МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КПІ»

Кафедра

автоматизованих систем обробки інформації та управління

**КУРСОВА РОБОТА**

з дисципліни

“ Об'єктно-орієнтоване програмування ”

на тему

"Інтерпретатор мови програмування"

|  |  |  |
| --- | --- | --- |
| Керівник : доц. Муха І.П |  | Виконали : Шелудько  Дмитро Максимович, Бугайов Олексій Олександрович, Ситник Михайло |
| Допущений до захисту |  | студенти гр. ІП-63,ФІОТ |
| І\_\_\_І \_\_\_\_\_\_\_\_ 201\_ \_\_\_\_\_\_  підпис |  | 2 курс  № ІП-63 |
| Захистив з оцінкою  \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_  оцінка підпис  І\_\_\_І \_\_\_\_\_\_\_\_\_ 201\_ |  | Підпис: |

Київ 2017

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ “КПІ”**

(назва вищого навчального закладу)

Кафедра автоматизованих систем обробки інформації і управління

Дисципліна Об’єктно-орієнтоване програмування

Напрям "Програмна інженерія"

Курс 2 Група ІП-63 Семестр 3

ЗАВДАННЯ

на курсову роботу студента

(прізвище, ім’я, по батькові)

1. Тема роботи

2. Строк здачі студентом закінченої роботи

3. Вихідні дані до роботи

4. Зміст розрахунково-пояснювальної записки (перелік питань, які підлягають розробці)

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

6. Дата видачі завдання

КАЛЕНДАРНИЙ ПЛАН

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Назва етапів курсової роботи | Термін виконання етапів роботи | Підписи керівника, студента |
| 1. | Отримання теми курсової роботи |  |  |
| 2. | Підготовка ТЗ |  |  |
| 3. | Пошук та вивчення літератури з питань курсової роботи |  |  |
| 4. | Розробка сценарію роботи програми |  |  |
| 6. | Узгодження сценарію роботи програми з керівником |  |  |
| 5. | Розробка (вибір) алгоритму рішення задачі |  |  |
| 6. | Узгодження алгоритму з керівником |  |  |
| 7. | Узгодження з керівником інтерфейсу користувача |  |  |
| 8. | Розробка програмного забезпечення |  |  |
| 9. | Налагодження розрахункової частини програми |  |  |
| 10. | Розробка та налагодження інтерфейсної частини програми |  |  |
| 11. | Узгодження з керівником набору тестів для контрольного прикладу |  |  |
| 12. | Тестування програми |  |  |
| 13. | Підготовка пояснювальної записки |  |  |
| 14. | Здача курсової роботи на перевірку |  |  |
| 15. | Захист курсової роботи |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Студент

(підпис)

Керівник Муха І. П.

(підпис) (прізвище, ім’я, по батькові)

"\_\_\_" \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_ р.

Анотація

Пояснювальна записка до курсової роботи: 55 сторінок, 24 рисунки, 28 таблиць, 4 посилання.

Об’єкт дослідження: написання інтерпретатора мови програмування

Мета роботи: дослідження способів написання інтерпретаторів мов програмування, створення програмного забезпечення для інтерпретації коду.

Вивчено методи написання усіх складових частин інтерпретатора, метод рекурсивного спуску. Приведені змістовні постановки задач, описано детальний процес розв’язання кожної з них.

Виконана програмна реалізація алгоритму рекурсивного спуску написання синтаксичних аналізаторів.

ІНТЕРПРЕТАТОР МОВИ ПРОГРАМУВАННЯ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Кафедра автоматизованих систем обробки інформації та управління

|  |  |  |
| --- | --- | --- |
| **РОЗРОБИВ** |  | **ЗАТВЕРДИВ** |
| **Студент**  \_\_\_\_\_\_\_\_\_\_ *М. О. Златокрилець*  (підпис) (ініціали, прізвище)  “10” листопада 2016 р. |  | **Керівник**  \_\_\_\_\_\_\_\_\_\_ *І.П.Муха*  (підпис) (ініціали, прізвище)  “11” листопада 2016 р. |

Інтерпретатор мови програмування

**ТЕХНІЧНЕ ЗАВДАННЯ**

на виконання курсової роботи

з дисципліни:

«Об'єктно-орієнтоване програмування»

Шифр КПІ ІП-6109.ТЗ

на 14 сторінках

Київ 2017

ЗМІСТ

[1 найменування та область застосування 8](#_Toc501994711)

[2 підстави для розробки 9](#_Toc501994712)

[2.1 Перелік документів, на підставі яких ведеться розробка 9](#_Toc501994713)

[2.2 Найменування і умовне позначення теми розробки 9](#_Toc501994714)

[3 призначення розробки 10](#_Toc501994715)

[3.1 Функціональне призначення розробки 10](#_Toc501994716)

[3.2 Експлуатаційне призначення 10](#_Toc501994717)

[4 вимоги до програмного забезпечення 11](#_Toc501994718)

[4.1 Вимоги до функціональних характеристик 11](#_Toc501994719)

[4.1.1 Вимоги до складу виконуваних функцій 11](#_Toc501994720)

[4.1.2 Вимоги до організації вхідних даних 11](#_Toc501994721)

[4.1.3 Вимоги до організації вихідних даних 11](#_Toc501994722)

[4.2 Вимоги до надійності 11](#_Toc501994723)

[4.2.1 Вимоги до надійного функціонування програми 11](#_Toc501994724)

[4.2.2 Час відновлення після відмови 11](#_Toc501994725)

[4.2.3 Відмови через некоректні дії користувача 12](#_Toc501994726)

[4.3 Умови експлуатації 12](#_Toc501994727)

[4.3.1 Кліматичні умови експлуатації 12](#_Toc501994728)

[4.3.2 Вимоги до видів обслуговування 12](#_Toc501994729)

[4.3.3 Вимоги до кількості та кваліфікації персоналу 12](#_Toc501994730)

[4.4 Вимоги до складу і параметрів технічних засобів 12](#_Toc501994731)

[4.5 Вимоги до інформаційної і програмної сумісності 13](#_Toc501994732)

[4.6 Вимоги до маркування та упаковки 13](#_Toc501994733)

[4.7 Вимоги до транспортування та зберігання 13](#_Toc501994734)

[5 вимоги до програмної документації 14](#_Toc501994735)

[5.1 Попередній склад програмної документації 14](#_Toc501994736)

[6 техніко-економічні показники 15](#_Toc501994737)

[7 стадії та етапи розробки 16](#_Toc501994738)

[7.1 Стадії розробки 16](#_Toc501994739)

[7.2 Етапи розробки 16](#_Toc501994740)

[7.3 Зміст роботи по етапам 16](#_Toc501994741)

[8 порядок контролю і прийому 18](#_Toc501994742)

[8.1 Види досліджень 18](#_Toc501994743)

[8.2 Загальні вимоги до прийняття роботи 18](#_Toc501994744)

1. найменування та область застосування

Найменування – «Інтерпретатор мови програмування».

Програма призначена до застосування на кафедрі Автоматизованих систем обробки інформації та управління Національного технічного університету України «Київського політехнічного інституту».

1. підстави для розробки
   1. Перелік документів, на підставі яких ведеться розробка

Підставою для проведення розробки являється наказ № 6 від 11 травня 2016 року. Наказ затверджено на засіданні кафедри АСОІУ, далі Замовник і прийнято до виконання студентом(ми) Златокрильцем Миколою Олександровичем, далі Виконавець(ці), дата затвердження ТЗ.

* 1. Найменування і умовне позначення теми розробки

Найменування теми розробки – «Розробка інтерпретатора мови програмування».

Умовне позначення теми розробки (шифр теми) – «РІМП 0.1».

1. призначення розробки
   1. Функціональне призначення розробки

Функціональним призначенням розробки програми є надання користувачам можливості написання та виконання коду на розробленій в ході даної роботи мові програмування.

* 1. Експлуатаційне призначення

Програма повинна експлуатуватися в об'єктах кафедри АСОІУ.

Кінцевими користувачами програми є співробітники та студенти кафедри АСОІУ, абітурієнти та випускники кафедри.

1. вимоги до програмного забезпечення
   1. Вимоги до функціональних характеристик
      1. Вимоги до складу виконуваних функцій

Програма повинна забезпечувати можливість виконання нижче наведених функцій:

1. інтерпретація коду, розміщеного у текстовому файлі;
2. інтерпретація коду пооператорно в режимі read-eval-print loop (далі - REPL);
3. в разі лексичної або синтаксичної помилки – виведення відповідного повідомлення
   * 1. Вимоги до організації вхідних даних

Якщо програма працює в режимі інтерпретації файлу з кодом, вхідні дані до програми повинні бути організовані у вигляді текстового файлу з кодом. Якщо програма працює в режимі REPL, вхідні дані до програми повинні надаватися регулярно у вигляді операторів мови програмування, що інтерпретується.

* + 1. Вимоги до організації вихідних даних

Вихідними даними для програми є результат виконання коду програми, якщо він не має лексичних та синтаксичних помилок. Інакше вихідними даними для програми є повідомлення про помилку, яке містить інформацію про тип помилки (лексична або синтаксична).

* 1. Вимоги до надійності
     1. Вимоги до надійного функціонування програми

Надійне функціонування програми повинно бути забезпечене виконанням замовником сукупності організаційно-технічних заходів, перелік яких наведено нижче:

1. організацією безперебійного струмопостачання технічних засобів;
2. використанням ліцензійного програмного забезпечення;
3. використанням своєчасно оновленого програмного забезпечення;
   * 1. Час відновлення після відмови

Час відновлення після відмови, що спричинена збоєм електропостачання технічних засобів, не крахом операційної системи, не повинно перевищувати 1 години за умови виконання умов експлуатації технічних і програмних засобів.

Час відновлення після відмови, що спричинена неполадкою технічних засобів, крахом операційної системи, не повинно перевищувати часу, який потрібен на ліквідацію неполадок технічних засобів та переустановленням програмних засобів.

* + 1. Відмови через некоректні дії користувача

Відмови програми можливі внаслідок некоректних дій користувача системи. Задля запобігання відмов програми через вище вказані причини слід обмежити коло користувачів з правами доступу «Адміністратор».

* 1. Умови експлуатації
     1. Кліматичні умови експлуатації

Кліматичні умови експлуатації, при яких повинні забезпечуватися задані характеристики, повинні задовольняти вимогам, що заявлені до технічних засобів зокрема до умов їх експлуатації.

* + 1. Вимоги до видів обслуговування

Див. Вимоги до забезпечення надійного функціонування програми.

Програма не потребує будь-яких видів обслуговування.

* + 1. Вимоги до кількості та кваліфікації персоналу

Мінімальна кількість персоналу, що необхідний для роботи програми, повинна складати не менше 1 штатної одиниці – користувач системи з правами «Адміністратор».

Користувач програми з правами «Адміністратор» повинен мати навички роботи з командним рядком операційної системи, мінімальний досвід програмування та повинен бути ознайомленим із правилами та синтаксичними конструкціями мови програмування, що інтерпретується.

Персонал повинен бути атестованим на ІІ кваліфікаційну групу по електробезпеці (для роботи з кафедральним обладнанням).

* 1. Вимоги до складу і параметрів технічних засобів

До комп’ютера, на якому має бути розгорнуто програмне забезпечення, висуваються наступні вимоги:

1. 64-розрядний процесор з тактовою частотою не нижче 1.6 ГГц;
2. достатній об’єм оперативної пам’яті (не менше 2 ГБ);
3. достатній об’єм жорсткого диску (не менше 5 ГБ).
   1. Вимоги до інформаційної і програмної сумісності

Вимоги до додаткового ПЗ не висуваються

* 1. Вимоги до маркування та упаковки

Вимоги до маркування та упаковки не висуваються

* 1. Вимоги до транспортування та зберігання

Вимоги до транспортування та зберігання не висуваються

1. вимоги до програмної документації
   1. Попередній склад програмної документації

Склад програмної документації повинен включати в себе:

1. технічне завдання;
2. керівництво користувача.
3. техніко-економічні показники

Орієнтовна економічна ефективність не обчислюється.

1. стадії та етапи розробки
   1. Стадії розробки

Розробка повинна бути проведена в три стадії:

1. розробка технічного завдання;
2. робоче проектування;
3. впровадження.
   1. Етапи розробки

На стадії розробки технічного завдання повинен бути виконаний етап розробки, узгодження і затвердження технічного завдання.

На стадії робочого проектування повинен бути виконаний наведений нижче перелік робіт:

1. розробка програми;
2. тестування програми;
3. дослідження програми;
4. розробка програмної документації.

На стадії впровадження виконується передача програми кінцевому замовнику.

* 1. Зміст роботи по етапам

На етапі розробки технічного завдання повинні бути виконані наступні роботи:

1. постановка задачі;
2. визначення і уточнення вимог до технічних засобів;
3. визначення вимог до програми;
4. визначення стадій, етапів і строків розробки програми та документації на неї;
5. вибір мов програмування;
6. узгодження та затвердження технічного завдання.

На етапі розробки програми повинні бути виконані роботи по програмуванню і налагодженню програми.

На етапі тестування повинні бути виконані роботи по тестуванню усіх основних синтаксичних конструкцій мови програмування, що інтерпретується, в обох режимах роботи програми: режимі інтерпретації файлу та режимі REPL.

На етапі досліджень програми повинні бути виконані наведені нижче види робіт:

1. розробка, узгодження і затвердження програми і методик дослідження;
2. проведення досліджень;
3. коригування програми і програмної документації по результатам досліджень.

На етапі розробки програмної документації повинна бути виконана розробка програмних документів.

На етапі підготовки та передачі програми повинна бути виконана робота по підготовці та передачі програми і програмної документації в експлуатацію на об’єктах кафедри АСОІУ.

1. порядок контролю і прийому
   1. Види досліджень

Приймально-здавальні випробування програмного продукту мають проводитися згідно з розробленою виконавцем і узгодженою із замовником “Програмою та методикою випробувань” на об’єкті замовника.

* 1. Загальні вимоги до прийняття роботи

На основі проведених досліджень Виконавець спільно з Замовником підписують Акт прийому-здачі програми в експлуатацію.

**Пояснювальна записка  
до курсової роботи**

|  |  |
| --- | --- |
| на тему: |  |
|  |  |
|  | |

Київ 2016

зміст

[1 постановка задачі 23](#_Toc501994749)

[2 викладення використовуваних методів 24](#_Toc501994750)

[2.1 Основні поняття 24](#_Toc501994751)

[2.2 Лексичний аналіз 24](#_Toc501994752)

[2.3 Синтаксичний аналіз 25](#_Toc501994753)

[2.4 Контроль типів і генерація коду 26](#_Toc501994754)

[3 опис алгоритму 27](#_Toc501994755)

[3.1 Загальний алгоритм 27](#_Toc501994756)

[3.2 Алгоритм лексичного аналізу 28](#_Toc501994757)

[3.3 Алгоритм синтаксичного аналізу 29](#_Toc501994758)

[3.4 Алгоритм контролю типів 29](#_Toc501994759)

[4 опис програмного забезпечення 31](#_Toc501994760)

[4.1 Опис діаграми класів програмного забезпечення 31](#_Toc501994761)

[4.2 Опис класів та їх методів 32](#_Toc501994762)

[5 тестування програмного забезпечення 52](#_Toc501994763)

[5.1 План тестування 52](#_Toc501994764)

[5.2 Приклади тестування 52](#_Toc501994765)

[5.2.1 Тестування при введенні некоректного шляху до файлу з кодом 52](#_Toc501994766)

[5.2.2 Тестування при наявності в коді лексичних та синтаксичних помилок 53](#_Toc501994767)

[5.2.3 Тестування при наявності в коді недозволених операцій 54](#_Toc501994768)

[5.2.4 Тестування коректності роботи арифметичних та логічних операцій 56](#_Toc501994769)

[6 інструкція користувача 58](#_Toc501994770)

[6.1 Призначення програми 58](#_Toc501994771)

[6.2 Вимоги до системи 58](#_Toc501994772)

[6.3 Інструкція по користуванню програмою 58](#_Toc501994773)

[Додаток А Тексти програмного коду 62](#_Toc501994774)

[Додаток Б граматика розробленої мови програмування 113](#_Toc501994775)

вступ

Дана робота присвячена розробці програмного забезпечення для інтерпретації коду, написаного на розробленій мові програмування (граматику розробленої мови у формі Бекуса-Наура надано у додатку Б).

Призначенням роботи є надання можливості написання коду на розробленій мові програмування, що дозволяє швидше виконувати операції з векторами та матрицями. Метою даної роботи є набуття навичок у розробці системного програмного забезпечення, зокрема – інтерпретаторів.

