В Python есть готовый фронтенд для вашего компилятора

Пётр Советов, РТУ МИРЭА

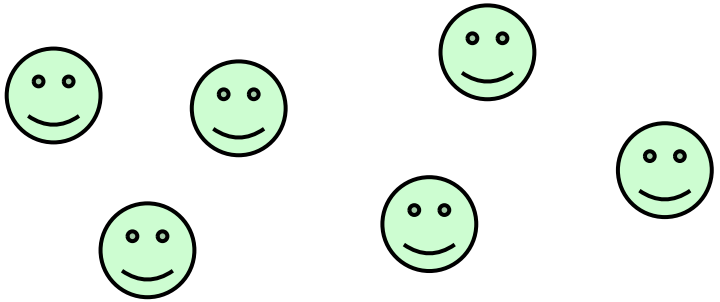
PiterPy
2023

Компиляторы пишут немногие специалисты?

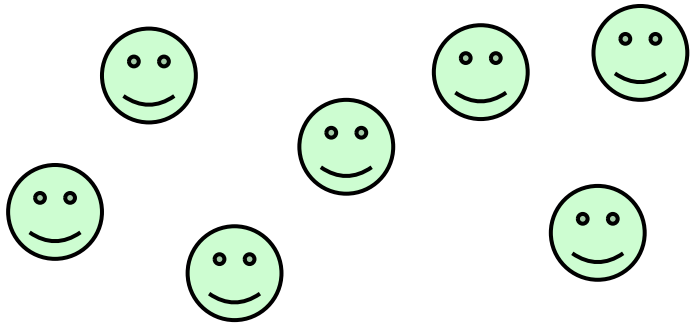
Языки общего назначения и
компиляторные фреймворки.

Компиляторы пишут многие! Иногда сами не зная того

Языки общего назначения и компиляторные фреймворки.

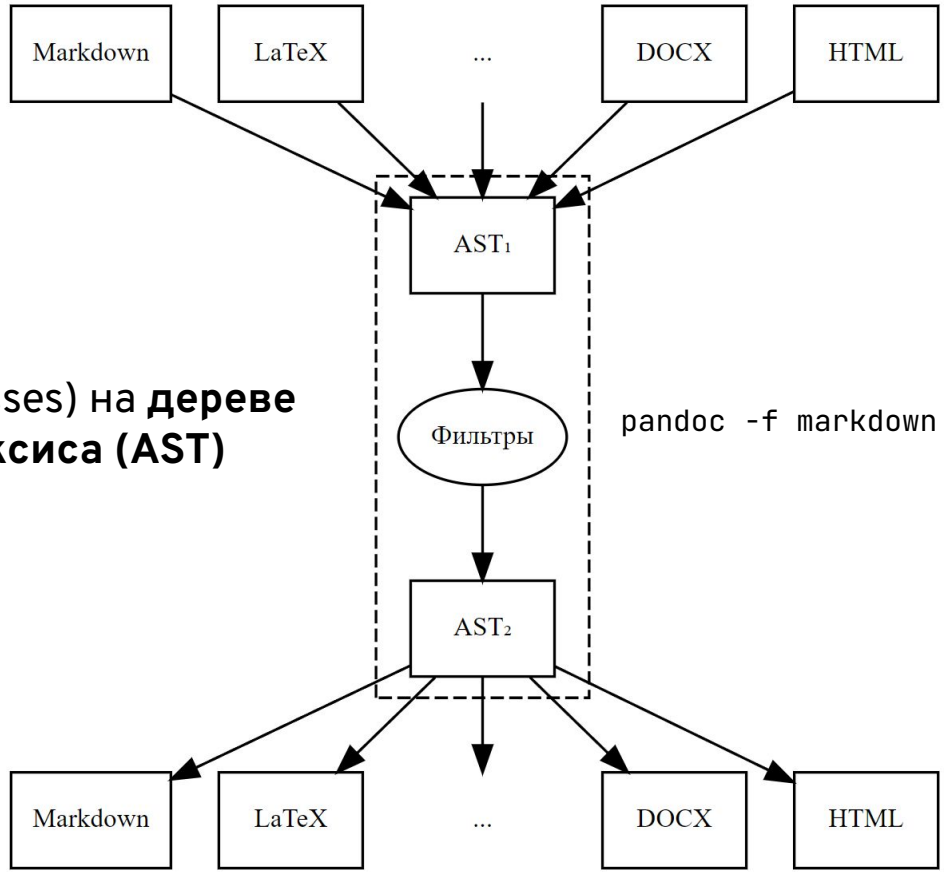


- **Предметно-ориентированные языки (DSL)** и **DSL-компиляторы.**
- **Визуализаторы кода.**
- **Статические анализаторы.**



Компиляторы повсюду, например, в архитектуре Pandoc

Многоязыковый **фронтенд**



Преобразования (passes) на **дереве абстрактного синтаксиса (AST)**

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

Многоязыковый бэкенд

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

О чём пойдет речь

- О подходе на основе **модуля ast** и **match/case** для разработки DSL-компиляторов, визуализаторов, статических анализаторов.
- Этот подход показан на примерах, каждый пример — **менее 100 строк** кода.
- **Ссылка** на репозиторий **будет**.

Некоторые плюсы подхода для создания DSL

- Выразительный синтаксис DSL.
- Бесплатный синтаксический разбор.
- Простота обработки синтаксических ошибок.
- Готовый AST для DSL-компилятора.
- Легкость интеграции с основным кодом на Python.
- Поддержка подсветки в IDE.

1. Visitor против match/case
2. Визуализатор Python AST
3. DSL-компилятор описания графов
4. DSL-компилятор Datalog
5. Визуализатор CFG
6. Поиск неиспользуемых переменных
7. DSL-компилятор в Wasm

AST арифметического выражения (Visitor против 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')
```

… и базовый “посетитель”


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

Задача форматирования кода


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

Форматирование кода: сравнение реализаций


```
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})'
```

Задача упрощения кода


```
>>> 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
```

Упрощение кода: сравнение реализаций


```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 и типы-суммы: еще одно определение AST


```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 и типы-суммы: генерация стекового кода


```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)
```

← Проверка на исчерпание альтернатив

```
>>> 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 против match/case
2. **Визуализатор Python AST**
3. DSL-компилятор описания графов
4. DSL-компилятор Datalog
5. Визуализатор CFG
6. Поиск неиспользуемых переменных
7. DSL-компилятор в Wasm

Модуль ast популярнее, чем иногда думают

- **Sphinx**: для генерации API-документации из кода.
- **Pyflakes**: для анализа кода на предмет ошибок.
- **Coverage**: для анализа покрытия кода.
- **Pytest**: для замены обычного assert более информативной версией.
- **Pandas**: для разбора запросов.
- **Kivy**: для поддержки выполнения Python-кода в kv-файлах.
- **PonyORM**: для реализации языка запросов.

