# Pattern matching for particle physics

Jim Pivarski

Princeton University – IRIS-HEP

May 22, 2019

Quite a few groups have been thinking about physics event processing languages, explicitly or implicitly.

- LHADA/ADL: Sezen Sekmen, Harry Prosper, Philippe Gras
- CutLang: Gokhan Unel
- IRIS-HEP Analysis Systems: Gordon Watts, Mason Proffitt, Emma Torro
- FAST-Carpenter (YAML): Benjamin Krikler
- NAIL: Andrea Rizzi
- RDataFrame: Enrico Guiraud, Danilo Piparo, and the ROOT Team
- AEACuS & RHADAManTHUS: Joel Walker (phenomenology)
- Femtocode: me, though not for several years. . .

see Analysis Description Languages Workshop (May 6–8)

For a physics domain-specific language (DSL) to be useful, it has to solve physics problems in a way that is clearly simpler than the general-purpose language.

For a physics domain-specific language (DSL) to be useful, it has to solve physics problems in a way that is clearly simpler than the general-purpose language.

This is different from just being “high-level,” solving any problem with less baggage than a low-level language.

## What is difficult in physics analyses?

- Distributing work to remote processors. General enough for a whole industry. (Spark, Parsl, Dask, Condor, Slurm. . . )
- Interactive analysis, publication-quality plotting. Also very general. (Jupyter)
- Workflow management: chaining analysis tasks. General. (CWL, Makefile)

## What is difficult in physics analyses?

- Distributing work to remote processors. General enough for a whole industry. (Spark, Parsl, Dask, Condor, Slurm. . . )
- Interactive analysis, publication-quality plotting. Also very general. (Jupyter)
- Workflow management: chaining analysis tasks. General. (CWL, Makefile)
- Booking, filling, managing histograms. Physics specific! (ROOT)
- Complex, multi-component distribution fitting. Physics specific! (RooFit)
- Complex multi-histogram fitting. Physics specific! (Combine, HistFactory)

## What is difficult in physics analyses?

- Distributing work to remote processors. General enough for a whole industry. (Spark, Parsl, Dask, Condor, Slurm. . . )
- Interactive analysis, publication-quality plotting. Also very general. (Jupyter)
- Workflow management: chaining analysis tasks. General. (CWL, Makefile)
- Booking, filling, managing histograms. Physics specific! (ROOT)
- Complex, multi-component distribution fitting. Physics specific! (RooFit)
- Complex multi-histogram fitting. Physics specific! (Combine, HistFactory)
- Signal/control regions, systematic variations. Physics specific! (slide 2)
- Particle combinatorics: complex decay chains. Physics specific! (???)

## Particle combinatorics

Even a simple Z-peak requires combinatorics: don't double-count the muons!

```
std::vector<Particle> Z;
for (int i = 0;  i < muons.size();  i++)
    for (int j = i + 1;  j < muons.size();  j++)  // not all j
        Z.push_back(muons[i] + muons[j]);
```

This is the reason we can't "just use SQL/Numpy/MATLAB."

## Particle combinatorics

Even a simple Z-peak requires combinatorics: don't double-count the muons!

```
std::vector<Particle> Z;
for (int i = 0;  i < muons.size();  i++)
    for (int j = i + 1;  j < muons.size();  j++)  // not all j
        Z.push_back(muons[i] + muons[j]);
```

This is the reason we can't "just use SQL/Numpy/MATLAB."

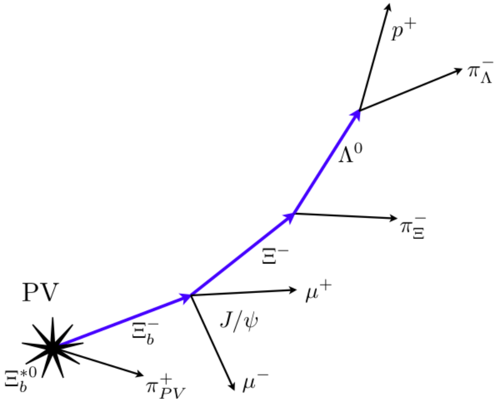
Awkward-Array addresses this with methods to generate per-event combinations.

- `A.cross(B)`: cross-join (Cartesian product) of `A` and `B`.
- `A.pairs()`: inner-join of `A` with itself, excluding duplicates.
- `A.distincts()`: inner-join of `A` with itself, also excluding $(A_i, A_i)$.
- `A.choose(n)`: like `distincts`, but for tuples of size $n \leq 5$.