Інтерпретатор повинен працювати у двох режимах: режимі інтерпретування цілого файлу з кодом і режимі інтерактивної інтерпретації read-eval-print loop (далі - REPL). Режим роботи вибирається користувачем самостійно. У випадку вибору першого режиму, користувач передає програмі шлях до файлу з кодом, який необхідно виконати. Якщо ж вибрано режим REPL, код вводиться у програму пооператорно та одразу виконується. Робота з програмою продовжується доти, доки користувач не забажає її завершити.

У випадку, коли код написано безпомилково, програма його виконає. Якщо у коді наявні лексичні або синтаксичні помилки, після першої ж помилки програма завершить свою роботу і видасть при цьому повідомлення, що містить інформацію про тип помилки (лексична або синтаксична).

Програмне забезпечення може використовуватися студентами технічних спеціальностей, а також інженерами у їх професійній діяльності.

# постановка задачі

Розробити програмне забезпечення для інтерпретації коду програми, написаної на розробленій мові програмування. Реалізувати два режими роботи програми: режим інтерпретування цілого файлу з кодом і режим REPL.

У випадку, коли вибрано режим інтерпретування файлу, вхідними даними до програми є шлях до файлу з кодом, написаному на розробленій мові програмування. У випадку, коли вибрано режим REPL, вхідними даними до програми є оператори мови програмування, які вводяться до програми один за одним.

Вихідними даними для програми є результат виконання переданого коду, у випадку, коли він не містить лексичних та синтаксичних помилок. Якщо ж код, переданий по програми, містить лексичні та/або синтаксичні помилки, вихідними даними для програми є результат виконання коду, який передує першій помилці, а після нього – інформація про помилку.

Перший режим роботи програми (режим інтерпретування файлу) повинен бути реалізованим наступним чином: програма повинна запускатися із командного рядка системи, приймаючи першим і єдиним параметром шлях до файлу з кодом.

Другий режим роботи програми (режим REPL) повинен бути реалізованим наступним чином. Програма повинна запускатися із командного рядка системи, не приймаючи жодного параметра. Після запуску, програма повинна видати повідомлення, що містить мінімальну інструкцію користувача і пропозицію ввести перший оператор коду. Після того, як введено черговий оператор, програма повинна проінтерпретувати та виконати його, після чого вивести результати виконання оператора, якщо вони присутні, і повідомлення, яке свідчить про готовність до прийому наступного оператора. У випадку, коли оператор містить лексичну або синтаксичну помилку, програма повинна видати відповідне повідомлення і пропозицію ввести наступний оператор програми. Після введення даних, які сигналізують про бажання завершити роботу з програмою, програма повинна завершити свою роботу.

# викладення використовуваних методів

## Основні поняття

Для того, щоб мати можливість писати програми власною мовою програмування, слід мати програмне забезпечення, яке б виконувало переклад програм із розробленої мови на одну із існуючих. Таке програмне забезпечення називають трансляторами.

Транслятор – це програма, яка перекладає програму з початкової мови на еквівалентну їй програму на кінцевій мові. Цей процес перекладу називають трансляцією програми. Розрізняють два варіанти трансляції: компіляція та інтерпретація. Компіляція – це автоматичний переклад всього тексту програми в машинний код і формування об’єктного модуля, який згодом може бути виконаний самостійно. Паралельно здійснюється синтаксичний контроль програми в цілому. Компіляція коду здійснюється програмою, яку називають компілятором. На процесі інтерпретації коду зупинимося детальніше.

Інтерпретація коду програми являє собою пооператорний переклад і виконання команд програми до першої помилки. Інтерпретація коду здійснюється програмою, яку називають інтерпретатором. Інтерпретатор отримує код, написаний на вхідній мові, аналізує його і потім виконує.

Процес інтерпретації коду складається з чотирьох етапів: лексичний аналіз, синтаксичний аналіз, контроль типів і генерація коду. Розглянемо детальніше ці етапи.

## Лексичний аналіз

Лексичний аналіз – це процес перетворення вхідного тексту програми, який складається з послідовності символів, на послідовність токенів. Лексичний аналіз виконується частиною інтерпретатора, що має назву лексичного аналізатора.

Токен – це шаблон, що визначає тип групи лексем. Існує два види токенів: ті, що не містять додаткової інформації (наприклад, знаки операцій, такі як «+», «-» тощо, ключові слова програми та інші) і ті, що потребують додаткової інформації (наприклад, токен «число» повинен містити інформацію про значення числа). Також існує токен, що позначає кінець програми. Він поміщається у кінець отриманої послідовності токенів після того, як лексичний аналіз програми завершено.

Вхідними даними для лексичного аналізатора є послідовність символів коду вхідної програми. Лексичний аналізатор посимвольно аналізує цю послідовність і виділяє в ній токени. Пробіли, символи переходу на новий рядок, повернення каретки та табуляції ігноруються. У випадку, коли токен сформувати неможливо (наприклад, ім’я змінної починається з цифри або ж незакриті лапки), робиться висновок про те, що вхідний код програми містить лексичну помилку.

На виході лексичного аналізатора, у випадку коли код програми не містить лексичних помилок, формується масив токенів, які було виділено із вхідної послідовності символів. Далі цей масив токенів подається на вхід синтаксичному аналізатору.

## Синтаксичний аналіз

Основним етапом інтерпретації програми є синтаксичний аналіз. Синтаксичний аналіз – це процес аналізу отриманих токенів з метою розбору граматичної структури тексту програми у відповідністю із заданою формальною граматикою. Синтаксичний аналіз виконується частиною інтерпретатора, що має назву синтаксичного аналізатора.

До основних функцій синтаксичного аналізатора відносять:

1. знаходження і виділення синтаксичних конструкцій у вхідному масиві токенів;
2. встановлення типу і перевірка правильності кожної синтаксичної конструкції;
3. представлення синтаксичних конструкцій у вигляді, зручному для подальшого їх виконання.

Вхідними даними для синтаксичного аналізатора є послідовність токенів, які було виокремлено з коду програми на етапі лексичного аналізу.

Синтаксичний аналізатор працює наступним чином. Із вхідної послідовності токенів синтаксичний аналізатор відбирає стільки, скільки необхідно для утворення логічно зв’язаного фрагмента. Це реалізується за допомогою алгоритму рекурсивного спуску, який буде подано у розділі 3. Якщо синтаксичну конструкцію розпізнано, синтаксичний аналізатор формує синтаксичне дерево оператора. Якщо ж структуру не розпізнано, тобто вона не підпала під жодне правило граматики, робиться висновок, що код містить синтаксичну помилку. До синтаксичних помилок відносять, наприклад, відсутність крапки з комою в кінці оператора, відсутність другого операнду після знаку бінарної операції «+» тощо.

Синтаксичне дерево оператора – це структура, яка дозволяє інтерпретатору розібратися в логіці програми. Це дерево містить у своїх вузлах, що не є листям, нетермінали граматики. Листя дерева містить термінали. Термінали – це мінімальні елементи граматики, що не мають власної граматичної структури (наприклад, число, ключове слово тощо). Нетермінали – це елементи граматики, що мають власні імена та структуру і складаються з терміналів і/або нетерміналів, поєднання яких визначається правилами граматики.

Вихідними даними для синтаксичного аналізатора є висновок про те, чи містить код програми синтаксичні помилки, і у випадку, якщо вони відсутні, - синтаксичне дерево оператора.

## Контроль типів і генерація коду

Після того, як було проведено синтаксичний аналіз коду і побудовано синтаксичне дерево, відбувається наступний етап інтерпретації програми – контроль типів, що являє собою перевірку узгодженості згрупованих лексем. На цьому етапі інтерпретатор перевіряє зміст як окремих лексем, так і їх груп. У випадку, якщо в якомусь місці програми присутня невідповідність типів (наприклад, ділення матриці на матрицю), видається повідомлення про синтаксичну помилку.

Після того, як проведено контроль типів, відбувається процес генерації коду, що являє собою формування команд на кінцевій мові програмування (у даній курсовій роботі – С++) з подальшим їх виконанням. Механізм реалізації генерації коду наступний: інтерпретатор виконує обхід синтаксичного дерева оператора, витягує з нього лексеми і виконує потрібну операцію.

# опис алгоритму

## Загальний алгоритм

Розглянемо загальний алгоритм виконання інтерпретації коду.

1. ПОЧАТОК
2. ЯКЩО вибрано режим зчитування з файлу, ТО
   1. Зчитування коду з файлу в рядок
   2. ЯКЩО помилка при зчитування, ТО
      1. Виведення повідомлення про помилку зчитування
      2. Перехід до п. 8
3. ІНАКШЕ
   1. Виведення повідомлення з короткою інструкцією та пропозицією ввести перший оператор програми
   2. ЯКЩО зчитано команду завершення роботи з інтерпретатором (finish), ТО перехід до п. 9
   3. Зчитування введеного оператора і занесення його в рядок
4. Лексичний аналіз рядка з кодом
   1. ЯКЩО лексична помилка, ТО
      1. Виведення повідомлення про лексичну помилку
      2. Перехід до п. 8
5. Синтаксичний аналіз коду
   1. ЯКЩО синтаксична помилка, ТО
      1. Виведення повідомлення про синтаксичну помилку
      2. Перехід до п. 8
6. Контроль типів
   1. ЯКЩО помилка типів, ТО
      1. Виведення повідомлення про синтаксичну помилку
      2. Перехід до п. 8
7. Виконання коду
8. ЯКЩО було обрано режим REPL, ТО
   1. Перехід до п. 3
9. КІНЕЦЬ

## Алгоритм лексичного аналізу

1. ПОЧАТОК
2. Зчитування токена
   1. Зчитування символа
   2. ЯКЩО зчитано символ закінчення рядка, ТО
      1. Формування токену кінця програми
      2. Перехід до п. 3
   3. ЯКЩО зчитано один із символів: « », «\n», «\t», ТО
      1. Перехід до п. 2.1
   4. ЯКЩО зчитано лапки, ТО
      1. Зчитування символів доти, доки не будуть зчитані лапки
      2. ЯКЩО зчитано символ закінчення рядка, ТО
         1. Перехід до п. 2.9
      3. Формування токену «текст»
      4. Перехід до п. 3
   5. ЯКЩО зчитано односимвольний токен, ТО
      1. Формування відповідного токену
      2. Перехід до п. 3
   6. Сформувати рядок, додати туди символ
   7. ПОКИ символ є цифрою, літерою, символами «\_» або «.»
      1. Зчитати символ та додати його в рядок
   8. ЯКЩО отриманий рядок є токеном, ТО
      1. Формування відповідного токену
      2. Перехід до п. 3
   9. Формування токену «помилка»
3. ЯКЩО отримано токен кінця програми, ТО
   1. Закінчення лексичного аналізу. Повернення масиву токенів
4. ЯКЩО отримано токен «помилка», ТО
   1. Закінчення лексичного аналізу. Повернення пустого масиву
5. Занесення токену в масив
6. Перехід до п. 2

## Алгоритм синтаксичного аналізу

За основу алгоритму синтаксичного аналізу взято алгоритм рекурсивного спуску, який наведено нижче.

1. ПОЧАТОК
2. Зчитування токена із масиву
3. ДЛЯ КОЖНОГО правила граматики, що задає оператор, ВИКОНАТИ
   1. Запустити метод перевірки, чи відповідає конструкція даному правилу граматики
      1. Виклик методів, які відповідають правилам граматики щодо даного оператора. Якщо один із методів зустрів неочікуваний токен, повернення помилки. Якщо ж усі методи зчитали очікувані токени, то сформувати оператор.
         1. ЯКЩО один із методів зустрів неочікуваний токен, ТО повернути помилку
         2. Після безпомилкового виклику методу додати вузол у синтаксичне дерево
      2. ЯКЩО оператор сформовано, ТО
         1. Перехід до п. 5
4. ЯКЩО конструкція не відповідає жодному правилу граматики, ТО
   1. Завершення синтаксичного аналізу. Виведення повідомлення про синтаксичну помилку.
5. ЯКЩО не було зчитано токен кінця програми, ТО
   1. Перехід до п. 2
6. КІНЕЦЬ

## Алгоритм контролю типів

1. ПОЧАТОК
2. Отримання дерева синтаксичного розбору оператора
3. ДЛЯ КОЖНОГО вузла дерева ВИКОНАТИ
   1. ЯКЩО тип токена не відповідає типу, який може знаходитися у даному місці дерева за правилом граматики даного оператора, ТО
      1. Завершення контролю типів. Виведення повідомлення про синтаксичну помилку
4. КІНЕЦЬ

# опис програмного забезпечення

## Опис діаграми класів програмного забезпечення

Програмний додаток містить головний файл, 10 заголовних файлів та 7 файлів реалізації. У програмі міститься 26 класів, із яких 19 – членів ієрархії класів правил граматики для побудови синтаксичного дерева.

Для зручності діаграму класів розбито на три частини. Першу частину діаграми, на якій містяться робочі класи, зображено на рисунку 4.1. Другу частину діаграми, на якій міститься ієрархія класів операторів мови програмування, зображено на рисунку 4.2. Третю частину діаграми, на якій міститься ієрархія класів виразів мови програмування, зображено на рисунку 4.3.

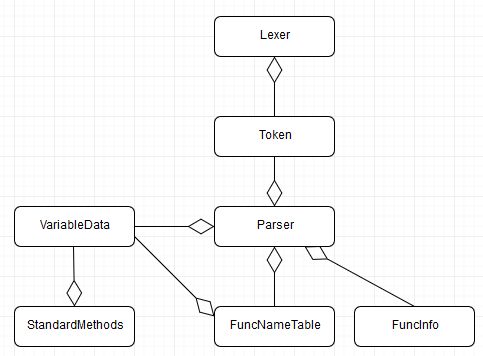


Рисунок 4.1 – UML-діаграма робочих класів

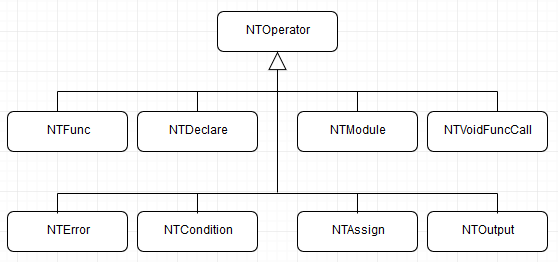


Рисунок 4.2 - UML-діаграма класів операторів мови програмування

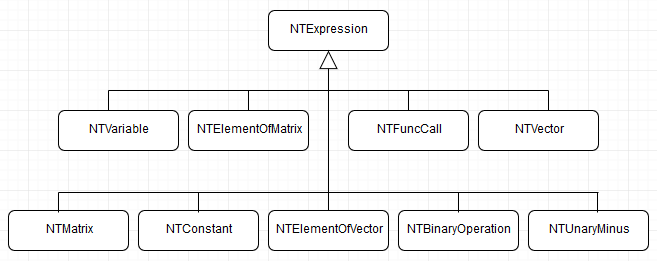


Рисунок 4.3 - UML-діаграма класів виразів мови програмування

## Опис класів та їх методів

Опис користувацьких класів програмного забезпечення та їх методів наведено у таблиці 4.1.

Опис стандартних класів та їх методів наведено у таблиці 4.2.

