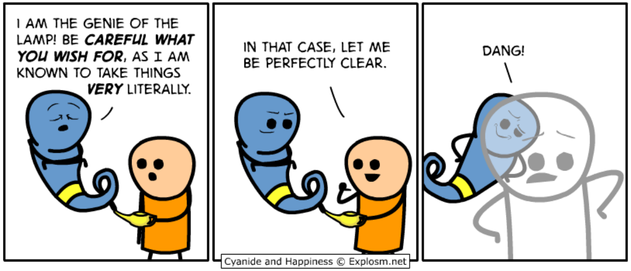
The Software Crisis

Now that our programming languages *do* push abacus beads, software engineering has become an odd discipline: saying something is the same as making it.

The Software Crisis

Now that our programming languages *do* push abacus beads, software engineering has become an odd discipline: saying something is the same as making it.

And yet, we *still* get it wrong.


I AM THE GENIE OF THE LAMP! BE ***CAREFUL WHAT YOU WISH FOR***, AS I AM KNOWN TO TAKE THINGS ***VERY*** LITERALLY.
IN THAT CASE, LET ME BE PERFECTLY CLEAR.
DANG!
Cyanide and Happiness © Explosm.net

We favor high-level languages because they have fewer concepts, hopefully just the ones that are essential for a problem.

We favor high-level languages because they have fewer concepts, hopefully just the ones that are essential for a problem.

But what about speed? Don't we choose languages for speed?

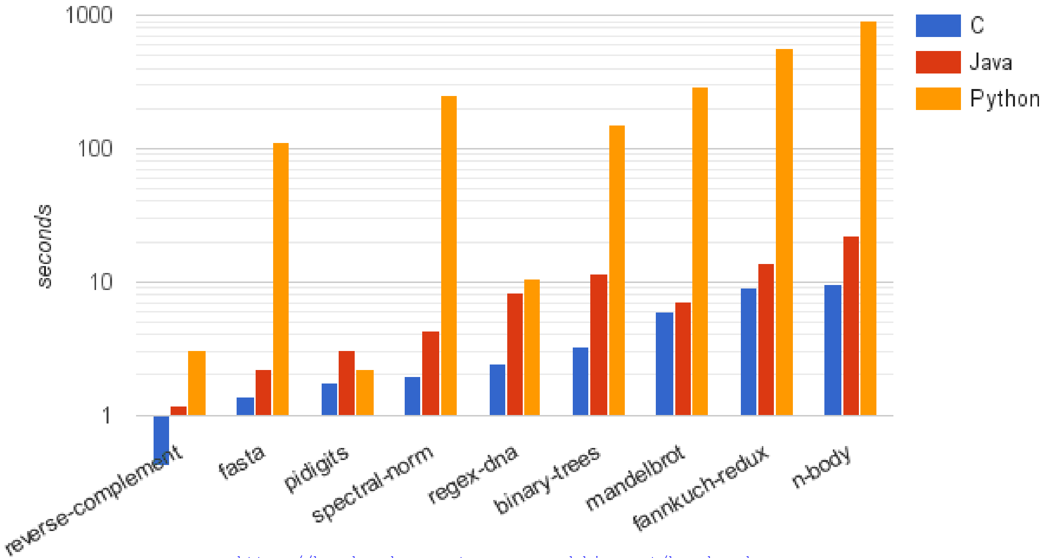
We favor high-level languages because they have fewer concepts, hopefully just the ones that are essential for a problem.

But what about speed? Don't we choose languages for speed?

"There's no such thing as a 'fast' or 'slow' language."

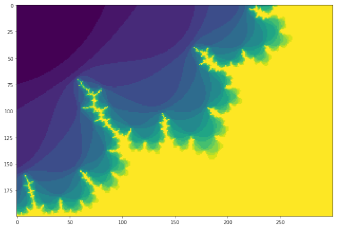
— so sayeth the StackOverflow

Except Python. Python is slow, right?



https://benchmarksgame-team.pages.debian.net/benchmarksgame

But it really isn't the language; it's the implementation.

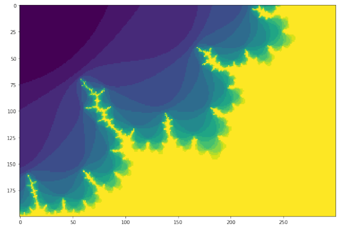


```
import numpy

def run(height, width, maxiterations=20):
    y, x = numpy.ogrid[-1:0:height*1j, -1.5:0:width*1j]
    c = x + y*1j
    fractal = numpy.full(c.shape, maxiterations,
                                        dtype=numpy.int32)
    for h in range(height):
        for w in range(width):                      # for each pixel (h, w)...
            z = c[h, w]
            for i in range(maxiterations):          # iterate at most 20 times
                z = z**2 + c[h, w]                  # applying z → z² + c
                if abs(z) > 2:                      # if it diverges (|z| > 2)
                    fractal[h, w] = i               # color the plane with the iteration number
                    break                           # we're done, no need to keep iterating

    return fractal
```

But it really isn't the language; it's the implementation.



```
import numpy, numba
@numba.jit
def run(height, width, maxiterations=20):
    y, x = numpy.ogrid[-1:0:height*1j, -1.5:0:width*1j]
    c = x + y*1j
    fractal = numpy.full(c.shape, maxiterations,
                                  dtype=numpy.int32)
    for h in range(height):
        for w in range(width):                  # for each pixel (h, w)...
            z = c[h, w]
            for i in range(maxiterations):      # iterate at most 20 times
                z = z**2 + c[h, w]              # applying z → z² + c
                if abs(z) > 2:                  # if it diverges (|z| > 2)
                    fractal[h, w] = i           # color the plane with the iteration number
                    break                       # we're done, no need to keep iterating

    return fractal
```

Now **50× faster**, about equal to C code (-O3).

Here's the catch

The `@numba.jit` decorator translates *a subset of* Python bytecode to machine instructions. You only get a speedup for statically typable, numeric code.

Here's the catch

The `@numba.jit` decorator translates *a subset of* Python bytecode to machine instructions. You only get a speedup for statically typable, numeric code.

Same language (subset), completely different implementation.

Here's the catch

The `@numba.jit` decorator translates *a subset of* Python bytecode to machine instructions. You only get a speedup for statically typable, numeric code.

Same language (subset), completely different implementation.

Pure Python is slower than Numba or C because it has more hurdles in the way: dynamic typing, pointer-chasing, garbage collection, hashtables, string equality. . .

