Содержание

[PAGEREF \_Toc4172825 \hError: Reference source not found](#_Toc4172825)

1. Постановка задачи 3

2[. Теория](#_Toc4172826) 4

[PAGEREF \_Toc4172827 \hError: Reference source not found](#_Toc4172827)

3[.](#_Toc4172828) Результаты работы интерпретатора на коде с ошибками16

4[. Результат работы интерпретатора](#_Toc4172828)

[Выводы](#_Toc4172828) 16

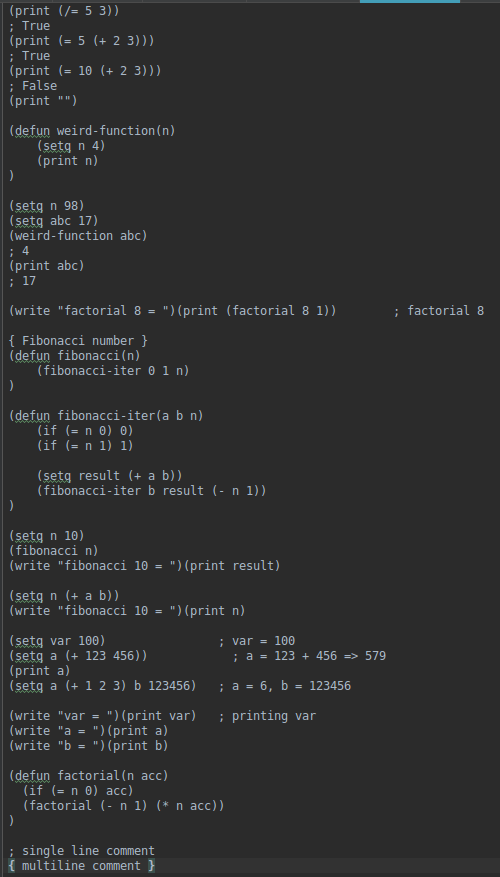
[Приложение А. - Исходный код анализирумой программы 2](#_Toc4172829)0

[Приложение Б. - Исходный код интепретатора](#_Toc4172829) 23

1. **Постановка задачи**

В данной работе ставится задача полной реализации интерпретатора языка Lisp на объектно-ориентированном языке программирования Python. Основной целью работы является реализация виртуальной машины, способной обрабатывать команды языка Lisp и выдавать результат при исполнении, то есть быть интерпретатором. Это является последним шагом на пути в создании программы, способной выполнять программу самостоятельно. Данная работа была дополнена классов в файле vm.py, Procedure, а также главной точкой входа выполнения интерпретирующей машины в файле app.py, где соединены все анализаторы предыдущих лабораторных работ.

Исследуемый код языка представлен ниже:



*Рис. 1.1. Исходный код программы на языке Lisp*

1. **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:

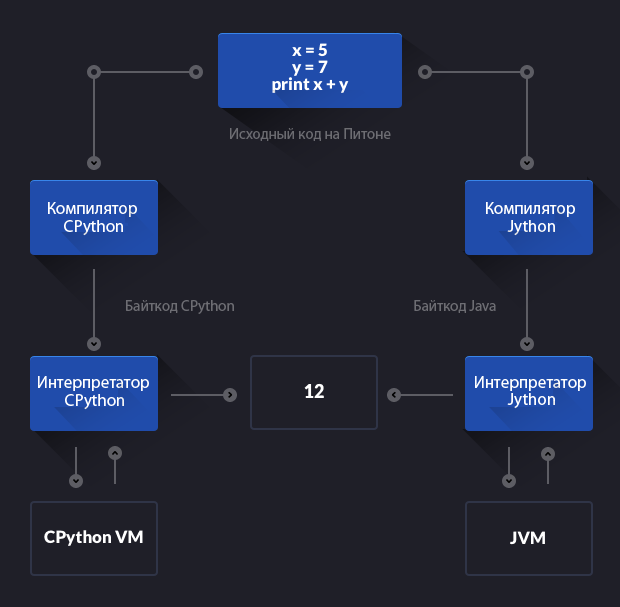
1. CPython компилирует ваш исходный код на Питоне в байткод.
2. Этот байткод запускается на виртуальной машине CPython.

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.

