МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ

Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: «РОЗРОБКА ІНТЕРПРЕТАТОРА СКРИПТОВОЇ МОВИ»

Виконав: студент 2 курсу, групи 8.1219

спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)

освітньої програми інженерія програмного забезпечення
(назва освітньої програми)

О.О. Пархоменко
(ініціали та прізвище)

Керівник декан математичного факультету, професор, д.т.н. Гоменюк С.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент
Лісняк А.О.
(підпис)

« » 2020 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Пархоменко Олександру Олександровичу
(прізвище, ім'я та по-батькові)

1. Тема роботи (проекту) Розробка інтерпретатора скриптової мови

керівник роботи (проекту) Гоменюк Сергій Іванович, д.т.н., професор
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » травня 2020 року № 576-с

2. Строк подання студентом роботи 30.11.2020

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1. Постановка задачі
   2. Основні теоретичні відомості
   3. Розробка програмного додатку

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Гоменюк С. І., д. т. н., професор		
2	Гоменюк С. І., д. т. н., професор		
3	Гоменюк С. І., д. т. н., професор		

7. Дата видачі завдання 12.08.2020

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	17.08.2020	Виконано
2.	Збір вихідних даних.	07.09.2020	Виконано
3.	Обробка методичних та теоретичних	01.10.2020	Виконано
	джерел.		
4.	Розробка першого розділу.	19.10.2020	Виконано
5.	Розробка другого розділу.	09.11.2020	Виконано
6.	Оформлення та нормоконтроль	30.11.2020	Виконано
	кваліфікаційної роботи.		
7.	Захист кваліфікаційної роботи.	16.12.2020	Виконано

Студент ____________ О.О. Пархоменко
(підпис) (ініціали та прізвище)

Керівник роботи ____________ С.І. Гоменюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер ____________ О.В. Кудін
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка інтерпретатора скриптової мови»: 59 с., 32 рис., 5 табл., 7 джерел, 3 додатки.

СКРИПТОВА МОВА, ІНТЕРПРЕТАТОР, ТРАНСЛЯТОР, AST, РОЗШИРЕНА ФОРМА БЕКУСА-НАУРА, ЛЕКСЕР, ПАРСЕР, КОМПІЛЯТОР, АБСТРАКТНЕ СИНТАКСИЧНЕ ДЕРЕВО, ТОКЕН, ЛЕКСЕМА, СИНТАКСИЧНИЙ АНАЛІЗ, ЛЕКСИЧНИЙ АНАЛІЗ.

Об’єкт дослідження – процес розробки трансляторів, розуміння інструментів та методів, які лежать у основі побудови інтерпретаторів.

Предмет дослідження – розробка інтерпретатора, побудованого для розробленої скриптової мови.

Мета роботи – розробити інтерпретатор скриптової мови.

Метод дослідження – аналітичний, порівняльний.

У кваліфікаційній роботі розглядається процес створення програмного забезпечення, а саме інтерпретатора для розробленної скриптової мови. Розглянуто основні види скриптових мов, відмінність інтерпретатора від компілятора та існуючі види інтерпретаторів на даний момент. Проаналізовано вільне поширюване програмне забезпечення , за допомогою якого розроблено інтерпретатор та скриптову мову. Результати можуть бути використані у практичній діяльності, під час написання базових скриптів.

ABSTRACT

Master's Qualification Thesis «Development of a Scripting Language Interpreter»: 59 pages, 32 figures, 5 tables, 7 references, 3 supplements.

SCRIPTING LANGUAGE INTERPRETERS, TRANSLATORS, AST, EXTENDED BACKUS-NAUR FORM, LEXERS, PARSERS, COMPILERS, ABSTRACT PARSE TREE, TOKENS, TOKENS, SYNTACTIC ANALYSIS, LEXICAL ANALYSIS.

The object of research is the process of developing translators, understanding the tools and methods that underlie the construction of interpreters.

The subject of research is the development of its own interpreter, built for its own scripting language.

The purpose of the work is to develop a scripting language interpreter.

The methods of research are analytical, comparative.

The qualification work considers the process of creating software, namely an interpreter for the developed scripting language. The main types of scripting languages, the difference between the interpreter and the compiler and the existing types of interpreters at the moment are considered. Free distributed software was analyzed, with the help of which an interpreter and a scripting language were developed. The results can be used in practice, when writing basic scripts.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Abstract 5
Вступ 8
1 Розгляд існуючих інтерпретаторів та скриптових мов програмування 10
1.1 Опис завдання 10
1.2 Опис предметної області і напрямів дослідження 10
1.3 Аналіз і характеристика об’єкта проектування 11
1.4 Огляд існуючих скриптових мов програмування та їх інтерпретаторів 11
1.4.1 Загальні характеристики скриптових мов програмування 11
1.4.2 Класифікація скриптових мов програмування 12
1.4.3 Командно-сценарні мови 13
1.4.4 Прикладні-сценарні мови 14
1.4.5 Мови загального призначення 15
1.5 Інтерпретатор і його відмінність від компілятора 16
1.6 Типи інтерпретаторів 18
1.7 Основні поняття абстрактного синтаксичного дерева 21
2 Проектування програмного додатку 24
2.1 Технічне завдання 24
2.1.1 Найменування та область застосування 24
2.1.2 Призначення розробки 24
2.2 Вимоги до програмного додатку 24
2.2.1 Вимоги до надійності 25
2.2.2 Вимоги до складу і параметрів технічних засобів 25
2.3 Проектування системи 25
2.3.1 Переваги та недоліки Python 25
2.3.2 Розширена форма Бекуса-Наура 27
3 Ппрограмна реалізація 29
3.1 Структура проекту 29
3.2 Опис основних модулів проекту 30

3.2.1 Огляд функціональності модулів pscript_lexer та pscript_parser 31
3.2.2 Приклад роботи модулів pscript_lexer та pscript_parser 38
3.3 Приклад роботи додатку 43
Висновки 47
Перелік посилань 48
Додаток А 49
Додаток Б 51
Додаток В 55

ВСТУП

Більша частина роботи кожного програміста пов'язана з написанням програмного коду, тестуванням та налагодженням програм написаних на одній з безлічі мов програмування. Кожна мова програмування підтриму свої стилі програмування. Єдина мова, яку безпосередньо виконує процесор – це машинна мова. Колись всі програмісти розробляли програми в машинному коді, але зараз ця робота вже не робиться. Замість неї, програмісти пишуть вихідний код на мовах програмування високого рівня, який кожен комп'ютер (використовуючи одну з програм трансляторів до яких відносяться компілятор, інтерпретатор, асемблер) транслює в один або декілька етапів, перетворюючи кожний рядок програми, в машинний код, який в свою чергу виконується за допомогою процесора. Написання початкового програмного коду за допомогою будь-якої з мов програмування полегшує його розуміння і редагування програмістом. Існують програми, які інтерпретуються спеціальними програмами-інтерпретаторами або операційною системою. Для таких програм, як правило, не застосовується процес компіляції, тому вони називаються скриптами або «сценаріями».

Актуальність дослідження: по перше написати інтерпретатор – означає розвинути відразу кілька різних технічних навичок. Причому навичок, які виявляться корисними в програмуванні взагалі, а не тільки при написанні трансляторів. А по друге отриманні навички дозволять створити мову програмування, яка заповнює видимі недоліки існуючих.

Метою кваліфікаційної роботи є розробка інтерпретатора для скриптової мови. Для виконання поставленої мети було вирішено розробити скриптову мову.

Виходячи із мети, можна сформувати наступні **завдання**:

– дослідити програми транслятори, а саме – інтерпретатор та розібратися з чого він складається;

- спроектувати програмний застосунок;
- розробити скриптову мову та інтерпретатор;
- виконати тестування розробленого інтерпретатору;
- оформити програмну документацію.

Об’єктом дослідження є процес створення програм трансляторів, розуміння інструментів та методів, які лежать у основі побудови інтерпретаторів.

Предметом дослідження є розробка інтерпретатора, побудованого для розробленої скриптової мови.

1 РОЗГЛЯД ІСНУЮЧИХ ІНТЕРПРЕТАТОРІВ ТА СКРИПТОВИХ МОВ ПРОГРАМУВАННЯ

1.1 Опис завдання

Метою роботи є розробка інтерпретатора для скриптової мови. Було вирішено розробляти інтерпретатор на високорівневій мові програмування загального призначення Python, що дозволить виконувати скрипти написані на створеній мові за допомогою розробленного інтерпретатора.

1.2 Опис предметної області і напрямів дослідження

Фактично, інтерпретатор, що розробляється в даній роботі, це один із різновидів програм трансляторів.

Ідея цього розділу програмних додатків є переклад вихідного коду програми (програму, написану на одній з високорівневих мов програмування) в об'єктний код, який використовується процесором комп'ютера, або в проміжний код для подальшої інтерпретації. Крім здійснення перекладу, транслятори можуть виявляти в вихідному коді помилки, оптимізувати вихідний код, додавати в вихідний код налагоджувальні процедури, формувати словники ідентифікаторів і інше. Існують також зворотні транслятори, які здійснюють переказ з машинного коду в зрозумілий користувачеві мову програмування.

1.3 Аналіз і характеристика об’єкта проектування

Основний об’єкт проектування — описаний вище, а саме інтерпретатор для створеної скриптової мови програмування.

