21–22 Nov 2022
Kyungpook National University

# Running analyses with ADL: CutLang

Analysis Description Language Tutorial and Hackathon

Gökhan Ünel (ATLAS) UCIRVINE

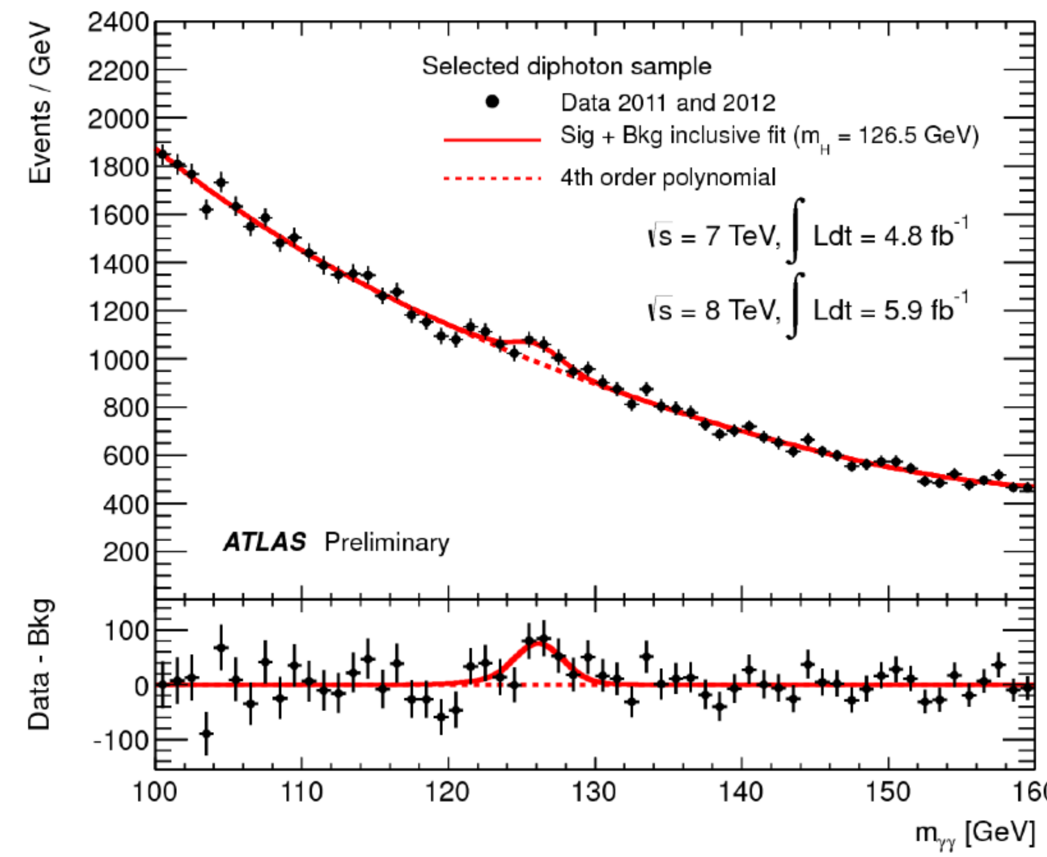
and
the ADL/CutLang team

arXiv:1801.05727

https://cutlang.hepforge.org

- arXiv paper: internals, user manual, how to run examples etc.
- Web page: user manual, examples & source code tgz:

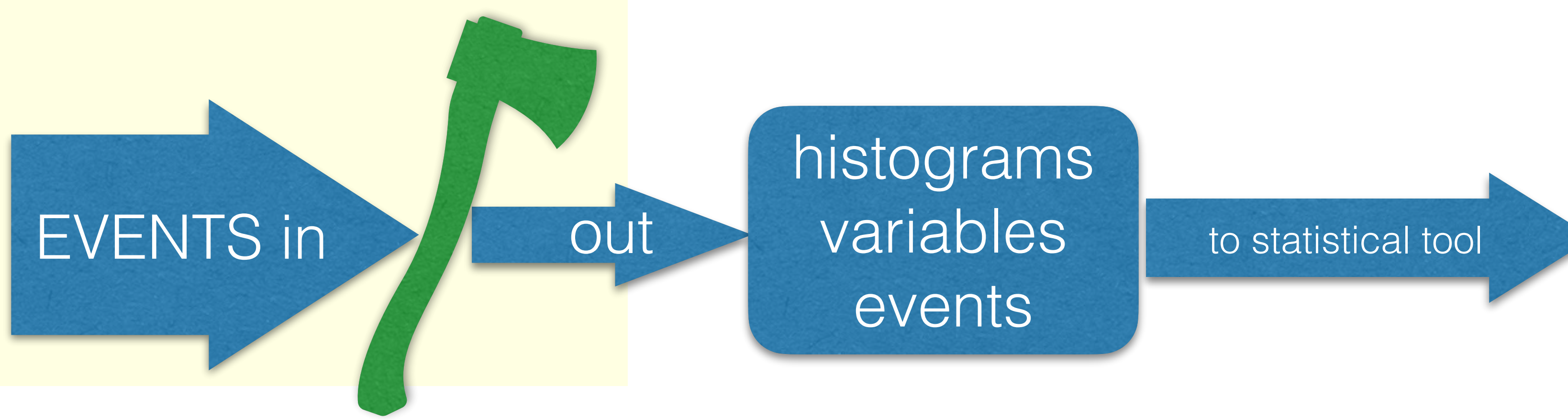
# Data Analysis




“ legacy method

```
 // if (Cut(ientry) < 0) continue;
eff->Fill(1);
jmult->Fill(Jet_);
lmult->Fill(Muon_ + Electron_);
for ( int i=0; i<Jet_; i++) {
  jets[i].SetPtEtaPhiM (Jet_PT[i], Jet_Eta[i], Jet_Phi[i], Jet_Mass[i
  jeteta->Fill (jets[i].Eta() );
  jetphi->Fill (jets[i].Phi() );
  jetPT->Fill (jets[i].Pt() );
}
if ( Jet_ != 2) continue;
eff->Fill(2);
MJJ=jets[0]+jets[1];
jjmass->Fill( MJJ.M() );

MET->Fill(MissingET_MET[0]);
if ( MissingET_MET[0] <20 ) continue;
eff->Fill(3);
```



# there are other tools

CheckMate

```
double HT = 0.;
for(int j = 0; j < hardjets.size(); j++)
   HT += hardjets[j]->PT;
double mEffincl = HT + missingET->PT;

double mEff2 = missingET->PT + jets[0]->PT + jets[1]->PT;
double rEff2 = missingET->PT/mEff2;

double mEff3 = 0;
if (jets.size() >= 3)
    mEff3 =  mEff2 + jets[2]->PT;
double rEff3 = 0;
if (jets.size() >= 3)
    rEff3 = missingET->PT/mEff3;

double mEff4 = 0;
if (jets.size() >= 4)
    mEff4 =  mEff3 + jets[3]->PT;
double rEff4 = 0;
if (jets.size() >= 4)
    rEff4 = missingET->PT/mEff4;
```

MadAnalysis

```
// Muons
for(unsigned int mu=0; mu<event.rec()->muons().size(); mu++)
{
  const RecLeptonFormat *CurrentMuon = &(event.rec()->muons()[mu]);
  if(CurrentMuon->pt()>10. && fabs(CurrentMuon->eta())<2.7)
    Muons.push_back(CurrentMuon);
}
```

Rivet

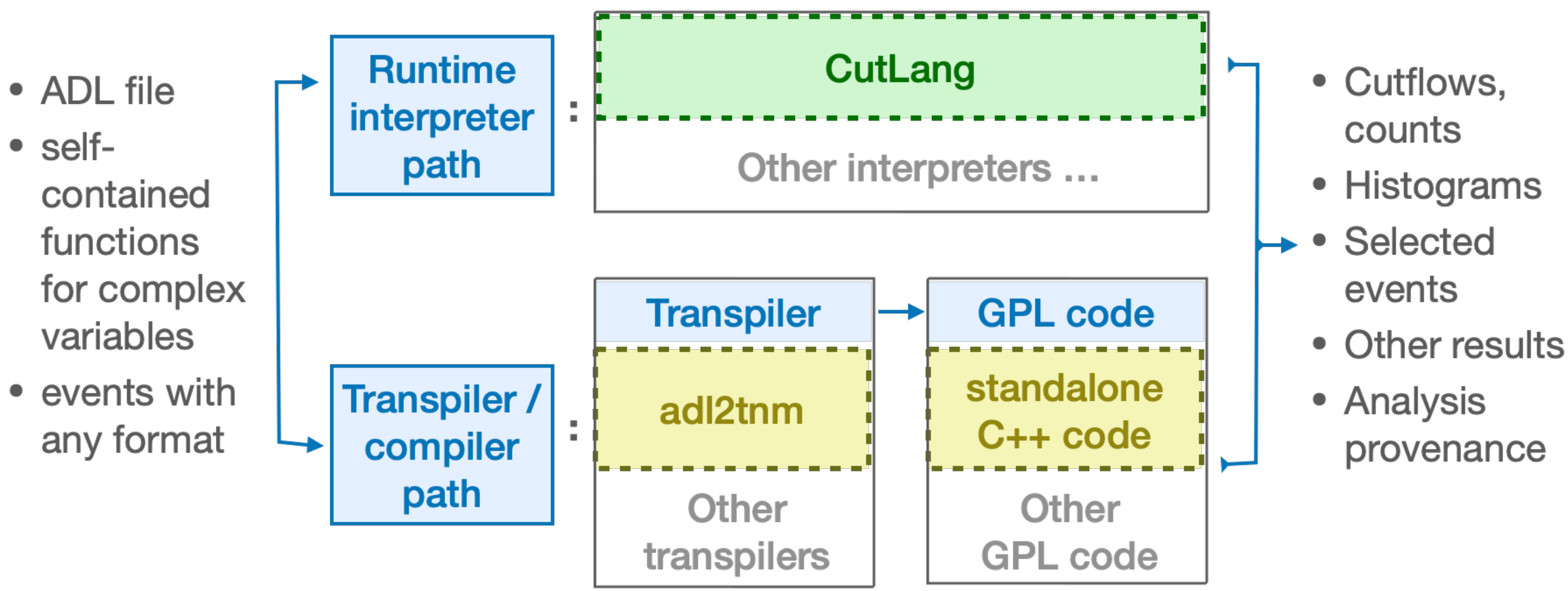
```
const Jets jets =
  applyProjection<JetAlg>(evt, "Jets").jetsByPt(20*GeV);
foreach (const Jet& j, jets) {
  foreach (const Particle& p, j.particles()) {
    const double dr =
      deltaR(j.momentum(), p.momentum());
  }
}
```

# Running analyses with ADL

Once an analysis is written, it needs to run on events.

