

# 解释器



# 程序语言

## ◎ 高级语言

- ◆ Statements and expressions are interpreted by another program or compiled (translated) into another language
  - Provide means of abstraction such as naming, function definition, and objects
  - Independent of hardware and operating system

## ◎ 机器语言

- ◆ statements are interpreted by the hardware itself (sequence of 0s and 1s)
  - Instructions invoke operations implemented by the circuitry of CPU
  - no abstraction mechanism

# 程序语言

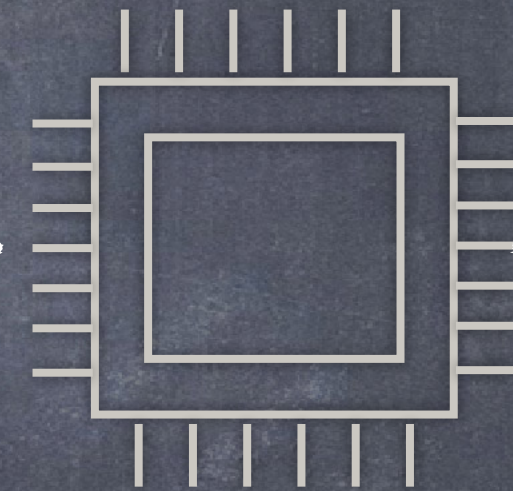
计算机只理解机器语言，如何让计算机理解高级语言？

翻译为机器语言（编译原理）

Print ("hello, world")



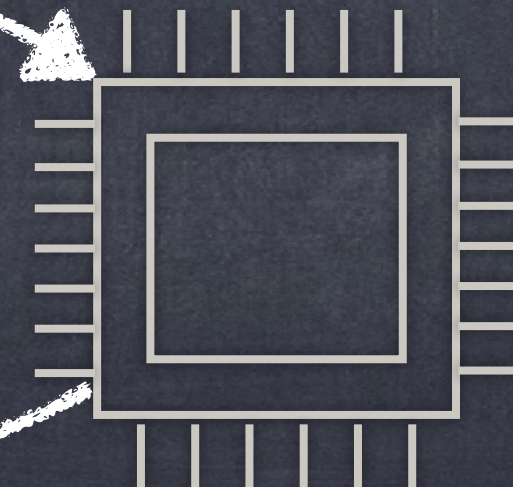
0100100010001



Hello, world

利用解释器作为桥梁

Print ("hello, world")



hello, world

Our focus



# 什么是解释 ( Interpret ) ?

• To decide what the intended meaning of something is

◆ 比如，我们赋予一个“狗”这个字的意义是某种动物🐶

◆ 又比如，对于  $5 + 4$  这样的字符序列，我们想让计算机理解为一个计算的表达式，并最终求值为 9



“这不是一个烟斗”