Власне інтерпретатор – це виконуваний файл, який поетапно читає програму, а потім обробляє, відразу виконуючи її інструкції. Іншими словами, інтерпретатор виконує програму поетапно.Об'єктний код не передається процесору, інтерпретатор сам є об'єктним кодом, побудованим таким чином, щоб його можна було викликати в певний час. В процесі роботи над кваліфікаційною роботою, в першу чергу увага приділялася розробці синтаксису скриптової мови, та простому використанні інтерпретатора.

1.4 Огляд існуючих скриптових мов програмування та їх інтерпретаторів

Історія виникнення перших скриптових мов починається з 1960-их років. Перші скриптові мови розроблялися з метою автоматизації процесів таких, як редагування, компіляція, лінковка. Спочатку їх можливості були дуже мізерні і вони просто рятували від необхідності повтору кількох команд, тобто це були просто текстові файли, що містять команди, які зазвичай вводилися операторами. У більш вузькому сенсі під скриптовою мовою можна розуміти спеціалізований мова для розширення можливостей командної оболонки або текстового редактора і засобів адміністрування операційних систем.

1.4.1 Загальні характеристики скриптових мов програмування

До загальних характеристик скриптових мов програмування можна віднести:

– динамічна типізація – тобто, змінна отримує тип, під час надання значення, а не оголошення змінної. Таким чином, одна і та ж змінна може набувати значень різних типів. За рахунок того, що немає необхідності оголошувати змінні відбувається зменшення програмного коду, пропадають операції приведення типу, а функції приймають параметри різних типів. Але помилки можна виявити тільки в момент виконання;

– інтерпретація – тобто вихідний код програми виконується за допомогою спеціальної програми-інтерпретатора без перетворення в машинний код для безпосереднього виконання центральним процесором. Для того, щоб писати код який буде підтримуватися на різних платформах, потрібно тільки мати багатоплатформовий інтерпретатор. Але через це програма працює повільніше, ніж написана на компільовані мови програмування;

– «склеювання» – процес який використовується при розробці ігор для «склеювання» та налаштування функціональності ігрового движка, який найчастіше розробляється на об'єктно-орієнтовній мові програмування C ++.

1.4.2 Класифікація скриптових мов програмування

Мови програмування та скриптові мови можуть бути класифіковані великою кількістю різних способів. Якщо розглядати швидкодію, скриптові мови діляться на певні категорії, це мови динамічного розбору програмного коду і попередньо компільовані мови. Принцип роботи мови динамічного розбору доволі простий. Спочатку такі мови зчитують інструкції з файлу програми мінімальними блоками, і виконують ці блоки, без подальшого зчитування залишеного коду. Попередньо компільовані мови в свою чергу перетворюють всю програму в так званий байт-код і лише потім виконують його.

По застосуванню мови можна розділити на три типи:

– командно-сценарні;

– прикладні сценарні;

– універсальні сценарні.

Далі в роботі буде розглянуто кожен тип скриптової мови, та наведені приклади деяких інтерпретаторів.

1.4.3 Командно-сценарні мови

Перші командно-сценарні мови з'явилися в 60-х роках. Їхнім призначенням було управління завданнями в операційних системах. До командно-сценарних мов входять мови пакетної обробки та мови командних оболонок, до яких відносяться sh, csh для Unix. До мов, які найчастіше використовуються в пакетному режимі обробки відносяться:

– AutoHotkey

– JCL

– sh

– bash

– csh

– ksh

– Pilot

– REXX

– AppleScript

– VBScript

– PowerShell

– AutoIt

Наприклад, скрипти написані на мові VBScript (Visual Basic Script) можуть бути інтерпретовані за допомогою двох інтерпретаторів: віконного WScript (див. рис. 1.1) та консольного CScript (див. рис. 1.2), обидва інтерпретатора це Windows Script Host (WSH).

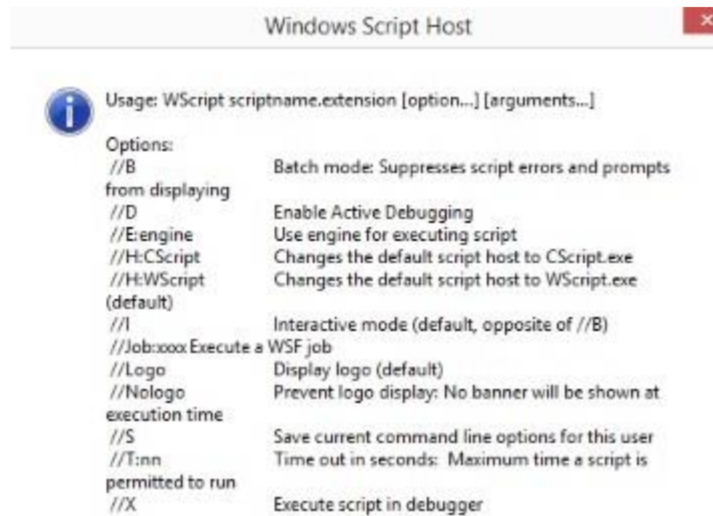

Рисунок 1.1 – Приклад роботи WScript

Рисунок 1.2 – Приклад роботи CScript

1.4.4 Прикладні-сценарні мови

Прикладні-сценарні мови починають з'являтися в 80-і роки, саме тоді, коли на промислових персональних комп'ютерах, у користувача з’являється можливість інтерактивного спілкування з операційною системою. Такі мови працювали у клієнтській частині програмного забезпечення, клієнт-серверної архітектури. До таких мов можна віднести:

– AutoLISP;
– ECMAScript і його діалекти (JScript, JavaScript, ActionScript) ;
– Emacs Lisp;
– ERM;
– Game Maker Language;

- LotusScript ;
- Lua;
- MQL4 script;
- UnrealScript;
- VBA;
- Вбудована мова програмування 1С: Підприємство.

1.4.5 Мови загального призначення

Мови загального призначення отримали свою популярність серед користувачів завдяки веб-програмуванню. Мови цього типу почали розвиватися з 90-х років. До мов загально призначення можна віднести:

- Tcl (Tool command language);
- Lua;
- Perl;
- PHP;
- Python;
- REBOL;
- Ruby;
- JavaScript;

Слід зауважити, що багато мов цієї категорії мають ширше застосування, ніж в якості просто скриптових мов програмування, наприклад JavaScript, який в деяких випадках може виконуватися на сервері. Одними з найпоширеніших інтерпретаторів JavaScript є V8 та Rhino.

1.5 Інтерпретатор і його відмінність від компілятора

Як компілятор так і інтерпретатор мають одне призначення – конвертувати інструкції мови високого рівня (як C або Java) в бінарну форму, зрозумілу комп'ютеру. Це програмне забезпечення, яке використовується для запуску високорівневих програм і кодів виконуваних різні завдання. Для різних високорівневих мов розроблені специфічні компілятори / інтерпретатори. Не дивлячись на те, що, як компілятор так і інтерпретатор переслідують одну і ту ж мету, вони відрізняються способом виконання свого завдання, тобто конвертації високорівневої мови в машинні інструкції [1].

Компілятори це спеціальні програми. Вони приймають програму, яку ви написали. Потім аналізують і розбирають кожну частину програми і будують машинний код для процесора. Часто його також називають об'єктним кодом.

На одному з етапів процесу опрацювання використовується компоновщик, що приймає частини програми, які окремо були перетворені в об'єктний код, і пов'язує їх в один виконуваний файл. На рисунку 1.3, можна побачити даний процес:

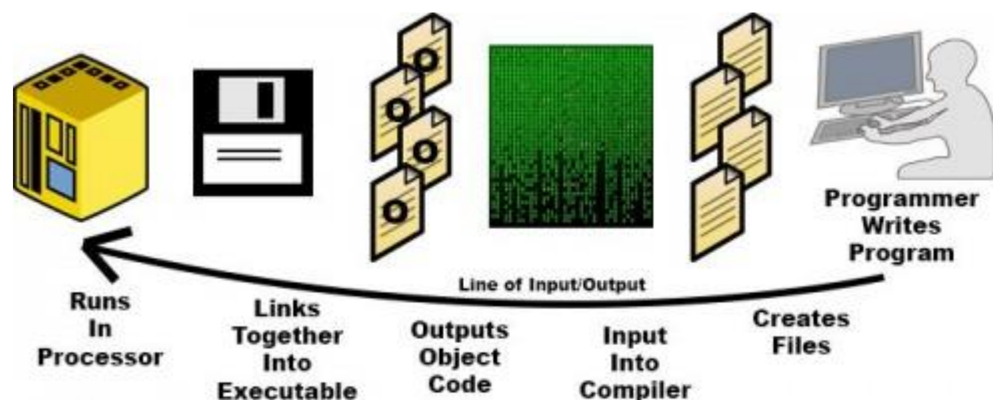

Рисунок 1.3 – Процес роботи компілятора

Кінцевим елементом цього процесу є виконуваний файл. Коли ви запускаєте або повідомляєте комп'ютера, що це виконуваний файл, він відразу запускає програму і виконує її без будь-якого додаткового перетворення. Це

ключова характеристика процесу компіляції – його результат повинен бути виконуваним файлом, що не вимагає додаткового перекладу, щоб процесор міг почати виконувати першу інструкцію і всі наступні за нею [1].