ADL is multipurpose & framework-independent: It can be translated / integrated into any language or framework for analysis tasks:

Experimental / phenomenology analysis model with ADL




Physics information is fully contained in ADL. Current compiler infrastructures can be easily replaced by future tools / languages / frameworks.

# ADL & CutLang v2

- CutLang v2 is an ADL runtime interpreter
  - Run time interpretation of the ADL file: *No compiling!*
    - replace any cut, any ADL line and rerun the analysis
  - ADL blocks have to be in order:
    - [initializations] [definitions] [objects] [definitions] commands

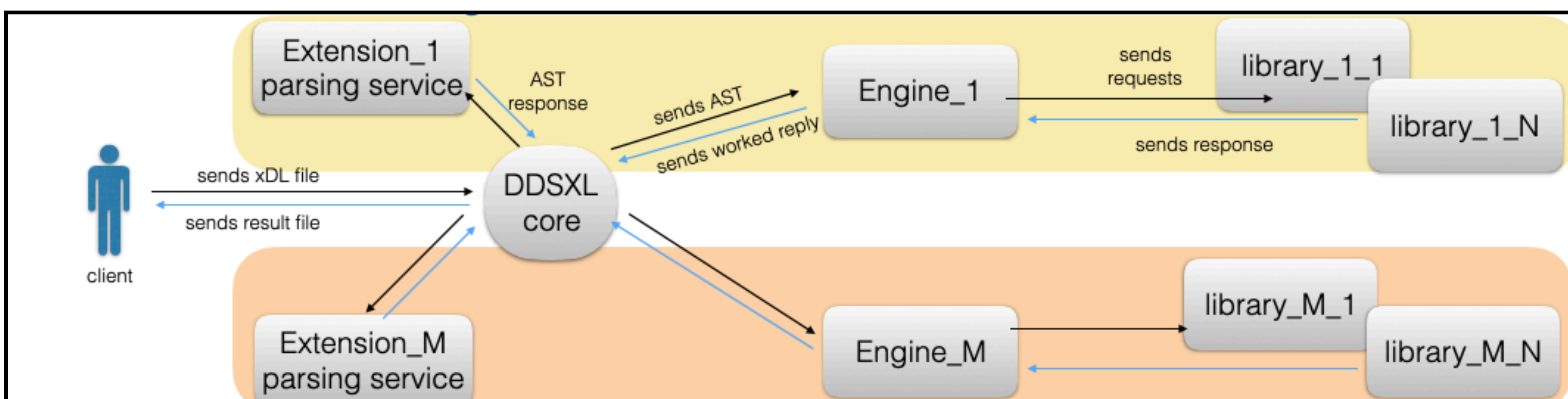
- CutLang v2 provides a full working environment
  - Works on any *nix system, relies on ROOT & C++, lex & yacc for parsing
  - Works with multiple input data formats
    - Currently: LVL0, ATLAS OpenData, CMS OpenData, Delphes, LHCO, FCC, CMSNANOAOD,….
    - more can be easily added…
  - Additional tools to help the analyst and the advisor
    - All definitions, cuts and object selections are saved into the output ROOT file
    - Shell & Python scripts for plotting & addition of “user functions” being updated

# Root, Lex and Yacc

- Root: standart HEP library and analysis tool
  - available since 1994
  - 4 vector operations, histogramming utilities are based on these libraries
- lex: standard unix lexical analyzer
  - available since 1975
  - Linux version is flex
  - helps us to define the "keywords" in ADL / CL and to make it case insensitive
- yacc: yet another compiler compiler
  - available since 1975
  - Linux version is bison
  - helps us to define the grammar of ADL/CL, what to do with the keywords
- An update is in progress
  - all these tools produce a single executable in C/C++, heavy to maintain
  - functions & attributes into different libraries for easy maintenance

# DDSXL

- After DSL-grammar decoupling, next is multiple grammars for multiple domains.
- We designed a new protocol called Dynamic Domain Specific eXtensible Language (DDSXL)
  - it can contain numerous programming languages and frameworks.
    - each developer to integrate their own module independently from other modules
    - 3 independent developer types: maximum efficiency for the developers
    - it allows each micro team to use/integrate solutions they are experts in
  - it integrates a domain ecosystem (such as CL) into the development environment
  - a set of rules determined through communication over the network.



# DDSXL components

- DDSXL Core
  - main service & entry point, always alive, no dependencies
  - all packages register to this service
- Extension (ADL)
  - produces an Abstract Syntax Tree (AST) for the associated engine
- Engine (CutLang)
  - receives the AST, can do basic arithmetic & logic,
  - depends on Library/ies for specific functions
- Library/ies (ML functions, complex kinematic variable functions...)
  - offers recipes for specific functions,
  - can be many running on different hosts / addresses

