Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

Отчет по лабораторной работе №5

Интерпретация исходного кода.

Выполнил:

студент гр. 953504 Басенко К. А.

Проверил:

Шиманский В.В.

Минск 2022

Содержание:

1. Постановка задачи. 3
2. Теория. 5
3. Результат работы интерпретатора. 19
4. Выводы. 21

1. Постановка задачи

В данной работе ставится задача реализации интерпретатора языка C# на объектно-ориентированном языке программирования Python. Основной целью работы является реализация виртуальной машины, способной обрабатывать команды языка C# и выдавать результат при исполнении, то есть быть интерпретатором. Это является последним шагом на пути в создании программы, способной выполнять программу самостоятельно.

Исследуемый код языка представлен ниже:

```
_ = "string string";

_ =          5 + 5;

_ = 1 + 2;
_ = 0 -                3;
_ = 9 / 5;
_ = 4 * 3           ;

_ = (1 +2) * 3;
_ = (9 - 4)/7;
_ = 5 + 3 - 2;

float prevar;
int variable = 2;

const int _1 =1; const int _2= 2;const int _3=3;

var a = 3d;
double b = a + 5;
var c=(1.5435e6+b)*0.543f;


(_, _) = (2f, 2d);
(_, _) = (2m, 2u);
(_, _) = (2l, 2L);
(_, _) = (2ul, 2UL);

a += c - b;
b -= (a + c) * 0.11e2D;
c *= a * b;

_ = a < b;
_ = b >= c;

```

```
// return a - b + c
int func(int a, int b, int c)
{
    for (int i = 0; i < b; i++)
    {
        a--;
    }

    while (c > 0)
    {
        a ++;
        c--;
    }

    return a;
}

c = func(5, 4, 2);
WriteLine(c);

(a, b) = (1, 2 + c);
WriteLine(a);
WriteLine(b);

WriteLine(a < b);
WriteLine((a + b) * 2);

int e = 10;
WriteLine(1 << e);

```

2. Теория

Интерпрета́тор — программа (разновидность транслятора), выполняющая интерпретацию; пооператорный (покомандный, построчный) анализ, обработка и тут же выполнение исходной программы или запроса (в отличие от компиляции, при которой программа транслируется без её выполнения).

Типы интерпретаторов:

- Простой интерпретатор анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора. Достоинством такого подхода является мгновенная реакция. Недостаток — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды (или строки) с ошибкой.
- Интерпретатор компилирующего типа — это система из компилятора, переводящего исходный код программы в промежуточное представление, например, в байт-код или p-код, и собственно интерпретатора, который выполняет полученный промежуточный код (так называемая виртуальная машина). Достоинством таких систем является большее быстродействие выполнения программ (за счёт выноса анализа исходного кода в отдельный, разовый проход, и минимизации этого анализа в интерпретаторе). Недостатки — большее требование к ресурсам и требование на корректность исходного кода. Применяется в таких языках, как Java, PHP, Tcl, Perl, REXX, а также в различных СУБД.

В случае разделения интерпретатора компилирующего типа на компоненты получаются компилятор языка и простой интерпретатор с минимизированным анализом исходного кода. Причём исходный код для такого интерпретатора не обязательно должен иметь текстовый формат или быть байт-кодом, который понимает только данный интерпретатор, это может быть машинный код какой-то существующей аппаратной платформы.

Некоторые интерпретаторы (например, для языков Lisp, Python, VB и др.) могут работать в режиме диалога или так называемого цикла чтения-вычисления-печати. В таком режиме интерпретатор считывает законченную конструкцию языка (например, s-expression в языке Lisp), выполняет её, печатает результаты, после чего переходит к ожиданию ввода пользователем следующей конструкции.

Генерация кода (и его оптимизация) или интерпретация, т.е. заключительная фаза трансляции - по своему способу включения в транслятор аналогична семантическому анализу. При выполнении шага синтаксического разбора («свертке» правила в восходящем разборе или подборе правила в нисходящем разборе) вызывается семантическая процедура, по результатам которой выполнения которой вызывается аналогичная процедура для генерации кода или интерпретации. В принципе эти две процедуры можно объединить.

