Special Topics I

Intro to Compilers and CPython Internals

Parts of this lecture are heavily inspired from "See CPython run: Getting to know your Python interpreter" by James Bennett

Announcements

- Hw06 and Lab12 due **today 8/03**
- Scheme Released
  - Checkpoint 2 due **Friday, 8/04**
  - Project party **today 8/03 3-5:30 PM Wozniak Lounge**
  - Whole project due Tuesday 8/08. EC for submitting on 8/7
  - Submit to the correct autograder!
- **Scheme contest** due **Friday, 8/04**
- Hw05 recovery released!
- There were some technical issues with the website. Materials will be uploaded shortly after lecture.
- Homework 7 released **tomorrow**. It will be on the shorter end!
- Final exam on 8/10 6-9 PM
  - Submit exam alteration form ASAP

Levels of Languages

High-level Language

(Python, Scheme, SQL, Java)

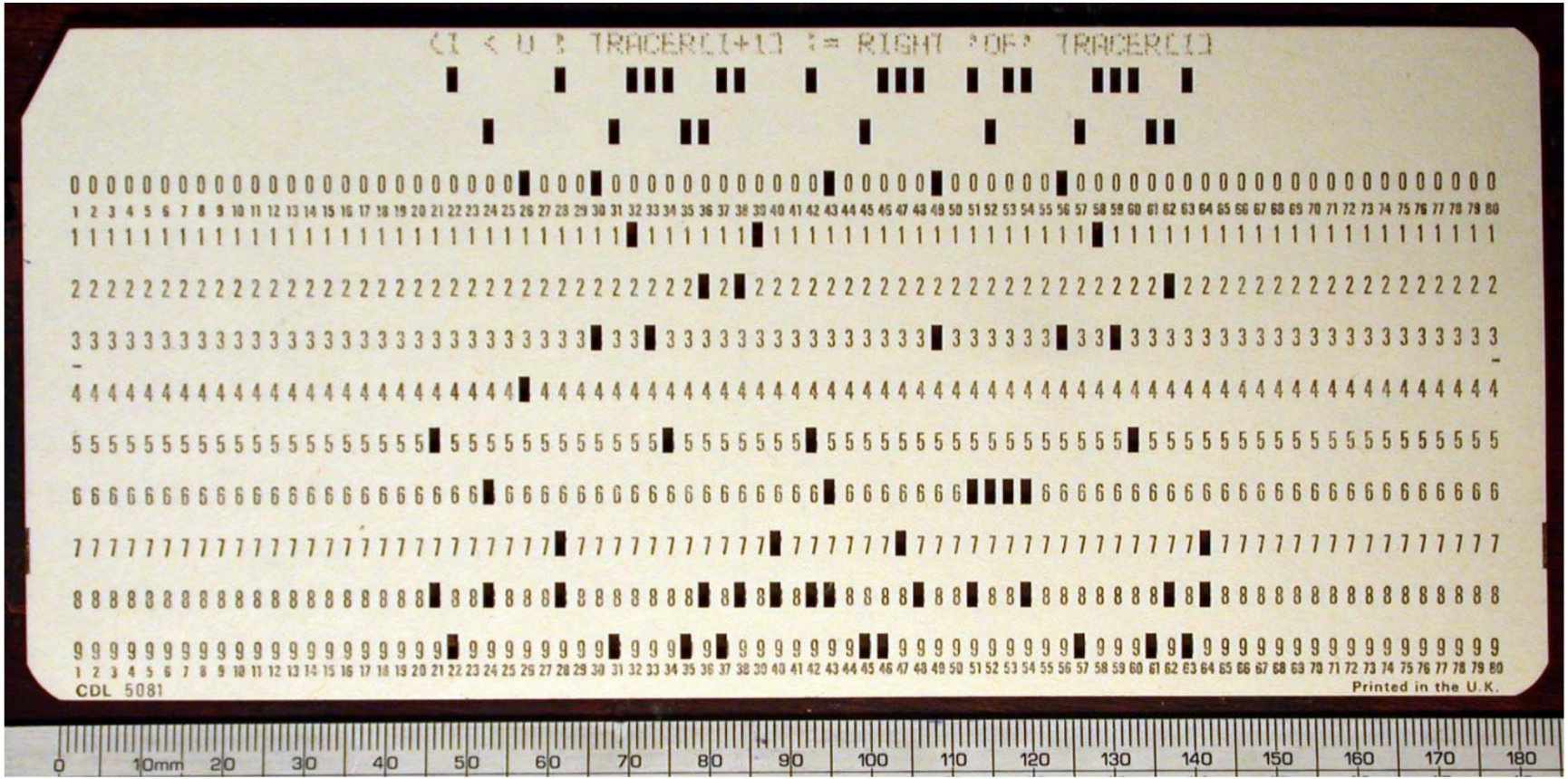
Assembly Language

(RISC-V Assembly, x86 Assembly)

Machine Language

(RISC-V Instruction Set, x86 Instruction Set)

Punchcard



Programming Languages

A computer typically executes programs written in many different programming languages

Machine languages: statements are interpreted by the hardware itself

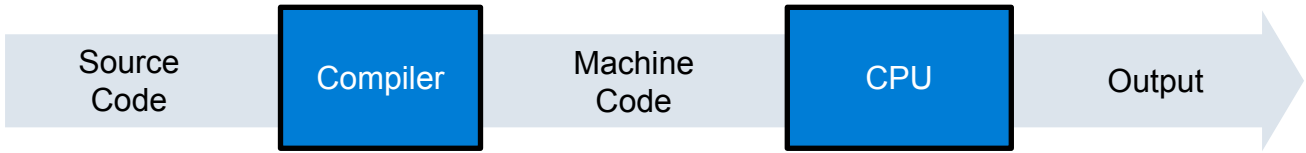
- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

High-level languages: statements & expressions are interpreted by another program or compiled (translated) into another language