By Magritte

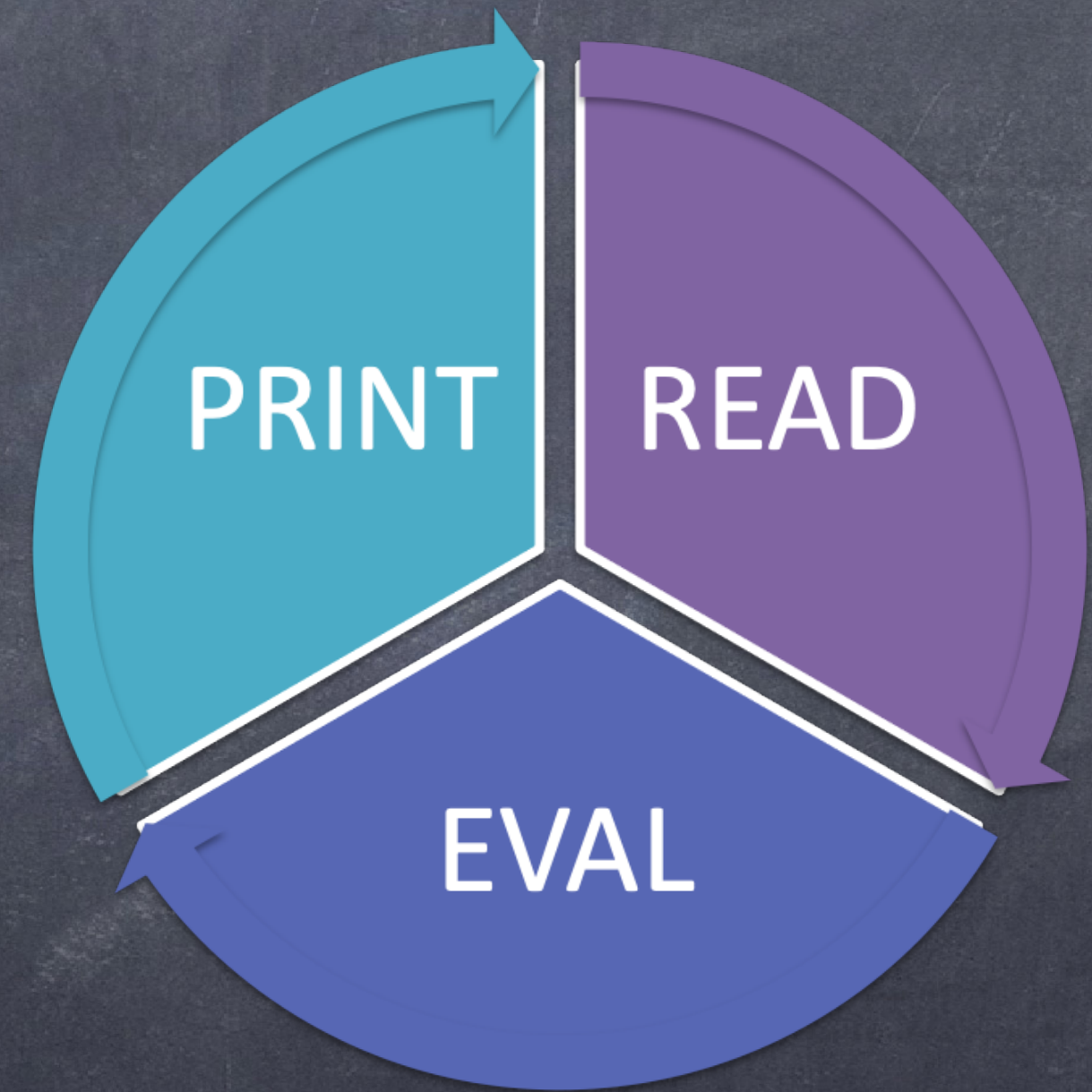
# 解释器

- ① 对于一个编程语言而言，一个解释器可以对一个属于该语言的表达式（复合语法规则的字符串）进行求值（赋予意义）

# Read-Eval-Print Loop (REPL)

很多解释器遵循着如下的基本流程：

- ◆ 读取 (Read) 用户的输入 (并转化为表达式)
- ◆ 求值 (Evaluate) 表达式
- ◆ 打印 (Print) 结果
- ◆ 循环 (Loop) 上述过程直至结束



这个流程就是 Read-Eval-Print Loop (REPL)

一个简单例子：计算器语言

# The calculator language ("Calc")

- 下面我们“创造”一个新的语言叫做“Calc”，并为其实实现解释器
  - ◆ 这个过程也叫“Metalinguistic Abstraction”，新创造的语言也被称为“domain-specific languages”
  - ◆ 一个新的语言往往有利于更好的描述问题空间 (Markdown、DOT、SQL、Unreal...)



# The calculator language ("Calc")

```
calc> add(3, 4)
```

```
7
```

```
calc> add(3, mul(4, 5))
```

```
23
```

```
calc> +(3, *(4, 5), 6)
```

```
29
```

```
calc> div(3, 0)
```

```
ZeroDivisionError: division by zero
```

# The calculator language ("Calc")

- 下面我们将用 python (the implementation language) 来实现 "Calc" (the implemented language)
- 用一个语言去解释另一个语言是常见的解释器设计方案
  - ◆ 比如 Python 用 C 解释 (Cpython), 用 Java 解释 (Jython), 甚至用 Python 解释 (PyPy)

# Syntax and Semantics of "Calc"

How expressions are structured

Syntax

Calc的两种类型的表达式：

- ◆ 基本表达式 (primitive expression) 就是一个数字
- ◆ 调用表达式 (call expression) 是一个操作符名字，加上括号，括号里是由逗号隔开的操作数列表

# Syntax and Semantics of "Calc"

## Syntax

① 操作符设定有四种：

◆ add(or +)