Greg Owen's talk on Spark 2.0

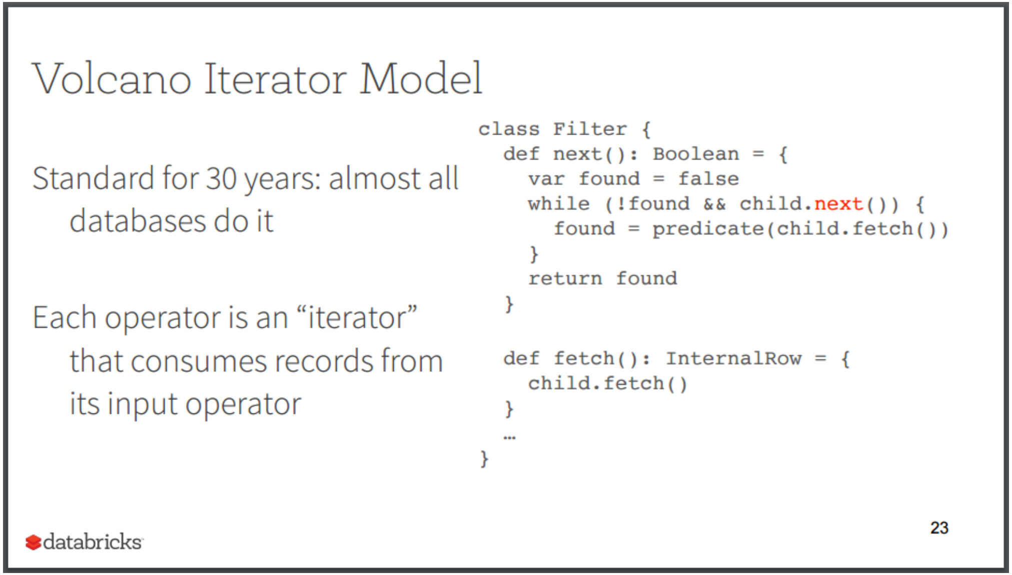
Performance Improvements in Spark 2.0

Greg Owen
2016-05-25

databricks

1

Greg Owen's talk on Spark 2.0



Volcano Iterator Model

Standard for 30 years: almost all databases do it

Each operator is an "iterator" that consumes records from its input operator

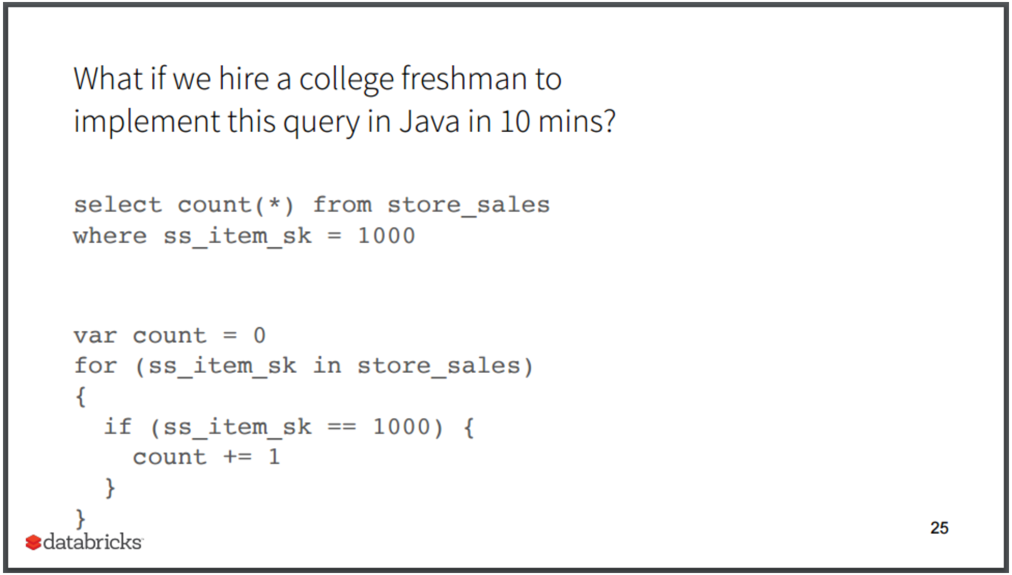
```
class Filter {
  def next(): Boolean = {
    var found = false
    while (!found && child.next()) {
      found = predicate(child.fetch())
    }
    return found
  }

  def fetch(): InternalRow = {
    child.fetch()
  }
  …
}
```

databricks

23

Greg Owen's talk on Spark 2.0



What if we hire a college freshman to implement this query in Java in 10 mins?