- Provide means of abstraction such as naming, function definition, and objects
- Abstract away system details to be independent of hardware and operating system

Compilers

Compilers: translate source code into machine code so that the machine code can be distributed and run repeatedly




```
int a = (b + c) - (d + e);
```

C Code

```
add t1, s1, s2
add t2, s3, s4
sub s0, t1, t2
```

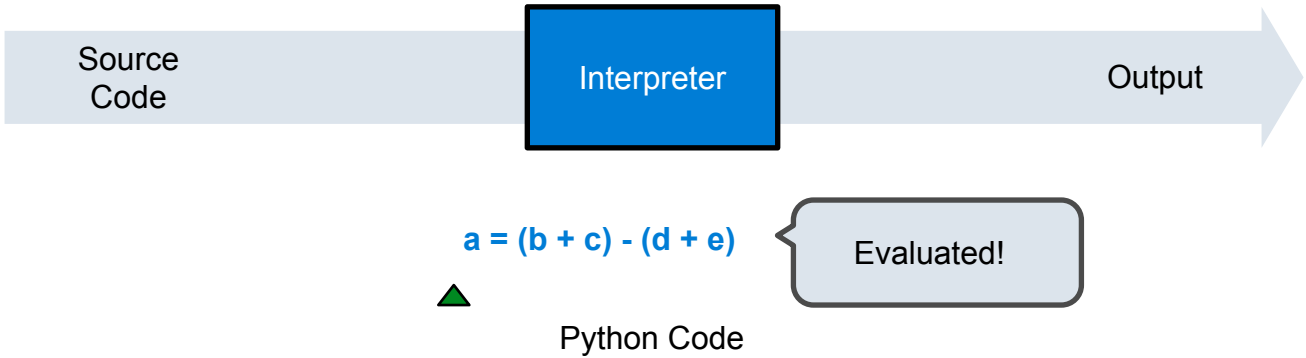
Assembly Language (RISC-V)

```
0000 0001 0010 0100 1000 0011 0011 0011
0000 0001 0100 1001 1000 0011 1011 0011
0100 0000 0111 0011 0000 0100 0011 0011
```

Machine Language

Interpreters

Interpreters: run source code directly producing an output/value, without first compiling it into machine code

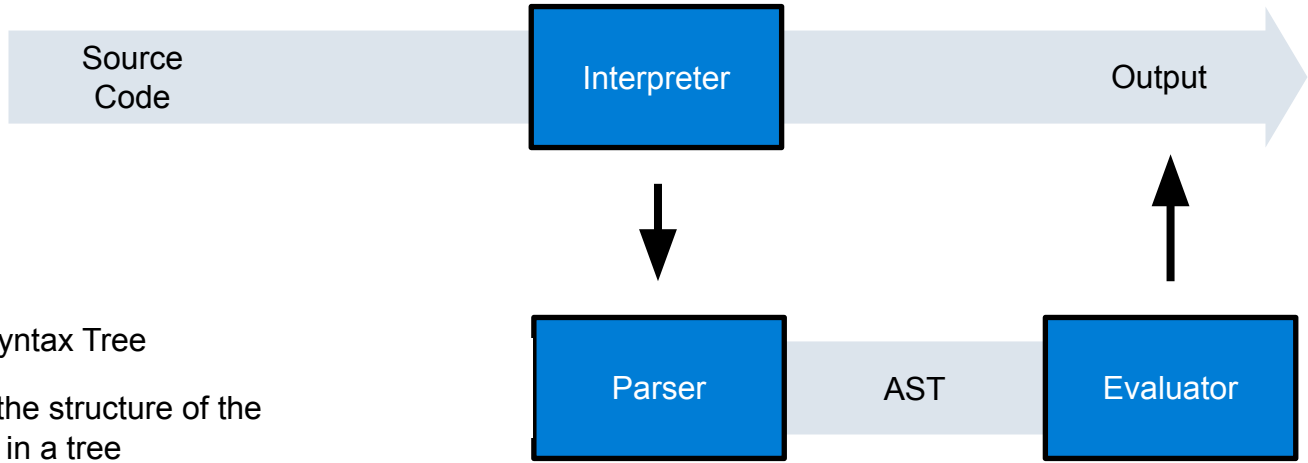


Tradeoffs:

Understanding Source Code

In order to interpret source code, a **parser** must be written to understand that source code

In the context of interpreters:



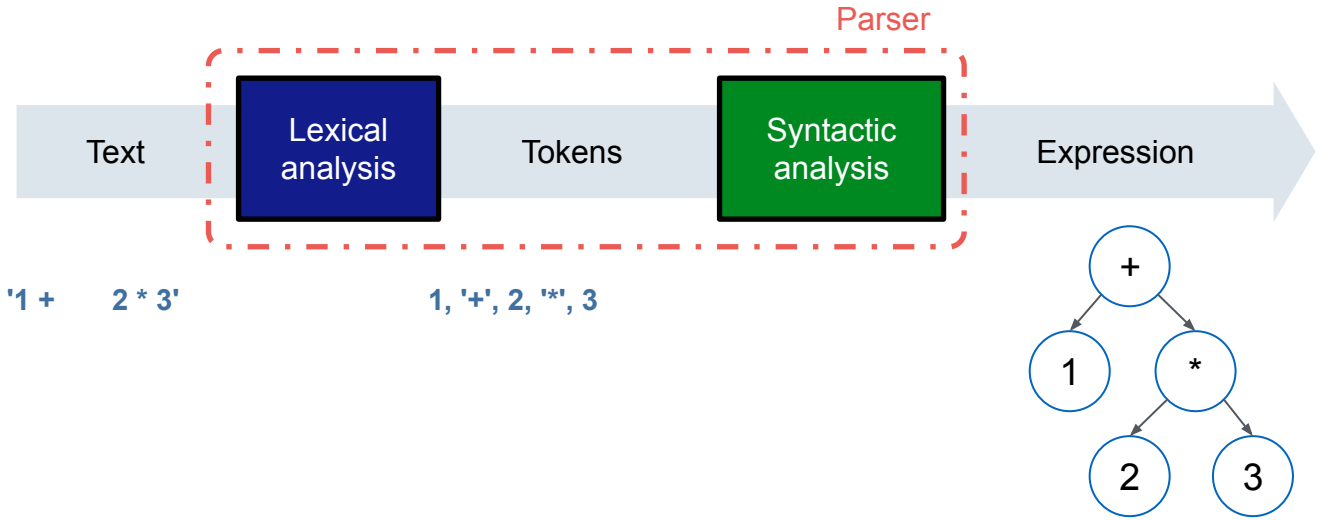
AST - Abstract Syntax Tree

- Represents the structure of the source code in a tree

Parsing

Parsing

A Parser takes in text and returns an expression that represents the text in a tree-like structure