(Cleverly implemented by Jaydeep Nandi and Nick Smith without internal for loops!)

## But it's still not enough

Building a complex decay chain using only these combinatorial primitives would be a struggle.



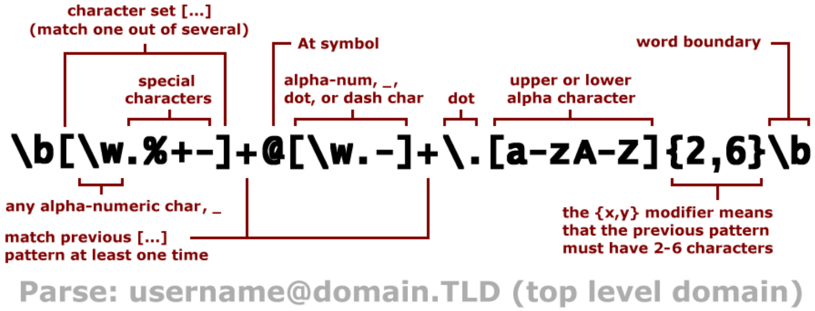
## Pattern matching

We want to fit a disconnected set of particles to a structured decay chain, which reminds me of pattern matching.

## Pattern matching

We want to fit a disconnected set of particles to a structured decay chain, which reminds me of pattern matching.

Pattern matching is an uncommon programming language feature, like regular expressions, but for data structures: you make a model of what you want.


character set [...] (match one out of several)
special characters
At symbol
alpha-num, _, dot, or dash char
dot
upper or lower alpha character
word boundary
\b[\w.%+-]+@[\w.-]+\.[a-zA-Z]{2,6}\b
any alpha-numeric char, _
match previous [...] pattern at least one time
the {x,y} modifier means that the previous pattern must have 2-6 characters
Parse: username@domain.TLD (top level domain)

## Examples of pattern matching

**Python:** (limited; can only unpack iterables)

```
def tree():
    return [[[1, 2], [3, 4]], [5, 6]]
(((a, b), c), _) = tree()    # a → 1, b → 2, c → [3, 4]
```

## Examples of pattern matching

**Python:** (limited; can only unpack iterables)

```
def tree():
    return [[[1, 2], [3, 4]], [5, 6]]
(((a, b), c), _) = tree()     # a → 1, b → 2, c → [3, 4]
```

**Haskell:**

```
pz :: (Particle particle) => particle -> Float
pz (Neutral pt eta _)    = pt * sinh(eta)
pz (Charged pt eta _ q)  = pt * sinh(eta)
```

## Examples of pattern matching

**Python:** (limited; can only unpack iterables)

```python
def tree():
    return [[[1, 2], [3, 4]], [5, 6]]
(((a, b), c), _) = tree()     # a → 1, b → 2, c → [3, 4]
```

**Haskell:**

```haskell
pz :: (Particle particle) => particle -> Float
pz (Neutral pt eta _)    = pt * sinh(eta)
pz (Charged pt eta _ q)  = pt * sinh(eta)
```

**Scala:**

```scala
def pz(particle: Particle) = match particle {
    case Neutral(pt, eta, _)     => pt * sinh(eta)
    case Charged(pt, eta, _, q)  => pt * sinh(eta)
}
```

# Toy language with particle pattern matching

```
start:       (NEWLINE | ";")* (statement (NEWLINE | ";")+)* statement (NEWLINE | ";")*

statement:  assignment | funcassign
funcassign: CNAME "(" [CNAME ("," CNAME)*] ")" "=" block
assignment: CNAME "=" expression

patassign:  CNAME "=" expression
          | CNAME "~" expression   -> symmetric
          | CNAME "~~" expression  -> all_symmetric
          | CNAME "!~" expression  -> asymmetric
          | CNAME "!~~" expression -> all_asymmetric
pattern:     (NEWLINE | ";")* (patassign (NEWLINE | ";")+)* patassign (NEWLINE | ";")*
function:   paramlist "=>" block

      ... skip the boring part (standard expression grammar, same as Python) ...

atom:       CNAME -> symbol | INT -> int | FLOAT -> float
          | "(" block ")" -> pass
          | "" pattern "" -> pass
          | "join" "" pattern "" -> join
```

See https://github.com/diana-hep/rejig/blob/master/pattern-match/define-and-run.py