◆ sub(or -)

◆ mul(or \*)

◆ div(or /)

② 这些操作符设定为前缀 (prefix) 操作符，即放在所有操作数之前

# Syntax and Semantics of "Calc"

What do expressions mean

Semantics

我们用“值”来给表达式赋予意义，表达式的值是递归定义的：

- ◆ 基本表达式的值就是数字本身（比如，5, 3.14等）
- ◆ 调用表达式的值就是操作符作用在操作数上的结果：
  - `add(or +)`: 返回所有实参的求和
  - `sub(or -)`: 如果只有一个实参，求值为其负数。否则从第一个实参中减去剩余的实参
  - `mul(or *)`: 返回所有实参的积
  - `div(or /)`: 从第一个实参中除以剩余的实参

# 使用 Read-Eval-Print Loop

```
def read_eval_print_loop():  
    while True:  
        try:  
            expression_tree = calc_parse(input('calc> '))  
            print(calc_eval(expression_tree))  
        except ...:  
            # Error-handling code not shown
```

The diagram illustrates the Read-Eval-Print Loop process. It features four callout boxes: a green box labeled 'Read' pointing to the `input` function, a grey box labeled 'Parse' pointing to the `calc_parse` function, a purple box labeled 'Print' pointing to the `print` function, and a blue box labeled 'Eval' pointing to the `calc_eval` function. The code is written in a monospaced font on a dark background.

# Read

● 读取部分就是简单的讲用户输入的字符串读入到一个变量里

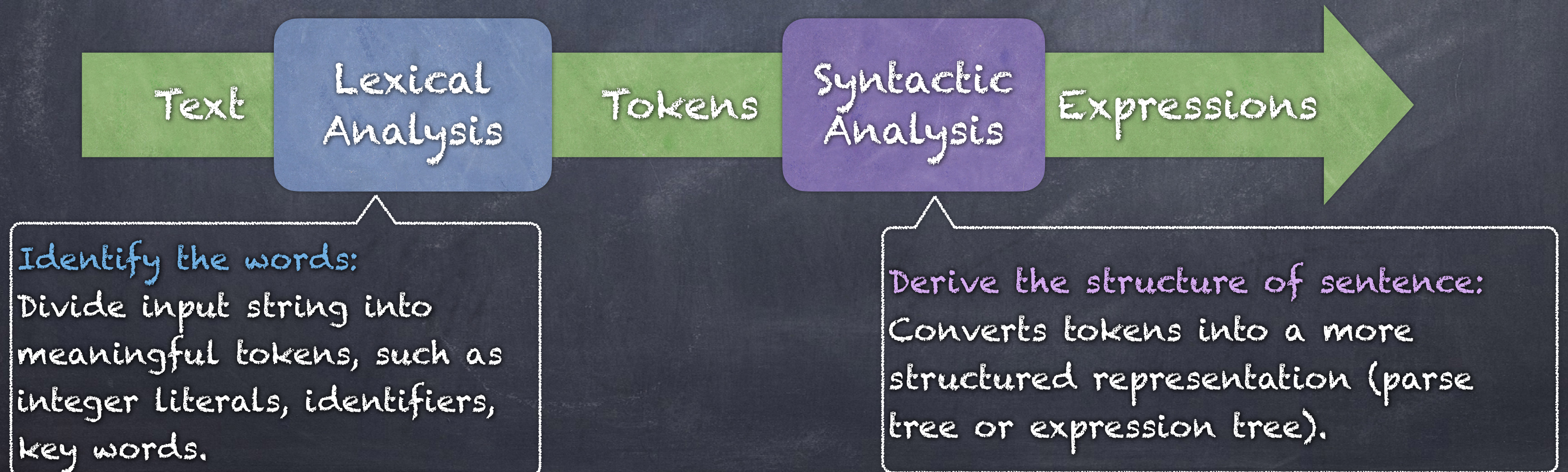
● 这里 `input('calc>')` 可以在终端打印提示符“calc>”，然后等待用户输入，返回输入的一行

但仅有原始的字符串是不够的，对于计算机而言，一个简单的结构化的表达式更加容易理解，其实人也是一样，句子得划分成合适的主、谓、宾等，太长的句子还不太容易看懂。。。。。。因此我们要进一步对原始的字符串进行“解析”！