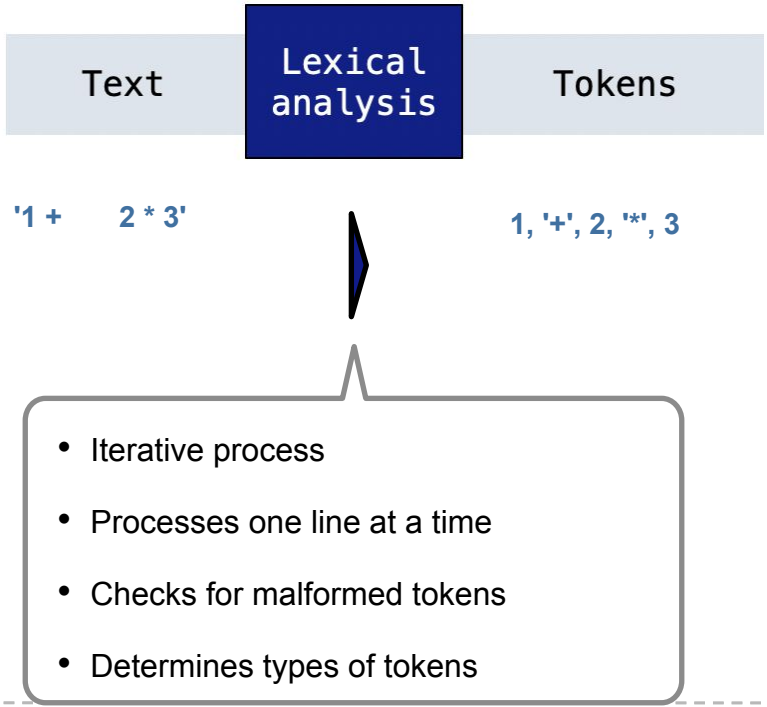


Let's break this down!

Lexical Analysis

Lexical analysis converts input text into a list of tokens

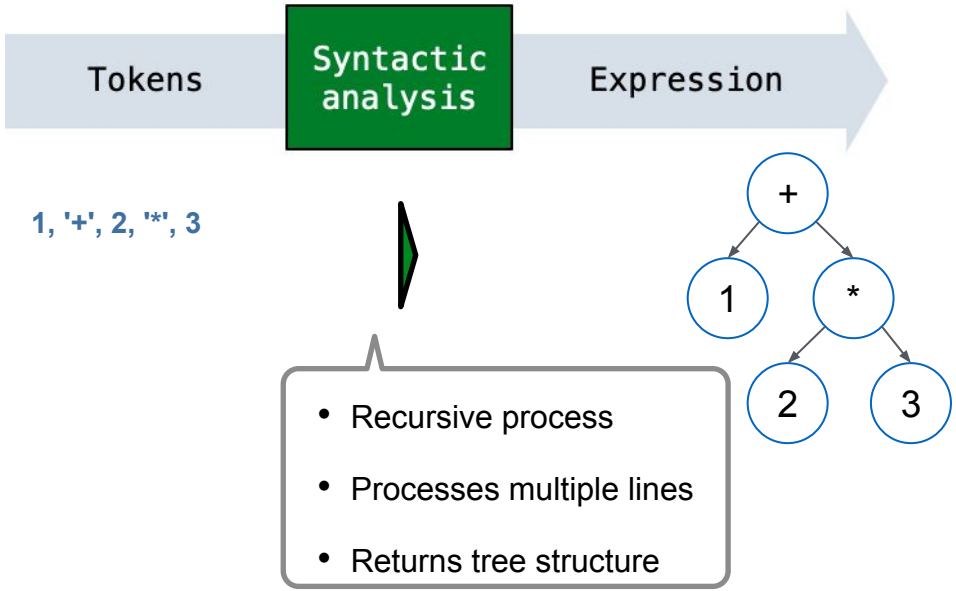
- Each token represents **the smallest unit of information**



Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression

- Formal way of representing the tokens generated from lexical analysis
- Symbols can be “nested”



BNF

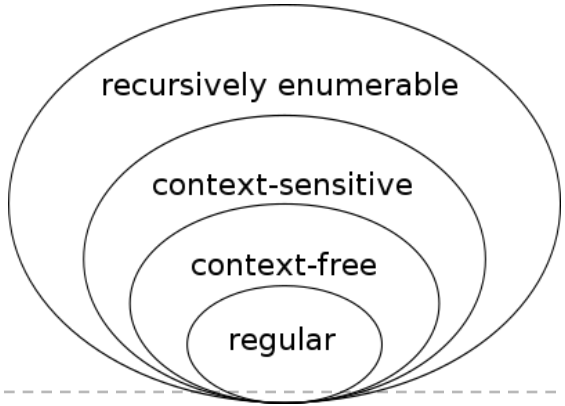
Backaus-Naur Form is a schema designed specifically for describing the syntax of programming languages using context-free grammars

A context-free grammar can be parsed statement by statement without needing prior context. Not all grammars are context free.