Перші компілятори були написані безпосередньо через машинний код або з використанням асемблерів. Але мета компілятора очевидна: перевести програму в виконуваний машинний код для конкретного процесора.

Альтернативою компіляції є інтерпретація. Тому схема роботи програми змінюється (див. рис.1.4):

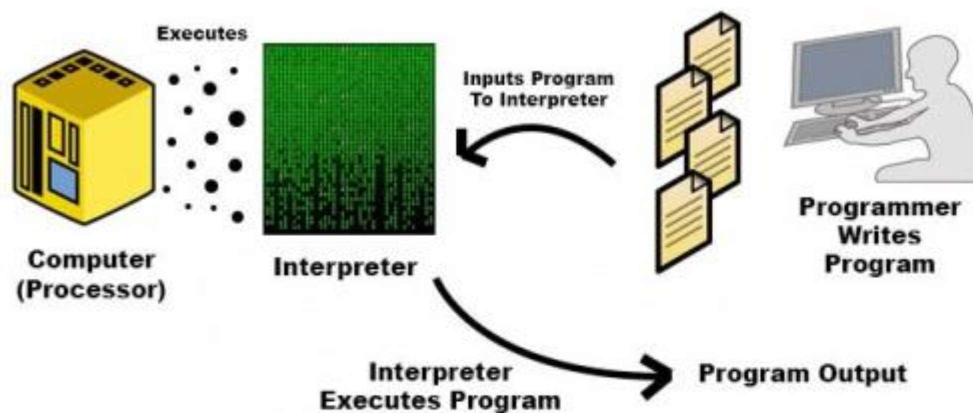

Рисунок 1.4 – Процес роботи інтерпретатора

На ній ми бачимо, що на відміну від компілятора, інтерпретатор завжди повинен бути під рукою, щоб ми могли викликати його і запустити програму. У певному сенсі інтерпретатор стає процесором. Програми, написані для інтерпретації, називаються «скриптами», тому що вони є сценаріями дій для іншої програми, а не прямим машинним кодом [1].

Наприклад, так працюють такі мови програмування, як Python. Ви пишете програму. Потім вводите код в інтерпретатор Python, і він виконує всі описані вами кроки. У командному рядку ви можете ввести приблизно наступне **python myprogram.py**. У цій команді Python – це виконуваний файл. Ви вводите в нього все, що знаходиться в файлі myprogram.py, і він виконує ці інструкції. Комп'ютер не запустить myprogram.py без Python. Це не машинний код, який розуміє процесор. Можна скомпілювати програми Python в об'єктний

або машинний код і запустити його безпосередньо в процесорі. Але ця процедура включає в себе компіляцію коду і додавання як її частини всього інтерпретатора Python.

1.6 Типи інтерпретаторів

Інтерпретатор зазвичай використовує одну з наступних стратегій для виконання програми:

– розбираємо в вихідний код і виконати його безпосередньо;

– перетворює вихідний код в деякий ефективне проміжне представлення і негайно виконує його;

– явно виконує збережений попередньо скомпільований код, створений компілятором, який є частиною системи інтерпретатора.

Виходячи с цього існує така умовна варіація інтерпретаторів:

– інтерпретатори байт-кодів – існує цілий спектр можливостей між інтерпретацією і компіляцією, в залежності від обсягу аналізу, виконаного перед виконанням програми. Наприклад, Emacs, Lisp компілюється в байт-код, який представляє собою сильно стислий і оптимізоване уявлення вихідного коду Lisp, але не є машинним кодом (і, отже, не прив'язаний до якогось конкретного устаткування). Цей «скомпільований» код потім інтерпретується інтерпретатором байт-коду (сам написаний на C). Скомпільований код в даному випадку є машинний код для віртуальної машини, який реалізований не апаратно, а в інтерпретаторі байт-коду. Такі компілятори- інтерпретатори іноді також називають компіляторами. У інтерпретаторі байт-коду кожна інструкція починається з байта, тому інтерпретатори байт-коду мають до 256 інструкцій, хоча не всі можуть використовуватися. Деякі байт-коди можуть займати кілька байтів і можуть бути довільно складними. Таблиці управління, яким не обов'язково коли-небудь проходити етап компіляції, диктують

відповідний алгоритм алгоритмічного управління через настроюються інтерпретатори аналогічно інтерпретаторів байт-коду;

– потокові інтерпретатори коду – інтерпретатори потокового коду схожі на інтерпретатори байт-коду, але замість байтів вони використовують покажчики. Кожна «інструкція» – це слово, яке вказує на функцію або послідовність інструкцій, за яким, можливо, слід параметр. Інтерпретатор многопоточного коду або зациклює вибірку інструкцій і виклик функцій, на які вони вказують, або вибирає першу інструкцію і переходить до неї, і кожна послідовність інструкцій закінчується вибіркою і переходом до наступної інструкції. На відміну від байт-коду немає ефективного обмеження на кількість різних інструкцій, крім доступної пам'яті і адресного простору. Класичним прикладом многопоточного коду є код Forth, який використовується в системах Open Firmware: вихідний мову компілюється в «F-код» (байт-код), який потім інтерпретується віртуальною машиною;

– інтерпретатори абстрактного синтаксичного дерева – В спектрі між інтерпретацією і компіляцією існує інший підхід: перетворити вихідний код в оптимізоване абстрактне синтаксичне дерево (AST), потім виконати програму, дотримуючись цієї деревовидної структури, або використовувати її для генерації коду [3]. При такому підході кожне речення потрібно аналізувати тільки один раз. Як перевага перед байт-кодом AST зберігає глобальну структуру програми і відносини між операторами (які губляться в поданні байт-коду), а при стисненні забезпечує більш компактне представлення. Таким чином, використання AST було запропоновано як кращий проміжний формат для оперативних компіляторів, ніж байт-код. Крім того, це дозволяє системі виконувати кращий аналіз під час виконання. Однак для інтерпретаторів AST викликає більше накладних витрат, ніж інтерпретатор байт-коду, через вузли, пов'язаних з синтаксисом, які не виконують корисної роботи, менш послідовного уявлення (що вимагає обходу більшої кількості покажчиків) і накладних витрат на відвідування дерева;

– інтерпретатори своєчасної компіляція – ще більше стирає відмінність між інтерпретаторами, інтерпретаторами байт-коду і компіляцією – це своєчасна компіляція (JIT), метод, при якому проміжне представлення компілюється в машинний код під час виконання. Це забезпечує ефективність виконання коду за рахунок часу запуску та збільшення використання пам'яті при першій компіляції байт-коду або AST [6]. Найраніший опубліковані JIT компілятор, як правило, пов'язаний з роботою на LISP від Джона Маккарті в 1960 р Адаптивної оптимізації є додатковою технікою, в якій інтерпретатор профілів в виконуваної програмою і компілює його найбільш часто виконуються частини в машинний код. Останній метод з'явився кілька десятиліть назад в таких мовах, як Smalltalk, в 1980-х роках. Компіляція «точно в строк» в останні роки привернула до себе загальну увагу розробників мов: Java, .NET Framework, більшість сучасних реалізацій JavaScript і Matlab тепер включають JIT;

– самостійний інтерпретатор – самоінтерпретатор – це інтерпретатор мови програмування, написаний на мові програмування, який може інтерпретувати сам себе; приклад – інтерпретатор BASIC, написаний на BASIC. Створення самоінтерпретатора вимагає реалізації мови на основній мові (який може бути іншою мовою програмування або асемблером). При наявності першого інтерпретатора, такого як цей, система завантажується, і нові версії інтерпретатора можуть бути розроблені на самій мові. Таким чином, Дональд Кнут розробив інтерпретатор TANGLE для мови WEB промислової стандартної системи набору тексту TeX. Важливим аспектом проектування при реалізації самоінтерпретатора є те, чи реалізована функція інтерпретується мови з такою ж функцією на основній мові інтерпретатора. Прикладом є те, чи реалізовано замикання на мові, подібній Lisp, з використанням замикань на мові інтерпретатора або реалізовано «вручну» зі структурою даних, явно зберігаючою середу. Чим більше функцій реалізовано тієї ж функцією на основній мові, тим менше можливостей у програміста інтерпретатора; інша поведінка при роботі з переповненням чисел не може

бути реалізовано, якщо арифметичні операції делеговані відповідними операціями на основній мові.

Отже для виконання поставленої задачі, було вирішено розробляти інтерпретатор абстрактного синтаксичного дерева. Далі ми розглянемо основні моменти такого інтерпретатора.

1.7 Основні поняття абстрактного синтаксичного дерева

AST – це Abstract Syntax Tree (абстрактне синтаксичне дерево), тобто дерево, яке в абстрактному вигляді представляє структуру програми. AST створюється парсером в міру синтаксичного розбору програми. Таке дерево містить повну синтаксичну модель програми без зайвих деталей (таких, як пробільні символи або коментарі). Щоб зрозуміти, як це працює, розглянемо, з чого складається типова мова програмування.

Типовий мова програмування складається з трьох синтаксичних елементів:

– вирази (expressions);
– інструкції (statements);
– оголошення (declarations).

Розглянемо кожен із цих елементів.