# Parsing

● 一个解析器 (Parser) 接受字符串的输入并返回一个表达式

◆ 其含有两个主要步骤: 词法分析 (Lexical Analysis) 和语法分析 (Syntactic Analysis)





# Parsing

lexical analysis

```
def calc_parse(line):  
    tokens = tokenize(line)  
    expression_tree = analyze(tokens)  
    return expression_tree
```

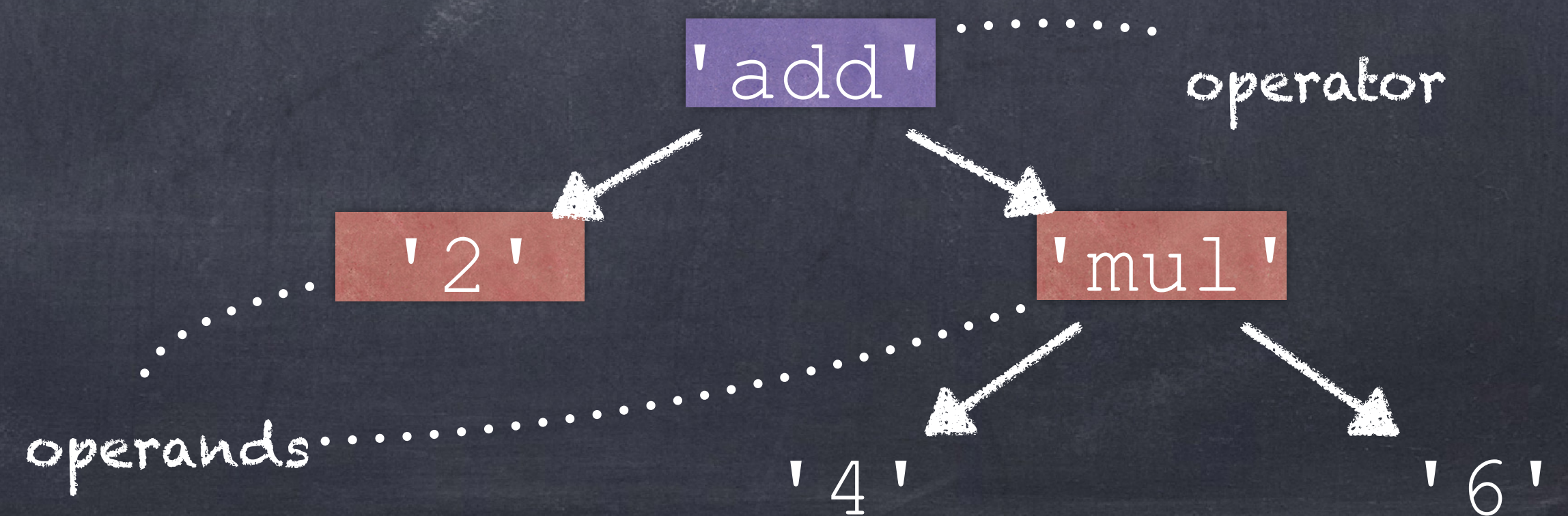
Syntactic analysis

```
tokenize('add(2, mul(4, 6))')
```

```
['add', '(', '2', ',', 'mul',  
'(', '4', ',', '6', ')', ')']
```

symbol: name      Delimiter      Symbol: Literal      Delimiter

```
['add', '(', '2', ',', 'mul',  
'(', '4', ',', '6', ')', ')']
```