Модуль ast

- **Отсутствуют** определения классов AST, они реализованы на C (модуль _ast). **Грамматику AST** придется регулярно смотреть: https://docs.python.org/3/library/ast.html
- В модуле ast есть функции для преобразования текста в AST и обратно, а также классы-посетители для обхода и преобразования деревьев:

```
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):
    ...
```

Модуль ast: **только с match/case!**

- **Отсутствуют** определения классов AST, они реализованы на C (модуль _ast). **Грамматику AST** придется регулярно смотреть: https://docs.python.org/3/library/ast.html
- В модуле ast есть функции для преобразования текста в AST и обратно, а также ~~классы-посетители для обхода и преобразования деревьев~~:

```
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):
        ...
```

Как получить AST функции


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

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

Как узнать имена полей узла AST: _fields

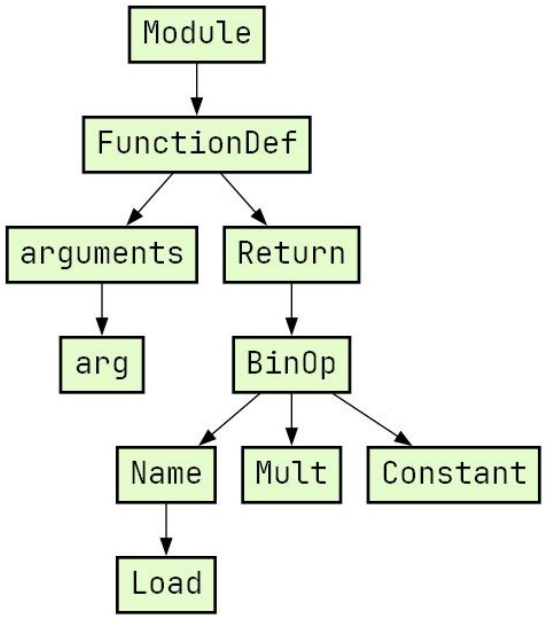

```
def foo(x):
    return x * 2
```

```
>>> 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>]
```

Набросок визуализатора AST

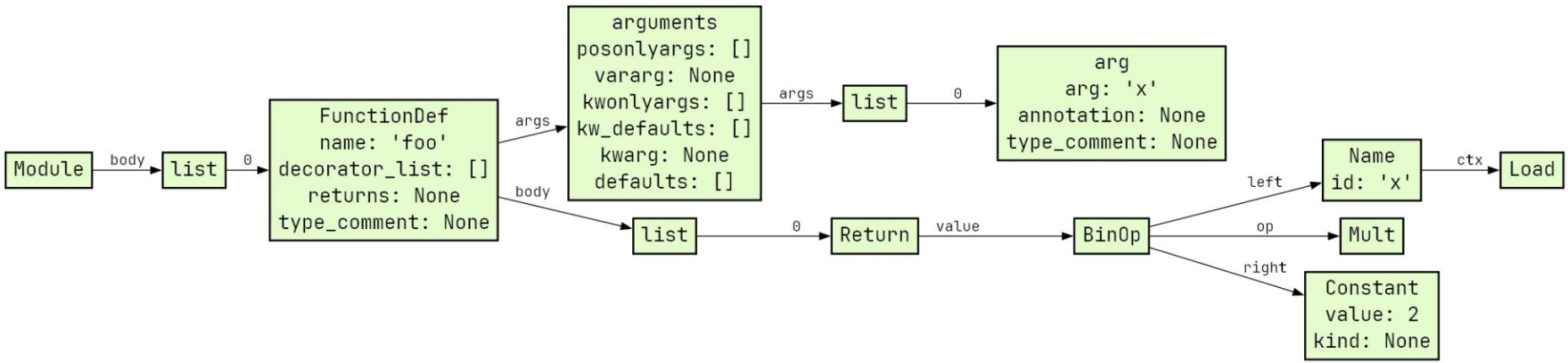

```python
def ast_viz(tree):
    graph, labels = {}, {}

    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():                    ← Базовый класс AST
                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)          ← Использую Graphviz
```

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



Результат работы расширенной версии визуализатора

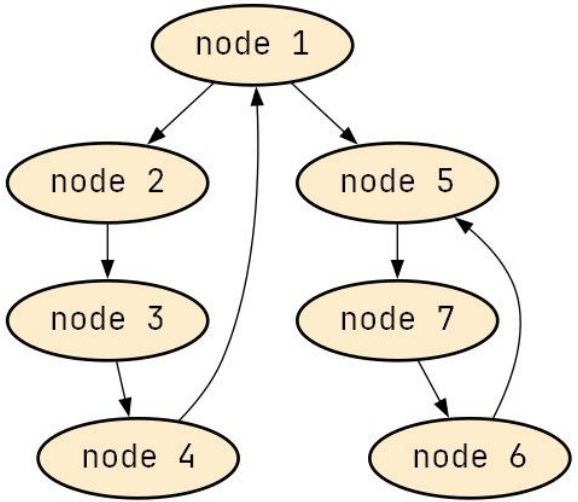

```
def foo(x):
    return x * 2
```



1. Visitor против match/case
2. Визуализатор Python AST
3. **DSL-компилятор описания графов**
4. DSL-компилятор Datalog
5. Визуализатор CFG
6. Поиск неиспользуемых переменных
7. DSL-компилятор в Wasm

Язык описания графов


```
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'
'''
```

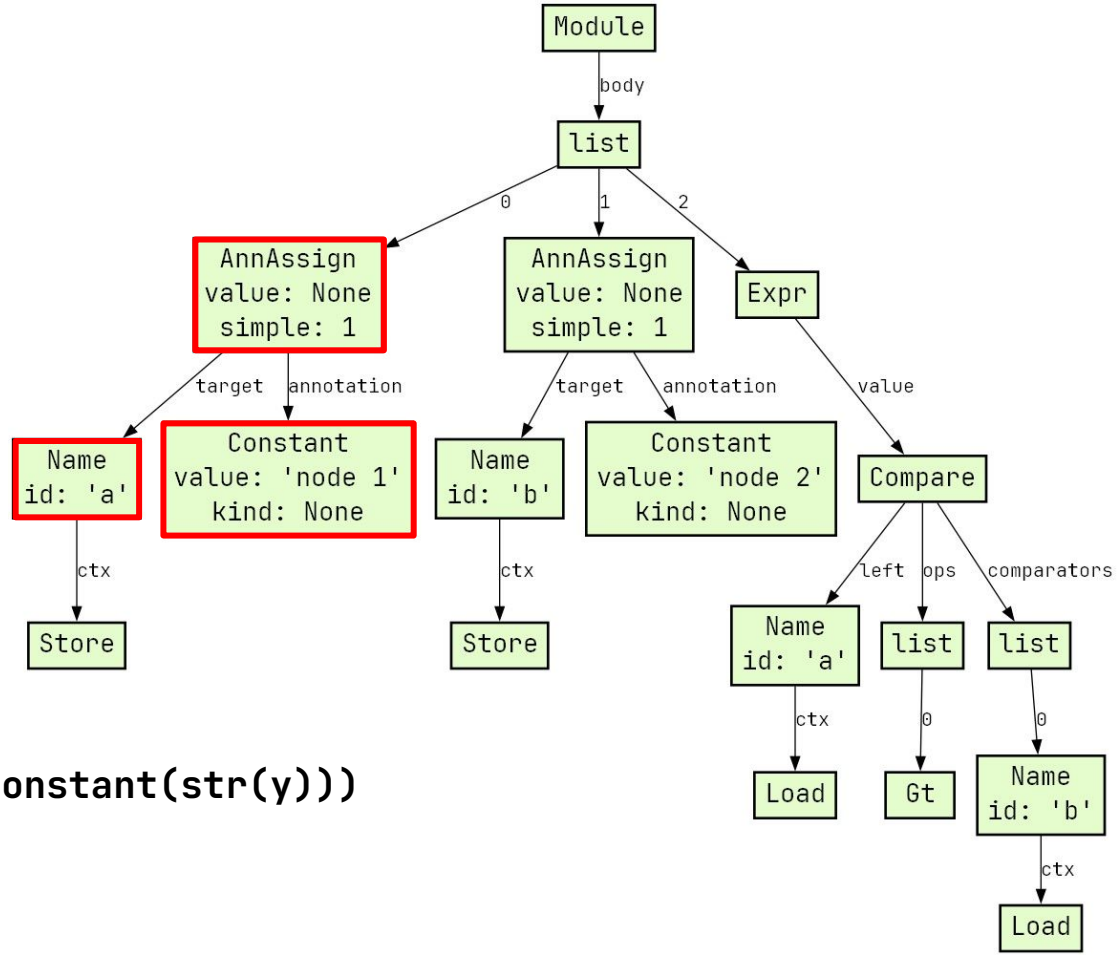



```
print(graph_viz(src))
```

Переменные определяются **по факту их появления** в тексте.
В качестве бэкенда используется Graphviz.

