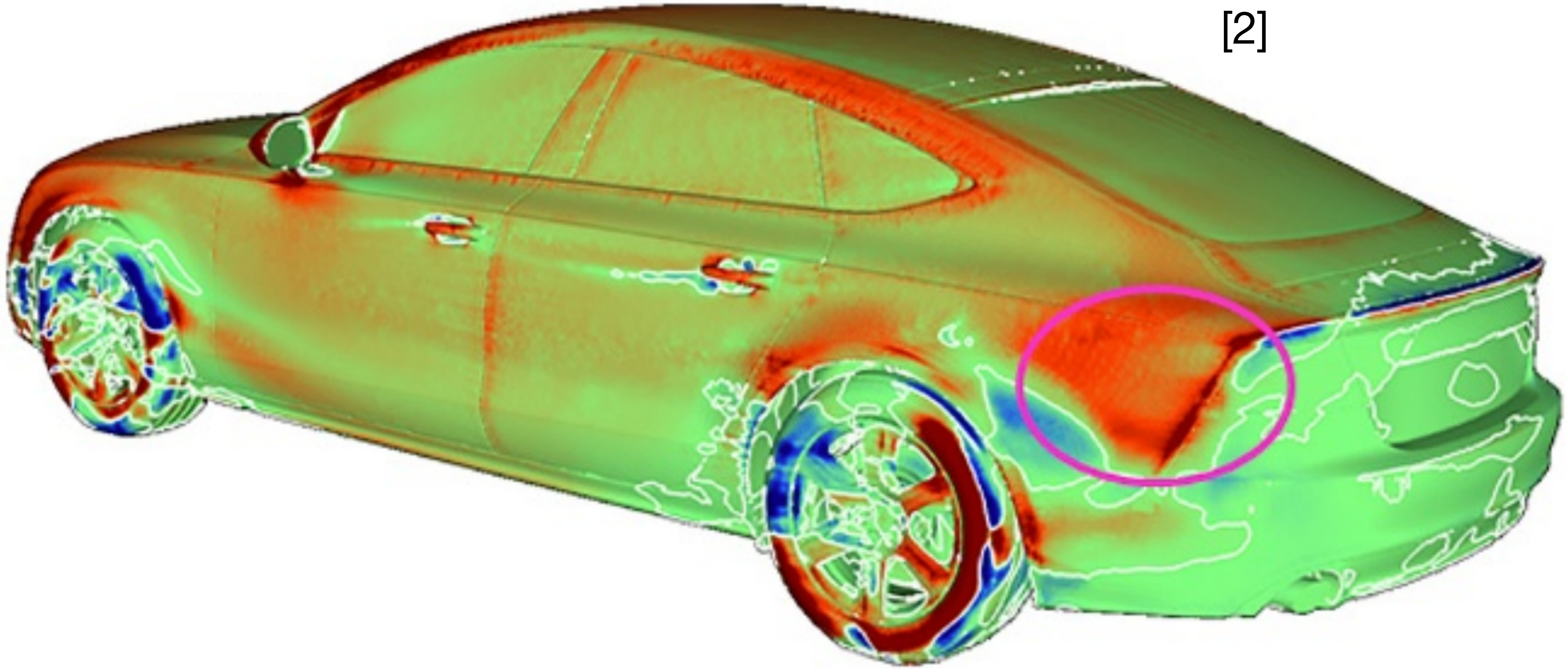
The adjoint method and code generation for shape optimization

Span Spanbauer

Shape optimization

Shape optimization

Goal: Adjust a shape as part of a PDE-constrained optimization



[2]

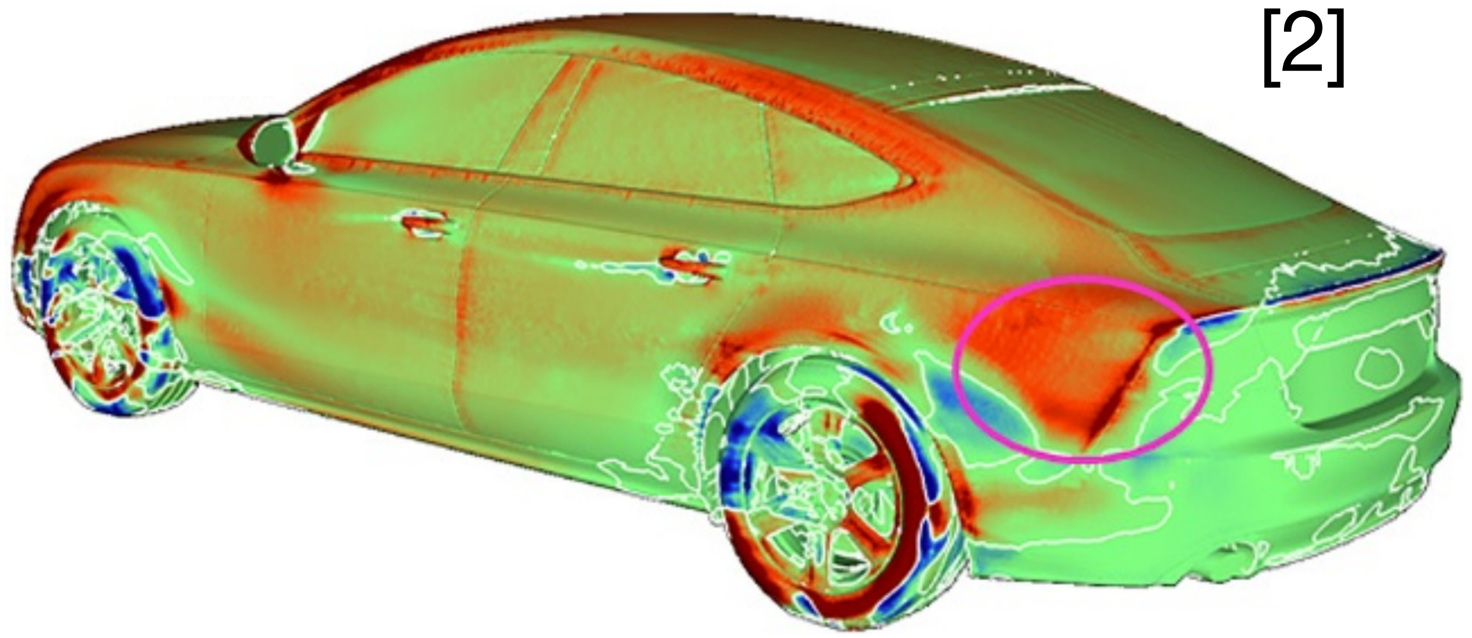
To improve aerodynamics: push red in, pull blue out

This is still a new field: applications in aeronautics began in the 1990s. [1]

[1] Bijan Mohammadi and Olivier Pironneau. “Shape Optimization in Fluid Mechanics.” Annu. Rev. Fluid Mech. 2004. 36:255-79
[2] Carsten Orthmer. “Adjoint methods for car aerodynamics.” Journal of Mathematics in Industry 2014, 4:6

The adjoint method

Adjoint method



[2]

This depicts the gradient of a cost function, specifically the drag coefficient.