## Example: Higgs $\to$ ZZ $\to$ $4\ell$

```
higgs(flavor1, flavor2) =
    join {
        z1 ~ {
            lep1 ~ flavor1
            lep2 ~ flavor1
            mass = (lep1.p4 + lep2.p4).mass
        }
        z2 ~ {
            lep1 ~ flavor2
            lep2 ~ flavor2
            mass = (lep1.p4 + lep2.p4).mass
        }
    }.filter(h => h.z1.lep1.charge != h.z1.lep2.charge and
                  h.z2.lep1.charge != h.z2.lep2.charge)
     .sort(h => (h.z1.mass - 91)**2 + (h.z2.mass - 91)**2)

higgs4e    = higgs(electrons, electrons)
higgs4mu   = higgs(muons, muons)
higgs2e2mu = higgs(electrons, muons)
```

## Example: Higgs $\to$ ZZ $\to 4\ell$

```
higgs(flavor1, flavor2) =           // define a join pattern in a function
    join {
        z1 ~ {                      // Z boson subpattern
            lep1 ~ flavor1          // lep1, lep2 from flavor1 collection
            lep2 ~ flavor1          // leptons are NOT double-counted
            mass = (lep1.p4 + lep2.p4).mass
        }
        z2 ~ {                      // another Z boson
            lep1 ~ flavor2          // lep1, lep2 from flavor2, which
            lep2 ~ flavor2          // might be the same as flavor1
            mass = (lep1.p4 + lep2.p4).mass
        }                           // filter and sort with functionals
    }.filter(h => h.z1.lep1.charge != h.z1.lep2.charge and
                  h.z2.lep1.charge != h.z2.lep2.charge)
     .sort(h => (h.z1.mass - 91)**2 + (h.z2.mass - 91)**2)

higgs4e    = higgs(electrons, electrons)    // use the function
higgs4mu   = higgs(muons, muons)            // to match patterns
higgs2e2mu = higgs(electrons, muons)
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {    // cross-join of the two (distinct) input collections
  a ~ x
  b ~ y
}
```

**Output:**

```
[(1.1, "one"),   (2.2, "one"),   (3.3, "one"),   (4.4, "one"),
 (1.1, "two"),   (2.2, "two"),   (3.3, "two"),   (4.4, "two"),
 (1.1, "three"), (2.2, "three"), (3.3, "three"), (4.4, "three"),
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {     // inner join of one collection requiring uniqueness
  a ~ x
  b ~ x
}
```

**Output:**

```
[           (1.1, 2.2), (1.1, 3.3), (1.1, 4.4), (1.1, 5.5),
                        (2.2, 3.3), (2.2, 4.4), (2.2, 5.5),
                                    (3.3, 4.4), (3.3, 5.5),
                                                (4.4, 5.5)]
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {    // two tildes doesn't require uniqueness
  a ~~ x
  b ~~ x
}
```

**Output:**

```
[(1.1, 1.1), (1.1, 2.2), (1.1, 3.3), (1.1, 4.4), (1.1, 5.5),
             (2.2, 2.2), (2.2, 3.3), (2.2, 4.4), (2.2, 5.5),
                         (3.3, 3.3), (3.3, 4.4), (3.3, 5.5),
                                     (4.4, 4.4), (4.4, 5.5),
                                                 (5.5, 5.5)]
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {    // exclamation point means don't constrain symmetry
  a !~ x
  b !~ x
}
```

**Output:**

```
[           (1.1, 2.2), (1.1, 3.3), (1.1, 4.4), (1.1, 5.5),
 (2.2, 1.1),            (2.2, 3.3), (2.2, 4.4), (2.2, 5.5),
 (3.3, 1.1), (3.3, 2.2),            (3.3, 4.4), (3.3, 5.5),
 (4.4, 1.1), (4.4, 2.2), (4.4, 3.3),            (4.4, 5.5),
 (5.5, 1.1), (5.5, 2.2), (5.5, 3.3), (5.5, 4.4)            ]
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {    // apply both operators for full generality
  a !~~ x
  b !~~ x
}
```

**Output:**

```
[(1.1, 1.1), (1.1, 2.2), (1.1, 3.3), (1.1, 4.4), (1.1, 5.5),
 (2.2, 1.1), (2.2, 2.2), (2.2, 3.3), (2.2, 4.4), (2.2, 5.5),
 (3.3, 1.1), (3.3, 2.2), (3.3, 3.3), (3.3, 4.4), (3.3, 5.5),
 (4.4, 1.1), (4.4, 2.2), (4.4, 3.3), (4.4, 4.4), (4.4, 5.5),
 (5.5, 1.1), (5.5, 2.2), (5.5, 3.3), (5.5, 4.4), (5.5, 5.5)]
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {      // separate nested structures break the symmetry
  a ~ x     // but uniqueness is global across the whole tree
  b ~ { c ~ x }
}
```