BNF is composed of a series of “production rules”, which can be thought of as symbol substitutions

BNF has been taught formally in previous iterations of this class, but isn’t a focus this semester

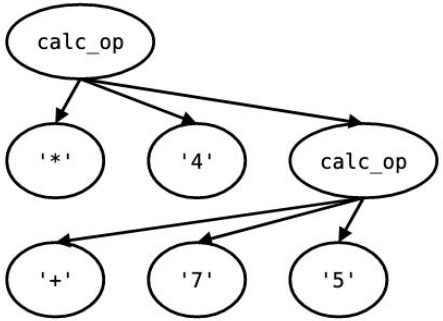
Python has a grammar!



https://docs.python.org/3/reference/grammar.html

BNF for (part of) Calculator

```
?start: calc_expr
?calc_expr: NUMBER | calc_op
calc_op: "(" OPERATOR calc_expr* ")"
OPERATOR: "+" | "-" | "*" | "/"
```



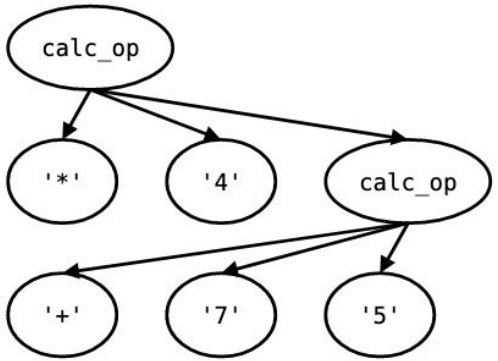
If I make a really good CFG for the Scheme language, I can actually pass a BNF grammar into an algorithm to make my parser, which would have saved you a lot of work in Lab 9

ASTs

AST - short for **abstract syntax tree**. Represents the hierarchical structure of formal languages.

ASTs …

- are **unambiguous**
- can be annotated. Very important for **statically** typed languages.
- can hold additional information about code
- don’t include extra structural information! No parentheses!
- can be transformed
- are typically built by the **parser**




```
(* 4 (+ 7 5))
```

Parsing Python

Inspecting Python

Ever wonder about those syntax check questions?

```
def two_of_three(i, j, k):
    """Return m*m + n*n, where m and n are  the two smallest members of the
    positive numbers i, j, and k.
    """
    return _____

def two_of_three_syntax_check():
    """Check that your two_of_three code consists of nothing but a return statement.

    >>> # You aren't expected to understand the code of this test.
    >>> import inspect, ast
    >>> [type(x).__name__ for x in ast.parse(inspect.getsource(two_of_three)).body[0].body]
    ['Expr', 'Return']
    """
    # You don't need to edit this function. It's just here to check your work.
```

How does this work?

We're going to think about this!

Inspect and Ast Modules

Inspect - module that has useful functions to get information about live objects such as classes, functions, frames, etc. For example, it can retrieve the source code of a method

Ast - module to help process trees of the Python abstract syntax grammar

```
def two_of_three_syntax_check():
    """Check that your two_of_three code consists of nothing but a return statement.
    >>> import inspect, ast
    >>> [type(x).__name__ for x in ast.parse(inspect.getsource(two_of_three)).body[0].body]
    ['Expr', 'Return']
    """
```

Returns an AST node!

