# Deep Learning with Myia

Olivier Breuleux
Research Developer, MILA

Arnaud Bergeron (MILA)
Bart van Merriënboer (MILA, Google Brain)
Pascal Lamblin (Google Brain)

# The Needs

What we need from a language for deep learning

# Autodiff

What it is, how it works, what the challenges are

# Representation

The best representation for our needs

# Type system

Flexible inference for performance and robustness

# The Needs

What we need from a language for deep learning

## Autodiff

What it is, how it works, what the challenges are

## Representation

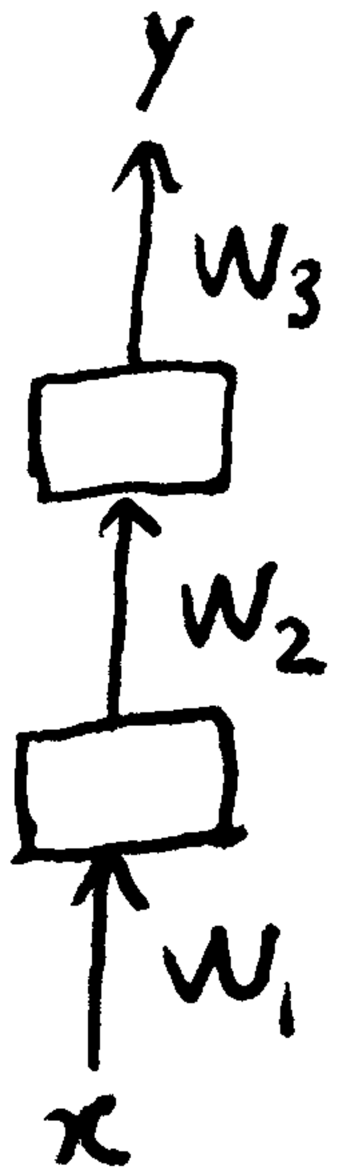
The best representation for our needs

## Type system

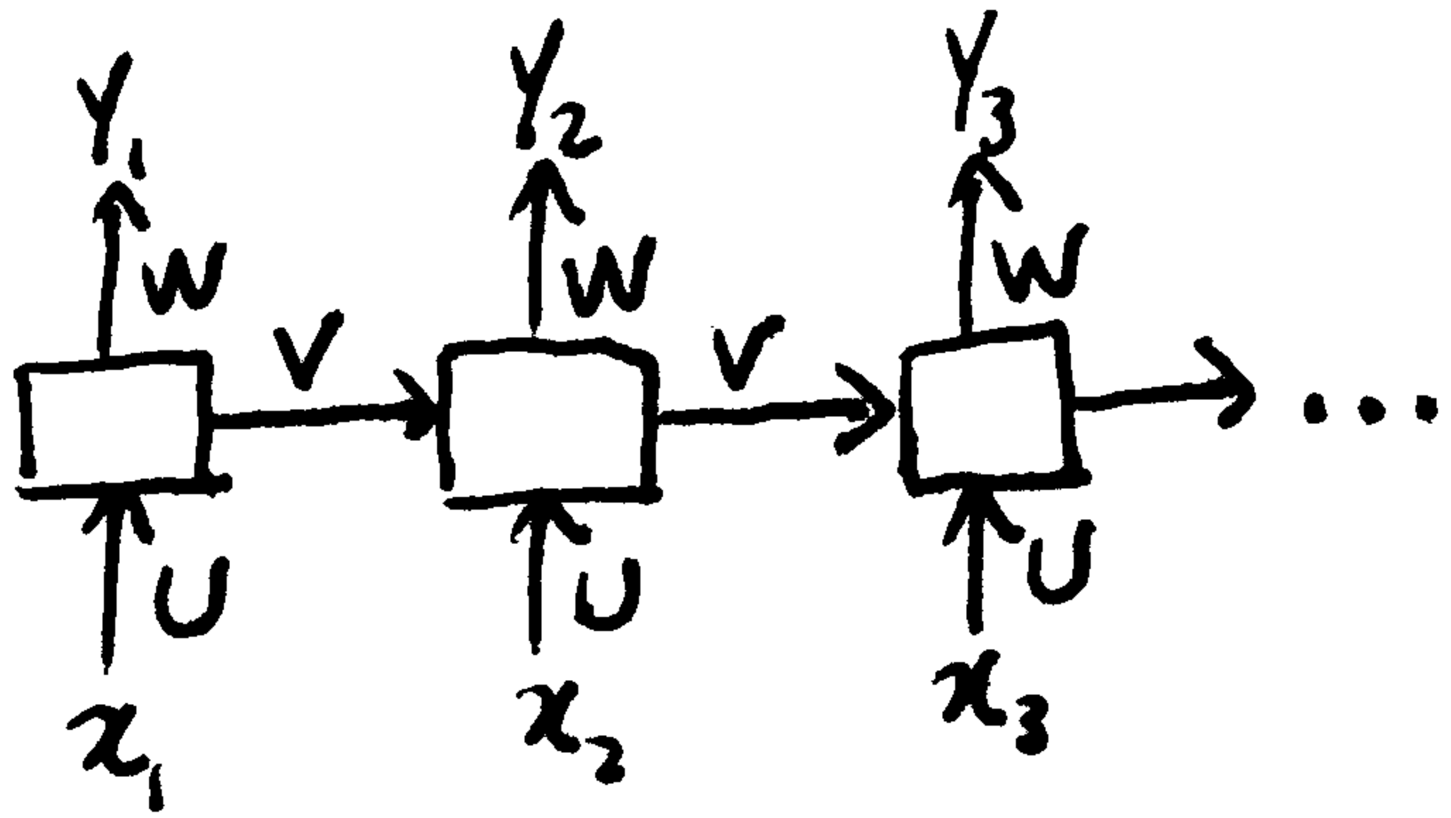
Flexible inference for performance and robustness