The adjoint method **efficiently computes gradients of cost functions,**

that is, the derivative of a single quantity with respect to many parameters. [3] [4] [5] [8]

This is the same procedure that is used to train neural networks, called **backpropagation** in that field.

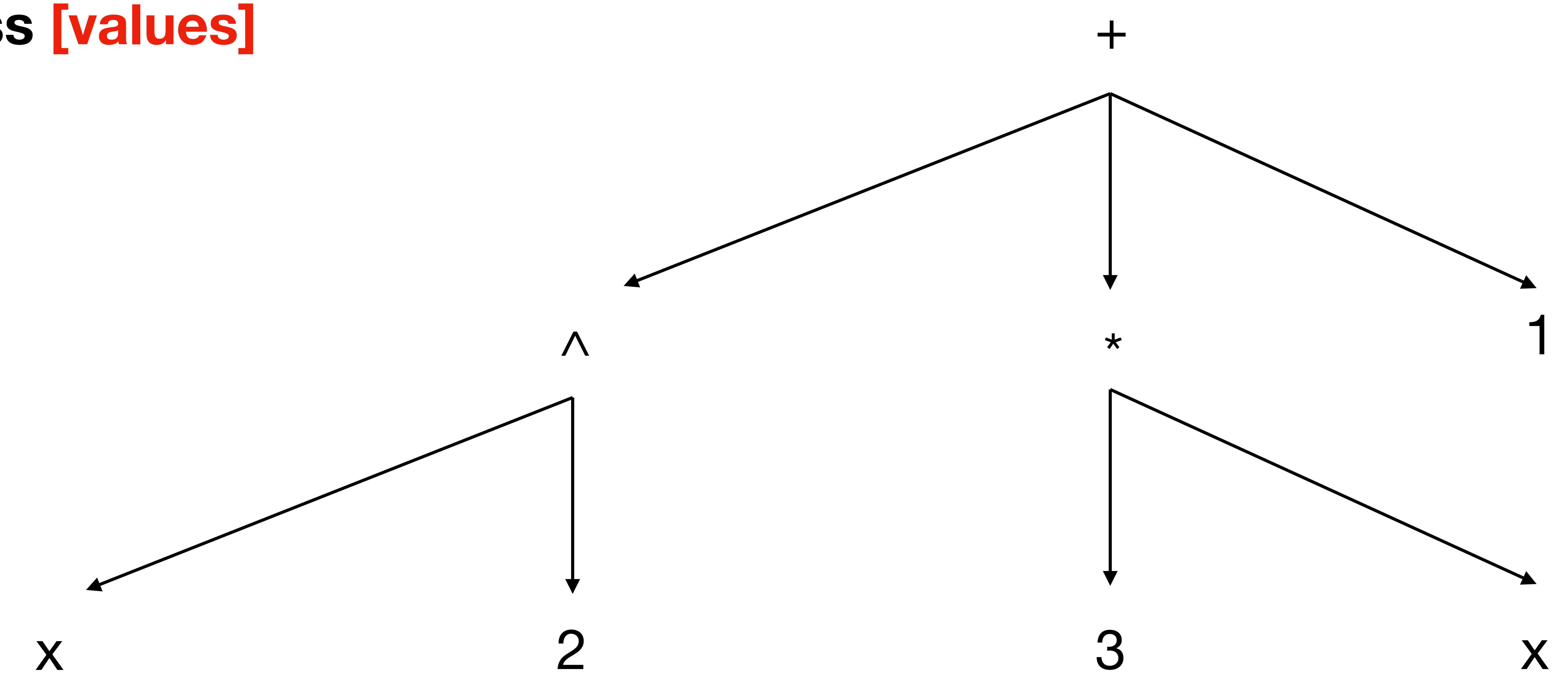
[2] Carsten Orthmer. “Adjoint methods for car aerodynamics.” Journal of Mathematics in Industry 2014, 4:6
[3] Steven Johnson. “Notes on Adjoint Methods for 18.335”, Spring 2006, updated Dec. 17, 2012.
[4] Gregoire Allaire. “A review of adjoint methods for sensitivity analysis, uncertainty quantification, and optimization in numerical codes.” Ingenieurs de l’Automobile, SIA, 2015, 836, pp.33-36.
[5] Dougal Maclaurin. “Modeling, Inference and Optimization with Composable Differentiable Procedures.” Thesis, Harvard 2016.
[8] Cristian Homescu. “Adjoints and automatic (algorithmic) differentiation in computational finance” arXiv:1107.1831v1 [q-fin.CP] 10 Jul 2011. Good general introduction, but errors in some technical details.

Adjoint method

Idea: Store the process involved in calculating y, then work backwards calculating the derivative of each subexpression using the chain rule.