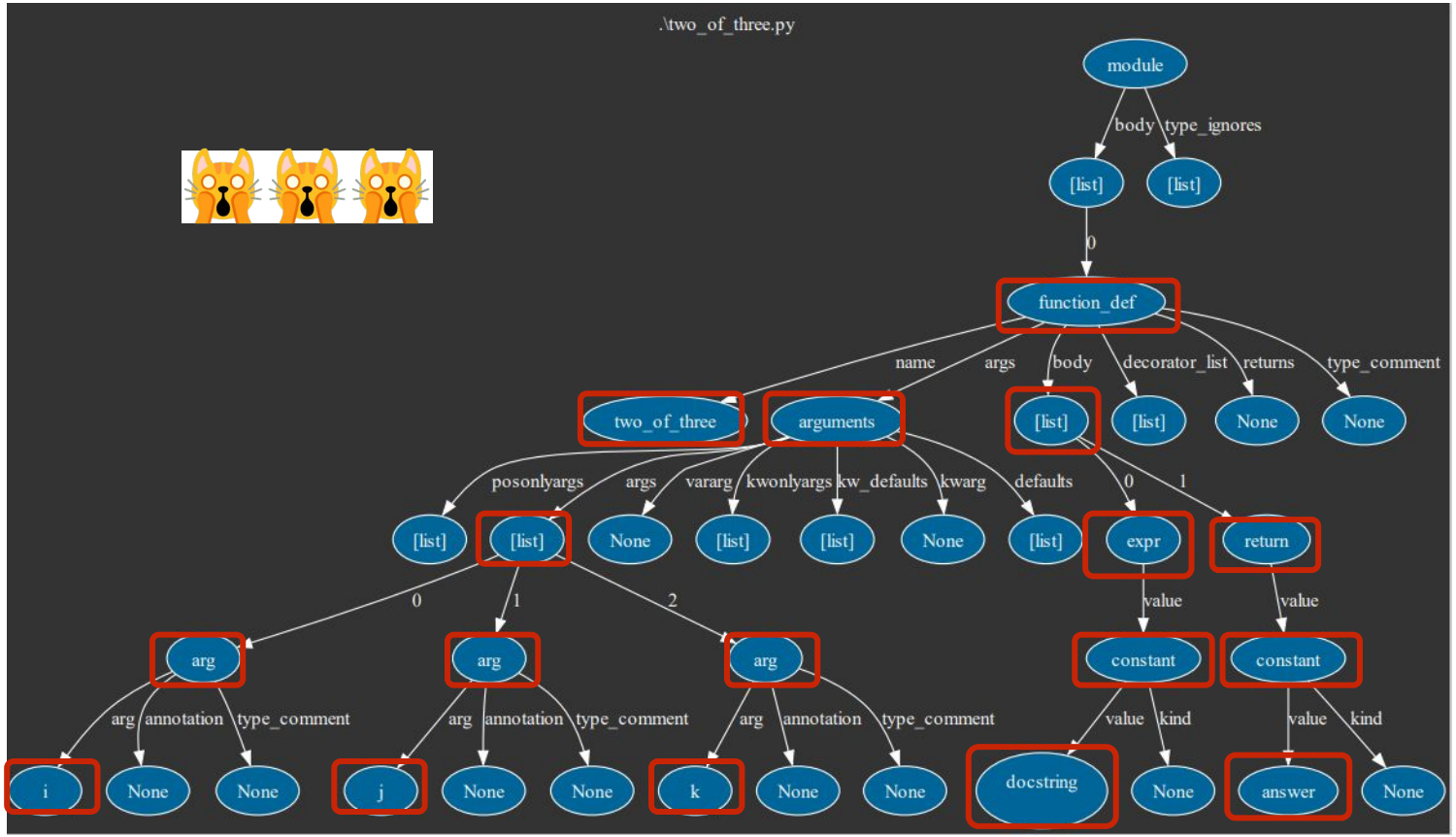
```
<ast node>
```

Returns the source code!

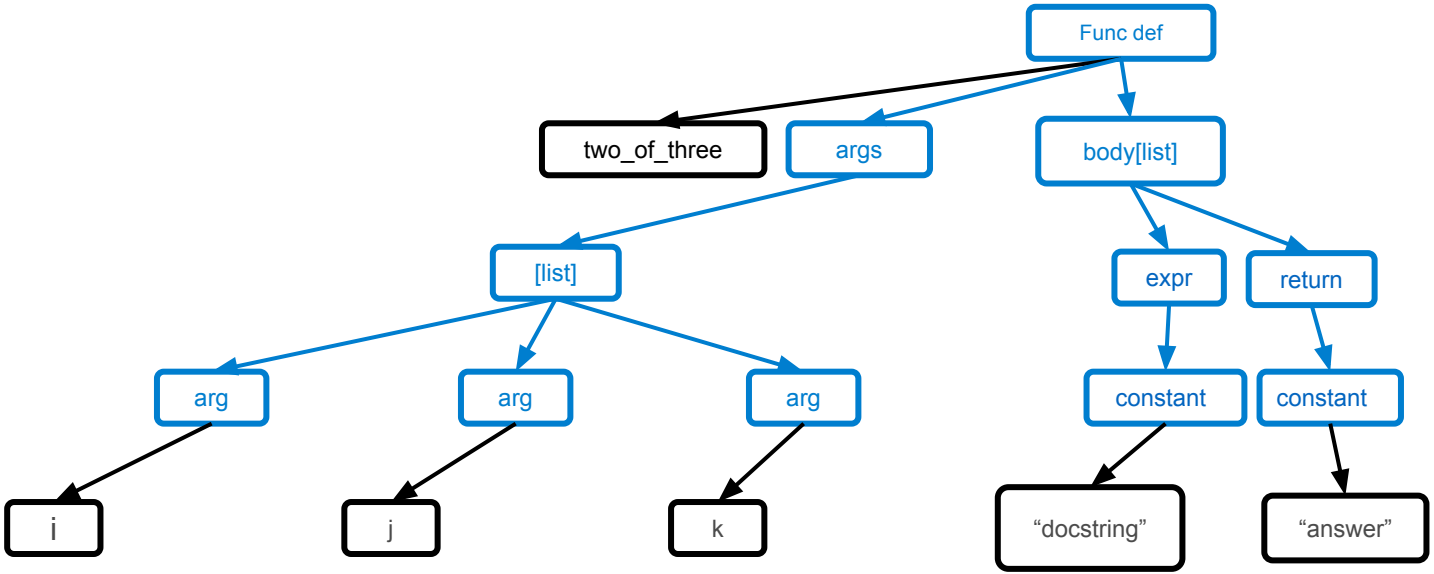
```
'def two_of_three(i, j, k):\n
"""docstring"""\n    return "answer"\n
```

(Demo)

What Python Sees



What We Care About



Dangling Else (Variation)

Ternary Expression recap:

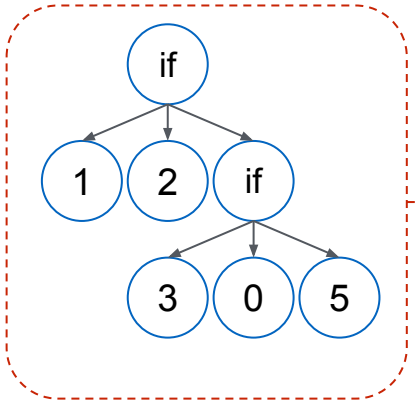
```
<if-true> if <cond> else <if-false>
```

```
>>> 1 if True else 3
1
>>> 1 if False else 3
3
```

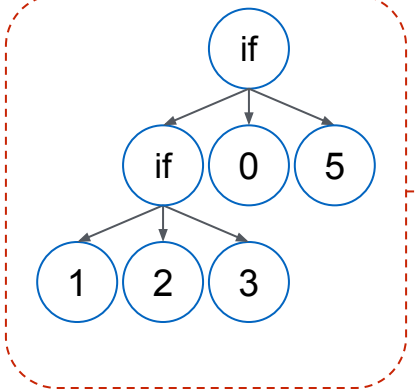
What does this evaluate to?

```
>>> 1 if 2 else 3 if 0 else 5
```

What does the AST look like?



Evaluates to 1!



Evaluates to 5!

