Project 4: A Symbolic Calculator

Due Wednesday 3pm next week

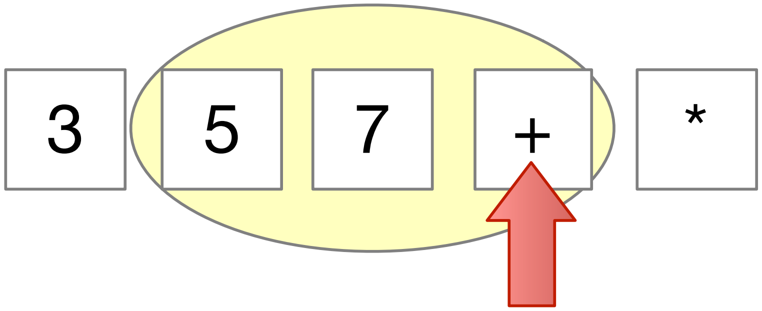
Postfix ("reverse Polish") notation:

```
$ python3 calc.py

expression/'help'/'quit': 3 5 +
(3 + 5) -> 8
expression/'help'/'quit': 3 5 7 + *
(3 * (5 + 7)) -> 36
expression/'help'/'quit':
```

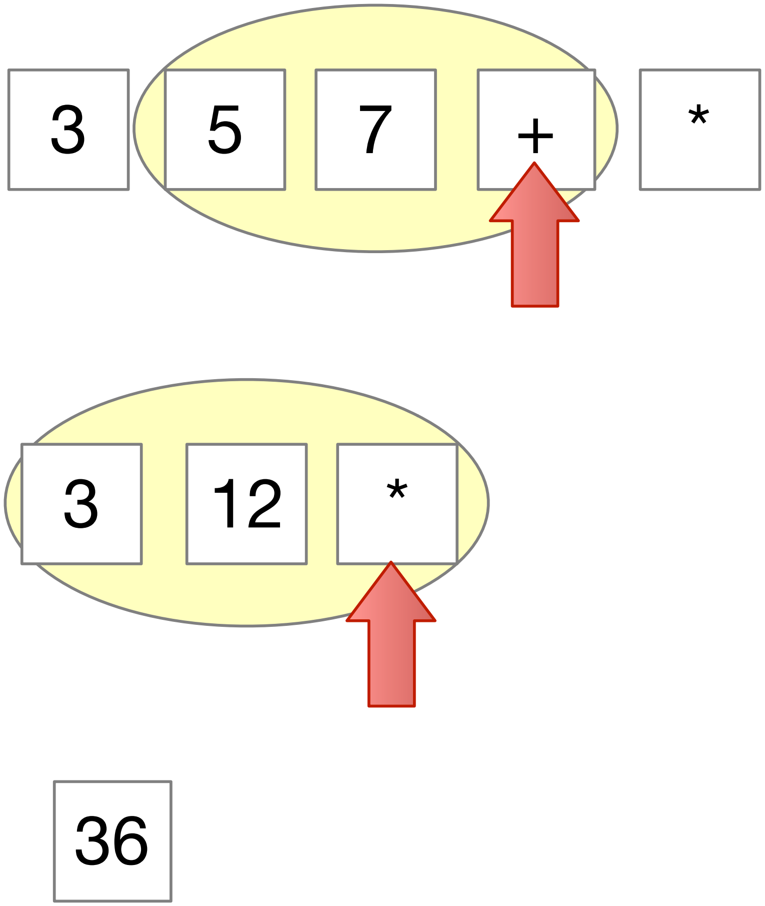
Why postfix?

- Very easy to parse
  - Much simpler than algebraic notation
  - No need for parentheses or precedence



Operator applies
immediately
preceding
operands

```
expression/'help'/'quit': 3 5 7 + *
(3 * (5 + 7)) -> 36
```



3 5 7 + *

3 12 *

36

With memory, of course ...

```
expression/'help'/'quit': x 7 =
let x = 7 -> 7
expression/'help'/'quit': x 3 *
(x * 3) -> 21
expression/'help'/'quit':
```

But wait … there’s more!

```
$ python3 calc.py
expression/'help'/'quit': y 7 x * =
let y = (7 * x) -> (7 * x)
expression/'help'/'quit': x 3 =
let x = 3 -> 3
expression/'help'/'quit': y
y -> 21
expression/'help'/'quit':
```