This expression is x^2 + 3*x + 1

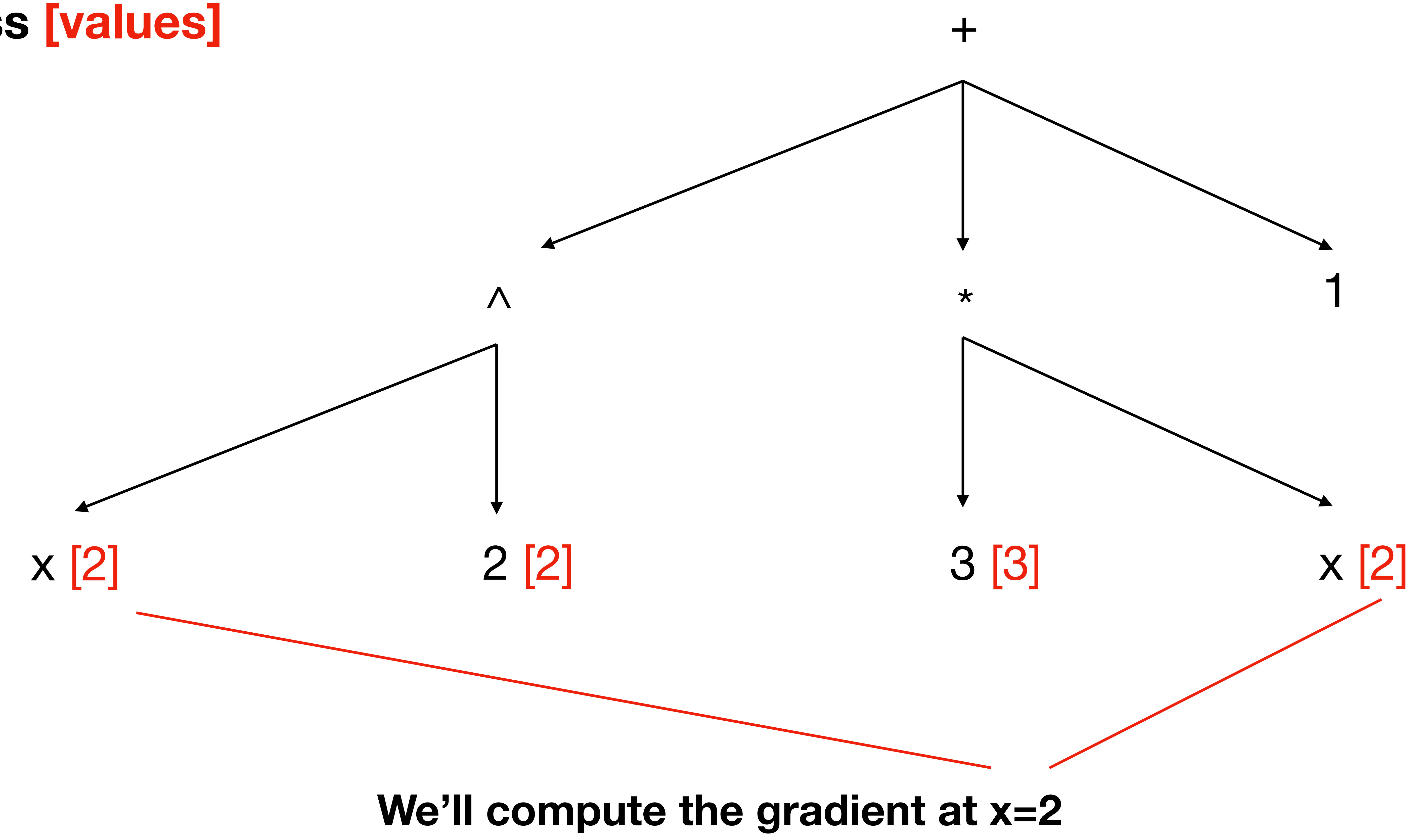
Forward Pass **[values]**



Adjoint method

Idea: Store the process involved in calculating y, then work backwards calculating the derivative of each subexpression using the chain rule.

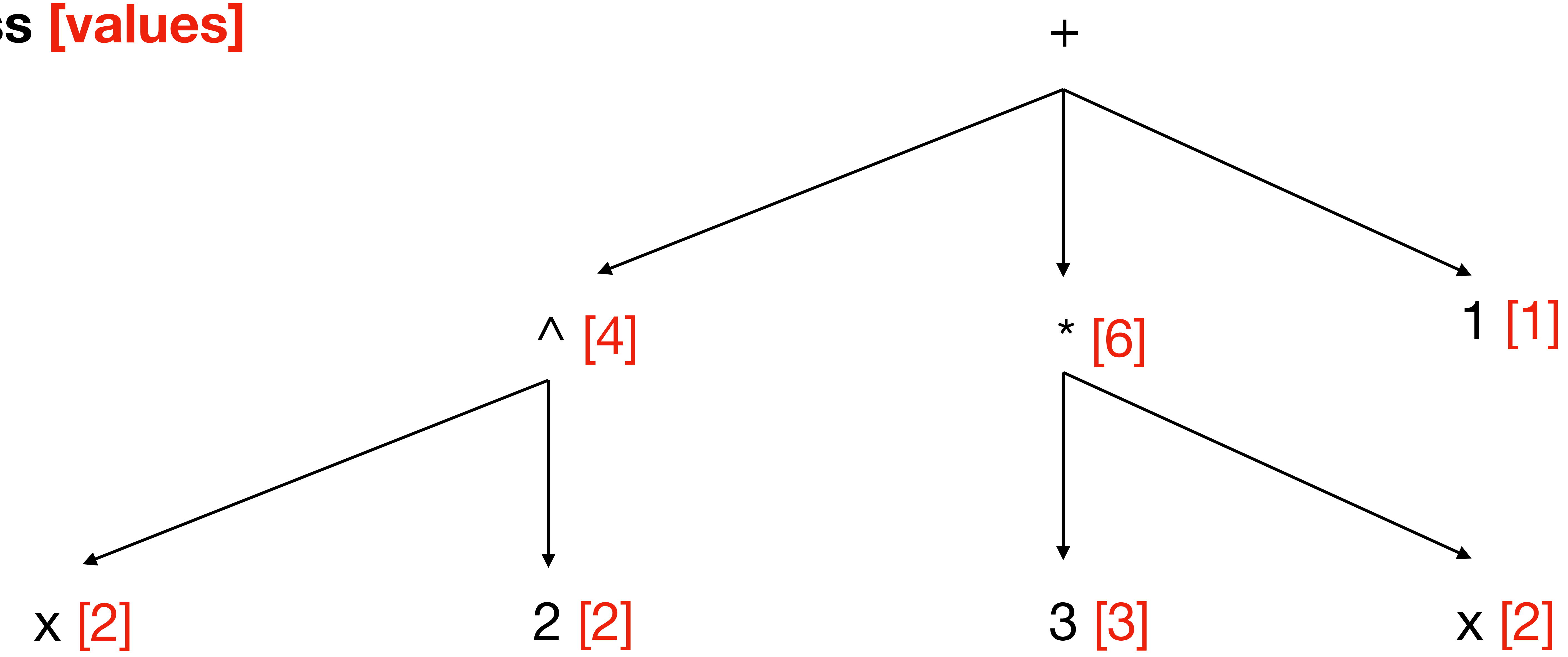
Forward Pass [values]



Adjoint method

Idea: Store the process involved in calculating y, then work backwards calculating the derivative of each subexpression using the chain rule.