Таблиця 4.1 – Опис користувацьких класів та методів

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 1 | Lexer | setInput | Задання коду, лексичний аналіз якого необхідно виконати | String input – рядок, який містить код | Функція не повертає результату | Lexer.h |
| 2 | Lexer | convert | Виконання лексичного аналізу. Формування масиву токенів | Відсутні | Повертає vector<Token> - масив токенів, виокремлених з тексту коду. В разі лексичної помилки повертається пустий масив | Lexer.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 3 | Lexer | readToken | Зчитування наступного токена | Відсутні | Повертає Token – наступний зчитаний токен | Lexer.h |
| 4 | Lexer | readSymbol | Зчитування наступного символа | Відсутні | Відсутні | Lexer.h |
| 5 | Lexer | identifyToken | Визначення, чи є послідовність символів токено. Якщо так – визначення типу токена | Token &token – токен, який буде отримано в результаті впізнання  string value – послідовність символів, в якій необхідно впізнати токен | Повертає значення bool. True, якщо токен впізнано і false – якщо ні. | Lexer.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 6 | Lexer | isNumber | Визначення, чи є послідовність символів дійсним числом | string line – послідовність символів, в якій треба впізнати число | Повертає значення bool. True, якщо число впізнано і false – якщо ні. | Lexer.h |
| 7 | Lexer | isDigit | Визначення, чи є символ цифрою | char symbol – Символ, в якому треба впізнати цифру | Повертає значення bool. True, якщо символ є цифрою і false – якщо ні. | Lexer.h |
| 8 | Lexer | isName | Визначення, чи є послідовність символів іменем | string line – послідовність символів, в якій треба впізнати ім’я | Повертає значення bool. True, якщо рядок є іменем і false – якщо ні. | Lexer.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 9 | Lexer | isLetter | Визначення, чи є символ літерою | char symbol – Символ, в якому треба впізнати літеру | Повертає значення bool. True, якщо символ є літерою і false – якщо ні. | Lexer.h |
| 10 | Token | setType | Встановлення типу токена | Type type – новий тип токена | Функція не повертає результату | Token.h |
| 11 | Token | setValue | Встановлення значення токена | string value – нове значення токена | Функція не повертає результату | Token.h |
| 12 | Token | getType | Повертання типу токена | Відсутні | Повертає значення Type – тип токена | Token.h |
| 13 | Token | getValue | Повертання значення токена | Відсутні | Повертає значення string – значення токена | Token.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 14 | Parser | setTokens | Встановлення масиву токенів для аналізу | vector<Token> tokens – новий масив токенів | Функція не повертає результату | Parser.h |
| 15 | Parser | addTokens | Доповнення масиву токенів новим масивом | vector<Token> tokens – новий масив токенів | Функція не повертає результату | Parser.h |
| 16 | Parser | analize | Виконання синтаксичного аналізу коду | Відсутні | Функція не повертає результату | Parser.h |
| 17 | Parser | addToTable | Додання нової змінної в таблицю імен | string name – ім’я змінної, VariableData \*data – значення змінної.  bool isParam – чи є змінна параметром | Повертає значення типу bool. True – якщо операція виконана, false – якщо змінна вже існує | Parser.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 18 | Parser | changeValueByKee | Зміна значення змінної | string name – ім’я змінної. VariableData \*newData – нове значення змінної. | Повертає значення типу bool. True – якщо операція виконана, false – якщо помилка | Parser.h |
| 19 | Parser | getValueByKee | Отримання значення змінної з таблиці імен | string name – ім’я змінної | Повертає значення типу VariableData – значення змінної. Якщо змінна відсутня в таблиці, повертається nullptr | Parser.h |
| 20 | Parser | ruleBlock | Перевірка, чи є наступна послідовність токенів блоком | vector<NTOperator\*> &operators – масив оператов блоку, який необхідно заповнити. | Повертає значення типу bool. True – якщо послідовність токенів є блоком, false – якщо ні | Parser.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 21 | Parser | ruleOperator | Виокремлення оператора з наступної послідовності токенів | Відсутні | Повертає значення типу NTOperator\* - виокремлений оператор. Якщо послідовність токенів не є оператором, повертається nullptr | Parser.h |
| 22 | Parser | ruleDeclare | Виокремлення оператора оголошення змінної з наступної послідовності токенів | Відсутні | Повертає значення типу NTOperator\* - виокремлений оператор. Якщо послідовність токенів не є оператором оголощення, повертається nullptr | Parser.h |
| 23 | Parser | ruleAssign | Виокремлення оператора присвоєння змінної з наступної послідовності токенів | Відсутні | Повертає значення типу NTOperator\* - виокремлений оператор. Якщо послідовність токенів не є оператором присвоєння, повертається nullptr | Parser.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 24 | Parser | ruleExpression | Виокремлення виразу з наступної послідовності токенів | Відсутні | Повертає значення типу NTExpression\* - виокремлений вираз. Якщо послідовність токенів не є виразом, повертається nullptr | Parser.h |
| 25 | Parser | ruleOutput | Виокремлення оператора виведення з наступної послідовності токенів | Відсутні | Повертає значення типу NTOperator\* - виокремлений оператор. Якщо послідовність токенів не є оператором виведення, повертається nullptr | Parser.h |
| 26 | Parser | ruleCondition | Виокремлення умовного оператора з наступної послідовності токенів | Відсутні | Повертає значення типу NTOperator\* - виокремлений оператор. Якщо послідовність токенів не є умовним оператором, повертається nullptr | Parser.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 27 | Parser | ruleFunc | Виокремлення оператора оголошення функції з наступної послідовності токенів | Відсутні | Повертає значення типу NTOperator\* - виокремлений оператор. Якщо послідовність токенів не є оператором оголошення функції, повертається nullptr | Parser.h |
| 28 | Parser | ruleModule | Виокремлення оператора підключення модуля з наступної послідовності токенів | Відсутні | Повертає значення типу NTOperator\* - виокремлений оператор. Якщо послідовність токенів не є оператором підключення модуля, повертається nullptr | Parser.h |
| 29 | Parser | ruleVoidFuncCall | Виокремлення оператора виклику функції з наступної послідовності токенів | Відсутні | Повертає значення типу NTOperator\* - виокремлений оператор. Якщо послідовність токенів не є оператором виклику функції, повертається nullptr | Parser.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 30 | NTOperator | execute | Виконання оператора | Відсутні | Повертає значення типу bool. True – якщо оператор виконано безпомилково, false – якщо ні | SyntaxTree.h |
| 31 | NTDeclare | execute | Виконання оператора оголошення змінної | Відсутні | Повертає значення типу bool. True – якщо оператор виконано безпомилково, false – якщо ні | SyntaxTree.h |
| 32 | NTAssign | execute | Виконання оператора присвоєння | Відсутні | Повертає значення типу bool. True – якщо оператор виконано безпомилково, false – якщо ні | SyntaxTree.h |
| 33 | NTOutput | execute | Виконання оператора виведення | Відсутні | Повертає значення типу bool. True – якщо оператор виконано безпомилково, false – якщо ні | SyntaxTree.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 34 | NTCondition | execute | Виконання умовного оператора | Відсутні | Повертає значення типу bool. True – якщо оператор виконано безпомилково, false – якщо ні | SyntaxTree.h |
| 35 | NTFunc | execute | Виконання оператора оголошення функції | Відсутні | Повертає значення типу bool. True – якщо оператор виконано безпомилково, false – якщо ні | SyntaxTree.h |
| 36 | NTModule | execute | Виконання оператора підключення модуля | Відсутні | Повертає значення типу bool. True – якщо оператор виконано безпомилково, false – якщо ні | SyntaxTree.h |
| 37 | NTVoidFuncCall | execute | Виконання оператора виклику функції | Відсутні | Повертає значення типу bool. True – якщо оператор виконано безпомилково, false – якщо ні | SyntaxTree.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 38 | FuncNameTable | addParam | Додання параметра до таблиці імен функціі | string kee – ім’я параметра.  VariableData \*data – значення параметра | Повертає значення типу bool. True – якщо параметр додано успішно, false – якщо ні | FuncNameTable.h |
| 39 | FuncNameTable | addLocalVar | Додання локальної змінної до таблиці імен функціі | string kee – ім’я змінної.  VariableData \*data – значення змінної | Повертає значення типу bool. True – якщо змінну додано успішно, false – якщо ні | FuncNameTable.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 40 | FuncNameTable | getParam | Отримання параметра з таблиці імен функціі | string kee – ім’я параметра. | Повертає значення типу  VariableData\* – значення параметра | FuncNameTable.h |
| 41 | FuncNameTable | getLocalVar | Отримання локальної змінної з таблиці імен функціі | string kee – ім’я змінної. | Повертає значення типу  VariableData\* – значення змінної | FuncNameTable.h |
| 42 | FuncNameTable | changeLocalVar | Зміна значення локальної змінної в таблиці імен функціі | string kee – ім’я змінної.  VariableData \*newData – значення змінної | Повертає значення типу bool. True – якщо змінну змінено успішно, false – якщо ні | FuncNameTable.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 43 | StandardMethods | checkFuncName | Перевірка, чи є рядок викликом стандартної функції | string \_name – рядок, який необхідно перевірити | Повертає значення типу STANDARD\_FUNC\_NAME – стандартний метод, який викликає функція. Якщо рядок не є викликом стандартного методу, повертається NONE | StandardMethods.h |
| 44 | StandardMethods | findFuncValue | Знаходження результату стандартної функції | STANDARD\_FUNC\_NAME name – ім’я метода. vector<VariableData\*> params – список параметрів | Повертає значення типу VariableData\* - результат виконання методу | StandardMethods.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 45 | StandardMethods | standard\_Det | Знаходження визначника матриці | vector<vector<double>> matrix – матриця. | Повертає значення типу double – визначник матриці | StandardMethods.h |
| 46 | StandardMethods | standard\_Solve | Знаходження розв’язку СЛАР | vector<vector<double>> matrix – матриця.  vector<double> freeMembers – стовпець вільних членів | Повертає значення типу vector<double> – розв’язок системи | StandardMethods.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 47 | StandardMethods | standard\_Transpose | Транспонування матриці | vector<vector<double>> matrix – матриця. | Повертає значення типу vector<vector<double>> – відтранспонована матриця | StandardMethods.h |
| 48 | StandardMethods | standard\_Invert | Знаходження оберненої матриці | vector<vector<double>> matrix – матриця. | Повертає значення типу vector<vector<double>> – обернена матриця | StandardMethods.h |

Продовження таблиці 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 49 | StandardMethods | makeTriangleMatrix | Зведення матриці до трикутного вигляду | vector<vector<double>> matrix – матриця. bool &isSignChanged – відслідковування того, чи потрібно змінювати знак визначника | Повертає значення типу vector<vector<double>> – трикутна матриця | StandardMethods.h |

Таблиця 4.2 – Опис стандартних класів та методів

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 1 |  | getline | Зчитування інформації із потоку | istream &is – поток зчитування. String &str – рядок, у який відбувається зчитування | Функція повертає значення типу istream & - поток зчитування | string |
| 2 | vector | clear | Очищення масиву | Відсутні | Функція не повертає результату | vector |
| 3 | vector | push\_back | Вставка елементу в кінець масиву | T element – елемент для вставки (тип співпадає з типом вектора) | Функція не повертає результату | vector |
| 4 | vector | empty | Перевірка, чи є масив порожнім | Відсутні | Функція повертає результат типу bool. True, якщо масив порожній, False – якщо ні | vector |

Продовження таблиці 4.2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № п/п | Назва класу | Назва функції | Призначення функції | Опис вхідних параметрів | Опис вихідних параметрів | Заголовний файл |
| 5 | map | insert | Вставка елементу в хеш-таблицю | pair<T1, T2> element – пара елементів для вставки (типи співпадають з типами хеш-таблиці) | Функція не повертає результату | map |
| 6 | map | count | Визначення кількості елементів хеш-таблиці із заданим ключем | T kee –елемент для перевірки (тип співпадає з типом ключа хеш-таблиці) | Функція повертає результат типу int – кількість елементів таблиці із заданим ключем | map |

# тестування програмного забезпечення

## План тестування

Для перевірки працездатності програмного забезпечення було проведено його тестування. Програмне забезпечення повинне правильно обробляти вхідні дані, в разі їх некоректності сповіщати про це користувача. Програма повинна адекватно реагувати на введення некоректного шляху до файлу з кодом. В разі введення коду, який містить лексичну помилку, програма повинна сповіщати про це користувача. В разі введення коду, який містить синтаксичну помилку, програма повинна сповіщати про це користувача. Також програма повинна коректно обробляти усі нештатні ситуації, які можуть виникнути в ході обрахунків, наприклад, ділення на нуль, знаходження визначника неквадратної матриці тощо.

План тестування програмного забезпечення на відповідність усім вищесказаним вимогах наведено нижче.

1. Тестування при введенні некоректного шляху до файлу з кодом
2. Тестування при наявності в коді лексичних та синтаксичних помилок
   1. Тестування при наявності в коді лексичної помилки
   2. Тестування при наявності в коді синтаксичної помилки
3. Тестування при наявності в коді недозволених операцій
   1. Тестування при наявності в коді некоректного присвоювання
   2. Тестування при наявності в коді ділення на нуль
4. Тестування коректності роботи арифметичних та логічних операцій
   1. Тестування операції піднесення до степеню
   2. Тестування логічної операції «дорівнює»

## Приклади тестування

Тестування програмного забезпечення було проведено згідно з планом, наведеним у п. 5.1. В ході тестування було отримано наступні результати.

### Тестування при введенні некоректного шляху до файлу з кодом

Після введення шляху до неіснуючого файлу було отримано повідомлення, яке зображено на рисунку 5.1.

Результати тестування при введенні некоректного шляху до файлу з кодом наведено у таблиці 5.1.

Таблиця 5.1 – Приклад роботи програми при введенні некоректних значень

|  |  |
| --- | --- |
| Мета тесту | Перевірити можливість введення некоректного шляху до файлу з кодом |
| Початковий стан програми | Програму не запущено |
| Вхідні дані | D:\nonexistent.txt |
| Схема проведення тесту | Введення некоректного шляху до файлу з кодом |
| Очікуваний результат | Повідомлення про помилку  зчитування з файлу |
| Стан програми після проведення випробувань | Видано помилку «Файл не існує» |



Рисунок 5.1 - Тестування при введенні некоректного шляху до файлу з кодом

### Тестування при наявності в коді лексичних та синтаксичних помилок

Після спроби проінтерпретувати код, який містить лексичну помилку, було отримано результат, який зображено на рисунку 5.2.

Після спроби проінтерпретувати код, який містить синтаксичну помилку, було отримано результат, який зображено на рисунку 5.3.

Результати тестування при наявності в коді лексичних та синтаксичних помилок наведено у таблиці 5.2.

Таблиця 5.2 – Приклад роботи програми при наявності в коді лексичних та синтаксичних помилок

|  |  |
| --- | --- |
| Мета тесту | Перевірити можливість інтерпретування коду з лексичними та синтаксичними помилками |
| Початковий стан програми | Робоча програма |
| Вхідні дані | num 6abc;  num abc |
| Схема проведення тесту | Введення коду, що містить лексичну помилку.  Введення коду, що містить синтаксичну помилку. |
| Очікуваний результат | Повідомлення про лексичну помилку.  Повідомлення про лексичну помилку. |
| Стан програми після проведення випробувань | Видано помилку «Лексична помилка».  Видано помилку «Синтаксична помилка». |

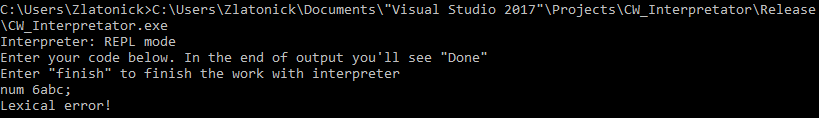


Рисунок 5.2 - Тестування при введенні коду з лексичною помилкою

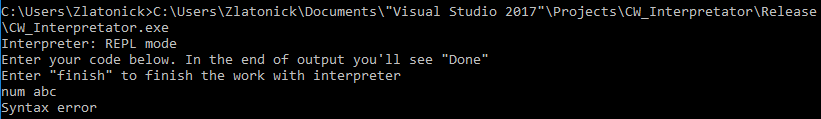


Рисунок 5.3 - Тестування при введенні коду з синтаксичною помилкою

### Тестування при наявності в коді недозволених операцій

Після спроби проінтерпретувати код, який містить некоректне присвоєння, було отримано результат, який зображено на рисунку 5.4.

Після спроби проінтерпретувати код, який містить ділення на нуль, було отримано результат, який зображено на рисунку 5.5.

Результати тестування при наявності в коді недозволених операцій наведено у таблиці 5.3.

Таблиця 5.3 – Приклад роботи програми при наявності в коді недозволених операцій

|  |  |
| --- | --- |
| Мета тесту | Перевірити можливість інтерпретування коду з недозволеними операції |
| Початковий стан програми | Робоча програма |
| Вхідні дані | num number; vector vec;  number = 5; vec = number;  number = 5 / 0; |
| Схема проведення тесту | Введення коду, що містить некоретне присвоєння.  Введення коду, що містить ділення на нуль. |
| Очікуваний результат | Повідомлення про синтаксичну помилку |
| Стан програми після проведення випробувань | Видано помилку «Синтаксична помилка». |

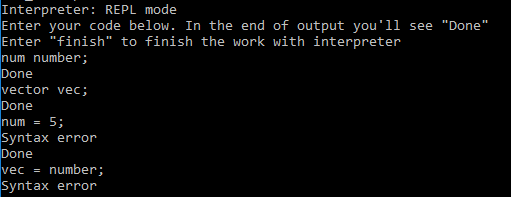


Рисунок 5.4 - Тестування при введенні коду з некоректним присвоєнням

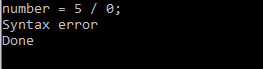


Рисунок 5.5 - Тестування при введенні коду з діленням на нуль

### Тестування коректності роботи арифметичних та логічних операцій

Після спроби проінтерпретувати код, який містить операцію піднесення до степеню, було отримано результат, який зображено на рисунку 5.6.

Після спроби проінтерпретувати код, який містить логічну операцію «дорівнює», було отримано результат, який зображено на рисунку 5.7.

Результати тестування при наявності в коді недозволених операцій наведено у таблиці 5.4.