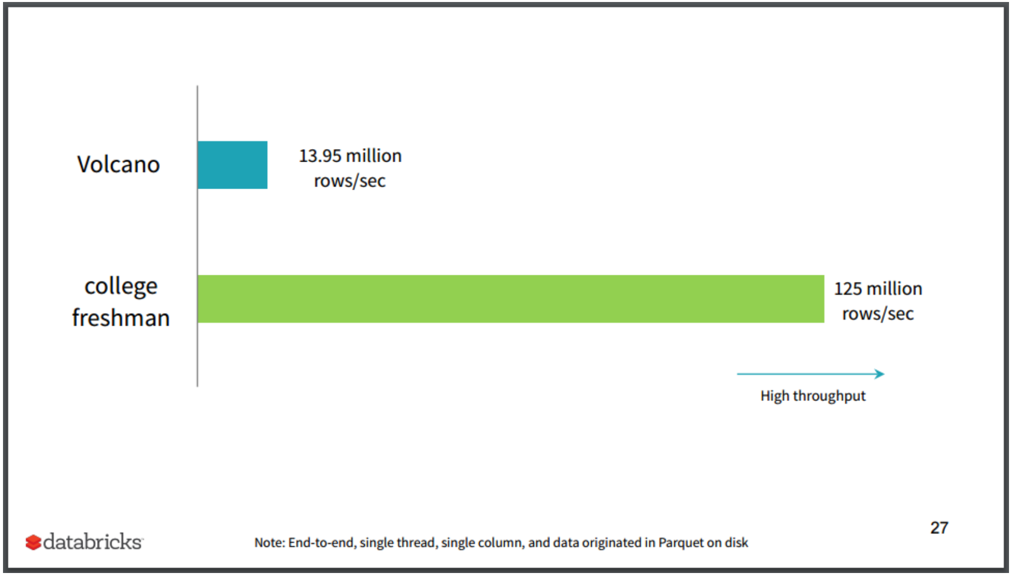
```
select count(*) from store_sales
where ss_item_sk = 1000


var count = 0
for (ss_item_sk in store_sales)
{
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

databricks

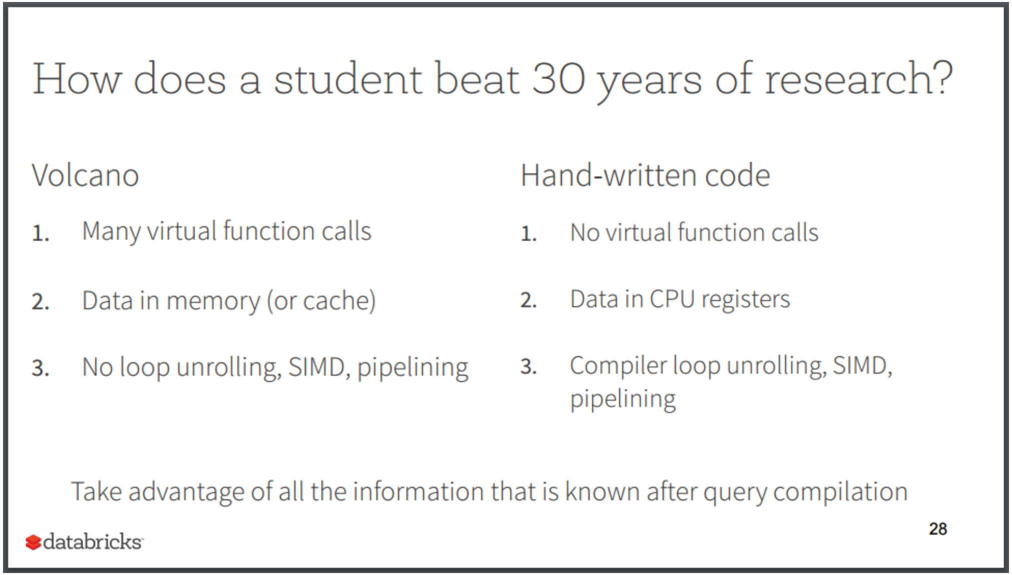
25

Greg Owen's talk on Spark 2.0



Note: End-to-end, single thread, single column, and data originated in Parquet on disk

Greg Owen's talk on Spark 2.0



How does a student beat 30 years of research?

Volcano

1. Many virtual function calls
2. Data in memory (or cache)
3. No loop unrolling, SIMD, pipelining

Hand-written code

1. No virtual function calls
2. Data in CPU registers
3. Compiler loop unrolling, SIMD, pipelining

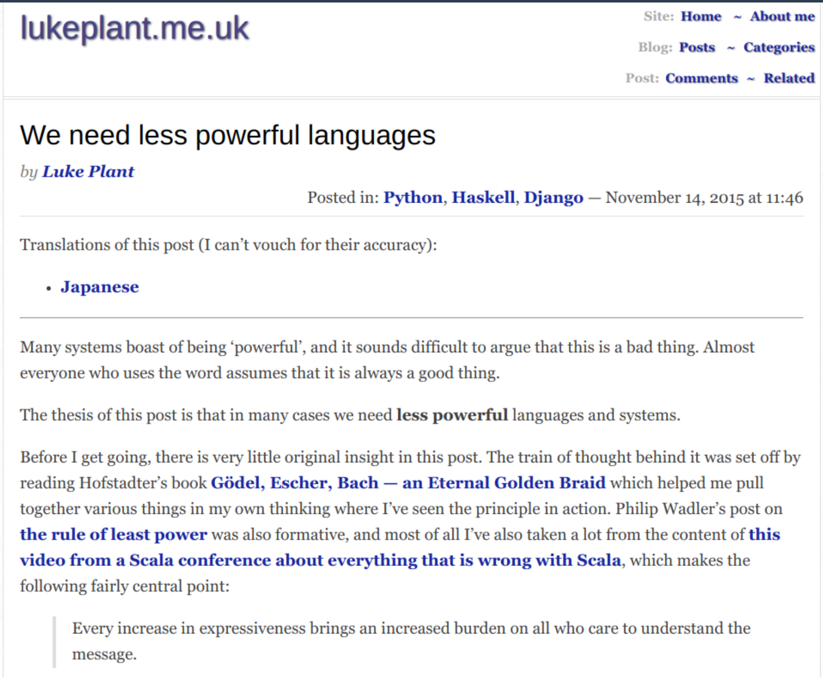
Take advantage of all the information that is known after query compilation

databricks

28

So although it's the implementation, not the language, that's slow,

that implementation can be hampered by the flexibility
that the language promises.



lukeplant.me.uk

Site: Home ~ About me
Blog: Posts ~ Categories
Post: Comments ~ Related

We need less powerful languages

by ***Luke Plant***

Posted in: **Python**, **Haskell**, **Django** — November 14, 2015 at 11:46

Translations of this post (I can't vouch for their accuracy):

- **Japanese**

Many systems boast of being 'powerful', and it sounds difficult to argue that this is a bad thing. Almost everyone who uses the word assumes that it is always a good thing.

The thesis of this post is that in many cases we need **less powerful** languages and systems.

Before I get going, there is very little original insight in this post. The train of thought behind it was set off by reading Hofstadter's book **Gödel, Escher, Bach — an Eternal Golden Braid** which helped me pull together various things in my own thinking where I've seen the principle in action. Philip Wadler's post on **the rule of least power** was also formative, and most of all I've also taken a lot from the content of **this video from a Scala conference about everything that is wrong with Scala**, which makes the following fairly central point:

> Every increase in expressiveness brings an increased burden on all who care to understand the message.

Domain-specific languages:

specialized languages for narrowly defined problems.

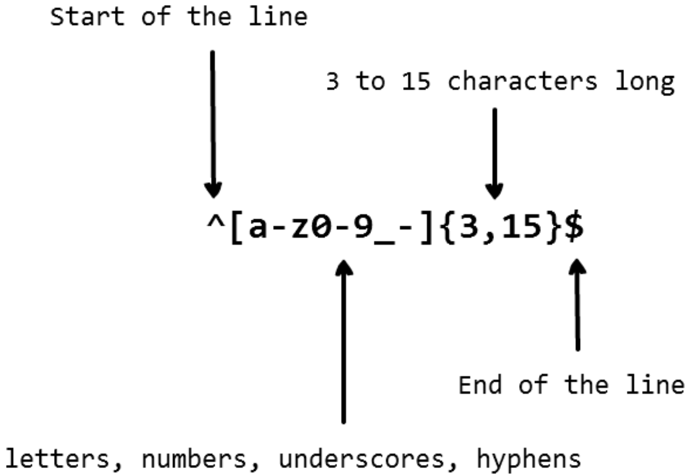
- Main purpose: reduces complexity, the mental clutter that obscures general-purpose languages.
- Secondary purpose: limited flexibility allows for streamlined implementations.

Domain-specific languages that you're probably already using

Any guesses?

Domain-specific languages that you're probably already using

Regular expressions

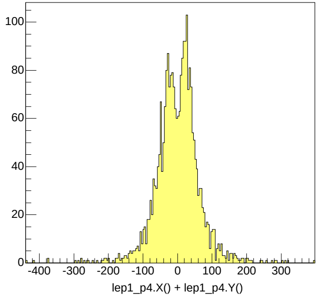