# DDSXL development & status

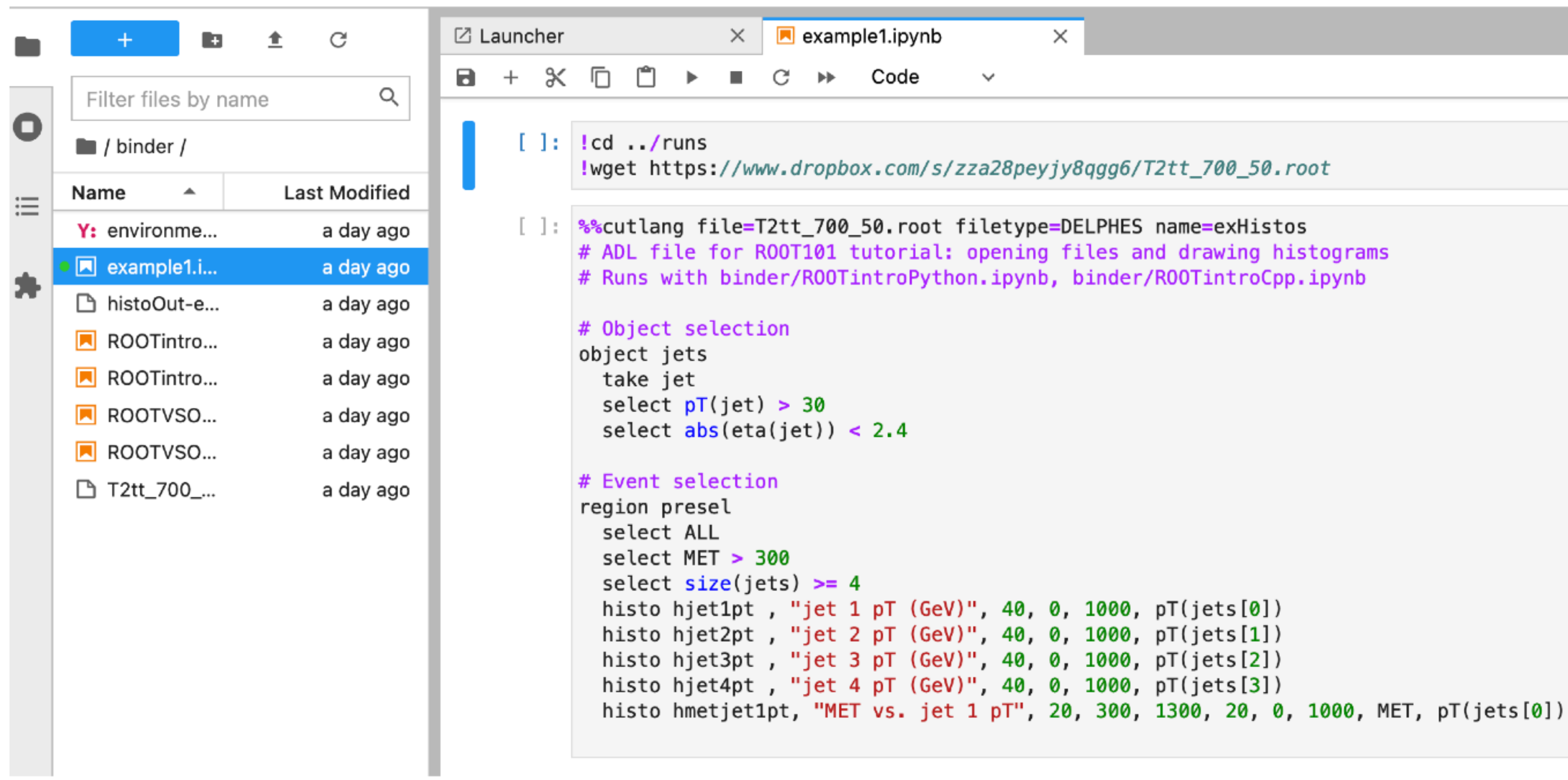
- Developer types in DDSXL excosystem
  - Core developer: experts in RPC, network communications, etc...
  - Extension developer: specializes in parsers, compilers, AST etc...
  - Engine developer: experts in the relevant domain that can solve problems
  - Library developer: researchers in the relevant domain only

- Status
  - Execution protocol steps and technologies to be used are identified
    - gRPC (https://grpc.io/) & GraphQL (https://graphql.org/)
  - Test servers and clients are written, functionality validated
  - Run time library addition successfull
  - Development ongoing

# CutLang works in several environments

To reach the widest possible audience, CutLang runs on various computer platforms:

| | Linux | MacOS | Windows |
|---|---|---|---|
| Native | ✓ | ✓ | |
| Docker | ✓ | ✓ | ✓ |
| Conda cmd line | ✓ | ✓ | |
| Conda -> Jupyter | ✓ | ✓ | |
| Web -> Jupyter | ✓ | ✓ | ✓ |