AST языка описания графов (1)

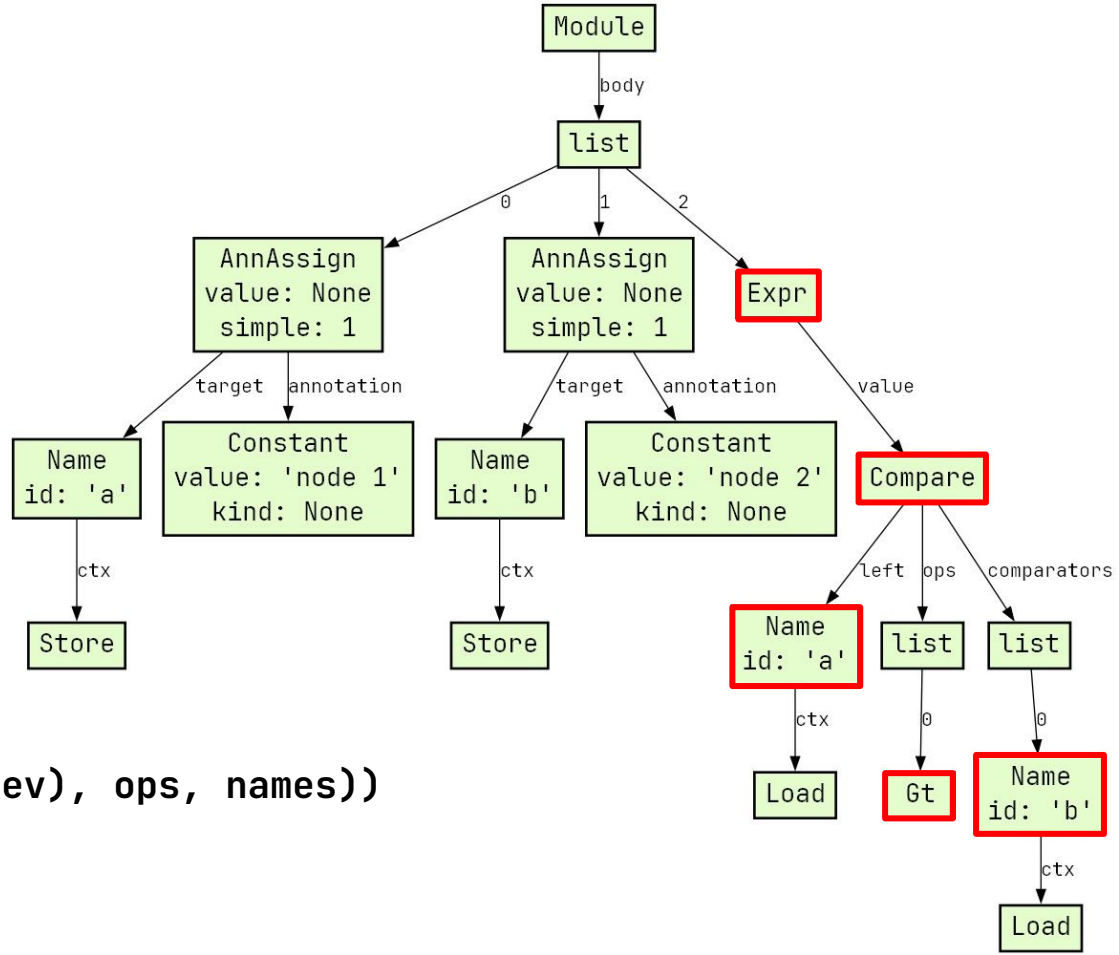

```
a: 'node 1'
b: 'node 2'
a > b
```




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

AST языка описания графов (2)


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



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

DSL-компилятор описания графов: реализация


```
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 + ['}'])
```

Пример обработки ошибок


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

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

Атрибуты узлов и get_error_details()

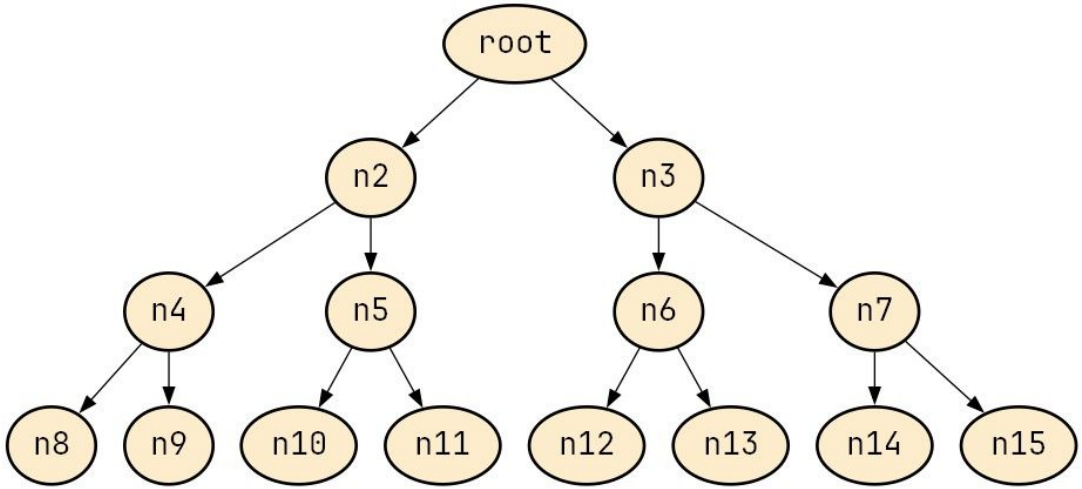

```
>>> tree = ast.parse(src).body[0]
>>> tree._attributes
('lineno', 'col_offset', 'end_lineno', 'end_col_offset')
```

```
    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)
```

← Совместима с SyntaxError

Пример работы визуализатора графа


```
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 против match/case
2. Визуализатор Python AST
3. DSL-компилятор описания графов
4. **DSL-компилятор Datalog**
5. Визуализатор CFG
6. Поиск неиспользуемых переменных
7. DSL-компилятор в Wasm

Datalog

- Логический DSL, миниатюрный вариант Prolog.
- Язык для БД с поддержкой рекурсивных запросов.
- Основные применения: графовые БД и статический анализ программ.

Некоторые реализации: Soufflé, Datomic, μZ в составе решателя Z3 (есть для Python).

Datalog: пример в духе SQL


```
city(1, 'Москва').
city(2, 'Санкт-Петербург').
city(3, 'Новосибирск').
ordered(1, 1).
ordered(1, 2).
ordered(3, 3).
product(1, 'чай').
product(2, 'хлеб').
product(3, 'цветы').

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

← Факты

← Правило. Переменные с большой буквы

Datalog: пример в духе SQL (реальный синтаксис)


```
city(1, 'Москва').
city(2, 'Санкт-Петербург').
city(3, 'Новосибирск').
ordered(1, 1).
ordered(1, 2).
ordered(3, 3).
product(1, 'чай').
product(2, 'хлеб').
product(3, 'цветы').

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

← Факты

← Правило. Переменные с большой буквы

Datalog: пример в духе SQL (запросы)


```
city(1, 'Москва').
city(2, 'Санкт-Петербург').
city(3, 'Новосибирск').
ordered(1, 1).
ordered(1, 2).
ordered(3, 3).
product(1, 'чай').
product(2, 'хлеб').
product(3, 'цветы').
```

← Факты

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

← Правило. Переменные с большой буквы

```
> ship(ProdName, 'Москва')?
ProdName=Чай
ProdName=Хлеб
> ship(ProdName, City)?
ProdName=хлеб, City=Москва
ProdName=цветы, City=Новосибирск
ProdName=чай, City=Москва
```

← Запросы

Datalog: пример запроса с отрицанием


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

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

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

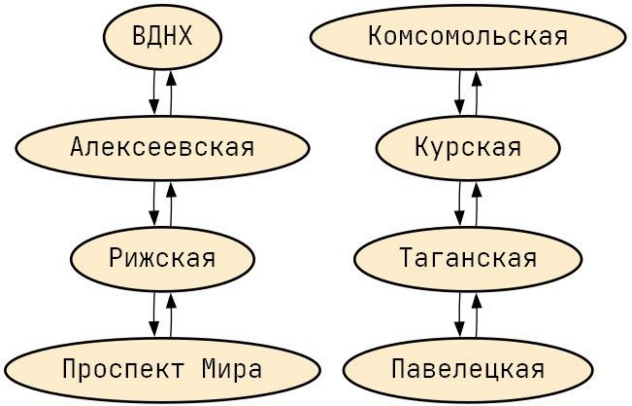
← Факты

← Правило. Переменные с большой буквы

← Запрос

Datalog: пример рекурсивного запроса


```
links(1, 'ВДНХ', 'Алексеевская').
links(1, 'Алексеевская', 'Рижская').
links(1, 'Рижская', 'Проспект Мира').
links(2, 'Комсомольская', 'Курская').
links(2, 'Курская', 'Таганская').
links(2, 'Таганская', 'Павелецкая').

