Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №5

По теме “Создание интерпретатора”

Выполнил:

студент гр. 753504

Осипик И.Ф.

Проверил:

Ст. преподаватель КИ Шиманский В. В.

Минск 2020

**Содержание**

[Введение 3](#_Toc8321103)

[1. Теоретические сведения 4](#_Toc8321104)

[2. Примеры интерпретации 9](#_Toc8321105)

[3. Обработка ошибок времени выполнения 12](#_Toc8321106)

[Приложение. Код программы 16](#_Toc8321107)

# **Введение**

В данной лабораторной работе необходимо выполнить заключительную стадию – используя написанные ранее лексический, синтаксический и семантический анализаторы создать работающий интерпретатор подможества языка Python 3.7.

На вход программы подается текстовый файл, содержащий строки текста программы, в консоль выводятся ошибки, если они есть, а если нет, то вывод происходит только если в интерпретируемом коде есть обращение к встроенной функции print. Опционально можно выводить промежуточные таблицы токенов, синтаксическое дерево.

# **1. Теоретические cведения**

Интерпретатор — это программа (разновидность транслятора), которая выполняет интерпретацию.

Интерпретация — это построчный анализ, обработка и выполнение исходного кода программы или запроса (в отличие от компиляции, где весь текст программы, перед запуском, анализируется и транслируется в машинный или байт-код, без её выполнения)

Транслятор — это программа, которая переводит входную программу на исход­ном (входном) языке в эквивалентную ей выходную программу на результирую­щем (выходном) языке.

В этом определении слово «программа» встречается три раза, и это не ошибка и не тавтология. В работе транслятора, действительно, уча­ствуют всегда три программы.

Во-первых, сам транслятор является программой(теоретически возможна реализация транслятора с помощью аппаратных средств, однако широкое практическое применение их не из­вестно.) — обычно он входит в состав системного программного обеспечения вычислительной системы. То есть транс­лятор — это часть программного обеспечения (ПО), он представляет собой на­бор машинных команд и данных и выполняется компьютером, как и все прочие программы в рамках операционной системы (ОС) Все составные части трансля­тора представляют собой фрагменты или модули программы со своими входны­ми и выходными данными.

Во-вторых, исходными данными для работы транслятора служит текст входной программы — некоторая последовательность предложений входного языка про­граммирования. Обычно это символьный файл, но этот файл должен содержать текст программы, удовлетворяющий синтаксическим и семантическим требова­ниям входного языка. Кроме того, этот файл несет в себе некоторый смысл, оп­ределяемый семантикой входного языка.

В-третьих, выходными данными транслятора является текст результирующей программы. Результирующая программа строится по синтаксическим правилам, заданным в выходном языке транслятора, а ее смысл определяется семантикой выходного языка. Важным требованием в определении транслятора является эк­вивалентность входной и выходной программ. Эквивалентность двух программ означает совпадение их смысла с точки зрения семантики входного языка (для исходной программы) и семантики выходного языка (для результирующей про­граммы). Без выполнения этого требования сам транслятор теряет всякий прак­тический смысл.

Итак, чтобы создать транслятор, необходимо прежде всего выбрать входной и выходной языки. С точки зрения преобразования предложений входного язы­ка в эквивалентные им предложения выходного языка транслятор выступает как переводчик. Например, трансляция программы с языка С в язык ассемблера, по сути, ничем не отличается от перевода, скажем, с русского языка на английский, с той только разницей, что сложность языков несколько иная (почему не суще­ствует трансляторов с естественных языков — см раздел «Классификация язы­ков и грамматик»). Поэтому и само слово «транслятор» (английское: translator) означает «переводчик».

Результатом работы транслятора будет результирующая программа, но только в том случае, если текст исходной программы является правильным — не со­держит ошибок с точки зрения синтаксиса и семантики входного языка. Если исходная программа неправильная (содержит хотя бы одну ошибку), то резуль­татом работы транслятора будет сообщение об ошибке (как правило, с допол­нительными пояснениями и указанием места ошибки в исходной программе). В этом смысле транслятор сродни переводчику, например, с английского, кото­рому подсунули неверный текст.

Кроме понятия «транслятор» широко употребляется также близкое ему по смыс­лу понятие «компилятор». Компилятор — это транслятор, который осуществляет перевод исходной програм­мы в эквивалентную ей объектную программу на языке машинных команд или на языке ассемблера.

Таким образом, компилятор отличается от транслятора лишь тем, что его ре­зультирующая программа всегда должна быть написана на языке машинных ко­дов или на языке ассемблера. Результирующая программа транслятора, в общем случае, может быть написана на любом языке — возможен, например, транслятор программ с языка Pascal на язык С. Соответственно, всякий компилятор являет­ся транслятором, но не наоборот — не всякий транслятор будет компилятором. Например, упомянутый выше транслятор с языка Pascal на С компилятором яв­ляться не будет.

Само слово «компилятор» происходит от английского термина «compiler» («со­ставитель», «компоновщик»). Видимо, термин обязан своему происхождению способности компиляторов составлять объектные программы на основе исход­ных программ.

Результирующая программа компилятора называется «объектной программой» или «объектным кодом». Файл, в который она записана, обычно называется «объ­ектным файлом». Даже в том случае, когда результирующая программа порож­дается на языке машинных команд, между объектной программой (объектным файлом) и исполняемой программой (исполняемым файлом) есть существенная разница.

Порожденная компилятором программа не может непосредственно выполняться на компьютере, так как она не привязана к конкретной области па­мяти, где должны располагаться ее код и данные. Здесь есть еще один «подводный камень», связанный с терминологической путаницей. Термины «объектный код» и «объектный файл» возникли достаточно давно. Первона­чально с ними не было связано никаких проблем. Однако сейчас появилось понятие «объектно-ориентированное программирование» (а также анализ и проектирование), где под термином «объект» подразумевается совершенно иное, нежели в «объектном коде».