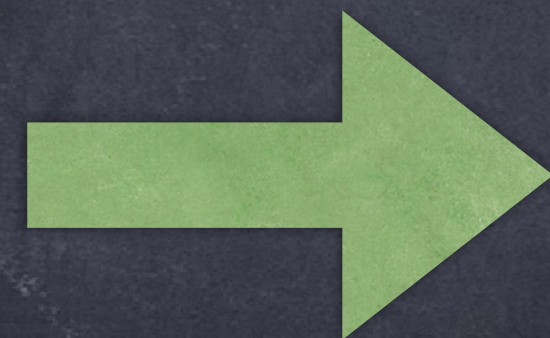
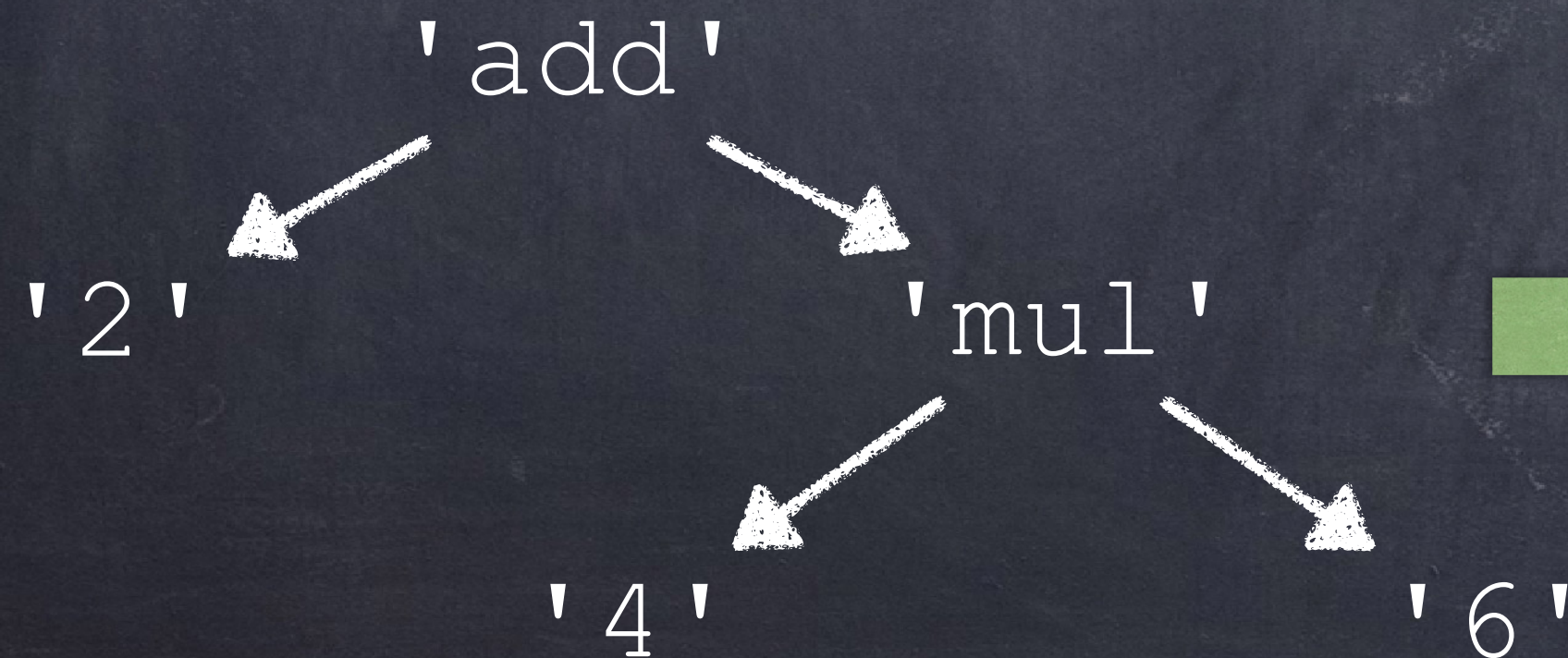
# Parsing

## ① 表达式 expression

String representing the operator of a Calc expression

```
class Exp:  
    def __init__(self, operator, operands):  
        self.operator = operator  
        self.operands = operands
```

List of either integers, floats, or other (children) Exp trees



Exp('add', [2,

Exp('mul', [4, 6]))]

# 回到Read-Eval-Print Loop

```
def read_eval_print_loop():  
    while True:  
        try:  
            expression_tree = calc_parse(input('calc> '))  
            print(calc_eval(expression_tree))  
        except ...:  
            # Error-handling code not shown
```

The diagram illustrates the Read-Eval-Print Loop process. It features four callout boxes: a green box labeled 'Read' pointing to the `input` function, a grey box labeled 'Parse' pointing to the `calc_parse` function, a purple box labeled 'Print' pointing to the `print` function, and a blue box labeled 'Eval' pointing to the `calc_eval` function. The code is written in a monospaced font on a dark background.