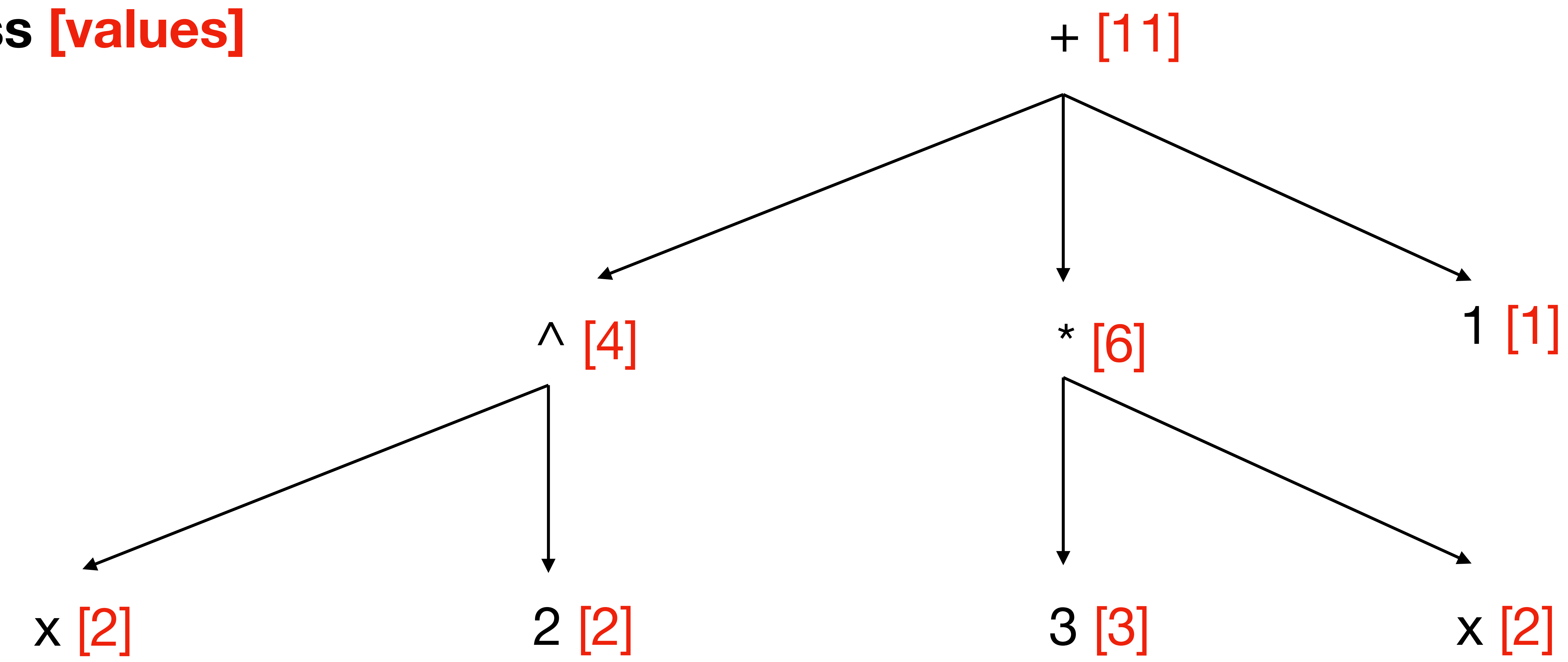
Forward Pass [values]



Adjoint method

Idea: Store the process involved in calculating y, then work backwards calculating the derivative of each subexpression using the chain rule.

Forward Pass [values]

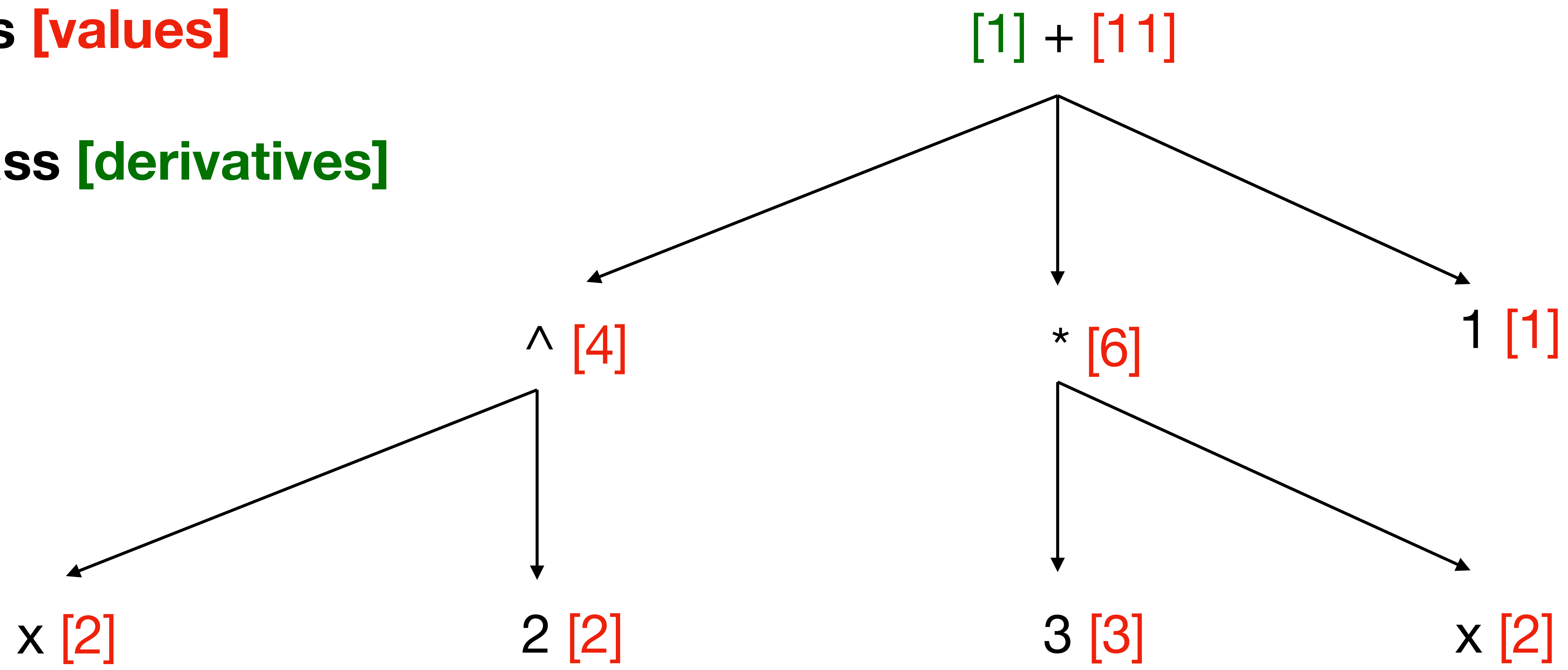


Adjoint method

Idea: Store the process involved in calculating y, then work backwards calculating the derivative of each subexpression using the chain rule.