Следует помнить, что «объектный код» никакого отношения к «объекту» с точки зрения объектно-ориентированного программирования не имеет! И хотя в результате компиля­ции программ, написанных на объектно-ориентированных языках, тоже получаются объектный код и объектные файлы, это никак не связывает между собой совершенно различные по смыслу термины.

Компиляторы, безусловно, самый распространенный вид трансляторов (многие считают их вообще единственным видом трансляторов, хотя это не так). Они име­ют самое широкое практическое применение, которым обязаны широкому рас­пространению всевозможных языков программирования. Далее всегда будем говорить о компиляторах, подразумевая, что выходная программа написана на языке машинных кодов или языке ассемблера (если это не так, то это будет спе­циально указываться отдельно). Следует особо упомянуть, что сейчас в современных системах программирования стали появляться компиляторы, в которых результирующая программа создается не на языке машинных команд и не на языке ассемблера, а на промежуточном языке.

Сам по себе этот промежуточный язык не может непосредственно исполняться на компьюте­ре, а требует специального промежуточного интерпретатора для выполнения написан­ных на нем программ. Хотя в данном случае термин «транслятор» был бы, наверное, более правильным, в литературе употребляется понятие «компилятор», поскольку про­межуточный язык является языком очень низкого уровня, будучи родственным машин­ным командам и языкам ассемблера.

Интерпретаторы не очень сильно отличаются от компиляторов. Они также конвертируют высокоуровневые языки в читаемые машиной бинарные эквиваленты. Каждый раз когда интерпретатор получает на выполнение код языка высокого уровня, то прежде чем сконвертировать его в машинный код, он конвертирует этот код в промежуточный язык. Каждая часть кода интерпретируется и выполняется отдельно и последовательно, и если в какой-то части будет найдена ошибка, она остановит интерпретацию кода без трансляции следующей части кода.

Очертание процесса интерпретации на рисункее выше показывает, что сначала исходный код конвертируется в промежуточную форму, а затем выполняется интерпретатором.

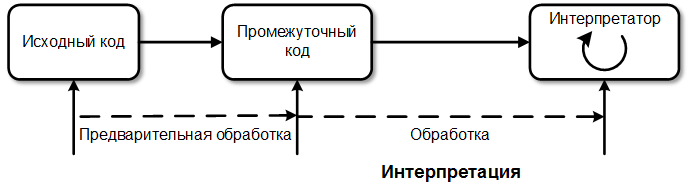
[](http://devnuances.com/wp-content/uploads/2013/11/Protsess-interpretatsii.png)

Рис 1.1. – Схема работы интепретатора

Главные отличия между компилятором и интерпретатором следующие:

* Интерпретатор берет одну инструкцию, транслирует и выполняет ее, а затем берет следующую инструкцию. Компилятор же транслирует всю программу сразу, а потом выполняет ее.
* Компилятор генерирует отчет об ошибках после трансляции всего, в то время как интерпретатор прекратит трансляцию после первой найденной ошибки.
* Компилятор по сравнению с интерпретатором требует больше времени для анализа и обработки языка высокого уровня.
* Помимо времени на обработку и анализ, общее время выполнения кода компилятора быстрее в сравнении с интерпретатором.

Естественно, трансляторы и компиляторы, как и все прочие программы, разраба­тывает человек (люди) — обычно это группа разработчиков.   
В принципе они могли бы создавать его непосредственно на языке машинных команд, однако объем кода и данных современных компиляторов таков, что их создание на язы­ке машинных команд практически невозможно в разумные сроки при разумных трудозатратах. Поэтому практически все современные компиляторы также соз­даются с помощью компиляторов (обычно в этой роли выступают предыдущие версии компиляторов той же фирмы-производителя). И в этом качестве ком­пилятор является уже выходной программой для другого компилятора, которая ничем не лучше и не хуже всех прочих порождаемых выходных программ.

Кроме схожих между собой понятий «транслятор» и «компилятор» существует принципиально отличное от них понятие интерпретатора. В отличие от трансляторов интерпретаторы не порождают результирующую про­грамму (и вообще какого-либо результирующего кода). Интерпретатор, так же как и транслятор, анализирует текст исходной программы. Однако он не порождает результирующей программы, а сразу же выполняет исходную в соответствии с ее смыслом, заданным семанти­кой входного языка. Таким образом, результатом работы интерпретатора будет результат, заданный смыслом исходной программы, в том случае, если эта про­грамма правильная, или сообщение об ошибке, если исходная программа неверна.

Конечно, чтобы исполнить исходную программу, интерпретатор так или иначе должен преобразовать ее в язык машинных кодов, поскольку иначе выполнение программ на компьютере невозможно. Он и делает это, однако полученные ма­шинные коды не являются доступными — их не видит пользователь интерпрета­тора. Эти машинные коды порождаются интерпретатором, исполняются и уничтожаются по мере надобности — так, как того требует конкретная реализация интерпретатора. Пользователь же видит результат выполнения этих кодов — то есть результат выполнения исходной программы (требование об эквивалентно­сти исходной программы и порожденных машинных кодов и в этом случае, без­условно, должно выполняться).

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 (сохраняется результат парсинга исходного кода), 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. К примеру, виртуальные машины вроде QEMU, Bochs, VMware включают в себя интерпретаторы машинного кода процессоров семейства x86.

Некоторые интерпретаторы (например, для языков Лисп, Scheme, Python, Бейсик и других) могут работать в режиме диалога или так называемого цикла чтения-вычисления-печати (англ. *read-eval-print loop, REPL*). В таком режиме интерпретатор считывает законченную конструкцию языка (например, s-expression в языке Лисп), выполняет её, печатает результаты, после чего переходит к ожиданию ввода пользователем следующей конструкции.

Уникальным является язык [Forth](https://ru.wikipedia.org/wiki/Forth), который способен работать как в режиме интерпретации, так и компиляции входных данных, позволяя переключаться между этими режимами в произвольный момент, как во время трансляции исходного кода, так и во время работы программ.