Calculator evaluates as far as possible

```
expression/'help'/'quit':  a 3 7 + y 4 + * =
let a = ((3 + 7) * (y + 4)) -> (10 * (y + 4))
expression/'help'/'quit': a
a -> (10 * (y + 4))
expression/'help'/'quit': y 3 =
let y = 3 -> 3
expression/'help'/'quit': a
a -> 70
```

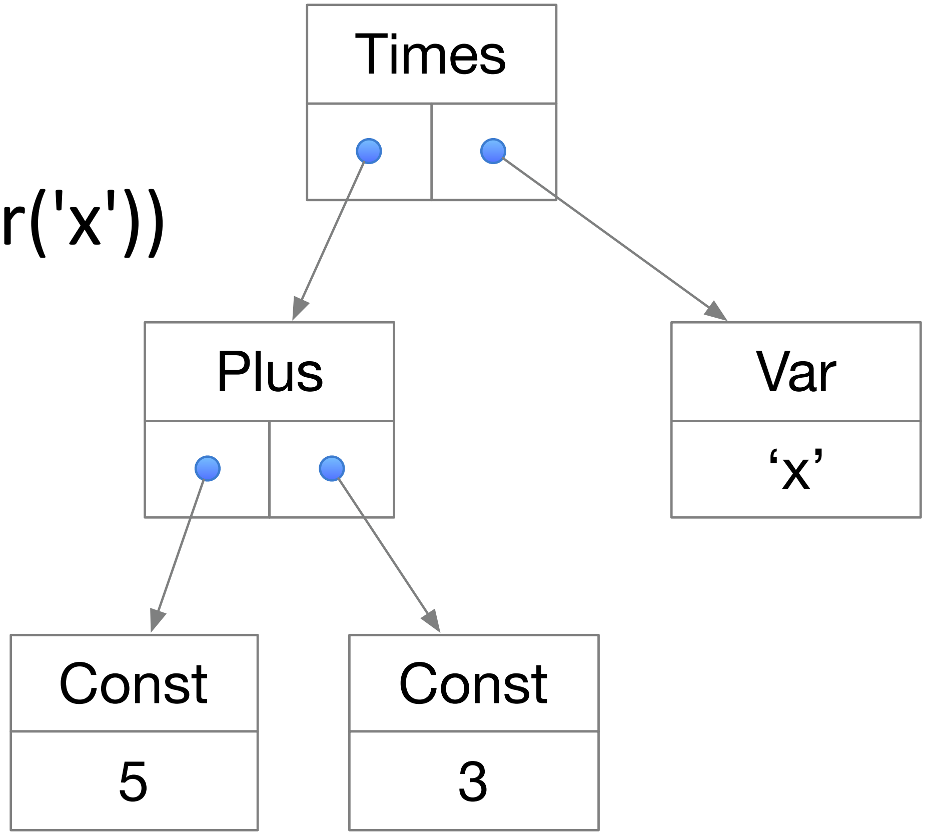
But doesn’t that mean ...
we could have a problem?!