## 求值 ( Evaluation )

- 给定一个解析后的表达式 ( 表达式树 ) ,  
Evaluation 过程用来求出该表达式的值
- 改过程首先确定表达式的形式, 然后根据既定的  
求值规则来进行求值

# 求值规则

① 基本表达式直接求值为其所表示的数值

② 调用表达式递归的求值：

◆ 对每个操作数进行求值 (Evaluate)

◆ 将求得的值放到实参的列表上

◆ 将操作符作用在实参列表上 (Apply)

# 求值

If the expression is a number, then return the number itself.  
Numbers are self-evaluating: they are their own values.

```
def calc_eval(exp):  
    if type(exp) in (int, float):  
        return exp  
    elif type(exp) == Exp:  
        arguments = list(map(calc_eval, exp.operands))  
        return calc_apply(exp.operator, arguments)
```

Otherwise, evaluate the arguments recursively...

apply the operator on the argument values.

# 作用 (Apply)

```
def calc_apply(operator, args):  
    if operator in ('add', '+'):  
        return sum(args)  
    if operator in ('sub', '-'):  
        if len(args) == 1:  
            return -args[0]  
        return sum(args[0] + [-arg for arg in args[1:]])  
    ...
```

define all the behaviors of operators we want Calc to understand

# Eval and Apply

- Eval调用了Apply, 也递归地调用了Eval自身
- 形成了一个eval-apply cycle
  - ◆ 这是程序语言解释器的核心



一个复杂例子：解释一个python的子集语言Py

# 一个比Calc更加复杂语言Py

## ● 有了环境

- ◆ 有了变量可以在“记忆”信息，这些变量可以在环境中可访问
- ◆ 函数有参数、函数体、并且知道当被调用时所在的环境

## ● 有了语句

- ◆ 语句比表达式更加复杂，表达了自身求值、改变环境等操作

# REPL For Py

```
def read_eval_print_loop(interactive=True):
```

```
    while True:
```

```
        try:
```

```
            statement = py_read_and_parse()
```

```
            value = statement.evaluate(the_global_environment)
```

```
            if value != None and interactive:
```

```
                print(repr(value))
```

```
        except ...:
```

```
            ...
```

Read in a statement and produce a Stmt object.

Print the value

Evaluate the statement we read.  
Use the global environment.

## 环境 (Environment)

- 环境的作用是追踪变量名字和相应的值，以便支持在该环境下的计算。
  - ◆ 可以利用字典这样的容器对名字和值进行绑定

# Py: Environment

```
class Environment:  
    ...  
    def __init__(self, enclosing_environment=None):  
        self.enclosing = enclosing_environment  
        self.bindings = {}  
        self.nonlocals = []  
    ...
```

**self.enclosing**

parent environment

**self.bindings**

name1: value1  
name2: value2  
name3: value3  
...

**self.nonlocals**

name1' name2' name3'  
...

# Py: Environment

```
class Environment:  
    ...  
    def is_global(self):  
        return self.enclosing is None  
  
    def note_nonlocal(self, var):  
        self.nonlocals.append(var)  
    ...
```

全局帧没有父帧

标记nonlocal变量（不隶属于当前帧）

# Py: Environment

在环境中查找  
名字绑定的值

定义了该方法后，可以通过 对象名[var] 来获得相应的属性值

```
class Environment:  
    ...  
    def __getitem__(self, var):  
        if var in self.bindings:  
            return self.bindings[var]  
        elif not self.is_global():  
            return self.enclosing[var]  
        else:  
            raise ...  
    ...
```

先从当前绑定找

否则就往其父帧找 (递归)

都找不到，就异常

# Py: Environment

在环境中改变  
名字绑定的值

如果是nonlocal就到父帧改变值，  
否则就改变binding的值

```
class Environment:  
    ...  
    def set_variable(self, name, value, is_nonlocal=False):  
        if is_nonlocal:  
            self.enclosing[name] = value  
        else:  
            self.bindings[name] = value  
    def setitem(self, name, value):  
        self.set_variable(name, value, name in self.nonlocals)  
    ...
```

定义了该方法后，可以通过 对象名[name] = value 来设置的属性值



# 语句 ( Statement )

## ● 有多种不同类型的语句

- ◆ 赋值语句：自动对表达式进行求值，并将值绑定到目标名字上
- ◆ 调用语句：对操作符和操作数分别求值，然后将执行Apply
- ◆ 控制语句：复合语句，包含布尔表达式，语句块，首先对布尔表达式进行求值，根据结果是否对语句块中的语句进行求值

# Py: Assign Statement

Create the statement with all of the information that the statement is composed of.

```
class AssignStmt(Stmt):  
    def __init__(self, target, expr):  
        self.target = target  
        self.expr = expr  
  
    def evaluate(self, env):  
        env[self.target] = self.expr.evaluate(env)
```

Evaluate this statement in a given environment.