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

> reach('ВДНХ', Station)
Station=Рижская
Station=Проспект Мира
Station=ВДНХ
Station=Алексеевская
```



Реализация рекурсивного запроса на Python/Z3


```
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)
```

Реализация на Z3: **нужен DSL!**

```
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-компилятор Datalog: факты и правила


```
@datalog
def metro():
    links(1, 'ВДНХ', 'Алексеевская')
    links(1, 'Алексеевская', 'Рижская')
    links(1, 'Рижская', 'Проспект Мира')
    links(2, 'Комсомольская', 'Курская')
    links(2, 'Курская', 'Таганская')
    links(2, 'Таганская', 'Павелецкая')

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

Используются **декоратор**, внутри которого вызов `ast.parse(inspect.getsource(func))`.

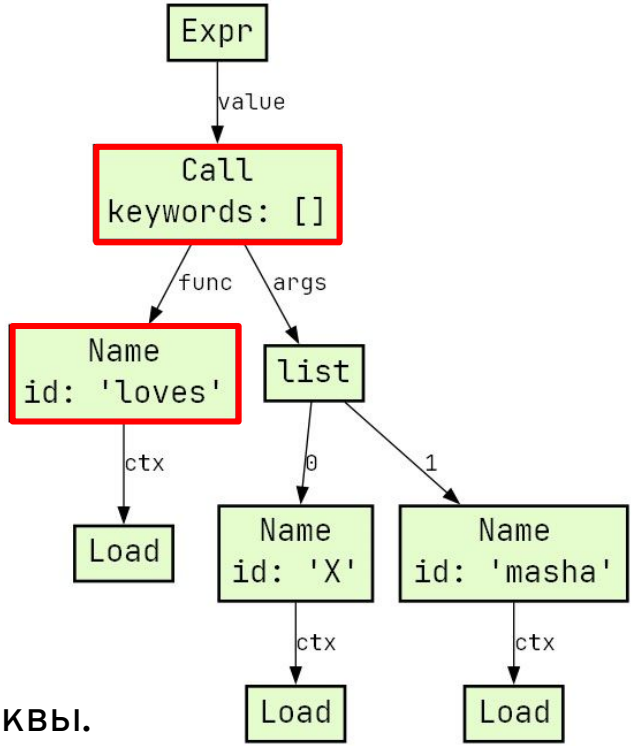
DSL-компилятор Datalog: запрос


```
@datalog
def metro():
    links(1, 'ВДНХ', 'Алексеевская')
    links(1, 'Алексеевская', 'Рижская')
    links(1, 'Рижская', 'Проспект Мира')
    links(2, 'Комсомольская', 'Курская')
    links(2, 'Курская', 'Таганская')
    links(2, 'Таганская', 'Павелецкая')

    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("ВДНХ", Station)')
>>> pprint(rows)
[{'Station': 'Рижская'},
 {'Station': 'ВДНХ'},
 {'Station': 'Проспект Мира'},
 {'Station': 'Алексеевская'}]
```

Синтаксис атома: факт, подцель или запрос

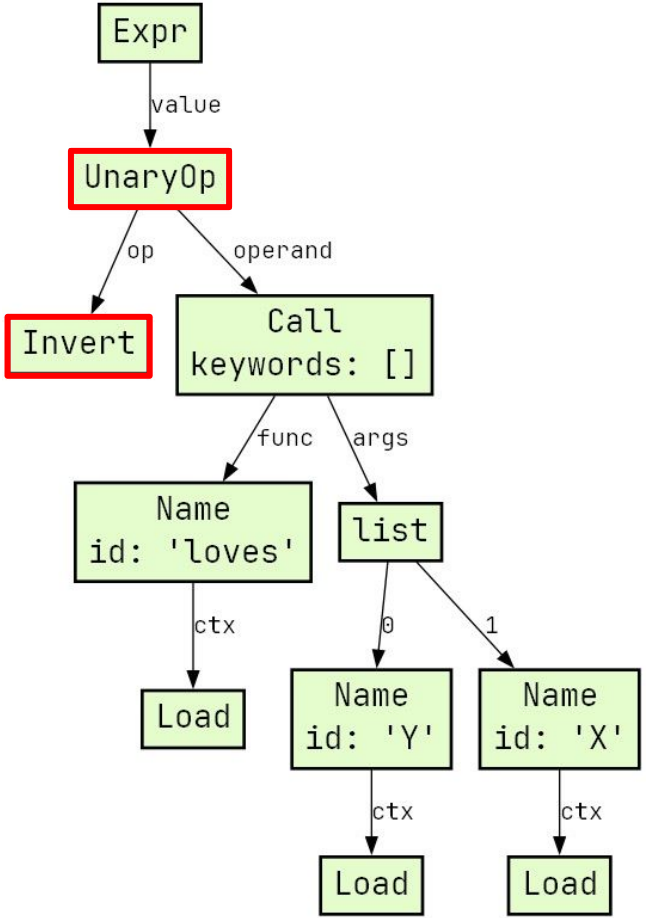

```
"loves(X, masha)"

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



Переменные начинаются с большой буквы.