# Deep Learning

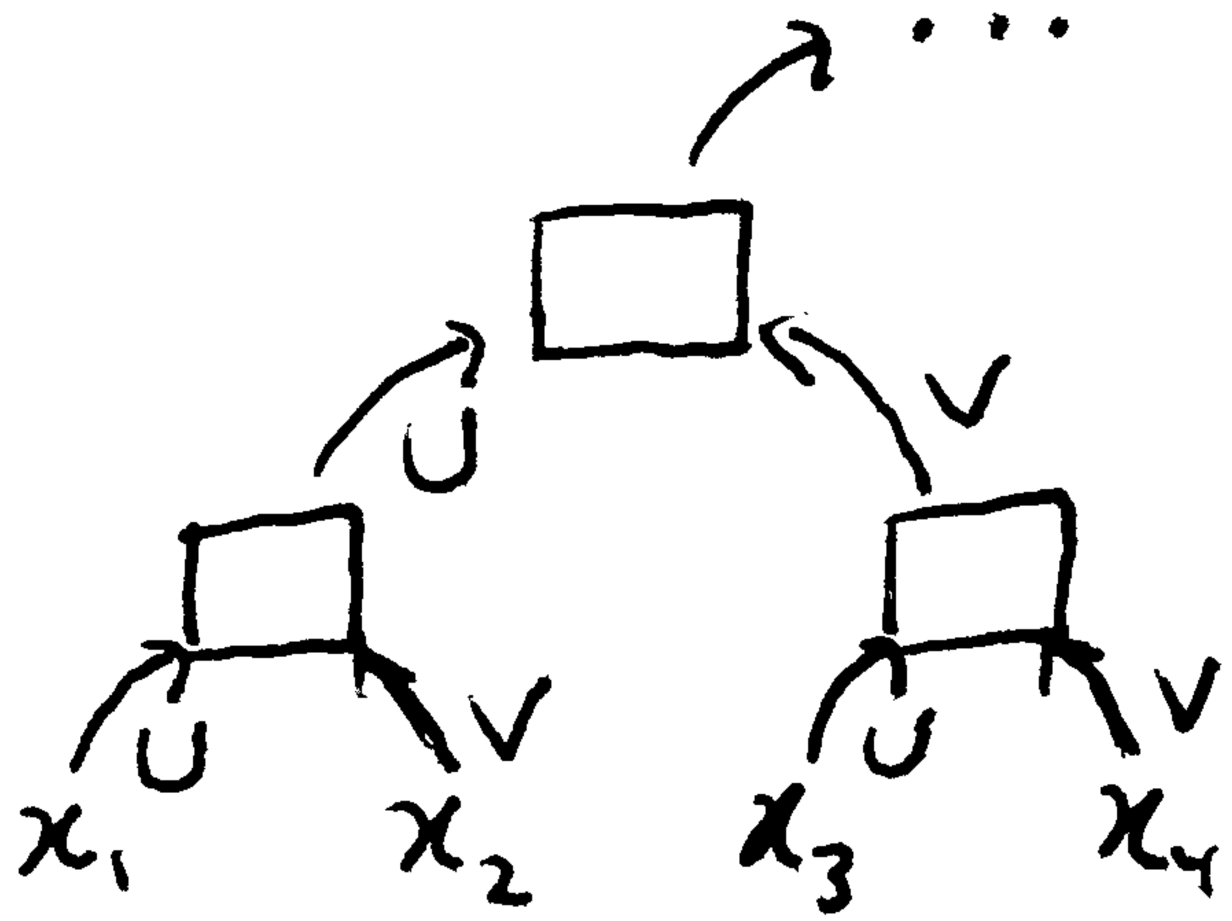
DL algorithms are **increasingly complex**