Forward Pass [values]

Backward Pass [derivatives]

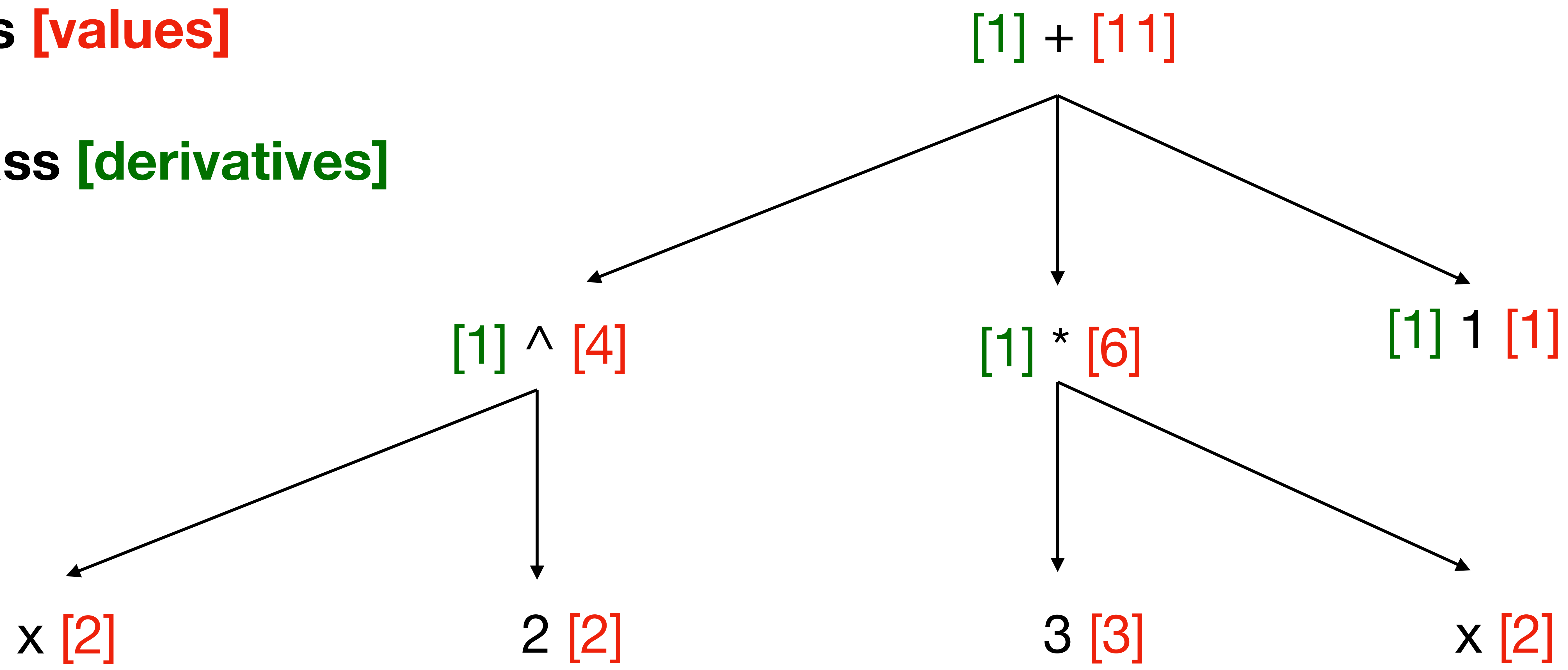


Adjoint method

Idea: Store the process involved in calculating y, then work backwards calculating the derivative of each subexpression using the chain rule.

Forward Pass [values]

Backward Pass [derivatives]

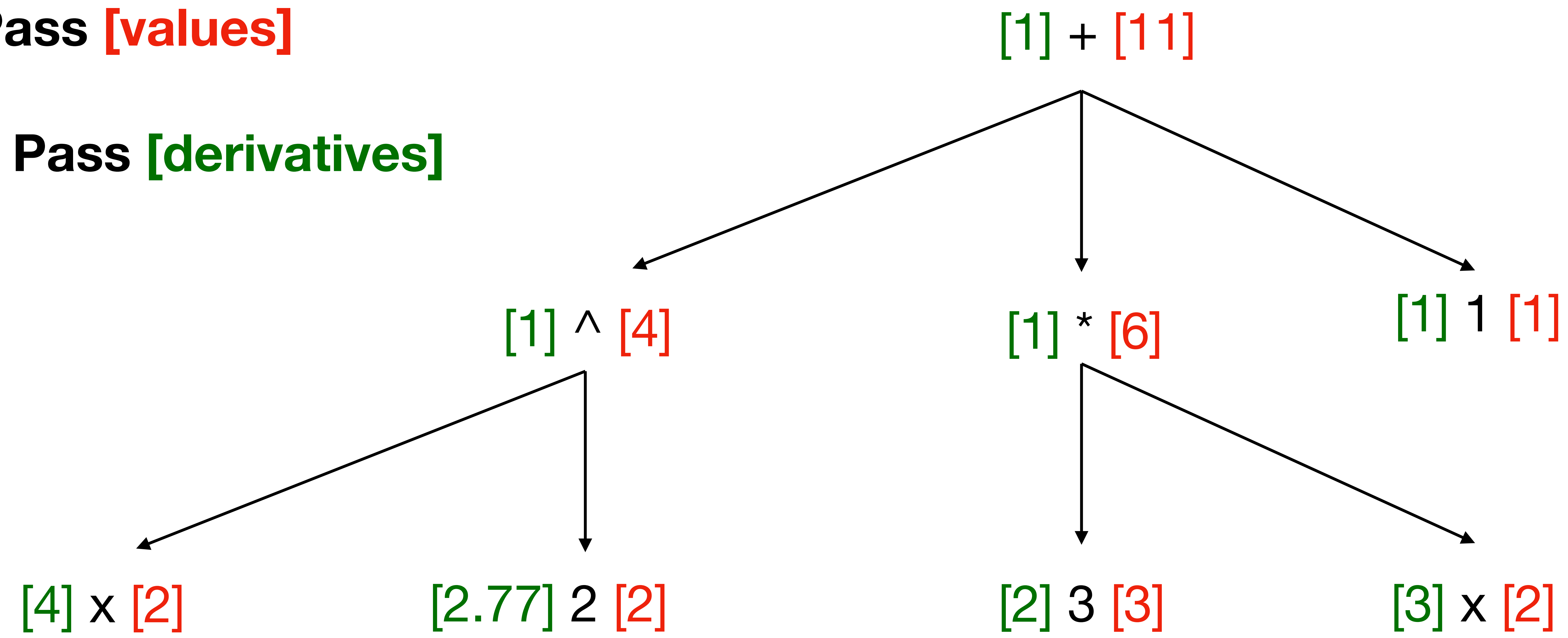


Adjoint method

Idea: Store the process involved in calculating y, then work backwards calculating the derivative of each subexpression using the chain rule.

