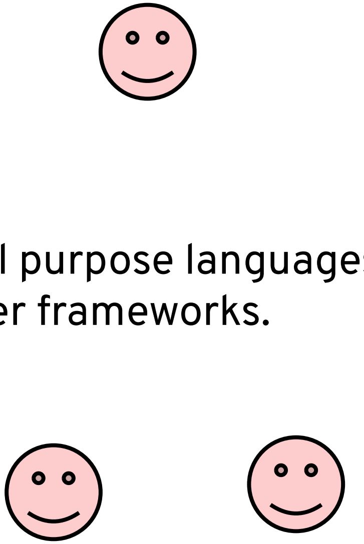
Python already has a frontend for your compiler

Peter Sovietov, RTU MIREA

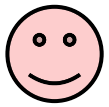
PiterPy
2023

Are compilers written by a few specialists?

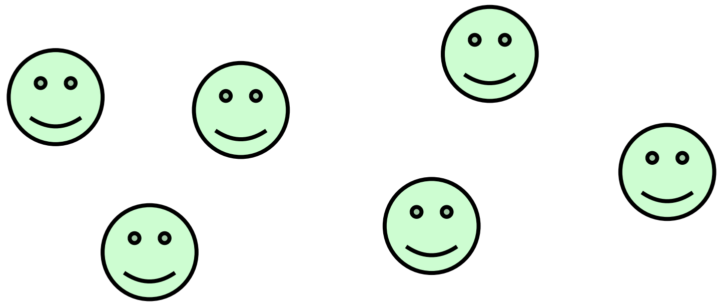


General purpose languages and
compiler frameworks.

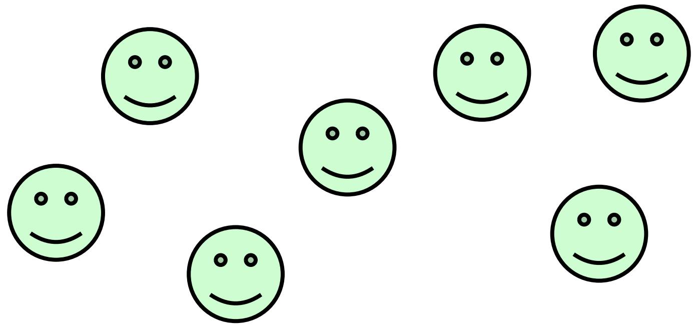
Many people write compilers! Sometimes without knowing it



General purpose languages and compiler frameworks.



- **Domain-specific languages (DSLs) and DSL compilers.**
- **Code visualizers.**
- **Static analyzers.**

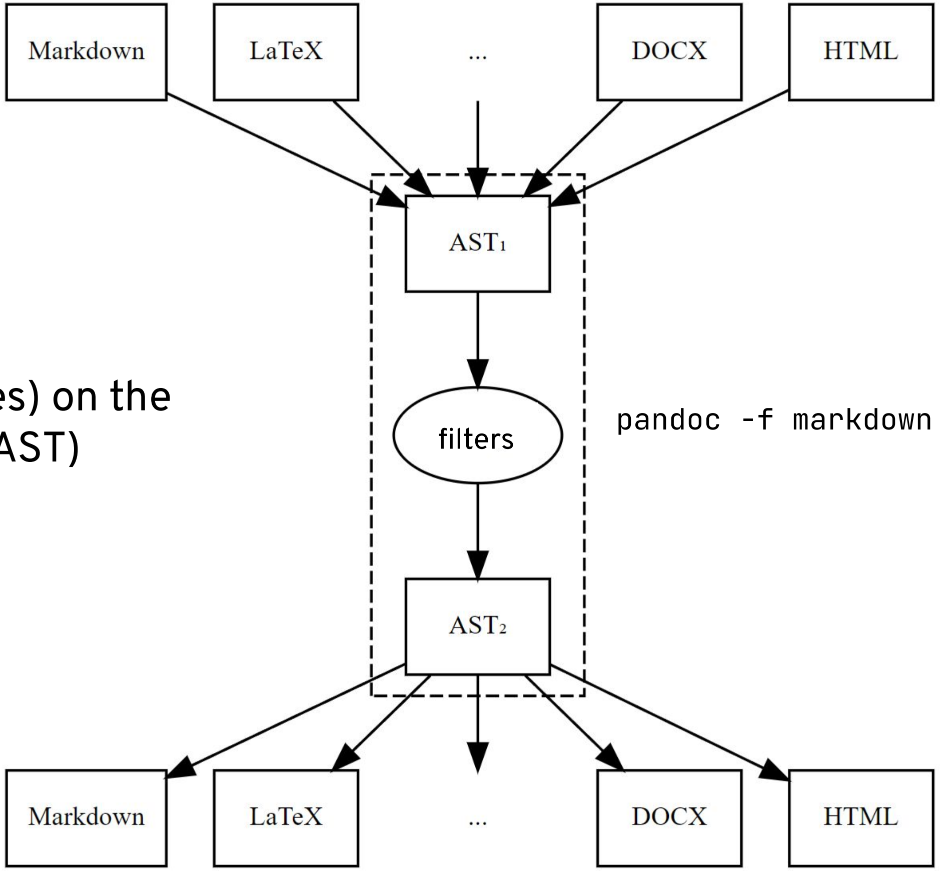


Compilers are everywhere, for example in the Pandoc architecture

Multi-language **frontend**

Transformations (passes) on the **abstract syntax tree** (AST)

Multi-language backend



```
pandoc -f markdown -t html5 -o out.html in.md
```

https://github.com/true-grue/kisscm

What will I talk about

- An approach using the **ast module** and the **match/case** construct to develop DSL compilers, visualizers and static analyzers.
- The approach is illustrated by examples, each of which contains **less than 100 lines of code**.
- I will provide **a link** to the repository.

Some pros of this approach

- Expressive DSL syntax.
- No need for lexical/syntactic analysis.
- Easy handling of syntax errors.
- Pre-build AST for the DSL compiler.
- Easy integration with core Python code.
- Support for highlighting in IDE.

1. **Visitor vs. match/case**
2. Python AST visualizer
3. Graph description compiler
4. Datalog compiler
5. CFG visualizer
6. Check for unused variables
7. PyWasm compiler

AST of arithmetic expression (Visitor vs. match/case)

```python
@dataclass
class Expr:
    pass

@dataclass
class Num(Expr):
    val: int

@dataclass
class Var(Expr):
    name: str

@dataclass
class Add(Expr):
    x: Expr
    y: Expr

@dataclass
class Mul(Expr):
    x: Expr
    y: Expr

...
```

```python
Num = namedtuple('Num', 'val')
Var = namedtuple('Var', 'name')
Add = namedtuple('Add', 'x y')
Mul = namedtuple('Mul', 'x y')
```

… and a base “visitor”

```python
class BaseVisitor:
    def visit(self, tree):
        meth = 'visit_' + type(tree).__name__
        return getattr(self, meth)(tree)
```

Code formatting task

```
>>> tree = Add(Mul(Var('x'), Num(2)), Mul(Var('y'), Num(4)))
>>> print(FormatVisitor().visit(tree))
((x * 2) + (y * 4))
```

Code formatting: comparison of implementations

```
class FormatVisitor(BaseVisitor):
    def visit_Num(self, tree):
        return str(tree.val)

    def visit_Var(self, tree):
        return tree.name

    def visit_Add(self, tree):
        x = self.visit(tree.x)
        y = self.visit(tree.y)
        return f'({x} + {y})'

    def visit_Mul(self, tree):
        x = self.visit(tree.x)
        y = self.visit(tree.y)
        return f'({x} * {y})'
```

```
def format_expr(tree):
    match tree:
        case Num(val) | Var(val):
            return str(val)
        case Add(x, y):
            x = format_expr(x)
            y = format_expr(y)
            return f'({x} + {y})'
        case Mul(x, y):
            x = format_expr(x)
            y = format_expr(y)
            return f'({x} * {y})'
```

Code simplification task

```
>>> tree = Add(Mul(Num(0), Var('x')), Add(Var('y'), Num(0)))
>>> print(FormatVisitor().visit(tree))
((0 * x) + (y + 0))
>>> print(FormatVisitor().visit(SimplifyVisitor().visit(tree)))
y
```

Code simplification: comparison of implementations

```python
class SimplifyVisitor(BaseVisitor):
    def visit_Num(self, tree):
        return tree

    def visit_Var(self, tree):
        return tree

    def visit_Add(self, tree):
        x = self.visit(tree.x)
        y = self.visit(tree.y)
        if isinstance(x, Num) and isinstance(y, Num):
            return Num(x.val + y.val)
        elif isinstance(x, Num) and x.val == 0:
            return y
        elif isinstance(y, Num) and y.val == 0:
            return x
        return Add(x, y)

    def visit_Mul(self, tree):
        x = self.visit(tree.x)
        y = self.visit(tree.y)
        if isinstance(x, Num) and isinstance(y, Num):
            return Num(x.val * y.val)
        elif isinstance(x, Num) and x.val == 0:
            return Num(0)
        elif isinstance(y, Num) and y.val == 0:
            return Num(0)
        return Mul(x, y)
```

```python
def simplify(tree):
    match tree:
        case Add(Num(x), Num(y)):
            return Num(x + y)
        case Mul(Num(x), Num(y)):
            return Num(x * y)
        case Add(Num(0), x) | Add(x, Num(0)):
            return x
        case Mul(Num(0), x) | Mul(x, Num(0)):
            return Num(0)
    return tree

def simplify_expr(tree):
    result = tree
    match tree:
        case Num() | Var():
            result = tree
        case Add(x, y):
            result = Add(simplify_expr(x),
                         simplify_expr(y))
        case Mul(x, y):
            result = Mul(simplify_expr(x),
                         simplify_expr(y))
    return simplify(result)
```

match/case and sum types: another AST definition

```python
from __future__ import annotations
from typing import NamedTuple, assert_never

class Num(NamedTuple):
    val: int

class Var(NamedTuple):
    name: str

class Add(NamedTuple):
    x: Expr
    y: Expr

class Mul(NamedTuple):
    x: Expr
    y: Expr

Expr = Num | Var | Add | Mul
```

match/case and sum types: stack code generation

```python
def compile_expr(tree: Expr) -> str:
    match tree:
        case Num(val) | Var(val):
            return f'PUSH {repr(val)}'
        case Add(a, b):
            x = compile_expr(a)
            y = compile_expr(b)
            return f'{x}\n{y}\nADD'
        case Mul(a, b):
            x = compile_expr(a)
            y = compile_expr(b)
            return f'{x}\n{y}\nMUL'
        case _ as unreachable:
            assert_never(unreachable)
```

← Static checking for exhaustion of alternatives

```
>>> tree = Add(Mul(Var('x'), Num(2)),
... Mul(Var('y'), Num(4)))
>>> print(compile_expr(tree))
PUSH 'x'
PUSH 2
MUL
PUSH 'y'
PUSH 4
MUL
ADD
```