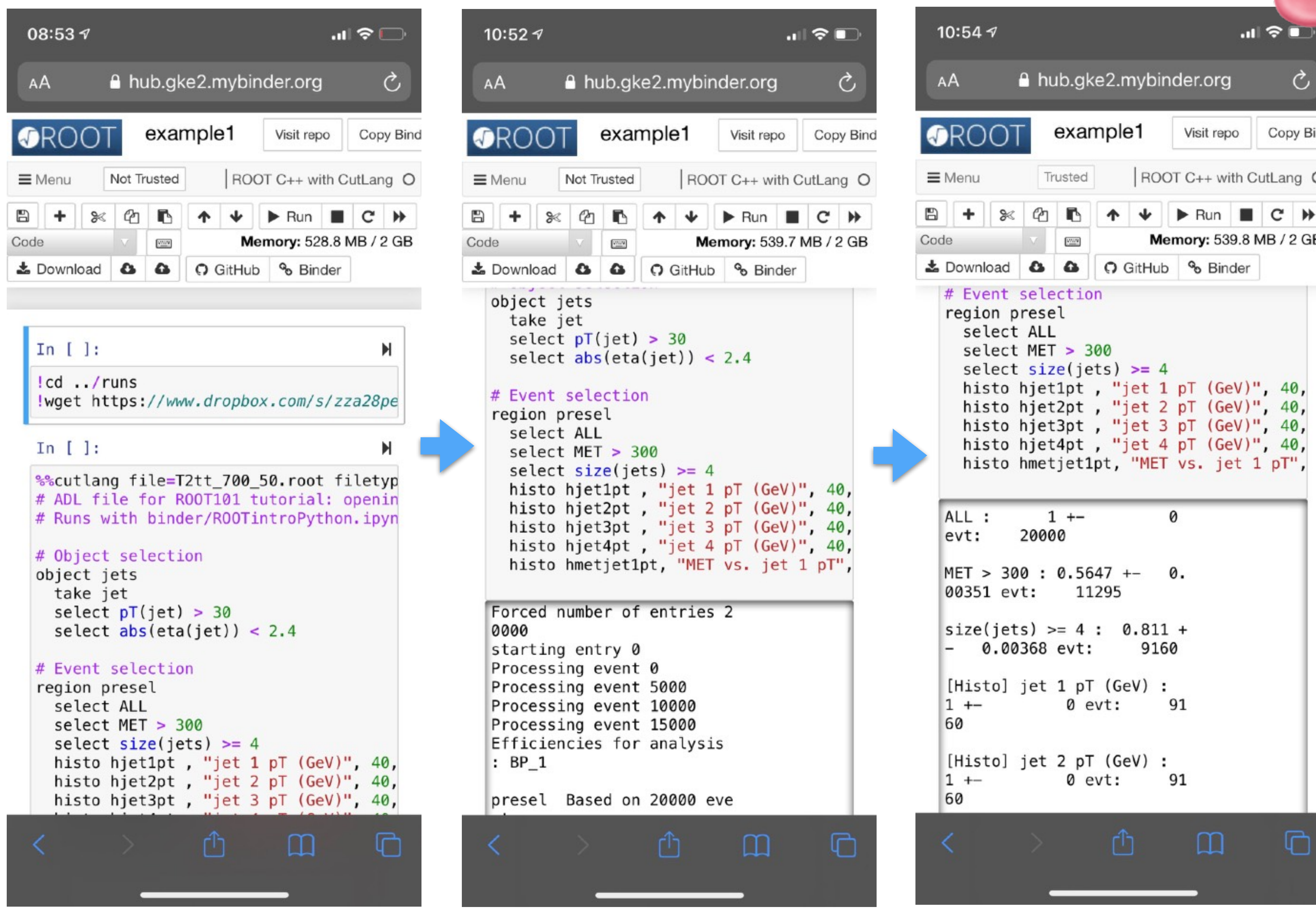

A ROO/C++/CutLang kernel available in jupyter/Binder.

*Docker, Conda interfaces and CutLang kernel for Jupyter written by Burak Şen, METU.*

# CutLang works in several environments

You could even run CutLang on your mobile!

(via Jupyter/Binder interface)




(enter the “binder” dir)

# ADL/CL Syntax (1)

- The execution order is top to bottom.
  - units are in GeV, comment character is #, mostly case insensitive
- Most mathematical functions are available
  - sin(), sinh(), cos(), cosh(), tan(), tanh(), Hstep(), abs(), sqrt(), ^, *, /, +, -, interval inclusion [] and exclusion ][
- Predefined concepts
  - particles are: **ELE**CTRON, **MUO**N, **TAU**, **PHO**TON, **JET**, **F**AT**JET**, **MET**
    - particles are already sorted in decreasing transverse momentum order
  - particle attributes and functions are: charge **q** mass **m**, energy **E**, transverse momentum **pT**, total momentum **P**, pseudorapidity **Eta**, angular distances **dPhi**,…

# particle notation

- On the blackboard, we write

  $jet_1$

  - When you type it in latex it is jet_1
  - CL understands *particleName_index* notation:

| Highest Pt object | Second Highest Pt object |
|---|---|
| ELE_0 | ELE_1 |
| MUO_0 | MUO_1 |

- On the computer, we write

  jet[3]

  - CL understands *particleName*[*index*] notation:

```
muonsVeto[0]
photons[0]
```

# properties, attributes & functions

- Is pseudo rapidity a property of a particle or an attribute? What about the mass? is it an attribute? is it a function?
  - all suggestions are equally valid, and can be used interchangibly
  - I only care about the result of my analysis
- However, when I speak or write I might say either of
  - "the mass of a particle set" m ( )
  - "the particle set's mass" { }m ← more natural in Turkish
- CL understands both notations

| Meaning | Operator | Operator |
|---|---|---|
| Mass of | `m( )` | `{ }m` |
| Charge of | `q( )` | `{ }q` |
| Phi of | `Phi( )` | `{ }Phi` |
| Eta of | `Eta( )` | `{ }Eta` |
| Absolute value of Eta of | `AbsEta( )` | `{ }AbsEta` |
| Pt of | `Pt( )` | `{ }Pt` |