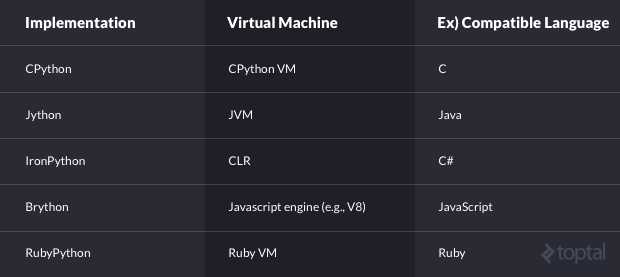
***Jython и IronPython:***

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) (Рис. 2.1).. 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 Cи. 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 (Рис. 2.2).

*Рис. 2.1. Исходный код программы на языке Python, который запустили на разных компиляторах (CPython и Jython).*

*Рис. 2.2. Соответствие инструментария, виртуальной машины и код, в который питон будет скомпилирован.*

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 c JIT-компиляцией. Но сначала, зачем нам нужна такая компиляция?

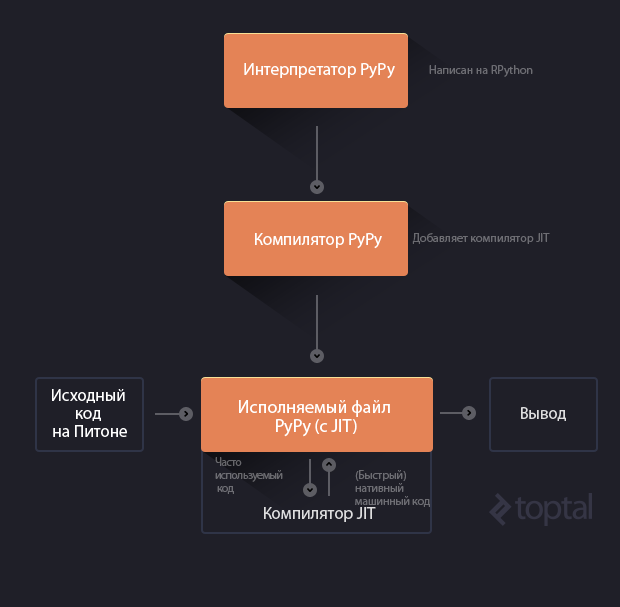
Нативный машинный код намного быстрее байткода – это нам известно. Ну, а что, если бы можно было компилировать часть байткода и запускать его как нативный код? Пришлось бы “заплатить” некоторую цену (время) за компиляцию байткода, но если результат будет работать быстрее, то это здорово! Этим и мотивируется 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.

В конце концов результатом будет самостоятельный исполняемый файл, который интерпретирует исходный код на Питоне и использует оптимизацию JIT (Рис.2.3).

*Рис. 2.3. Схема работы PyPy.*

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.

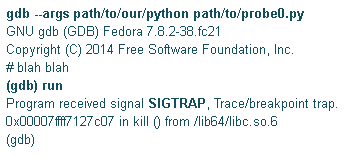
Давайте использовать GDB - GNU Debugger. Не пугайтесь, нам понадобится лишь самый минимум его возможностей, помимо этого мы частенько будем подглядывать в исходники - вы можете делать это прямо из отладчика.

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

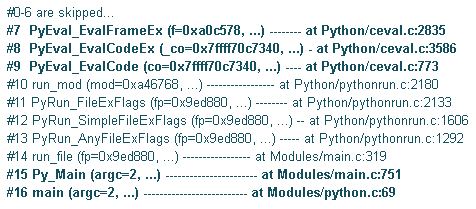
import os

import signal

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

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 (Рис.2.4).

*Рис. 2.4. Результат первого перехвата.*

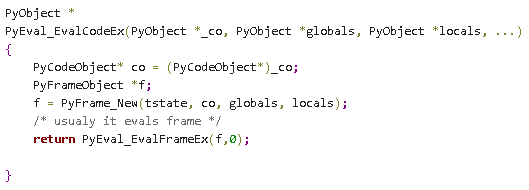
Мы имеем интерпретатор, остановленный на строке с лямбдой. Чтобы посмотреть нативный «callstack», достаточно вызвать команду «backtrace» (или ее короткую версию bt).

*Рис. 2.5. Результат «backtrace» первого перехвата.*

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 «-с» или, например, пользовательского ввода из REPL-а. 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:***

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.

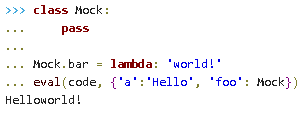
Питоновский фрейм очень похож на нативный фрейм, который создала бы программа на Си, внутри которой происходит вызов подпрограммы, но в конечном счёте - это обычные объекты, у них есть свой конструктор, память под них выделяется на куче, а не из стековой памяти, как для нативных фреймов, а также они 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.