1. Visitor vs. match/case
2. **Python AST visualizer**
3. Graph description compiler
4. Datalog compiler
5. CFG visualizer
6. Check for unused variables
7. PyWasm compiler

The ast module is more popular than it sometimes seems

- **Sphinx:** for generating API documentation from code.
- **Pyflakes:** for analysing code for errors.
- **Coverage:** to analyse code coverage.
- **Pytest:** to replace the normal assert with a more informative version.
- **Pandas:** for parsing queries.
- **Kivy:** to support executing Python code in kv files.
- **PonyORM:** for implementing a query language.

ast module

- AST class definitions are **not available**, they are implemented in C (_ast module). **AST grammar** needs to be checked on regularly: https://docs.python.org/3/library/ast.html
- The ast module has functions for converting text to AST and back, as well as visitor classes for traversing and transforming trees:

```python
class NodeVisitor(object):
    ...
    def visit(self, node):
        """Visit a node."""
        method = 'visit_' + node.__class__.__name__
        visitor = getattr(self, method, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        """Called if no explicit visitor function exists for a node."""
        for field, value in iter_fields(node):
            if isinstance(value, list):
                for item in value:
                    if isinstance(item, AST):
                        self.visit(item)
            elif isinstance(value, AST):
                self.visit(value)

    def visit_Constant(self, node):
    ...
```

ast module: **match/case only!**

- AST class definitions are **not available**, they are implemented in C (_ast module). **AST grammar** needs to be checked on regularly: https://docs.python.org/3/library/ast.html
- The ast module has functions for converting text to AST and back, as well as ~~visitor classes for traversing and transforming trees~~:

```
class NodeVisitor(object):
    ...
    def visit(self, node):
        """Visit a node."""
        method = 'visit_' + node.__class__.__name__
        visitor = getattr(self, method, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        """Called if no explicit visitor function exists for a node."""
        for field, value in iter_fields(node):
            if isinstance(value, list):
                for item in value:
                    if isinstance(item, AST):
                        self.visit(item)
            elif isinstance(value, AST):
                self.visit(value)

    def visit_Constant(self, node):
    ...
```

How to get AST of function

```python
def foo(x):
    return x * 2
```

```
>>> tree = ast.parse(inspect.getsource(foo))
>>> tree
<ast.Module object at 0x00000218E11240A0>
```

How to get AST node field names: _fields

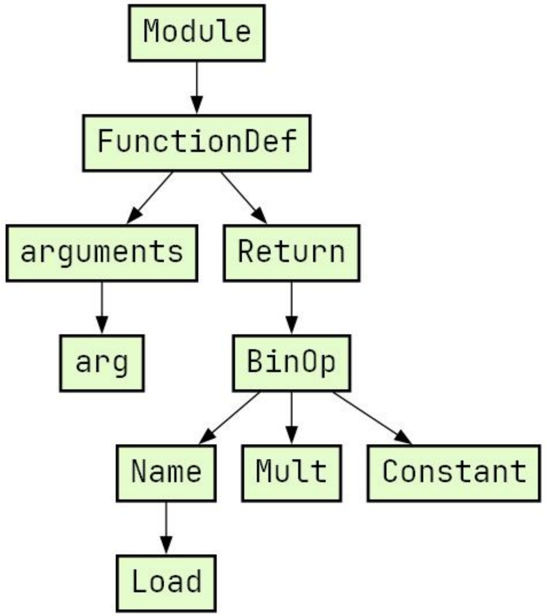
```python
def foo(x):
    return x * 2
```

```
>>> tree = ast.parse(inspect.getsource(foo))
>>> tree._fields
('body', 'type_ignores')
>>> tree = getattr(tree, 'body')
>>> tree
[<ast.FunctionDef object at 0x0000014AB13C8520>]
>>> tree[0]._fields
('name', 'args', 'body', 'decorator_list', 'returns', 'type_comment')
>>> getattr(tree[0], 'body')
[<ast.Return object at 0x0000014AB13C8550>]
```

AST visualizer prototype

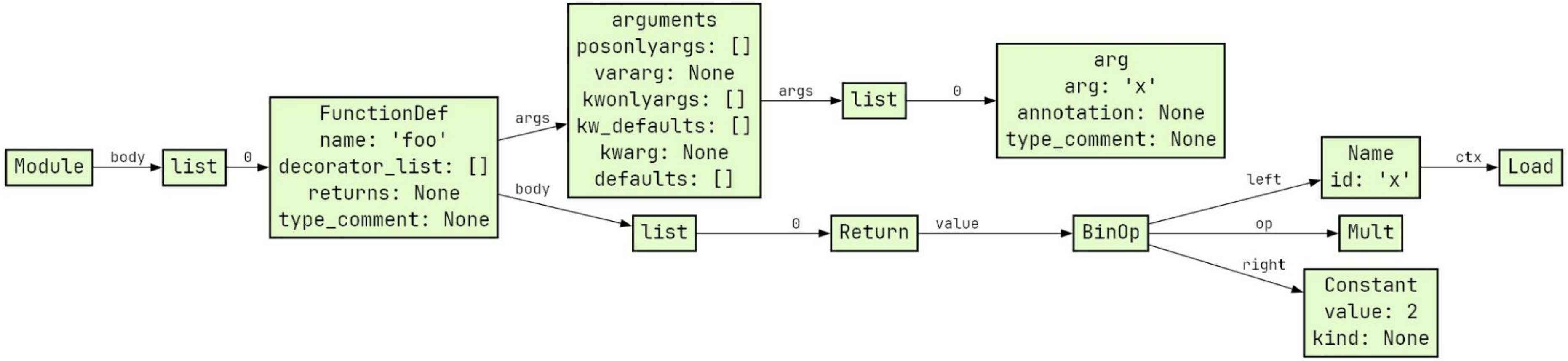
```
def ast_viz(tree):
    graph, labels = {}, {}

    def make_node(tree):
        node_id = len(graph)
        graph[node_id] = []
        labels[node_id] = type(tree).__name__
        return node_id

    def walk(parent_id, tree):
        match tree:
            case ast.AST():                  ← Base AST class
                node_id = make_node(tree)
                graph[parent_id].append(node_id)
                for field in tree._fields:
                    walk(node_id, getattr(tree, field))
            case list():
                for elem in tree:
                    walk(parent_id, elem)

    walk(make_node(tree), tree.body)
    return to_dot(graph, labels)         ← With help of Graphviz
```

```
def foo(x):
    return x * 2
```



Result of the extended version of the visualizer

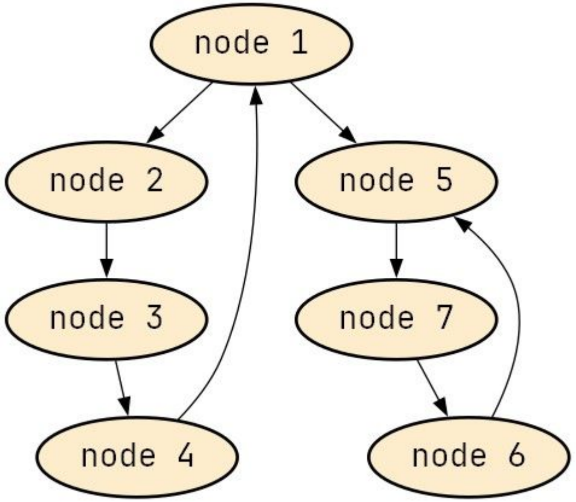
```
def foo(x):
    return x * 2
```



1. Visitor vs. match/case
2. Python AST visualizer
3. **Graph description compiler**
4. Datalog compiler
5. CFG visualizer
6. Check for unused variables
7. PyWasm compiler

Graph description language

```
src = '''
a > b > c > d > a
e < f < g < e < a
a: 'node 1'
b: 'node 2'
c: 'node 3'
d: 'node 4'
e: 'node 5'
f: 'node 6'
g: 'node 7'
'''

print(graph_viz(src))
```

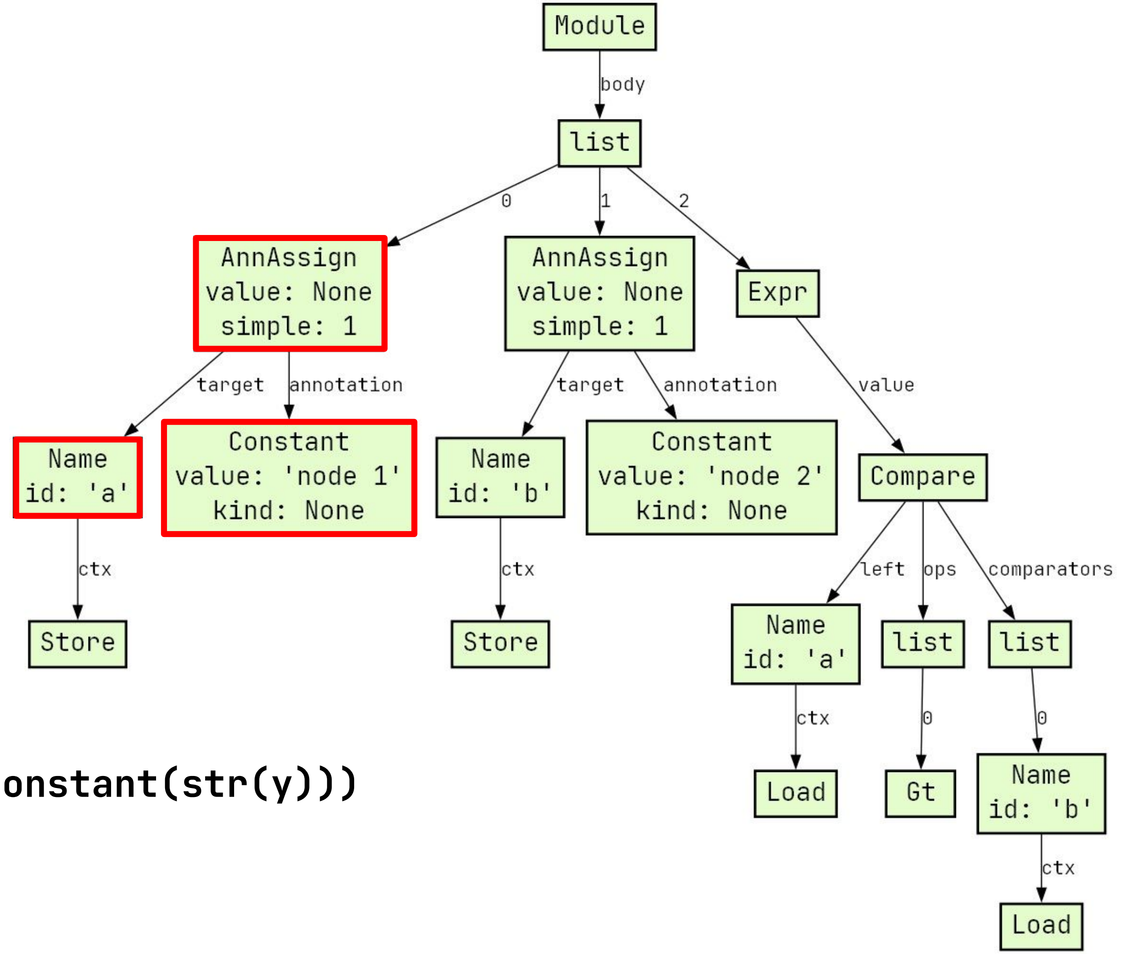


Variables are defined **as they appear** in the text. Graphviz is used as a backend.

