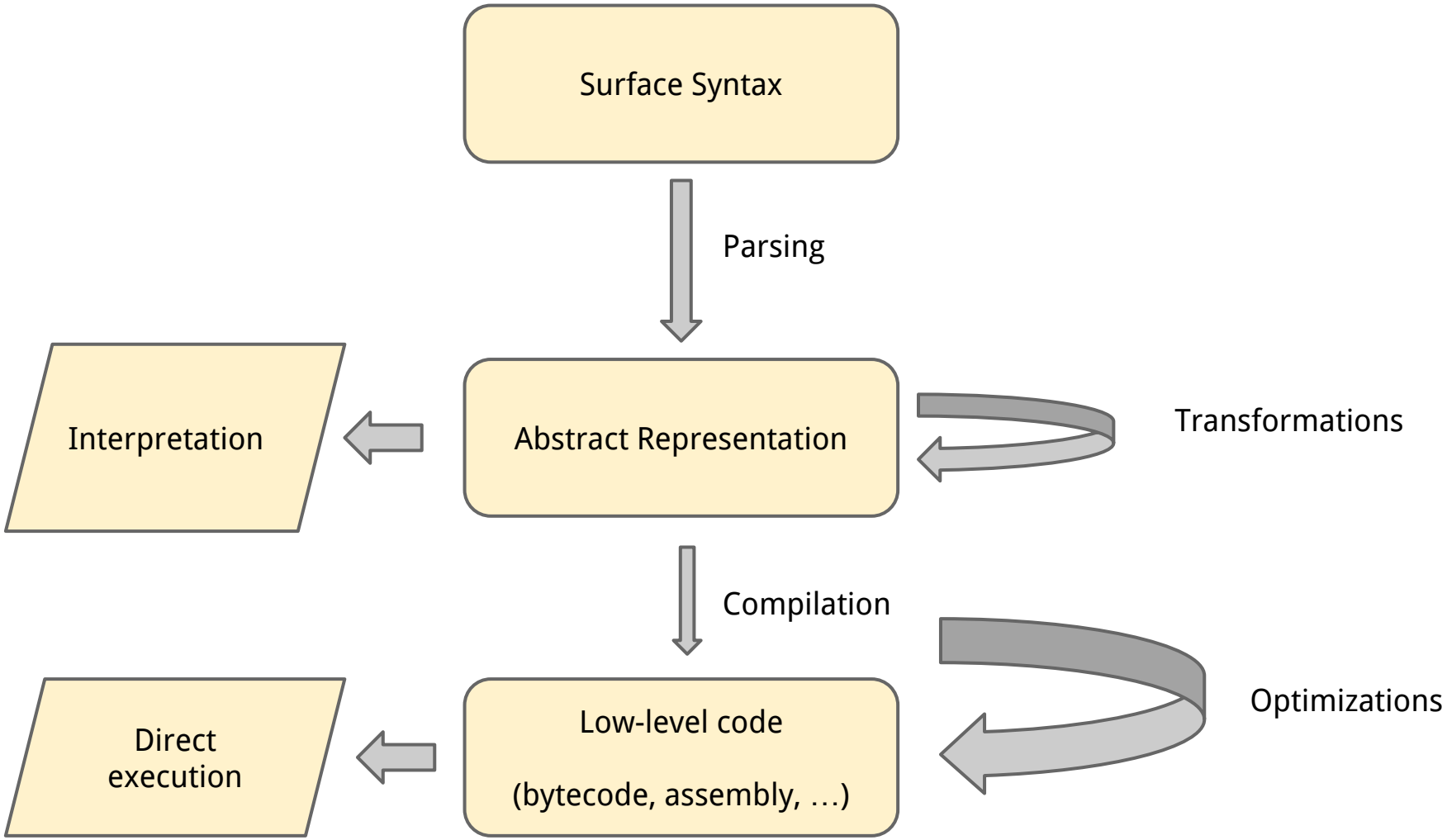
Introduction to Interpretation

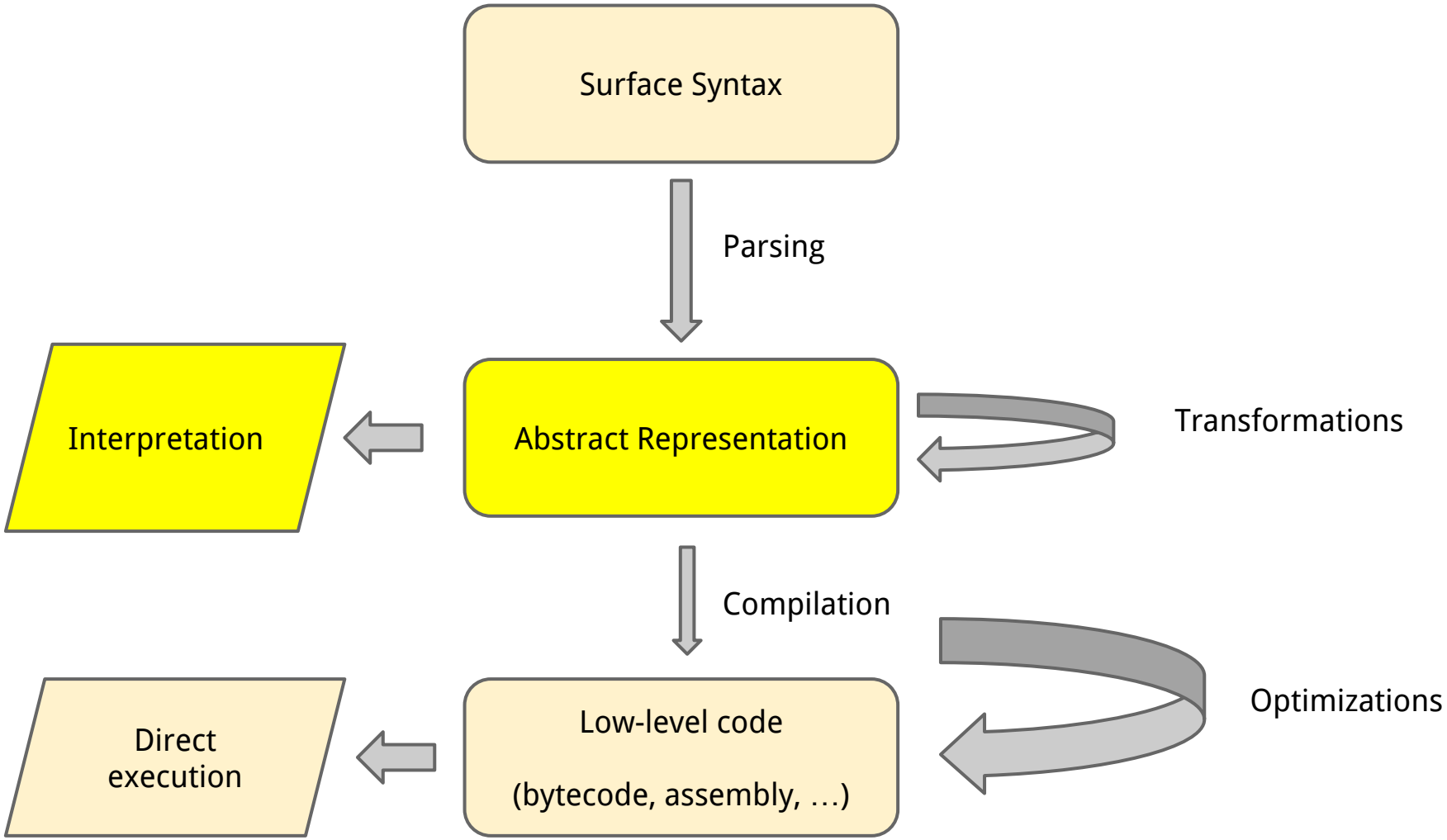
September 6, 2016

Riccardo Pucella

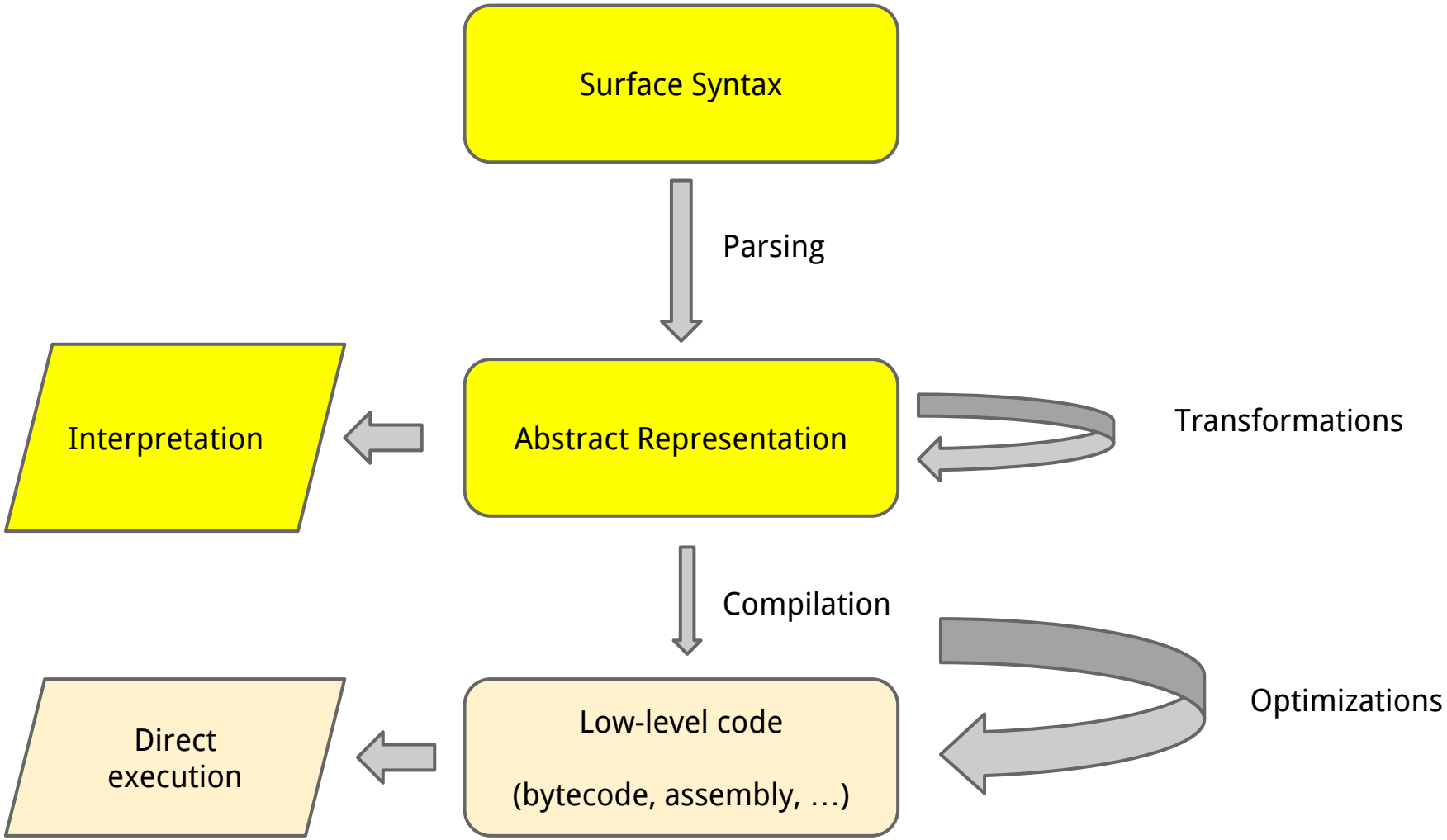
The structure of language execution