(Demo)

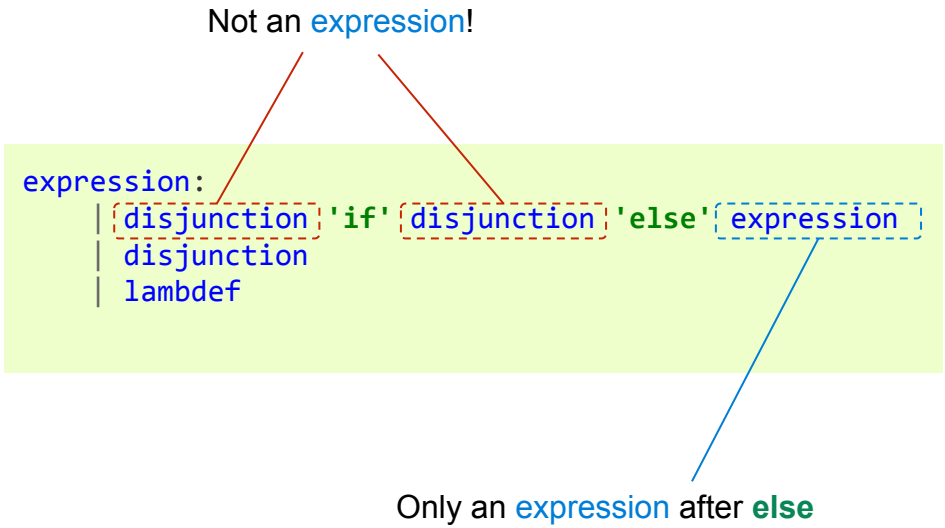
Dangling Else (Variation)

What does this evaluate to?

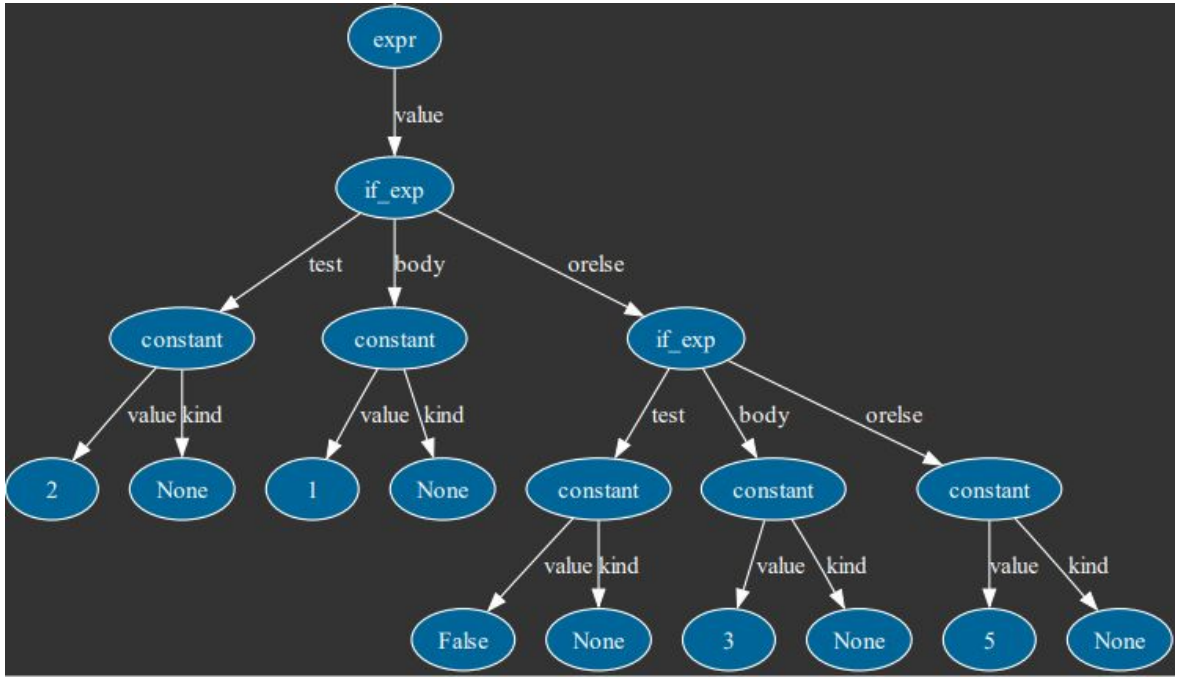
```
>>> 1 if 2 else 3 if False else 5
```

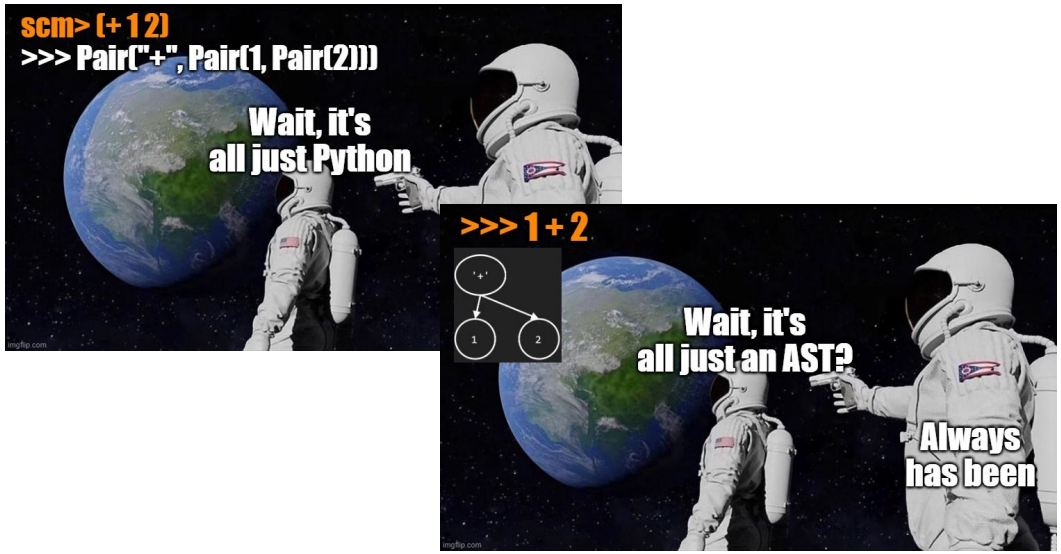
What does Python do?