Синтаксис атома с отрицанием


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

UnaryOp(Invert(), atom)
```



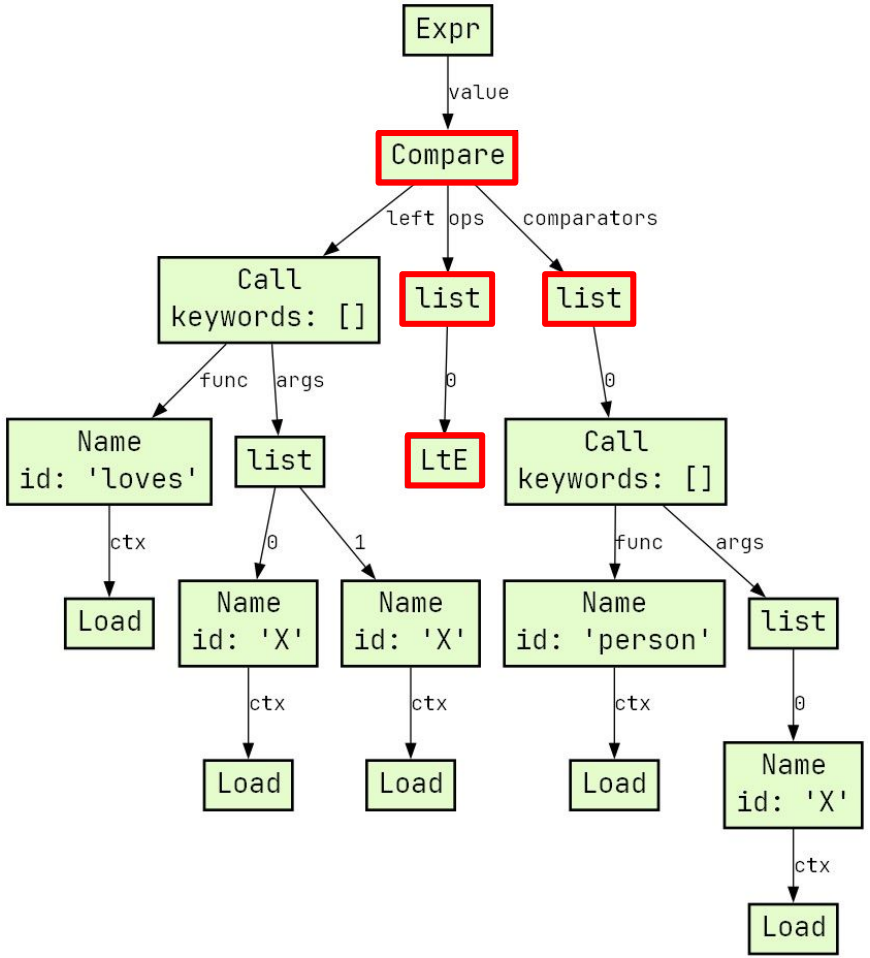
Синтаксис атома с отрицанием: а почему не с not?


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

Синтаксис правила


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

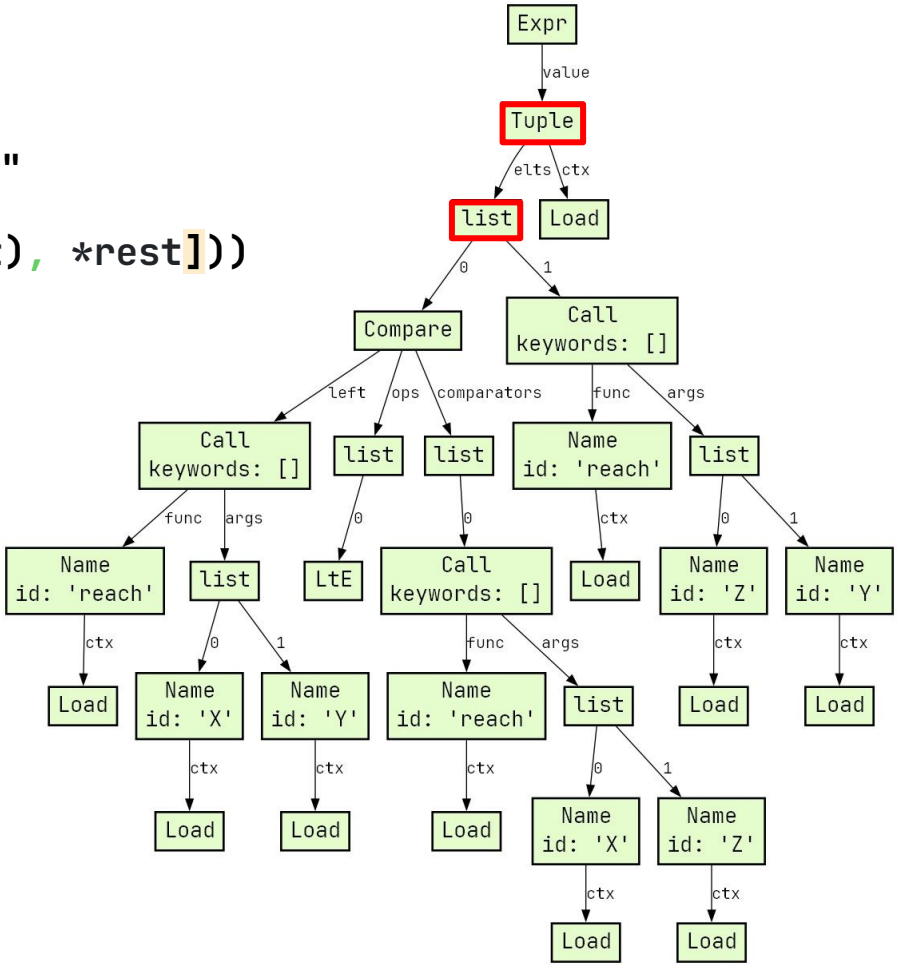
```
Expr(Compare(head, [LtE()], [first]))
```



Синтаксис длинного правила: “перегрузка” запятой

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

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



Термы: переменные и значения


```
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)
```

Значения

Значения в Z3 для Datalog — только **битовые векторы**.
Поэтому надо сопоставить каждому значению **номер из хеш-таблицы**.

```
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)
```

К Datalog я ещё вернусь!

1. Visitor против match/case
2. Визуализатор Python AST
3. DSL-компилятор описания графов
4. DSL-компилятор Datalog
5. **Визуализатор CFG**
6. Поиск неиспользуемых переменных
7. DSL-компилятор в Wasm

CFG — основа для серьезного статического анализа

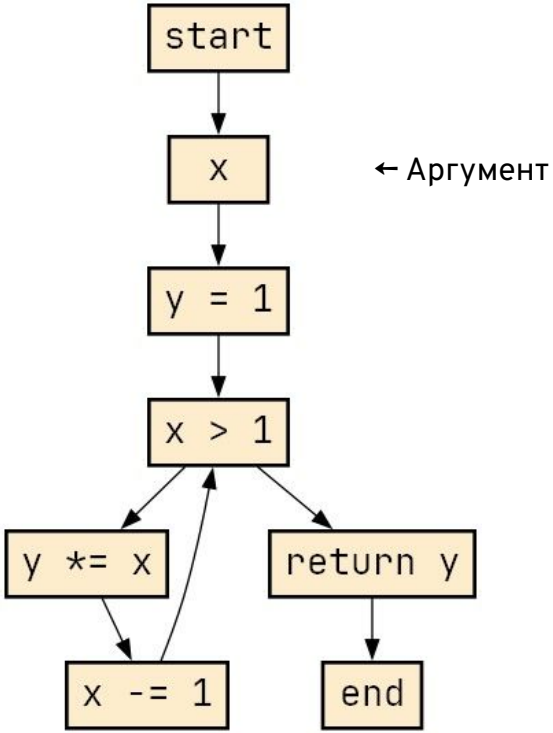
CFG (control flow graph) — **граф потока управления**.

В CFG **узлы это операторы**, а **рёбра — переходы** между операторами.

Кстати, CPython тоже строит этот граф, но он **недоступен** прикладному программисту. Это уже уровень C: https://devguide.python.org/internals/

Пример графа потока управления

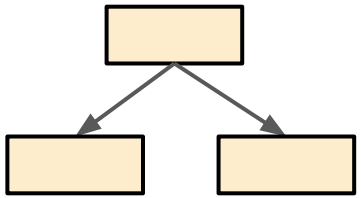

```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```



Как построить CFG? Пройти по AST!

Мы можем соединять операторы по цепочке: один за одним.

Но что делать, например, с оператором if, у которого **две** ветви исполнения?

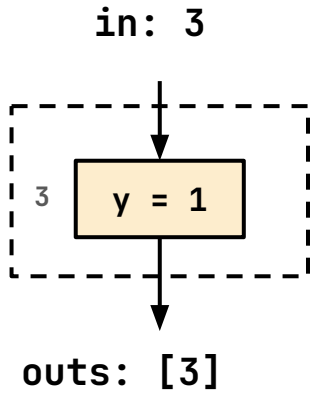


Пусть **каждый оператор** имеет:

- **один узел-вход (in),**
- **множество узлов-выходов (outs)**.