Следует также отметить, что режимы интерпретации можно найти не только в программном, но и [аппаратном обеспечении](https://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5). Так, многие [микропроцессоры](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) интерпретируют машинный код с помощью встроенных [микропрограмм](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), а процессоры семейства x86, начиная с [Pentium](https://ru.wikipedia.org/wiki/Pentium) (например, на архитектуре [Intel P6](https://ru.wikipedia.org/wiki/Intel_P6)), во время исполнения машинного кода предварительно транслируют его во внутренний формат (в последовательность микроопераций).

Учитывая, что в рамках данной лабораторной работы был написан компилятор собственного языка C# (по типу – интерпретатор компилирующего типа), будет описана непосредственно работа компилятора на понятном языке.

Поскольку в этой работе нас не интересуют лексер, парсер и семантический анализатор, не имеет значения, как создаются наборы команд. Поэтому на данном этапе принято, что все, что имеет значение, — это то, что нашему переводчику дано правильное расположение инструкций, и мы это обеспечили в предыдущих лабораторных работах.

По сути, наш интерпретатор будет являться виртуальной машиной с своим стеком для хранения команд и их обработки. Непосредственно на стеке и будут выполняться все команды, например есть выражение 2 + 3, и мы для каждого операнда поместим значение на стек, таким образом, у нас будет два значения в стеке, а третей командой оператор выполнит команду, требующую два операнда – и соответственно будет вынято из стека два зна-чения и проведена команда сложения, которая и даст нам результат 5.   
Если же забегать вперед, то можно отметить что в случае использования переменных мы будем загружать на стек переменную с помощью команды (например LOAD\_VALUE), затем в отдельное хранилище (назовем его словарем) будем загружать по имени переменной значение из стека (назовем команду STORE\_NAME) и также при непосредственно суммировании мы будем до-ставать значения из словаря в стек с помощью команды, допустим, LOAD\_NAME.

Вообще, в проекте каждая функция собирается в отдельный скомпилированный объект со своим байт кодом. Он будет представлен последовательностью различных чисел. Например, представим следующий пример: есть выражение x = 3. И есть следующий байт код для этого выражения – 100, 1, 0, 125, 0, 0. Каждое из чисел – 1 байт, т е максимальный размер 255. Данная последовательность представляет собой инструкции LOAD\_VALUE (данной инструкции соответствует число 100), 1, 0 – это операнды описанной инструкции (1 значит первый (второй по индексу) элемент константа или пере-менная в нашей программе, в нашем случае 3), а 0 – ничего не значит. Точно также 125 - STORE\_NAME, 0 – идентифицирует переменную x, 0 – ничего не значит. Почему же спрашивается нужны ничего не значащие нули? Дело в том, что если у нас будет в программе много переменных, то 255 значений не хватит чтобы описать их всех, поэтому второй 0 – это потенциально дополнительный байт идентификации. Итого у нас может быть теперь не 256, а 65536 вариантов значений, чего на практике достаточно.

Циклы и условные операторы реализованы как инструкции POP\_JUMP\_IF\_FALSE, которая смотрит что результат сравнения операции COMPARE\_OP на стек и перемещается на указанную инструкцию в нашей ленте байт кода если условие не удовлетворяется. Точно также устроены и циклы, так как, по сути, мы в конце цикла проверяем, вышли ли мы из него, и если нет – то прыгаем по на указанный оператор т. е. начало цикла.

# 

# **2.** **Примеры интерпретации**

2.1 Числа Фибоначчи (пример интерпретации простых операций):

fib1 = 1

fib2 = 1

n = 102

i = 0

while i < n - 2:

fib\_sum = fib1 + fib2

fib1 = fib2

fib2 = fib\_sum

i = i + 1

print("fibonacchi number ", fib2)

*Листинг 2.1. Программа вычисления n-го числа Фибоначчи*

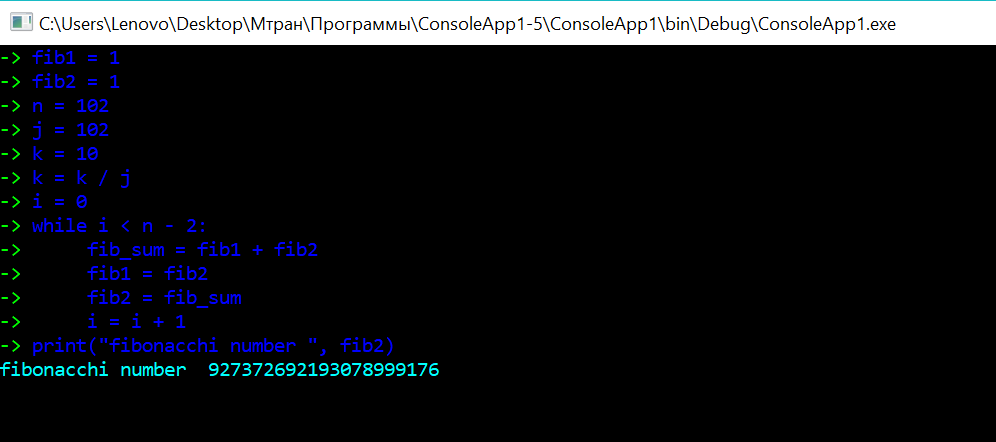


Рис 2.1. – Результат интерпретации

2.2 Перемножение векторов (интерпретация работы с массивами):

a = [ 1, 2, 3, 4, 5, 6]

b = [7, 8, 9, 10, 11, 12]

def multiplicate\_vectors(a, b) :

if len(a) != len(b) :

print( "Error, length is not equal" )

return

mult = list( )

for i in range( len(a) ) :

m = a[i] \* b[i]

mult.append(m)

return mult

mult = multiplicate\_vectors(a, b)

print( "-------------" )

print(mult)