Таблиця 5.4 – Приклад роботи програми при наявності в коді недозволених операцій

|  |  |
| --- | --- |
| Мета тесту | Перевірити правильність інтерпретування арифметичних та логічних операцій |
| Початковий стан програми | Робоча програма |
| Вхідні дані | num number; number = 5 ^ 2;  console(number);  if (number : 25) {  console(‘Correct’);  } |
| Схема проведення тесту | Введення коду, що містить арифметичні та логічні операції |
| Очікуваний результат | 25 Correct |
| Стан програми після проведення випробувань | Виведено «25 Correct» |

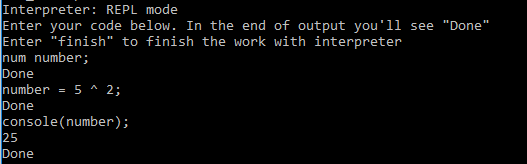


Рисунок 5.6 - Тестування при введенні коду з некоректним присвоєнням



Рисунок 5.7 - Тестування при введенні коду з некоректним присвоєнням

# інструкція користувача

## Призначення програми

Головним призначення програми є інтерпретування коду, написаного на розробленій мові програмування. Програму можна використовувати для того, щоб писати програми, у яких необхідно часто виконувати операції над векторами та матрицями. Адже розроблена мова програмування має спрощену систему команд, орієнтовану на роботу з матрицями та векторами, а також набір стандартних методів для їх обробки.

## Вимоги до системи

* Операційна система: Windows 7 або новіша
* RAM: 2 Gb або більше
* Вільне місце на диску: 5 Gb або більше
* Процесор: Intel Pentium 1.6 GHz або кращий
* Дисплей: 1024 x 768 або більший

## Інструкція по користуванню програмою

Програма складається з одного файлу CW\_Interpretator.exe розміром 108 Кб.

Програма може працювати у двох режимах: режимі інтерпретування цілого файлу з кодом і режимі інтерактивного інтерпретування read-eval-print loop (REPL).

Для того, щоб запустити програму в першому режимі, треба в командному рядку системи прописати повний шлях до файлу CW\_Interpretator.exe і після цього прописати шлях до файлу з кодом і натиснути Enter. Після цього в консоль буде виведено результат інтерпретування коду. Приклад використання програми у такому режимі зображено на рисунку 6.1.

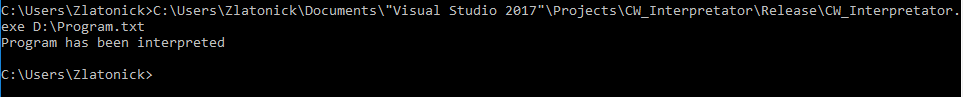


Рисунок 6.1 – Використання програми в режимі інтерпретування цілого файлу

Для того, щоб запустити програму в режимі REPL, треба в командному рядку системи прописати повний шлях до файлу CW\_Interpretator.exe і натиснути Enter. Після цього в консоль буде виведено мінімальну інструкцію користувача і пропозиція ввести перший оператор програми, що продемонстровано на рисунку 6.2.

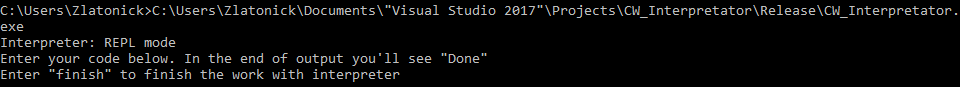


Рисунок 6.2 – Запуск програми в режимі REPL

Далі слід ввести оператор мови програмування і натиснути Enter. Буде виведено результат інтерпретування введеного оператора і після нього слово «Done». Далі можна продовжувати вводити оператори таким чином. Коли роботу з програмою слід буде завершити, слід прописати слово «finish» і натиснути Enter, після чого робота програми буде закінчена. Приклад роботи з програмою наведено на рисунку 6.3.

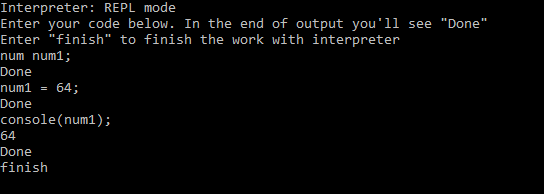


Рисунок 6.3 – Запуск програми в режимі REPL

висновки

На етапі підготовки до написання курсової роботи були детально розглянуті та вивчені усі аспекти написання інтерпретатора, найбільш ретельно – методи написання синтаксичних аналізаторів, способи зберігання введених даних про змінні, а також способи побудови синтаксичних дерев розбору.

Наступним етапом було проектування програми, під час чого було створено ієрархію класів для зберігання синтаксичних конструкцій, які були виокремлені з коду вхідної програми після проведення синтаксичного аналізу.

Після цього було проведено програмну реалізацію вивчених аспектів, перевірку їх працездатності при різних вхідних даних. Після завершення написання програми було проведено її тестування та перевірка правильності роботи. Було встановлено, що програма коректно обробляє будь-які вхідні дані.

У ході виконання курсової роботи було ретельно вивчено основні аспекти написання інтерпретаторів. Було розроблено програмне забезпечення, яке дає змогу розробляти та інтерпретувати програми на розробленій мові програмування.

перелік посилань

1. Молчанов А. Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. - СПб.: Питер, 2010.
2. Афанасьев А. Н. Формальные языки и грамматики: учеб. пособие. — Ульяновск: УлГТУ, 1997. — 84 с.
3. Ахо А. Компиляторы: принципы, технологии и инструментарий / А. Ахо [и др.]. - М.: Издательский дом «Вильямс», 2008. — 1184 с.
4. Гордеев А. В., Молчанов А. Ю. Системное программное обеспечение. — СПб.: Питер, 2001. — 734 с.
5. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975. — 544 с.

Додаток А Тексти програмного коду

*студента групи ІП-61 ІІ курсу*

*Златокрильця М.О.*

(Обсяг програми (документа), арк., Кб)

*112 арк, 108 Кб*

(Вид носія даних)

*CD-RW*

(Найменування програми (документа))

*Тексти програмного коду програмного забезпечення інтерпретатора мови програмування*

**А.1 globals.h**

#pragma once

#include <cstdlib>

#include <iostream>

#include <iomanip>

#include <string>

#include <vector>

#include <map>

#include <fstream>

#include <sstream>

**A.2 Type.h**

#pragma once

#include "globals.h"

/\*

\* Token types

\*/

using namespace std;

enum Type {

NUM, // num

VECTOR, // vector

MATR, // matr

CONSOLE, // console

IF, // if

FUNC, // func

VOID, // void

RETURN, // return

INCLUDE, // include

COMMA, // ,

SEMICOLON, // ;

ASSIGN, // =

PLUS, // +

MINUS, // -

MULTIPLY, // \*

DIVIDE, // /

POWER, // ^

EQUALS, // :

NOT\_EQUALS, // !

MORE, // >

LESS, // <

OPEN\_BRACKET, // (

CLOSE\_BRACKET, // )

OPEN\_SQUARE\_BRACKET, // [

CLOSE\_SQUARE\_BRACKET, // ]

OPEN\_FIGURE\_BRACKET, // {

CLOSE\_FIGURE\_BRACKET, // }

NUMBER, // 3.86

NAME, // matrix1

TEXT, // 'sample text'

END, // end of file

ERROR // error (not a token)

};

Type stringToType(string line);

string typeToString(Type type);

**A.3 Type.cpp**

#include "Type.h"

using namespace std;

Type stringToType(string line) {

if (line == "num") return NUM;

if (line == "vector") return VECTOR;

if (line == "matr") return MATR;

if (line == "console") return CONSOLE;

if (line == "if") return IF;

if (line == "func") return FUNC;

if (line == "void") return VOID;

if (line == "return") return RETURN;

if (line == "include") return INCLUDE;

if (line == ",") return COMMA;

if (line == ";") return SEMICOLON;

if (line == "=") return ASSIGN;

if (line == "+") return PLUS;

if (line == "-") return MINUS;

if (line == "\*") return MULTIPLY;

if (line == "/") return DIVIDE;

if (line == "^") return POWER;

if (line == ":") return EQUALS;

if (line == "!") return NOT\_EQUALS;

if (line == ">") return MORE;

if (line == "<") return LESS;

if (line == "(") return OPEN\_BRACKET;

if (line == ")") return CLOSE\_BRACKET;

if (line == "[") return OPEN\_SQUARE\_BRACKET;

if (line == "]") return CLOSE\_SQUARE\_BRACKET;

if (line == "{") return OPEN\_FIGURE\_BRACKET;

if (line == "}") return CLOSE\_FIGURE\_BRACKET;

return END;

}

string typeToString(Type type) {

switch (type) {

case 0: return "num";

case 1: return "vector";

case 2: return "matr";

case 3: return "console";

case 4: return "if";

case 5: return "func";

case 6: return "void";

case 7: return "return";

case 8: return "include";

case 9: return ",";

case 10: return ";";

case 11: return "=";

case 12: return "+";

case 13: return "-";

case 14: return "\*";

case 15: return "/";

case 16: return "^";

case 17: return ":";

case 18: return "!";

case 19: return ">";

case 20: return "<";

case 21: return "(";

case 22: return ")";

case 23: return "[";

case 24: return "]";

case 25: return "{";

case 26: return "}";

case 27: return "number";

case 28: return "name";

case 29: return "text";

}

return "";

}

**A.4 Token.h**

#pragma once

#include "globals.h"

#include "Type.h"

using namespace std;

class Token {

Type type;

string value; // for types NUMBER, NAME and TEXT

public:

Token();

Token(Type type, string value);

void setType(Type type);

Type getType();

void setValue(string value);

string getValue();

};

**A.5 Token.cpp**

#include "Token.h"

using namespace std;

Token::Token() {

this->value = "";

}

Token::Token(Type type, string value) {

this->type = type;

this->value = value;

}

void Token::setType(Type type) {

this->type = type;

}

Type Token::getType() {

return this->type;

}

void Token::setValue(string value) {

this->value = value;

}

string Token::getValue() {

return this->value;

}

**A.6 Lexer.h**

#pragma once

#include "globals.h"

#include "Type.h"

#include "Token.h"

using namespace std;

/\*

\* Lexical analizer

\* TEXT ---> ARRAY OF TOKENS + catching the lexical errors

\*/

class Lexer {

string input; // Code for converting

int currentElement;

char currentSymbol;

Token readToken(); // Read next token

void readSymbol(); // Read next symbol

bool identifyToken(Token &token, string value); // Identifying the token. False if is not a token

bool isNumber(string line); // Checking if line is a float number

bool isDigit(char symbol); // Checking if symbol is a digit

bool isName(string line); // Checking if line is a name

bool isLetter(char symbol); // Checking if symbol is a letter

public:

Lexer();

Lexer(string input);

void setInput(string input);

vector<Token> convert(); // Executing a lexical analysis and forming the array of tokens.

// If input contains a lexical error, returns an empty array

};

**A.7 Lexer.cpp**

#include "Lexer.h"

using namespace std;

Lexer::Lexer() {

this->input = "";

this->currentElement = 0;

this->currentSymbol = NULL;

}

Lexer::Lexer(string input) {

this->input = input;

this->currentElement = 0;

this->currentSymbol = NULL;

}

void Lexer::setInput(string input) {

this->input = input;

}

vector<Token> Lexer::convert() {

vector<Token> result;

this->currentElement = 0;

this->currentSymbol = NULL;

Token token = readToken();

while (true) {

if (token.getType() == ERROR) {

result.clear();

return result;

}

if (token.getType() == END) {

result.push\_back(token);

break;

}

result.push\_back(token);

token = readToken();

}

return result;

}

Token Lexer::readToken() {

Token token;

string symbols = "";

readSymbol();

// Ignoring spaces, tabs and enters

while (currentSymbol == ' ' || currentSymbol == '\n' || currentSymbol == '\t') {

readSymbol();

}

// If the end of line

if (currentSymbol == '\0') {

token.setType(END);

return token;

}

// If a text token

if (currentSymbol == '\'') {

while (true) {

readSymbol();

// Unclosed quote

if (currentSymbol == '\0') {

token.setType(ERROR);

return token;

}

if (currentSymbol == '\'')

break;

symbols += currentSymbol;

}

token.setType(TEXT);

token.setValue(symbols);

return token;

}

symbols += currentSymbol;

// If single-character token

if (identifyToken(token, symbols)) {

return token;

}

// Reading the whole word

readSymbol();

while (isDigit(currentSymbol) || isLetter(currentSymbol) || currentSymbol == '\_' || currentSymbol == '.') {

symbols += currentSymbol;

readSymbol();

}

currentElement--; // Correcting the current position

// If a key-word token

if (identifyToken(token, symbols)) {

return token;

}

// Checking for number

if (isNumber(symbols)) {

token.setType(NUMBER);

token.setValue(symbols);

return token;

}

// Checking for name

if (isName(symbols)) {

token.setType(NAME);

token.setValue(symbols);

return token;

}

// Not a token

token.setType(ERROR);

return token;

}

void Lexer::readSymbol() {

this->currentSymbol = this->input[this->currentElement];

this->currentElement++;

}

bool Lexer::identifyToken(Token &token, string value) {

vector<string> singleCharacters = { ",", ";", "\'", "=", "+", "-", "\*", "/", "^", ":", "!",

">", "<", "(", ")", "[", "]", "{", "}" };

vector<string> keyWords = { "num", "vector", "matr", "console", "if", "func", "void",

"return", "include" };

// Checking for single-character token

for (int i = 0; i < singleCharacters.size(); i++) {

if (value == singleCharacters[i]) {

token.setType(stringToType(value));

return true;

}

}

// Checking for keyword token

for (int i = 0; i < keyWords.size(); i++) {

if (value == keyWords[i]) {

token.setType(stringToType(value));

return true;

}

}

return false;

}

bool Lexer::isNumber(string line) {

if (line.length() == 0)

return false;

bool dotOnce = true; // True if no dots

for (int i = 0; i < line.length(); i++) {

if (!isDigit(line[i])) {

if (line[i] == '.' && dotOnce == true)

dotOnce = false;

else

return false;

}

}

return true;

}

bool Lexer::isDigit(char symbol) {

if (symbol - '0' < 0 || symbol - '0' > 9)

return false;

else

return true;

}

bool Lexer::isName(string line) {

if (line.length() == 0)

return false;

// First symbol must be a letter or "\_"

if (!isLetter(line[0]) && line[0] != '\_')

return false;

// Letter, digit or "\_"

for (int i = 1; i < line.length(); i++) {

if (!isLetter(line[i]) && !isDigit(line[i]) && line[i] != '\_')

return false;

}

return true;

}

bool Lexer::isLetter(char symbol) {

if (symbol - 'A' < 0 || symbol - 'A' > 25) {

if (symbol - 'a' < 0 || symbol - 'a' > 25)

return false;

}

return true;

}

**A.8 Parser.h**

#pragma once

#include "globals.h"

#include "Token.h"

#include "Type.h"

#include "FuncNameTable.h"

#include "SyntaxTree.h"

#include "FuncInfo.h"

using namespace std;

/\*

\* Syntactical analyzer (singleton)

\* ARRAY OF TOKENS ---> PARSE TREE + catching the syntax errors

\*/

class Parser {

static Parser \*\_this; // Pointer on the object of class Parser

vector<Token> tokens; // Received array of tokens

int currIndex; // Current index

// Tables of names

static map<string, VariableData\*> globalNameTable; // Table of global variable names

static vector<FuncNameTable> funcNameTables; // Table of local variable names (for each function)

static map<string, FuncInfo\*> funcInformation; // Information about each function (index of nameTable and body)

static int currTable; // Current functions name table (-1 if global table)

// Opened modules

static vector<string> openedModules;

// Working with syntax tree

vector<NTOperator\*> syntaxTree; // Syntax tree

bool ruleBlock(vector<NTOperator\*>& operators);

NTOperator\* ruleOperator();

// If returns nullptr - checking another operator

// If returns NTError - there is a syntax error

NTOperator\* ruleDeclare();

bool ruleType(Type& type);

NTOperator\* ruleAssign();

NTExpression\* ruleExpression();

bool ruleArgs(vector<NTExpression\*>& \_args);

bool ruleVector(vector<NTExpression\*>& line);

bool ruleElement(vector<NTExpression\*>& \_line);

bool ruleMatrix(vector<vector<NTExpression\*>>& values);

bool ruleLine(vector<vector<NTExpression\*>>& lines);

NTOperator\* ruleOutput();

NTOperator\* ruleCondition();

bool ruleLogicalExpr(Type& operType, NTExpression\*& oper1, NTExpression\*& oper2);

NTOperator\* ruleFunc();

bool ruleParam(vector<pair<Type, string>>& \_params);

bool ruleReturn(NTExpression \*&expression);

NTOperator\* ruleModule();

NTOperator\* ruleVoidFuncCall();

public:

Parser();