Вирази – це вираз формули в вихідному коді. У типовому якій мові програмування є як мінімум наступні типи виразів:

– доступ до змінної (var access): наприклад, y;
– літерал (literal): наприклад, 7.18 або "some string";
– унарний оператор (unary op): наприклад, -y;
– бінарний оператор (binary op): наприклад, y + 7.18 або y == 7.18;

– різні бінарні оператори зазвичай мають різний пріоритет і можуть групуватися дужками, але наприклад в функціональних мовах, оператори можуть не відрізнятися від функцій;

– типовий набір операторів: арифметичні, логічні, порівняння; такий набір вже дозволяє створювати повноцінні програми;

– виклик функції (fun call): наприклад, sqrt (pow (a, 4.0) + pow (b, 4.0)).

Інструкції – це дії в вихідному коді. Приклади інструкцій, характерних для процедурних мов:

– оголошення змінної з опціональною / обов'язковою инициализацией, наприклад, let value = ...; або int y = 300;

– присвоєння змінної, наприклад, variable = variable + acceleration

– спеціальні інструкції, наприклад, друк print x + 1 або assert isinstance (y, int) в мові Python;

– умовні інструкції, такі як if/else, switch;

– цикли, такі як while, do / while, for, foreach;

– інструкції потоку управління, наприклад, повернення з функцій return y + 2; або переривання циклів break;

– повернення (return) потрібні не у всіх мовах – іноді функція просто повертає останнім обчислене вираз;

– блоки коду, такі як {doA (); doB (); return 7; }.

Оголошення – це створення нової іменованої суті, такий як функція або тип. Оголошення типів бувають різноманітними: різні мови можуть дозволяти оголошувати новий тип як синонім старого типу, як структуру, як клас, як інтерфейс або іншим чином.

Далі буде наведено приклад AST для алгоритма Евкліда (див. рис. 1.5):

Рисунок 1.5 – AST для алгоритму Евкліда

AST інтенсивно використовується під час семантичного аналізу, де компілятор перевіряє правильність використання елементів програми та мови. Компілятор також генерує таблиці символів на основі AST під час семантичного аналізу. Повна обробка дерева дозволяє перевірити правильність програми. Після перевірки правильності AST служить базою для генерації коду.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ДОДАТКУ

2.1 Технічне завдання

2.1.1 Найменування та область застосування

Програмний продукт, що розробляється, отримує найменування “PSCRIPT”. Це скриптова мова програмування з створеним для неї інтерпретатором.

2.1.2 Призначення розробки

Основними можливостями даної розробки є написання простих скриптів завдяки реалізації простого синтаксису даної мови, що базується на так званій базовій мові. Самі скрипти виконуються як с командної строки так і з вихідних файлів.

2.2 Вимоги до програмного додатку

Скриптова мова повинна забезпечувати можливість виконання наступних функцій властивих базовій мові програмування:

- визначення змінних;
- базові математичні операції над змінними;
- використання конструкції if/else;
- використання конструкцій for;
- визначення функцій та їх використання.

2.2.1 Вимоги до надійності

Передбачити виняткові ситуації та забезпечити цілісність та точність вихідних данних.

2.2.2 Вимоги до складу і параметрів технічних засобів

Скриптова мова побудована за допомогою високорівневої мови програмування Python, саме тому для повноцінного користування розробкою, користувач повинен мати на своєму компьютері такі мінімальні конфігурації:

– версія Python 3.6;
– операційна система Windows 7/8/10, Linux.

2.3 Проектування системи

Для виконання поставленої задачі, було вирішено писати додаток на високорівневій мові програмування Python, в середовищі розробки PyCharm IntelliJ IDEA.

2.3.1 Переваги та недоліки Python

У зв'язку з тим що для написання скриптової мови, а також інтерпретатора для неї, була обрана мова програмування Python, далі буде наведено деякі переваги та недоліки мови Python [2].

До переваг можна віднести:

– гнучкість – це, на мою думку, основна перевага мови, так як завдяки своїй гнучкості мова отримала популярність серед багатьох розробників;
– можливість розширення – один із слоганів мови звучить як – Just

Import! – що повністю пояснює, наскільки мова розширюємо і був розширений за останні роки. Існують бібліотеки і фреймворки під будь-який тип завдань і потреб. Також величезним плюсом є те, що ми можемо використовувати C код з Python;

– простота синтаксису. Синтаксис – це саме те, через що я закохався в Python, з синтаксису було прибрано все зайве, код чистий і зрозумілий без зайвих дужок і виразів;

– інтерпретація. Інтерпретатор Python існує для всіх популярних платформ і за замовчуванням входить в більшість дистрибутивів Linux, а значить є на більшості серверів «з коробки»;

– PEP – єдиний стандарт для написання коду, що робить код підтримуваним і читабельним навіть при переході від одного програміста до іншого. Це підтримує популярність Python;

– Open Source – код інтерпретатора Python є відкритим, що дозволяє будь-кому, хто зацікавлений у розвитку мови взяти участь в його розробці і поліпшити його. Якщо дивитися деталі релізу однією з версій мови, то можна помітити, що величезні частини нового функціоналу реалізовані сторонніми розробниками;

– ком'юніті – навколо Python утворилося досить дружнє і приємне ком'юніті, яке готове прийти на допомогу будь-якому починаючому або вже вмілому розробнику і розібратися в його проблемі.

До недоліків відноситься:

– продуктивність. Більшість розробників, та й сам творець мови, сходяться на думці, що Python не так спритний, наскільки хотілося б. Це обумовлено тим, що Python інтерпретована мова. Але навіть у порівнянні з іншими інтерпретуються мовами помітно, що Python програє в продуктивності. Але це легко можна нівелювати за допомогою C реалізацій того чи іншого проблемного ділянки коду. В умовах сьогоднішніх потужностей

– це несильно помітно;

– динамічна типізація – через динамічної типізації Python споживає більше ресурсів, ніж міг би, але це часто компенсується внутрішнім кешування;

– Global Interpriter Lock. На даний момент це є основною проблемою продуктивності в Python, а також цим обумовлена погана реалізація багатопоточності. Код GIL не змінювався з першої версії мови. Це явно вказує на те, що він застарів. Залишається сподіватися, що розробники приділять цьому увагу в найближчих релізах.

2.3.2 Розширена форма Бекуса-Наура

Задля розуміння синтаксису та можливостей розробленої скриптової мови, мною була розроблена розширена форма Бекуса-Наура тобто формальна система визначення синтаксису, в якій одні синтаксичні категорії послідовно визначаються через інші.

Формальний опис основних символів у розробленій мові, можна представити у наведеному нижче вигляді.

letter = ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’ | ’h’ | ’i’ | ’j’ | ’k’ | ’l’ | ’m’ | ’n’ | ’o’ | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’ | ’v’ | ’w’ | ’x’ | ’y’ | ’z’ | ’A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’ | ’G’ | ’H’ | ’I’ | ’J’ | ’K’ | ’L’ | ’M’ | ’N’ | ’O’ | ’P’ | ’Q’ | ’R’ | ’S’ | ’T’ | ’U’ | ’V’ | ’W’ | ’X’ | ’Y’ | ’Z’;

digit = ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’;

symbol = "(" | ")" | """ | "=" | "+" | "*" | "-" | "/";

special symbol = "[" | "]" | "{" | "}" | "==" | "|" | "," | ";" | " " | "!>";

character = letter | digit | "_";

identifier = letter , { letter | digit | "_" };

number = ["-"], digit, { digit };

string = '"', character, { character }, '"';

Основні операції у розробленій мові будуються на декількох наведених нижче конструкція.

expression = identifier | number | string | '(', expression, ')'

| expression, '+', expression | expression, '-', expression

| expression, '*', expression | expression, '/', expression;

condition = expression, "==", expression;

assignment = identifier , "=" , expression;

statement = (expression | assignment);

А перелік доступних конструкцій, які у своїй основі містять наведений вище опис основних символів та операцій, має такий вигляд.

variable assignment = "VAR", " ", assignment;

output value = "PRINT", " ", identifier;

if stmnt = "IF", " ", condition, " ", statement, " ", "ELSE", " ", statement;

for stmnt = "FOR", " ", assignment, " ", "TO", " ", expression, " ", "THEN", " ", statement;

func definition = "FUN", " ", identifier, "(", ")", "!>", " ", statement;

func call = identifier, "(", ")";

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

Скриптова мова та інтерпретатор написані на високорівневій мові програмування Python, в середовищі розробки PyCharm. Додатково при розробці інтерпретатору використовувалась бібліотека для написання парсерів SLY.

3.1 Структура проекту

Для початку потрібно розуміти, що для створення інтерпретатору потрібні три речі. По перше це лексер, завдяки ньому відбувається такий процес , як лексичний аналіз, тобто відбувається розбір вхідної послідовності символів на розпізнані групи – лексеми – з метою отримання на виході ідентифікованих послідовностей, так званих «токенів». Наступним кроком потрібно провести синтаксичний аналіз, тобто на основі списку токенів відбувається побудова абстрактного синтаксичного дерева. Цю задачу виконує парсер. Після побудови абстрактного синтаксичного дерева його потрібно проаналізувати та виконати, цим займається сам інтерпретатор.

Саме тому до основних модулів проекту відносяться (див. рис. 3.1):

– pscript_interpreter;
– pscript_lexer;
– pscript_parser.