*Листинг 2.2. - Программа умножения векторов*

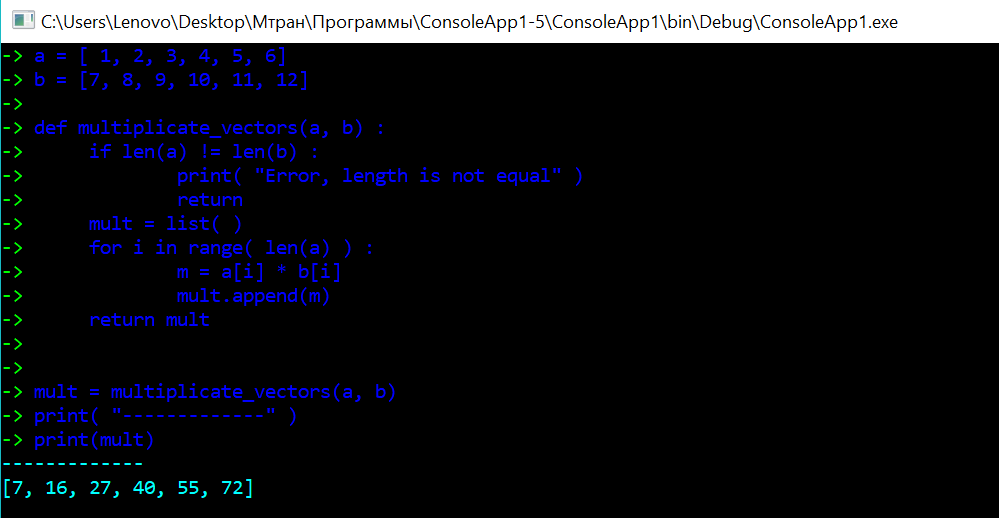


Рис. 2.2. – Результат интерпретации

2.3 Сортировка слиянием:

def mergeSort(alist) :

print("Splitting ",alist)

if len(alist) > 1:

mid = int(len(alist) / 2)

lefthalf = alist[ :mid ]

righthalf = alist[ mid: ]

mergeSort(lefthalf)

mergeSort(righthalf)

i=0

j=0

k=0

while i < len(lefthalf) and j < len(righthalf) :

if lefthalf[i] < righthalf[j] :

alist[k] = lefthalf[i]

i = i + 1

else :

alist[k] = righthalf[j]

j = j + 1

k = k + 1

while i < len(lefthalf) :

alist[k] = lefthalf[i]

i = i + 1

k = k + 1

while j < len(righthalf) :

alist[k] = righthalf[j]

j = j + 1

k = k + 1

print("Merging ",alist)

alist = [54, 26, 93, 17]

mergeSort(alist)

print(alist)

*Листинг 2.3. – Сортировка слиянием*

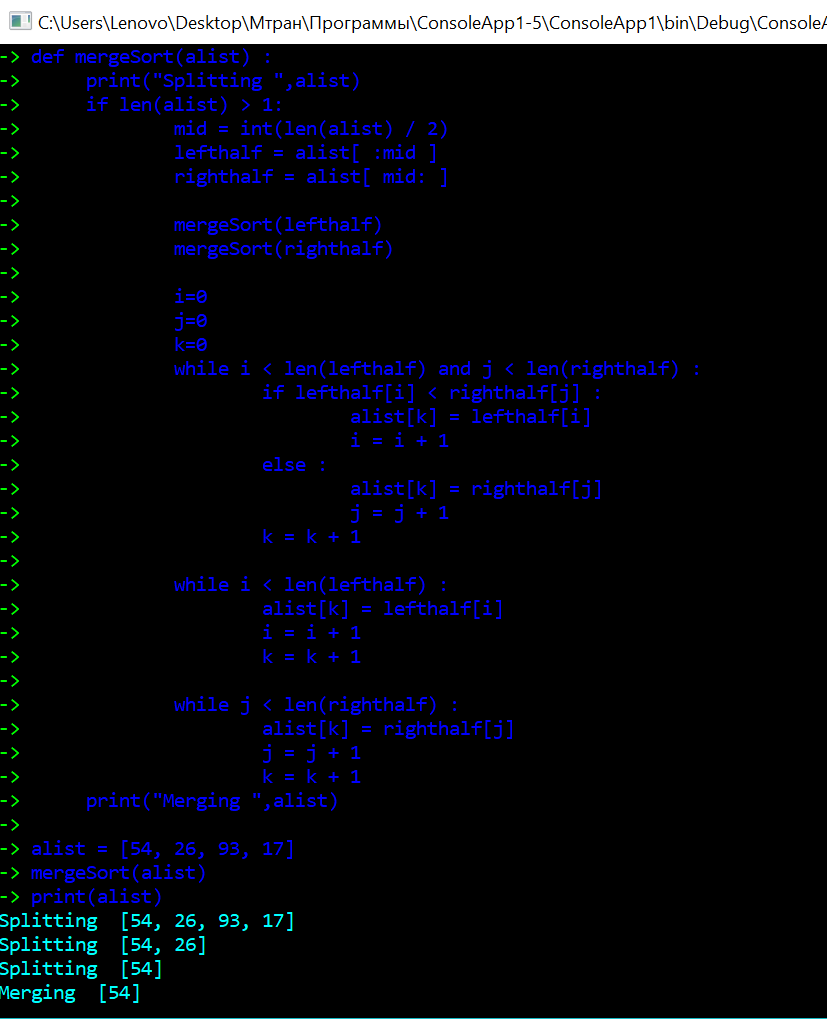


Рис 2.3. – Интерпретация сортировки

# 

# **3. Обработка ошибок времени выполнения**

3.1 Деление на переменную, равную 0

fib1 = 1

fib2 = 1

n = 102

j = 102 - n

k = 10

k = k / j # деление на переменную, равную 0

i = 0

while i < n - 2:

fib\_sum = fib1 + fib2

fib1 = fib2

fib2 = fib\_sum

i = i + 1

print("fibonacchi number ", fib2)