Feedforward
(trivial)




Recurrent
(loops)




Recursive
(recursion)

...?

# Deep Learning

DL algorithms are **increasingly complex**

- More and more language features needed
- Most existing frameworks are **limited**
- **High level** abstraction increases productivity
  - Focus on the **algorithm** over implementation details
- Effortless abstractions **encourage their use**

# Needs

Mila

**Goal:** a language adapted to the needs of machine learning, past *and* future

**General purpose:** Capable of expressing complex control flow.

**Differentiable:** Should be able to take nth-order derivative of any program.

**Debuggable:** Clear errors, inspectable, instrumentable.

**Fast:** Must leverage parallelism and GPU.

**Portable:** Serializable, support multiple hardware.

# Needs

**Myia:** a language adapted to the needs of machine learning, past *and* future

**General purpose:** Conditionals, loops, recursion, data structures.

**Differentiable:** Transformation at the intermediate representation level.

**Debuggable:** Type+shape inference, step debugger.

**Fast & portable:** Choose from various backends such as NNVM/Relay.

## The Needs

What we need from a language for deep learning

## Autodiff

What it is, how it works, what the challenges are

## Representation

The best representation for our needs

## Type system

Flexible inference for performance and robustness

# Differentiability

**How to train a model**

- Initialize a model's parameters $\theta$
- Compute some quantity using the parameters $f(x;\theta)$
- Compute a cost or "loss function" $L(f(x;\theta), y)$
- Update parameters using the *gradient of the loss* $\theta \leftarrow \theta - \lambda \frac{\partial L(f(x;\theta), y)}{\partial \theta}$
- Rinse and repeat

**Gradients**

- Can be computed exactly and automatically
- But: no mainstream language supports this natively
- **Computational strategies**: forward or reverse
- **Implementation strategies**: operator overloading or source transform

# Forward vs Reverse

$$y_1 = f(x)$$
$$y_2 = g(y_1)$$
$$y_3 = h(y_2)$$

$$f : \mathbb{R}_m \to \mathbb{R}_p$$
$$g : \mathbb{R}_p \to \mathbb{R}_q$$
$$h : \mathbb{R}_q \to \mathbb{R}_n$$

The derivative of a straight composition of functions is **the multiplication of their Jacobians**

$$\underbrace{\mathbf{J}_{\mathbf{h}\circ\mathbf{g}\circ\mathbf{f}}(\mathbf{x})}_{n\times m} = \underbrace{\mathbf{J}_{\mathbf{h}}(\mathbf{y_2})}_{n\times q} \; \underbrace{\mathbf{J}_{\mathbf{g}}(\mathbf{y_1})}_{q\times p} \; \underbrace{\mathbf{J}_{\mathbf{f}}(\mathbf{x})}_{p\times m}$$

**In what order?**

# Forward vs Reverse

## Forward

$$\underbrace{\mathbf{J_h}(\mathbf{y_2})}_{n\times q}\ \textcolor{red}{(}\underbrace{\underbrace{\mathbf{J_g}(\mathbf{y_1})}_{q\times p}\ \underbrace{\mathbf{J_f}(\mathbf{x})}_{p\times m}}_{q\times m}\textcolor{red}{)}$$