The structure of language execution



The structure of language execution



A simple expression language

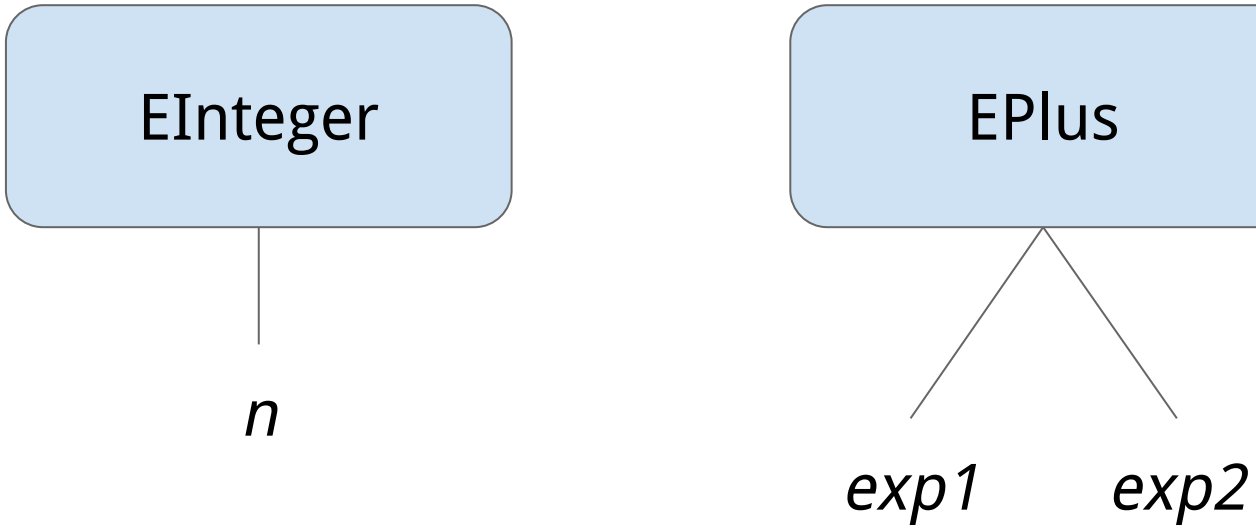
- We’re going to build-up a small language of mathematical expressions

- Computing over the integers
  - Operations +, -, *

- Abstract representation needs to account for the fact that expressions can be nested
  - E.g., (3 + 4) * (5 + 6)
  - I'm often going to use *prefix notation*
    (* (+ 3 4) (+ 5 6))