*Листинг 3.1. – Код с ошибкой*

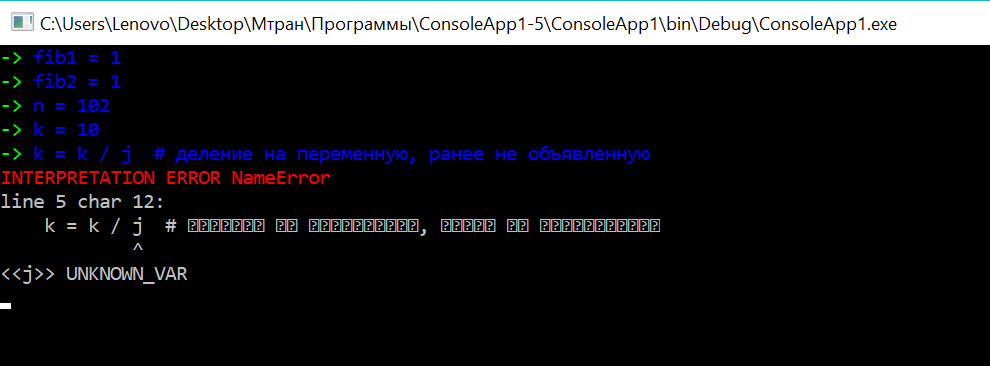


Рис 3.1. — Вывод ошибки

3.2 Использование неизвестной переменной

fib1 = 1

fib2 = 1

n = 102

k = 10

k = k / j # деление на переменную, ранее не объявленную

i = 0

while i < n - 2:

fib\_sum = fib1 + fib2

fib1 = fib2

fib2 = fib\_sum

i = i + 1

print("fibonacchi number ", fib2)

*Листинг 3.2. – код с ошибкой*

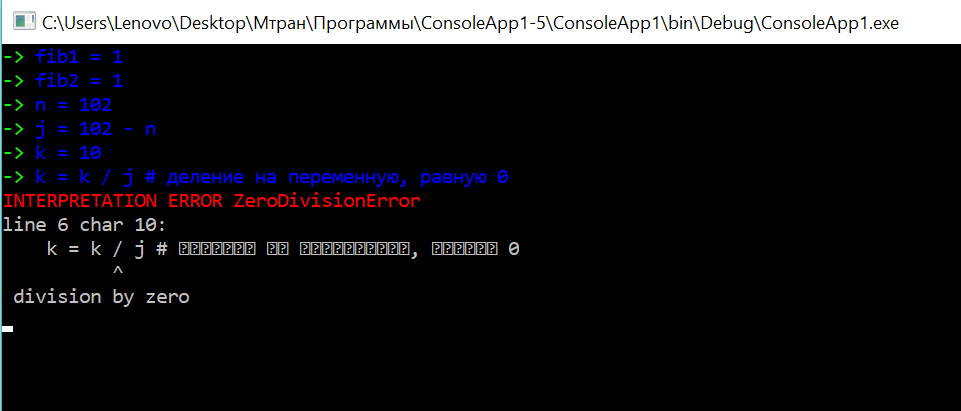


Рис 3.2 — Вывод ошибки

3.3 Выход за границы массива, размер которого известен только во время выполнения

arr = list( )

for i in range(5) :

arr.append(i)

for i in range(6) :

print( arr[i] )

*Листинг 3.3. – Код создает массив из 5 чисел, а потом пытается извлечь из него последовательно 6 элементов*

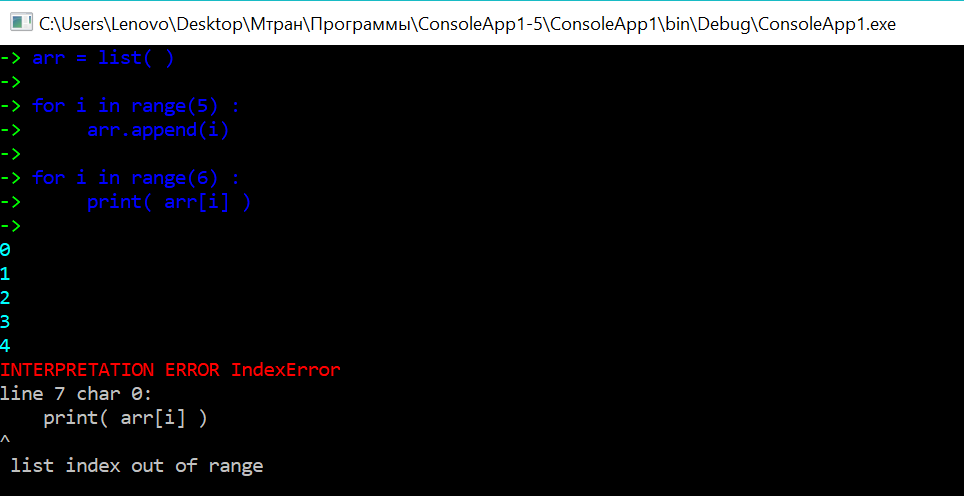


Рис. 3.3. – Вывод ошибки

# **3. Вывод**

В ходе лабораторной работы были изучены теоретические сведения по построению интерпретаторов. Изучены базовые понятия по теме интерпретации программ, повторены основные теоретические сведения об компиляторах и трансляторах, была построена схема запуска сходного кода при помощи интерпретатора.

Выявлены основные различия работы компилятора и интерпретатора:

* Компилятор транслирует всю программу сразу, а потом выполняет ее. Интерпретатор же берет инструкцию, транслирует и выполняет ее.
* Помимо времени на обработку и анализ, общее время выполнения кода компилятора быстрее в сравнении с интерпретатором.
* Компилятор генерирует отчет об ошибках после трансляции всего, в то время как интерпретатор прекратит трансляцию после первой найденной ошибки.
* Компилятор по сравнению с интерпретатором требует больше времени для анализа и обработки языка высокого уровня.