AST of the graph description language (1)

```
a: 'node 1'
b: 'node 2'
a > b
```

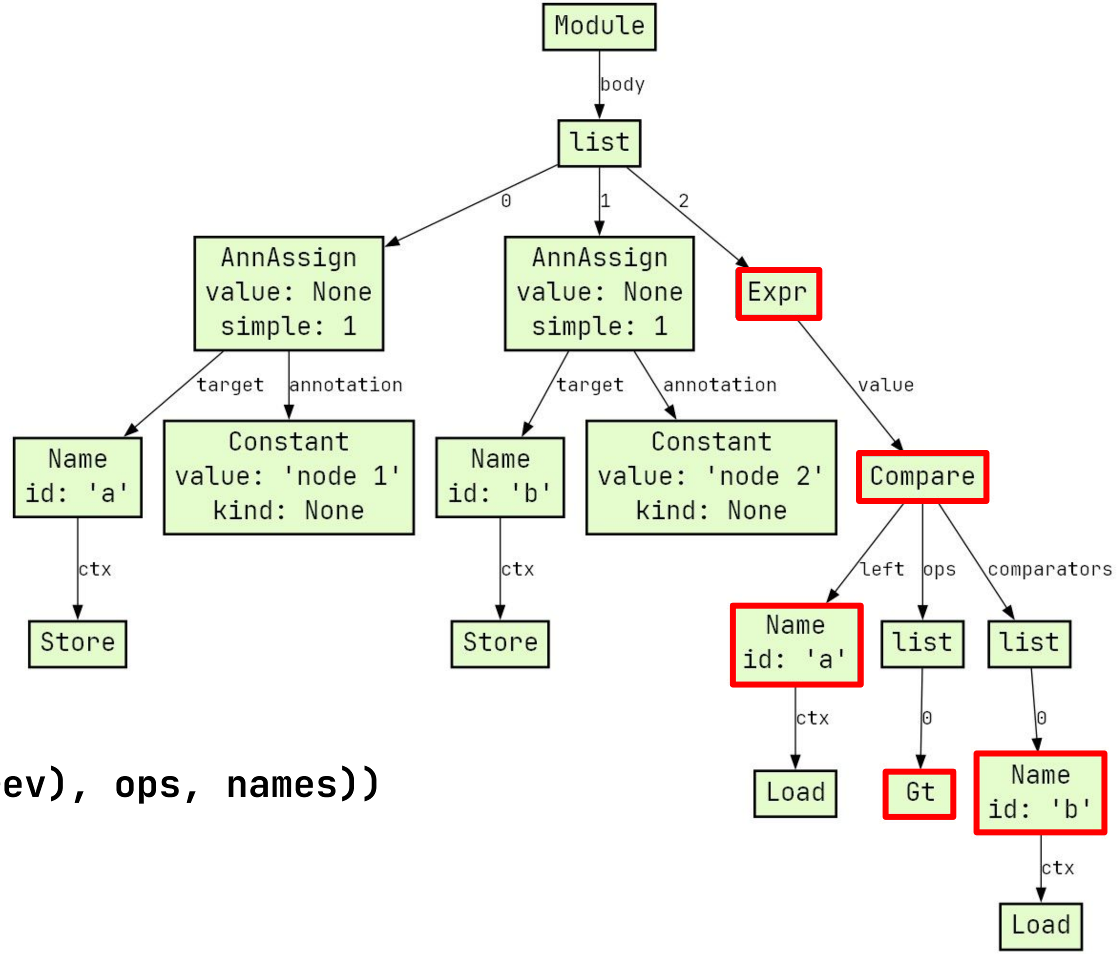
```
AnnAssign(Name(x), Constant(str(y)))
```



AST of the graph description language (2)

```
a: 'node 1'
b: 'node 2'
a > b
```

```
Expr(Compare(Name(prev), ops, names))
```



DSL compiler for graph descriptions: implementation

```python
def add_edges(dot, prev, ops, names):
    for op, name in zip(ops, names):
        match op:
            case ast.Gt():
                dot.append(f'{prev} -> {name.id}')
            case ast.Lt():
                dot.append(f'{name.id} -> {prev}')
        prev = name.id

def graph_viz(src):
    dot = [f'digraph G {{\n{DOT_STYLE}']
    for stmt in ast.parse(src).body:
        match stmt:
            case ast.Expr(ast.Compare(ast.Name(prev), ops, names)) \
                    if all_instances_of(ops, (ast.Gt, ast.Lt)) \
                    and all_instances_of(names, ast.Name):
                add_edges(dot, prev, ops, names)
            case ast.AnnAssign(ast.Name(x), ast.Constant(str(y))):
                dot.append(f'{x} [label="{y}"]')
            case _:
                raise SyntaxError('bad graph syntax',
                                  get_error_details(src, stmt))
    return '\n'.join(dot + ['}'])
```

Example of error handling

```
src = '''
a > b
a == c
'''
```

```
    raise SyntaxError('bad graph syntax',
File "", line 3
a == c
^^^^^^
SyntaxError: bad graph syntax
```

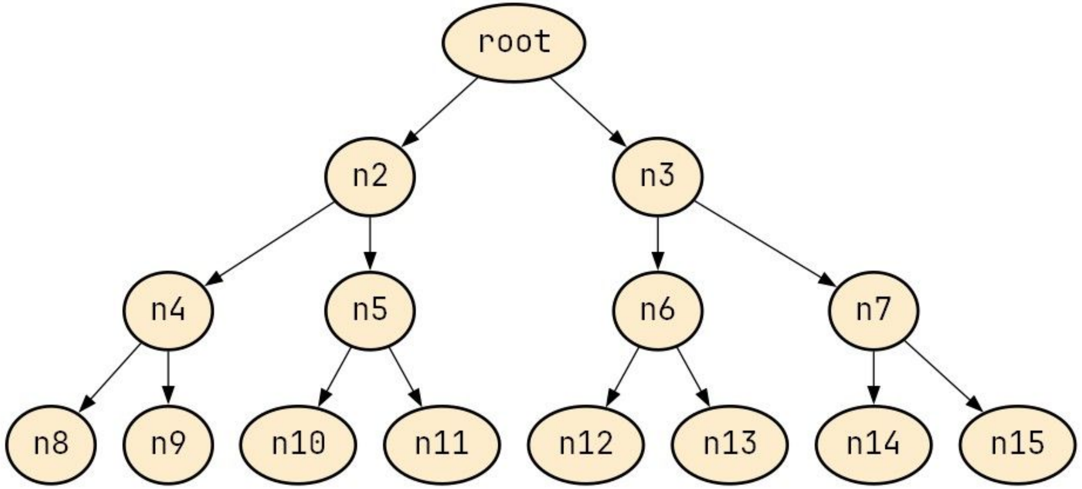
Node attributes and get_error_details()

```
>>> tree = ast.parse(src).body[0]
>>> tree._attributes
('lineno', 'col_offset', 'end_lineno', 'end_col_offset')
```

```
def get_error_details(src, node, filename=''):
    return (filename,
            node.lineno,
            node.col_offset + 1,
            ast.get_source_segment(src, node),
            node.end_lineno,
            node.end_col_offset + 1)      ← Compatible with SyntaxError
```

Example of graph visualization

```
src = '''
n1 > n2 > n4 > n8
n1 > n3 > n6 > n12
n2 > n5 > n10
n3 > n7 > n14
n4 > n9
n5 > n11
n6 > n13
n7 > n15
n1: ‘root’
'''
```



1. Visitor vs. match/case
2. Python AST visualizer
3. Graph description compiler
4. **Datalog compiler**
5. CFG visualizer
6. Check for unused variables
7. PyWasm compiler

Datalog

- Logic DSL, a tiny variant of Prolog.
- A language for databases with recursive query support.
- Main applications: graph databases and static program analysis.

Some implementations: Soufflé, Datomic, μZ as part of the Z3 solver (available for Python).

Datalog: SQL-like example

```
city(1, 'Moscow').
city(2, 'St. Petersburg').
city(3, 'Novosibirsk').
ordered(1, 1).
ordered(1, 2).
ordered(3, 3).
product(1, 'tea').
product(2, 'bread').
product(3, 'flowers').

ship(ProdName, City) IF city(CustNo, City) AND
  ordered(CustNo, ProdNo) AND product(ProdNo, ProdName).
```

← Facts

← Rule. Variables begin with a capital letter.

Datalog: SQL-like example (real syntax)

```
city(1, 'Moscow').
city(2, 'St. Petersburg').
city(3, 'Novosibirsk').
ordered(1, 1).
ordered(1, 2).
ordered(3, 3).
product(1, 'tea').
product(2, 'bread').
product(3, 'flowers').

ship(ProdName, City) ← city(CustNo, City),
  ordered(CustNo, ProdNo), product(ProdNo, ProdName).
```

← Facts

← Rule. Variables begin with a capital letter.

Datalog: SQL-like example (queries)

```
city(1, 'Moscow').
city(2, 'St. Petersburg').
city(3, 'Novosibirsk').
ordered(1, 1).
ordered(1, 2).
ordered(3, 3).
product(1, 'tea').
product(2, 'bread').
product(3, 'flowers').
```

← Facts

```
ship(ProdName, City) ← city(CustNo, City),
  ordered(CustNo, ProdNo), product(ProdNo, ProdName).
```

← Rule. Variables begin with a capital letter.

```
> ship(ProdName, 'Moscow')?
ProdName=tea
ProdName=bread
> ship(ProdName, City)?
ProdName=bread, City=Moscow
ProdName=flowers, City=Novosibirsk
ProdName=tea, City=Moscow
```

← Queries

Datalog: example of a query with negation

```
person(vasya).
person(masha).
loves(vasya, masha).
```

← Facts

```
one_sided_love(X) ← loves(X, Y), not loves(Y, X).
```

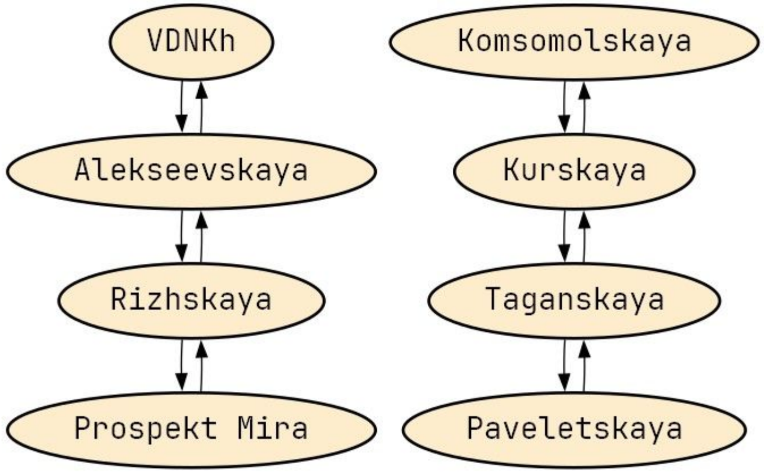
← Rule. Variables begin with a capital letter.

```
> one_sided_love(Who)?
Who=vasya
```