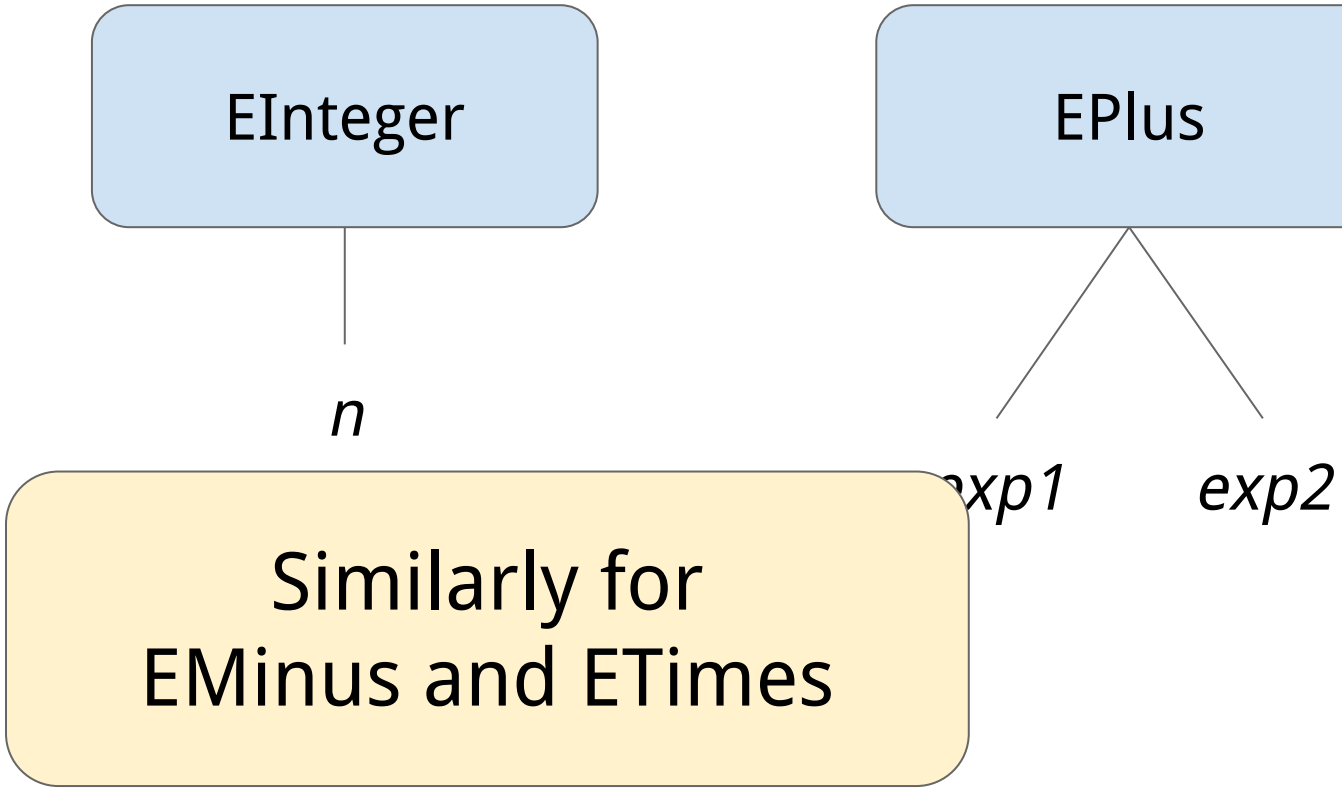
Abstract representation

An expression is a tree. Nodes are kinds of expressions:

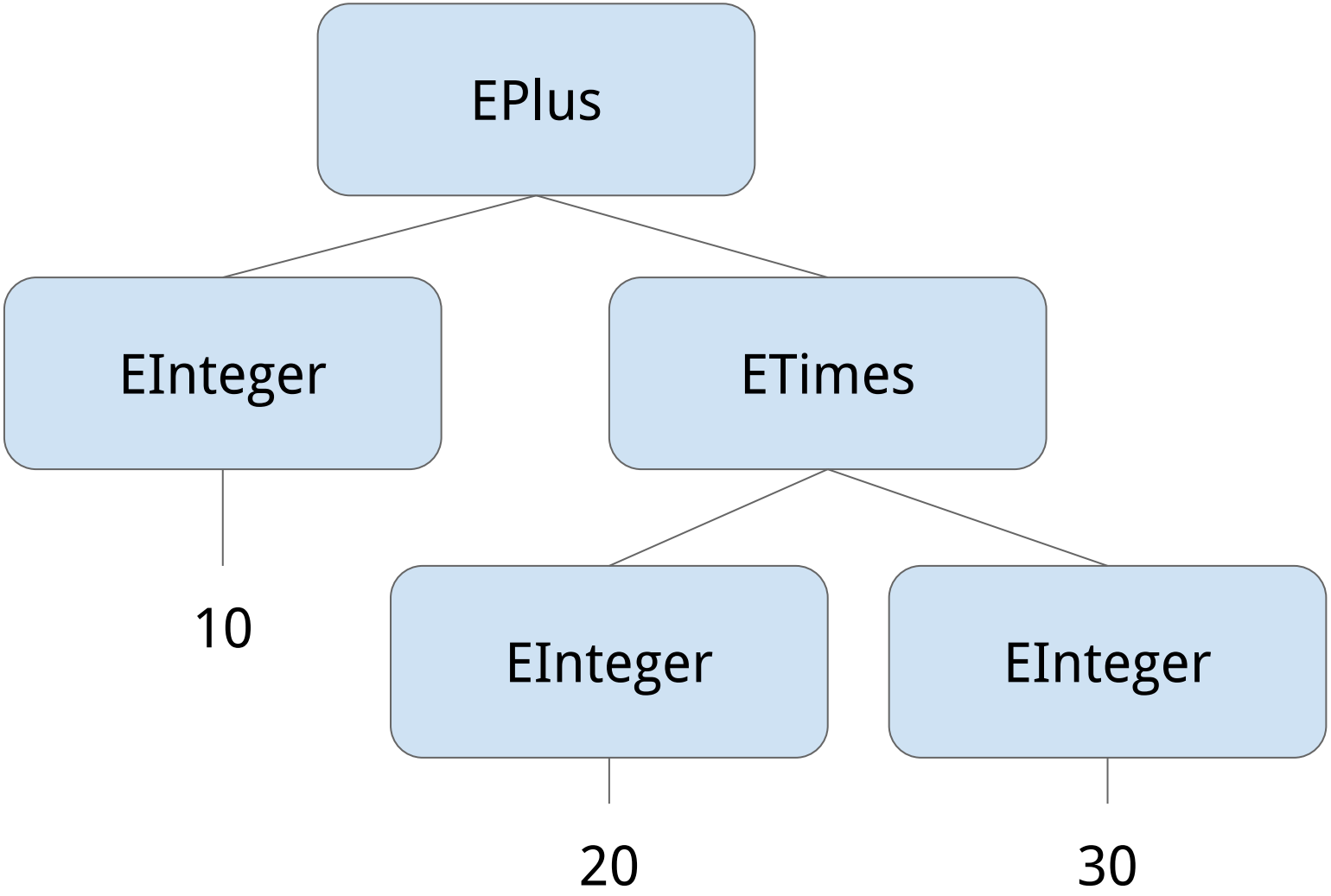


Abstract representation

An expression is a tree. Nodes are kinds of expressions:



Example: (+ 10 (* 20 30))



Abstract Representation in Python

```
class Exp (object):
  pass

class EInteger (Exp):
    def __init__ (self,i):
        self._integer = i

class EPlus (Exp):
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

# Also: EMinus, ETimes
```

Abstract Representation in Python

```
class Exp (object):
  pass

class EInt
    def _
        s

class EPl
    def _
        s
        s

# Also: E
```

Constructing our example:

```
EPlus(EInteger(10),
      ETimes(EInteger(20),
             EInteger(30)))
```

Evaluation

Evaluation is the process of taking an expression and reducing it to a value

- aka *execution*

Every node in the abstract representation has an evaluation method that evaluates the expression to a value

- evaluating an expression generally requires recursively evaluating subexpressions

Semantics?

Evaluation for integer literals

```
class EInteger (Exp):
    def __init__ (self,i):
        self._integer = i

    def eval (self):
        return self._integer
```

Evaluation for addition

```
class EPlus (Exp):
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

    def eval (self):
        return self._exp1.eval()
                + self._exp2.eval()
```

Evaluation for subtraction

```
class EMinus (Exp):
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

    def eval (self):
        return self._exp1.eval()
                - self._exp2.eval()
```

Evaluation for multiplication

```
class ETimes (Exp):
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

    def eval (self):
        return self._exp1.eval()
                * self._exp2.eval()
```

Demo

Booleans and conditionals

Let's add a new type of value: Booleans

- `true`, `false`
- need to extend the class of values

Booleans support a bunch of operations