Изучены особенности интерпретации как программного, так и аппаратного обеспечения компьютера. Например, язык [Forth](https://ru.wikipedia.org/wiki/Forth), который способен работать как в режиме интерпретации. Или, например, [микропроцессоры](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80), которые интерпретируют машинный код с помощью встроенных [микропрограмм](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) (процессоры семейства x86, начиная с [Pentium](https://ru.wikipedia.org/wiki/Pentium)).

Был разработан интерпретатор языка, аналогичного языку Python, и была продемонстрирована работа данного итерпретатора.

# **Приложение. Код программы**

namespace ConsoleApp1

{

class SyntaxAnalizer

{

protected int OpenedBracketsLevel = 0;

protected int CurrentBlockLevel = 0;

public ExpressionNode Analyse(IEnumerable<Token> tokens, out bool startNewBlock, out bool isElifElseNode)

{

OpenedBracketsLevel = 0;

startNewBlock = false;

isElifElseNode = false;

var firstToken = tokens.FirstOrDefault();

if (firstToken?.IsBlockOpeningOperation == true)

{

startNewBlock = true;

isElifElseNode = firstToken.TokenType == TokenTypes.ELSE || firstToken.TokenType == TokenTypes.ELIF;

if (tokens.LastOrDefault()?.TokenType != Token.TokenTypes.COLON)

{

var t = tokens.LastOrDefault();

throw new SyntaxErrorException("colon expected", t.Value, t.CodeLineIndex, t.CodeLineNumber);

}

}

ExpressionNode root = BuildTree(tokens);

if (OpenedBracketsLevel != 0)

{

throw new SyntaxErrorException("brackets do not match", tokens.Last().Value, tokens.Last().CodeLineIndex, tokens.Last().CodeLineNumber);

}

return root;

}

protected ExpressionNode BuildTree(IEnumerable<Token> tokens, ExpressionNode parent = null)

{

ExpressionNode root = null;

ExpressionNode left = null;

Token token = tokens.FirstOrDefault();

if (token is null)

return null;

if (token.IsConstant || token.TokenType == Token.TokenTypes.ID || token.TokenType == Token.TokenTypes.BUILT\_IN\_FUNCTION)

{

left = new ExpressionNode()

{

Operator = token,

Type = ExpressionNode.TokensToExpressionTypes.GetOrDefault(token.TokenType, ExpressionNode.ExpressionTypes.UNKNOWN)

};

var tt = tokens.ElementAtOrDefault(1)?.TokenType;

if (tt == Token.TokenTypes.OPENING\_ROUND\_BRACKET)

{

root = left;

left = null;

root.Type = ExpressionNode.ExpressionTypes.FUNCTION\_CALL;

root.Right = BuildTree(tokens.Skip(1));

}

else if (tt == Token.TokenTypes.COLON)

{

root = left;

left = null;

root.Right = BuildTree(tokens.Skip(2));

// TODO: maybe throw error here if there is something after COLON

}

else

{

root = BuildTree(tokens.Skip(1));

left.Parent = root;

}

}

else if (token.IsOpeningBracket)

{

this.OpenedBracketsLevel++;

root = BuildTree(tokens.Skip(1));

root.OperatorPriority++;

}

else if (token.IsClosingBracket)

{

this.OpenedBracketsLevel--;

root = BuildTree(tokens.Skip(1));

if (root != null)

root.OperatorPriority--;

}

else if (token.IsOperation)

{

root = new ExpressionNode()

{

Operator = token,

Type = ExpressionNode.TokensToExpressionTypes.GetOrDefault(token.TokenType, ExpressionNode.ExpressionTypes.UNKNOWN)

};

if (token.TokenType == Token.TokenTypes.MULTIPLICATION || token.TokenType == Token.TokenTypes.DIVISION)

{

root.OperatorPriority++;

}

root.Right = BuildTree(tokens.Skip(1), root);

}

if (root is null)

{

if (left is null)

return null;

left.Parent = parent;

return left;

}

root.Parent = parent;

if (left != null)

root.InsertDeepLeft(left);

if (root.Right != null && root.Operator.IsOperation && root.Right.Operator.IsOperation && root.OperatorPriority > root.Right.OperatorPriority)

return root.LeftRotation();

return root;

}

public static ExpressionNode ValidateNode(ExpressionNode node)

{

switch (node.Type)

{

case ExpressionNode.ExpressionTypes.BINARY\_OPERATION:

if (node.Left == null || node.Right == null)

{

throw new SyntaxErrorException(

"binary operation lacks operand",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

}

break;

case ExpressionNode.ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION:

if (node.Left != null || node.Right == null)

throw new SyntaxErrorException(

"conditional operator wrong usage",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

break;

case ExpressionNode.ExpressionTypes.UNKNOWN:

throw new SyntaxErrorException(

"unknown expression",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

case ExpressionNode.ExpressionTypes.OPERAND:

if (node.Left != null)

throw new SyntaxErrorException(

"unknown operator",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

break;

default:

break;

}

return node;

}

public class SyntaxErrorException : FormatException

{

public string Value { get; set; }

public int PositionInLine { get; set; }

public int LineNumber { get; set; }

public SyntaxErrorException(string message, string value, int positionInLine, int lineNumber) : base(message)

{

Value = value;

PositionInLine = positionInLine;

LineNumber = lineNumber;

}

}

public class ExpressionNode

{

public ExpressionNode Left = null;

public Token Operator = null;

public ExpressionTypes Type;

public int OperatorPriority = 0;

public ExpressionNode Right = null;

public ExpressionNode Parent = null;

public TreeList<ExpressionNode> Block = new TreeList<ExpressionNode>(null);

public ExpressionNode LeftRotation()

{

ExpressionNode newRoot = new ExpressionNode()

{

Right = this.Right.Right,

Operator = this.Right.Operator,

Type = this.Right.Type,

Parent = this.Parent

};

newRoot.Left = new ExpressionNode()