**Output:**

```
[            (1.1,(2.2)),(1.1,(3.3)),(1.1,(4.4)),(1.1,(5.5)),
 (2.2,(1.1)),            (2.2,(3.3)),(2.2,(4.4)),(2.2,(5.5)),
 (3.3,(1.1)),(3.3,(2.2)),            (3.3,(4.4)),(3.3,(5.5)),
 (4.4,(1.1)),(4.4,(2.2)),(4.4,(3.3)),            (4.4,(5.5)),
 (5.5,(1.1)),(5.5,(2.2)),(5.5,(3.3)),(5.5,(4.4))            ]
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {     // computing new fields doesn't change combinatorics
  a ~ x
  b ~ x
  c = a * b
}
```

**Output:**

```
[(1.1,2.2,2.42), (1.1,3.3,3.63), (1.1,4.4,4.84), (1.1,5.5,6.05),
                 (2.2,3.3,7.26), (2.2,4.4,9.68), (2.2,5.5,12.1),
                                 (3.3,4.4,14.5), (3.3,5.5,18.2),
                                                 (4.4,5.5,24.2)]
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {    // not all fields need to be matches
  a ~ x
  b = y
}
```

**Output:**

```
[(1.1, ["one", "two", "three"]),
 (2.2, ["one", "two", "three"]),
 (3.3, ["one", "two", "three"]),
 (4.4, ["one", "two", "three"]),
 (5.5, ["one", "two", "three"])]
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {     // in fact, none of them really need to be
  a = x
  b = y
}
```

**Output:**

```
[([1.1, 2.2, 3.3, 4.4, 5.5], ["one", "two", "three"])]
```

## Simpler, more pedagogical cases

**Input:**

```
x = [1.1, 2.2, 3.3, 4.4, 5.5]
y = ["one", "two", "three"]
```

**Expression:**

```
join {     // joins can be nested, but the matches don't mix
  a = x
  b = join { c ~ x }         // a and c can match the same x
}
```

**Output:**

```
[(1.1, [1.1, 2.2, 3.3, 4.4, 5.5]),
 (2.2, [1.1, 2.2, 3.3, 4.4, 5.5]),
 (3.3, [1.1, 2.2, 3.3, 4.4, 5.5]),
 (4.4, [1.1, 2.2, 3.3, 4.4, 5.5]),
 (5.5, [1.1, 2.2, 3.3, 4.4, 5.5])]
```

## Back to example: Higgs $\rightarrow$ ZZ $\rightarrow 4\ell$

```
higgs(flavor1, flavor2) =
    join {
        z1 ~ {
            lep1 ~ flavor1
            lep2 ~ flavor1
            mass = (lep1.p4 + lep2.p4).mass
        }
        z2 ~ {
            lep1 ~ flavor2
            lep2 ~ flavor2
            mass = (lep1.p4 + lep2.p4).mass
        }
    }.filter(h => h.z1.lep1.charge != h.z1.lep2.charge and
                  h.z2.lep1.charge != h.z2.lep2.charge)
     .sort(h => (h.z1.mass - 91)**2 + (h.z2.mass - 91)**2)

higgs4e    = higgs(electrons, electrons)
higgs4mu   = higgs(muons, muons)
higgs2e2mu = higgs(electrons, muons)
```

## Other examples?

### Example: gen-reco matching

```
matched_gens =
    join {
        gen ~ gens
        matched = join {                // nested for len(gens) results
            reco ~ recos
            dR = delta_R(gen, reco)
        }.filter(x => x.dR < 0.5)       // match must be within 0.5
         .sort(x => x.dR)[:1]           // only the best match or none
    }.filter(x => len(matched) > 0)     // keep only matched
```

### Example: jet cleaning

```
clean_jets =
    jets.filter(jet => leptons.all(lep => delta_R(jet, lep) > 0.5))
```

(Doesn't even need pattern matching.)

## Database indexing

Matching is equivalent to nested `xs.map(x => ys.map(y => ...))`, apart from the uniqueness and symmetry criteria.

## Database indexing

Matching is equivalent to nested `xs.map(x => ys.map(y => ...))`, apart from the uniqueness and symmetry criteria.

Thus, the key consideration is to maintain the *identity* of each object, including composite identity (e.g. Z in terms of its constituent leptons).

## Database indexing

Matching is equivalent to nested `xs.map(x => ys.map(y => ...))`, apart from the uniqueness and symmetry criteria.