Forward Pass [values]

Backward Pass [derivatives]

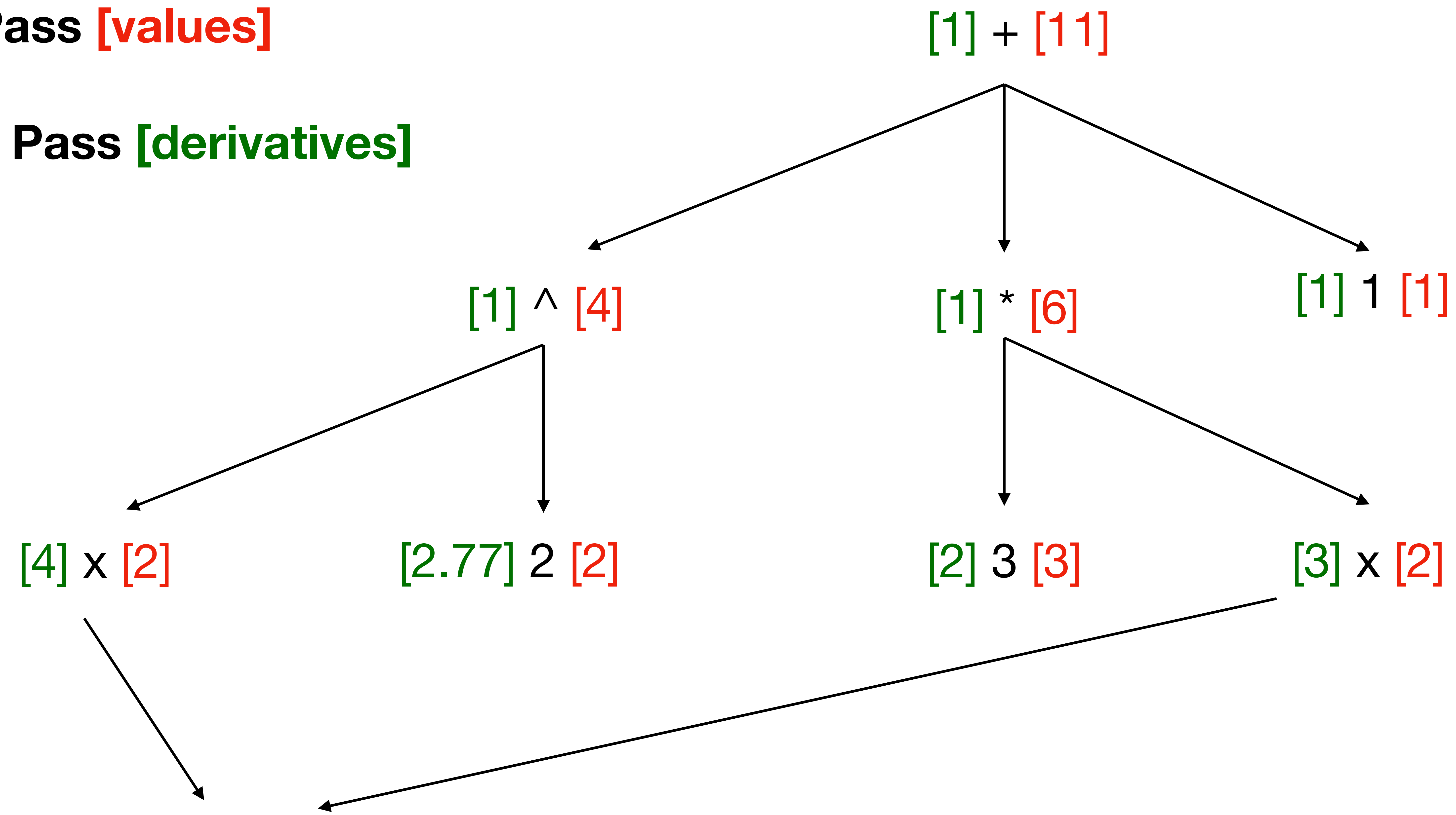


Adjoint method

Idea: Store the process involved in calculating y, then work backwards calculating the derivative of each subexpression using the chain rule.

Forward Pass [values]

Backward Pass [derivatives]



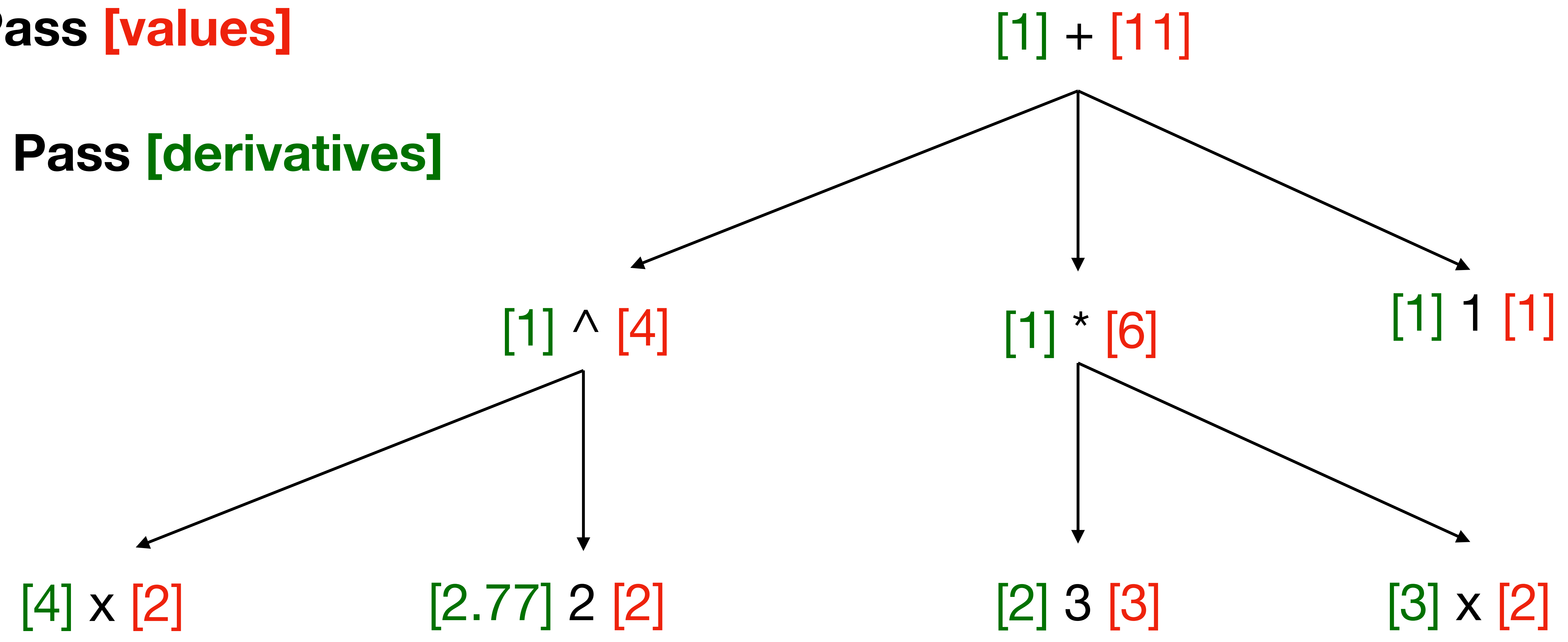
So at x=2, dy/dx = 4 + 3 = 7

Adjoint method

Idea: Store the process involved in calculating y, then work backwards calculating the derivative of each subexpression using the chain rule.