← Query

Datalog: recursive query example

```
links(1, 'VDNKh', 'Alekseevskaya').
links(1, 'Alekseevskaya', 'Rizhskaya').
links(1, 'Rizhskaya', 'Prospekt Mira').
links(2, 'Komsomolskaya', 'Kurskaya').
links(2, 'Kurskaya', 'Taganskaya').
links(2, 'Taganskaya', 'Paveletskaya').

reach(X, Y) ← links(L, X, Y).
reach(X, Y) ← links(L, Y, X).
reach(X, Y) ← reach(X, Z), reach(Z, Y).

> reach('VDNKh', Station)
Station=Rizhskaya
Station=Prospekt Mira
Station=VDNKh
Station=Alekseevskaya
```



Implementing a recursive query in Python/Z3

```python
import z3

fp = z3.Fixedpoint()
fp.set(engine='datalog')

bty = z3.BitVecSort(32)

links = z3.Function('links', bty, bty, bty, z3.BoolSort())
fp.register_relation(links)

fp.add_rule(links(z3.BitVecVal(0, 32), z3.BitVecVal(1, 32), z3.BitVecVal(2, 32)))
fp.add_rule(links(z3.BitVecVal(0, 32), z3.BitVecVal(2, 32), z3.BitVecVal(3, 32)))
fp.add_rule(links(z3.BitVecVal(0, 32), z3.BitVecVal(3, 32), z3.BitVecVal(4, 32)))
fp.add_rule(links(z3.BitVecVal(5, 32), z3.BitVecVal(6, 32), z3.BitVecVal(7, 32)))
fp.add_rule(links(z3.BitVecVal(5, 32), z3.BitVecVal(7, 32), z3.BitVecVal(8, 32)))
fp.add_rule(links(z3.BitVecVal(5, 32), z3.BitVecVal(8, 32), z3.BitVecVal(9, 32)))

X = z3.Const('X', bty)
Y = z3.Const('Y', bty)
Z = z3.Const('Z', bty)
L = z3.Const('L', bty)
Station = z3.Const('Station', bty)
fp.declare_var(X, Y, Z, L)

reach = z3.Function('reach', bty, bty, z3.BoolSort())
fp.register_relation(reach)

fp.add_rule(reach(X, Y), links(L, X, Y))
fp.add_rule(reach(X, Y), links(L, Y, X))
fp.add_rule(reach(X, Y), z3.And(reach(X, Z), reach(Z, Y)))

q = z3.Exists([Station], reach(z3.BitVecVal(1, 32), Station))
print(fp.query(q))
print(fp.get_answer())
```

```
sat
Or(Var(0) == 3, Var(0) == 1, Var(0) == 2, Var(0) == 4)
```

Implementation in Z3: **DSL is needed!**

```python
import z3

fp = z3.Fixedpoint()
fp.set(engine='datalog')

bty = z3.BitVecSort(32)

links = z3.Function('links', bty, bty, bty, z3.BoolSort())
fp.register_relation(links)

fp.add_rule(links(z3.BitVecVal(0, 32), z3.BitVecVal(1, 32), z3.BitVecVal(2, 32)))
fp.add_rule(links(z3.BitVecVal(0, 32), z3.BitVecVal(2, 32), z3.BitVecVal(3, 32)))
fp.add_rule(links(z3.BitVecVal(0, 32), z3.BitVecVal(3, 32), z3.BitVecVal(4, 32)))
fp.add_rule(links(z3.BitVecVal(5, 32), z3.BitVecVal(6, 32), z3.BitVecVal(7, 32)))
fp.add_rule(links(z3.BitVecVal(5, 32), z3.BitVecVal(7, 32), z3.BitVecVal(8, 32)))
fp.add_rule(links(z3.BitVecVal(5, 32), z3.BitVecVal(8, 32), z3.BitVecVal(9, 32)))

X = z3.Const('X', bty)
Y = z3.Const('Y', bty)
Z = z3.Const('Z', bty)
L = z3.Const('L', bty)
Station = z3.Const('Station', bty)
fp.declare_var(X, Y, Z, L)

reach = z3.Function('reach', bty, bty, z3.BoolSort())
fp.register_relation(reach)

fp.add_rule(reach(X, Y), links(L, X, Y))
fp.add_rule(reach(X, Y), links(L, Y, X))
fp.add_rule(reach(X, Y), z3.And(reach(X, Z), reach(Z, Y)))

q = z3.Exists([Station], reach(z3.BitVecVal(1, 32), Station))
print(fp.query(q))
print(fp.get_answer())
```

```
sat
Or(Var(0) == 3, Var(0) == 1, Var(0) == 2, Var(0) == 4)
```

DSL compiler Datalog: facts and rules

```
@datalog
def metro():
    links(1, 'VDNKh', 'Alekseevskaya')
    links(1, 'Alekseevskaya', 'Rizhskaya')
    links(1, 'Rizhskaya', 'Prospekt Mira')
    links(2,  'Komsomolskaya', 'Kurskaya')
    links(2, 'Kurskaya', 'Taganskaya')
    links(2, 'Taganskaya', 'Paveletskaya')

    reach(X, Y) <= links(L, X, Y)
    reach(X, Y) <= links(L, Y, X)
    reach(X, Y) <= reach(X, Z), reach(Z, Y)
```

The **decorator** is used with the call ast.parse(inspect.getsource(func)).

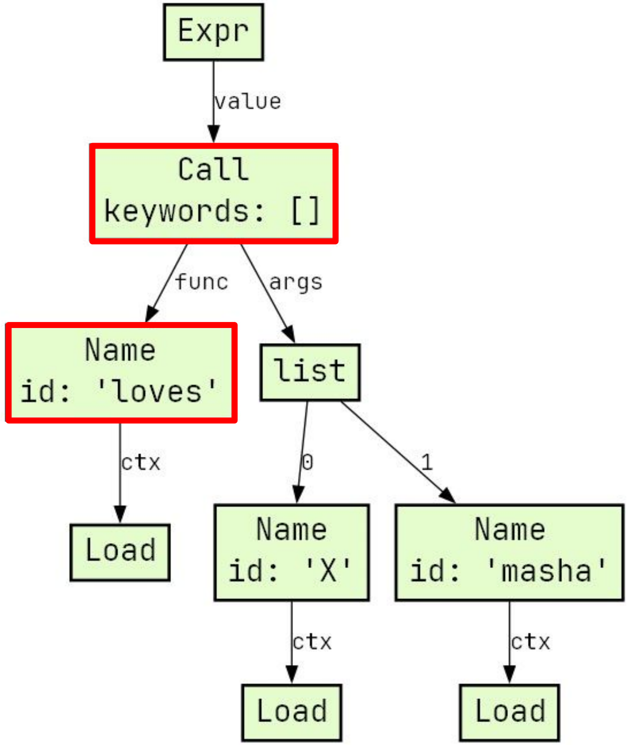
DSL compiler Datalog: query

```
@datalog
def metro():
    links(1, 'VDNKh', 'Alekseevskaya')
    links(1, 'Alekseevskaya', 'Rizhskaya')
    links(1, 'Rizhskaya', 'Prospekt Mira')
    links(2,  'Komsomolskaya', 'Kurskaya')
    links(2, 'Kurskaya', 'Taganskaya')
    links(2, 'Taganskaya', 'Paveletskaya')

    reach(X, Y) <= links(L, X, Y)
    reach(X, Y) <= links(L, Y, X)
    reach(X, Y) <= reach(X, Z), reach(Z, Y)
```

```
>>> _, rows = metro().query('reach("VDNKh", Station)')
>>> pprint(rows)
[{'Station': 'Rizhskaya'},
 {'Station': 'VDNKh'},
 {'Station': 'Prospekt Mira'},
 {'Station': 'Alekseevskaya'}]
```

Atom syntax: fact, subgoal or query

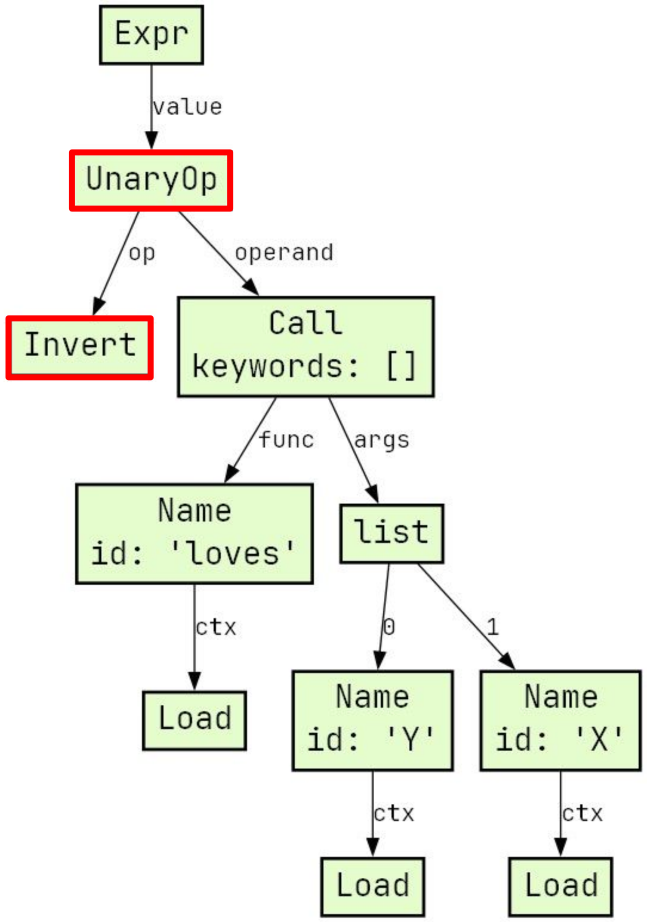
```
"loves(X, masha)"

Call(Name(name), args)
```



Variables begin with a capital letter.

