Статический анализатор программ на ЯТЬ

Общее описание и система оценки

В этом задании вам предлагается реализовать два класса: `PureCheckVisitor` и `NoReturnValueCheckVisitor`. Каждый из них должен реализовывать как минимум метод visit с одним параметром - синтаксическим деревом программы. С использованием паттерна “Visitor” каждый класс должен решать отведённую ему задачу. Ваши реализации должны проходить тесты, а реализация PureCheckVisitor - ещё и хорошо выглядеть, тогда тема засчитывается. Если при этом также хорошо выглядит ваш NoReturnValueCheckVisitor, то вы получаете 2,5 балла за тему.

Постановка задачи для PureCheckVisitor

Назовём программу на языке ЯТЬ *чистой*, если в ней отсутствуют вызовы методов Read и Write. Например, следующая программа, записанная в переменную `prog1`, является чистой:

```
prog1 = Conditional(BinaryOperation(Number(4), "=", Number(5)), [
    Number(123),
], [
    Number(456),
])
```

А вот такая, записанная в переменную `prog2` -- не является, несмотря на то, что первая ветка Conditional никогда не может быть выполнена:

```
prog2 = Conditional(BinaryOperation(Number(4), "=", Number(5)), [
    Print(Number(123)),
], [
    Number(456),
])
```

Метод visit у PureCheckVisitor должен возвращать `True`, если переданное ему синтаксическое дерево представляет собой чистую программу, и False в противном случае. Например:

```
>>> PureCheckVisitor().visit(prog1)
True
>>> PureCheckVisitor().visit(prog2)
False
```

Методу visit в качестве параметра может быть передан экземпляр любого из следующих классов: Number, Function, FunctionDefinition, Conditional, Print, Read, FunctionCall, Reference, BinaryOperation, UnaryOperation. Обратите внимание, что Print

и Read могут также встречаться внутри BinaryOperation (т.е. ограничений, подобных в задании ЯТЬ-Visitors, нет), например:

```
Conditional(BinaryOperation(Read("foo"), "=", Number(5)), [
    Number(10),
], [
    Number(20),
])
```

Моя реализация этого класса занимает 36 строк и 1002 байта.

Постановка задачи для NoReturnValueCheckVisitor

В первом задании на ЯТЬ не определялось значение функции evaluate в следующих двух случаях:

1. Выполняется пустая ветка Conditional.
2. Выполняется пустое тело функции.

Также понятно, что из-за этого нельзя разумно определить, скажем, результат функции, у которой последней операцией является Conditional с пустыми ветками - так как функция должна возвращать результат последней операции, а результат последней операции не определён, определить результат функции невозможно.

Вам требуется написать класс NoReturnValueCheckVisitor, который будет обходить предоставленное ему синтаксическое дерево и искать функции, результат вычисления которых может быть не определён. Более точно, назовём следующие выражения *плохими* (а остальные, соответственно, *хорошими*):

1. Conditional, у которого хотя бы одна ветка пустая.
2. Function с пустым телом.
3. Выражение, при вычислении значения которого может потребоваться значение плохого выражения. Например:
    a. При вычислении значения Conditional может потребоваться последнее значение любой из веток (мы не будем дополнительно разбираться, может ли условие Conditional быть истинным или ложным).
    b. При вычислении FunctionDefinition вообще ничего не требуется - результат вычисления сразу известен (это объявляемый Function).

Также примем следующие допущения:

1. Reference мы всегда считаем хорошим.
2. Read мы всегда считаем хорошим (несмотря на то, что пользователь потенциально может ввести не-число и порушить программу).
3. Мы предполагаем, что FunctionCall является хорошим тогда и только тогда, когда и все аргументы вызова и выражение, вычисляющееся в функцию, являются хорошими. Что происходит при этом внутри вызываемой функции, нас не интересует.

Например, следующие выражения являются плохими:

```
Conditional(Number(0), [], []) # Пустая ветка в Conditional
Conditional(Number(0), [], [Number(10)])
Conditional(Number(0), [Number(1)], [])
```

```
Function([], [
    Number(123),
    # Этот Conditional плохой и делает плохой всю функцию
    Conditional(Number(1), [
        Number(2),
    ])
])
Conditional(Number(1), [
    Number(2),
], [
    # Этот Conditional плохой (в одной ветке нет операций) и
    # он делает плохим внешний Conditional.
    Conditional(Number(3), [
        Number(4)
    ], [
    ])
])
```

А следующие - хорошими:

```
Conditional(Number(1), [
    # Этот Conditional плохой, но результат его вычисления не
    # влияет на результат внешнего, так как за ним следует ещё
    # одна команда
    Conditional(Number(0), [], []),
    Number(2),
], [
    Number(3)
])
# Все FunctionDefinition хорошие, независимо от хорошести
# объявленных в них функций.
FunctionDefinition("foo", Function([], [
    Number(123),
    Conditional(Number(1), [
        Number(2),
    ])
]))
# Несмотря на то, что функция выше foo плохая, мы предполагаем,
# что вызов завершится успешно: вообще говоря, при выполнении
# этой строчки именем foo уже может называться какая-нибудь другая
# функция.
FunctionCall(Reference("foo"), [])
FunctionCall(Reference("foo"), [Reference("undefined_variable")])
```

Вам требуется реализовать класс NoReturnValueCheckVisitor, который будет искать в синтаксическом дереве объявления функций и выводить на экран имена

плохих функций. Гарантируется, что класс Function в синтаксическом дереве может встретиться только как параметр к FunctionDefinition (в частности, метод visit никогда не будет вызван от Function). Таким образом, у каждого Function можно однозначно установить имя.

Например, вывод при вызове от следующего абстрактного дерева:

```
Conditional(Number(1), [
    FunctionDefinition("foo", Function([], [
        Number(123),
        Conditional(Number(1), [
            Number(2),
        ])
    ])),
    FunctionDefinition("bar", Function([], [
        Number(123)
    ])),
    FunctionDefinition("baz", Function([], [
    ])),
    FunctionDefinition("foobar", Function([], [
        Conditional(Number(1), [
            Number(2),
        ]),
        Number(123)
    ])),
])
```

Вывод должен быть следующим (строчки могут идти в любом порядке):

```
foo
baz
```

Моя реализация этого класса занимает 45 строк и 1256 байт.

Правила сдачи

1. Решением являются два файла: model.py из предыдущего задания с добавленными методам accept() в нужные классы и файл static_analyzer.py с реализацией двух требуемых классов.
2. Предполагается, что оба файла лежат в пакете yat, т.е. при импортировании нужно писать import yat.model, а не просто import model. Мне это не потребовалось.
3. Вы можете сдавать классы по очереди - сначала один, потом второй.
4. В письме указывайте ссылку GitHub на файл static_analyzer.py, либо вкладывайте оба файла (model.py и static_analyzer.py) в каждое письмо с попыткой сдачи.
5. В каждом письме указывайте, какой именно из двух классов вы хотите попробовать сдать.