```
^[a-z0-9_-]{3,15}$
```

Domain-specific languages that you're probably already using

TTree::Draw (TTreeFormula)

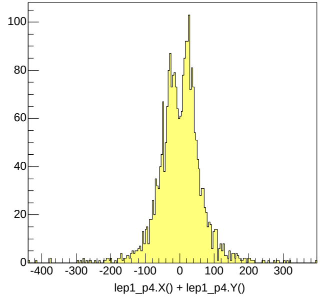
```
ttree->Draw("lep1_p4.X() + lep1_p4.Y()");
```



Domain-specific languages that you're probably already using

TTree::Draw (TTreeFormula)

```
ttree->Draw("lep1_p4.X() + lep1_p4.Y()");
```



Looping and reducing constructs:

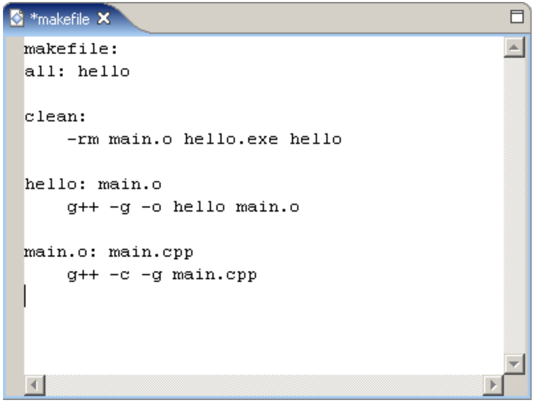
"fMatrix[][fResults[][]]" $\longrightarrow$

```
for (int i0; i0 < 3; i0++) {
   for (int j2; j2 < 5; j2++) {
      for (int j3; j3 < 2; j3++) {
         int i1 = fResults[j2][j3];
         use the value of fMatrix[i0][i1]
   }
}
```

`Length$(·)  Sum$(·)  Min$(·)  Max$(·)  MinIf$(·,·)  MaxIf$(·,·)  Alt$(·,·)`

Domain-specific languages that you're probably already using

Makefiles



```
makefile:
all: hello

clean:
	-rm main.o hello.exe hello

hello: main.o
	g++ -g -o hello main.o

main.o: main.cpp
	g++ -c -g main.cpp
```

Domain-specific languages that you're probably already using

Format strings

printf/scanf: distinct syntax from C/C++, must be quoted

```
printf("Error 0x%04x: %s", id, errors[id]);
```

I/O streams: defined within C/C++ via operator overloading

```
std::cout << "Error 0x" << std::hex << std::setfill('0')
          << std::setw(4) << id << ": " << errors[id] << std::endl;
```

Domain-specific languages that you're probably already using

Format strings

printf/scanf: distinct syntax from C/C++, must be quoted

```
printf("Error 0x%04x: %s", id, errors[id]);
```

I/O streams: defined within C/C++ via operator overloading

```
std::cout << "Error 0x" << std::hex << std::setfill('0')
          << std::setw(4) << id << ": " << errors[id] << std::endl;
```

printf/scanf is "external" and I/O streams is "internal" (embedded)

External versus internal (embedded) domain-specific languages

External: SQL has a distinct syntax from Python; must be quoted in PySpark.

```
import pyspark
pyspark.sql("""
    SELECT CONCAT(first, " ", last) AS fullname, AVG(age)
        FROM my_table WHERE age BETWEEN 18 AND 24
        GROUP BY fullname
""")
```

Internal (embedded): SparkSQL is an equivalent language, defined within Python.

```
import pyspark.sql.functions as F
df = pyspark.read.load("my_table")
(df.withColumn("fullname",
        F.concat(F.col("first"), F.lit(" "), F.col("last")))
   .select("fullname", "age")
   .where(df.age.between(18, 24))
   .groupBy("fullname")
   .agg(F.mean("age")))
```

Objection: a collection of libraries and operator overloads isn't a language!

Objection: a collection of libraries and operator overloads isn't a language!

My answer: programming languages are human modes of expression, implemented using other programming languages, all the way down.

What matters is whether it's a coherent set of concepts, not whether it was implemented by a parser.

Objection: a collection of libraries and operator overloads isn't a language!

My answer: programming languages are human modes of expression, implemented using other programming languages, all the way down.

What matters is whether it's a coherent set of concepts, not whether it was implemented by a parser.

(One might as well argue about the distinction between languages and dialects.)

Perhaps the most widespread domain-specific language in data analysis:

SQL

Perhaps the most widespread domain-specific language in data analysis:

SQL

But we rarely use it in particle physics. Why?

Structure of a collider physics query: C++

"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;
```