Parser(vector<Token> tokens); // Initial constructor

~Parser();

void setTokens(vector<Token> \_tokens); // Setting the array of tokens

void addTokens(vector<Token> \_tokens); // Adding tokens to the array

static Parser\* get(); // Getting the pointer on the object of class Parser

void analize(); // Checking for errors and creating a parse tree

// Tables of names: getters

static map<string, VariableData\*>\* getGlobalNameTable();

static vector<FuncNameTable>\* getFuncNameTables();

static map<string, FuncInfo\*>\* getFuncInformation();

static int\* getCurrTable();

// Working with tables of names

static bool addToTable(string name, VariableData \*data, bool isparam); // Add name to the table (false if already exists)

// Isparam - true if parameter, false if local var

static bool changeValueByKee(string name, VariableData \*newData); // Change value of variable. False if not found or incorrect types

static VariableData\* getValueByKee(string name); // ERROR if not found

// Array of already opened modules

static vector<string>\* getOpenedModules();

};

**A.9 Parser.cpp**

#include "Parser.h"

using namespace std;

Parser\* Parser::\_this = nullptr;

int Parser::currTable;

map<string, VariableData\*> Parser::globalNameTable;

vector<FuncNameTable> Parser::funcNameTables;

map<string, FuncInfo\*> Parser::funcInformation;

vector<string> Parser::openedModules;

Parser::Parser() {

// Singleton realization

if (\_this != nullptr)

exit(EXIT\_FAILURE);

\_this = this;

currIndex = 0;

currTable = -1;

}

Parser::Parser(vector<Token> tokens) {

// Singleton realization

if (\_this != nullptr)

exit(EXIT\_FAILURE);

\_this = this;

this->tokens = tokens;

currIndex = 0;

currTable = -1;

}

Parser::~Parser() {

map<string, VariableData\*>::iterator it1 = globalNameTable.begin();

while (it1 != globalNameTable.end()) {

delete (\*it1).second;

it1++;

}

map<string, FuncInfo\*>::iterator it2 = funcInformation.begin();

while (it2 != funcInformation.end()) {

delete (\*it2).second;

it2++;

}

}

void Parser::setTokens(vector<Token> \_tokens) {

this->tokens = \_tokens;

}

void Parser::addTokens(vector<Token> \_tokens) {

if (this->tokens.empty()) {

this->tokens = \_tokens;

}

else {

vector<Token>::iterator it = this->tokens.begin();

it += this->currIndex;

this->tokens.insert(it, \_tokens.begin(), \_tokens.end());

}

}

Parser\* Parser::get() {

return \_this;

}

bool Parser::addToTable(string name, VariableData \*data, bool isparam) {

if (currTable == -1) {

if (globalNameTable.count(name) > 0) {

return false;

}

else {

globalNameTable.insert(pair<string, VariableData\*>(name, data));

return true;

}

}

else {

if (isparam)

return funcNameTables[currTable].addParam(name, data);

else

return funcNameTables[currTable].addLocalVar(name, data);

}

}

bool Parser::changeValueByKee(string name, VariableData \*newData) {

if (currTable != -1) {

// Finding in local table

if (funcNameTables[currTable].changeLocalVar(name, newData))

return true;

// If not found, finding in global tables

}

map<string, VariableData\*> \*table = &(globalNameTable);

map<string, VariableData\*>::iterator it = table->find(name);

if (it != table->end()) {

if (it->second->getType() == newData->getType()) {

switch (newData->getType()) {

case NUM: { it->second->setNum(newData->getNum()); break; }

case VECTOR: { it->second->setVec(newData->getVec()); break; }

case MATR: { it->second->setMatr(newData->getMatr()); break; }

default: { return false; }

}

return true;

}

else {

return false;

}

}

else {

return false;

}

}

VariableData\* Parser::getValueByKee(string name) {

if (currTable != -1) {

// Finding in local table

VariableData \*result = funcNameTables[currTable].getLocalVar(name);

if (result->getType() == ERROR)

result = funcNameTables[currTable].getParam(name);

if (result->getType() != ERROR)

return result;

// If not found, finding in global tables

}

map<string, VariableData\*> \*table;

table = &(globalNameTable);

map<string, VariableData\*>::iterator it = table->find(name);

if (it != table->end()) {

return it->second;

}

else {

return new VariableData(); // Error

}

}

map<string, VariableData\*>\* Parser::getGlobalNameTable() {

return &globalNameTable;

}

vector<FuncNameTable>\* Parser::getFuncNameTables() {

return &funcNameTables;

}

map<string, FuncInfo\*>\* Parser::getFuncInformation() {

return &funcInformation;

}

int\* Parser::getCurrTable() {

return &currTable;

}

vector<string>\* Parser::getOpenedModules() {

return &openedModules;

}

void Parser::analize() {

NTOperator \*regularOperator;

bool isError = false;

while (tokens[currIndex].getType() != END) {

regularOperator = ruleOperator();

currIndex++;

if (!regularOperator->execute()) {

isError = true;

break;

}

delete regularOperator;

}

if (isError) {

cout << "Syntax error" << endl;

}

}

bool Parser::ruleBlock(vector<NTOperator\*>& operators) {

NTOperator \*ntOperator = ruleOperator();

if (ntOperator->getOperatorType() != "error") {

operators.push\_back(ntOperator);

currIndex++;

// Possible ends of block

if (tokens[currIndex].getType() != END &&

tokens[currIndex].getType() != CLOSE\_FIGURE\_BRACKET &&

tokens[currIndex].getType() != RETURN) {

if (!ruleBlock(operators)) {

for (int i = 0; i < operators.size(); i++)

delete operators[i];

operators.clear();

return false;

}

return true;

}

currIndex--;

return true;

}

return false;

}

NTOperator\* Parser::ruleOperator() {

NTOperator \*ntOperator;

ntOperator = ruleDeclare();

if (ntOperator != nullptr) {

return ntOperator;

}

ntOperator = ruleAssign();

if (ntOperator != nullptr) {

return ntOperator;

}

ntOperator = ruleOutput();

if (ntOperator != nullptr) {

return ntOperator;

}

ntOperator = ruleCondition();

if (ntOperator != nullptr) {

return ntOperator;

}

ntOperator = ruleFunc();

if (ntOperator != nullptr) {

return ntOperator;

}

ntOperator = ruleModule();

if (ntOperator != nullptr) {

return ntOperator;

}

ntOperator = ruleVoidFuncCall();

if (ntOperator != nullptr) {

return ntOperator;

}

return new NTError();

}

//

// DECLARE RULES

//

NTOperator\* Parser::ruleDeclare() {

Type type;

string name;

if (!ruleType(type)) {

return nullptr;

}

currIndex++;

if (tokens[currIndex].getType() != NAME) {

return new NTError();

}

name = tokens[currIndex].getValue();

currIndex++;

if (tokens[currIndex].getType() != SEMICOLON) {

return new NTError();

}

// Adding variable to the table of names

/\*if (!addToTable(name, type, false)) {

return new NTError();

}\*/

return new NTDeclare(type, name);

}

bool Parser::ruleType(Type& type) {

if (tokens[currIndex].getType() == NUM ||

tokens[currIndex].getType() == VECTOR ||

tokens[currIndex].getType() == MATR) {

type = tokens[currIndex].getType();

return true;

}

return false;

}

//

// ASSIGN RULES

//

NTOperator\* Parser::ruleAssign() {

int inputIndex = this->currIndex;

string name; // Name of the variable

NTExpression \*index1 = nullptr; // If vector or matrix

NTExpression \*index2 = nullptr;

NTExpression \*expression = nullptr; // Expression in the right side of "="

if (tokens[currIndex].getType() != NAME) {

return nullptr;

}

name = tokens[currIndex].getValue();

currIndex++;

// Assigning to the element of vector (matrix)

if (tokens[currIndex].getType() == OPEN\_SQUARE\_BRACKET) {

currIndex++;

index1 = ruleExpression();

if (index1 == nullptr) {

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_SQUARE\_BRACKET) {

delete index1;

return new NTError();

}

currIndex++;

// If matrix

if (tokens[currIndex].getType() == OPEN\_SQUARE\_BRACKET) {

currIndex++;

index2 = ruleExpression();

if (index2 == nullptr) {

delete index1;

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_SQUARE\_BRACKET) {

delete index1;

delete index2;

return new NTError();

}

currIndex++;

}

}

if (tokens[currIndex].getType() != ASSIGN) {

// Without error (can be function call)

if (index1 != nullptr) delete index1;

if (index2 != nullptr) delete index2;

currIndex = inputIndex;

return nullptr;

}

currIndex++;

expression = ruleExpression();

if (expression == nullptr) {

if (index1 != nullptr) delete index1;

if (index2 != nullptr) delete index2;

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != SEMICOLON) {

if (index1 != nullptr) delete index1;

if (index2 != nullptr) delete index2;

delete expression;

return new NTError();

}

return new NTAssign(name, index1, index2, expression);

}

NTExpression\* Parser::ruleExpression() {

int inputIndex = this->currIndex;

NTExpression \*expression = nullptr;

/\*if (tokens[currIndex].getType() == OPEN\_BRACKET) {

currIndex++;

if (!ruleExpression()) {

this->currIndex = inputIndex;

return nullptr;

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_BRACKET) {

this->currIndex = inputIndex;

return nullptr;

}

}

else\*/ if (tokens[currIndex].getType() == MINUS) {

currIndex++;

NTExpression \*inside\_expression = ruleExpression();

if (inside\_expression == nullptr) {

this->currIndex = inputIndex;

return nullptr;

}

expression = new NTUnaryMinus(inside\_expression);

}

else if (tokens[currIndex].getType() == NUMBER) {

expression = new NTConstant(tokens[currIndex].getValue());

}

else if (tokens[currIndex].getType() == NAME) {

string name = tokens[currIndex].getValue();

currIndex++;

// If function call

if (tokens[currIndex].getType() == OPEN\_BRACKET) {

currIndex++;

vector<NTExpression\*> args;

if (tokens[currIndex].getType() != VOID) {

if (!ruleArgs(args)) {

this->currIndex = inputIndex;

return nullptr;

}

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_BRACKET) {

this->currIndex = inputIndex;

return nullptr;

}

expression = new NTFuncCall(name, args);

}

// vector[] or matr[][]

else if (tokens[currIndex].getType() == OPEN\_SQUARE\_BRACKET) {

currIndex++;

NTExpression \*index1 = ruleExpression();

if (index1 == nullptr) {

this->currIndex = inputIndex;

return nullptr;

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_SQUARE\_BRACKET) {

delete index1;

this->currIndex = inputIndex;

return nullptr;

}

// Checking for matr[][]

currIndex++;

if (tokens[currIndex].getType() == OPEN\_SQUARE\_BRACKET) {

currIndex++;

NTExpression \*index2 = ruleExpression();

if (index2 == nullptr) {

delete index1;

this->currIndex = inputIndex;

return nullptr;

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_SQUARE\_BRACKET) {

delete index1;

delete index2;

this->currIndex = inputIndex;

return nullptr;

}

expression = new NTElementOfMatrix(name, index1, index2);

}

else {

currIndex--;

expression = new NTElementOfVector(name, index1);

}

}

else {

// Expression is a simple name

currIndex--;

expression = new NTVariable(name);

}

}

else if (tokens[currIndex].getType() == OPEN\_SQUARE\_BRACKET) {

vector<vector<NTExpression\*>> valuesMatr;

if (!ruleMatrix(valuesMatr)) {

vector<NTExpression\*> valuesVec;

if (!ruleVector(valuesVec)) {

this->currIndex = inputIndex;

return nullptr;

}

else {

expression = new NTVector(valuesVec);

}

}

else {

expression = new NTMatrix(valuesMatr);

}

}

else {

this->currIndex = inputIndex;

return nullptr;

}

currIndex++;

if (tokens[currIndex].getType() == PLUS ||

tokens[currIndex].getType() == MINUS ||

tokens[currIndex].getType() == MULTIPLY ||

tokens[currIndex].getType() == DIVIDE ||

tokens[currIndex].getType() == POWER) {

Type type = tokens[currIndex].getType();

currIndex++;

NTExpression \*secondOperand = ruleExpression();

if (secondOperand == nullptr) {

delete expression;

this->currIndex = inputIndex;

return nullptr;

}

return new NTBinaryOperation(type, expression, secondOperand);

}

else {

currIndex--;

return expression;

}

}

bool Parser::ruleArgs(vector<NTExpression\*>& \_args) {

NTExpression \*ntExpression = ruleExpression();

if (ntExpression != nullptr) {

\_args.push\_back(ntExpression);

currIndex++;

if (tokens[currIndex].getType() == COMMA) {

currIndex++;

if (!ruleArgs(\_args)) {

return false;

}

return true;

}

currIndex--;

return true;

}

return false;

}

bool Parser::ruleVector(vector<NTExpression\*>& line) {

int initIndex = currIndex; // Without errors

// Instead of vector can be matrix

if (tokens[currIndex].getType() == OPEN\_SQUARE\_BRACKET) {

// (vvp , vvp , ... , vvp)

currIndex++;

if (!ruleElement(line)) {

currIndex = initIndex;

return false;

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_SQUARE\_BRACKET) {

// Deleting the array

for (int i = 0; i < line.size(); i++)

delete line[i];

line.clear();

currIndex = initIndex;

return false;

}

return true;

}

return false;

}

bool Parser::ruleElement(vector<NTExpression\*>& \_line) {

NTExpression \*element = ruleExpression();

if (element != nullptr) {

\_line.push\_back(element);

currIndex++;

if (tokens[currIndex].getType() == COMMA) {

currIndex++;

if (!ruleElement(\_line)) {

for (int i = 0; i < \_line.size(); i++)

delete \_line[i];

\_line.clear();

return nullptr;

}

return true;

}

currIndex--;

return true;

}

return false;

}

bool Parser::ruleMatrix(vector<vector<NTExpression\*>>& values) {

int initIndex = currIndex; // Without errors

// Instead of matrix can be vector

if (tokens[currIndex].getType() == OPEN\_SQUARE\_BRACKET) {

currIndex++;

if (!ruleLine(values)) {

currIndex = initIndex;

return false;

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_SQUARE\_BRACKET) {

for (int i = 0; i < values.size(); i++)

for (int j = 0; j < values[i].size(); j++)

delete values[i][j];

values.clear();

currIndex = initIndex;

return false;

}

return true;

}

return false;

}

bool Parser::ruleLine(vector<vector<NTExpression\*>>& lines) {

vector<NTExpression\*> line;

if (ruleVector(line)) {

lines.push\_back(line);

currIndex++;

if (tokens[currIndex].getType() == COMMA) {

currIndex++;

if (!ruleLine(lines)) {

for (int i = 0; i < lines.size(); i++)

for (int j = 0; j < lines[i].size(); j++)

delete lines[i][j];

lines.clear();

return false;

}

return true;

}

currIndex--;

return true;

}

return false;

}

//

// OUTPUT RULES

//

NTOperator\* Parser::ruleOutput() {

if (tokens[currIndex].getType() != CONSOLE) {

return nullptr;

}

bool isText = true;

string text = "";

NTExpression \*expression = nullptr;

currIndex++;

if (tokens[currIndex].getType() != OPEN\_BRACKET) {

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() == TEXT) {

text = tokens[currIndex].getValue();

}

else {

isText = false;

expression = ruleExpression();

if (expression == nullptr) {

return new NTError();

}

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_BRACKET) {

delete expression;

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != SEMICOLON) {

delete expression;

return new NTError();

}

return new NTOutput(isText, text, expression);

}

//

// CONDITION RULES

//

NTOperator\* Parser::ruleCondition() {

if (tokens[currIndex].getType() != IF)

return nullptr;

Type logicalOperation;

NTExpression \*operand1 = nullptr;

NTExpression \*operand2 = nullptr;

vector<NTOperator\*> block;

currIndex++;

if (tokens[currIndex].getType() != OPEN\_BRACKET) {

return new NTError();

}

currIndex++;

if (!ruleLogicalExpr(logicalOperation, operand1, operand2)) {

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_BRACKET) {

delete operand1;

delete operand2;

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != OPEN\_FIGURE\_BRACKET) {

delete operand1;

delete operand2;

return new NTError();

}

currIndex++;

if (!ruleBlock(block)) {

delete operand1;

delete operand2;

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_FIGURE\_BRACKET) {

delete operand1;

delete operand2;

return new NTError();

}

return new NTCondition(logicalOperation, operand1, operand2, block);

}

bool Parser::ruleLogicalExpr(Type& operType, NTExpression\*& oper1, NTExpression\*& oper2) {

int initIndex = currIndex; // Without errors

oper1 = ruleExpression();

if (oper1 != nullptr) {

currIndex++;

if (tokens[currIndex].getType() != EQUALS &&

tokens[currIndex].getType() != NOT\_EQUALS &&

tokens[currIndex].getType() != MORE &&

tokens[currIndex].getType() != LESS) {

currIndex = initIndex;

delete oper1;

return false;