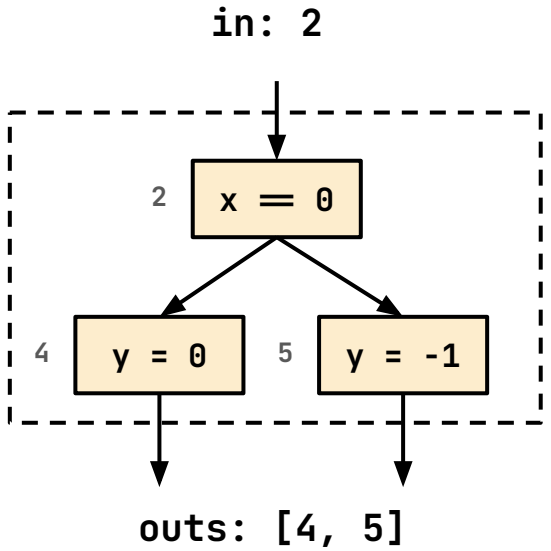
In/outs в процессе построения оператора if (1)


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



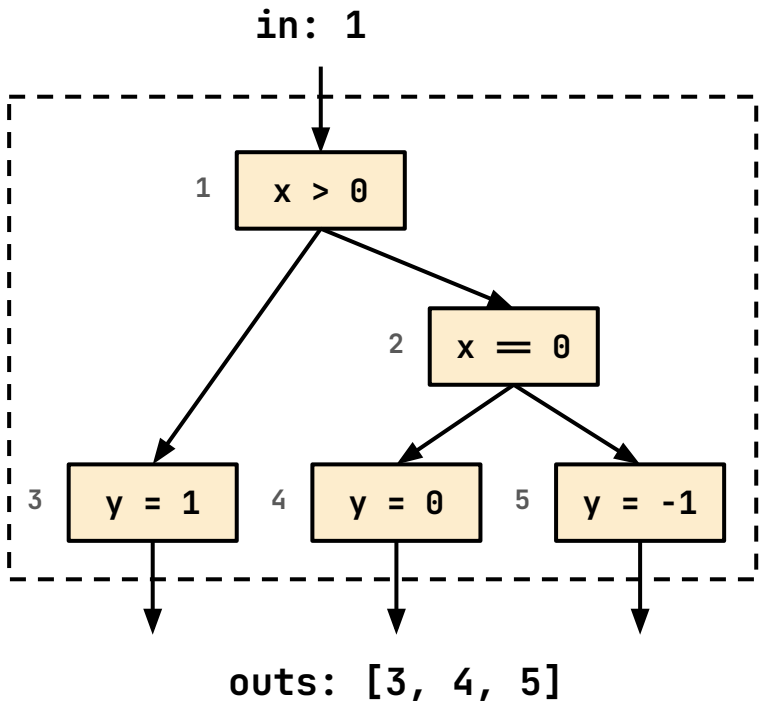
In/outs в процессе построения оператора if (2)


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



In/outs в процессе построения оператора if (3)


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



Обход CFG

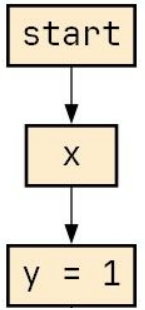

```
# Предоставляет пользователь

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

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

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

```
# Реализация обхода CFG

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:
    ...
```

Как построить цикл while (1)

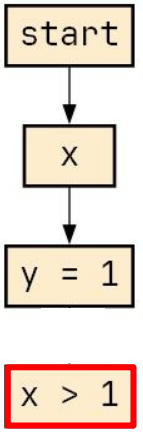

```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```




```
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
```

Как построить цикл while (2)

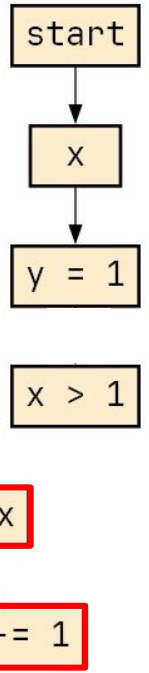

```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```




```
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
```

Как построить цикл while (3)

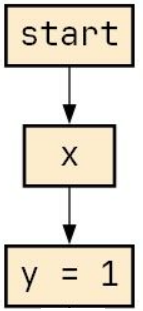

```
def fact(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```



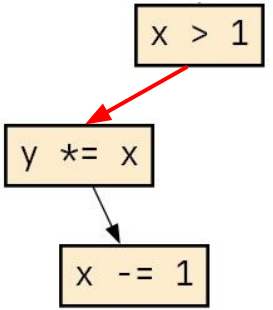

```
def walk_while(graph, test, body):
    test_in, test_outs = add_node(graph, test)
    body_in, body_outs = walk_block(graph, body)
    connect(graph, test_outs, body_in)
    connect(graph, body_outs, test_in)
    return test_in, test_outs
```

Как построить цикл while (4)


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

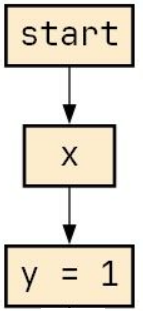



```
def walk_while(graph, test, body):
    test_in, test_outs = add_node(graph, test)
    body_in, body_outs = walk_block(graph, body)
    connect(graph, test_outs, body_in)
    connect(graph, body_outs, test_in)
    return test_in, test_outs
```

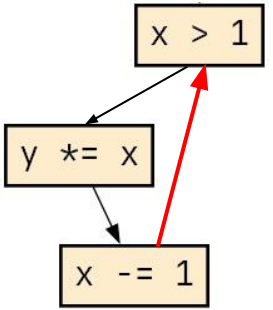


Как построить цикл while (5)


```
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
```



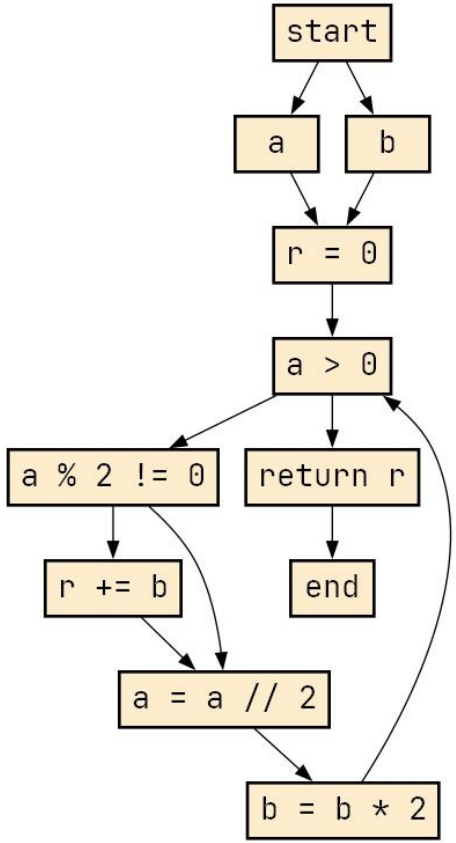
Визуализатор CFG


```
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())
...
```

Пример визуализации CFG


```
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 против match/case
2. Визуализатор Python AST
3. DSL-компилятор описания графов
4. DSL-компилятор Datalog
5. Визуализатор CFG
6. **Поиск неиспользуемых переменных**
7. DSL-компилятор в Wasm

Есть ли здесь неиспользуемые переменные?