Structure of a collider physics query: SQL

"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
```

The problem is that collisions produce a variable number of particles per event: the tables are "jagged."

The problem is that collisions produce a variable number of particles per event: the tables are "jagged."

This *can be* described using SQL's relational concepts:

- separate tables for events and particles
- linked by a common "event number" index.

But each type of particle has to be a separate table and each operation has to be **`INNER JOIN`**ed to maintain events as objects.

The problem is that collisions produce a variable number of particles per event: the tables are "jagged."

This *can be* described using SQL's relational concepts:

- separate tables for events and particles
- linked by a common "event number" index.

But each type of particle has to be a separate table and each operation has to be **`INNER JOIN`**ed to maintain events as objects.

SQL makes particle physics problems *harder*, not *easier*, which defeats the point.

It seems like there's an opportunity here

Would a domain specific language for particle physics

- make analysis code easier to read?
- make mistakes more evident?
- make it easier to synchronize analyses from different groups/experiments?
- make it easier to preserve them in executable/recastable form?
- highlight physics concepts, like control regions, systematic variations, event weights, combinatorics with symmetries?
- hide irrelevant concepts like managing files, memory, load balancing, and other performance tweaks?

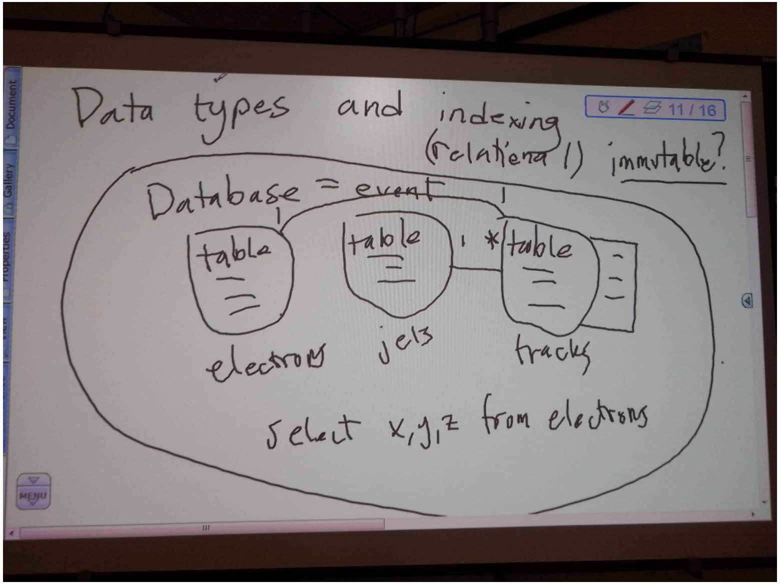
It seems like there's an opportunity here

Would a domain specific language for particle physics

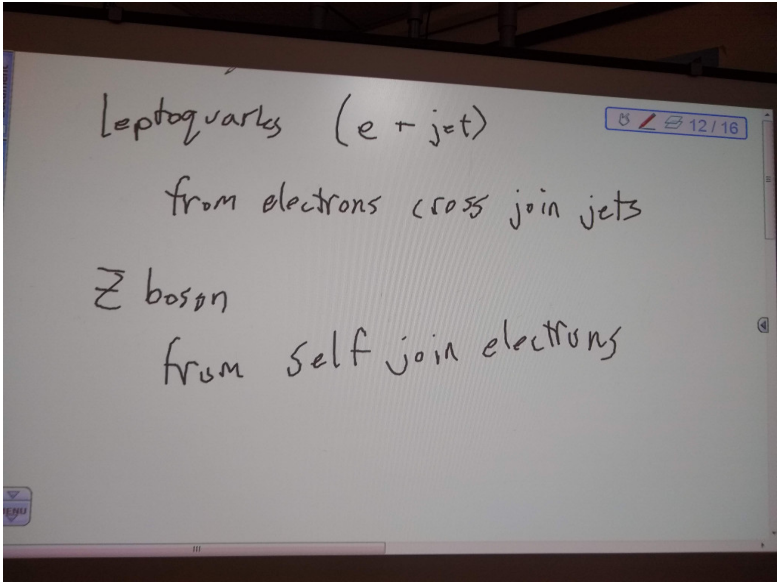
- make analysis code easier to read?
- make mistakes more evident?
- make it easier to synchronize analyses from different groups/experiments?
- make it easier to preserve them in executable/recastable form?
- highlight physics concepts, like control regions, systematic variations, event weights, combinatorics with symmetries?
- hide irrelevant concepts like managing files, memory, load balancing, and other performance tweaks?

That was the subject of the Analysis Description Language Workshop.

In fact, about that SQL…



In fact, about that SQL. . .



Leptoquarks (e + jet)

from electrons cross join jets

Z boson

from self join electrons

Why hasn't this been done before?

(Why hasn't it succeeded before?)

I think the answer is cultural, so I'll take a historical perspective. . .

I think the answer is cultural, so I'll take a historical perspective. . .

Starting in 1880.

The U.S. Census's problem

The U.S. does a census every 10 years. The 1880 census took 8 years to process.

$\longrightarrow$ Big data problem!

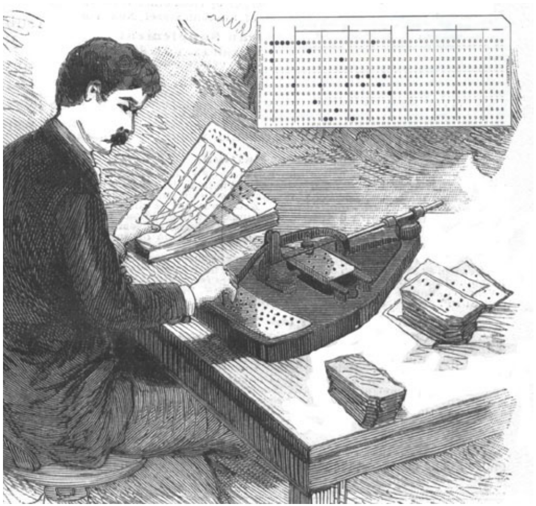
The U.S. Census's problem

The U.S. does a census every 10 years. The 1880 census took 8 years to process.

$\longrightarrow$ Big data problem!

Held a competition for a new method; winner was 10$\times$ faster than the rest:

Census records on punch cards, which filtered electrical contacts



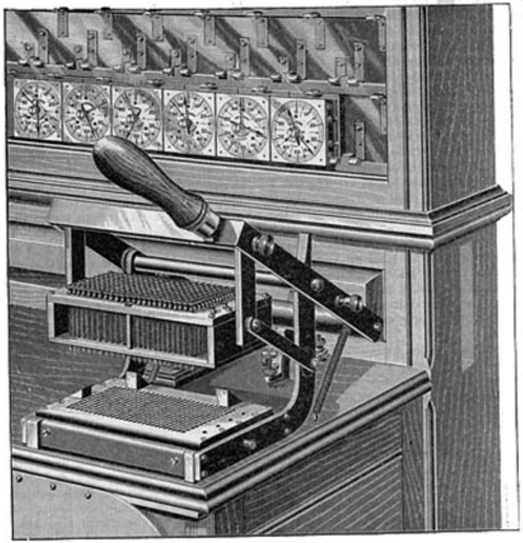


Fig. 8.—Circuit-Closing Press.

Hollerith's Electric Sorting and Tabulating Machine

Wired to a machine that opens a door for each matching pattern

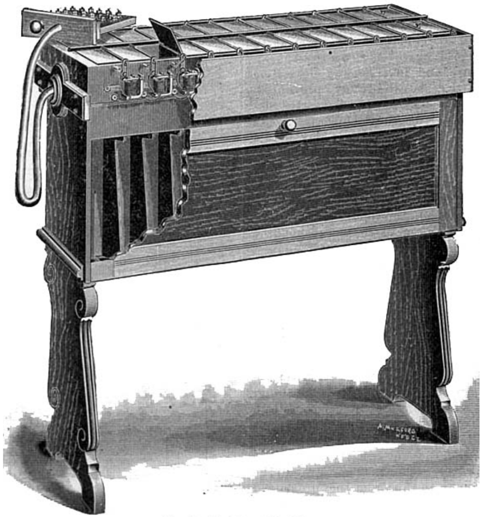


Fig. 3.—Sorting Machine.

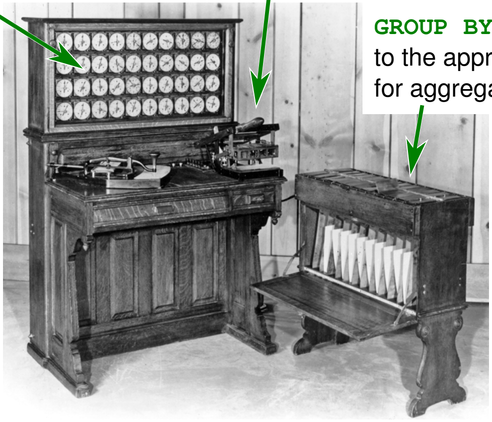
Hollerith's Electric Sorting and Tabulating Machine.

It was an SQL machine: 3 basic clauses of most SQL queries

SELECT: pre-programmed (wired up) counters

WHERE: pins pass through punch card and template

GROUP BY: door opens to the appropriate bin for aggregation



SELECT name **WHERE** literate **GROUP BY** marital_status

Herman Hollerith (inventor) incorporated the Tabulating Machine Company, which after a series of mergers became International Business Machines (IBM) in 1924.

Herman Hollerith (inventor) incorporated the Tabulating Machine Company, which after a series of mergers became International Business Machines (IBM) in 1924.

The computation represented by this machine is not universal (Turing complete), but has many applications.

Herman Hollerith (inventor) incorporated the Tabulating Machine Company, which after a series of mergers became International Business Machines (IBM) in 1924.

The computation represented by this machine is not universal (Turing complete), but has many applications.

Most recently as “map-reduce.”

Google's problem

In the early 2000's, Google was struggling to keep up with the growing web (index 5 months out of date, routine hardware failures, scale sensitive to bit flips).

Google's problem

In the early 2000's, Google was struggling to keep up with the growing web (index 5 months out of date, routine hardware failures, scale sensitive to bit flips).

At that time, each programmer had to divide tasks, distribute, and combine results manually and account for failures manually.

Google's problem

In the early 2000's, Google was struggling to keep up with the growing web (index 5 months out of date, routine hardware failures, scale sensitive to bit flips).

At that time, each programmer had to divide tasks, distribute, and combine results manually and account for failures manually.

2003: MapReduce created to abstract task management from analysis logic.

Google’s problem

In the early 2000’s, Google was struggling to keep up with the growing web (index 5 months out of date, routine hardware failures, scale sensitive to bit flips).

At that time, each programmer had to divide tasks, distribute, and combine results manually and account for failures manually.

2003: MapReduce created to abstract task management from analysis logic.

MapReduce is distributed $\underbrace{\texttt{SELECT-WHERE}}_{\text{“map”}}$-$\underbrace{\texttt{GROUP BY}}_{\text{“reduce”}}$.

Google's problem

In the early 2000's, Google was struggling to keep up with the growing web (index 5 months out of date, routine hardware failures, scale sensitive to bit flips).

At that time, each programmer had to divide tasks, distribute, and combine results manually and account for failures manually.

2003: MapReduce created to abstract task management from analysis logic.

MapReduce is distributed $\underbrace{\texttt{SELECT-WHERE}}_{\text{``map''}}$ - $\underbrace{\texttt{GROUP BY}}_{\text{``reduce''}}$.

2004: published as a paper by Jeffrey Dean and Sanjay Ghemawat.
2006: reimplemented as open-source software: Apache Hadoop.

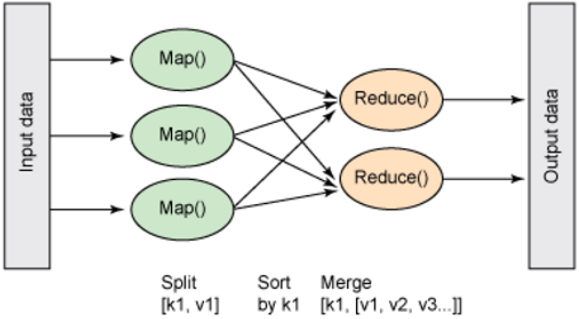
Problems like "index all webpages" plug into this framework.

SELECT-WHERE: filter and transform each input to a ⟨key, value⟩ pair.