}

operType = tokens[currIndex].getType();

currIndex++;

oper2 = ruleExpression();

if (oper2 == nullptr) {

currIndex = initIndex;

delete oper1;

return false;

}

return true;

}

return false;

}

//

// PROCEDURE RULES

//

NTOperator\* Parser::ruleFunc() {

if (tokens[currIndex].getType() != FUNC)

return nullptr;

string funcName = "";

vector<pair<Type, string>> params;

vector<NTOperator\*> block;

NTExpression \*result = nullptr;

currIndex++;

if (tokens[currIndex].getType() != NAME) {

return new NTError();

}

funcName = tokens[currIndex].getValue();

currIndex++;

if (tokens[currIndex].getType() != OPEN\_BRACKET) {

return new NTError();

}

currIndex++;

// Choosing the function name table

FuncNameTable funcNameTable;

this->funcNameTables.push\_back(funcNameTable);

this->currTable = this->funcNameTables.size() - 1;

if (tokens[currIndex].getType() != VOID && !ruleParam(params)) {

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_BRACKET) {

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != OPEN\_FIGURE\_BRACKET) {

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != RETURN) {

if (!ruleBlock(block))

return new NTError();

else

currIndex++;

}

if (!ruleReturn(result)) {

// Deleting block

for (int i = 0; i < block.size(); i++)

delete block[i];

block.clear();

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_FIGURE\_BRACKET) {

// Deleting block

for (int i = 0; i < block.size(); i++)

delete block[i];

block.clear();

if (result != nullptr)

delete result;

return new NTError();

}

NTFunc \*thisFunc = new NTFunc(funcName, params, block, result);

// Saving the information about function

if (Parser::funcInformation.count(funcName) > 0) {

return new NTError();

}

else {

Parser::funcInformation.insert(pair<string, FuncInfo\*>(funcName, new FuncInfo(false, this->currTable, thisFunc)));

}

// Choosing the global nametable

this->currTable = -1;

return thisFunc;

}

bool Parser::ruleParam(vector<pair<Type, string>>& \_params) {

int initIndex = currIndex; // Without errors;

Type type;

string name;

if (ruleType(type)) {

currIndex++;

if (tokens[currIndex].getType() != NAME) {

initIndex = currIndex;

return false;

}

name = tokens[currIndex].getValue();

// Adding parameter to the table

/\*if (!addToTable(name, type, true)) {

return false;

}\*/

\_params.push\_back(pair<Type, string>(type, name));

currIndex++;

if (tokens[currIndex].getType() == COMMA) {

currIndex++;

if (!ruleParam(\_params)) {

initIndex = currIndex;

return false;

}

return true;

}

currIndex--;

return true;

}

return false;

}

bool Parser::ruleReturn(NTExpression \*&expression) {

if (tokens[currIndex].getType() == RETURN) {

currIndex++;

if (tokens[currIndex].getType() == VOID) {

expression = nullptr;

}

else {

expression = ruleExpression();

if (expression == nullptr) {

return false;

}

}

currIndex++;

if (tokens[currIndex].getType() != SEMICOLON) {

delete expression;

return false;

}

return true;

}

return false;

}

//

// MODULE RULES

//

NTOperator\* Parser::ruleModule() {

if (tokens[currIndex].getType() != INCLUDE) {

return nullptr;

}

string moduleName;

currIndex++;

if (tokens[currIndex].getType() != TEXT) {

return new NTError();

}

moduleName = tokens[currIndex].getValue();

currIndex++;

if (tokens[currIndex].getType() != SEMICOLON) {

return new NTError();

}

return new NTModule(moduleName);

}

//

// VOID FUNCTION RULES

//

NTOperator\* Parser::ruleVoidFuncCall() {

if (tokens[currIndex].getType() != NAME) {

return nullptr;

}

string funcName = tokens[currIndex].getValue();

vector<NTExpression\*> args;

currIndex++;

if (tokens[currIndex].getType() != OPEN\_BRACKET) {

// Can be an assign

currIndex--;

return nullptr;

}

currIndex++;

if (tokens[currIndex].getType() != VOID) {

if (!ruleArgs(args)) {

return new NTError();

}

}

currIndex++;

if (tokens[currIndex].getType() != CLOSE\_BRACKET) {

return new NTError();

}

currIndex++;

if (tokens[currIndex].getType() != SEMICOLON) {

return new NTError();

}

return new NTVoidFuncCall(funcName, args);

}

**A.10 SyntaxTree.h**

#pragma once

#include "globals.h"

#include "Type.h"

#include "VariableData.h"

class NTOperator {

protected:

string operatorType;

public:

void setOperatorType(string \_type) { operatorType = \_type; }

string getOperatorType() { return this->operatorType; }

virtual bool execute() = 0;

};

// If getting error while scanning the operator

class NTError : public NTOperator {

public:

NTError() { this->operatorType = "error"; }

bool execute() { return false; }

};

// Declare operator

class NTDeclare : public NTOperator {

Type type; // Type of the variable

string name; // Name of the variable

public:

NTDeclare(Type type, string name) : type(type), name(name) { this->operatorType = "declare"; }

Type getType() const { return this->type; }

string getName() const { return this->name; }

bool execute();

};

// Assign operator

class NTExpression {

protected:

string expressionType; // constant, variable, funcCall, etc.

public:

void setExpressionType(string \_type) { expressionType = \_type; }

string getExpressionType() { return this->expressionType; }

virtual VariableData\* calculate() = 0;

};

class NTAssign : public NTOperator {

string receiverValue; // Name of the receiver variable

NTExpression \*receiverIndex1; // First index (if vector or matrix)

NTExpression \*receiverIndex2; // Second index (if matrix)

NTExpression \*sender; // Expression in the right side of "="

public:

NTAssign(string recVal, NTExpression \*recIndex1, NTExpression \*recIndex2, NTExpression \*sender) :

receiverValue(recVal), receiverIndex1(recIndex1), receiverIndex2(recIndex2), sender(sender)

{ this->operatorType = "assign"; }

~NTAssign() { delete receiverIndex1; delete receiverIndex2; delete sender; }

string getReceiverValue() const { return this->receiverValue; }

NTExpression\* getReceiverIndex1() const { return this->receiverIndex1; }

NTExpression\* getReceiverIndex2() const { return this->receiverIndex2; }

NTExpression\* getSender() const { return this->sender; }

bool execute();

};

class NTConstant : public NTExpression {

string number;

public:

NTConstant(string num) : number(num) { this->expressionType = "constant"; }

string getNumber() const { return this->number; }

VariableData\* calculate();

};

class NTVariable : public NTExpression {

string name;

public:

NTVariable(string \_name) : name(\_name) { this->expressionType = "variable"; }

string getName() const { return this->name; }

VariableData\* calculate();

};

class NTFuncCall : public NTExpression {

string funcName;

vector<NTExpression\*> args;

public:

NTFuncCall(string \_funcName, vector<NTExpression\*> \_args) : funcName(\_funcName), args(\_args)

{ this->expressionType = "funcCall"; }

string getFuncName() const { return this->funcName; }

vector<NTExpression\*> getArgs() const { return this->args; }

VariableData\* calculate();

};

class NTBinaryOperation : public NTExpression {

Type operType;

NTExpression \*firstOperand;

NTExpression \*secondOperand;

public:

NTBinaryOperation(Type \_type, NTExpression \*\_firstOperand, NTExpression \*\_secondOperand) :

operType(\_type), firstOperand(\_firstOperand), secondOperand(\_secondOperand)

{ this->expressionType = "binaryOperation"; }

Type getOperType() const { return this->operType; }

NTExpression\* getFirstOperand() const { return this->firstOperand; }

NTExpression\* getSecondOperand() const { return this->secondOperand; }

VariableData\* calculate();

};

class NTUnaryMinus : public NTExpression {

NTExpression \*operand;

public:

NTUnaryMinus(NTExpression \*\_operand) : operand(\_operand) { this->expressionType = "unaryMinus"; }

NTExpression\* getOperand() const { return this->operand; }

VariableData\* calculate();

};

class NTElementOfVector : public NTExpression {

string name; // Name of vector

NTExpression \*index;

public:

NTElementOfVector(string \_name, NTExpression \*\_index) :

name(\_name), index(\_index) { this->expressionType = "elementOfVector"; }

string getName() const { return this->name; }

NTExpression\* getIndex() const { return this->index; }

VariableData\* calculate();

};

class NTElementOfMatrix : public NTExpression {

string name; // Name of vector

NTExpression \*index1;

NTExpression \*index2;

public:

NTElementOfMatrix(string \_name, NTExpression \*\_index1, NTExpression \*\_index2) :

name(\_name), index1(\_index1), index2(\_index2) { this->expressionType = "elementOfMatrix"; }

string getName() const { return this->name; }

NTExpression\* getIndex1() const { return this->index1; }

NTExpression\* getIndex2() const { return this->index2; }

VariableData\* calculate();

};

class NTVector : public NTExpression {

vector<NTExpression\*> values; // Coordinates of the vector

public:

NTVector(vector<NTExpression\*> \_values) : values(\_values) { this->expressionType = "vector"; }

vector<NTExpression\*> getValues() const { return this->values; }

VariableData\* calculate();

};

class NTMatrix : public NTExpression {

vector<vector<NTExpression\*>> values; // Elements of the matrix

public:

NTMatrix(vector<vector<NTExpression\*>> \_values) : values(\_values) { this->expressionType = "matrix"; }

vector<vector<NTExpression\*>> getValues() const { return this->values; }

VariableData\* calculate();

};

// Output operator

class NTOutput : public NTOperator {

bool isText; // true if text, false if expression

string text;

NTExpression \*expression;

public:

NTOutput(bool \_isText, string \_text, NTExpression\* \_expression) :

isText(\_isText), text(\_text), expression(\_expression)

{ this->operatorType = "output"; }

bool getIsText() const { return this->isText; }

string getText() const { return this->text; }

NTExpression\* getExpression() const { return this->expression; }

bool execute();

};

// Condition operator

class NTCondition : public NTOperator {

Type logicalOperation; // : ! < >

NTExpression \*operand1;

NTExpression \*operand2;

vector<NTOperator\*> block;

public:

NTCondition(Type \_logOper, NTExpression\* \_oper1, NTExpression\* \_oper2, vector<NTOperator\*> \_block) :

logicalOperation(\_logOper), operand1(\_oper1), operand2(\_oper2), block(\_block)

{ this->operatorType = "condition"; }

Type getLogicalOperation() const { return this->logicalOperation; }

NTExpression\* getOperand1() const { return this->operand1; }

NTExpression\* getOperand2() const { return this->operand2; }

vector<NTOperator\*> getBlock() const { return this->block; }

bool execute();

};

// Func operator

class NTFunc : public NTOperator {

string funcName;

vector<pair<Type, string>> params;

vector<NTOperator\*> block;

NTExpression \*result; // Expression after "return"

public:

NTFunc(string \_funcName, vector<pair<Type, string>> \_params, vector<NTOperator\*> \_block,

NTExpression\* \_result) : funcName(\_funcName), params(\_params), block(\_block), result(\_result)

{ this->operatorType = "func"; }

string getFuncName() const { return this->funcName; }

vector<pair<Type, string>> getParams() const { return this->params; }

vector<NTOperator\*> getBlock() const { return this->block; }

NTExpression\* getResult() const { return this->result; }

bool execute();

};

// Module operator

class NTModule : public NTOperator {

string moduleName;

public:

NTModule(string \_moduleName) : moduleName(\_moduleName)

{ this->operatorType = "module"; }

string getModuleName() const { return this->moduleName; }

bool execute();

};

// VoidFuncCall operator

class NTVoidFuncCall : public NTOperator {

string funcName;

vector<NTExpression\*> args;

public:

NTVoidFuncCall(string \_funcName, vector<NTExpression\*> \_args) : funcName(\_funcName), args(\_args)

{ this->operatorType = "voidFuncCall"; }

string getFuncName() const { return this->funcName; }

vector<NTExpression\*> getArgs() const { return this->args; }

bool execute();

};

**A.11 SyntaxTree.cpp**

#include "SyntaxTree.h"

#include "Parser.h"

#include "Lexer.h"

#include "StandardMethods.h"

using namespace std;

vector<vector<double>> multiplyMatrixes(vector<vector<double>> matr1, vector<vector<double>> matr2);

bool NTDeclare::execute() {

// Allocating memory for the variable / vector / matrix

switch (this->type) {

case NUM: {

double variable = 0;

if (!Parser::addToTable(this->name, new VariableData(variable), false)) {

return false;

}

break;

}

case VECTOR: {

vector<double> doubleArray;

if (!Parser::addToTable(this->name, new VariableData(doubleArray), false)) {

return false;

}

break;

}

case MATR: {

vector<vector<double>> doubleMatrix;

if (!Parser::addToTable(this->name, new VariableData(doubleMatrix), false)) {

return false;

}

break;

}

default: {

return false;

}

}

return true;

}

bool NTAssign::execute() {

VariableData \*result = this->sender->calculate();

if (this->getReceiverIndex1() != nullptr) {

VariableData \*index1\_val = this->receiverIndex1->calculate();

if (index1\_val->getType() != NUM)

return false;

if (this->getReceiverIndex2() != nullptr) {

// Assigning to the element of matrix

// Checking the result

if (result->getType() != NUM)

return false;

// Checking the second index

VariableData \*index2\_val = this->receiverIndex2->calculate();

if (index2\_val->getType() != NUM)

return false;

// Finding the matrix

VariableData \*matrixInTable = Parser::getValueByKee(this->receiverValue);

if (matrixInTable->getType() != MATR)

return false;

vector<vector<double>> matrixChange = matrixInTable->getMatr();

// Checking for correct indexes

if ((int)index1\_val->getNum() >= matrixChange.size() || (int)index1\_val->getNum() < 0)

return false;

if ((int)index2\_val->getNum() >= matrixChange[0].size() || (int)index2\_val->getNum() < 0)

return false;

// Changing the matrix

matrixChange[(int)index1\_val->getNum()][(int)index2\_val->getNum()] = result->getNum();

VariableData newMatrix(matrixChange);

if (!Parser::changeValueByKee(this->receiverValue, &newMatrix))

return false;

return true;

}

else {

// Assigning to the element of vector

// Checking the result

if (result->getType() != NUM)

return false;

// Finding the vector

VariableData \*vectorInTable = Parser::getValueByKee(this->receiverValue);

if (vectorInTable->getType() != VECTOR)

return false;

vector<double> vectorChange = vectorInTable->getVec();

// Checking for correct index

if ((int)index1\_val->getNum() >= vectorChange.size() || (int)index1\_val->getNum() < 0)

return false;

// Changing the vector

vectorChange[(int)index1\_val->getNum()] = result->getNum();

VariableData newVector(vectorChange);

if (!Parser::changeValueByKee(this->receiverValue, &newVector))

return false;

return true;

}

}

// Finding the receiver

VariableData \*receiver = Parser::getValueByKee(this->receiverValue);

// Checking the receiver type

switch (receiver->getType()) {

case NUM: {

if (result->getType() != NUM)

return false;

if (!Parser::changeValueByKee(this->receiverValue, &VariableData(result->getNum())))

return false;

return true;

}

case VECTOR: {

if (result->getType() != VECTOR)

return false;

if (!Parser::changeValueByKee(this->receiverValue, &VariableData(result->getVec())))

return false;

return true;

}

case MATR: {

if (result->getType() != MATR)

return false;

if (!Parser::changeValueByKee(this->receiverValue, &VariableData(result->getMatr())))

return false;

return true;

}

}

return false;

}

bool NTOutput::execute() {

if (this->getIsText()) {

// Outputting the text

cout << this->text << endl;

}

else {

// Outputting the expression

VariableData \*result = this->expression->calculate();

const int accuracy = 7;

const int weight = 10;

switch (result->getType()) {

case NUM: {

cout << setprecision(accuracy) << result->getNum() << endl;

break;

}

case VECTOR: {

for (int i = 0; i < result->getVec().size(); i++) {

cout << setprecision(accuracy) << result->getVec()[i] << " ";

}

cout << endl;

break;

}

case MATR: {

for (int i = 0; i < result->getMatr().size(); i++) {

for (int j = 0; j < result->getMatr()[i].size(); j++) {

cout << setprecision(accuracy) << setw(weight) << result->getMatr()[i][j] << " ";

}

cout << endl;

}

cout << endl;

break;

}

default: {

return false;

}

}

}

return true;

}

bool NTCondition::execute() {

VariableData \*oper1 = this->operand1->calculate();

VariableData \*oper2 = this->operand2->calculate();

// Same operands only

if (oper1->getType() != oper2->getType() || oper1->getType() == ERROR)

return false;