## Cost

$$qpm + nqm$$
$$= \textcolor{blue}{m}(qp + nq)$$

## Reverse

$$\underbrace{\textcolor{red}{(}\underbrace{\mathbf{J_h}(\mathbf{y_2})}_{n\times q}\ \underbrace{\mathbf{J_g}(\mathbf{y_1})}_{q\times p}\textcolor{red}{)}}_{n\times p}\ \underbrace{\mathbf{J_f}(\mathbf{x})}_{p\times m}$$

## Cost

$$nqp + npm$$
$$= \textcolor{green}{n}(qp + pm)$$

# Forward vs Reverse

**Forward mode** is good when there are **few inputs**.

- **Easy to implement: dual numbers.**

$$x \to \left(y_1, \frac{dy_1}{dx}\right) \to \left(y_2, \frac{dy_2}{dx}\right) \to \left(y_3, \frac{dy_3}{dx}\right)$$

**Reverse mode** is good when there are **few outputs**.

- **Hard to implement: execution is reversed.**

$$x \to y_1 \to y_2 \to y_3 \to \frac{dy_3}{dy_2} \to \frac{dy_3}{dy_1} \to \frac{dy_3}{dx}$$

# Forward vs Reverse

**Deep learning** involves computing the gradient of **millions of parameters** with respect to **a loss**.

$$\theta \leftarrow \theta - \epsilon \frac{\partial \mathcal{L}}{\partial \theta}$$

$$\text{where } \theta = (\mathbf{W_1}, \mathbf{W_2}, \ldots, \mathbf{b_1}, \mathbf{b_2}, \ldots)$$

We need **reverse mode**.

# OO vs SCT: Operator Overloading

```python
def f(x):
    i = 0
    while i < 3:
        i = i + 1
        x = tanh(x)
    x = x * 10
    return x
```

**Program**

**Trace**

```python
i = 0
i = i + 1
x = tanh(x)
i = i + 1
x = tanh(x)
i = i + 1
x = tanh(x)
x = x * 10
```

**Tape**

**Backprop**

- Overload every operation to log itself on a **tape**.
- At the end, we walk the tape backward.
- “Define-by-run”, “Dynamic graph”
- **Easy to implement**, but **lots of overhead**
    - Discourages composing small & cheap operations

# OO vs SCT: Source Code Transformation

Mila

- Transform a **function** that computes a value into a **new function** that computes the derivative.
  - Operate on source code or intermediate representation
  - Applies the chain rule to code

- Standard language optimizations apply: can eliminate overhead

- Easier to apply to functional languages
  - Reverse mode AD interacts badly with mutation and side effects
  - Requires deep analysis and optimization to remove dead code

```
def bprop_pow(x, y, out, dout):
    dx = dout * y * x ** (y - 1)
    dy = dout * out * log(x)
    return dx, dy
```

What if we don't need dy?