Как и семантический анализ в целом, задача генерации кода/интерпретации не имеет формальной системы описания, как синтаксис и семантика. Она распадается на ряд частных решений и рекомендаций, применимых к большинству языков программирования (например, генерация кода для арифметических выражений, управляющих структур – операторов, распределение регистровой памяти и т.п.).

По способу включения в семантическую компоненту генераторы кода и интерпретаторы часто реализуются в виде независимой компоненты транслятора. В этом случае транслятор обычно использует промежуточные формы представления программного кода (обратная польская запись, система тетрад). Однако, следует дополнить, что язык программирования Lisp сам по своему синтаксису и использует обратную польскую нотацию, что несомненно даёт огромную экономию по времени и сокращает сложность интерпретации к минимуму.

Алгоритм работы простого интерпретатора

1. прочитать инструкцию;

2. проанализировать инструкцию и определить соответствующие действия;
3. выполнить соответствующие действия;
4. если не достигнуто условие завершения программы, прочитать следующую инструкцию и перейти к пункту 2.

Архитектура языка программирования Python

Так как вся работа была проделана на языке программирования Python, то необходимо рассказать и о его внутренней работе. Стоит начать с того, что сегодня существует огромное количество технологий *ython при работе с Питоном, что описать как работает каждая очень сложно. Говоря о Питоне, мы чаще всего имеем ввиду CPython, то есть тот Питон, что получил наибольшую популярность за свою стойкую позицию в плане производительности, простоты установки, да и он еще не самый сложный из инструментария (не то что PyPy, о котором пойдёт описание дальше).

Не смотря на схожесть в названиях технологий (CPython, PyPy, RPython, IronPython, JPython, Jython и многие другие), некоторые из них имеют совсем другие задачи (или, как минимум, работают совершенно иными способами). Итак, поехали!

CPython:

Все начинается с понимания того, чем на самом деле является “Питон”. Питон интерпретируемый или компилируемый?

- Первое, что необходимо понять: “Питон” – это интерфейс. Существует спецификация, описывающая, что должен делать Питон, и как он должен себя вести (что справедливо для любого интерфейса). И существует несколько имплементаций (что также справедливо для любого интерфейса).
- Второе: “интерпретируемый” и “компилируемый” - это свойства имплементации, но не интерфейса.

В случае с самой распространенной реализацией (то есть CPython) -ответ: *интерпретируемый, с некоторой компиляцией*. Цепочка операций на CPython выглядит следующим образом:

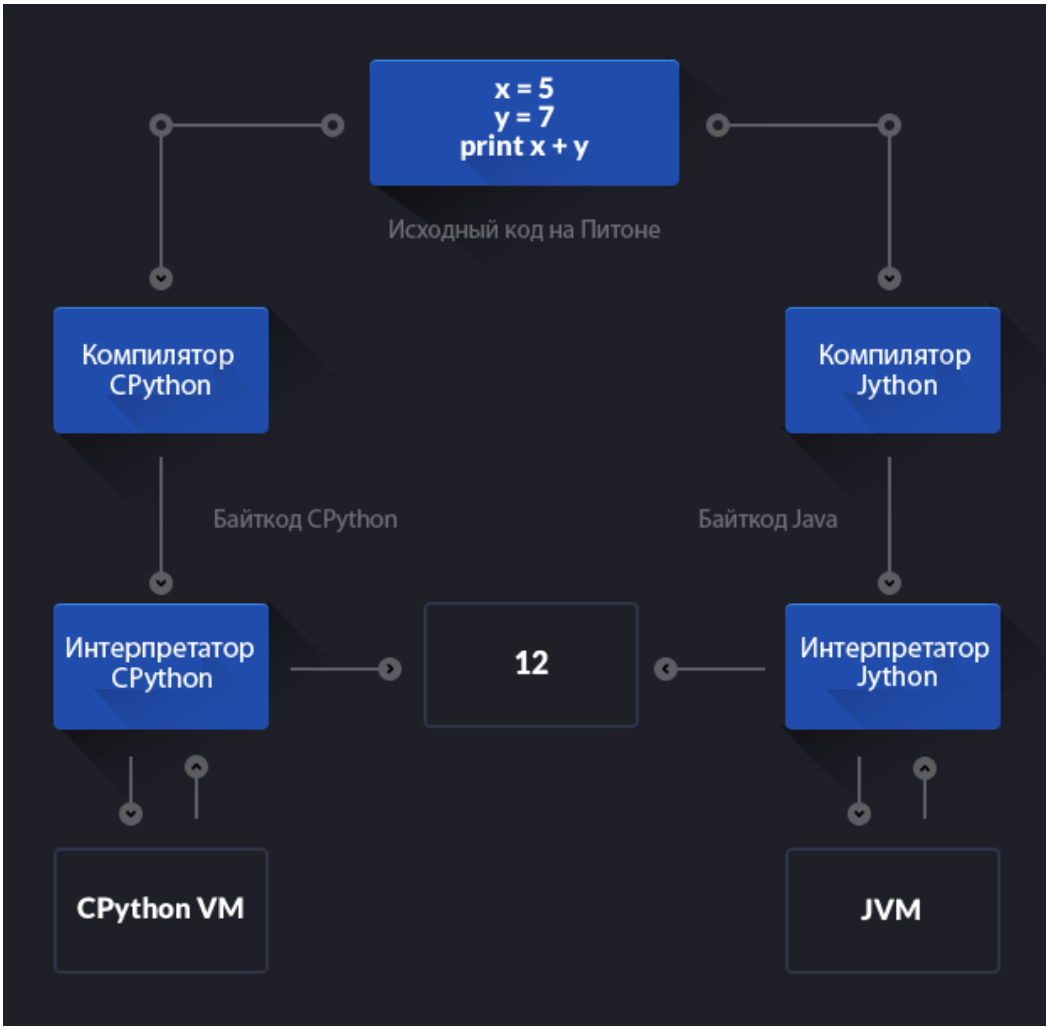
1. CPython компилирует ваш исходный код на Питоне в байткод.
2. Этот байткод запускается на виртуальной машине CPython.

Многие люди, при работе с Питоном часто замечают, что появляются файлы .pyc-файлы – это скомпилированный байткод, который впоследствии интерпретируется. Так что если вы запускали ваш код на Питоне, и у вас есть .pyc-файл, то во второй раз он будет работать быстрее, потому что ему не нужно будет заново компилироваться в байткод.

Jython и IronPython:

Одна из наиболее видных это инструментариев это Jython, реализация Питона на Java, которая использует JVM. В то время как CPython генерирует байткод для запуска на CPython VM, Jython генерирует байткод Java для запуска на JVM (это то же самое, что генерируется при компиляции программы на Java). IronPython это другая популярная реализация Питона, написанная полностью на C# и предназначенная для .NET. В частности, она запускается на виртуальной машине .NET, если ее можно так назвать, на Common Language Runtime (CLR), от Майкрософт, сравнимым с JVM.

“Зачем может понадобиться использовать альтернативную реализацию?”, спросите вы. Ну, для начала, разные реализации хорошо ладят с разными наборами технологий.



CPython упрощает написание C-расширений для кода на Питоне потому что в конце он запускается интерпретатором Си. Jython в свою очередь упрощает работу с другими программами на Java: вы можете импортировать любые Java-классы без дополнительных усилий, призывая и используя ваши Java-классы из программ на Jython. Всё то же самое будет

верно и про IronPython, так как в итоге аналогично происходит на язык C# и преобразование в MSIL – промежуточный ассемблер.