# ADL/CL Syntax (2)

- Main keywords:
    - use **select** / **reject** (or **cmd**) to select/reject events
    - use **define** (or **def**) to define constants, functions and composite particles
    - use **histo** to book and fill histograms
    - use **region** (or **algo**) to define independent algorithms
    - use **object** (or **obj**) to define new/composite particle objects
    - use **sort** to sort particles according to a property
    - use **table** to define a table (currently 1D only)
    - use **weight** to define an event weight
    - use **save** to record surviving events
    - use **Union** to define a new set of particles
    - use **Comb** to construct probability combinatorics

# Derived objects

- Further cleaning or refining can be achieved using derived objects
  - Derived objects can be used to derive further refined objects
    - JETS —> goodJETs —> cleanJETs —> verycleanJets …
  - Multiple selection criteria can be applied
  - The criteria selection line can contain at most 2 different type of objects
    - e.g. reject jets too close to electrons
  - The whole criteria line returns a boolean for the considered pair ($j_i$ and $p_j$)
    - intrinsic loop

```
# jets - no photon
object AK4jetsNOpho : AK4jets
  select dR(AK4jets , photons  ) >=0.4 OR {photons }Pt/{AK4jets }Pt ][ 0.5 2.0
```

- Analysis algorithms can use the original objects or derived objects

# Unions of objects

- It is possible to group together
  - charged leptons,
  - derived objects

- The resulting group is not sorted.
  - use sort cmd

```
object goodEle : ELE
  select  Pt(ELE)         >   10
  select abs({ELE}Eta)    <   2.4
  select     {ELE}AbsEta ][  1.442 1.556

object GMUO : MUO
  select  Pt(MUO)         >   10
  select abs({MUO}Eta)    <   2.4


object geps  : Union( MUO , ELE, TAU)            #add all leptons into
object gleps : Union( goodEle , GMUO )          #add all good electron

region   test
  select      ALL                 # to count all events
  select      Size (goodEle)  >= 1  # events with 2 or more electrons
  select      Size (GMUO)  >= 1  # events with 2 or more electrons
  select      Size (gleps) > 2  # events with 2 or more leptons
```

# using the CMDLine

```
CLA i.root iTYPE -i my.adl -e 20000 -s 100 -j 4
```

```
/CutLang/runs/CLA.sh
/CutLang/runs
/CutLang
ERROR: not enough arguments
/CutLang/runs/CLA.sh ROOTfile_name ROOTfile_type [-h]
    -i|--inifile
    -e|--events
    -s|--start
    -h|--help
    -d|--deps
    -v|--verbose
    -j|--parallel
ROOT file type can be:
 "LHCO"
 "FCC"
 "POET"
 "DELPHES2"
 "LVL0"
 "DELPHES"
 "ATLASVLL"
 "ATLMIN"
 "ATLTRT"
 "ATLASOD"
 "ATLASODR2"
 "CMSOD"
 "CMSODR2"
 "CMSNANO"
 "VLLBG3"
 "VLLG"
 "VLLF"
 "VLLSIGNAL"
```

- i.root : root file of your events
- iTYPE : the type of your root file
- -i my.adl : your analysis text file
- -e 20000 : run for 20000 events
- -s 100 : start from event no 100
- -j 4 : use 4 cores. *(if you have!)*

# Output file

- is a ROOT file
- all regions are processed in parallel and saved as TDirectories
- each TDirectory contains the associated
  - cutlist, definitions and derived objects

```
TFile**         histoOut-ex5.root
 TFile*         histoOut-ex5.root
  KEY: TDirectoryFile   preselection;1  preselect
  KEY: TDirectoryFile   testA;1 testA
  KEY: TDirectoryFile   testB;1 testB
```

```
root [2] testA->cd()
(bool) true
root [3] .ls
TDirectoryFile*         testA   testA
 KEY: TText    CLA2cuts;1
  select       ALL
  select       Size(ELE)     >= 2
  select       {Zreco}q == 0
  histo        h2mReco    , "Z candidate mass (GeV)", 100, 0, 200, m(Zreco)
  select       ALL

 KEY: TText    CLA2defs;1
define Zreco : ELE[0] ELE[1]
define goodZreco : goodEle[0] goodEle[1]

 KEY: TText    CLA2Objs;1
object goodEle : ELE
  select    Pt(ELE_)        >   10
  select abs({ELE_}Eta )  <   2.4
  select      {ELE_}AbsEta ][  1.442 1.556

 KEY: TH1F      eff;1   selection efficiencies
 KEY: TNtuple   rntuple;1       run info
 KEY: TH1D      h2mReco;1       "Z candidate mass (GeV)"
```

# saving and printing

- It is possible to save events, histograms, etc at any stage of the analysis
  - save filename
    - do **not** write the .root extension, it will be added automatically.
  - there can be multiple save commands in the same region
    - use different names

- it is possible to save variables for later use
  - save filename cvs variablelist
    - ```
      save     mid1file csv pT(ELE_0) {ELE_0 ELE_1}m  eta(ELE_1)
      ```