// Finding the value of logical expression

bool logicalExprValue = false;

switch (this->getLogicalOperation()) {

case EQUALS: {

switch (oper1->getType()) {

case NUM: {

if (oper1->getNum() == oper2->getNum())

logicalExprValue = true;

else

logicalExprValue = false;

break;

}

case VECTOR: {

vector<double> vector1 = oper1->getVec();

vector<double> vector2 = oper2->getVec();

if (vector1.size() != vector2.size()) {

logicalExprValue = false;

}

else {

logicalExprValue = true;

for (int i = 0; i < vector1.size(); i++) {

if (vector1[i] != vector2[i]) {

logicalExprValue = false;

}

}

}

break;

}

case MATR: {

vector<vector<double>> matr1 = oper1->getMatr();

vector<vector<double>> matr2 = oper2->getMatr();

if (matr1.empty() && matr2.empty()) {

logicalExprValue = true;

break;

}

if (matr1.size() != matr2.size() || matr1[0].size() != matr2[0].size()) {

logicalExprValue = false;

}

else {

logicalExprValue = true;

for (int i = 0; i < matr1.size(); i++) {

for (int j = 0; j < matr1[i].size(); j++) {

if (matr1[i][j] != matr2[i][j]) {

logicalExprValue = false;

}

}

}

}

break;

}

default: { return false; }

}

break;

}

case NOT\_EQUALS: {

switch (oper1->getType()) {

case NUM: {

if (oper1->getNum() != oper2->getNum())

logicalExprValue = true;

else

logicalExprValue = false;

break;

}

case VECTOR: {

vector<double> vector1 = oper1->getVec();

vector<double> vector2 = oper2->getVec();

if (vector1.size() != vector2.size()) {

logicalExprValue = true;

}

else {

logicalExprValue = false;

for (int i = 0; i < vector1.size(); i++) {

if (vector1[i] != vector2[i]) {

logicalExprValue = true;

}

}

}

break;

}

case MATR: {

vector<vector<double>> matr1 = oper1->getMatr();

vector<vector<double>> matr2 = oper2->getMatr();

if (matr1.empty() && matr2.empty()) {

logicalExprValue = false;

break;

}

if (matr1.size() != matr2.size() || matr1[0].size() != matr2[0].size()) {

logicalExprValue = true;

}

else {

logicalExprValue = false;

for (int i = 0; i < matr1.size(); i++) {

for (int j = 0; j < matr1[i].size(); j++) {

if (matr1[i][j] != matr2[i][j]) {

logicalExprValue = true;

}

}

}

}

break;

}

default: { return false; }

}

break;

}

case MORE: {

// Numbers only

if (oper1->getType() != NUM)

return false;

if (oper1->getNum() > oper2->getNum())

logicalExprValue = true;

else

logicalExprValue = false;

break;

}

case LESS: {

// Numbers only

if (oper1->getType() != NUM)

return false;

if (oper1->getNum() < oper2->getNum())

logicalExprValue = true;

else

logicalExprValue = false;

break;

}

default: { return false; }

}

// Executing of block

if (logicalExprValue) {

for (int i = 0; i < this->block.size(); i++) {

if (!this->block[i]->execute())

return false;

}

}

return true;

}

bool NTFunc::execute() {

// The code of function is not executing while declaring the function

return true;

}

bool NTModule::execute() {

// If already opened

for (int i = 0; i < Parser::getOpenedModules()->size(); i++) {

if ((\*Parser::getOpenedModules())[i] == this->moduleName) {

return false;

}

}

ifstream moduleFile(this->moduleName);

if (!moduleFile)

return false;

// Adding to the array of opened modules

Parser::getOpenedModules()->push\_back(this->moduleName);

// Reading the whole text of the file

stringstream fileBuffer;

fileBuffer << moduleFile.rdbuf();

string moduleText = fileBuffer.str();

moduleFile.close();

// Interpreting the code of module

Lexer lexer(moduleText);

vector<Token> tokens = lexer.convert();

if (tokens.empty()) {

// Lexical error

return false;

}

// Removing the END token

tokens.pop\_back();

Parser::get()->addTokens(tokens);

return true;

}

bool NTVoidFuncCall::execute() {

// Finding the information about the function

map<string, FuncInfo\*> \*funcTable = Parser::getFuncInformation();

map<string, FuncInfo\*>::iterator it = funcTable->find(this->funcName);

if (it == funcTable->end())

return false;

// Running the function

if (it->second->getIsRunning()) {

return false; // avoiding the recursion

}

else {

it->second->setIsRunning(true);

}

// Getting the local table and body of function

int tableIndex = it->second->getTableIndex();

FuncNameTable \*localTable = &((\*Parser::getFuncNameTables())[tableIndex]);

NTFunc \*ntFunc = it->second->getBody();

// Checking the type of parameters and adding them to the funcTable

vector<pair<Type, string>> ntFuncArgs = ntFunc->getParams();

if (this->args.size() != ntFuncArgs.size())

return false;

vector<pair<string, VariableData\*>> \*params = localTable->getParamTable();

for (int i = 0; i < this->args.size(); i++) {

VariableData \*currParam = this->args[i]->calculate();

// Checking the types

if (currParam->getType() != ntFuncArgs[i].first)

return false;

// Adding the value of parameter

params->push\_back(pair<string, VariableData\*>(ntFuncArgs[i].second, currParam));

}

// Switching the current table

\*Parser::getCurrTable() = tableIndex;

// Executing the body of function

vector<NTOperator\*> funcBlock = ntFunc->getBlock();

for (int i = 0; i < funcBlock.size(); i++) {

if (!funcBlock[i]->execute()) {

return false;

}

}

// Deleting the params and local vars, switching the current table to global

params->clear();

localTable->getLocalVarsTable()->clear();

\*Parser::getCurrTable() = -1;

return true;

}

VariableData\* NTConstant::calculate() {

return new VariableData(stod(this->number));

}

VariableData\* NTVariable::calculate() {

return Parser::getValueByKee(this->name);

}

VariableData\* NTFuncCall::calculate() {

// Checking if standard function

STANDARD\_FUNC\_NAME standardFuncName = StandardMethods::checkFuncName(this->funcName);

if (standardFuncName != NONE) {

// Finding the value of parameters

vector<VariableData\*> foundedParams;

for (int i = 0; i < this->args.size(); i++) {

foundedParams.push\_back(this->args[i]->calculate());

}

return StandardMethods::findFuncValue(standardFuncName, foundedParams);

}

// Finding the information about the function

map<string, FuncInfo\*> \*funcTable = Parser::getFuncInformation();

map<string, FuncInfo\*>::iterator it = funcTable->find(this->funcName);

if (it == funcTable->end())

return new VariableData();

// Running the function

if (it->second->getIsRunning()) {

return new VariableData(); // avoiding the recursion

}

else {

it->second->setIsRunning(true);

}

// Getting the local table and body of function

int tableIndex = it->second->getTableIndex();

FuncNameTable \*localTable = &((\*Parser::getFuncNameTables())[tableIndex]);

NTFunc \*ntFunc = it->second->getBody();

// Checking the type of parameters and adding them to the funcTable

vector<pair<Type, string>> ntFuncArgs = ntFunc->getParams();

if (this->args.size() != ntFuncArgs.size())

return new VariableData();

vector<pair<string, VariableData\*>> \*params = localTable->getParamTable();

for (int i = 0; i < this->args.size(); i++) {

VariableData \*currParam = this->args[i]->calculate();

// Checking the types

if (currParam->getType() != ntFuncArgs[i].first)

return new VariableData();

// Adding the value of parameter

params->push\_back(pair<string, VariableData\*>(ntFuncArgs[i].second, currParam));

}

// Switching the current table

\*Parser::getCurrTable() = tableIndex;

// Executing the body of function

vector<NTOperator\*> funcBlock = ntFunc->getBlock();

for (int i = 0; i < funcBlock.size(); i++) {

if (!funcBlock[i]->execute()) {

return new VariableData();

}

}

// Calculating the return expression

VariableData \*result = ntFunc->getResult()->calculate();

// Deleting the params and local vars, switching the current table to global

params->clear();

localTable->getLocalVarsTable()->clear();

\*Parser::getCurrTable() = -1;

return result;

}

VariableData\* NTBinaryOperation::calculate() {

VariableData \*oper1 = this->firstOperand->calculate();

VariableData \*oper2 = this->secondOperand->calculate();

if (oper1->getType() == ERROR || oper2->getType() == ERROR) {

return new VariableData();

}

switch (oper1->getType()) {

case NUM: {

switch (oper2->getType()) {

case NUM: {

switch (this->operType) {

case PLUS: {

return new VariableData(oper1->getNum() + oper2->getNum());

}

case MINUS: {

return new VariableData(oper1->getNum() - oper2->getNum());

}

case MULTIPLY: {

return new VariableData(oper1->getNum() \* oper2->getNum());

}

case DIVIDE: {

if (oper2->getNum() == 0)

return new VariableData();

return new VariableData(oper1->getNum() / oper2->getNum());

}

case POWER: {

return new VariableData(pow(oper1->getNum(), oper2->getNum()));

}

default: { return new VariableData(); }

}

}

case VECTOR: {

if (this->operType != MULTIPLY)

return new VariableData();

vector<double> resultVec = oper2->getVec();

for (int i = 0; i < resultVec.size(); i++)

resultVec[i] \*= oper1->getNum();

return new VariableData(resultVec);

}

case MATR: {

if (this->operType != MULTIPLY)

return new VariableData();

vector<vector<double>> resultMatr = oper2->getMatr();

if (!resultMatr.empty()) {

for (int i = 0; i < resultMatr.size(); i++) {

for (int j = 0; j < resultMatr[i].size(); j++) {

resultMatr[i][j] \*= oper1->getNum();

}

}

}

return new VariableData(resultMatr);

}

default: { return new VariableData(); }

}

}

case VECTOR: {

switch (oper2->getType()) {

case NUM: {

if (this->operType == MULTIPLY) {

vector<double> resultVec = oper1->getVec();

for (int i = 0; i < resultVec.size(); i++)

resultVec[i] \*= oper2->getNum();

return new VariableData(resultVec);

}

else if (this->operType == DIVIDE) {

if (oper2->getNum() == 0)

return new VariableData();

vector<double> resultVec = oper1->getVec();

for (int i = 0; i < resultVec.size(); i++)

resultVec[i] /= oper2->getNum();

return new VariableData(resultVec);

}

else {

return new VariableData();

}

}

case VECTOR: {

switch (this->operType) {

case PLUS: {

vector<double> addendum1 = oper1->getVec();

vector<double> addendum2 = oper2->getVec();

if (addendum1.size() != addendum2.size())

return new VariableData();

vector<double> resultVec;

for (int i = 0; i < addendum1.size(); i++) {

resultVec.push\_back(addendum1[i] + addendum2[i]);

}

return new VariableData(resultVec);

}

case MINUS: {

vector<double> addendum1 = oper1->getVec();

vector<double> addendum2 = oper2->getVec();

if (addendum1.size() != addendum2.size())

return new VariableData();

vector<double> resultVec;

for (int i = 0; i < addendum1.size(); i++) {

resultVec.push\_back(addendum1[i] - addendum2[i]);

}

return new VariableData(resultVec);

}

case MULTIPLY: {

vector<double> addendum1 = oper1->getVec();

vector<double> addendum2 = oper2->getVec();

if (addendum1.size() != addendum2.size())

return new VariableData();

double resultNum = 0;

for (int i = 0; i < addendum1.size(); i++) {

resultNum += addendum1[i] \* addendum2[i];

}

return new VariableData(resultNum);

}

default: { return new VariableData(); }

}

}

default: { return new VariableData(); }

}

}

case MATR: {

switch (oper2->getType()) {

case NUM: {

switch (this->operType) {

case MULTIPLY: {

vector<vector<double>> resultMatr = oper1->getMatr();

if (!resultMatr.empty()) {

for (int i = 0; i < resultMatr.size(); i++) {

for (int j = 0; j < resultMatr[i].size(); j++) {

resultMatr[i][j] \*= oper2->getNum();

}

}

}

return new VariableData(resultMatr);

}

case DIVIDE: {

if (oper2->getNum() == 0)

return new VariableData();

vector<vector<double>> resultMatr = oper1->getMatr();

if (!resultMatr.empty()) {

for (int i = 0; i < resultMatr.size(); i++) {

for (int j = 0; j < resultMatr[i].size(); i++) {

resultMatr[i][j] /= oper2->getNum();

}

}

}

return new VariableData(resultMatr);

}

case POWER: {

if (oper2->getNum() < 0)

return new VariableData();

vector<vector<double>> resultMatr = oper1->getMatr();

if (!resultMatr.empty()) {

// Checking if matrix is square

if (resultMatr.size() != resultMatr[0].size())

return new VariableData();

int powerInt = (int)oper2->getNum();

if (powerInt == 0) {

for (int i = 0; i < resultMatr.size(); i++) {

for (int j = 0; j < resultMatr[i].size(); j++) {

if (i == j)

resultMatr[i][j] = 1;

else

resultMatr[i][j] = 0;

}

}

return new VariableData(resultMatr);

}

if (powerInt == 1) {

return new VariableData(resultMatr);

}

for (int i = 0; i < powerInt; i++) {

resultMatr = multiplyMatrixes(resultMatr, resultMatr);

}

}

return new VariableData(resultMatr);

}

default: { return new VariableData(); }

}

}

case MATR: {

switch (this->operType) {

case PLUS: {

vector<vector<double>> addendum1 = oper1->getMatr();

vector<vector<double>> addendum2 = oper2->getMatr();

vector<vector<double>> resultMatr;

if (addendum1.empty() && addendum1.empty())

return new VariableData(resultMatr);

// Checking width and height

if (addendum1.size() != addendum2.size() || addendum1[0].size() != addendum2[0].size())

return new VariableData();

for (int i = 0; i < addendum1.size(); i++) {

vector<double> line;

for (int j = 0; j < addendum1[i].size(); j++) {

line.push\_back(addendum1[i][j] + addendum2[i][j]);

}

resultMatr.push\_back(line);

}

return new VariableData(resultMatr);

}

case MINUS: {

vector<vector<double>> addendum1 = oper1->getMatr();

vector<vector<double>> addendum2 = oper2->getMatr();

vector<vector<double>> resultMatr;

if (addendum1.empty() && addendum1.empty())

return new VariableData(resultMatr);

// Checking width and height

if (addendum1.size() != addendum2.size() || addendum1[0].size() != addendum2[0].size())

return new VariableData();

for (int i = 0; i < addendum1.size(); i++) {

vector<double> line;

for (int j = 0; j < addendum1[i].size(); j++) {

line.push\_back(addendum1[i][j] - addendum2[i][j]);

}

resultMatr.push\_back(line);

}

return new VariableData(resultMatr);

}

case MULTIPLY: {

vector<vector<double>> addendum1 = oper1->getMatr();

vector<vector<double>> addendum2 = oper2->getMatr();

vector<vector<double>> resultMatr;

if (addendum1.empty() && addendum1.empty())

return new VariableData(resultMatr);

// Checking width and height

if (addendum1[0].size() != addendum2.size())

return new VariableData();

resultMatr = multiplyMatrixes(addendum1, addendum2);

return new VariableData(resultMatr);

}

default: { return new VariableData(); }

}

}

default: { return new VariableData(); }

}

}

default: { return new VariableData(); }

}

return new VariableData();

}

VariableData\* NTUnaryMinus::calculate() {

VariableData \*operand = this->operand->calculate();

switch (operand->getType()) {

case NUM: {

return new VariableData(0. - operand->getNum());

}

case VECTOR: {

vector<double> operVector = operand->getVec();

for (int i = 0; i < operVector.size(); i++)

operVector[i] \*= -1.;

return new VariableData(operVector);

}

case MATR: {

vector<vector<double>> operMatr = operand->getMatr();

for (int i = 0; i < operMatr.size(); i++) {

for (int j = 0; j < operMatr[0].size(); j++) {

operMatr[i][j] \*= -1.;

}

}

return new VariableData(operMatr);

}

default: { return new VariableData(); }

}

return new VariableData();

}

VariableData\* NTElementOfVector::calculate() {

VariableData \*operVector = Parser::getValueByKee(this->name);

VariableData \*operIndex = this->index->calculate();

if (operVector->getType() != VECTOR || operIndex->getType() != NUM)

return new VariableData();

vector<double> \_operVector = operVector->getVec();

double \_operIndex = operIndex->getNum();

// Checking for correct indexes

if ((int)\_operIndex < 0 || (int)\_operIndex >= \_operVector.size())

return new VariableData();

return new VariableData(\_operVector[(int)\_operIndex]);

}

VariableData\* NTElementOfMatrix::calculate() {

VariableData \*operMatrix = Parser::getValueByKee(this->name);

VariableData \*operIndex1 = this->index1->calculate();