```
 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
```

Результат работы статического анализатора


```
 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
```

Почти все готово для реализации такого поиска!

Сформулировать на Datalog правила нахождения неиспользуемых переменных.

1. **Обойти CFG** программы и **собрать факты** о переменных в виде БД для Datalog.
2. Сделать **запрос** на Datalog.

Правила для “живых” переменных (1)

1. Переменная V “жива” **перед** (live in) оператором P, если она используется в этом операторе:

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

Правила для “живых” переменных (2)

1. Переменная V “жива” **перед** (live in) оператором P, если она используется в этом операторе:

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

2. Переменная V “жива” **перед** оператором P, если она не переопределена в P и она “жива” **после** оператора P:

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

Правила для “живых” переменных (3)

1. Переменная V “жива” **перед** (live in) оператором P, если она используется в этом операторе:

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

2. Переменная V “жива” **перед** оператором P, если она не переопределена в P и она “жива” **после** оператора P:

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

3. Переменная V “жива” **после** (live out) оператора P1, если есть переход из P1 в P2 и эта переменная “жива” **перед** оператором P2:

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

А где же неиспользуемые переменные?

Переменная V “мертва” в операторе P, если она определена в P, но не “жива” **после** P:

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

Правила для неиспользуемых переменных на DSL

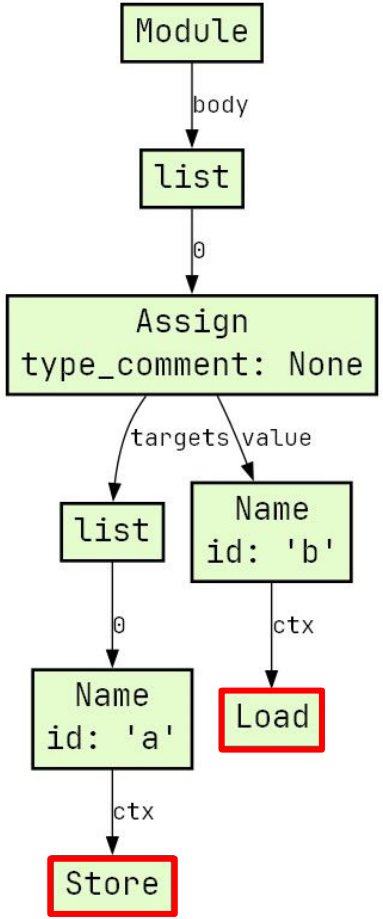

```
@datalog
def dead_var():
    live_in(P, V) <= used(P, V)
    live_in(P, V) <= ~defined(P, V), live_out(P, V)
    live_out(P1, V) <= edge(P1, P2), live_in(P2, V)
    dead_var(P, V) <= defined(P, V), ~live_out(P, V)
```

Реализация статического анализа


```
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]
```

Как найти defs и uses: проверить ctx


```
a = b
```



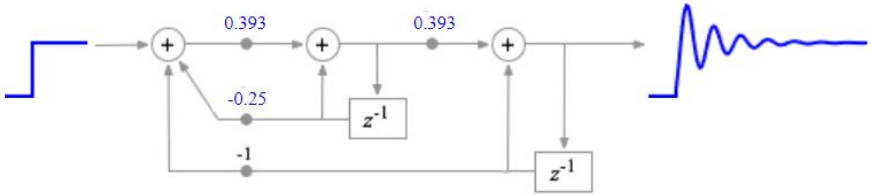
Реализация get_du()


```
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 против match/case
2. Визуализатор Python AST
3. DSL-компилятор описания графов
4. DSL-компилятор Datalog
5. Визуализатор CFG
6. Поиск неиспользуемых переменных
7. DSL-компилятор в Wasm

Интерактивные визуализации (explorable explanations)

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

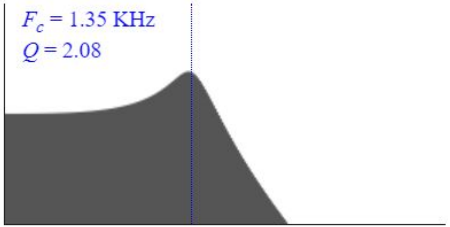


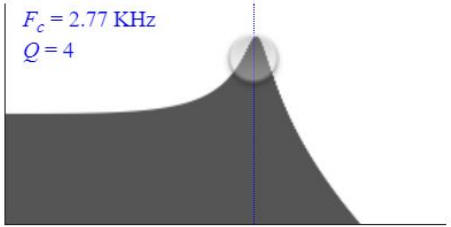
The coefficients and transfer function are:

$k_f = 0.393 \qquad k_q = 0.25$

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

Some example frequency responses:





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

Компилятор скриптов интерактивных графиков в Wasm

Требования:

- Выразительное **подмножество Python**, которое можно использовать **для прототипирования в Matplotlib**.
- **Производительность** сгенерированного кода , близкая к **JavaScript**.
- Скомпилированные **Wasm-модули** должны занимать **сотни байт**, а не десятки мегабайт.
- Реализация компилятора **<100 строк** кода.

Компилятор PyWasm: особенности реализации

- Поддерживаются **только** значения типа **float** (64 бита).
- **Списки** обрабатываются **отдельно** (см. далее).
- Поддерживаются **функции**, **if** и **while**. Цикл for не удалось втиснуть в общее ограничение <100 строк кода.
- Генерация кода мало отличается от примера **генерации стекового кода** в начале доклада.

Списки — это просто номера в JS-таблице mem


```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'
```

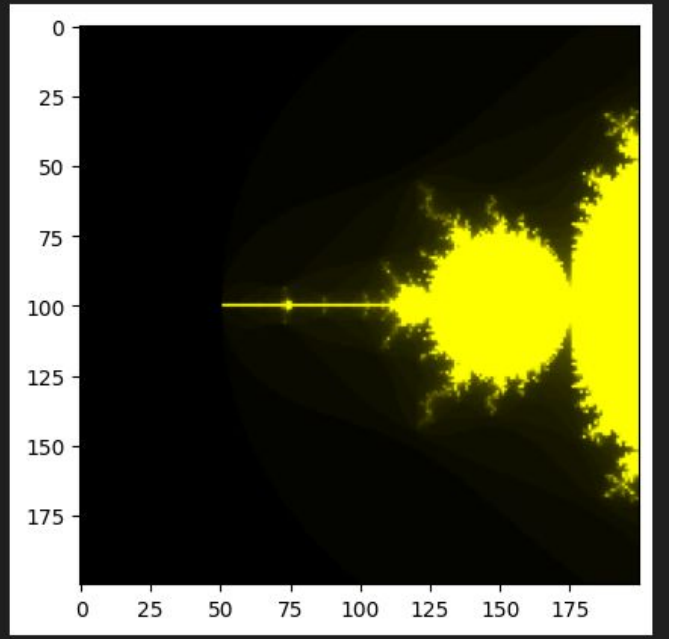
Python

```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;
    }
};
```

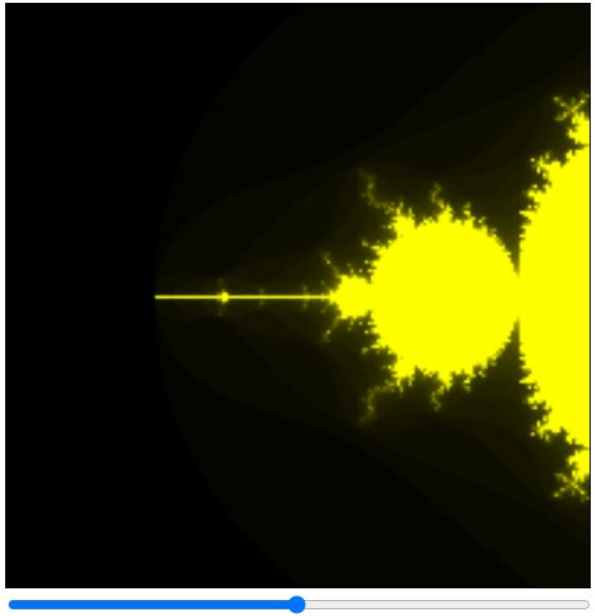
JavaScript

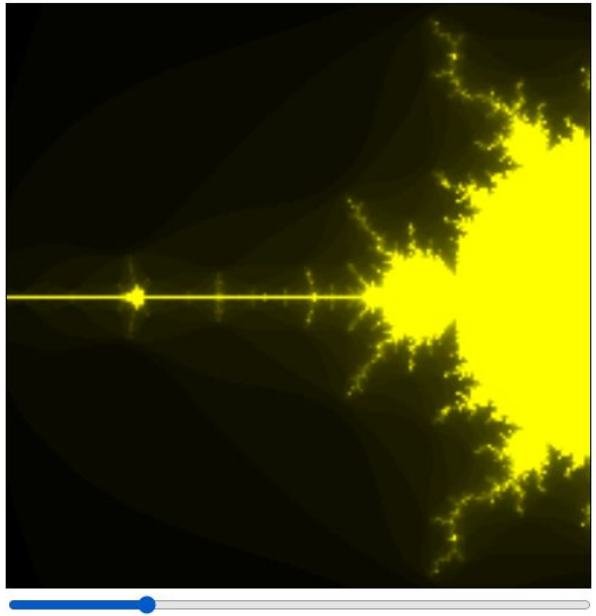
Прототип визуализации фрактала в Jupyter Notebook