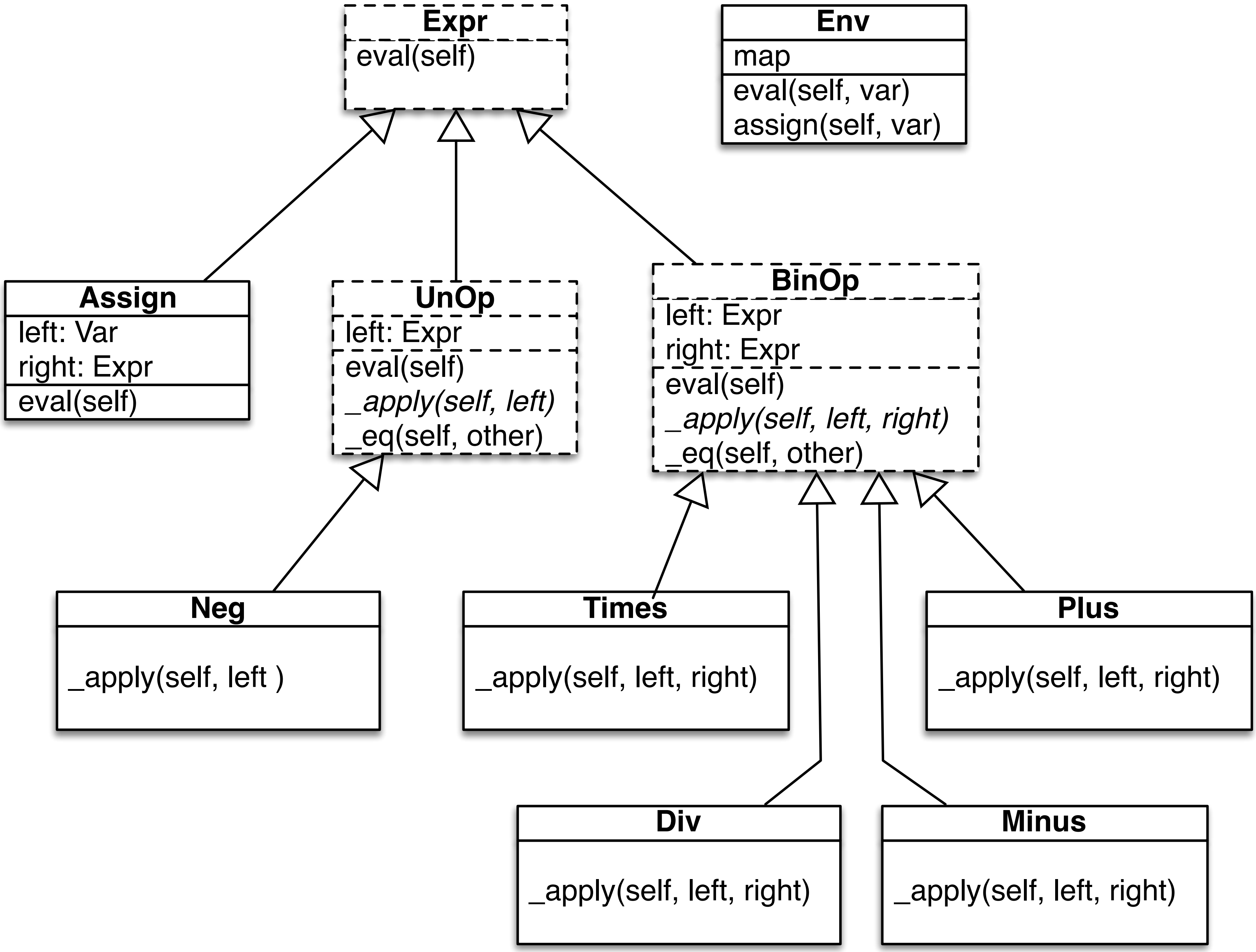
```
expression/'help'/'quit':  a  5  m  +  =
let  a  =  (5  +  m)  ->  (5  +  m)
expression/'help'/'quit':  m  5  a  +  =
let  m  =  (5  +  a)  ->  (5  +  (5  +  m))
expression/'help'/'quit':  r  m  5  +  =
WARNING:expr:Cyclic  reference to m?    Bailing.
let  r  =  (m  +  5)  ->  ((5  +  (5  +  (5  +  (5  +  m))))  +  5)
```