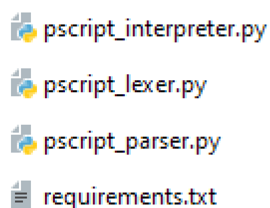

Рисунок 3.1 – Повний список модулів

Взаємодія модулів між собою наведена в таблиці 3.1.

Таблиця 3.1 – Взаємодія між модулями

Назва модулю	З якими модулями взаємодіє
pscript_interpreter	pscript_lexer pscript_parser
pscript_lexer	Sly
pscript_parser	Sly pscript_lexer

Далі будуть розглядатися основні модулі проекту, взаємодія користувача з інтерпретатором та реалізація основного функціоналу проекту. Повний код наведено у Додатку А .

3.2 Опис основних модулів проекту

Основу інтерпретатора базується на трьох основних модулях, за допомогою яких реалізовано головну функціональність. В таблиці 3.2 наведений опис кожного модуля.

Таблиця 3.2 – Опис основних модулів

Назва модулю	Опис
pscript_lexer	Модуль pscript_lexer використовується для розбиття отриманого коду на колекцію лексем, визначену правилами регулярних виразів.
pscript_parser	Модуль pscript_parser використовується для розпізнавання отриманих лексем, та побудови на їх основі абстрактно синтаксичного дерева.

Кінець таблиці 3.2

Назва модулю	Опис
pscript_interpreter	У даному модулі реалізовано порядковий обхід та виконання отриманого абстрактно синтаксичного дерева з модулю pscript_parser.

3.2.1 Огляд функціональності модулів pscript_lexer та pscript_parser

У модулі pscript_lexer визначений клас BasicLexer, що наслідує клас Lexer із бібліотеки SLY. У класі BasicLexer визначені наступні дані та методи (див. табл. 3.3).

Таблиця 3.3 – Дані та методи класу BasicLexer

Дані та методи класу	Призначення
tokens = { … }	Це змінна, яка визначає всі можливі імена токенів, які може створити лексер.
ignore = “ … “	Це змінна,в якій визначаються окремі символи, які слід повністю ігнорувати між токенами у вхідному потоці.
literals = { … }	Це один символ, який повертається "як є", коли лексер зустрічає його. Таким чином, якщо правило починається з одного з буквальних символів, воно завжди матиме пріоритет.

Продовження таблиці 3.3

Дані та методи класу	Призначення
… NAME = r'[a-zA-Z_][a-zA-Z0-9_]*' STRING = r'\".*?\"' …	Токени задаються написанням правила регулярного виразу. Ім'я кожного правила має збігатися з одним із імен токенів, наданих у наборі токенів.
@_(r'\d+') def NUMBER(self, t)	Це визначений декоратор та метод, який кожне отримане число із вхідного потоку, призводить до цілочисельного типу.

Модуль pscript_lexer це лексичний аналізатор. Лексичний аналізатор розділений на два процеси:

– сканування, яке складається з простих процесів, таких як видалення коментарів і ущільнення послідовностей пробільних символів в один;

– лексичний аналіз, який генерує токени з вихідного потоку сканера.

При розгляді лексичного аналізу необхідно відзначити деякі важливі терміни, пов'язані з цією фазою:

– токен – пара, що складається з імені токена і необов'язкового, атрибута. Ім'я токена є певний абстрактний символ, який є типом лексичної одиниці (наприклад, ключове слово мови або позначення, що даний токен є ідентифікатором).

– лексема – це послідовність символів вихідної програми, яка відповідає шаблоном токена і ідентифікується лексичним аналізатором як екземпляр токена.

Виходячи з теми кваліфікаційної роботи, а саме розробки інтерпретатора скриптової мови, було прийнято рішення, що дана мова буде базуватися на

концепціях Basic language, саме з цієї причини набір доступних токенів, які наш інтерпретатор буде розуміти. Зараз цей набір токенів має такий вигляд (див. рис. 3.2).

- Токен для позначення імені змінної
- Токен для позначення числа
- Токен для позначення рядка
- Токен для визначення змінної
- Токен для виводу змісту змінної на екран
- Токени для позначення умовного оператору
- Токени для позначення циклу
- Токени для визначення та виклику функцій

```
tokens = {"NAME", "NUMBER", "STRING",
          "VAR", "PRINT", "IF",
          "THEN", "ELSE", "FOR",
          "FUN", "TO", "ARROW", "EQEQ"}
```

Рисунок 3.2 – Набір токенів

Кожен токен який ми будемо використовувати при написанні скриптів нашою мовою має своє визначення у вигляді регулярного виразу (див. рис. 3.3).

```
VAR = r'VAR'
PRINT = r'PRINT'
IF = r'IF'
THEN = r'THEN'
ELSE = r'ELSE'
FOR = r'FOR'
FUN = r'FUN'
TO = r'TO'
ARROW = r'!>'
NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'
STRING = r'\".*?\"'
EQEQ = r'=='
```

Рисунок 3.3 – Визначення токенів

Як можна побачити, не всі токени ми визначаємо одразу. Наприклад токен NUMBER, потребує особливого контролю, так як розроблена мова підтримує математичні операції тільки над цілими числами. Саме тому для контролю над отриманими числами ми створюємо функцію, яка починається зі створення декоратору, який містить у собі умову для обробки токену у вигляді регулярного виразу (див. рис. 3.4).

```
@_(r'\d+')
def NUMBER(self, t):
    t.value = int(t.value)
    return t
```

Рисунок 3.4 – Обробка отриманих числових значень

Даний декоратор призначений для того щоб сказати, що це має бути одна або кілька цифр, і коли ми отримуємо число, ми хочемо перетворити його на ціле число, а потім повертаємо наш токен. Повний код модулю PScript_lexer знаходиться в Додатку А.

У модулі PScript_parser визначений клас BasicParser, що наслідує клас Parser із бібліотеки SLY. У класі BasicParser визначені наступні основні дані та методи (див. табл. 3.4):

Таблиця 3.4 – Дані та методи класу BasicParser.

Дані та методи класу	Призначення
tokens = …	Це змінна, яка отримує всі можливі імена токенів, визнечені у класі BasicLexer.

Продовження таблиці 3.4

Дані та методи класу	Призначення
precedence = (…)	Це змінна, призначена для упорядкування токенів за пріорітетом, від найнижчого до вищого пріоритету.
@_('') def statement(self, p)	Даний метод призначений для обробки інших токенів.
@_('expr') def statement(self, p)	Даний метод призначений для обробки чисел, символів та строк, та побудови різного роду виразів.
@_(‘var_assign’) def statement(self, p)	Даний метод призначений для визначення та пере визначення змінних.

Отож основним призначенням модулю pscript_parser є обробка отриманих синтаксичних конструкцій з модулю pscript_lexer, та побудова абстрактного синтаксичного дерева. Спочатку ми отримуємо всі можливі та визначенні імена токенів з імпортованого модулю pscript_lexer даною конструкцією (див. рис. 3.5).

```
tokens = pscript_lexer.BasicLexer.tokens
```

Рисунок 3.5 – Імпорт токенів до парсеру

Після отримання доступних токенів, нам потрібно визначити пріоритети для виконання математичних операцій. За це відповідає змінна precedence і виглядає вона наступним чином (див. рис. 3.6).

```
precedence = (
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('right', 'UMINUS'),
    )
```

Рисунок 3.6 – Пріоритети для виконання математичних операцій

Пріоритети впорядковані від найнижчого до найвищого.

Далі буде розглянуто реалізацію основних конструкцій мови таких, як визначення змінних, базові математичні операції, цикл For, умовний оператор If/Else, визначення функцій. Код всього модулю розташований у Додатку Б.

Розпочнемо з визначення змінних. Після того, як лексер обробляє отриманий код і розбиває його на лексеми, ці лексеми передаються в парсер. Для визначення змінної користувач повинен ввести ключове слово VAR, назву змінної та її значення. Коли парсер отримує розбиті лексеми, він починає пошук по своїм методам і шукає той, що відповідає даній конструкції (див. рис. 3.7).

```
@_('VAR NAME "=" expr')
def var_assign(self, p):
    return 'var_assign', p.NAME, p.expr
```

Рисунок 3.7 – Метод визначення змінної

Як можна побачити, шаблон лексеми прописаний в декораторі. Після декоратора визначається функція, яка повертає назву операції, назву змінної та вираження, що в даному випадку відповідає значенню змінної.