# The Needs

What we need from a language for deep learning

# Autodiff

What it is, how it works, what the challenges are

# Representation

The best representation for our needs

# Type system

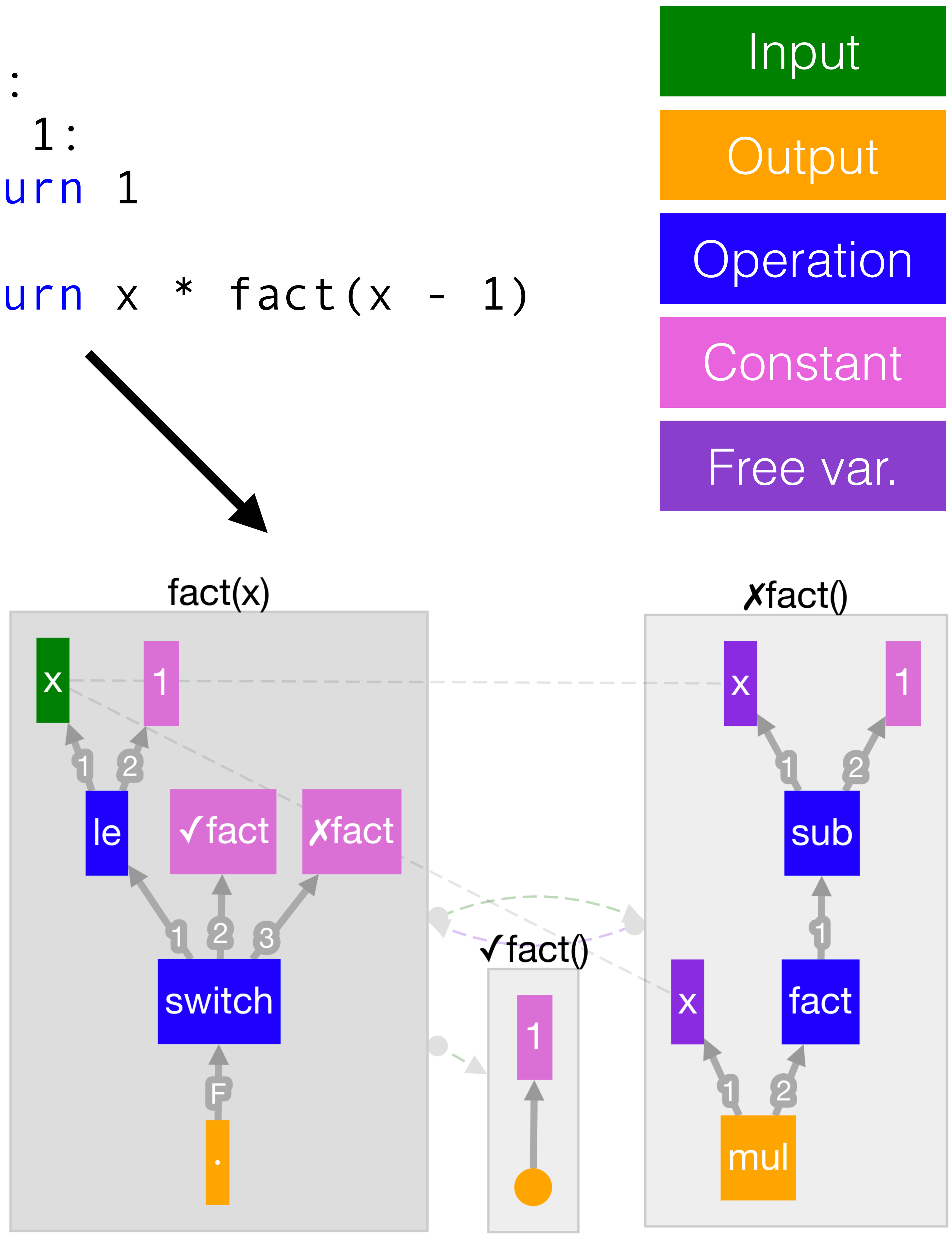
Flexible inference for performance and robustness

# About syntax

**Myia is an intermediate representation**

- High level
- No syntax of its own
- Multiple languages may target it

```python
def fact(x):
    if x <= 1:
        return 1
    else:
        return x * fact(x - 1)
```