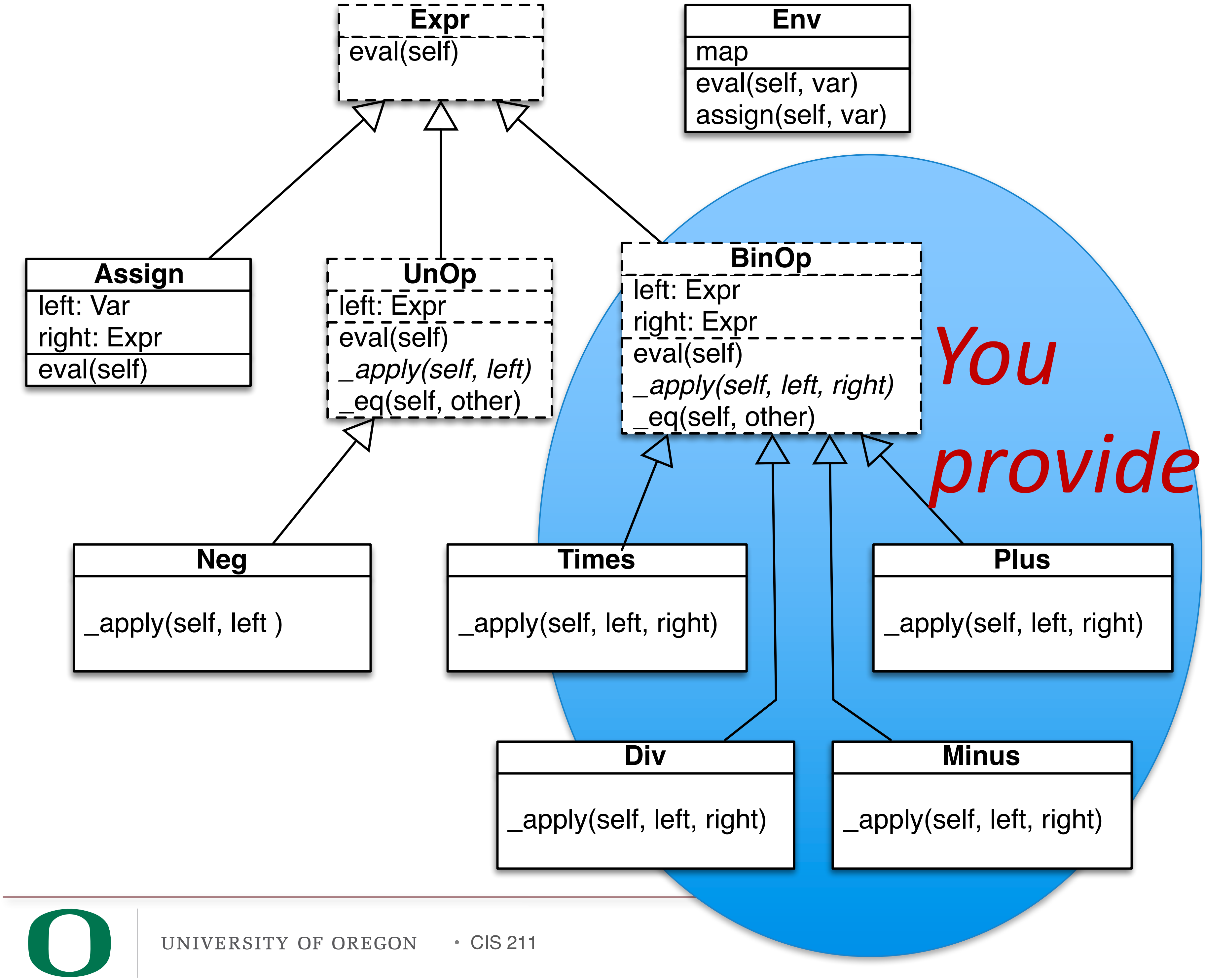
How does it work?

We will have a class for each operation (+, *, =, etc) and classes for constants and variables

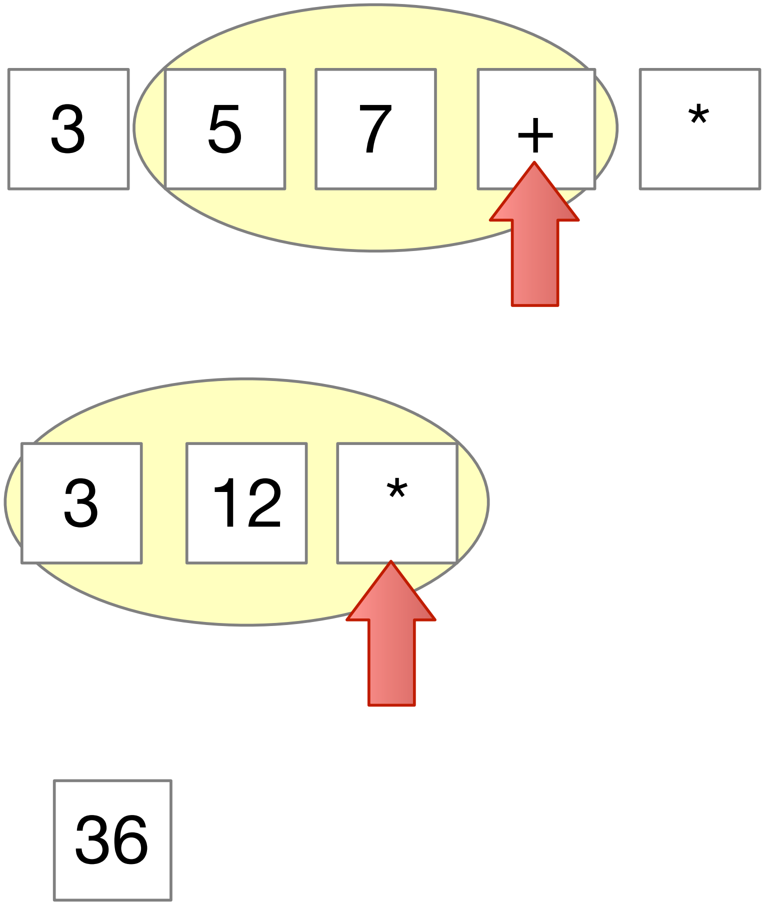
```
>>> rpn_parse.parse("5 3 + x *")
Times(Plus(Const(5),Const(3)),Var('x'))
```