Forward Pass [values]

Backward Pass [derivatives]



Notice that calculating the derivatives of **ALL** parameters took only approximately the same time as calculating the function itself!

Code generation

Code generation

Idea:

Instead of coding your simulation directly, represent it in a domain-specific language.

Then formally transform your description into code.

Benefits:

Can describe the simulation at a high level of abstraction

Can produce efficient, optimized C/FORTRAN code without ever having to read or write C/FORTRAN.

Can automatically generate adjoint code for computing gradients.

[6] Farrell, Ham, Funke, and Rognes. “Automated derivation of the adjoint of high-level transient finite element programs.” arXiv:1204.5577v2 [cs.MS] 16 Oct 2013.

Implementation

Implementation

Wrote a domain specific language and a code generation framework capable of generating forward and adjoint code.

```
example = Program
    [
    scalarV "a" (Just 0.2),
    scalarV "b" (Just 0.3),
    scalarV "c" (Just 0.4),
    scalarV "u" Nothing,
    scalarV "v" Nothing,
    scalarV "w" Nothing,
    scalarV "f" Nothing,
    ]
    [
    set "u" ( sin("a"*"b") + "c"*"b"**2 + "a"**3*"c"**2 ),
    set "v" ( exp("u"**2 - 1) + "a"**2 ),
    set "w" ( ln("v"**2+1) + cos("c"**2-1) ),
    set "f" (("w"-7)**2)
    ]
```

initialize variables

program

objective function

→ (next slide)

Also supports defining and solving linear systems (not shown)

```python
a = 0.2 ; b = 0.3 ; c = 0.4 ; u = 0 ; v = 0 ; w = 0 ; f = 0 ;

state = [{"a":a,"b":b,"c":c,"u":u,"v":v,"w":w,"f":f,"e":e,"q":q}]

state.append(copy(state[-1])) ; state[-1]["u"] = u; state[-1]["__updated"] = "u"
u = ((np.sin((a*b))+(c*(b**2.0)))+((a**3.0)*(c**2.0)))

state.append(copy(state[-1])) ; state[-1]["v"] = v; state[-1]["__updated"] = "v"
v = (np.exp(((u**2.0)-1.0))+(a**2.0))

state.append(copy(state[-1])) ; state[-1]["w"] = w; state[-1]["__updated"] = "w"
w = (np.log(((v**2.0)+1.0))+np.cos(((c**2.0)-1.0)))

state.append(copy(state[-1])) ; state[-1]["f"] = f; state[-1]["__updated"] = "f"
f = ((w-7.0)**2.0)



# BACKWARD PASS

_a = 0 ; _b = 0 ; _c = 0 ; _u = 0 ; _v = 0 ; _w = 0 ; _f = 1 ;

adjoint_state = len(state)

adjoint_state -= 1 ; exec(state[adjoint_state]["__updated"] + " = state[adjoint_state][state[adjoint_state][\"__updated\"]]")
_w += _f * (2.0*(w-7.0)) ;
__new = 0 ; _f = __new

adjoint_state -= 1 ; exec(state[adjoint_state]["__updated"] + " = state[adjoint_state][state[adjoint_state][\"__updated\"]]")
_c += _w * (-((2.0*c)*np.sin(((c**2.0)-1.0)))) ; _v += _w * ((2.0*v)/((v**2.0)+1.0)) ;
__new = 0 ; _w = __new

adjoint_state -= 1 ; exec(state[adjoint_state]["__updated"] + " = state[adjoint_state][state[adjoint_state][\"__updated\"]]")
_a += _v * (2.0*a) ; _u += _v * (np.exp(((u**2.0)-1.0))*(2.0*u)) ;
__new = 0 ; _v = __new

adjoint_state -= 1 ; exec(state[adjoint_state]["__updated"] + " = state[adjoint_state][state[adjoint_state][\"__updated\"]]")
_a += _u * ((b*np.cos((a*b)))+((c**2.0)*(3.0*(a**2.0)))) ; _b += _u * ((a*np.cos((a*b)))+(c*(2.0*b))) ; _c += _u * ((b**2.0)+((a**3.0)*(2.0*c))) ;
__new = 0 ; _u = __new
```

Forward pass

We keep track of state changes, which is used in the backward pass

initialize adjoint variables

Backward Pass

revert state

Results

Wrote an environment for performing Galerkin FEM

which computes the resulting linear system **symbolically**

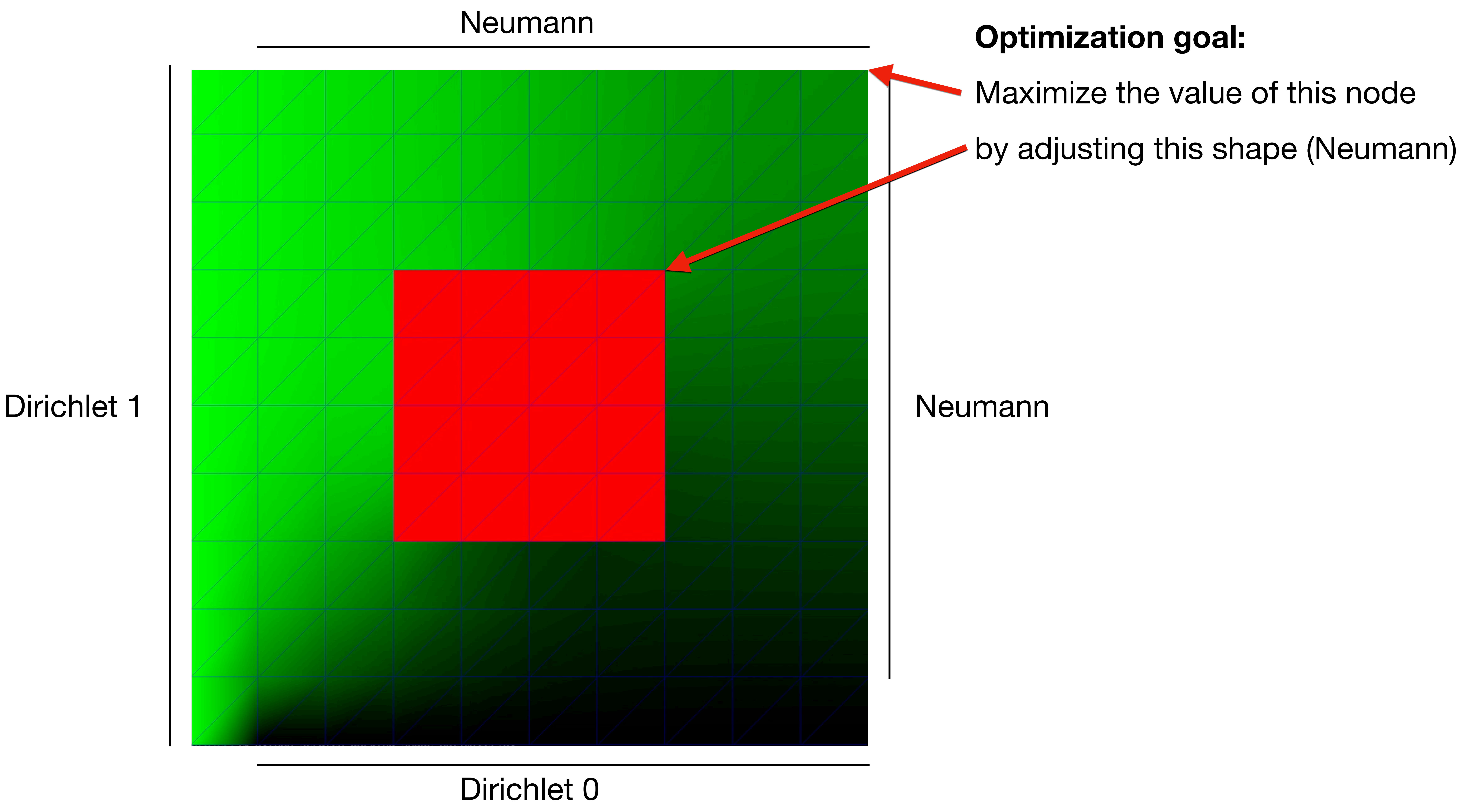
which is required for computing symbolic matrix derivatives used in the adjoint method.