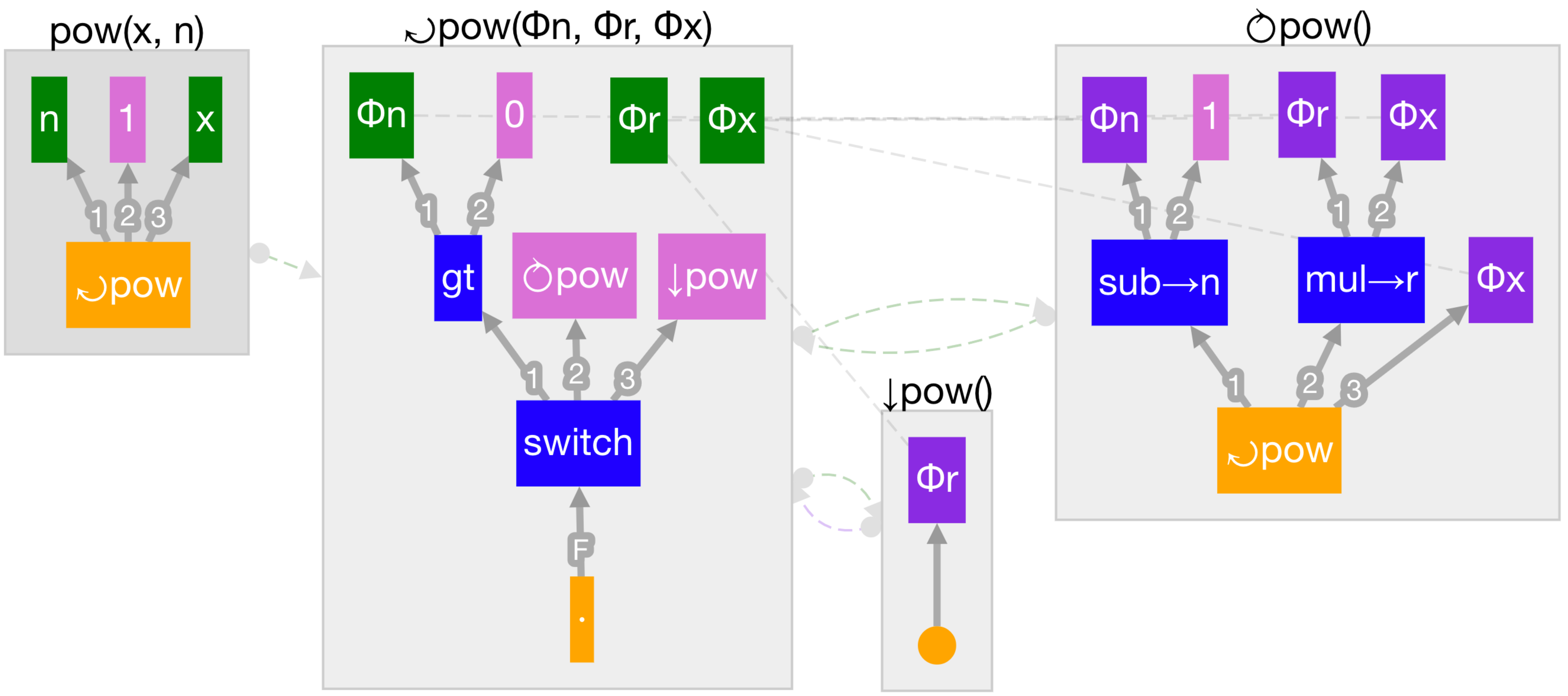

**Python frontend**

- Why? Most used language in DL
- Productive for research and prototyping
- Translate *functional subset* to Myia
  - **Control flow**: `if`, `while`, `for`, `def`, `lambda`
  - **Data**: lists, tuples, arrays, `@dataclass`
  - **Not supported**: mutation, side effects, `eval`
- One issue: translate dynamically typed code

# Needs

**Requirements for our representation**

- Powerful enough to represent recursion
- Minimal
- Easy to parallelize
- Easy to optimize
- Easy to extend




**Solutions**

- Functional (ANF)
- Graph-based
- Typed

# Why functional programming?

**Easier to transform**

- Referential transparency: same expression, same result

**Easier to think about**

- No side effects

**Easier to optimize**

- Order of operations can be changed
- Parallelizable
- Common subexpression elimination easy