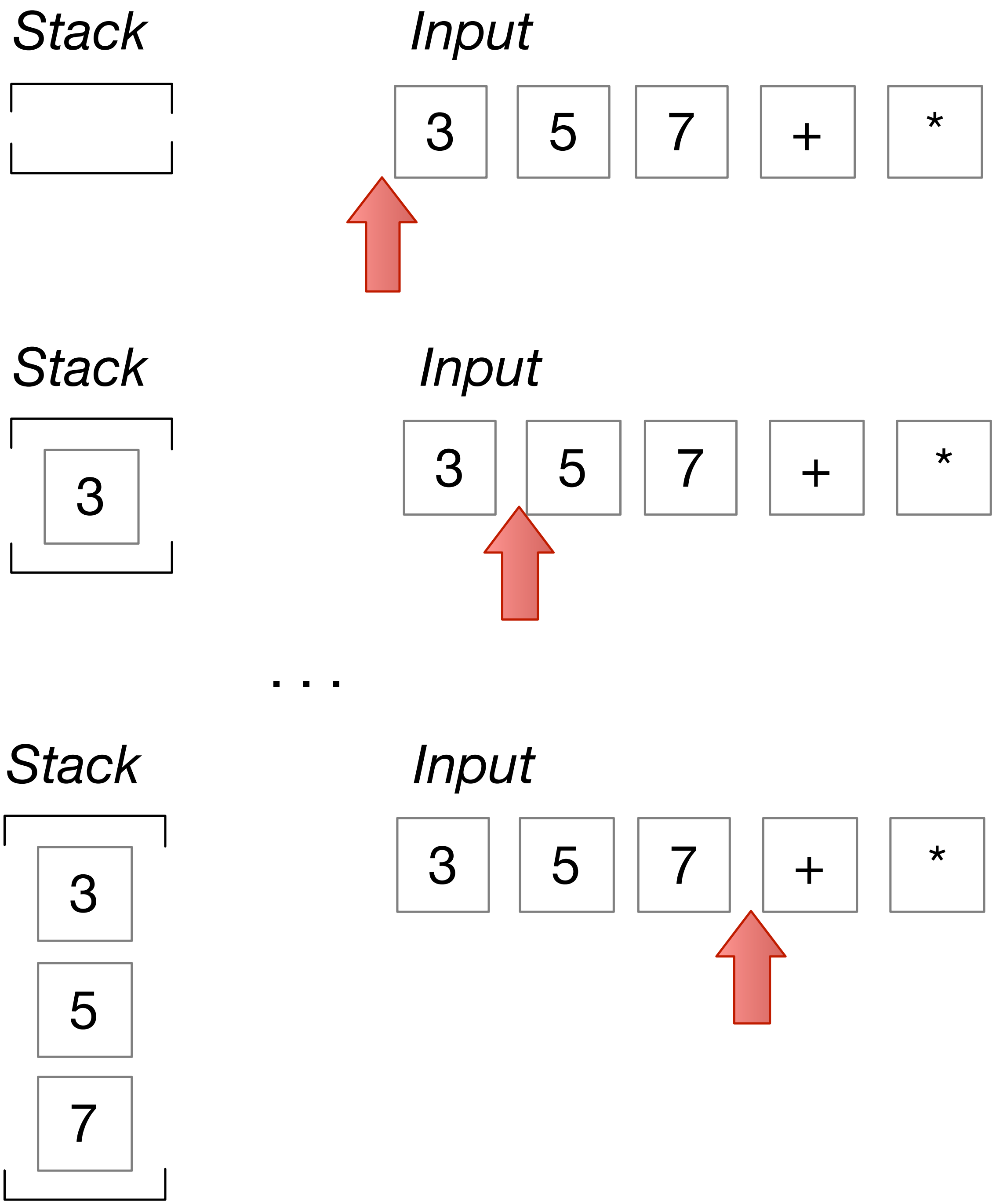

Expr
eval(self)
Env
map
eval(self, var)
assign(self, var)
Assign
left: Var
right: Expr
eval(self)
UnOp
left: Expr
eval(self)
_apply(self, left)
_eq(self, other)
BinOp
left: Expr
right: Expr
eval(self)
_apply(self, left, right)
_eq(self, other)
Neg
_apply(self, left)
Times
_apply(self, left, right)
Plus
_apply(self, left, right)
Div
_apply(self, left, right)
Minus
_apply(self, left, right)


Expr
eval(self)
Env
map
eval(self, var)
assign(self, var)
Assign
left: Var
right: Expr
eval(self)
UnOp
left: Expr
eval(self)
_apply(self, left)
_eq(self, other)
BinOp
left: Expr
right: Expr
eval(self)
_apply(self, left, right)
_eq(self, other)
You provide
Neg
_apply(self, left)
Times
_apply(self, left, right)
Plus
_apply(self, left, right)
Div
_apply(self, left, right)
Minus
_apply(self, left, right)

Parsing postfix (a.k.a. reverse Polish notation, RPN)



Except we want to form expression trees, not just values



Stack

Input

3 5 7 + *

Stack

3

Input

3 5 7 + *

. . .