```
def map(webpage):
    for word in webpage.split():
        if not stopword(word):
            yield (word, webpage)
```

GROUP BY: collect and transform all values with a given key.

```
def reduce(word, webpages):
    index[word] = set()
    for webpage in webpages:
        index[word].add(webpage)
```



That's how statisticians encountered computing.

Physics encountered computing differently.

Physicists got into computers when they became general-purpose

1944: John Mauchly (physicist) and J. Presper Eckert (electrical engineer) designed ENIAC to replace mechanical computers for ballistics.

Physicists got into computers when they became general-purpose

1944: John Mauchly (physicist) and J. Presper Eckert (electrical engineer) designed ENIAC to replace mechanical computers for ballistics.

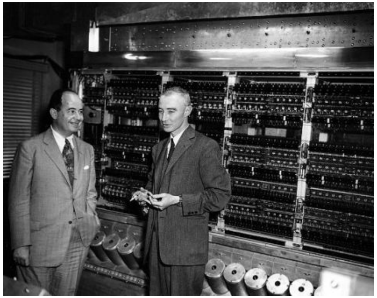
ENIAC was one of the first computers driven by machine code instructions, stored as a program in memory.

Physicists got into computers when they became general-purpose

1944: John Mauchly (physicist) and J. Presper Eckert (electrical engineer) designed ENIAC to replace mechanical computers for ballistics.

ENIAC was one of the first computers driven by machine code instructions, stored as a program in memory.

1945: John von Neumann learned of their work and suggested using it for nuclear simulations (H-bomb).



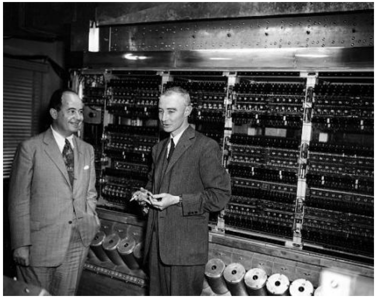
Physicists got into computers when they became general-purpose

1944: John Mauchly (physicist) and J. Presper Eckert (electrical engineer) designed ENIAC to replace mechanical computers for ballistics.

ENIAC was one of the first computers driven by machine code instructions, stored as a program in memory.

1945: John von Neumann learned of their work and suggested using it for nuclear simulations (H-bomb).

His internal memo describing ENIAC's stored programs was leaked; now known as "Von Neumann architecture."



Physicists got into computers when they became general-purpose

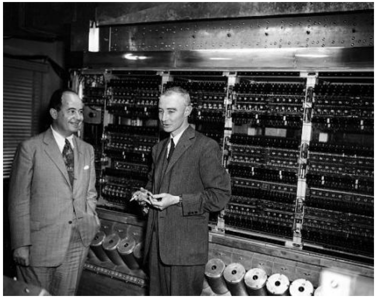
1944: John Mauchly (physicist) and J. Presper Eckert (electrical engineer) designed ENIAC to replace mechanical computers for ballistics.

ENIAC was one of the first computers driven by machine code instructions, stored as a program in memory.

1945: John von Neumann learned of their work and suggested using it for nuclear simulations (H-bomb).

His internal memo describing ENIAC's stored programs was leaked; now known as "Von Neumann architecture."

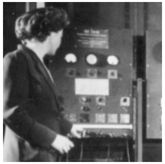
Los Alamos group led by Nicholas Metropolis, developed Monte Carlo techniques for physics problems.

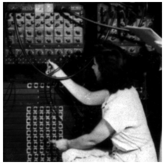


The actual programming was performed by these six women

Kathleen McNulty

Frances Bilas


Betty Jean Jennings


Ruth Lichterman

Elizabeth Snyder

Marlyn Wescoff

Eckert-Mauchly Computer Corporation → Remington Rand

Mauchly and Eckert "went into industry" selling computers; the first one (UNIVAC) to the U.S. Census.

Eckert-Mauchly Computer Corporation → Remington Rand

Mauchly and Eckert "went into industry" selling computers; the first one (UNIVAC) to the U.S. Census.

1950: Short Code, the first executable high-level language: a transliterated interpreter of mathematical formulas.