**Type system is easier**

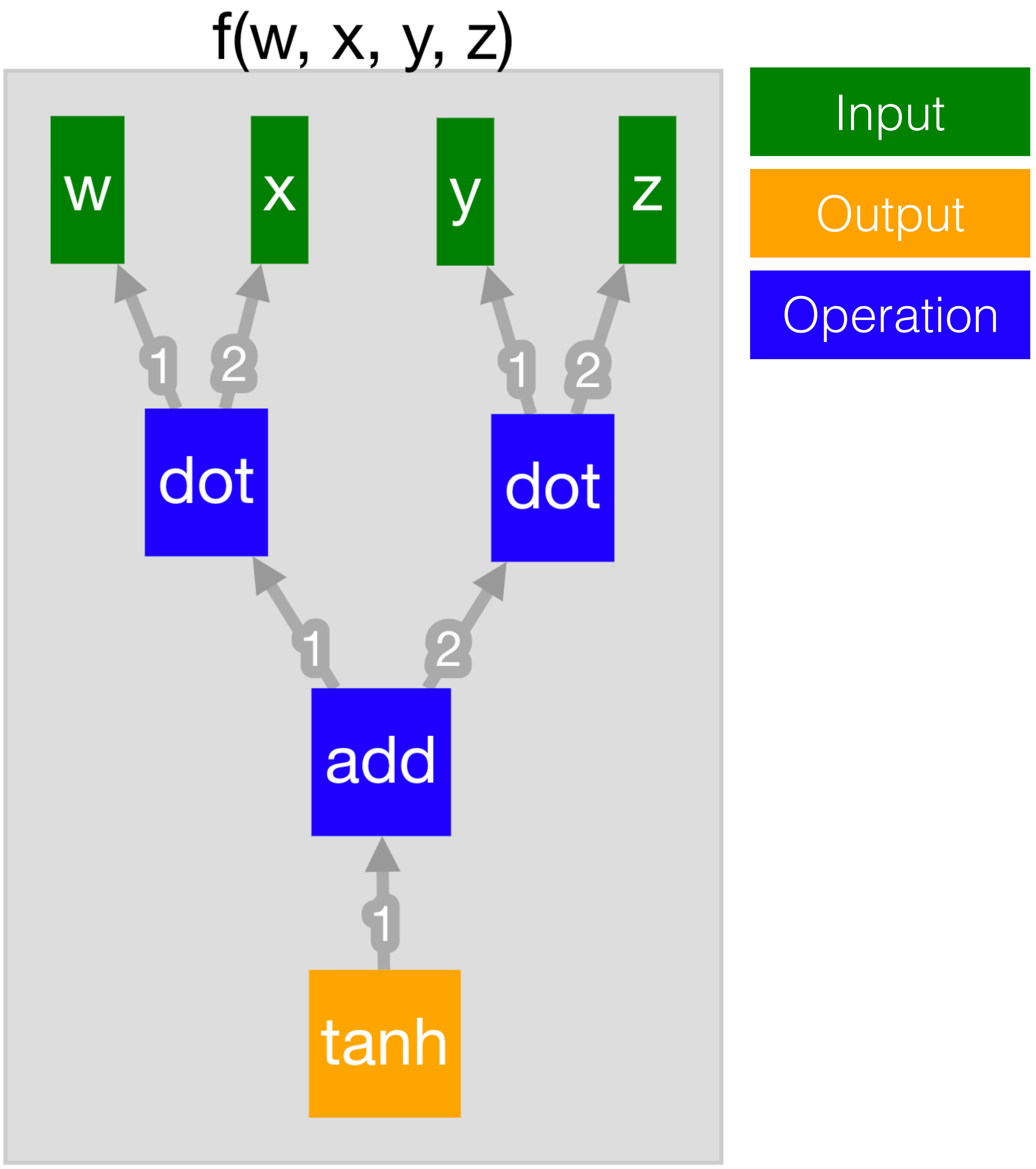
- No side effects

**Easier for automatic differentiation**

# Why graphs?

**Easy to parallelize**
- Only data flow relationships

**Easy to optimize**
- Direct use-def pointers (no names)
- Dead code elimination is trivial
- Inlining is easy



# Why static typing?

Mila

**Guarantees**

- Correctness of the user's program
- Type correctness of code transforms (autodiff)

**Performance**

- No runtime type checking = better performance
- Leverage shape information for optimization

**User experience**

- Prevent errors late in process

```
mlp.py:85
in step(
    model :: Model(
        layers :: (
            TanhLayer(W :: f64 x 10 x 12, b :: f64 x 1 x 12),
            TanhLayer(W :: f64 x 14 x 1, b :: f64 x 1 x 1)
        )
    ),
    x :: f64 x 3 x 10,
    y :: i8 x 3 x 1
)
85: dmodel = grad(cost)(model, x, y)
==========================================================================================
mlp.py:75
in cost(
    model :: Model(
        layers :: (
            TanhLayer(W :: f64 x 10 x 12, b :: f64 x 1 x 12),
            TanhLayer(W :: f64 x 14 x 1, b :: f64 x 1 x 1)
        )
    ),
    x :: f64 x 3 x 10,
    target :: i8 x 3 x 1
)
75: y = model.apply(x)
==========================================================================================
mlp.py:49
in apply(
    self :: Model(
        layers :: (
            TanhLayer(W :: f64 x 10 x 12, b :: f64 x 1 x 12),
            TanhLayer(W :: f64 x 14 x 1, b :: f64 x 1 x 1)
        )
    ),
    x :: f64 x 3 x 10
)
49: x = layer.apply(x)
==========================================================================================
mlp.py:39
in apply(
    self :: TanhLayer(W :: f64 x 14 x 1, b :: f64 x 1 x 1),
    input :: f64 x 3 x 12
)
39: return tanh(input @ self.W + self.b)
==========================================================================================
in dot(f64 x 3 x 12, f64 x 14 x 1)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
MyiaShapeError: Incompatible shapes in dot: (3, 12) and (14, 1)
```