# Py: Call Expressions

A call expression has an operator and a series of operands.

```
class CallExpr(Expr):
```

```
    def __init__(self, op_expr, opnd_exprs):
```

```
        self.op_expr = op_expr
```

```
        self.opnd_exprs = opnd_exprs
```

```
    def evaluate(self, env):
```

```
        func_value = self.op_expr.evaluate(env)
```

```
        opnd_values = [opnd.evaluate(env) for opnd in self.opnd_exprs]
```

```
        return func_value.apply(opnd_values)
```

Evaluate the operator.

Evaluate the operands

Apply the value of the operator onto the value of the operands.

核心是function对象的apply

# 构造函数表达

```
class Function:  
    def __init__(self, *args):  
        raise NotImplementedError()  
  
    def apply(self, operands):  
        raise NotImplementedError()
```

内置的基本函数

```
class PrimitiveFunction(Function):  
    def __init__(self, procedure):  
        self.body = procedure  
  
    def apply(self, operands):  
        return self.body(operands)
```

Apply就是直接调用相应的procedure，作用在实参之上

# 内建基本函数

具体的procedure

```
primitive_functions = [  
    ("or", PrimitiveFunction(lambda x, y: x or y)),  
    ("and", PrimitiveFunction(lambda x, y: x and y)),  
    ("not", PrimitiveFunction(lambda x: not x)),  
    ("eq", PrimitiveFunction(lambda x, y: x == y)),  
    ("ne", PrimitiveFunction(lambda x, y: x != y)),  
    ...  
]
```

```
def setup_global():  
    for name, primitive in primitive_functions:  
        the_global_environment[name] = primitive
```

调用这个函数可以讲这些基本内置函数  
绑定到全局帧中，从而被其他帧访问

# 用户定义函数

```
class CompoundFunction(Function):  
    def __init__(self, args, body, env):  
        self.args = args  
        self.body = body  
        self.env = env
```

函数包含：参数、函数体、所在环境

当调用时，首先创建一个帧

```
def apply(self, operands):  
    call_env = Environment(self.env)  
    if len(self.args) != len(operands):  
        raise TypeError("Wrong number of arguments passed to function!")  
    for name, value in zip(self.args, operands):  
        call_env[name] = value  
    for statement in self.body:  
        try:  
            statement.evaluate(call_env)  
        except StopFunction as sf:  
            return sf.return_value  
    return None
```

将实参绑定到形参上

直到返回了什么

在当前帧中，对函数体内每个语句进行求值

# 用户定义函数

将返回语句和异常实现的主要原因是：抛出异常之后直接终止正常的运行，进入到异常的处理过程，这个方式和返回语句语义相似：一旦遇到return语句，则不再顺序执行下面的代码，而是直接返回值

```
class StopFunction(BaseException):  
    def __init__(self, return_value):  
        self.return_value = return_value
```

```
class ReturnStmt(Stmt):  
    def __init__(self, expr=None):  
        self.expr = expr
```

返回的表达式

```
def evaluate(self, env):  
    if self.expr is None:  
        raise StopFunction(None)  
    raise StopFunction(self.expr.evaluate(env))
```

表达式的求值作为函数的返回值

# Py: Control Statement

```
class ControlStmt(Stmt):  
    def __init__(self, expr, blocks):  
        self.expr = expr  
        self.blocks = blocks
```

控制语句是复合语句，控制表达式和语句blocks

```
    def evaluate(self, env):  
        var = self.expr.evaluate(env)  
        executed = []  
        if var:  
            executed = blocks[0]  
        else:  
            executed = blocks[1]  
        for statement in executed:  
            statement.evaluate(env)
```

这里是简单的 if else 复合语句，如果有更多的 elif，则需要更多的处理



# Summary

- 我们实际上是用“python的程序语句”来表达对象语言（我们自己定义的语言）的语义
- 这种表达语义的方式就是操作语义（“Operational Semantics”），即描述语言是如何执行的

注意：实际的操作语义是更加形式化的数学语言描述

Thanks!