This is the first entry in the 91x91 matrix used to solve the Laplace equation

$$\frac{0.5\left((x[0,9]-x[1,9])^2+(y[0,9]-y[1,9])^2\right)}{\left(((x[1,9]-x[1,10])(y[0,9]-y[1,9])-(x[0,9]-x[1,9])(y[1,9]-y[1,10]))^{2.}\right)^{0.5}}+$$
$$\frac{0.5\left((x[0,9]-x[0,10])^2+(y[0,9]-y[0,10])^2\right)}{\left(((-x[0,9]+x[1,10])(y[0,10]-y[1,10])-(x[0,10]-x[1,10])(-y[0,9]+y[1,10]))^{2.}\right)^{0.5}}+$$
$$\frac{0.5\left((-x[1,9]+x[2,10])^2+(-y[1,9]+y[2,10])^2\right)}{\left(((-x[1,9]+x[2,10])(y[1,10]-y[2,10])-(x[1,10]-x[2,10])(-y[1,9]+y[2,10]))^{2.}\right)^{0.5}}$$

Results

Applied this framework to a shape optimization problem over the Laplace equation.



Observations / next steps

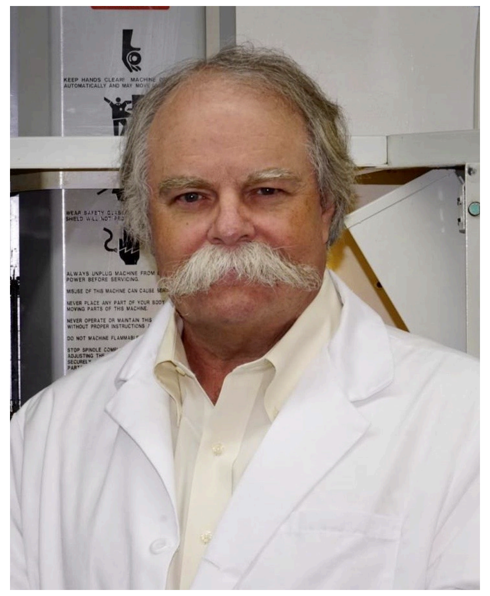
Observations:

1. Shape optimization leads to unstructured grids, so FEM is a good choice.
2. Variables shouldn't map to values, they should map to nodes in a computational tree. There was added complexity in the backward pass due to this imperative style. Treating variables in a functional style is much more natural here.

Next steps:

1. Move to a sparse solver. There are solvers optimized for the large, sparse, but complicated matrices from FEM.
2. Solve a more interesting shape optimization problem (e.g. optimize drag coefficient in Stokes flow.)
3. Re-generate mesh after some time.
4. Add functionality to the language. Newton-Raphson is not supported, but it could be (with automatic Jacobians)
5. Generate optimized code in a fast language.

Thanks!

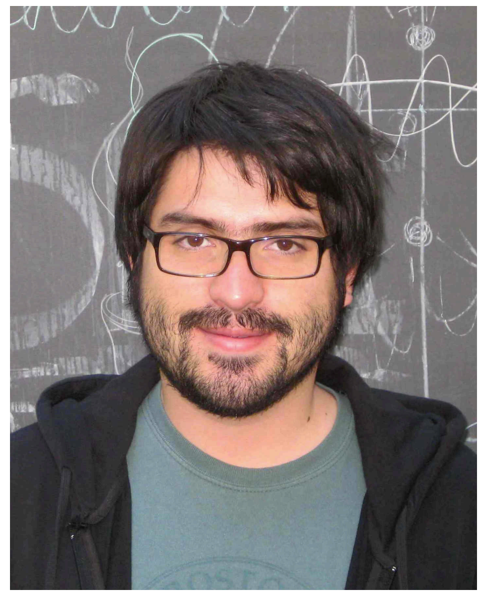


Ian Hunter

Pierre Lermusiaux

Abhinav Gupta

Steven Johnson



Carlos Pérez-Arancibia

Questions?

References

[1] Bijan Mohammadi and Olivier Pironneau. “Shape Optimization in Fluid Mechanics.” Annu. Rev. Fluid Mech. 2004. 36:255-79

[2] Carsten Orthmer. “Adjoint methods for car aerodynamics.” Journal of Mathematics in Industry 2014, 4:6

[3] Steven Johnson. “Notes on Adjoint Methods for 18.335”, Spring 2006, updated Dec. 17, 2012.

[4] Gregoire Allaire. “A review of adjoint methods for sensitivity analysis, uncertainty quantification, and optimization in numerical codes.” Ingenieurs de l’Automobile, SIA, 2015, 836, pp.33-36.

[5] Dougal Maclaurin. “Modeling, Inference and Optimization with Composable Differentiable Procedures.” Thesis, Harvard 2016.

[6] Farrell, Ham, Funke, and Rognes. “Automated derivation of the adjoint of high-level transient finite element programs.” arXiv:1204.5577v2 [cs.MS] 16 Oct 2013.

[7] S.W. Funke and P.E. Ferrell. “A framework for automated PDE-constrained optimisation.” ACM Trans. on Math. Softw. (preprint)

[8] Cristian Homescu. “Adjoints and automatic (algorithmic) differentiation in computational finance” arXiv:1107.1831v1 [q-fin.CP] 10 Jul 2011. NOTE: good general introduction, but there are errors in some technical details.

[9] Mike Giles. “An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation.” Oxford University Computing Lab, 2008.