{

Left = this.Left,

Right = this.Right.Left,

Operator = this.Operator,

Type = this.Type,

Parent = newRoot

};

return newRoot;

}

public void InsertDeepLeft(ExpressionNode node)

{

ExpressionNode temp = this;

while (!(temp.Left is null))

{

temp = temp.Left;

}

temp.Left = node;

}

public override string ToString()

{

return $"({Operator.ToString()})";

}

public enum ExpressionTypes

{

UNKNOWN,

UNARY\_OPERATION,

BINARY\_OPERATION,

BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

BLOCK\_OPENING\_OPERATION,

FUNCTION\_CALL,

FUNCTION\_DEF,

OPERAND

};

public static Dictionary<TokenTypes, ExpressionTypes> TokensToExpressionTypes = new Dictionary<TokenTypes, ExpressionTypes>()

{

[TokenTypes.ASSIGN] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.COMMA] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.DOT] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.IF] = ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

[TokenTypes.ELIF] = ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

[TokenTypes.ELSE] = ExpressionTypes.BLOCK\_OPENING\_OPERATION,

[TokenTypes.FOR] = ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

[TokenTypes.WHILE] = ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

[TokenTypes.PLUS] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.MINUS] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.MODULE] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.DIVISION] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.MULTIPLICATION] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.NOT] = ExpressionTypes.UNARY\_OPERATION,

[TokenTypes.AND] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.OR] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.IN] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.LOWER] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.LOWER\_OR\_EQUAL] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.GREATER] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.GREATER\_OR\_EQUAL] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.NOT\_EQUAL] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.EQUAL] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.FUNCTION\_DEFINITION] = ExpressionTypes.FUNCTION\_DEF,

[TokenTypes.STRING\_CONST] = ExpressionTypes.OPERAND,

[TokenTypes.INT\_NUM] = ExpressionTypes.OPERAND,

[TokenTypes.FLOAT\_NUM] = ExpressionTypes.OPERAND,

[TokenTypes.ID] = ExpressionTypes.OPERAND,

[TokenTypes.BUILT\_IN\_FUNCTION] = ExpressionTypes.OPERAND,

[TokenTypes.COLON] = ExpressionTypes.BLOCK\_OPENING\_OPERATION

};

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1

{

class SemanticItem

{

public string Name { get; set; }

public VarTypes VarType { get; set; }

public FunctionSpecification FSpecification { get; set; }

public enum VarTypes

{

NUMBER\_VAR,

INTEGER\_VAR,

FLOAT\_VAR,

STRING\_VAR,

FUNCTION\_VAR,

LIST\_VAR,

BOOL\_VAR,

NONE\_VAR,

}

public static Dictionary<string, VarTypes> StringVarTypes = new Dictionary<string, VarTypes>

{

["int"] = VarTypes.INTEGER\_VAR,

["float"] = VarTypes.FLOAT\_VAR,

["str"] = VarTypes.STRING\_VAR,

["bool"] = VarTypes.BOOL\_VAR,

["list"] = VarTypes.LIST\_VAR,

["None"] = VarTypes.NONE\_VAR,

["function"] = VarTypes.FUNCTION\_VAR

};

public struct FunctionSpecification

{

public VarTypes ReturnType { get; set; }

public int MaxArgumentsAmount { get; set; }

public int MinArgumentsAmount { get; set; }

}

public static Dictionary<string, FunctionSpecification> BuiltInFunctionsReference = new Dictionary<string, FunctionSpecification>

{

["print"] = new FunctionSpecification() { ReturnType = VarTypes.NONE\_VAR, MinArgumentsAmount = 0, MaxArgumentsAmount = 100 },

["input"] = new FunctionSpecification() { ReturnType = VarTypes.STRING\_VAR, MinArgumentsAmount = 0, MaxArgumentsAmount = 1 },

["range"] = new FunctionSpecification() { ReturnType = VarTypes.INTEGER\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 3 },

["type"] = new FunctionSpecification() { ReturnType = VarTypes.STRING\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 3 },

["abs"] = new FunctionSpecification() { ReturnType = VarTypes.NUMBER\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 1 },

["max"] = new FunctionSpecification() { ReturnType = VarTypes.NUMBER\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 100 },

["min"] = new FunctionSpecification() { ReturnType = VarTypes.NUMBER\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 100 },

["int"] = new FunctionSpecification() { ReturnType = VarTypes.INTEGER\_VAR, MinArgumentsAmount = 0, MaxArgumentsAmount = 1 },

["float"] = new FunctionSpecification() { ReturnType = VarTypes.FLOAT\_VAR, MinArgumentsAmount = 0, MaxArgumentsAmount = 1 }

};

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

using static ConsoleApp1.Token;

using static ConsoleApp1.LexicalAnalizer;

using ConsoleTables;

namespace ConsoleApp1

{

class Program

{

static void PrintTokensDictionary(Dictionary<string, Token> dictionary)

{

ConsoleTable consoleTable = new ConsoleTable("TOKEN", "DESCRIPTION");

foreach (Token token in dictionary.Values)

{

consoleTable.AddRow(token.Value, token.DescriptionString);

}

consoleTable.Write();

}

static string PrintNodeWithChildren(SyntaxAnalizer.ExpressionNode node, string indentation)

{

if (node == null)

{

return "";

}

SyntaxAnalizer.ValidateNode(node);

StringBuilder stringBuilder = new StringBuilder();

stringBuilder.AppendLine($"{indentation} {node.Operator.Value}");

if (node.Left != null)

stringBuilder.Append(PrintNodeWithChildren(node.Left, indentation + "\\"));

if (node.Right != null)

stringBuilder.Append(PrintNodeWithChildren(node.Right, indentation + "\\"));

return stringBuilder.ToString();

}

static void PrintSyntaxTree(IEnumerable<SyntaxAnalizer.ExpressionNode> nodes, int nestingLevel = 1)