Implementation	Virtual Machine	Ex) Compatible Language
CPython	CPython VM	C
Jython	JVM	Java
IronPython	CLR	C#
Brython	Javascript engine (e.g., V8)	JavaScript
RubyPython	Ruby VM	Ruby

toptal

Вполне реально выжить, не прикасаясь ни к чему, кроме CPython. Но, переходя на другие имплементации, вы получаете преимущество, в основном из-за используемого стека технологий. Используете много языков, основанных на JVM? Jython может вам подойти. Все на .NET? Возможно, стоит попробовать IronPython.

PyPy:

Итак, у нас есть имплементация Питона, написанная на Си, еще одна – на Java, и третья на C#. Следующий шаг: имплементация Питона, написанная на… Питоне с JIT-компиляцией. Но сначала, зачем нам нужна такая компиляция?

Нативный машинный код намного быстрее байткода – это нам известно. Ну, а что, если бы можно было компилировать часть байткода и запускать его как нативный код? Пришлось бы “заплатить” некоторую цену (время) за компиляцию байткода, но если результат будет работать быстрее, то это здорово! Этим и мотивируется JIT-компиляция, гибридная техника, которая совмещает в себе преимущества интерпретаторов и компиляторов. Вкратце, JIT старается использовать компиляцию, чтобы ускорить систему интерпретации.

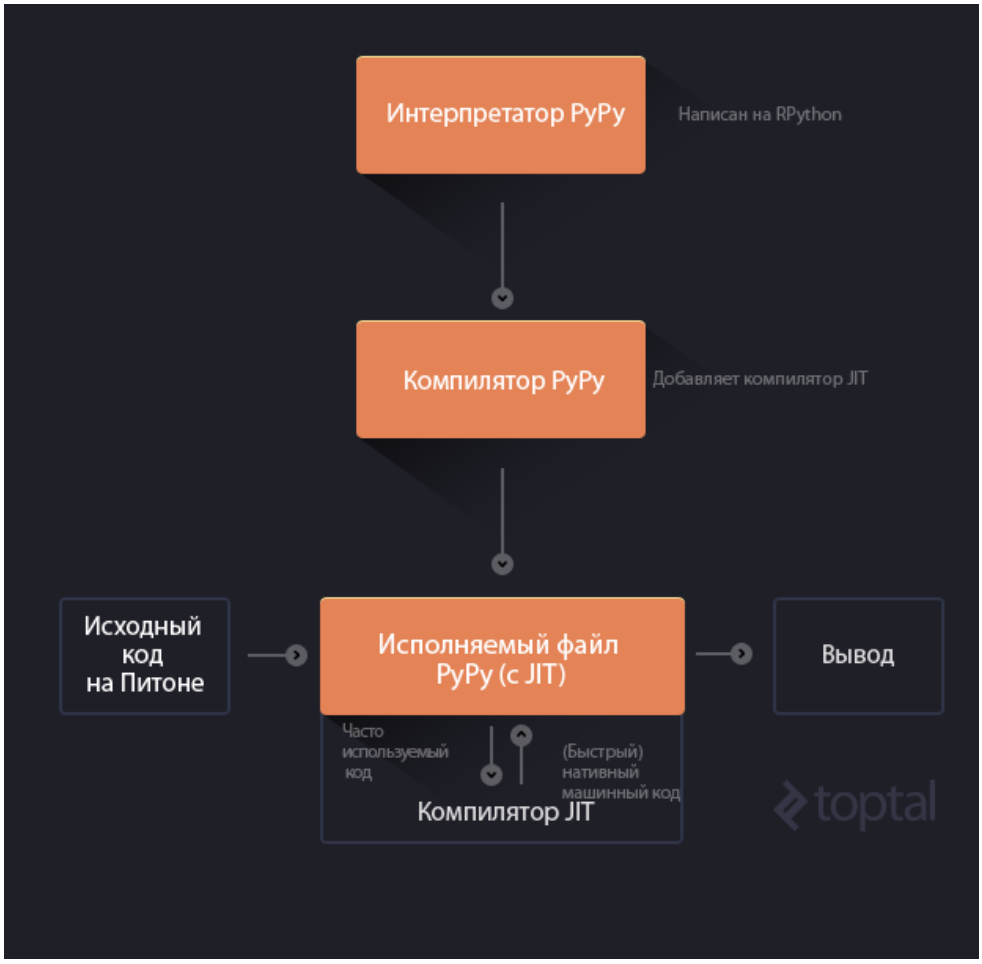
Основные операции JIT-компилятора:

1. Определить байткод, который запускается часто.
2. Скомпилировать его в нативный машинный код.