```
math: X3 =  (  X1 +  Y1 )  /  X1 * Y1
code: X3 03 09 X1 07 Y1 02 04 X1   Y1
```

50× slower than machine code because it was interpreted.

Eckert-Mauchly Computer Corporation → Remington Rand

Mauchly and Eckert “went into industry” selling computers; the first one (UNIVAC) to the U.S. Census.

1950: Short Code, the first executable high-level language: a transliterated interpreter of mathematical formulas.

```
math: X3 =  (  X1 +  Y1 )  /  X1 * Y1
code: X3 03 09 X1 07 Y1 02 04 X1   Y1
```

50× slower than machine code because it was interpreted.

1952–1959: At Remington Rand, Grace Hopper developed a series of *compiled* languages, ultimately COBOL.

Eckert-Mauchly Computer Corporation → Remington Rand

Mauchly and Eckert "went into industry" selling computers; the first one (UNIVAC) to the U.S. Census.

1950: Short Code, the first executable high-level language: a transliterated interpreter of mathematical formulas.

```
math: X3 =  (  X1 +  Y1 )  /  X1 * Y1
code: X3 03 09 X1 07 Y1 02 04 X1   Y1
```

50× slower than machine code because it was interpreted.

1952–1959: At Remington Rand, Grace Hopper developed a series of *compiled* languages, ultimately COBOL.

Meanwhile, IBM developed FORTRAN: 1954–1957.

Physicists drove programming language development in the 1940's and 1950's but stuck with FORTRAN until the 21st century.

Physicists drove programming language development in the 1940's and 1950's but stuck with FORTRAN until the 21$^{\text{st}}$ century.

In fact, FORTRAN (pre-Fortran 90) wasn't even a good fit to *data analysis* problems. It didn't handle jagged data well, much like SQL.

Physicists drove programming language development in the 1940's and 1950's but stuck with FORTRAN until the 21st century.

In fact, FORTRAN (pre-Fortran 90) wasn't even a good fit to *data analysis* problems. It didn't handle jagged data well, much like SQL.

This gap was filled with a library: ZEBRA provided a graph of structures and dynamic memory management, even though these were features of Ada, C, Pascal, and PL/I.

Physicists drove programming language development in the 1940's and 1950's but stuck with FORTRAN until the 21st century.

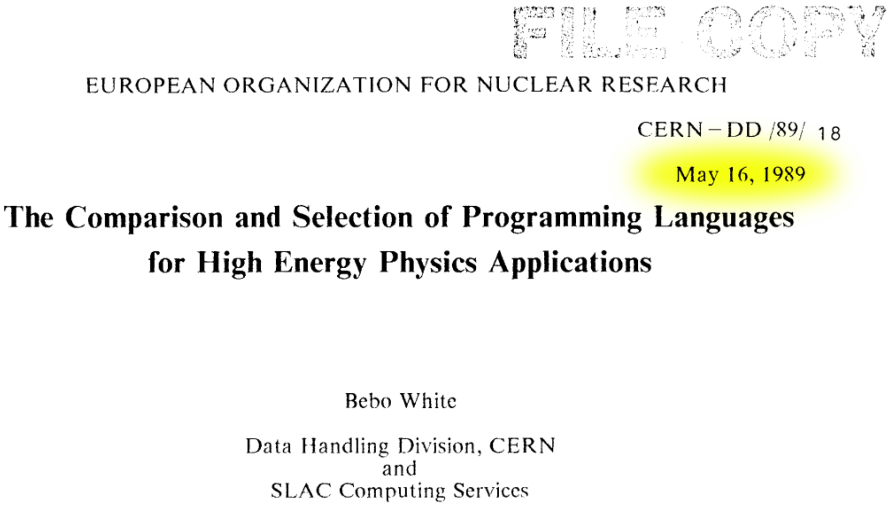
In fact, FORTRAN (pre-Fortran 90) wasn't even a good fit to *data analysis* problems. It didn't handle jagged data well, much like SQL.

This gap was filled with a library: ZEBRA provided a graph of structures and dynamic memory management, even though these were features of Ada, C, Pascal, and PL/I.

"The whole concept of ZEBRA is a manifestation of one of FORTRAN 77's needs."

— Bebo White in 1989

Ironically, a very similar talk was given almost 20 years ago today



FILE COPY

EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

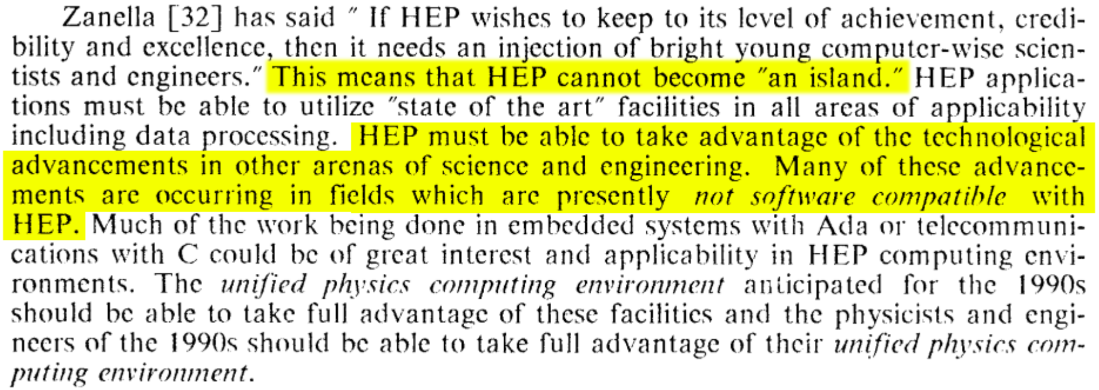
CERN – DD /89/ 18

May 16, 1989

The Comparison and Selection of Programming Languages for High Energy Physics Applications

Bebo White

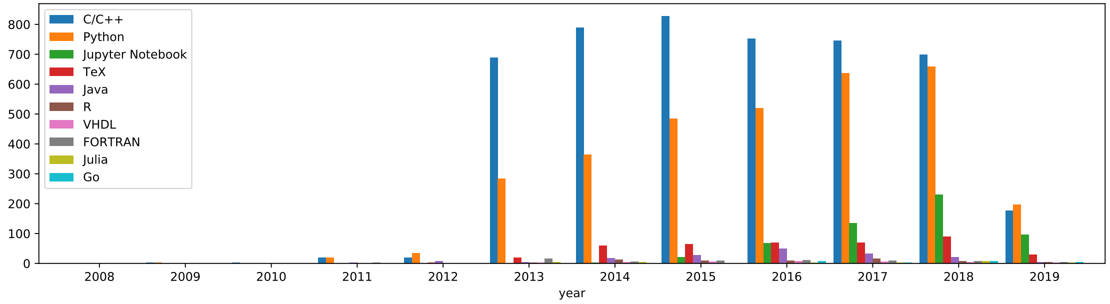
Data Handling Division, CERN
and
SLAC Computing Services