(Demo)



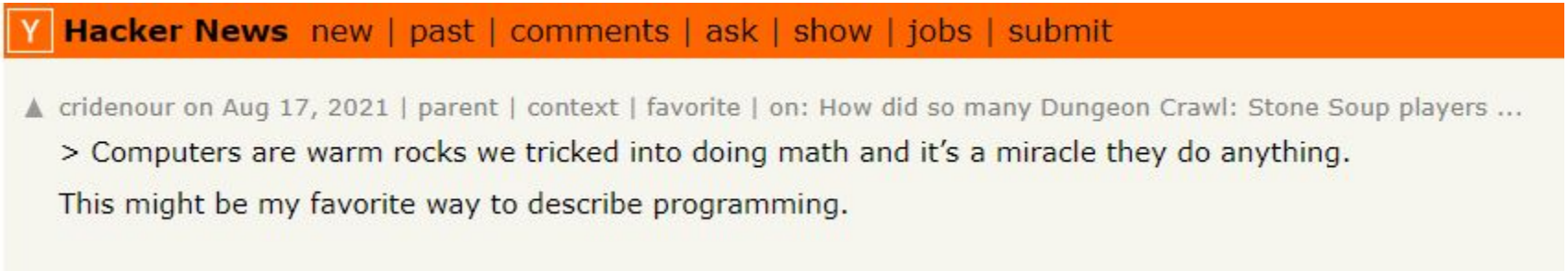
Python's Solution




scm> (+ 1 2)
>>> Pair("+", Pair(1, Pair(2)))
Wait, it's
all just Python
>>> 1 + 2
'+'
1
2
Wait, it's
all just an AST?
Always
has been

Break

Computers Are Magical

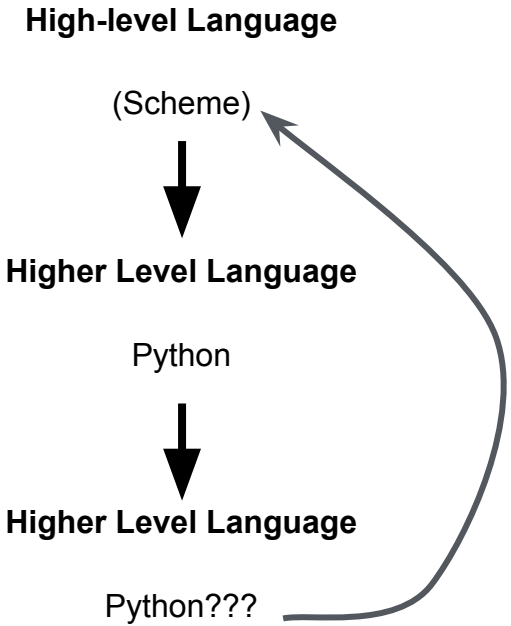


https://news.ycombinator.com/item?id=28205823

Is Python Interpreted or Compiled?

Yes

Who's Interpreting Who?



Python Implementations

Popular Implementations!

- CPython (What you download in this class!)
- PyPy (Python implemented with a stripped down version of Python)
- Jython (Java)
- Skybison (C++)
- ClPython (Lisp! The circle completes!)
- Brython! (Python in the browser using Javascript! It's how code.cs61a.org works!)
- RustPython
- MicroPython (Reduced language for embedded systems

Interesting Implementations

- LOLPython (Python but in I Can Haz Cheezburger speak)
- x-Python (CPython interpreter written in Python)
- Unladen Swallow (Google's attempt to speedup Python, but no longer supported)

https://wiki.python.org/moin/PythonImplementations

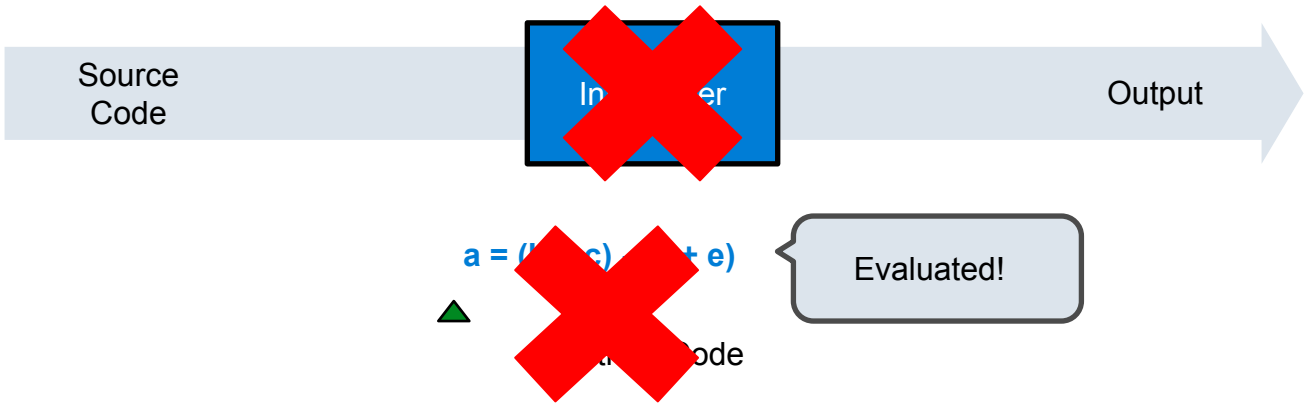
What is Python? (CPython)

Python is many things

- Python Interpreter
- Parser
- Compiler
- Virtual Machine
- Standard Library
- C API
- Big snek
- More…

Python Interpreters

Interpreters: run source code directly producing an output/value, without first compiling it into machine code