Stack

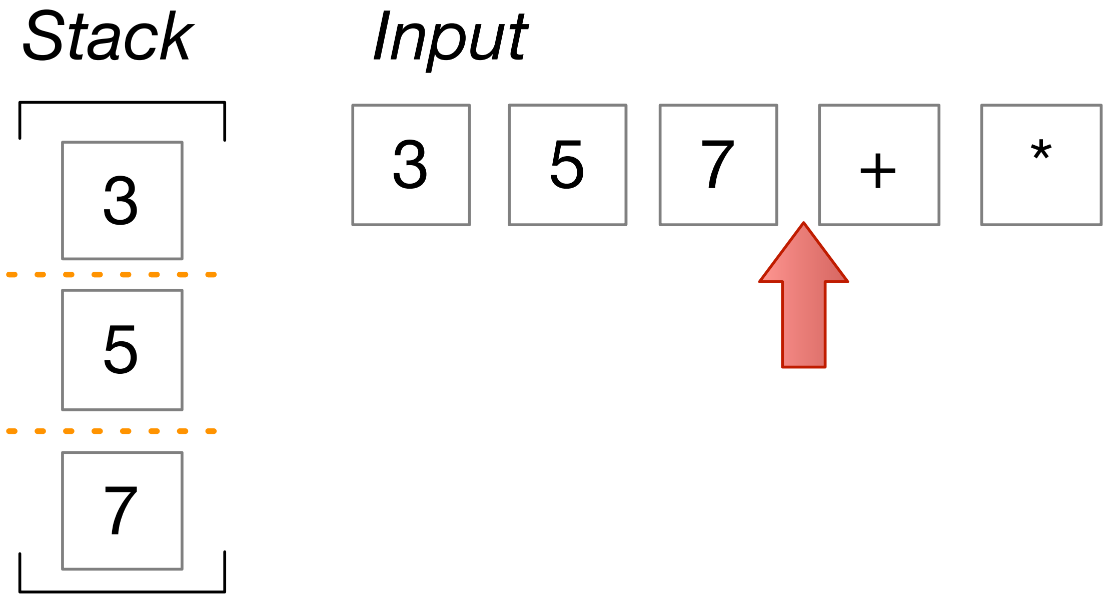
3

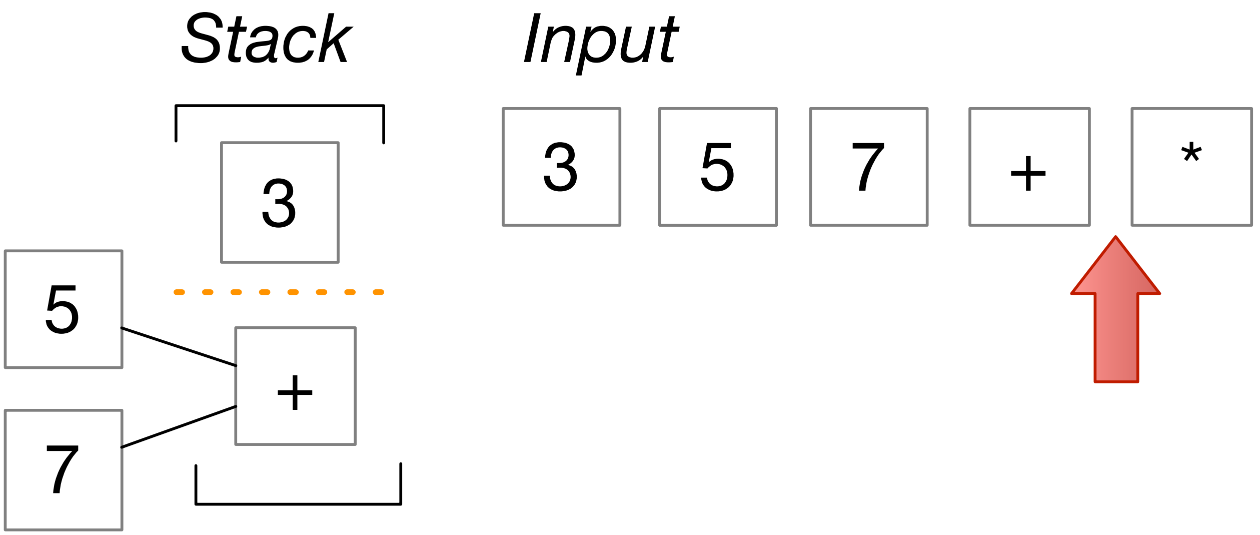
5

7

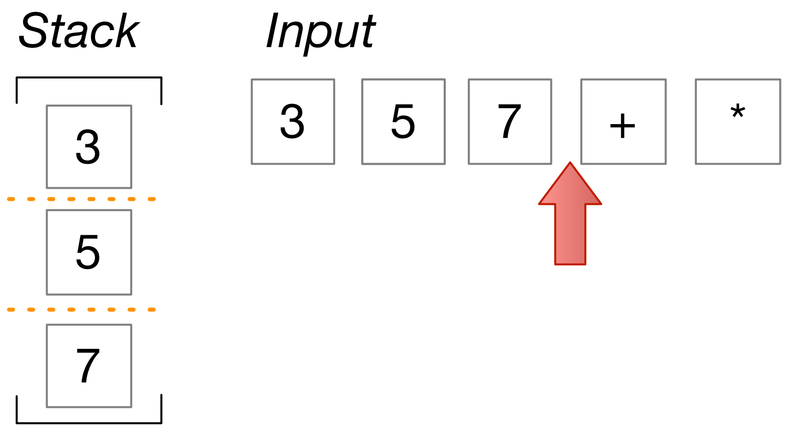
Input

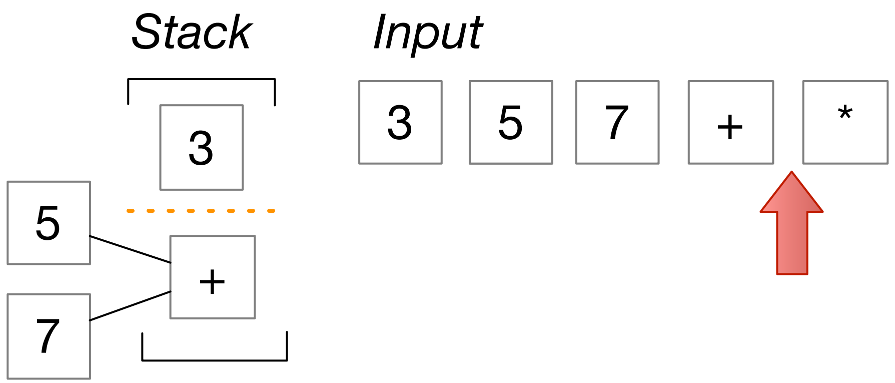
3 5 7 + *

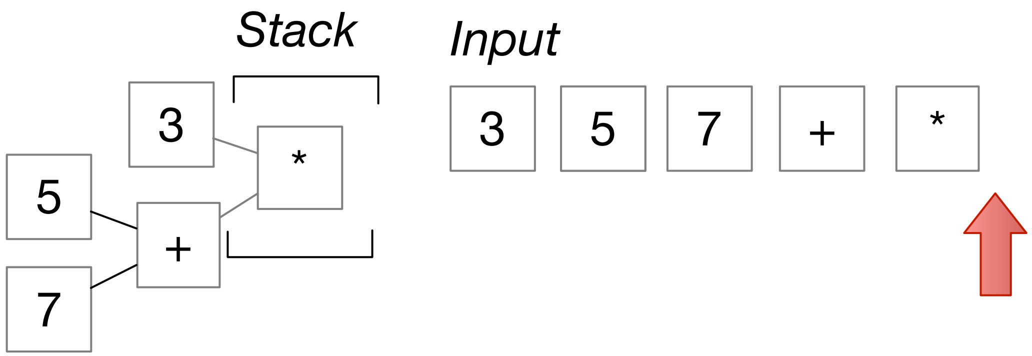

Stack
3
5
7
Input
3
5
7
+
*


Stack
3
5
+
7
Input
3
5
7
+
*

At end of input, we should have one expression node at the top of the stack.







“Stack” structure in Python

A ‘stack’ structure has ‘push’ (add to top) and ‘pop’ (take from top) operations

We could build a Stack class, but the ‘list’ built-in class is good enough:

Stack.push(el):   List.append(el)

Stack.pop() -> el: List.pop()

Completing the RPN parser

For each token:

- Pop the right number of operands
  - from 0 (for constants) to 2 (for +, *, etc)
  - after checking that they are on the stack
- Create the new node with operands
- Push new node onto stack

I provide lexical analysis, code for Assign and Neg nodes. You infer design of Binop.

Syntax table

syntax.py associates concrete syntax (e.g., “*”) with abstract syntax (classes in expr.py)