3. Закэшировать результат.
4. Всегда когда необходимо запустить тот же самый байткод, использовать уже скомпилированный машинный код получив тем самым прирост скорости.

В этом вся суть PyPy: использовать JIT в Питоне. Конечно, есть и другие цели: PyPy нацелен на кроссплатформенность, работу с небольшим количеством памяти и поддержку stackless (отказа от стека вызовов языка Си в пользу собственного стека). Но JIT это главное преимущество. Но с PyPy есть много путаницы. По моему мнению основная причина в том, что PyPy одновременно является:

1. Интерпретатором Питона, написанным на RPython - это подмножество Python со статичной типизацией (в RPython мы жертвуем некоторой гибкостью, но взамен получаем возможность гораздо проще управлять памятью и много чего еще, что помогает при оптимизации).
2. Компилятором, который компилирует код на RPython в разные форматы и поддерживает JIT. Платформой по-умолчанию является Си, то есть компилятор RPython-в-Си, но в качестве целевой платформы также можно выбрать JVM и другие.
3.



В конце концов результатом будет самостоятельный исполняемый файл, который интерпретирует исходный код на Питоне и использует оптимизацию JIT.

На самом деле, если копнуть глубже в абстракцию, теоретически можно написать интерпретатор любого языка, направить его в PyPy и получить JIT для этого языка. Это возможно потому, что PyPy концентрируется на оптимизации самого интерпретатора, а не деталей языка, который тот интерпретирует.

В качестве отступления я бы хотел заметить, что сам JIT совершенно восхитителен. Он использует технику под названием “отслеживание” (tracing), которая работает следующим образом:

1. Запустить интерпретатор и интерпретировать все (не добавляя JIT).
2. Провести легкое профилирование интерпретированного кода.
3. Определить операции, которые уже выполнялись ранее.
4. Скомпилировать эти части кода в машинный код.

Внутреннее устройство интерпретатора CPython

Я опишу работу Питона в коде CPython версии не ниже чем 3.5, который собран и запущен на ОС Linux, где покажу, что питоновские внутренности - это совсем не страшно (я про интерпретатор), в них может разобраться любой желающий.

Подготовительный этап:

Для того, чтобы узнать внутреннюю структуру CPython-а, нужно… скачать сам исходный код CPython-а! Чтобы это сделать, вам нужно зайти на github.com/python и скачать исходный код проекта. Затем, для корректной работы, нужно провести небольшие манипуляции с файлами: Если вы собираетесь экспериментировать с этими исходниками, то вот вам небольшой совет – если вы скачали tarball, рекомендую инициализировать внутри "git" репозиторий, добавить в «.gitignore» те бинарные файлы, которые произведет компилятор в процессе сборки и закоммитить («git commit») это состояние, на которое, если что, можно сделать восстановление (reset). Стандартный скрипт configure нужно будет запустить с ключем «--with-pydebug», это поможет нам в дальнейшем, и, напоследок, полезно прогнать тесты. В результате должен получиться полностью работоспособный интерпретатор.

Этап 1. Вскрытие:

Итак, с чего бы нам начать свой путь от полной несознанки? Взгляните на исходный код, которые вы только что скачали - дюжина директорий с неведомым содержимым. Совершенно не очевидно, что из этого является ядром системы и то, как отдельные компоненты связаны друг с другом. Думаю, что в таких условиях наиболее естественной была бы попытка пройти вместе с интерпретатором весь путь от начальной инициализации до интерпретации питоновской программы, что дало бы минимальное представление о сути проблемы.