Syntax of atom with negation

```
"~loves(Y, X)"

UnaryOp(Invert(), atom)
```

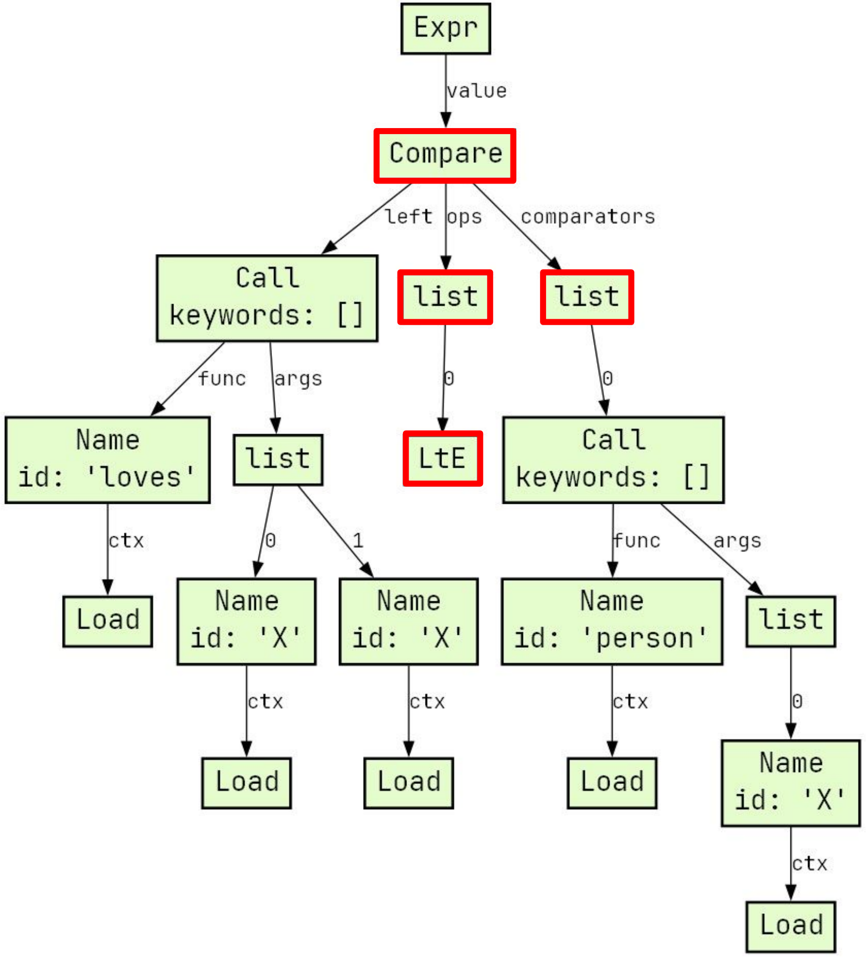


Atom syntax with negation: why not with “not”?

```
>>> f(X) <= not g(X)
  File "<stdin>", line 1
    f(X) <= not g(X)
            ^^^
SyntaxError: invalid syntax
```

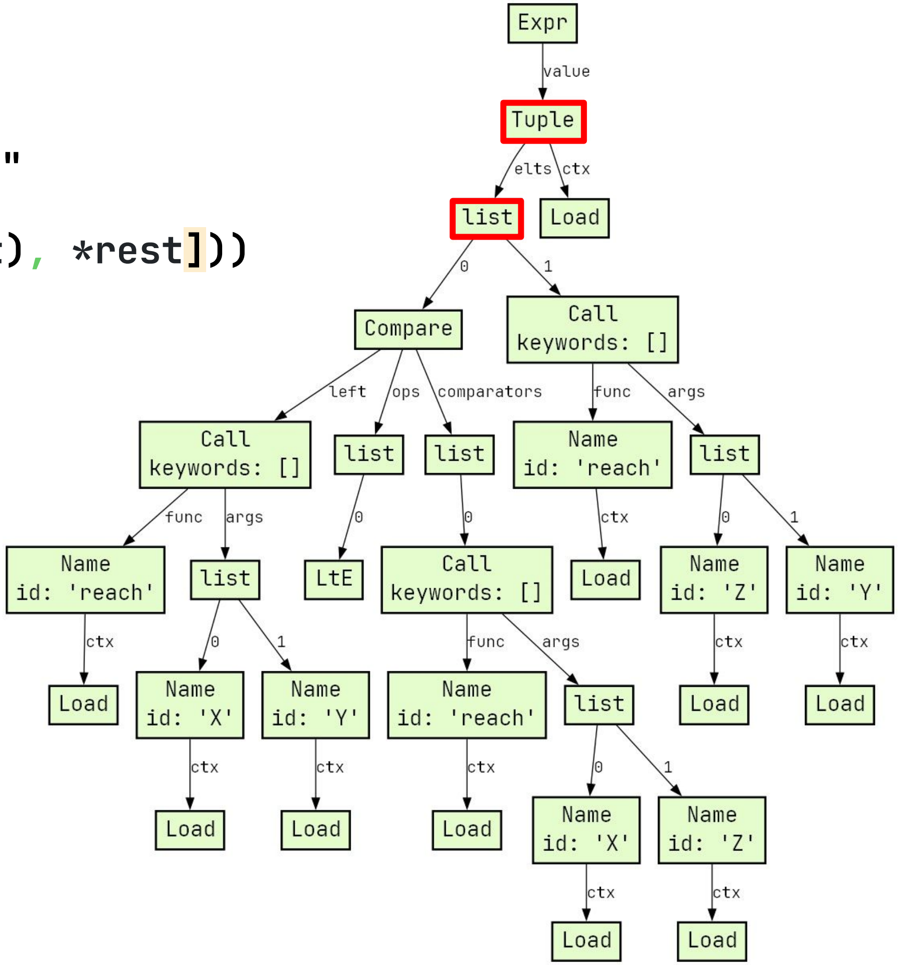
Rule syntax

```
"loves(X, X) <= person(X)"

Expr(Compare(head, [LtE()], [first]))
```



Long rule syntax: comma "overload"

```
"reach(X, Y) <= reach(X, Z), reach(Z, Y)"

Expr(Tuple([Compare(head, [LtE()], first), *rest]))
```



Terms: variables and values

```
def compile_term(self, term):
    match term:
        case ast.Name(name) if name[0].isupper():
            return self.get_var(name)
        case ast.Name(value) | ast.Constant(value):
            return self.get_value(value)
```

Values

The values in Z3/Datalog engine are only **bit vectors.**
So I map each **Python value** to a **number in the hash table**.

```
self.val_to_idx = {}
self.idx_to_val = {}
...
def get_value(self, value):
    if value not in self.val_to_idx:
        self.val_to_idx[value] = len(self.val_to_idx)
        self.idx_to_val[self.val_to_idx[value]] = value
    return z3.BitVecVal(self.val_to_idx[value], BV_SIZE)
```

I'll be back to Datalog soon!

1. Visitor vs. match/case
2. Python AST visualizer
3. Graph description compiler
4. Datalog compiler
5. **CFG visualizer**
6. Check for unused variables
7. PyWasm compiler

CFG — the basis for serious static analysis

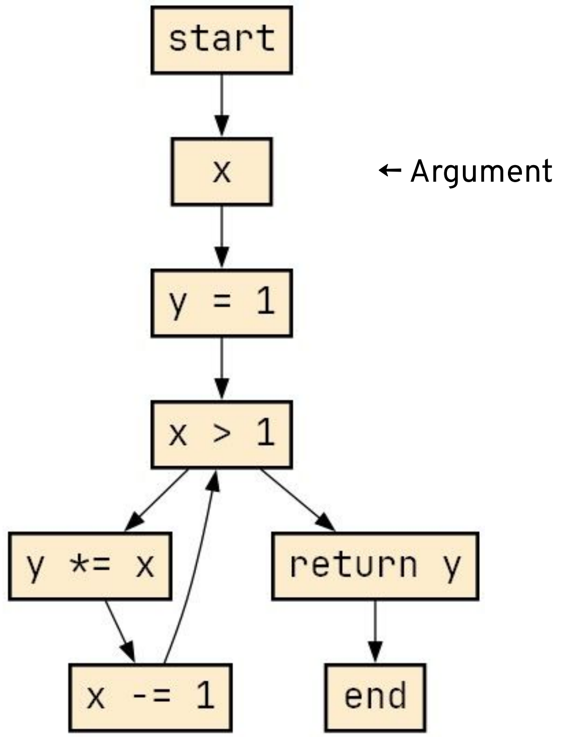
CFG — **Control Flow Graph**.

In CFG, **nodes are operators**, and **edges are transitions** (or jumps) between operators.

By the way, CPython also builds this graph, but it is not available to the Python programmer. It's implemented in C: https://devguide.python.org/internals/

Example of a CFG

```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```



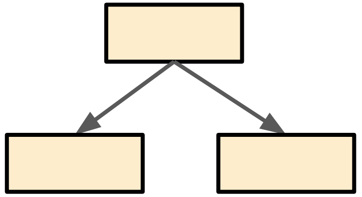
How to build a CFG? Walk an AST!

Operators can be chained together: one by one.

But what about, for example, the if operator, which has **two** branches of execution?

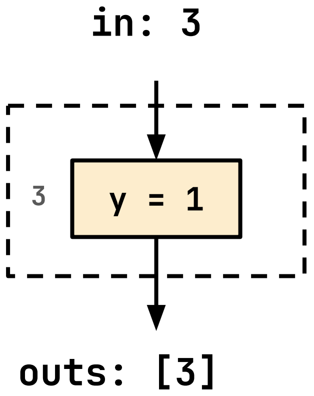
Let each operator have:

- one input node (**in**),
- set of output nodes (**outs**).



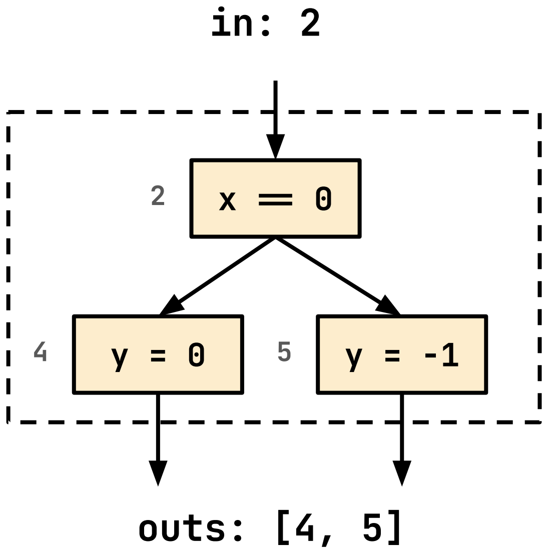
In/outs in the process of building an operator if (1)

```
if x > 0:
    y = 1
elif x == 0:
    y = 0
else:
    y = -1
```



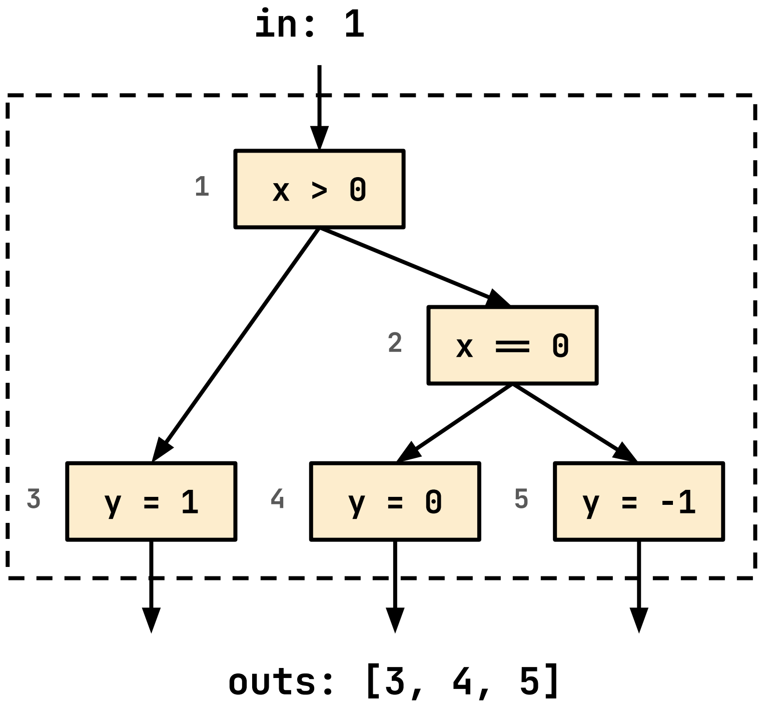
In/outs in the process of building an operator if (2)