Example:

```
# Category names  (used in parsing)
ASSIGN = "ASSIGN" # Left operand must be a variable
BINOP = "BINOP"   # Any other operator with two operands, like Times
UNOP = "UNOP"     # Any operator with one operand, like Neg
CONST = "CONST"
IDENT = "IDENT"
```

```
# Each kind of operation node should be bound to a
# symbol and class here (excluding CONST and IDENT)
OPS = { "*": (BINOP, expr.Times)
       ,"+": (BINOP, expr.Plus)
       ,"-": (BINOP, expr.Minus)
       ,"/": (BINOP, expr.Div)
       ,"=": (ASSIGN, expr.Assign)
       ,"~": (UNOP, expr.Neg)
      }
```

Using the syntax table

```
stack = [ ]
stream = lexer.Token_Stream(s)
while stream.has_more():
    token = stream.take()
    if token.kind == syntax.ASSIGN:
        if len(stack) < 2:
            raise InputError("Insufficient operands for {}".format(token))
        right = stack.pop()
        left = stack.pop()
        op_class = token.clazz
        if not isinstance(left, expr.Var):
            raise InputError("First operand of assignment must be" +
                                 " a variable, not {}".format(left))
        node = op_class(left, right)
        stack.append(node)
    elif token.kind == syntax.BINOP:
```

Side note: slightly weird notation

```
# Each kind of operation node should be bound to a
# symbol and class here (excluding CONST and IDENT)
OPS = { "*": (BINOP, expr.Times)
        ,"+": (BINOP, expr.Plus)
        ,"-": (BINOP, expr.Minus)
        ,"/": (BINOP, expr.Div)
        ,"=": (ASSIGN, expr.Assign)
        ,"~": (UNOP, expr.Neg)
      }
```

Why do you think I put the comma at the beginning of the line? Why be weird?

You will occasionally see odd notational conventions like this, often motivated by ease of change.

Summary

“Symbolic” calculator allows mix of unbound and bound variables

Evaluation uses current values of bound variables; evaluates as far as possible

You provide:

- Binary operation classes (Times, Plus, etc)
  - including *abstract base class* for BinOp
- Parsing of binary operations (“*”, “+”, etc)

calc.py is the main program; also text_expr.py