- print command is used for debugging
  - ```
    print pt(JET[0] ) pt(JET[1] ) chi2Vr eventno  index(VBoosted)
    ```

# Histogramming

- 1 and 2 D histograms are available
  - ```
    histo hjet1pt , "jet 1 pT (GeV)", 40, 0, 1000, pT(jets[0])
    ```
    - it is customary to give a histo name starting with h
    - any title text can be given within quotation marks
    - number of bins, min, max, function to be plotted
  - ```
    histo hmetjet1pt-abc1, "MET vs. jet 1 pT", 20, 300, 1300, 20, 0, 1000, MET, pT(jets[0])
    ```
    - do not use _ (underscore) but use - (minus) in names
    - x parameters, y parameters, x function, y function
- Variable bin size is also possible
  - ```
    histo hmeta, "MET (GeV)", 0.0  10.0  20.0  50.0  100.0  500.0, MET
    ```
    - bin boundaries are space separated not comma
    ```

# ADL syntax: functions

- Standard/internal functions: Sufficiently generic math and HEP operations are a part of the language and any tool that interprets it.
  - Math functions: abs(), sqrt(), sin(), cos(), tan(), log(), …
  - Collection reducers: size(), sum(), min(), max(), any(), all(),…
  - HEP-specific functions: dR(), dphi(), deta(), m(), ….
  - Object and collection handling: sort, comb(), union()…
- External/user functions: Variables that cannot be expressed using the available operators or standard functions would be encapsulated in self-contained functions that would be addressed from the ADL file.
  - Variables with non-trivial algorithms: MT2, aplanarity, razor variables, …
  - Non-analytic variables: Object/trigger efficiencies, variables/efficiencies computed with ML, …

ADL syntax rules: https://twiki.cern.ch/twiki/bin/view/LHCPhysics/ADL

# User (external) functions

- User defined selection functions are somewhat difficult to incorporate into an interpreter

- Currently we define a user function type and compile it in.
  - CLv2 will provide the means to do this automatically
  - Currently Razor functions are pre-integrated:

```
std::vector<TLorentzVector> fmegajets(std::vector<TLorentzVector> myjets);
double fMR(std::vector<TLorentzVector> j);
double fMTR(std::vector<TLorentzVector> j, TVector2 amet);
double fMTR2(std::vector<TLorentzVector> j, TLorentzVector amet);
```

- Simple functions can be interpreted using CL math functions

```
define MJ = Sum( m(largejets) )
define MTl = Sqrt( 2*pT(leptons[0]) * MET*(1-cos(phi(METLV[0]) - phi(leptons[0]) )))
define ST = fHT(jets) + pT(leptons[0])
```

# optimization functions

- search all possible combinations to find the "best" candidates

$$W \to jj$$

```
define jetA : goodJet[-1]
define jetB : goodJet[-2]
```

1) define your unknowns with negative indices

```
define Whad1 : jetA jetB
```

2) use defined objects as regular objects

```
define WWchi : (m(Whad1) - 80.4)^2 /10^2
```

3) define a $\chi^2$ to optimize

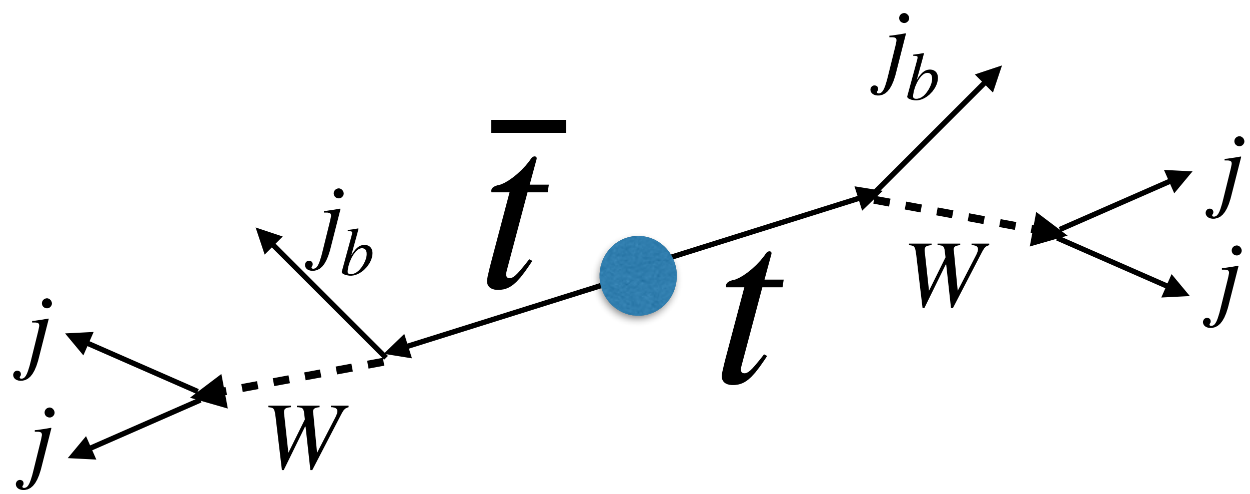
```
select  WWchi ~= 0
```

4) minimize the $\chi^2$ to find the unknowns

We use negative indices if they are to be determined at run time, using a criterion, such as: **~=**

Repeating the same negative value helps speeding up since $j_i\, j_k = j_k\, j_i$