```
if x > 0:
    y = 1
elif x == 0:
    y = 0
else:
    y = -1
```



In/outs in the process of building an operator if (3)

```
if x > 0:
    y = 1
elif x == 0:
    y = 0
else:
    y = -1
```



Walk a CFG

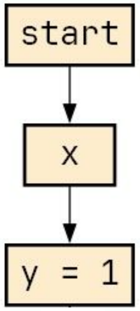
```
# Provided by the user

class Graph:
    def node(self, node):
        ...

    def edge(self, src, dst):
        ...
```

```
>>> g = Graph()
>>> walk_cfg(g, ast.parse(src))
```

```
# walk_cfg() implementation

def add_node(graph, node):
    graph.node(node)
    return node, [node]

def connect(graph, outs, node):
    for out in outs:
        graph.edge(out, node)
...
def walk_cfg(graph, tree):
    for stmt in tree.body:
        match stmt:
    ...
```

walk_while() implementation (1)

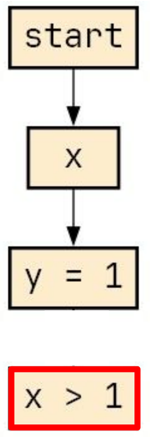
```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```



```
def walk_while(graph, test, body):
    test_in, test_outs = add_node(graph, test)
    body_in, body_outs = walk_block(graph, body)
    connect(graph, test_outs, body_in)
    connect(graph, body_outs, test_in)
    return test_in, test_outs
```

walk_while() implementation (2)

```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```

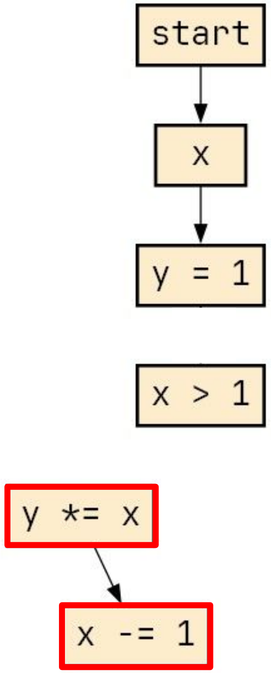


```
def walk_while(graph, test, body):
    test_in, test_outs = add_node(graph, test)
    body_in, body_outs = walk_block(graph, body)
    connect(graph, test_outs, body_in)
    connect(graph, body_outs, test_in)
    return test_in, test_outs
```

walk_while() implementation (3)

```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```

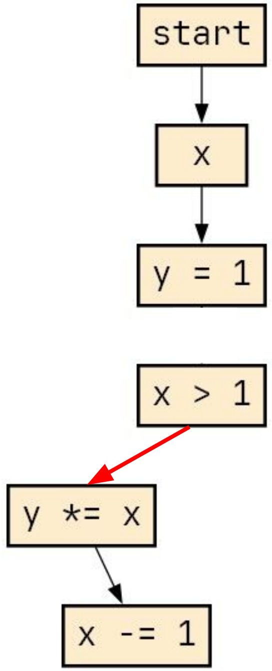
```
def walk_while(graph, test, body):
    test_in, test_outs = add_node(graph, test)
    body_in, body_outs = walk_block(graph, body)
    connect(graph, test_outs, body_in)
    connect(graph, body_outs, test_in)
    return test_in, test_outs
```



walk_while() implementation (4)

```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```

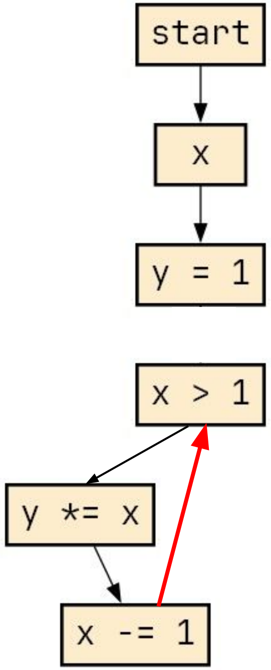
```
def walk_while(graph, test, body):
    test_in, test_outs = add_node(graph, test)
    body_in, body_outs = walk_block(graph, body)
    connect(graph, test_outs, body_in)
    connect(graph, body_outs, test_in)
    return test_in, test_outs
```



walk_while() implementation (5)

```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```

```
def walk_while(graph, test, body):
    test_in, test_outs = add_node(graph, test)
    body_in, body_outs = walk_block(graph, body)
    connect(graph, test_outs, body_in)
    connect(graph, body_outs, test_in)
    return test_in, test_outs
```



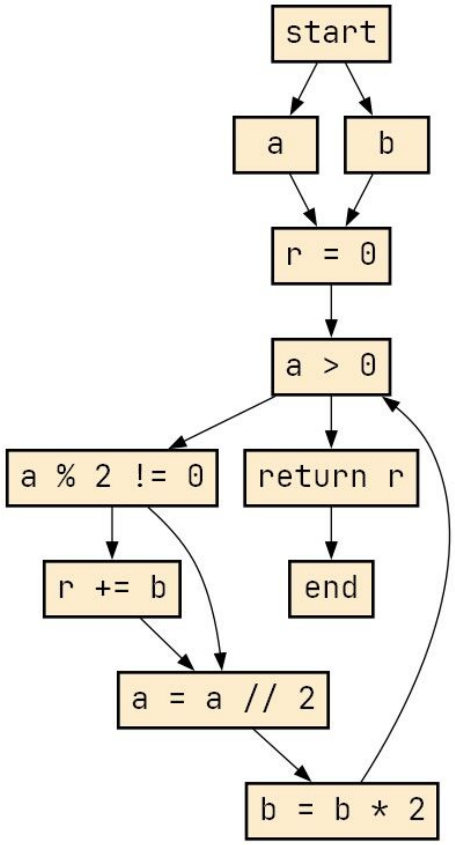
CFG Visualizer

```
class CFGViz:
    def __init__(self):
        self.dot = [f'digraph G {{\n{DOT_STYLE}']

    def node(self, node):
        label = node if node in ('start', 'end') else ast.unparse(node)
        self.dot.append(f'{id(node)} [label="{label}" shape=box]')

    def edge(self, src, dst):
        self.dot.append(f'{id(src)} -> {id(dst)}')

    def to_dot(self):
        return '\n'.join(self.dot + ['}'])
```

```
>>> g = CFGViz()
>>> walk_cfg(g, ast.parse(src))
>>> print(g.to_dot())
...
```

CFG visualization example

```python
def mult(a, b):
    r = 0
    while a > 0:
        if a % 2 ≠ 0:
            r += b
        a = a // 2
        b = b * 2
    return r
```



1. Visitor vs. match/case
2. Python AST visualizer
3. Graph description compiler
4. Datalog compiler
5. CFG visualizer
6. **Check for unused variables**
7. PyWasm compiler

Are there unused variables here?

```python
def foo(a, b, c):
    x = 0
    if a:
        a = 0
        x = 1
    else:
        x = 2
    a = 1
    b = 2
    return x
```

Static analysis result

```
 1: def foo(a, b, c):
 2:     x = 0
 3:     if a:
 4:         a = 0
 5:         x = 1
 6:     else:
 7:         x = 2
 8:     a = 1
 9:     b = 2
10:     return x
```

```
Dead assignment to 'a', line 8
Dead assignment to 'b', line 9
Dead assignment to 'c', line 1
Dead assignment to 'b', line 1
Dead assignment to 'a', line 4
Dead assignment to 'x', line 2
```

Almost everything is ready to implement such a search!

Formulate rules for finding unused variables on Datalog.

1. Traverse the CFG and **collect facts** about variables in the form of a database for Datalog.
2. **Make a query** to Datalog DB.

Rules for "live" variables (1)

1. Variable V is "live" **before** (live in) a P statement if it is used in that statement:

   ```
   live_in(P, V) <= used(P, V)
   ```

Rules for "live" variables (2)

1. Variable V is "live" **before** (live in) a P statement if it is used in that statement:

   ```
   live_in(P, V) <= used(P, V)
   ```

2. Variable V is "live" **before** operator P if it is not overwritten in P and it is "live" **after** operator P:

   ```
   live_in(P, V) <= ~defined(P, V), live_out(P, V)
   ```

Rules for "live" variables (3)

1. Variable V is "live" before (live in) a P statement if it is used in that statement:

   ```
   live_in(P, V) <= used(P, V)
   ```

2. Variable V is "live" **before** operator P if it is not overwritten in P and it is "live" **after** operator P:

   ```
   live_in(P, V) <= ~defined(P, V), live_out(P, V)
   ```

3. Variable V is "live" **after** (live out) statement P1 if there is a transition from P1 to P2 and this variable is "live" **before** statement P2:

   ```
   live_out(P1, V) <= edge(P1, P2), live_in(P2, V)
   ```

So what to do with unused variables?