Давайте использовать GDB - GNU Debugger. Не пугайтесь, нам понадобится лишь самый минимум его возможностей, помимо этого мы частенько будем подглядывать в исходники - вы можете делать это прямо из отладчика.

Запускать мы будем вот такой питоновский код в файле с именем «probe0.py»:

```
import os
```

```
import signal

os.kill(os.getpid(), signal.SIGTRAP)
```

```
gdb --args path/to/our/python path/to/probe0.py
GNU gdb (GDB) Fedora 7.8.2-38.fc21
Copyright (C) 2014 Free Software Foundation, Inc.
# blah blah
(gdb) run
Program received signal SIGTRAP, Trace/breakpoint trap.
0x00007ffff7127c07 in kill () from /lib64/libc.so.6
(gdb)
```

Что это и почему оно такое? Перед тем как начать выполнять какой-либо скрипт интерпретатор долго готовится, в частности, грузит всякое-разное из стандартной библиотеки, поэтому чтобы поймать в отладчике выполнение именно нашего скрипта мы пошлем себе SIGTRAP, который и будет перехвачен GDBой - вот такой нехитрый трюк.

```
#0-6 are skipped...
#7  PyEval_EvalFrameEx (f=0xa0c578, ...) -------- at Python/ceval.c:2835
#8  PyEval_EvalCodeEx (_co=0x7ffff70c7340, ...) - at Python/ceval.c:3586
#9  PyEval_EvalCode (co=0x7ffff70c7340, ...) ---- at Python/ceval.c:773
#10 run_mod (mod=0xa46768, ...) ----------------- at Python/pythonrun.c:2180
#11 PyRun_FileExFlags (fp=0x9ed880, ...) -------- at Python/pythonrun.c:2133
#12 PyRun_SimpleFileExFlags (fp=0x9ed880, ...) -- at Python/pythonrun.c:1606
#13 PyRun_AnyFileExFlags (fp=0x9ed880, ...) ----- at Python/pythonrun.c:1292
#14 run_file (fp=0x9ed880, ...) ----------------- at Modules/main.c:319
#15 Py_Main (argc=2, ...) ----------------------- at Modules/main.c:751
#16 main (argc=2, ...) -------------------------- at Modules/python.c:69
```

Мы имеем интерпретатор, остановленный на строке с лямбдой. Чтобы посмотреть нативный «callstack», достаточно вызвать команду «backtrace» (или ее короткую версию bt).

Первый столбик - это номера «activation frame»`ов C-шных функций интерпретатора, которые лежат в стековой памяти процесса в виде непрерывной последовательности. Очевидно, что в самом начале вызывается функция «main» - 16-ичный фрейм. Она выполняет самые первичные проверки: хватает ли памяти на запуск, удается ли декодировать аргументы командной строки и пр. Дальше, убедившись, что всё в порядке, управление передается функции «Py_Main». Пока мы не сможем оценить её роли, но именно там происходит вызов «Py_Initialize», который создает новый экземпляр интерпретатора и выполняет его первоначальную настройку. Далее, вплоть до 10 фрейма включительно, система пытается собрать экземпляр «PyCodeObject», где в этом процессе участвует

компилятор, на вход которому подается длинная строка из «*.py» файла, командной строки с ключем «-c» или, например, пользовательского ввода из REPL-a. Так или иначе должен получиться "код", способный управлять интерпретатором точно так же, как машинные инструкции управляют аппаратным процессором. Функция «PyEval_EvalCode» есть тончайшая обертка вокруг «PyEval_EvalCodeEx», которая как раз и решает, что делать с тем кодом, который собрался из нашего кода программы.

Тут я понял, что в CPython существует явным образом выделенная виртуальная машина, представленная функцией «PyEval_EvalCodeEx» и, во-вторых, эта машина на вход принимает специально собранный "код", представленный на низком уровне типом «PyCodeObject».