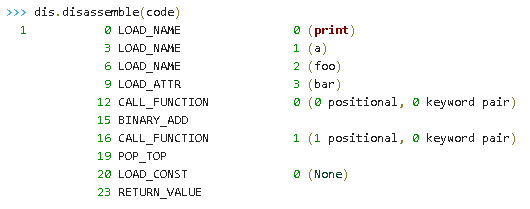
***3 Этап. Объект кода:***

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.

Мы можем запустить его, предоставив контекст:

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



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.

1. 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. Результаты работы интерпретатора на коде с ошибками:**

*Изменим исходный код программы, сознательно сделав несколько семантических ошибок.*

Код программы с 1-ой ошибкой Рис. 4.1. Результат работы программы с допущением 1-ой ошибки Рис. 4.2.

*Рис. 4.1. Семантическая ошибка c попыткой сложения строки и числа*

*Рис. 4.2. Результат работы программы с допущением 1-ой ошибки*

Код программы с 2-ой ошибкой Рис. 4.3. Результат работы программы с допущением 2-ой ошибки Рис. 4.4.

*Рис. 4.3. Семантическая ошибка c попыткой умножения строки и числа*

*Рис. 4.4. Результат работы программы с допущением 2-ой ошибки*

Код программы с 3-ей ошибкой Рис. 4.5. Результат работы программы с допущением 3-ей ошибки Рис. 4.6.

*Рис. 4.5. Семантическая ошибка c попыткой ссылки на функцию со строкой*

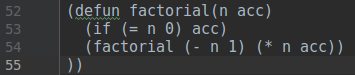
*Рис. 4.6. Результат работы программы с допущением 3-ей ошибки*

Код программы с 4-ой ошибкой Рис. 4.7. Результат работы программы с допущением 4-ой ошибки Рис. 4.8.

*Рис. 4.7. Семантическая ошибка cо сложением ссылки одного метода на ссылку второго*

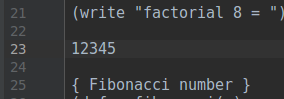
*Рис. 4.8. Результат работы программы с допущением 4-ой ошибки*

Код программы с 5-ой ошибкой Рис. 4.9. Результат работы программы с допущением 5-ой ошибки Рис. 4.10.

*Рис. 4.9. Синтаксическая ошибка в анализируемом коде (добавление лишней скобки)*

*Рис. 4.10. Результат работы с допущением 1-ой ошибки*

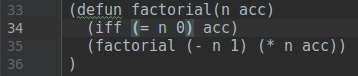
Код программы со 6-ой ошибкой Рис. 4.11. Результат работы программы с допущением 6-ой ошибки Рис. 4.12.



*Рис. 4.11. Синтаксическая ошибка (введение числа в случайное место)*

*Рис. 4.12. Результат работы программы с допущением 2-ой ошибки*

Код программы с 7-ей ошибкой Рис. 4.13. Результат работы программы с допущением 7-ей ошибки Рис. 4.14.

*Рис. 4.13. Синтаксическая ошибка в цикле IF*

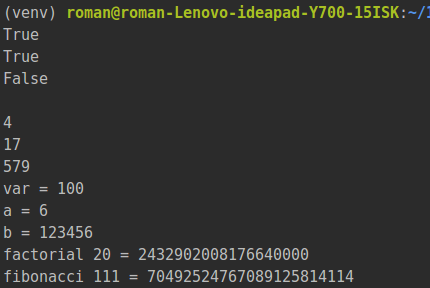
*Рис. 4.14. Результат работы программы с допущением 3-ой ошибки*

Код программы с 8-ой ошибкой Рис. 4.15. Результат работы программы с допущением 8-ой ошибки Рис. 4.16.

*Рис. 4.15. Синтаксическая ошибка использование лишней скобки*

*Рис. 4.16. Результат работы программы с допущением 4-ой ошибки*

**4. Результат работы интерпретатора**

Для создания анализатора я воспользовался предыдущими 3 лабораторными работами и как результат, получил готовый интерпретатор, в котором можно создавать различные программы. Данная программа проводит не сложное вычисление и выдаёт сложный результат, за счёт хвостовой рекурсии. Данный режим рекурсии включён в интерпретатор и включается при несложном условии.