Базові математичні операції будуються схожим способом. Тобто спочатку лексер розбиває отриманий код на лексеми, після чого передає їх у парсер і той в залежності від математичної операції повертає відповідну конструкцію (див. рис. 3.8).

```
@_('expr "+" expr')
def expr(self, p):
    return 'add', p.expr0, p.expr1

@_('expr "-" expr')
def expr(self, p):
    return 'sub', p.expr0, p.expr1

@_('expr "*" expr')
def expr(self, p):
    return 'mul', p.expr0, p.expr1

@_('expr "/" expr')
def expr(self, p):
    return 'div', p.expr0, p.expr1
```

Рисунок 3.8 – Методи базових математичних операцій

За схожим принципом обробляється цикл For (див. рис. 3.9), умовний оператор If/Else (див. рис. 3.10), визначення функцій (див. рис. 3.11).

```
@_('FOR var_assign TO expr THEN statement')
def statement(self, p):
    return 'for_loop', ('for_loop_setup', p.var_assign, p.expr), p.statement
```

Рисунок 3.9 – Метод цикла For

```
@_('IF condition THEN statement ELSE statement')
def statement(self, p):
    return 'if_stmt', p.condition, ('branch', p.statement0, p.statement1)
```

Рисунок 3.10 – Метод умовного оператору If/Else

```
@_('FUN NAME "(" ")" ARROW statement')
def statement(self, p):
    return 'fun_def', p.NAME, p.statement
```

Рисунок 3.11 – Метод визначення функції

3.2.2 Приклад роботи модулів pscript_lexer та pscript_parser

Розглянемо приклад роботи кожного модуля окремо. Після отримання вхідного набору символів лексер проводить лексичний аналіз і на виході ми отримуємо набір токенів (див. рис. 3.12).

```
[*] shell [*] > VAR a = 0
Token(type='VAR', value='VAR', lineno=1, index=0)
Token(type='NAME', value='a', lineno=1, index=4)
Token(type='=', value='=', lineno=1, index=6)
Token(type='NUMBER', value=0, lineno=1, index=8)
[*] shell [*] > FOR a = 0 TO 5 THEN a+1
Token(type='FOR', value='FOR', lineno=1, index=0)
Token(type='NAME', value='a', lineno=1, index=4)
Token(type='=', value='=', lineno=1, index=6)
Token(type='NUMBER', value=0, lineno=1, index=8)
Token(type='TO', value='TO', lineno=1, index=10)
Token(type='NUMBER', value=5, lineno=1, index=13)
Token(type='THEN', value='THEN', lineno=1, index=15)
Token(type='NAME', value='a', lineno=1, index=20)
Token(type='+', value='+', lineno=1, index=21)
Token(type='NUMBER', value=1, lineno=1, index=22)
```

Рисунок 3.12 – Набір токенів

Так як модуль pscript_lexer, імпортується в модуль pscript_parser, розроблений парсер одразу отримує список токенів та проводить синтаксичний аналіз, на основі якого будує абстрактно синтаксичне дерево (див. рис. 3.13).

```
[*] shell [*] > VAR a = 0
('var_assign', 'a', ('num', 0))
[*] shell [*] > FOR a = 0 TO 5 THEN a+1
('for_loop', ('for_loop_setup', ('var_assign', 'a', ('num', 0)), ('num', 5)), ('add', ('var', 'a'), ('num', 1)))
[*] shell [*] >
```

Рисунок 3.13 – Приклад AST

3.2.3 Огляд функціональності модулю pscript_interpreter

Модуль pscript_interpreter містить у собі розроблені та імпортовані модулі для лексичного та синтаксичного аналізу, а саме pscript_lexer та pscript_parser, також клас BasicExecute, який містить у собі метод для рекурсивного проходу та виконання абстрактного синтаксичного дерева яке ми отримуємо після синтаксичного аналізу, що робить його інтерпретатором для розробленої скриптової мови.У класі BasicExecute, визначені наступні дані та методи (див. табл. 3.5).

Таблиця 3.5 – Дані та методи класу BasicExecute

Дані та методи класу	Призначення
env = {}	Це змінні, які виконує роль оточення.
walkTree(self, node)	Це метод який відповідає за рекурсивне проходження абтрактного-синтаксичного дерева, та виконаня усіх інтсрукцій.

У модулі pscript_interpreter, визначено клас BasicExecute. У даному класі визначений метод walkTree(). В даному методі все починається з перевірки на звичайні значення, тобто звичайне число та рядок, перехід на новий рядок. Якщо умова вірна, то інтерпретатор просто повертає введене значення (див. рис. 3.14).

```
if isinstance(node, int):
    return node
if isinstance(node, str):
    return node
if node is None:
    return None
```

Рисунок 3.14 – Перевірка звичайних значень

У тому випадку, коли інтерпретатор отримує абстрактне синтаксичне дерево, наприклад, для визначення змінної воно має такий вигляд ('var_assign', 'a', ('num', 0)), інтерпретатор починає шукати умову яка відповідає першому ключовому слову, а саме var_assign, для даного ключового слова вона має такий вигляд (див. рис. 3.15).

```
if node[0] == 'var_assign':
    self.env[node[1]] = self.walkTree(node[2])
    return node[1]
```

Рисунок 3.15 – Визначення змінної

Тобто, якщо node[0] дорівнює var_assign, у визначене оточення, під отриманим іменем змінної з node[1], присвоюється результат рекурсивного проходу по дереву з node[2]. Це робиться для випадків коли зміна містить у собі будь яке значення і ми хочемо присвоїти це значення іншій змінній. У даному випадку node[2] – це конструкція типу **('num', 0)**, інтерпретатор завдяки рекурсивному методу walkTree(), знайде умову ‘num’ та поверне число (див. рис. 3.16).

```
if node[0] == 'num':
    return node[1]
```

Рисунок 3.16 – Умова для повернення числа

Точно таким способом, якби замість ‘num’, було б ‘str’, що відповідає за рядки, інтерпретатор повернув би й рядок.

У випадку з умовним оператором If/Else, для якого абстрактне синтаксичне дерево має наступний вигляд ('if_stmt', ('condition_eqeq', ('var', 'a'), ('num', 0)), ('branch', ('var_assign', 'a', ('add', ('var', 'a'), ('num', 1))), ('var_assign', 'a', ('str', '"Hello world"')))), інтерпретатор шукає потрібну умову, а саме if_stmt (див. рис. 3.17).

```
if node[0] == 'if_stmt':
    result = self.walkTree(node[1])
    if result:
        return self.walkTree(node[2][1])
    return self.walkTree(node[2][2])
```

Рисунок 3.17 – Умовній оператор If/Else

У змінну result, присвоюється рекурсивний прохід з node[1], а саме ('condition_eqeq', ('var', 'a'), ('num', 0)), інтерперетатор шукає потрібну умову, тобто 'condition_eqeq' і виконує код (див. рис. 3.18).

```
if node[0] == 'condition_eqeq':
    return self.walkTree(node[1]) == self.walkTree(node[2])
```

Рисунок 3.18 – Порівняння умови

І якщо результат істинний то виконується рекурсивний прохід першої частини коду ('var_assign', 'a', ('add', ('var', 'a'), ('num', 1))), якщо ні то другої частини коду ('var_assign', 'a', ('str', '"Hello world"')).

Для обробки функцій, існують дві команди, ‘fun_def’, та ‘fun_call’, для визначення та виклику функцій відповідно. У випадку визначення функції, у визначене оточення за вибраним ім’ям функції присвоюється дії, які вона повинна виконувати, а у випадку виклику функції, інтепретатор намагається рекурсивно викликати функцію, і якщо її не визначено у оточенні то інтепретатор виводить помилку (див. рис. 3.19).

```
if node[0] == 'fun_def':
    self.env[node[1]] = node[2]

if node[0] == 'fun_call':
    try:
        return self.walkTree(self.env[node[1]])
    except LookupError:
        print("Undefined function '%s'" % node[1])
```

Рисунок 3.19 – Умова визначення та виклику функції

Повний код модулю знаходиться у Додатку В.

3.3 Приклад роботи додатку

Коли ми запускаємо інтерпретатор то перш за все ми бачимо, що нам пропонується обрати режим роботи. Існує два способи роботи з інтерпретаторм, через командний рядок чи виконати код із вихідного файлу (див. рис. 3.20).

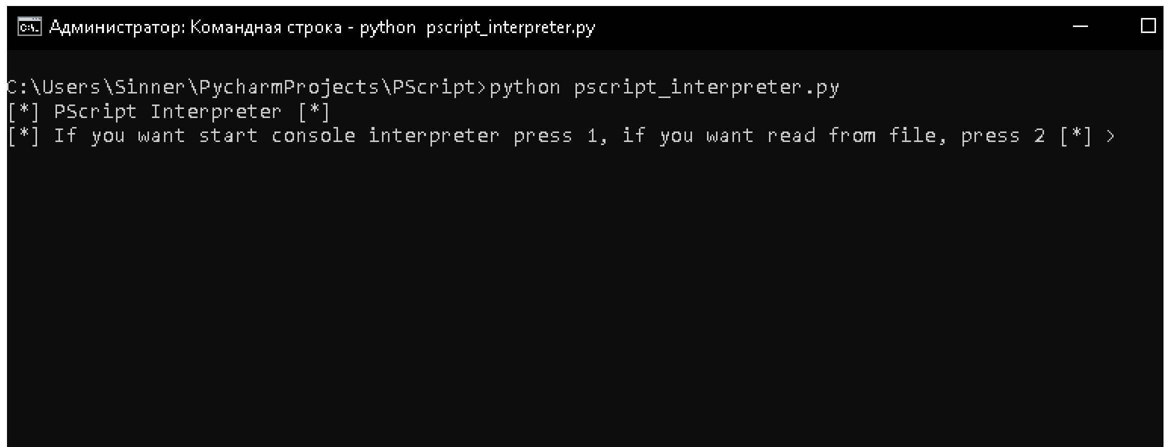

Рисунок 3.20 – Режими роботи інтерпретатору

Якщо користувач обирає роботу через командний рядок, інтепретатор запускається і користувач може починати писати скрипти (див. рис. 3.21).