2 Этап.Виртуальная машина:

```
PyObject *
PyEval_EvalCodeEx(PyObject *_co, PyObject *globals, PyObject *locals, ...)
{
    PyCodeObject* co = (PyCodeObject*)_co;
    PyFrameObject *f;
    f = PyFrame_New(tstate, co, globals, locals);
    /* usualy it evals frame */
    return PyEval_EvalFrameEx(f,0);

}
```

В сердце CPython лежит довольно простая стековая машина, управляемая байткодом, являющимся частью code object-a. Пойдем по порядку - сначала вызывается функция:

Фактически, все что она делает - это создает новый питоновский фрейм «f» и связывает с ним объект кода «_co», который нужно будет исполнить. Непосредственно в этот момент ваша программа еще не исполняется - дальше "обычно" управление передается функции «PyEval_EvalFrameEx», умеющей запускать фреймы. На вход подается указатель на питоновский фрейм. ***Фрейм*** - это внутреннее представление исполняющегося кода, то есть контейнер с кодом и метаданными внутри. Именно фрейм хранит информацию о том, какой «опкод» сейчас исполняется и куда передать управление, когда код отработает. Последнее достигается хранением в атрибуте «f_back» ссылки на родительский фрейм, так что множество фреймов образует односвязный список. На один и тот же объект кода могу ссылаться множество фреймов, так что объект кода - это что-то вроде шаблона.

Питоновский фрейм очень похож на нативный фрейм, который создала бы программа на Си, внутри которой происходит вызов подпрограммы, но в конечном счёте - это обычные объекты, у них есть свой конструктор, память под них выделяется на куче, а не из стековой памяти, как для нативных фреймов, а также они подвержены сборке мусора. Подобно другим объектам, фреймы исполнения имеют представление внутри питоновского кода, что вкупе с другими возможностями делает питон языком с очень богатой интроспекцией.

Вернемся к «PyEval_EvalFrameEx». Виртуальная машина, перебирая эти «опкоды» один за другим выполняет примитивные атомарные операции - программа на языке Питон выполняется.

3 Этап. Объект кода:

```
>>> code = compile('print(a + foo.bar())', '<string>', 'exec')
```

Экземпляр питоновского кода возникает всякий раз, когда компилятор видит независимую единицу исполнения - модуль, функцию (метод) или класс. Помимо этого код можно собрать самостоятельно из произвольного набора выражений с помощью встроенной функции «compile», например так

Обратите внимание, что переменные нигде не определены, при этом компилятор считает этот код совершенно валидным.

```
>>> class Mock:
...     pass
...
... Mock.bar = lambda: 'world!'
... eval(code, {'a':'Hello', 'foo': Mock})
Helloworld!
```

Мы можем запустить его, предоставив контекст:

Дизассемблируем, используя модуль «dis» в стандартной библиотеке, объекты питоновского кода:

```
>>> dis.disassemble(code)
  1           0 LOAD_NAME                0 (print)
              3 LOAD_NAME                1 (a)
              6 LOAD_NAME                2 (foo)
              9 LOAD_ATTR                3 (bar)
             12 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
             15 BINARY_ADD
             16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             19 POP_TOP
             20 LOAD_CONST               0 (None)
             23 RETURN_VALUE
```

Что мы видим? На стек нашей стековой машины последовательно грузятся сущности под именами «print, a, foo» и так далее. Откуда происходит эта загрузка? Точно не из байткода, потому что на момент компиляции корректность таких ссылок никто не гарантировал. Значит существует контекст, объект кода знает о его интерфейсе (он грузит из него сущности, зная их имена), но не знает о реализации, т.е. что там лежит. Это полезно в случае рекурсии, когда один и тот же код переиспользуется многократно внутри разных контекстов.