*Рис. 3.1. На рисунке представлен результат работы программы: присваивание и работа с числами, вычисление факториала, числа фибоначчи через хвостовую рекурсию. Нахождение факториала 20 и числа Фибоначи от 111.*

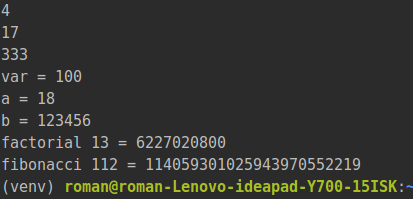
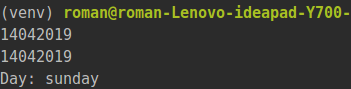
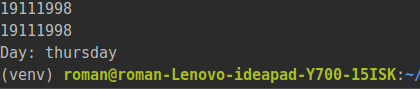
*Рис. 3.2. На рисунке представлен результат работы программы: присваивание и работа с числами, вычисление факториала, числа фибоначчи через хвостовую рекурсию. Нахождение факториала 13 и числа Фибоначи 112*

Рис. 4.1. На рисунке представлен результат работы программы: Вычисление дня недели по дате в формате DDWWYYYY. Дата 14.04.2019

Рис. 4.2. На рисунке представлен результат работы программы: Вычисление дня недели по дате в формате DDWWYYYY. Дата 19.11.1998

*Таким образом была продемонстрирована возможность интерпретации кода на языке Lisp.*

**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 Lisp.Создание виртуальной машины – задача не из лёгких, но с этим я справился!

Язык разбора Python — [высокоуровневый язык программирования](https://ru.wikipedia.org/wiki/Высокоуровневый_язык_программирования) общего назначения, ориентированный на повышение производительности разработчика и читаемости кода, в то же время [стандартная библиотека](https://ru.wikipedia.org/wiki/Стандартная_библиотека_Python) включает большой объём полезных функций. Основные архитектурные черты — [динамическая типизация](https://ru.wikipedia.org/wiki/Динамическая_типизация), [автоматическое управление памятью](https://ru.wikipedia.org/wiki/Сборка_мусора_(программирование)), полная [интроспекция](https://ru.wikipedia.org/wiki/Интроспекция_(программирование)), механизм [обработки исключений](https://ru.wikipedia.org/wiki/Обработка_исключений), поддержка [многопоточных вычислений](https://ru.wikipedia.org/wiki/Многопоточность) и удобные высокоуровневые [структуры данных](https://ru.wikipedia.org/wiki/Структура_данных). Код в Python организовывается в функции и [классы](https://ru.wikipedia.org/wiki/Класс_(программирование)), которые могут объединяться в [модули](https://ru.wikipedia.org/wiki/Модуль_(программирование)) (они в свою очередь могут быть объединены в пакеты).

Популярной реализацией Python является интерпретатор [CPython](https://ru.wikipedia.org/wiki/CPython), поддерживающий большинство активно используемых платформ. Он распространяется под [свободной лицензией](https://ru.wikipedia.org/wiki/Свободное_программное_обеспечение) Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях. Есть реализации интерпретаторов для [JVM](https://ru.wikipedia.org/wiki/JVM), [MSIL](https://ru.wikipedia.org/wiki/MSIL), [LLVM](https://ru.wikipedia.org/wiki/LLVM) и других

## **Приложение А. - Исходный код анализируемой программы**

Данный листинг с кодом демонстрирует исходный код с присваиванием и работе с числами, вычислением факториала, числом фибоначчи через хвостовую рекурсию

; однострочный комментарий

{ многострочный комментарий }

(print (/= 5 3))

; True

(print (= 5 (+ 2 3)))

; True

(print (= 10 (+ 2 3)))

; False

(print "")

(defun weird-function(n)

(setq n 4)

(print n)

)

(setq n 98)

(setq abc 17)

(weird-function abc)

; 4

(print abc)

; 17

(setq var 100) ; var = 100

(setq a (+ 123 456)) ; a = 123 + 456 => 579

(print a)

(setq a (+ 1 2 3) b 123456) ; a = 6, b = 123456

(write "var = ")(print var) ; printing var

(write "a = ")(print a)

(write "b = ")(print b)

(defun factorial(n acc)

(if (= n 0) acc)

(factorial (- n 1) (\* n acc))

)

(write "factorial 20 = ")(print (factorial 20 1)) ; factorial 20

{ Fibonacci number }

(defun fibonacci(n)