{

string indentation = new String('|', nestingLevel);

foreach (var node in nodes)

{

Console.Write(PrintNodeWithChildren(node, indentation));

Console.WriteLine(indentation);

PrintSyntaxTree(node.Block, nestingLevel+1);

}

}

static string errorDescription(int indexInCodeLine, string codeLine)

{

StringBuilder stringBuilder = new StringBuilder(codeLine);

stringBuilder.AppendLine();

stringBuilder.Append(new string(' ', indexInCodeLine));

stringBuilder.Append('^');

return stringBuilder.ToString();

}

static void DoTheJob(IEnumerable<string> codeLines)

{

Dictionary<string, Token> constants = new Dictionary<string, Token>();

Dictionary<string, Token> variables = new Dictionary<string, Token>();

Dictionary<string, Token> operators = new Dictionary<string, Token>();

Dictionary<string, Token> keywords = new Dictionary<string, Token>();

List<LexicalError> errors = new List<LexicalError>();

// running lexical analysis

TreeList<SyntaxAnalizer.ExpressionNode> tree = new TreeList<SyntaxAnalizer.ExpressionNode>(null);

TreeList<SyntaxAnalizer.ExpressionNode> currentBlock = tree;

int lineNumber = 0;

SyntaxAnalizer sa = new SyntaxAnalizer();

LexicalAnalizer la = new LexicalAnalizer();

int previousLineIndentation = 0;

foreach (string line in codeLines)

{

Construction construction = la.AnaliseLine(line, lineNumber);

if (construction.Tokens.Count == 0)

{

lineNumber++;

continue;

}

for (int i = 0; i < construction.Tokens.Count; i++)

{

Token token = construction.Tokens[i];

if (token.IsReservedIdToken)

keywords.TryAdd(token.Value, token);

else if (token.IsOperation)

operators.TryAdd(token.Value, token);

else if (token.IsConstant)

constants.TryAdd(token.Value, token);

else if (token.TokenType != TokenTypes.UNKNOWN)

{

variables.TryAdd(token.Value, token);

}

}

if (construction.HasErrors)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("\t\t ERRORS");

Console.ResetColor();

foreach (LexicalError error in construction.Errors)

{

Console.WriteLine($"line {error.CodeLineNumber + 1} char {error.IndexInCodeLine + 1} :: {error.ErrorType}");

Console.WriteLine(error.Description);

}

Console.Read();

Environment.Exit(1);

}

SyntaxAnalizer.ExpressionNode node = null;

bool isElifElseNode = false;

bool newBlockToOpen = false;

node = sa.Analyse(construction.Tokens, out newBlockToOpen, out isElifElseNode);

int indentationDiff = previousLineIndentation - construction.Indentation;

if (indentationDiff > 0)

{

for (int i = previousLineIndentation-1; i >= construction.Indentation; i--)

{

currentBlock = currentBlock.Parent;

if (currentBlock.Indentation == i)

break;

}

// currentBlock = currentBlock.Parent; // TODO: create parent relationship between BLOCKS to support >1 level nesting

if (node.Operator.IsElif && !currentBlock.Last().Operator.IsIf)

{

throw new SyntaxAnalizer.SyntaxErrorException(

"elif block not allowed here",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

}

else if (node.Operator.IsElse && !(currentBlock.Last().Operator.IsIf || currentBlock.Last().Operator.IsElif))

{

throw new SyntaxAnalizer.SyntaxErrorException(

"else block not allowed here",

line,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

}

}

previousLineIndentation = construction.Indentation;

lineNumber++;

if (newBlockToOpen)

{

if ((node.Operator.IsElif || node.Operator.IsElse) && !currentBlock.Last().Operator.IsIf && !currentBlock.Last().Operator.IsElif)

{

throw new SyntaxAnalizer.SyntaxErrorException(

"lacks IF clause for elif|else block to appear",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

}

currentBlock.Add(node);

currentBlock.Last().Block = new TreeList<SyntaxAnalizer.ExpressionNode>(currentBlock);

currentBlock = currentBlock.Last().Block;

currentBlock.Indentation = construction.Indentation;

continue;

}

currentBlock.Add(node);

}

if (errors.Any())

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("\t\t ERRORS");

Console.ResetColor();

foreach (LexicalError error in errors)

{

Console.WriteLine($"line {error.CodeLineNumber + 1} char {error.IndexInCodeLine + 1} :: {error.ErrorType}");

Console.WriteLine(error.Description);

}

}

Console.WriteLine("SYNTAX TREE:\n");

PrintSyntaxTree(tree);

// console tables output block

Console.WriteLine("\n \t\t CONSTANTS");

PrintTokensDictionary(constants);

Console.WriteLine("\n \t\t VARIABLES");

PrintTokensDictionary(variables);

Console.WriteLine("\n \t\t KEYWORS");

PrintTokensDictionary(keywords);

Console.WriteLine("\n \t\t OPERATORS");

PrintTokensDictionary(operators);

}

static void Main(string[] args)

{

Console.OutputEncoding = System.Text.Encoding.UTF8;

string FILENAME = @"D:/Documents/Univer/6\_sem/MTRANS/lab2/ConsoleApp1/test.py";

IEnumerable<string> codeLines = System.IO.File.ReadLines(FILENAME);

try

{

DoTheJob(codeLines);

}

catch (SyntaxAnalizer.SyntaxErrorException e)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine($"SYNTAX ERROR {e.Message}");

Console.ResetColor();

Console.WriteLine($"line {e.LineNumber} char {e.PositionInLine}:");

Console.WriteLine(errorDescription(e.PositionInLine, codeLines.ElementAt(e.LineNumber).Trim()));

}

catch (InvalidOperationException e)

{

Console.WriteLine($"SYNTAX ERROR {e.Message}");

Console.WriteLine("block opening element has nothing in its block!");

}

Console.Read();

}

}

}