# The Needs

What we need from a language for deep learning

# Autodiff

What it is, how it works, what the challenges are

# Representation

The best representation for our needs

# Type system

Flexible inference for performance and robustness

# Myia’s Types

**Scalars:** `Int/UInt/Float<8/16/32/64>, Bool`
**Tuples:** `Tuple<T1, T2, …>`

- Heterogeneously typed, static length

**Lists:** `List<T>`

- Homogeneously typed, dynamic length, fast append

**Arrays:** `Array<T, Shape<D1, D2, …>>`

- Homogeneously typed, shape part of the type

**Functions:** `Function<Args<TIn1, TIn2, ... >, TOut>`

Struct types are reduced to tuples in pre-processing.

# Why inference?

Mila

**Annotations are annoying**

- Polymorphic types are awkward to express
- Function types are awkward to express
- Impede rapid prototyping
- Duck typing is more natural
- This is why people like Python

**Type/shape inference**

- Infer from the input types from entry point
- Implicit polymorphism
- Feels dynamic
- Functions are re-compiled when they are given new input types

# Myia’s inference pipeline

1. **Transform inputs into abstract inputs**
   - Represent type and shape — no concrete values
   - More types: structs, polymorphic functions

2. **Run abstract interpreter on abstract inputs**
   - Bounded input signatures for each function
   - Recursive functions become fixed points

3. **Specialize functions to their possible signatures**
   - If function called with int, make int version, etc.
   - Higher order uses require signature uniqueness

4. **Update or re-run inference after optimizations or AD**

# Error reporting

Abstract inferrer shows compile-time tracebacks for type/shape errors.

```
mlp.py:85
in step(
    model :: Model(
        layers :: (
            TanhLayer(W :: f64 x 10 x 12, b :: f64 x 1 x 12),
            TanhLayer(W :: f64 x 14 x 1, b :: f64 x 1 x 1)
        )
    ),
    x :: f64 x 3 x 10,
    y :: i8 x 3 x 1
)
85: dmodel = grad(cost)(model, x, y)
================================================================================
mlp.py:75
in cost(
    model :: Model(
        layers :: (
            TanhLayer(W :: f64 x 10 x 12, b :: f64 x 1 x 12),
            TanhLayer(W :: f64 x 14 x 1, b :: f64 x 1 x 1)
        )
    ),
    x :: f64 x 3 x 10,
    target :: i8 x 3 x 1
)
75: y = model.apply(x)
```

```
================================================================================
mlp.py:49
in apply(
    self :: Model(
        layers :: (
            TanhLayer(W :: f64 x 10 x 12, b :: f64 x 1 x 12),
            TanhLayer(W :: f64 x 14 x 1, b :: f64 x 1 x 1)
        )
    ),
    x :: f64 x 3 x 10
)
49: x = layer.apply(x)
================================================================================
mlp.py:39
in apply(
    self :: TanhLayer(W :: f64 x 14 x 1, b :: f64 x 1 x 1),
    input :: f64 x 3 x 12
)
39: return tanh(input @ self.W + self.b)
================================================================================
in dot(f64 x 3 x 12, f64 x 14 x 1)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
MyiaShapeError: Incompatible shapes in dot: (3, 12) and (14, 1)
```

# Debugging

**Tracking correspondence to source code**

- Through parsing
- Through optimization
- Through automatic differentiation
- Through macros/code generation

**Debugging tools we need**

- Custom debugger for step by step execution
- Watching variables and gradients
- Breakpoints that trigger during the reverse phase
- Profiling and reporting which parts of the code are “hot”

# In Conclusion: Myia's focus

**General purpose, including recursion**

**Automatic differentiation**

- Code transform
- Optimizable, higher order gradients

**Type and shape inference**

- Can handle duck typed code

**Good debugging facilities**

- Step debugger, profiling
- Gradient debugging

⭐ **us on GitHub:** https://github.com/mila-iqia/myia