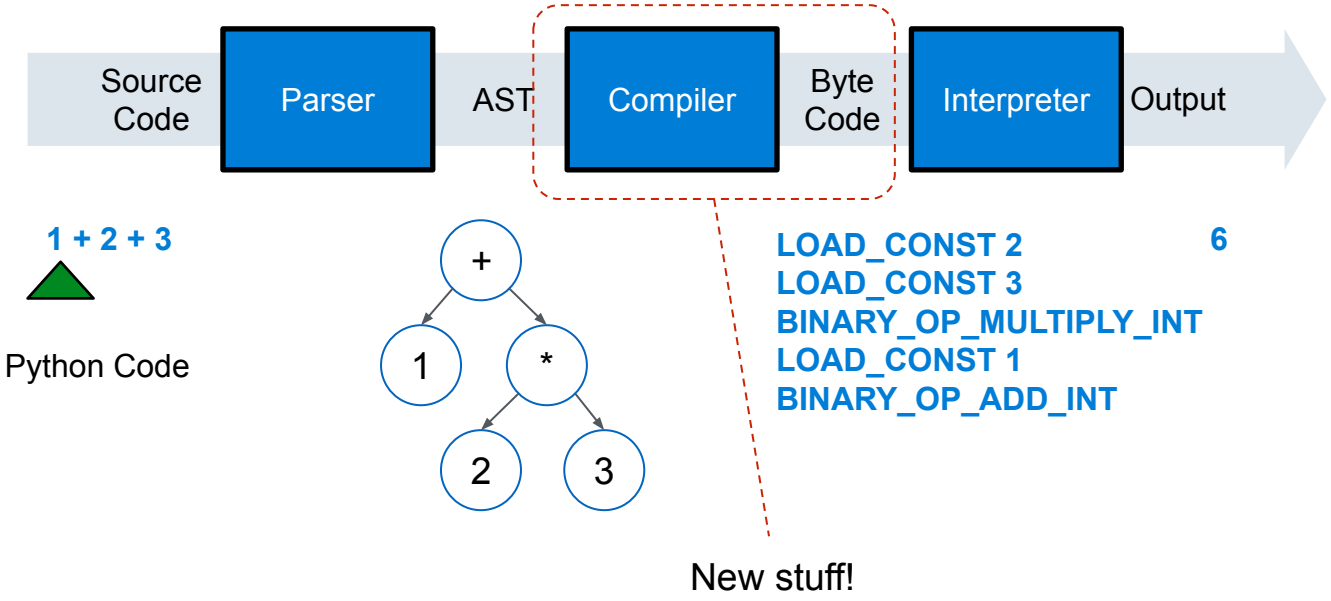


Lies! (In most cases)

CPython Internals

CPython Internals

CPython: runs **byte code** directly producing an output/value, but first compiles source code into byte code



Generating Bytecode

Dis Module

Dis - module for **dis**assembling Python code in Python Bytecode. Analyzes source code, functions, generators, etc. and outputs the Python bytecode for it.

```
>>> import dis
>>> dis.Bytecode("1 + 2 + 3")
>>> for instr in bytecode:
...     print(instr)

>>> dis.dis("x=2")
0 LOAD_CONST 0 (2)
2 STORE_NAME 0 (x)
4 LOAD_CONST 1 (None)
6 RETURN_VALUE
```

(Demo)

Bytecode Optimization

https://github.com/python/cpython/blob/main/Python/ast_opt.c

Interpreting Bytecode

https://github.com/python/cpython/blob/main/Python/bytecodes.c

Stacks

Stack - a data structure for storing and retrieving values. Can only retrieve the most recently added item! Last in first out or LIFO order!

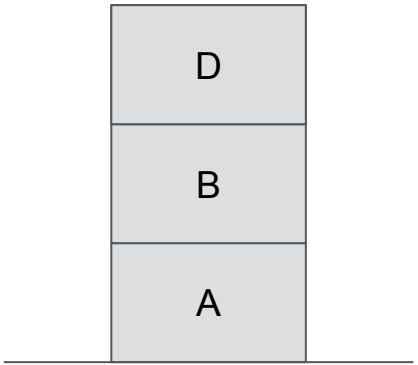
Push - adds an item to the top of the stack

Pop - removes an item from the top of the stack

Peek - looks at the top item of the stack without removing it

We can use a list as a stack!

```
stack = []
stack.append("a")
stack.append("b")
stack.append("c")
stack.pop()
stack.append("d")
```



Stack Machine

Stack Machine - a processor or virtual machine that computes by modifying values in a stack. Has very simple instructions!

Virtual Stack Machine - a stack machine that's simulated using software instead of hardware!

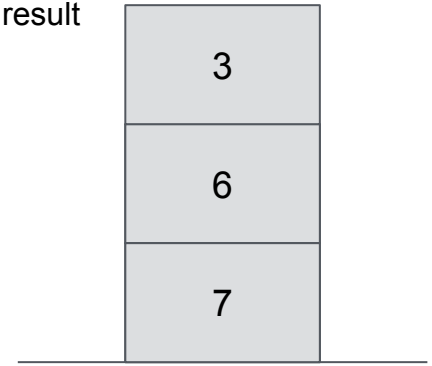
Push - adds an item to the top of the stack

Pop - removes an item from the top of the stack

Operator - combines the top two values in the stack and then pushes the result

```
1 + 2 * 3

stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
stack.multiply()
stack.add()
```



Interpreting Byte Code

(Demo)

Inspired by https://aosabook.org/en/500L/a-python-interpreter-written-in-python.html


scm> (+ 1 2)
>>> Pair("+", Pair(1, Pair(2)))
Wait, it's
all just Python


>>> 1 + 2
'+'
1
2
Wait, it's
all just an AST?


'+'
1
2
LOAD CONST 3
Wait, it's all
just a Python Byte Code?


LOAD_CONST 3
value = GETITEM(FRAME_CO_CONSTS, oparg);
Py_INCREF(value);
Wait, it's
all just C?
Always
has beer


value = 1 + 2;
li t0, 3
Wait, it's
all just assembly?


li t0, 3
00110000000000010010011
Wait, it's all
just machine code?
Always
has been