VariableData \*operIndex2 = this->index2->calculate();

if (operMatrix->getType() != MATR || operIndex1->getType() != NUM || operIndex2->getType() != NUM)

return new VariableData();

vector<vector<double>> \_operMatrix = operMatrix->getMatr();

double \_operIndex1 = operIndex1->getNum();

double \_operIndex2 = operIndex2->getNum();

// Checking for correct indexes

if ((int)\_operIndex1 < 0 || (int)\_operIndex1 >= \_operMatrix.size())

return new VariableData();

if ((int)\_operIndex2 < 0 || (int)\_operIndex2 >= \_operMatrix[0].size())

return new VariableData();

return new VariableData(\_operMatrix[(int)\_operIndex1][(int)\_operIndex2]);

}

VariableData\* NTVector::calculate() {

vector<double> result;

for (int i = 0; i < this->values.size(); i++) {

VariableData \*currElement = this->values[i]->calculate();

if (currElement->getType() != NUM)

return new VariableData();

result.push\_back(currElement->getNum());

}

return new VariableData(result);

}

VariableData\* NTMatrix::calculate() {

vector<vector<double>> result;

if (this->values.empty())

return new VariableData(result);

// All lines must have same lengths

int lineLength = this->values[0].size();

for (int i = 0; i < this->values.size(); i++) {

if (this->values[i].size() != lineLength)

return new VariableData();

vector<double> line;

for (int j = 0; j < lineLength; j++) {

VariableData \*currElement = this->values[i][j]->calculate();

if (currElement->getType() != NUM)

return new VariableData();

line.push\_back(currElement->getNum());

}

result.push\_back(line);

}

return new VariableData(result);

}

vector<vector<double>> multiplyMatrixes(vector<vector<double>> matr1, vector<vector<double>> matr2) {

vector<vector<double>> result;

if (matr1.empty() && matr2.empty())

return result;

vector<double> line;

for (int i = 0; i < matr1.size(); i++) {

line.clear();

for (int j = 0; j < matr2[0].size(); j++) {

double element = 0;

for (int k = 0; k < matr2.size(); k++) {

element += matr1[i][k] \* matr2[k][j];

}

line.push\_back(element);

}

result.push\_back(line);

}

return result;

}

**A.12 VariableData.h**

#pragma once

#include "globals.h"

#include "Type.h"

// Contains the type and value of the variable

class VariableData {

Type type; // NUM, VECTOR, MATR or ERROR

double resultNum;

vector<double> resultVec;

vector<vector<double>> resultMatr;

public:

VariableData(double \_number) : resultNum(\_number)

{ this->type = NUM; }

VariableData(vector<double> \_vector) : resultVec(\_vector)

{ this->type = VECTOR; }

VariableData(vector<vector<double>> \_matrix) : resultMatr(\_matrix)

{ this->type = MATR; }

VariableData()

{ this->type = ERROR; }

Type getType() const { return this->type; }

double getNum() const { return this->resultNum; }

vector<double> getVec() const { return this->resultVec; }

vector<vector<double>> getMatr() const { return this->resultMatr; }

void setType(Type \_type) { this->type = \_type; }

void setNum(double \_num) { this->resultNum = \_num; }

void setVec(vector<double> \_vec) { this->resultVec = \_vec; }

void setMatr(vector<vector<double>> \_matr) { this->resultMatr = \_matr; }

};

**A.13 FuncNameTable.h**

#pragma once

#include "globals.h"

#include "Type.h"

#include "VariableData.h"

class FuncNameTable {

vector<pair<string, VariableData\*>> paramTable; // Table of parameter names

map<string, VariableData\*> localVarsTable; // Table of local variable names

public:

bool addParam(string kee, VariableData \*value); // Add the parameter to the table of parameter names

// (false if already exists)

bool addLocalVar(string kee, VariableData \*value); // Add the variable to the table of local variable names

// (false if already exists)

VariableData\* getParam(string kee);

VariableData\* getLocalVar(string kee);

bool changeLocalVar(string kee, VariableData \*newData);

vector<pair<string, VariableData\*>>\* getParamTable();

map<string, VariableData\*>\* getLocalVarsTable();

};

**A.14 FuncNameTable.cpp**

#pragma once

#include "globals.h"

#include "Type.h"

#include "VariableData.h"

class FuncNameTable {

vector<pair<string, VariableData\*>> paramTable; // Table of parameter names

map<string, VariableData\*> localVarsTable; // Table of local variable names

public:

bool addParam(string kee, VariableData \*value); // Add the parameter to the table of parameter names

// (false if already exists)

bool addLocalVar(string kee, VariableData \*value); // Add the variable to the table of local variable names

// (false if already exists)

VariableData\* getParam(string kee);

VariableData\* getLocalVar(string kee);

bool changeLocalVar(string kee, VariableData \*newData);

vector<pair<string, VariableData\*>>\* getParamTable();

map<string, VariableData\*>\* getLocalVarsTable();

};

**A.15 FuncInfo.h**

#pragma once

#include "SyntaxTree.h"

class FuncInfo {

bool isRunning; // TRUE if function is running just now (necessary for forbidding the recursion)

int tableIndex; // The index of nametable

NTFunc \*body; // All information about the function (name, parameters, body and return expression)

public:

FuncInfo(bool \_isRunning, int \_tableIndex, NTFunc \*\_body)

: isRunning(\_isRunning), tableIndex(\_tableIndex), body(\_body) {}

void setIsRunning(bool \_isRunning) { this->isRunning = \_isRunning; }

void setTableIndex(int \_tableIndex) { this->tableIndex = \_tableIndex; }

void setBody(NTFunc \*\_body) { this->body = \_body; }

bool getIsRunning() { return this->isRunning; }

int getTableIndex() { return this->tableIndex; }

NTFunc\* getBody() { return this->body; }

};

**A.16 StandardMethods.h**

#pragma once

#include "globals.h"

#include "VariableData.h"

using namespace std;

enum STANDARD\_FUNC\_NAME {

DET,

TRIANGLE,

SOLVE,

TRANSPOSE,

INVERT,

NONE

};

class StandardMethods {

static vector<vector<double>> makeTriagleMatrix(vector<vector<double>> \_matr, bool &isSignChanged); // Finding the triangle matrix

static double standard\_Det(vector<vector<double>> \_matr); // Finding the determinate of matrix

static vector<double> standard\_Solve(vector<vector<double>> \_matr, vector<double> freeMembers); // Finding the solution of the system of equations

static vector<vector<double>> standard\_Transpose(vector<vector<double>> \_matr); // Transposing the matrix

static vector<vector<double>> standard\_Invert(vector<vector<double>> \_matr); // Inverting the matrix

public:

static STANDARD\_FUNC\_NAME checkFuncName(string \_name); // Checking if the name is standard function

static VariableData\* findFuncValue(STANDARD\_FUNC\_NAME \_name, vector<VariableData\*> \_params); // Finding the value of standard function

};

**A.17 StandardMethods.cpp**

#include "StandardMethods.h"

STANDARD\_FUNC\_NAME StandardMethods::checkFuncName(string \_name) {

if (\_name == "det") return DET;

if (\_name == "triangle") return TRIANGLE;

if (\_name == "solve") return SOLVE;

if (\_name == "transpose") return TRANSPOSE;

if (\_name == "invert") return INVERT;

return NONE;

}

VariableData\* StandardMethods::findFuncValue(STANDARD\_FUNC\_NAME \_name, vector<VariableData\*> \_params) {

switch (\_name) {

case DET: {

if (\_params.size() != 1) {

return new VariableData();

}

if (\_params[0]->getType() != MATR) {

return new VariableData();

}

return new VariableData(standard\_Det(\_params[0]->getMatr()));

}

case TRIANGLE: {

if (\_params.size() != 1) {

return new VariableData();

}

if (\_params[0]->getType() != MATR) {

return new VariableData();

}

bool temp;

return new VariableData(makeTriagleMatrix(\_params[0]->getMatr(), temp));

}

case SOLVE: {

if (\_params.size() != 2) {

return new VariableData();

}

if (\_params[0]->getType() != MATR) {

return new VariableData();

}

if (\_params[1]->getType() != VECTOR) {

return new VariableData();

}

return new VariableData(standard\_Solve(\_params[0]->getMatr(), \_params[1]->getVec()));

}

case TRANSPOSE: {

if (\_params.size() != 1) {

return new VariableData();

}

if (\_params[0]->getType() != MATR) {

return new VariableData();

}

return new VariableData(standard\_Transpose(\_params[0]->getMatr()));

}

case INVERT: {

if (\_params.size() != 1) {

return new VariableData();

}

if (\_params[0]->getType() != MATR) {

return new VariableData();

}

return new VariableData(standard\_Invert(\_params[0]->getMatr()));

}

}

return new VariableData();

}

vector<vector<double>> StandardMethods::makeTriagleMatrix(vector<vector<double>> \_matr, bool &isSignChanged) {

vector<vector<double>> result(\_matr);

if (result.size() <= 1)

return result;

for (int glob = 0; glob < result.size() - 1; glob++) {

// Finding the number of column with the biggest number in the line

double maxInLine = fabs(result[glob][0]);

double maxColumn = 0;

for (int j = 0; j < result[glob].size(); j++) {

if (fabs(result[glob][j]) > maxInLine) {

maxInLine = fabs(result[glob][j]);

maxColumn = j;

}

}

// Substraction the line

for (int i = glob + 1; i < result.size(); i++) {

double coef = result[i][maxColumn] / result[glob][maxColumn];

for (int j = glob; j < result.size(); j++) {

result[i][j] -= coef \* result[glob][j];

}

}

// Moving the column with the biggest number lefts

if (maxColumn != glob) {

for (int i = 0; i < result.size(); i++) {

double k = result[i][maxColumn];

result[i][maxColumn] = result[i][glob];

result[i][glob] = k;

}

isSignChanged = !isSignChanged;

}

}

return result;

}

double StandardMethods::standard\_Det(vector<vector<double>> \_matr) {

if (\_matr.empty()) {

return 0;

}

// Non-square matrix

if (\_matr.size() != \_matr[0].size()) {

return 0;

}

bool isSignChanged = false;

vector<vector<double>> triangleMatr = makeTriagleMatrix(\_matr, isSignChanged);

double result = 1;

for (int i = 0; i < triangleMatr.size(); i++) {

result \*= triangleMatr[i][i];

}

if (isSignChanged)

result \*= -1;

return result;

}

vector<double> StandardMethods::standard\_Solve(vector<vector<double>> \_matr, vector<double> freeMembers) {

vector<double> result;

if (\_matr.empty()) {

return result;

}

// Non-square matrix

if (\_matr.size() != \_matr[0].size()) {

return result;

}

// Degenerate matrix

if (standard\_Det(\_matr) == 0) {

return result;

}

// Sizes don't match

if (\_matr.size() != freeMembers.size()) {

return result;

}

vector<vector<double>> invertMatr = standard\_Invert(\_matr);

// Multiplying matrix by the vector

for (int i = 0; i < invertMatr.size(); i++) {

double sum = 0;

for (int j = 0; j < invertMatr.size(); j++) {

sum += invertMatr[i][j] \* freeMembers[j];

}

result.push\_back(sum);

}

return result;

}

vector<vector<double>> StandardMethods::standard\_Invert(vector<vector<double>> \_matr) {

vector<vector<double>> result;

if (\_matr.empty()) {

return result;

}

// Non-square matrix

if (\_matr.size() != \_matr[0].size()) {

return result;

}

// One-element matrix

if (\_matr.size() == 1) {

vector<double> line;

line.push\_back(1 / \_matr[0][0]);

result.push\_back(line);

return result;

}

double determinate = standard\_Det(\_matr);

if (determinate == 0) {

return result;

}

for (int i = 0; i < \_matr.size(); i++) {

vector<double> line;

for (int j = 0; j < \_matr[0].size(); j++) {

// Finding the minor of the element

vector<vector<double>> minor;

for (int \_i = 0; \_i < \_matr.size(); \_i++) {

if (\_i != i) {

vector<double> minorLine;

for (int \_j = 0; \_j < \_matr[0].size(); \_j++) {

if (\_j != j) {

minorLine.push\_back(\_matr[\_i][\_j]);

}

}

minor.push\_back(minorLine);

}

}

double minorDet = standard\_Det(minor);

// Changing the sign, if necessary

if ((i + j) % 2 != 0) {

minorDet = 0 - minorDet;

}

// Dividing by the determinate

minorDet /= determinate;

// Adding element to the result

line.push\_back(minorDet);

}

result.push\_back(line);

}

result = standard\_Transpose(result);

return result;

}

vector<vector<double>> StandardMethods::standard\_Transpose(vector<vector<double>> \_matr) {

vector<vector<double>> resultMatr;

if (\_matr.empty()) {

return resultMatr;

}

for (int j = 0; j < \_matr[0].size(); j++) {

vector<double> line;

for (int i = 0; i < \_matr.size(); i++) {

line.push\_back(\_matr[i][j]);

}

resultMatr.push\_back(line);

}

return resultMatr;

}

**A.18 main.cpp**

#include "globals.h"

#include "Lexer.h"

#include "Parser.h"

#include "Token.h"

#include "Type.h"

#include "FuncNameTable.h"

#include "SyntaxTree.h"

using namespace std;

int main(int argc, char\* argv[]) {

/\*int \_argc = 2;

char \*argl = "D:\\Program1.txt";

char\*\* \_argv = new char\*[2];

\_argv[1] = argl;\*/

// If received parameters

if (argc > 1) {

if (argc != 2) {

cout << "Too many arguments!" << endl;

cout << "Interpreting one file only" << endl;

exit(EXIT\_FAILURE);

}

// Reading code from file

ifstream inFile(argv[1]);

if (!inFile) {

cout << "File does not exist" << endl;

exit(EXIT\_FAILURE);

}

string input = "";

stringstream buffer;

buffer << inFile.rdbuf();

input = buffer.str();

inFile.close();

// Lexical analysis

Lexer lexer(input);

vector<Token> tokens = lexer.convert();

if (tokens.empty()) {

cout << "Lexical error!" << endl;

getchar();

exit(1);

}

// Syntactical analysis

Parser parser(tokens);

parser.analize();

}

else {

// REPL mode

cout << "Interpreter: REPL mode" << endl;

cout << "Enter your code below. In the end of output you'll see \"Done\"" << endl;

cout << "Enter \"finish\" to finish the work with interpreter" << endl;

Lexer lexer;

Parser parser;

// Reading the line

string codeLine;

getline(cin, codeLine);

while (codeLine != "finish") {

lexer.setInput(codeLine);

vector<Token> tokens = lexer.convert();

if (tokens.empty()) {

cout << "Lexical error!" << endl;

}

else {

parser.addTokens(tokens);

parser.analize();

cout << "Done" << endl;

}

// Reading the next line

getline(cin, codeLine);

}

}

return 0;

}

**Додаток Б  
до курсової роботи**

|  |  |
| --- | --- |
| на тему: | *Граматика розробленої мови програмування* |
|  |  |
|  | |

Київ 2017

ПРОПИСНИМИ літерами написано термінали

рядковими літерами - термінали

програма ::= блок

блок ::= оператор | блок оператор

оператор ::= оголошення | присвоєння | вивелення | умовний\_оператор |

процедура | підключення\_модуля | виклик\_функції **;**

оголошення ::= тип ІМ’Я **;**

тип ::= NUM | VECTOR | MATR

присвоєння ::= ІМ’Я = вираз **; |** ІМ’Я **[**вираз**]** = вираз**; |** ІМ’Я **[**вираз**] [**вираз**]** = вираз**;**

вираз::= **(**вираз**)** | ЧИСЛО | ІМ’Я | вираз операція вираз | - вираз | ІМ’Я **[**вираз**]** | ІМ’Я **[**вираз**] [**вираз**]**| виклик\_функції | вектор | матриця

операція ::= **+ | - | \* | / | ^**

виклик\_функції::= ІМ’Я **(** аргументи **)**

аргументи ::= аргументи **,** аргументи | вираз

вектор ::= **[** елемент **]**

елемент::= елемент**,** елемент | вираз

матриця ::= **[** рядок **]**

рядок::= рядок**,** рядок | **[**елемент**]**

виведення ::= CONSOLE **(**фраза**) ;**

фраза ::= ТЕКСТ | вираз

умовний\_оператор ::= IF **(** логічний\_ вираз **) {** блок **}**

логічний \_ вираз ::= вираз лог\_операція вираз

лог\_операція ::= **: | ! | > | <**

процедура ::= FUNC ІМ’Я **(** параметри **) {** блок повернення\_результату **}**

параметри ::= VOID | параметр

параметр ::= параметр **,** параметр | тип ІМ’Я

повернення\_результату::= RETURN вираз | RETURN VOID **;**

підключення\_модуля ::= INCLUDE ТЕКСТ **;**