Zanella [32] has said " If HEP wishes to keep to its level of achievement, credibility and excellence, then it needs an injection of bright young computer-wise scientists and engineers." This means that HEP cannot become "an island." HEP applications must be able to utilize "state of the art" facilities in all areas of applicability including data processing. HEP must be able to take advantage of the technological advancements in other arenas of science and engineering. Many of these advancements are occurring in fields which are presently *not software compatible* with HEP. Much of the work being done in embedded systems with Ada or telecommunications with C could be of great interest and applicability in HEP computing environments. The *unified physics computing environment* anticipated for the 1990s should be able to take full advantage of these facilities and the physicists and engineers of the 1990s should be able to take full advantage of their *unified physics computing environment*.

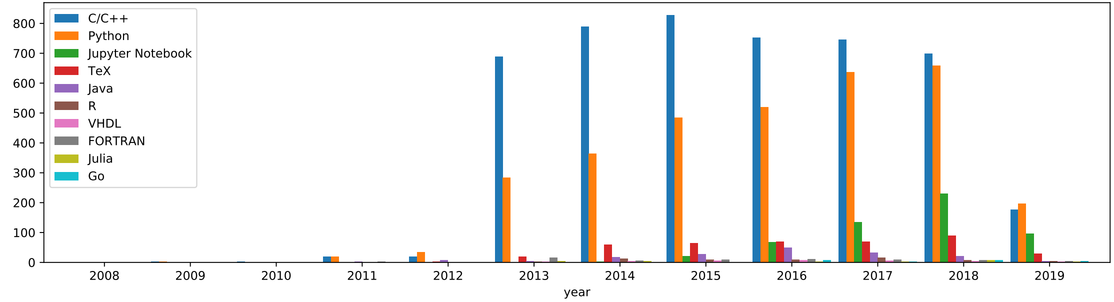
Are we halfway through the second major language shift?

Languages of $\underbrace{\text{non-fork repos}}_{\text{their own work}}$ for $\underbrace{\text{GitHub users who also fork }\texttt{cms-sw/cmssw}}_{\text{physicists, specifically CMS}}$



Are we halfway through the second major language shift?

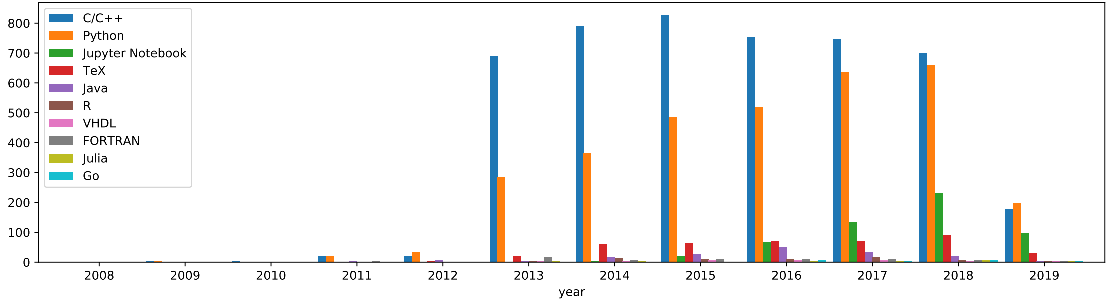
$\text{Languages of } \underbrace{\text{non-fork repos}}_{\text{their own work}} \text{ for } \underbrace{\text{GitHub users who also fork } \texttt{cms-sw/cmssw}}_{\text{physicists, specifically CMS}}$



The shift from Fortran to C++ was a decision made by collaboration leaders.

Are we halfway through the second major language shift?

$\underbrace{\text{Languages of non-fork repos}}_{\text{their own work}}$ for $\underbrace{\text{GitHub users who also fork } \texttt{cms-sw/cmssw}}_{\text{physicists, specifically CMS}}$



The shift from Fortran to C++ was a decision made by collaboration leaders.

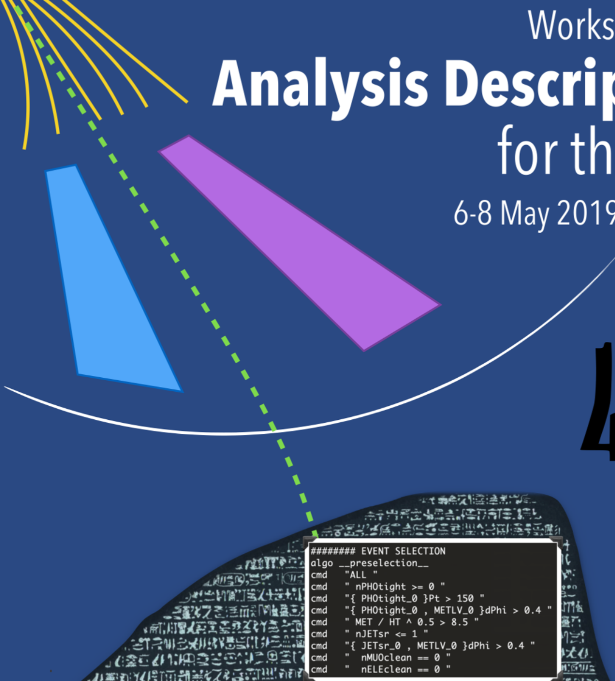
What we see here are individuals choosing a language for their own work.

Workshop on

Analysis Description Languages for the LHC

6-8 May 2019, Fermilab LPC

https://indico.cern.ch/event/769263/



```
######## EVENT SELECTION
algo __preselection__
cmd    "ALL "
cmd    " nPHOtight >= 0 "
cmd    "{ PHOtight_0 }Pt > 150 "
cmd    "{ PHOtight_0 , METLV_0 }dPhi > 0.4 "
cmd    " MET / HT ^ 0.5 > 8.5 "
cmd    " nJETsr <= 1 "
cmd    "{ JETsr_0 , METLV_0 }dPhi > 0.4 "
cmd    "  nMUOclean == 0 "
cmd    "  nELEclean == 0 "
```

An analysis description language (ADL) is a human readable declarative language that unambiguously describes the contents of an analysis in a standard way, independent of any computing framework.

Adopting ADLs would bring numerous benefits for the LHC experimental and phenomenological communities, ranging from analysis preservation beyond the lifetimes of experiments or analysis software to facilitating the abstraction, design, visualization, validation, combination, reproduction, interpretation and overall communication of the

Informal summary of the workshop at tomorrow's LPC Physics Forum at 1:30pm.