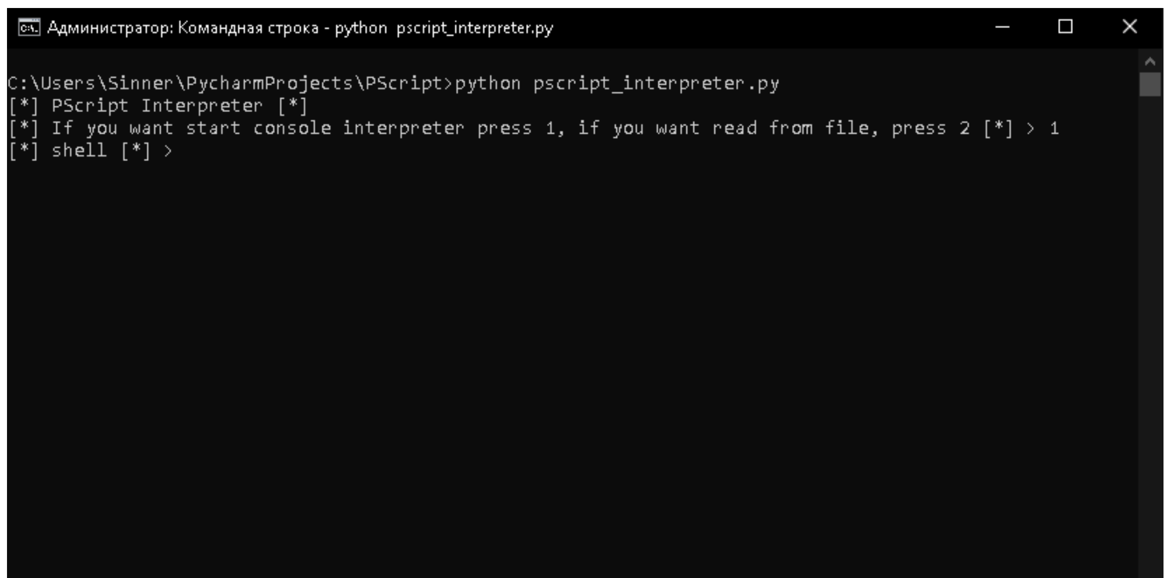

Рисунок 3.21 – Робота у консольному режимі

Розглянемо базові можливості інтепретатору. По перше це визначення змінної за допомогою ключового слова VAR та вивід значення цієї змінної на екран (див. рис. 3.22).

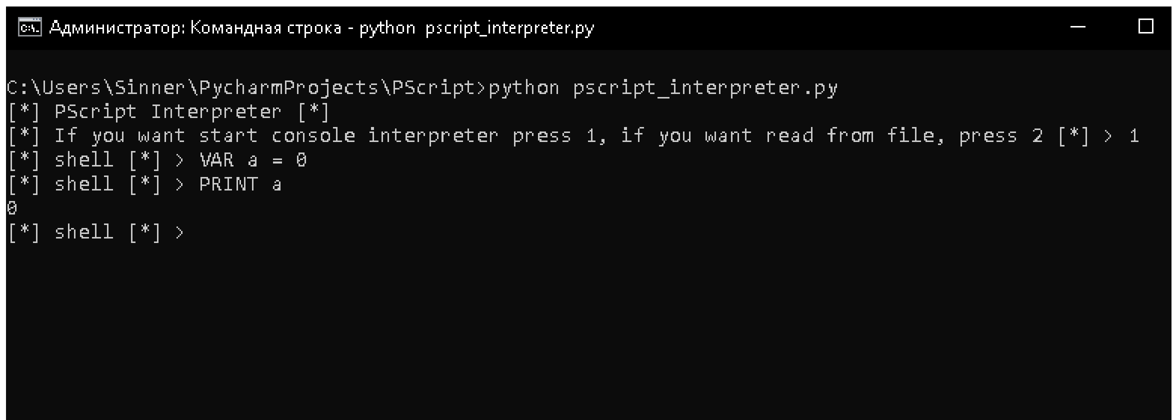

Рисунок 3.22 – Визначення змінної та вивід значення користувачу

По друге користувач може ввести будь який математичний вираз і одразу побачити його результат (див. рис. 3.23).

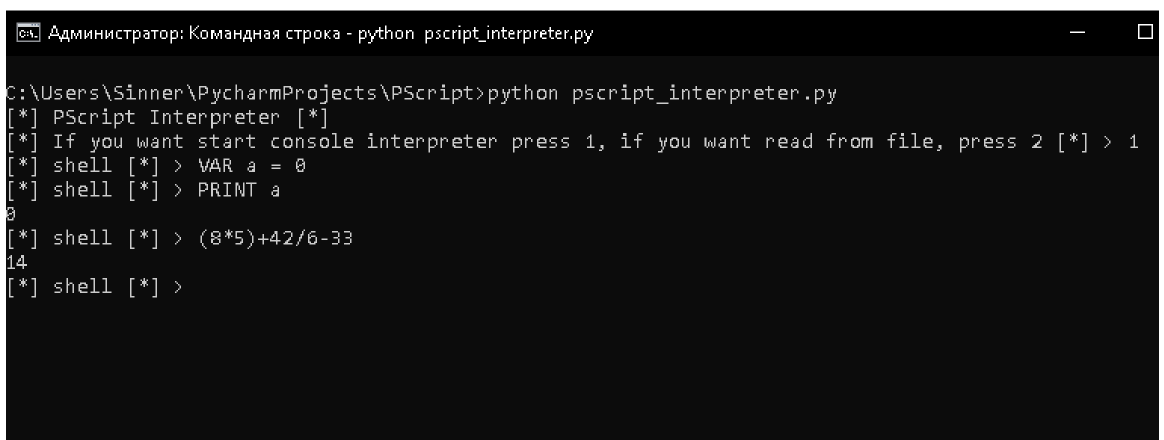

Рисунок 3.23 – Математичний вираз та його результат

Користувач може визначити свою функцію та викликати її за допомогою вводу її імені (див. рис. 3.24).

Администратор: Командная строка - python pscript_interpreter.py

```
C:\Users\Sinner\PycharmProjects\PScript>python pscript_interpreter.py
[*] PScript Interpreter [*]
[*] If you want start console interpreter press 1, if you want read from file, press 2 [*] > 1
[*] shell [*] > VAR a = 0
[*] shell [*] > PRINT a
0
[*] shell [*] > (8*5)+42/6-33
14
[*] shell [*] > VAR b = "Hello World!"
[*] shell [*] > FUN output() !> PRINT b
[*] shell [*] > output()
"Hello World!"
[*] shell [*] >
```

Рисунок 3.24 – Визначення та виклик функції

У розпоряджені користувача присутній умовний оператор if/else та конструкція циклу for (див. рис. 3.25).

Администратор: Командная строка - python pscript_interpreter.py

```
C:\Users\Sinner\PycharmProjects\PScript>python pscript_interpreter.py
[*] PScript Interpreter [*]
[*] If you want start console interpreter press 1, if you want read from file, press 2 [*] > 1
[*] shell [*] > VAR a = 0
[*] shell [*] > PRINT a
0
[*] shell [*] > (8*5)+42/6-33
14
[*] shell [*] > VAR b = "Hello World!"
[*] shell [*] > FUN output() !> PRINT b
[*] shell [*] > output()
"Hello World!"
[*] shell [*] > IF a == 0 THEN a = a + 5 ELSE a = a + 10
[*] shell [*] > PRINT a
5
[*] shell [*] > FOR a = 0 TO 5 THEN PRINT b
"Hello World!"
"Hello World!"
"Hello World!"
"Hello World!"
"Hello World!"
[*] shell [*] >
```

Рисунок 3.25 – Умовний оператор if/else, цикл for

Якщо при запуску інтерпретатору користувач обрав другий метод, тобто виконання коду із вхідного файлу, йому потрібно ввести шлях до даного файлу і результат виведеться на екран (див. рис. 3.26 – 3.27).

```
1  VAR a = 5
2  VAR c = "Hello World!"
3  FUN output() !> IF a == 10 THEN PRINT a ELSE PRINT c
4  output()
```

Рисунок 3.26 – Вихідний код програми

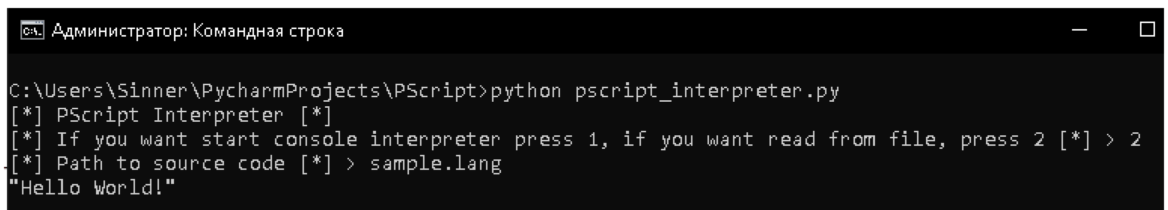

Рисунок 3.27 – Результат виконаного скрипта

ВИСНОВКИ

Таким чином у результаті виконання кваліфікаційної роботи були виконані наступні завдання:

- проведено аналіз програм трансляторів;
- спроектовано скриптову мову та інтерпретатор;
- розроблено інтерпретатор;
- виконано тестування розробленого інтерпретатору;
- оформлено програмну документацію.