Variable V is "dead" in operator P if it is defined in P but not "live" **after** P:

```
dead_var(P, V) <= defined(P, V), ~live_out(P, V)
```

Rules for unused variables in DSL

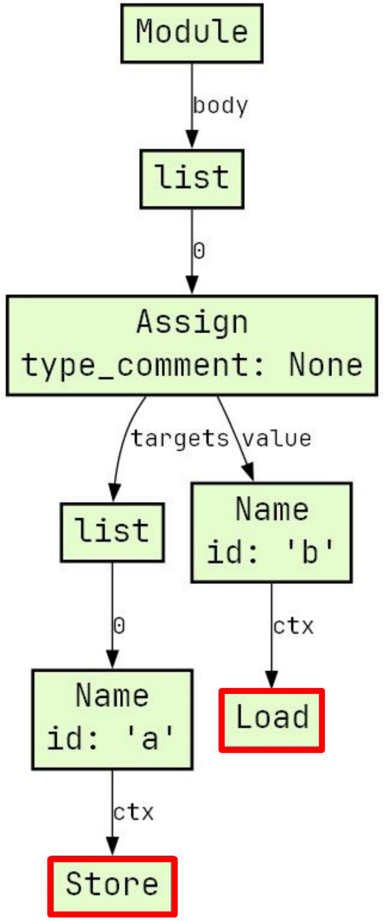
```
@datalog
def dead_var():
    live_in(P, V) <= used(P, V)
    live_in(P, V) <= ~defined(P, V), live_out(P, V)
    live_out(P1, V) <= edge(P1, P2), live_in(P2, V)
    dead_var(P, V) <= defined(P, V), ~live_out(P, V)
```

Implementation of static analysis

```
class CFGAnalysis:
    def __init__(self):
        self.dlog = dead_var()

    def node(self, node):
        if node not in ('start', 'end'):
            defs, uses = get_du(node, [], [])
            for d in defs:
                self.dlog.add_fact('defined', node, d)
            for u in uses:
                self.dlog.add_fact('used', node, u)

    def edge(self, src, dst):
        self.dlog.add_fact('edge', src, dst)

    def get_dead_vars(self):
        _, dead_vars = self.dlog.query('dead_var(Node, Var)')
        return [(row['Var'], row['Node']) for row in dead_vars]
```

How to find defs and uses: check ctx

```
a = b
```

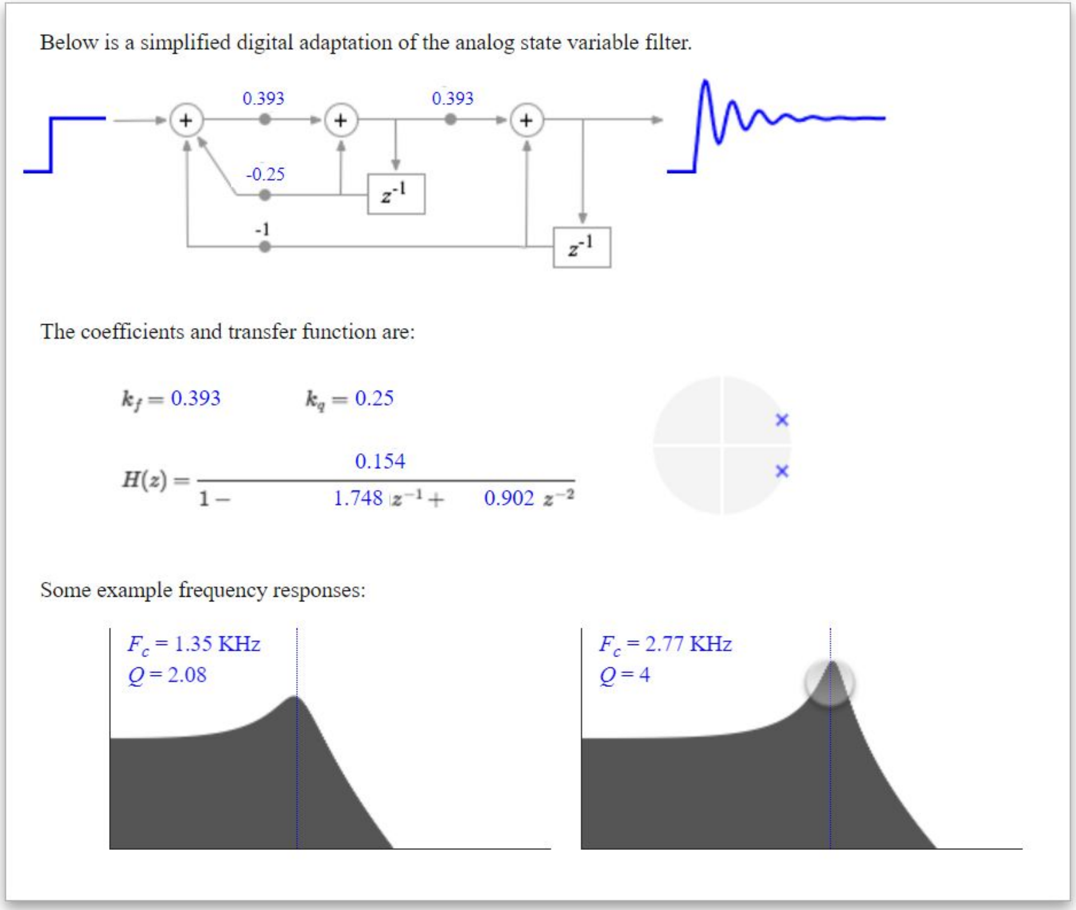


Implementation of get_du()

```
def get_du(node, defs, uses):
    match node:
        case ast.Name(name, ast.Load()):
            uses.append(name)
        case ast.Name(name, ast.Store()) | ast.arg(name):
            defs.append(name)
        case ast.AST():
            for field in node._fields:
                defs, uses = get_du(getattr(node, field), defs, uses)
        case list():
            for elem in node:
                defs, uses = get_du(elem, defs, uses)
    return defs, uses
```

1. Visitor vs. match/case
2. Python AST visualizer
3. Graph description compiler
4. Datalog compiler
5. CFG visualizer
6. Check for unused variables
7. **PyWasm compiler**

Explorable explanations



Below is a simplified digital adaptation of the analog state variable filter.

The coefficients and transfer function are:

$k_f = 0.393$ $k_q = 0.25$

$$H(z) = \frac{0.154}{1 - 1.748\ z^{-1} + 0.902\ z^{-2}}$$

Some example frequency responses:

http://worrydream.com/ExplorableExplanations/#explorableExample

A compiler of interactive visualizations to Wasm

Requirements:

- An expressive enough **subset of Python** that can be used for prototyping in **Jupyter/Matplotlib**.
- The **performance** of the generated code is close to **JavaScript**.
- Compiled **Wasm modules** should take **hundreds of bytes**, not tens of megabytes.
- Compiler implementation in **<100 lines** of code.

PyWasm compiler: implementation details

- **Only** values of **float type** (64 bits) are supported.
- **Lists** are treated **separately** (see next slide)**.**
- **If**, **while**, and **functions** are supported. The **for** loop is **excluded** due to the <100 lines of code limitation.
- Code generation is the same as the **stack code generation** example at the beginning of this talk.

Lists are just indexes in a JS mem table

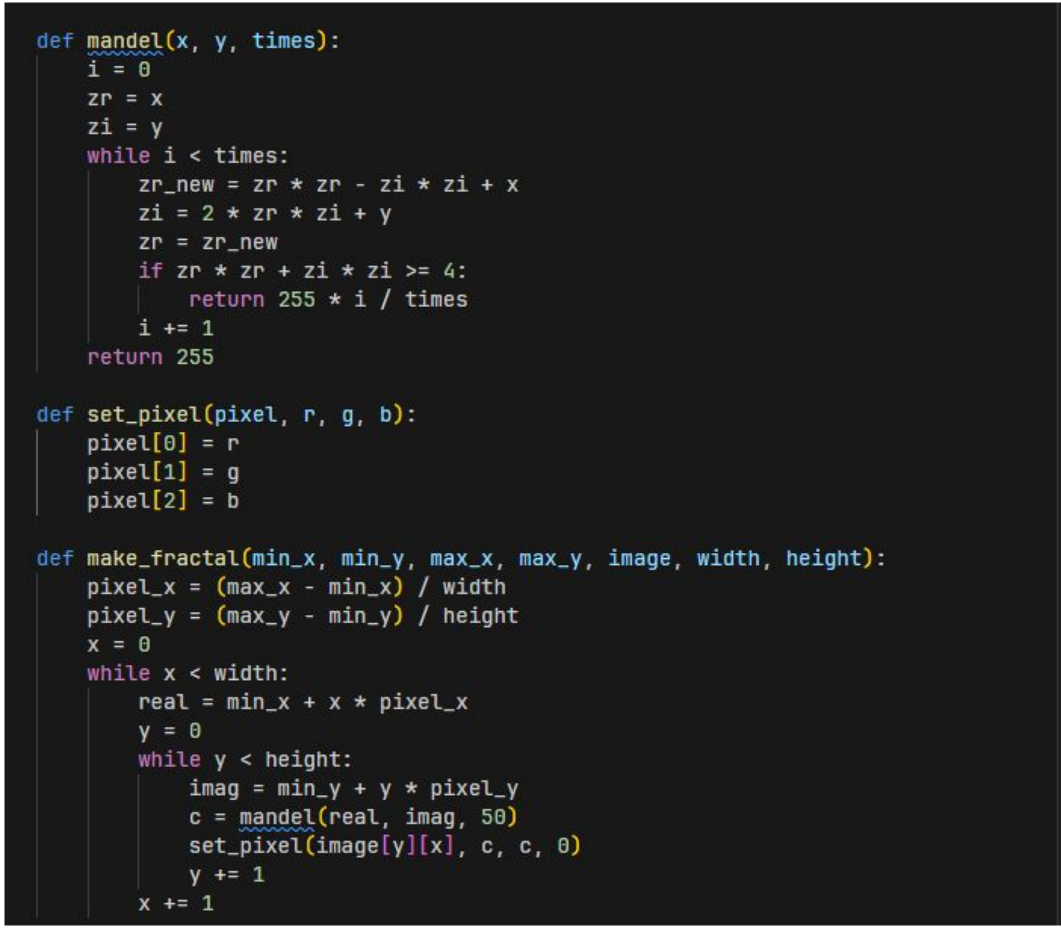
```python
case ast.List([]):
    return f'call $list'

case ast.Expr(ast.Call(ast.Attribute(name, 'append'),
                       [value])):
    name = compile_expr(env, name)
    value = compile_expr(env, value)
    return f'{name}\n{value}\ncall $append'

case ast.Subscript(name, slice=slice):
    name = compile_expr(env, name)
    slice = compile_expr(env, slice)
    return f'{name}\n{slice}\ncall $get'

case ast.Assign([ast.Subscript(name, slice=slice)], expr):
    name = compile_expr(env, name)
    slice = compile_expr(env, slice)
    expr = compile_expr(env, expr)
    return f'{name}\n{slice}\n{expr}\ncall $set'
```

Python

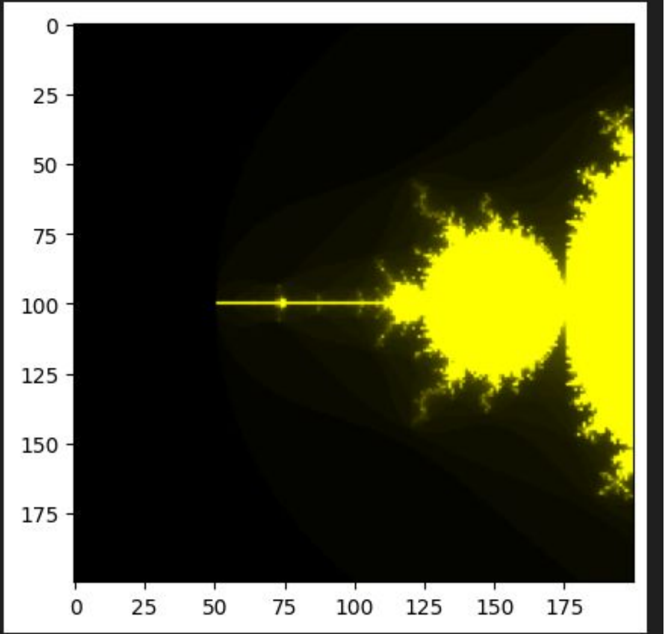
```javascript
var mem = [];
var lib = {
    list: function () {
        mem.push([]);
        return mem.length - 1;
    },
    append: function (n, x) {
        mem[n].push(x);
    },
    get: function (n, i) {
        return mem[n][i];
    },
    set: function (n, i, x) {
        mem[n][i] = x;
    }
};
```