Thus, the key consideration is to maintain the *identity* of each object, including composite identity (e.g. Z in terms of its constituent leptons).

Gordon Watts introduced the idea of considering each physics event as a (tiny) relational database. In this implementation, each input table (collection of particles) has a different surrogate index. Derived quantities propagate those indexes, and joining tables creates composite indexes.

## Consequences of database indexing

- Every collection, even a collection of simple numbers, has an index. A table like `muons` is a set of columns with a common index.

## Consequences of database indexing

- Every collection, even a collection of simple numbers, has an index. A table like `muons` is a set of columns with a common index.
- Could be tracked as compile-time types, preventing typical jagged array errors.

## Consequences of database indexing

- Every collection, even a collection of simple numbers, has an index. A table like `muons` is a set of columns with a common index.
- Could be tracked as compile-time types, preventing typical jagged array errors.
- `union(muons.filter(cuts1), muons.filter(cuts2))` is equal to `muons.filter(m => cuts1(m) or cuts2(m))`, without the double-counting that would come from list concatenation.

## Consequences of database indexing

- Every collection, even a collection of simple numbers, has an index. A table like `muons` is a set of columns with a common index.
- Could be tracked as compile-time types, preventing typical jagged array errors.
- `union(muons.filter(cuts1), muons.filter(cuts2))` is equal to `muons.filter(m => cuts1(m) or cuts2(m))`, without the double-counting that would come from list concatenation.
- If the user attaches some columns to a table named `muons`, it is equivalent to the original `muons` in uniqueness and symmetry matching.

## Consequences of database indexing

- Every collection, even a collection of simple numbers, has an index. A table like `muons` is a set of columns with a common index.
- Could be tracked as compile-time types, preventing typical jagged array errors.
- `union(muons.filter(cuts1), muons.filter(cuts2))` is equal to `muons.filter(m => cuts1(m) or cuts2(m))`, without the double-counting that would come from list concatenation.
- If the user attaches some columns to a table named `muons`, it is equivalent to the original `muons` in uniqueness and symmetry matching.
- User can combine `leptons = union(electrons, muons)`, but each particle remembers that it is an electron or a muon in uniqueness testing.

# Consequences of database indexing

- Every collection, even a collection of simple numbers, has an index. A table like `muons` is a set of columns with a common index.
- Could be tracked as compile-time types, preventing typical jagged array errors.
- `union(muons.filter(cuts1), muons.filter(cuts2))` is equal to `muons.filter(m => cuts1(m) or cuts2(m))`, without the double-counting that would come from list concatenation.
- If the user attaches some columns to a table named `muons`, it is equivalent to the original `muons` in uniqueness and symmetry matching.
- User can combine `leptons = union(electrons, muons)`, but each particle remembers that it is an electron or a muon in uniqueness testing.

In short, particle collections are **sets** with equality by **reference** independent of how they may be **dressed** by attributes.

## What about syntax?

This was an example syntax in a toy language: in no way final.

- Grammar requires an Earley algorithm (slow); may want to finagle it to work with LALR (fast).
- It might be possible to embed this in a host language like C++ or Python, but at the expense of readability (in my attempts).
- What about two-dimensional syntax? (Next page.)

## Behold Racket (Scheme)'s two-dimensional syntax

```
(define (subtype? a b)
  #2dmatch
  +-----------+-----------+--------+-----------+
  |   a   b   | 'Integer  | 'Real  | 'Complex  |
  +-----------+-----------+--------+-----------+
  | 'Integer  |              #t                |
  +-----------+-----------+                    |
  | 'Real     |           |                    |
  +-----------+           +--------+           |
  | 'Complex  |        #f          |           |
  +-----------+--------------------+-----------+)
```

See https://docs.racket-lang.org/2d/index.html

## So maybe something like this?

The ASCII art of the decay would literally be the code used to match it.

```
Higgs: sort (Z1.mass - 91)**2 + (Z2.mass - 91)**2
    |
    +--> Z1: cut lep1.charge != lep2.charge
    |       +--> lep1, lep2 in electrons or lep1, lep2 in muons
    |
    +--> Z2: cut lep3.charge != lep4.charge
            +--> lep3, lep4 in electrons or lep3, lep4 in muons
```

Maybe the arrows are unnecessary; maybe an indentation rule like Python's?

Maybe not.

## Conclusions

- Investigated the use of a pattern-matching syntax to match particle decay chains.
- Good use of the event-is-a-database concept!
- What do you think of the syntax?