- The most important is probably the conditional expression:
  (**`if`** *`cond then-part else-part`*)

Value class

```
class Value (object):
    pass

class VInteger (Value):
    def __init__ (self,i):
        self.value = i
        self.type = "integer"

class VBoolean (Value):
    def __init__ (self,b):
        self.value = b
        self.type = "boolean"
```

Evaluation for literals

```
class EInteger (Exp):
    def __init__ (self,i):
        self._integer = i

    def eval (self):
        return VInteger(self._integer)

class EBoolean (Exp):
    def __init__ (self,b):
        self._boolean = b

    def eval (self):
        return VBoolean(self._boolean)
```

Evaluation for addition

```
class EPlus (Exp):
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

    def eval (self):
        v1 = self._exp1.eval()
        v2 = self._exp2.eval()
        if v1.type == "integer" and v2.type == "integer":
            return VInteger(v1.value + v2.value)
        raise Exception ("Error: adding non-numbers")
```

Evaluation for addition

```
class EPlus (Exp):
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

    def eval (self):
        v1 = self._exp1.eval()
        v2 = self._exp2.eval()
        if v1.type == "integer" and v2.type == "integer":
            return VInteger(v1.value + v2.value)
        raise Exception ("Error: adding non-numbers")
```

Entirely similar for `EMinus` and `ETimes`

Evaluation for conditional

```
class EIf (Exp):
    def __init__ (self,e1,e2,e3):
        self._cond = e1
        self._then = e2
        self._else = e3

    def eval (self):
        v = self._cond.eval()
        if v.type != "boolean":
            raise Exception ("Error: non-Boolean condition")
        if v.value:
            return self._then.eval()
        else:
            return self._else.eval()
```

First homework

- Add Boolean operators AND, OR, NOT
- Add vectors of values
- Add rational numbers and division