JavaScript

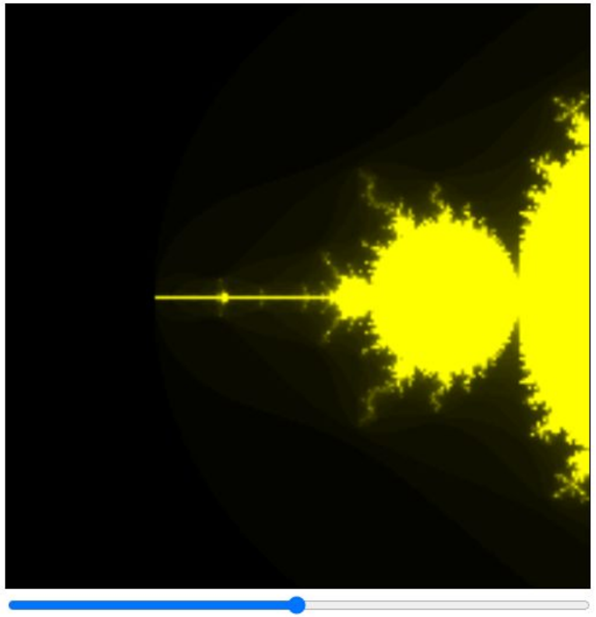
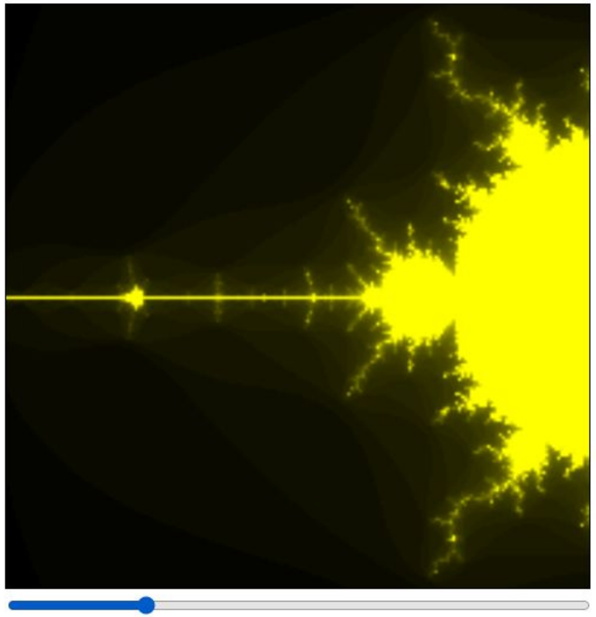
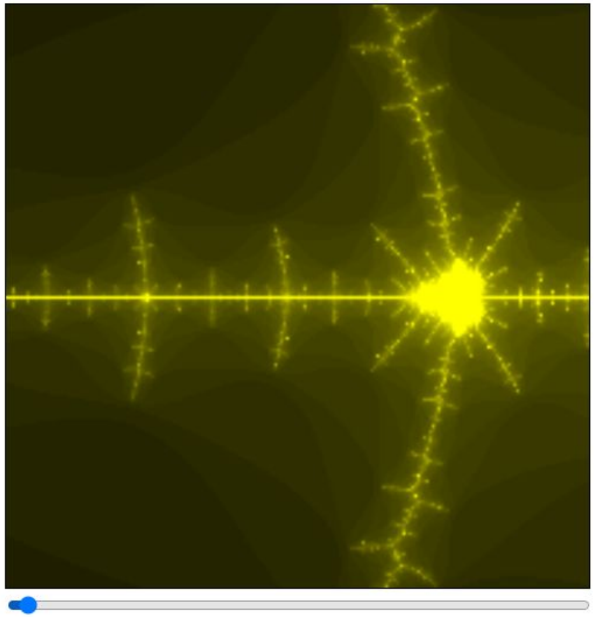
Prototype of fractal visualization in Jupyter Notebook



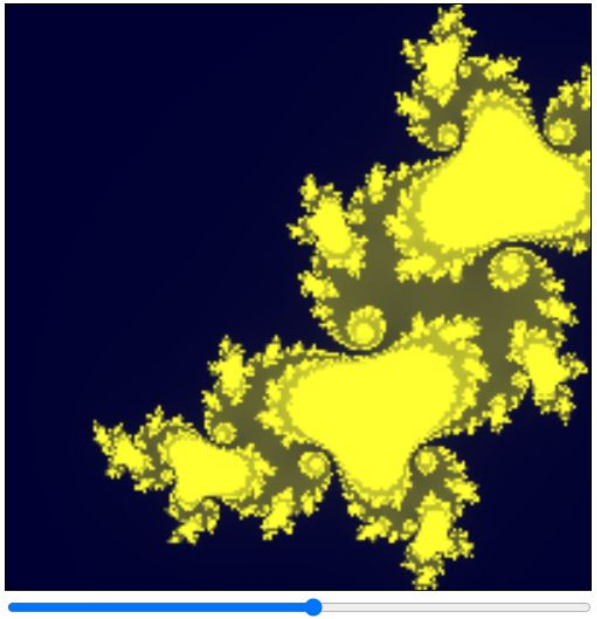
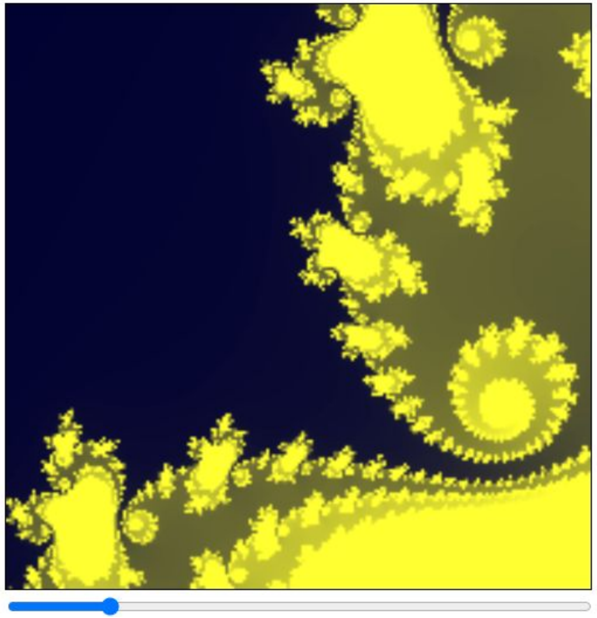
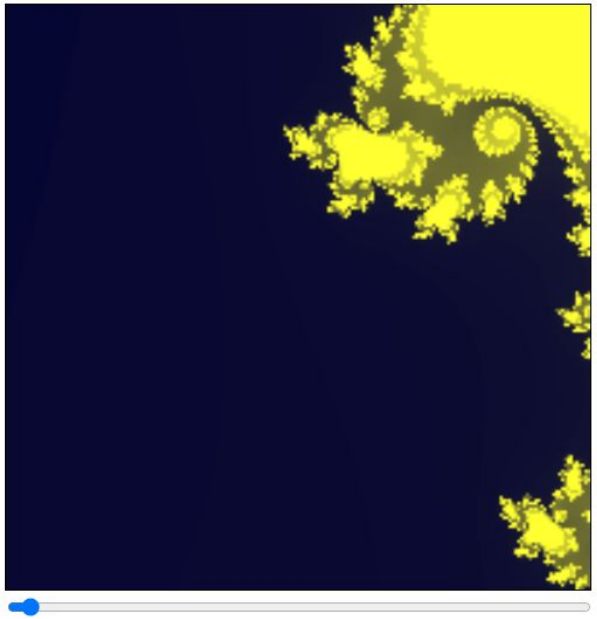
```python
def mandel(x, y, times):
    i = 0
    zr = x
    zi = y
    while i < times:
        zr_new = zr * zr - zi * zi + x
        zi = 2 * zr * zi + y
        zr = zr_new
        if zr * zr + zi * zi >= 4:
            return 255 * i / times
        i += 1
    return 255

def set_pixel(pixel, r, g, b):
    pixel[0] = r
    pixel[1] = g
    pixel[2] = b

def make_fractal(min_x, min_y, max_x, max_y, image, width, height):
    pixel_x = (max_x - min_x) / width
    pixel_y = (max_y - min_y) / height
    x = 0
    while x < width:
        real = min_x + x * pixel_x
        y = 0
        while y < height:
            imag = min_y + y * pixel_y
            c = mandel(real, imag, 50)
            set_pixel(image[y][x], c, c, 0)
            y += 1
        x += 1
```



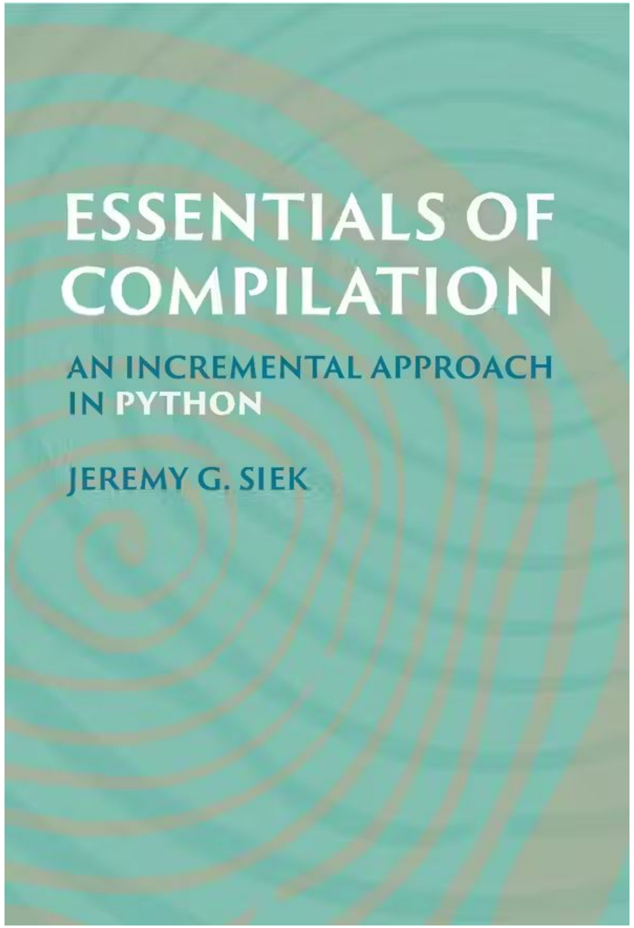
Web visualization in PyWasm: Mandelbrot fractal

Web visualization in PyWasm: Julia fractal

What's new to read about compiler development


ESSENTIALS OF COMPILATION

AN INCREMENTAL APPROACH IN PYTHON

JEREMY G. SIEK

Thanks for your attention!

Repository with all examples:
https://github.com/true-grue/python-dsls