Детали реализации «PyCodeObject» вы можете посмотреть в исходном коде (Include/code.h), но вот что следует выделить из полученного:

1. Он включает в себя байтовую строку - байткод, который непосредственно рулит виртуальной машиной.
2. Этот байткод запускается внутри определенного контекста, для обращения к которому ему так же нужен интерфейс.
3. Байткод и интерфейс собраны в один объект - объект кода.
4. Контекст является внешней по отношению к коду сущностью.
5. Фрейм исполнения содержит в себе ссылку на объект кода и на контекст, когда исполняется ваш питоновский скрипт, на низком уровне запущена «PyEval_EvalFrameEx».

Это базовые вещи, о которых стоило описать. Дальше идут тонкости разработчиков Python-а, о которых не имеет смысла описывать. Пора перейти к собственному интерпретатору.

3. Результат работы интерпретатора

Точкой входа в программу служит файл *interpretator.py*, в котором задействованы модули из предыдущих работ:

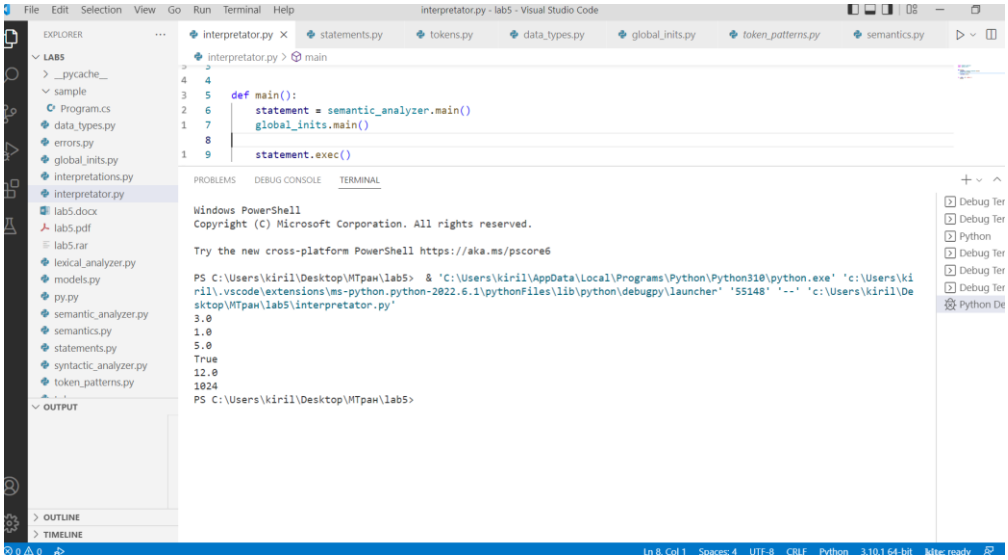
Interpretator.py:

```
1   import semantic_analyzer
2   import global_inits
3
4
5   def main():
6       statement = semantic_analyzer.main()
7       global_inits.main()
8
9       statement.exec()
10
11
```

Semantic_analyzer.py:

```
1   import syntactic_analyzer
2
3
4   def main():
5       statement = syntactic_analyzer.main()
6
7       statement.semantic_analyze()
8
9       return statement
10
```

Вывод выполнения исследуемого файла:



4. Выводы

Я изучил основы методов трансляции и разработал собственный интерпретатор, работающий на 3 основных фазах: лексический, синтаксический и семантический анализаторы. Язык программирования Python оказался очень достойным языком для изучения данного раздела и Lisp, как схожий в отдельных частях (динамичный и функциональный), был отличным инструментом, для изучения.

Сложность заключалась в реализации конечного автомата, способного самостоятельно, используя анализаторы, реализовать конечную задумку и выдать результат обработки кода на C#.

Язык разбора Python — высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода, в то же время стандартная библиотека включает большой объём полезных функций. Основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных. Код в Python организовывается в функции и классы, которые могут объединяться в модули (они в свою очередь могут быть объединены в пакеты).

Популярной реализацией Python является интерпретатор CPython, поддерживающий большинство активно используемых платформ. Он распространяется под свободной лицензией Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях. Есть реализации интерпретаторов для JVM, MSIL, LLVM и других