# $t\bar{t}$ Reconstruction example




$$t \rightarrow Wb \rightarrow jjj_b$$

There are 6 jets in the event <u>of which 2 can be b-tagged</u>

+ LOTS of *other jets* from spectator quarks and QCD effects

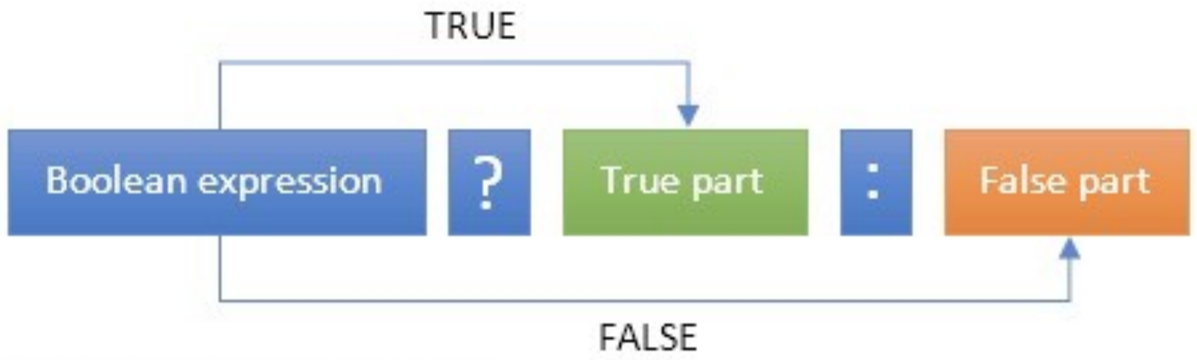
## Which one is which?

with the $\chi^2$ defined as:

$$\chi^2 = \frac{(m_{b_1 j_1 j_2} - m_{b_2 j_3 j_4})^2}{\sigma^2_{\Delta m_{bjj}}} + \frac{(m_{j_1 j_2} - m_{\mathrm{W}}^{\mathrm{MC}})^2}{\sigma^2_{m_{\mathrm{W}}^{\mathrm{MC}}}} + \frac{(m_{j_3 j_4} - m_{\mathrm{W}}^{\mathrm{MC}})^2}{\sigma^2_{m_{\mathrm{W}}^{\mathrm{MC}}}}.$$

# the ternary function

- We use C notation




```
select size(jets) >= 3 ? dphi(jets[2], MHTLV) > 0.3 : ALL
select size(jets) >= 4 ? dphi(jets[3], MHTLV) > 0.3 : ALL
```

- condition ? do if true
  : do if false

# weights

- weights are needed for MC processes
  - simulate the relative importance of certain events
  - simulate the efficiencies (trigger, pileup, vertex, others…)
- Two possibilities
  - via a simple coefficient
  - via  a table

```
weight   randWeight   1.123
weight   effWeight effTable( {ELE_0}pT )   # new

histo    h1ept, "E0 pt (GeV)", 100, 0, 2000, {ELE_0}pT
```

```
table    effTable
#        value   min     max
           0.1   0.0    10.0
           0.2  10.0    20.0
           0.4  20.0    50.0
           0.7  50.0    70.0
           0.95 70.0    1000.0



region    test
  select  ALL                      # to
```

Lets assume we have 5 jets 1 2 3 4 5

we can make 2 hadronic Zs

# Combinations

12 34
12 35
12 45
13 24
13 25
13 45
……

CutLang code to **define** all possibilities, with some cuts:

```
object hZs : COMB( jets[-1] jets[-2] ) alias ahz
     select { ahz }AbsEta < 3.0
     select {jets[-2] }Pt > 2.1
     select {jets[-1] }Pt > 5.1
     select {jets[-1], ahz }dR < 0.6   #--- means a member of hZs a
     select { ahz }m [] 10 200           # does get the paricle num
```

CutLang code to **use** those cuts:

```
region testA
select Count(hZs)   >= 2
```

12 ~~34~~
12 35
12 ~~45~~
13 24
13 25
13 ~~45~~
……

Some combinations are removed because of the selection cuts above.
Lines with 1 remaining Zh are removed since we required at least 2 hadronic Zs

*But which combination to use?*

12 35
13 24
13 25
……

```
define zham : {hZs[-1]}m
define zhbm : {hZs[-2]}m
define chi2 : (zham - 91.2)^2 + (zhbm - 91.2)^2
….
select chi2  ~= 0
```

# Enjoy analyzing LHC data with CutLang


you will learn <u>more</u> during the next session

**Hands-on exercises**

p r a c t i c e   m a k e s   p e r f e c t

# why not python?

- Python is very popular, all new students know it
  - they also know math and english, expressing analysis algorithm in a human readable language is easier to understand and debug
  - decoupling computing and physics helps with both technical and algorithm point of views
- Python has lots of external libraries (sciPy,NumPy,..)
  - when your only tool is a hammer, it is natural to see all problems as nails.
    - why should we read & decipher complex math functions to understand the analysis algorithm? Separate the two.
  - you can access the same functions via ADL "external" function calls
- Learning Python will be useful for the life after physics!
  - let us worry about physics first! Our goal should not be teaching a computing language or helping people to practice it.
  - using the best tool for the job helps getting the best results.
    - swiss army knife (a GPL) has a screw driver too, but when I work on my projects I use an easy grip screwdriver (a DSL)
- Python has better graphics libraries, young people hate ROOT.
  - ADL is a language, feel free to write your own interpreter/compiler with Python.
- It is possible to write a clean analysis code with Python too!
  - Sure. When one writes such a clean code and decouples from computing technicalities, one ends up with ADL.
  - But, we usually have either spagetti code with physics ideas entangled with classes or "organized" code with classes calling classes calling classes calling...