(fibonacci-iter 0 1 n)

)

(defun fibonacci-iter(a b n)

(if (= n 0) 0)

(if (= n 1) 1)

(setq result (+ a b))

(fibonacci-iter b result (- n 1))

)

(setq n 111)

(fibonacci n)

(write "fibonacci 111 = ")(print result)

Данный листинг с кодом демонстрирует исходный код с поиском дня недели по дате

; найти какой день был под данной датой

(defun whatDay(day month year)

(setq a (// (- 14 month) 12))

(setq y (- year a))

(setq m (- (+ (\* 12 a) month) 2))

(setq ab (// (\* 31 m) 12))

(setq bc (- (// y 4) (// y 100)))

(setq xy (+ day y bc (// y 400) ab))

(setq result (- (% (+ 7000 xy) 7) 1))

(setq result (reculc result))

(if (= result (- 0 1)) "sunday")

(if (= result 0) "monday")

(if (= result 1) "tuesday")

(if (= result 2) "wednesday")

(if (= result 3) "thursday")

(if (= result 4) "friday")

(if (= result 5) "saturday")

(if (= result 6) "sunday" result)

)

(defun delneg(n)

(if (< n 0) (- 0 n) n)

)

(defun ifneg(n)

(if (< n 0) 1 0)

)

(defun reculc(result)

(if (= isneg 1) (- result 1) result)

)

(readint n)

(setq isneg (ifneg n))

(setq n (delneg n))

(setq curyear (% n 10000))

(setq n (// n 10000))

(setq curmonth (% n 100))

(setq n (// n 100))

(setq curday (% n 100))

(setq n (// n 100))

; (print curyear)

; (print curmonth)

; (print curday)

(print "")

(write "Day: ")

(print (whatDay curday curmonth curyear))

## **Приложение Б. - Исходный код интерпретатора**

Данный листинг с кодом демонстрирует дополненость необходимых операций в код программы, для создания интерпретатора. Создан главный скрипт с соединением предыдущих анализаторов и дополнен vm.py – файл виртуальной машины.

import os

import sys

import texttable as tt

from lib.lexer import Lexer

from lib.parser import Parser

from lib.vm import GLOBAL, execute

def lab\_1*(*args, tokens*)*:

if "lab\_1" in args:

tab = tt.Texttable*()*

headings = *[*'Value (token)', 'Tag', 'Row', 'Column'*]*

tab.header*(*headings*)*

values = list*()*

tags = list*()*

rows = list*()*

columns = list*()*

for token in tokens:

values.append*(*token.value*)*

tags.append*(*token.tag*)*

rows.append*(*token.row*)*

columns.append*(*token.col*)*

for row in zip*(*values, tags, rows, columns*)*:

tab.add\_row*(*row*)*

s = tab.draw*()*

print*(*s*)*

def lab\_2*(*args, ast, tabs*)*:

if "lab\_2" in args:

for i in ast:

if isinstance*(*i, list*)*:

lab\_2*(*args, i, tabs + 1*)*

else:

result = tabs \* ' |'

print*(*'{}{}'.format*(*result, i.value*))*

def lab\_3*(*args*)*:

if "lab\_3" in args:

tab = tt.Texttable*()*

headings = *[*'Variable', 'Type'*]*

tab.header*(*headings*)*

variables = list*()*

types = list*()*

for key, value in GLOBAL.items*()*:

if isinstance*(*value, int*)* or isinstance*(*value, str*)*:

variables.append*(*key*)*

types.append*(*type*(*value*))*

for row in zip*(*variables, types*)*:

tab.add\_row*(*row*)*

s = tab.draw*()*

print*(*s*)*

def main*(*args*)*:

*"""*

*main*

*"""*

path = args*[*0*]*

if len*(*path*)* > 1:

if not os.path.exists*(*path*)*:

print*(*'Error! File "%s" not found!' % path*)*

exit*(*1*)*

else:

print*(*'Error! Expected file, but given nothing!'*)*

exit*(*1*)*

args = args*[*1:*]*

lexer = Lexer*(*path*)*

parser = Parser*()*

tokens = lexer.tokens*()*

if lexer.errors\_list:

lexer.errors*()*

lab\_1*(*args, tokens*)*

try:

ast = parser.build*(*tokens*)*

lab\_2*(*args, ast, 3*)*

""" Lab 4 """

for i in ast:

execute*(*i, GLOBAL*)*

lab\_3*(*args*)*

except Exception as ex:

print*(*ex*)*

exit*(*1*)*

if \_\_name\_\_ == '\_\_main\_\_':

main*(*sys.argv*[*1:*])*

import operator as op

from .lexer import ID, Token

GLOBAL = dict*()*

GLOBAL*[*'+'*]* = lambda env, \*x: obs*(*env, op.add, \*x*)*

GLOBAL*[*'-'*]* = lambda env, \*x: obs*(*env, op.sub, \*x*)*

GLOBAL*[*'\*'*]* = lambda env, \*x: obs*(*env, op.mul, \*x*)*

GLOBAL*[*'/'*]* = lambda env, \*x: obs*(*env, op.truediv, \*x*)*

GLOBAL*[*'//'*]* = lambda env, \*x: obs*(*env, op.floordiv, \*x*)*

GLOBAL*[*'%'*]* = lambda env, \*x: obs*(*env, op.mod, \*x*)*

GLOBAL*[*'='*]* = lambda env, \*x: obs*(*env, op.eq, \*x*)*

GLOBAL*[*'/='*]* = lambda env, \*x: obs*(*env, op.ne, \*x*)*

GLOBAL*[*'>'*]* = lambda env, \*x: obs*(*env, op.gt, \*x*)*

GLOBAL*[*'<'*]* = lambda env, \*x: obs*(*env, op.lt, \*x*)*

GLOBAL*[*'>='*]* = lambda env, \*x: obs*(*env, op.ge, \*x*)*

GLOBAL*[*'<='*]* = lambda env, \*x: obs*(*env, op.le, \*x*)*

GLOBAL*[*'~'*]* = lambda env, \*x: obs*(*env, op.ne, \*x*)*

GLOBAL*[*'setq'*]* = lambda env, \*x: setq*(*env, \*x*)*

GLOBAL*[*'defun'*]* = lambda env, \*x: defun*(*env, \*x*)*

GLOBAL*[*'if'*]* = lambda env, \*x: compare*(*env, \*x*)*

GLOBAL*[*'write'*]* = lambda env, \*x: write*(*env, \*x*)*

GLOBAL*[*'print'*]* = lambda env, \*x: write\_line*(*env, \*x*)*

GLOBAL*[*'readint'*]* = lambda env, \*x: readint*(*env, \*x*)*

class Procedure*(*object*)*:

*"""*

*procedure*

*"""*

def \_\_init\_\_*(*self, params, \*body*)*:

*"""*

*params, expr, env*

*"""*

self.params, self.body = params, body

def \_\_call\_\_*(*self, env, \*args*)*:

if len*(*args*)* != len*(*self.params*)*:

msg = "Too many args! Expected %s, given %s" % *(*len*(*self.params*)*, len*(*args*))*

msg += ' in line {}, column {}'.format*(*args*[*0*]*.col, args*[*0*]*.row*)*

raise TypeError*(*msg*)*

for i, par in enumerate*(*self.params*)*:

env*[*par.value*]* = execute*(*args*[*i*]*, env*)*

magic = False

while True:

if magic:

for i, par in enumerate*(*self.params*)*:

env*[*par.value*]* = args*[*i*]*

# вычисляем тело функции

length = len*(*self.body*)* - 1

for i, expr in enumerate*(*self.body*)*:

if i < length: # если это не последнее выражение

result = execute*(*expr, env*)*

magic = True

if magic and result:

return result

else:

if isinstance*(*env*[*expr*[*0*]*.value*]*, Procedure*)*:

proc = env*[*expr*[*0*]*.value*]*

self.params = proc.params

self.body = proc.body

args = *[*execute*(*i, env*)* for i in expr*[*1:*]]*

magic = True

else:

result = execute*(*expr, env*)*

return result

def obs*(*env, fun, \*args*)*:

*"""*

*obs*

*"""*

result = execute*(*args*[*0*]*, env*)*

for i in args*[*1:*]*:

result = fun*(*result, execute*(*i, env*))*

return result

def defun*(*env, \*args*)*:

*"""*

*defune new function*

*"""*

name, params, \*body = args

proc = Procedure*(*params, \*body*)*

if not name.value in env:

env*[*name.value*]* = proc

else:

msg = 'Function "%s" already exists!' % name.value

msg += 'in line {}, column {}'.format*(*name.col, name.row*)*

raise Exception*(*msg*)*

def compare*(*env, \*args*)*:

*"""*

*get condition and execute first or second body*

*"""*

if execute*(*args*[*0*]*, env*)*:

return execute*(*args*[*1*]*, env*)*

elif len*(*args*)* == 3:

return execute*(*args*[*2*]*, env*)*

def write*(*env, \*args*)*:

*"""*

*write line*

*"""*

from sys import stdout

stdout.write*(*str*(*execute*(*args*[*0*]*, env*)))*

stdout.flush*()*

def write\_line*(*env, \*args*)*:

*"""*

*write new line*

*"""*

from sys import stdout

stdout.write*(*'%s\n' % str*(*execute*(*args*[*0*]*, env*)))*

stdout.flush*()*

def readint*(*env, \*args*)*:

*"""*

*read line*

*"""*

i = 0

env*[*args*[*i*]*.value*]* = int*(*input*())*

from sys import stdout

if isinstance*(*args*[*i*]*.value, str*)*:

stdout.write*(*str*(*execute*(*args*[*0*]*, env*)))*

stdout.flush*()*

def setq*(*env, \*args*)*:

*"""*

*define new variables*

*"""*

i = 0

while i < len*(*args*)*:

env*[*args*[*i*]*.value*]* = execute*(*args*[*i + 1*]*, env*)*

i += 2

def execute*(*expr, env*)*:

*"""*

*execute*

*"""*

if isinstance*(*expr, Token*)*:

if expr.tag == ID and expr.value in env:

return env*[*expr.value*]*

else:

return expr.value

else:

first, \*second = expr

if first.value in env and callable*(*env*[*first.value*])*:

return env*[*first.value*](*env, \*second*)*

else:

msg = 'Function "%s" not exists!' % first.value

msg += 'in line {}, column {}'.format*(*first.col, first.row*)*

raise Exception*(*msg*)*

RESERVED = 'RESERVED'

UNKNOWN = 'UNKNOWN'

NUMBER = 'NUMBER'

STRING = 'STRING'

QUOTE = 'QUOTE'

ID = 'ID'

class Token:

*"""*

*docstring for Token*

*"""*

def \_\_init\_\_*(*self, value, tag, row, col*)*:

self.value = value

self.tag = tag

self.row = row

self.col = col

def \_\_str\_\_*(*self*)*:

return '<{}, {}, {}, {}>'.format*(*self.value, self.tag, self.row, self.col*)*

def \_\_repr\_\_*(*self*)*:

return self.\_\_str\_\_*()*

class Lexer*(*dict*)*:

*"""*

*docstring for Lexer*

*"""*

def \_\_init\_\_*(*self, file, \*args*)*:

super*()*.\_\_init\_\_*(*\*args*)*

self.pos, self.row, self.col = 0, 1, 1

self.char = ''

self.file = open*(*file, 'r'*)*

self.string = self.file.readline*()*

self.errors\_list = list*()*

def errors*(*self*)*:

*"""*

*print all errors*

*"""*

import sys

self.file.close*()*

sys.stderr.write*(*'Lexer errors:\n'*)*

for i in self.errors\_list:

sys.stderr.write*(*'\t%s\n' % i*)*

sys.stderr.flush*()*

exit*(*1*)*

def error*(*self, text*)*:

*"""*

*print error*

*"""*

self.errors\_list.append*(*

'{} in line {}, column {}'.format*(*text, self.row, self.col*))*

def next\_char*(*self*)*:

*"""*

*set next char*

*"""*

if self.pos < len*(*self.string*)*:

self.char = self.string*[*self.pos*]*

if self.char != '\n':

self.col += 1

self.pos += 1

else:

self.string = self.file.readline*()*

self.col = 1

self.row += 1

self.pos = 0

else:

self.char = '#0'

def skip\_space*(*self*)*:

*"""*

*skip spaces*

*"""*

while self.char.isspace*()*:

self.next\_char*()*

def next\_token*(*self*)*:

*"""*

*return token*

*"""*

self.skip\_space*()*

lexem = ''

# if current char is alpha or \_

if self.char.isalpha*()* or self.char == '\_' or self.char in '+-\*/%><=^!?':

lexem = self.char

self.next\_char*()*

# adding all alpha and digit

while self.char.isalpha*()* or self.char.isdigit*()* or self.char in '+-\*/%><=^!?':

lexem += self.char

self.next\_char*()*

return Token*(*lexem, ID, self.col, self.row*)*

# if current char is digit

elif self.char.isdigit*()*:

# while is digit

count = 0

while self.char.isdigit*()* or self.char == '.':

if self.char == '.':

count += 1

if count > 1:

self.error*(*'Incorrect format of number: "%s"' % self.char*)*

lexem += self.char

self.next\_char*()*

return Token*(*int*(*lexem*)* if count == 0 else float*(*lexem*)*, NUMBER, self.col, self.row*)*

elif self.char in *(*'(', ')'*)*:

lexem = self.char

self.next\_char*()*

return Token*(*lexem, RESERVED, self.col, self.row*)*

elif self.char == '#0':

return Token*(*'EOF', None, self.col, self.row*)*

elif self.char == '-':

lexem = self.char

self.next\_char*()*

if self.char.isdigit*()*:

count = 0

while self.char.isdigit*()* or self.char == '.':

if self.char == '.':

count += 1

if count > 1:

self.error*(*'Incorrect format of number: "%s"' %

self.char*)*

lexem += self.char

self.next\_char*()*

return Token*(*int*(*lexem*)* if count == 0 else float*(*lexem*)*, NUMBER, self.col, self.row*)*

elif self.char in *(*';', '{'*)*:

# skip comments in file

return self.skip\_comments*(*'\n' if self.char == ';' else '}'*)*

elif self.char == '"':

self.next\_char*()*

while self.char != '"':

if self.char == '\\':

lexem += self.char

self.next\_char*()*

lexem += self.char

self.next\_char*()*

continue

lexem += self.char

self.next\_char*()*

self.next\_char*()*

return Token*(*lexem, STRING, self.col, self.row*)*

elif self.char == "'":

lexem = self.get\_quote*()*

return Token*(*lexem*[*1:*]*, QUOTE, self.col, self.row*)*

elif self.char in self:

lexem = self.char

self.next\_char*()*

return Token*(*lexem, self*[*lexem*]*, self.col, self.row*)*

else:

lexem = self.char

self.error*(*'Unknown character: "%s"' % self.char*)*

self.next\_char*()*

return Token*(*lexem, UNKNOWN, self.col, self.row*)*

return None

def get\_quote*(*self, skip\_spaces=True*)*:

*"""*

*return quote*

*"""*

lexem = self.char

if skip\_spaces:

self.skip\_space*()*

self.next\_char*()*

while True:

if self.char == '(':

lexem += self.get\_quote*(*False*)*

if self.char == ')':

lexem += self.char

self.next\_char*()*

if self.char != ')':

lexem += self.char

self.next\_char*()*

else:

return lexem

def skip\_comments*(*self, char*)*:

*"""*

*skip comments*

*"""*

while self.char != char:

self.next\_char*()*

self.next\_char*()*

return self.next\_token*()*

def gettoken*(*self*)*:

*"""*

*return token*

*"""*

self.next\_char*()*

while True:

result = self.next\_token*()*

if not result:

continue

if result.value == 'EOF':

break

yield result

def tokens*(*self*)*:

*"""*

*retun list of tokens*

*"""*

result = *[*i for i in self.gettoken*()]*

return result

def raw\_input*(*self, user\_string*)*:

*"""*

*return raw user input*

*"""*

self.string = user\_string

return self.tokens*()*

class Parser*(*object*)*:

*"""*

*class Parser*

*"""*

def \_\_init\_\_*(*self*)*:

self.tokens = None

def \_node*(*self, pos*)*:

*"""*

*return new node and pos*

*"""*

node = list*()*

while self.tokens*[*pos*]*.value != ')':

if self.tokens*[*pos*]*.value == '(':

new\_node, pos = self.\_node*(*pos + 1*)*

node.append*(*new\_node*)*

else:

node.append*(*self.tokens*[*pos*])*

pos += 1

return node, pos

def build*(*self, tokens*)*:

*"""*

*return ast*

*"""*

ast = list*()*

if tokens:

pos = 0

self.tokens = tokens

while pos < len*(*tokens*)*:

if tokens*[*pos*]*.value == '(':

node, pos = self.\_node*(*pos + 1*)*

pos += 1

ast.append*(*node*)*

else:

msg = 'Parser error! Expected "(" but given "%s"' % tokens*[*pos*]*.value

msg += ' in line {}, column {}'.format*(*tokens*[*pos*]*.col - 1, tokens*[*pos*]*.row*)*

raise Exception*(*msg*)*

return ast