Під час виконання кваліфікаційної роботи були розглянуті існуючі скриптові мови програмування, також різниця між компіляторами та інтерпретаторами та основні види інтерпретаторів. Була спроектована скриптова мова. У кваліфікаційній роботі було розроблено три модулі, які потрібні для роботи інтерпретатору. Розроблений програмний додаток має зручний інтерфейс, декілька режимів роботи, він буде зрозумілий кожній людині, яка буде користуватися даним додатком.

ПЕРЕЛІК ПОСИЛАНЬ

1. Abelson H., G. J. Sussman Structure and Interpretation of Computer Programs - 2nd Edition : The Massachusetts Institute of Technology, 1996. 883 ст.
2. Рамальо Л. Python. До вершин майстерності. Москва: ДМК Пресс, 2016. 768с.
3. Ranta A. Implementing Programming Languages: An Introduction to Compilers and Interpreters: Individual author and College Publications , 2012. 226 с.
4. Ахо А. Компиляторы: принципы, технологии и инструменты – 2-е издание / А. Ахо [та ін.]. Київ : Вильямс, 2008. 1186 с.
5. Хантер Р. Проектирование и конструирование компиляторов. Москва : Финансы и статистика, 1984. 232 с
6. Крутіков М.П. Командний інтерпретатор C-shell. Санкт-Петербург : «Пітер», 2007. 231с.

Додаток А

Вихідний код модулю pscript_lexer:

```
from sly import Lexer


class BasicLexer(Lexer):
    tokens = {NAME, NUMBER, STRING, VAR, PRINT, IF, THEN, ELSE, FOR, FUN, TO, ARROW, EQEQ}
    ignore = '\t '

    literals = { '=', '+', '-', '/', '*', '(', ')', ',', ';' }

    # Define tokens
    VAR = r'VAR'
    PRINT = r'PRINT'
    IF = r'IF'
    THEN = r'THEN'
    ELSE = r'ELSE'
    FOR = r'FOR'
    FUN = r'FUN'
    TO = r'TO'
    ARROW = r'!>'
    NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'
    STRING = r'\".*?\"'
    EQEQ = r'=='

    @_(r'\d+')
    def NUMBER(self, t):
```

```
        t.value = int(t.value)
        return t

    @_(r'#.*')
    def COMMENT(self, t):
        pass

    @_(r'\n+')
    def newline(self,t ):
        self.lineno = t.value.count('\n')


if __name__ == '__main__':
    lexer = BasicLexer()
    env = {}
    while True:
        try:
            text = input('[*] shell [*] > ')
        except EOFError:
            break
        if text:
            lex = lexer.tokenize(text)
            for token in lex:
                print(token)
```

Додаток Б

Вихідний код модулю pscript_parser:

```
import pscript_lexer
from sly import Parser



class BasicParser(Parser):
    tokens = pscript_lexer.BasicLexer.tokens

    precedence = (
        ('left', '+', '-'),
        ('left', '*', '/'),
        ('right', 'UMINUS'),
        )

    def __init__(self):
        self.env = { }

    @_('')
    def statement(self, p):
        pass

    @_('FOR var_assign TO expr THEN statement')
    def statement(self, p):
        return ('for_loop', ('for_loop_setup', p.var_assign, p.expr), p.statement)

    @_('IF condition THEN statement ELSE statement')
    def statement(self, p):
```

```
    return ('if_stmt', p.condition, ('branch', p.statement0, p.statement1))

@_('FUN NAME "(" ")" ARROW statement')
def statement(self, p):
    return ('fun_def', p.NAME, p.statement)

@_('NAME "(" ")"')
def statement(self, p):
    return ('fun_call', p.NAME)

@_('expr EQEQ expr')
def condition(self, p):
    return ('condition_eqeq', p.expr0, p.expr1)

@_('var_assign')
def statement(self, p):
    return p.var_assign

@_('VAR NAME "=" expr')
def var_assign(self, p):
    return 'var_assign', p.NAME, p.expr

@_('PRINT NAME')
def statement(self, p):
    return 'ret_var', p.NAME

@_('NAME "=" expr')
def var_assign(self, p):
    return 'var_assign', p.NAME, p.expr
```

```
@_('STRING')
def expr(self, p):
    return 'str', p.STRING

@_('"(" expr ")"')
def expr(self, p):
    return p.expr

@_('expr')
def statement(self, p):
    return p.expr

@_('expr "+" expr')
def expr(self, p):
    return 'add', p.expr0, p.expr1

@_('expr "-" expr')
def expr(self, p):
    return 'sub', p.expr0, p.expr1

@_('expr "*" expr')
def expr(self, p):
    return 'mul', p.expr0, p.expr1

@_('expr "/" expr')
def expr(self, p):
    return 'div', p.expr0, p.expr1

@_('"-" expr %prec UMINUS')
def expr(self, p):
```

```
        return p.expr

    @_('NAME')
    def expr(self, p):
        return ('var', p.NAME)

    @_('NUMBER')
    def expr(self, p):
        return ('num', p.NUMBER)


if __name__ == '__main__':
    lexer = pscript_lexer.BasicLexer()
    parser = BasicParser()
    env = {}

    while True:
        try:
            text = input('[*] shell [*] > ')
        except EOFError:
            break
        if text:
            tree = parser.parse(lexer.tokenize(text))
            print(tree)
```

Додаток В

Вихідний код модулю pscript_interpreter:

```
import pscript_lexer
import pscript_parser


class BasicExecute:

    def __init__(self, tree, env):
        self.env = env
        result = self.walkTree(tree)
        if result is not None and isinstance(result, int):
            print(result)
        if isinstance(result, str) and result[0] == '"':
            print(result)

    def walkTree(self, node):

        if isinstance(node, int):
            return node
        if isinstance(node, str):
            return node

        if node is None:
            return None

        if node[0] == 'program':
            if node[1] is None:
```

```
            self.walkTree(node[2])
        else:
            self.walkTree(node[1])
            self.walkTree(node[2])

    if node[0] == 'num':
        return node[1]

    if node[0] == 'str':
        return node[1]

    if node[0] == 'if_stmt':
        result = self.walkTree(node[1])
        if result:
            return self.walkTree(node[2][1])
        return self.walkTree(node[2][2])

    if node[0] == 'condition_eqeq':
        return self.walkTree(node[1]) == self.walkTree(node[2])

    if node[0] == 'fun_def':
        self.env[node[1]] = node[2]

    if node[0] == 'fun_call':
        try:
            return self.walkTree(self.env[node[1]])
        except LookupError:
            print("Undefined function '%s'" % node[1])
            return 0
```

```
        if node[0] == 'add':
            try:
                if isinstance(self.walkTree(node[1]), str) and
isinstance(self.walkTree(node[1]), str):
                    node1 = self.walkTree(node[1])[:-1]
                    node2 = self.walkTree(node[2])[1:]
                    return node1 + node2
                else:
                    return self.walkTree(node[1]) + self.walkTree(node[2])
            except TypeError:
                print("Type error!")
        elif node[0] == 'sub':
            try:
                return self.walkTree(node[1]) - self.walkTree(node[2])
            except TypeError:
                print("Type error!")
        elif node[0] == 'mul':
            try:
                return self.walkTree(node[1]) * self.walkTree(node[2])
            except TypeError:
                print("Type error!")
        elif node[0] == 'div':
            try:
                return self.walkTree(node[1]) // self.walkTree(node[2])
            except TypeError:
                print("Type error!")

        if node[0] == 'var_assign':
            self.env[node[1]] = self.walkTree(node[2])
            return node[1]
```

```
        if node[0] == 'var' or node[0] == 'ret_var':
            try:
                return self.env[node[1]]
            except LookupError:
                print("Undefined variable '" + node[1] + "' found!")
                return 0

        if node[0] == 'for_loop':
            if node[1][0] == 'for_loop_setup':
                loop_setup = self.walkTree(node[1])

                loop_count = self.env[loop_setup[0]]
                loop_limit = loop_setup[1]

                for i in range(loop_count + 1, loop_limit + 1):
                    res = self.walkTree(node[2])
                    if res is not None:
                        print(res)
                    self.env[loop_setup[0]] = i
                del self.env[loop_setup[0]]

        if node[0] == 'for_loop_setup':
            return (self.walkTree(node[1]), self.walkTree(node[2]))


if __name__ == '__main__':
    lex = pscript_lexer.BasicLexer()
    pars = pscript_parser.BasicParser()
    env = { }
```

```
print("[*] PScript Interpreter [*]")
welcome = input("[*] If you want start console interpreter press 1, if you
want read from file, press 2 [*] > ")

try:
    welcome = int(welcome)
except ValueError:
    print('Bad value!')

if welcome == 1:
    while True:
        try:
            text = input('[*] shell [*] > ')
        except EOFError:
            break
        if text:
            tree = pars.parse(lex.tokenize(text))
            BasicExecute(tree, env)
elif welcome == 2:
    text = open(input('[*] Path to source code [*] > '), 'r')
    Lines = text.readlines()

    for line in Lines:
        tree = pars.parse(lex.tokenize(line))
        BasicExecute(tree, env)
```