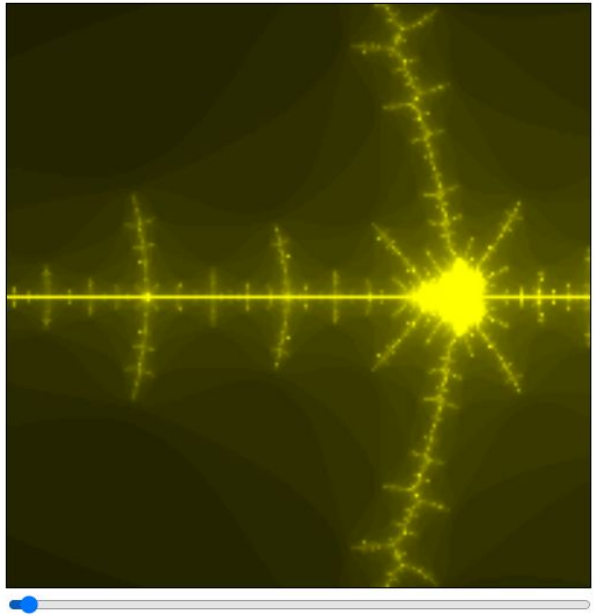

```python
def mandel(x, y, times):
    i = 0
    zr = x
    zi = y
    while i < times:
        zr_new = zr * zr - zi * zi + x
        zi = 2 * zr * zi + y
        zr = zr_new
        if zr * zr + zi * zi >= 4:
            return 255 * i / times
        i += 1
    return 255

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
```



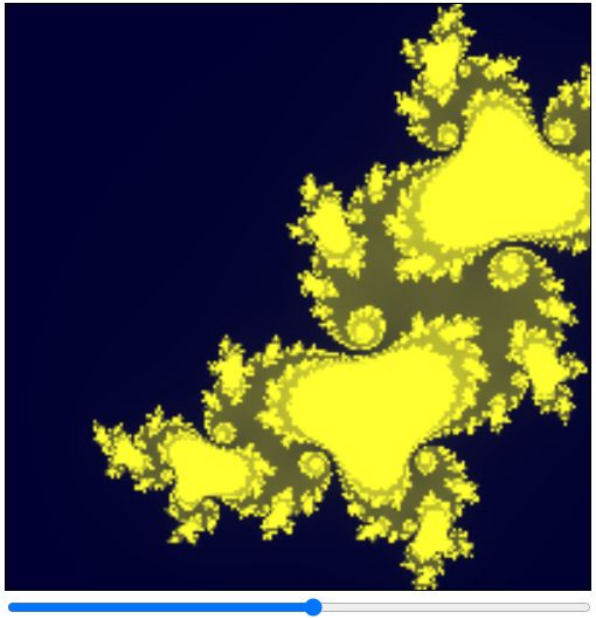
Веб-визуализация на PyWasm: фрактал Мандельброта

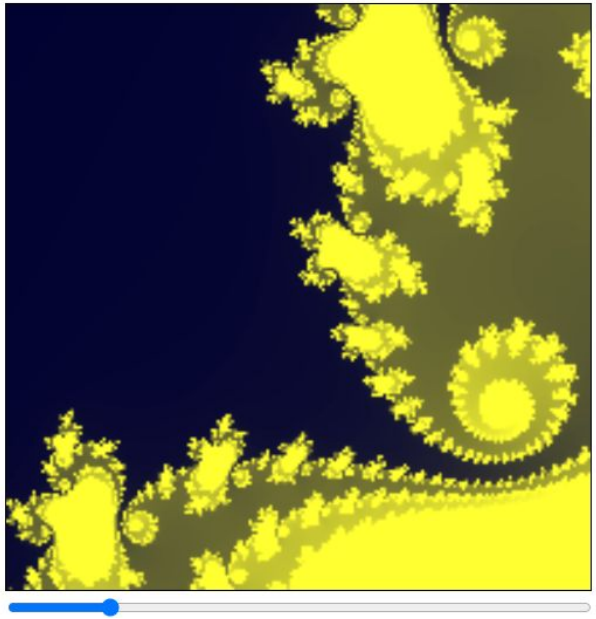


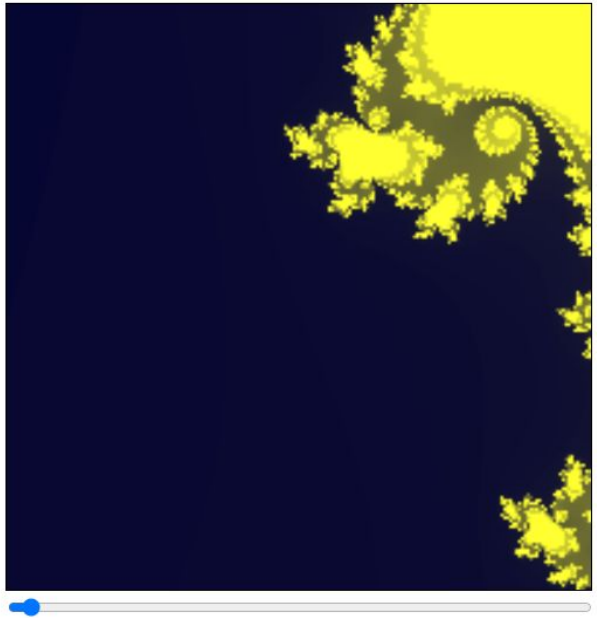




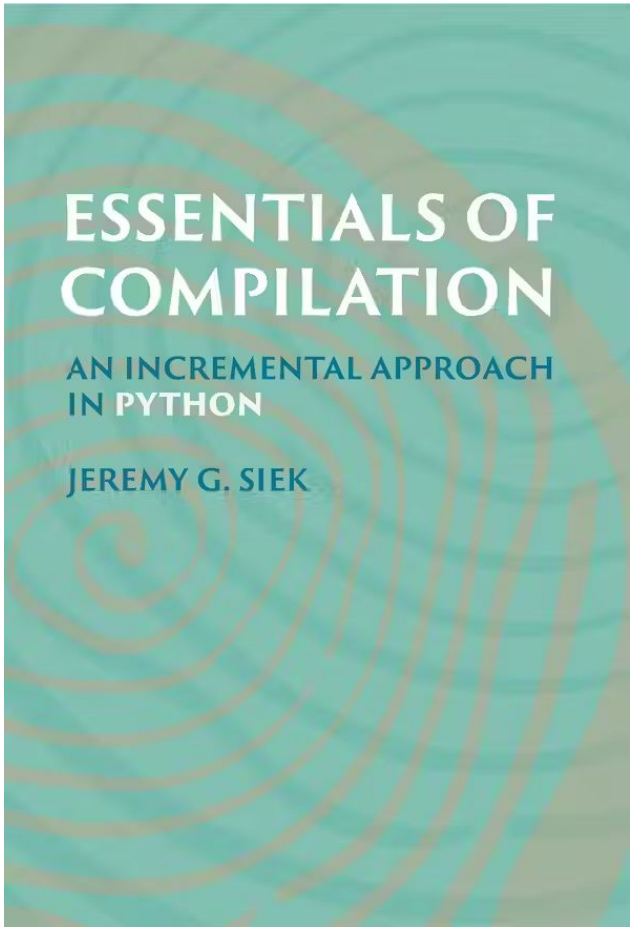
Веб-визуализация на PyWasm: фрактал Жюлиа







Что современного почитать о разработке компиляторов



Спасибо за внимание!

Репозиторий со всеми представленными примерами:
https://github.com/true-grue/python-dsls