**Національний технічний університет України**

**„Київський політехнічний інститут** **імені Ігоря Сікорського”**

**Кафедра автоматизації проектування енергетичних процесів і систем**

***КУРСОВА РОБОТА***

**з дисципліни: « Основи розробки трансляторів - 2 »**

**тема: « Розробка транслятора »**

|  |  |
| --- | --- |
| **Керівник Тарнавський Ю. А.** | **Виконав** |
| **Допущено до захисту** | **Студент ІІІ курсу** |
| **„\_\_\_”\_\_\_\_\_\_\_\_\_2019 р.** | **Групи ТВ-61** |
| **Захищено з оцінкою** | **залікова книжка ТВ-6133** |
| **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Калашников-Травін Владислав** |

***2019***

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Кафедра автоматизації проектування енергетичних процесів і систем

**КУРСОВА РОБОТА**

з дисципліни « Основи розробки трансляторів - 2 » (назва дисципліни)

на тему: « Розробка транслятора »

Студента групи ТВ-61

напряму підготовки ***6.050103 Програмна інженерія***

Калашникова-Травіна В. В.

(прізвище та ініціали)

Керівник: доц. Тарнавський Ю. А.

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_Оцінка: ECTS \_\_\_\_\_

Члени комісії \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали

Київ - 2019 рік

# Анотація

Дана курсова робота спрямована на дослідження галузі трансляторів та розробку власної мови програмування разом із супутнім до неї транслятором. Пояснювальна записка містить у собі повний опис мови програмування “SmallScript”, опис структури її транслятора і його компонентів, а також інструкцію користувача. Результатом виконання роботи має стати готовий продукт - програма зі зручним інтерфейсом користувача з можливістю створювати та/або редагувати код, компілювати його, переглядати його проміжне представлення, відслідковувати помилки компіляції та виконувати код. Реалізований код має бути відлагодженим та протестованим задля відсутності неочікуваного завершення та забезпечення стабільної роботи програми.

# Зміст

[Анотація 3](#_Toc18)

[Зміст 4](#_Toc19)

[1. Вступ 6](#_Toc20)

[2. Структура проекту 8](#_Toc21)

[3. Технічне завдання 10](#_Toc22)

[4. Граматика мови 11](#_Toc23)

[5. Структура транслятора 13](#_Toc24)

[5.1 Лексичний аналізатор 14](#_Toc25)

[5.2 Синтаксичний аналізатор 18](#_Toc26)

[5.3 Генерація польського інверсного запису 20](#_Toc27)

[5.3.1 Генерація ПОЛІЗу інверсного оператора 22](#_Toc28)

[5.3.2 Генерація ПОЛІЗу оператора циклу 23](#_Toc29)

[5.3.3 Генерація ПОЛІЗу операторів вводу/виводу 24](#_Toc30)

[5.4 Інтерпретація польського інверсного запису 25](#_Toc31)

[6. Інструкція користувача 27](#_Toc32)

[7. Висновок 31](#_Toc33)

[Список використаної літератури 32](#_Toc34)

[Додаток 1. Специфікація 33](#_Toc35)

[Додаток 2. Текст програмного модуля 35](#_Toc36)

[Додаток 3. Опис програмного модуля 41](#_Toc37)

**Національний технічний університет України**

**“Київський політехнічний інститут імені Ігоря Сікорського”**

Факультет \_ТЕПЛОЕНЕРГЕТИЧНИЙ\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

( повна назва )

Кафедра \_автоматизації проектування енергетичних процесів та систем\_\_\_\_\_\_\_\_\_\_

( повна назва )

Спеціальність \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(шифр і назва)

**З А В Д А Н Н Я**

**НА КУРСОВУ РОБОТУ СТУДЕНТУ**

Калашникову-Травіну Владиславу Володимировичу

(прізвище, ім’я, по батькові)

1.Тема роботи Розробка транслятора

керівник курсової роботи – доцент Тарнавський Ю. А.

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Строк подання роботи студентом – \_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. Вихідні дані до проекту (роботи): середовище розробки – JetBrains Rider 2019.1, мова програмування – C#

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) – лексичний аналізатор, синтаксичний аналізатор, генератор ПОЛІЗ, інтерпретатор ПОЛІЗ, графічний інтерфейс користувача

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапів виконання дипломного  проекту (роботи) | С Строк виконання етапів  проекту (роботи) | Примітка |
| 1. | Затвердження теми роботи |  |  |
| 2. | Вивчення та аналіз задачі |  |  |
| 3. | Розробка лексичного аналізатора |  |  |
| 4. | Розробка синтаксичного аналізатора |  |  |
| 5. | Переведення в ПОЛІЗ |  |  |
| 6. | Виконання ПОЛІЗ |  |  |
| 7. | Оформлення пояснювальної записки |  |  |

**Студент \_\_\_\_\_\_\_\_\_ Калашников-Травін В. В.**

( підпис ) (прізвище та ініціали)

**Керівник курсової роботи \_\_\_\_\_\_\_\_\_ Тарнавський Ю. А.**

( підпис ) (прізвище та ініціали)

# 1. Вступ

Транслятор - програма або комплекс програм, що виконують трансляцію з однієї мови програмування у іншу. Транслятори (компілятори) зустрічаються майже повсюди у сфері програмування. Оскільки процесор комп’ютера з машинно-орієнтованими кодами, що є занадто семантично незрозумілими для людей, було винайдено мови високого рівня, що є більш зрозумілими для читання та написання коду такою мовою. З ціллю перетворення мов високого рівня у машинні мови були розроблені транслятори. Також важливо розуміти, що транслятор не завжди має перекладати код мови високого рівня у код мови низького рівня, тобто транслятор може транслювати коди мов одного рівня.

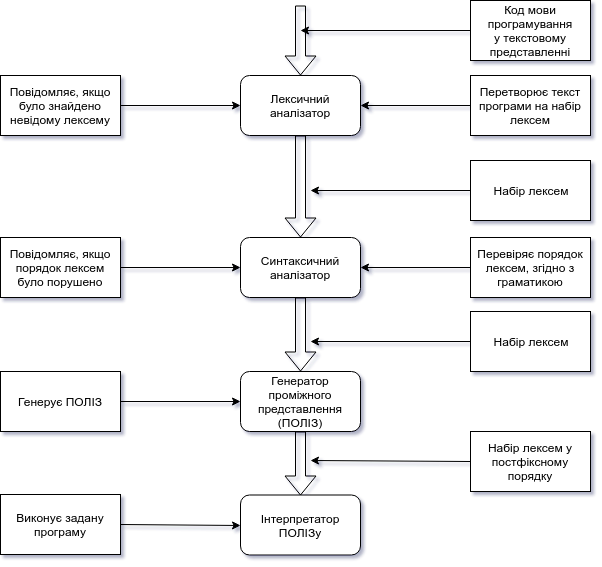
Трансляція виконується у декілька кроків. Деякі з них можуть об’єднуватися у один, а деякі можуть бути взагалі відсутні. Задля виконання кожного кроку, транслятор поділяють на декілька модулів, зокрема:

* лексичний аналізатор
* синтаксичний аналізатор
* генератор проміжного представлення
* оптимізатор
* генератор машинного коду

У контексті цієї курсової роботи розроблено наступні чотири модулі:

* лексичний аналізатор
* синтаксичний аналізатор
* генератор інверсного польського запису
* інтерпретатор інверсного польського запису

Далі у записці буде детально оглянуто кожен модуль транслятора.



Блок-схема 1. Процес трансляції коду

# 2. Структура проекту

Для розробки програмного продукту було обрано мову програмування C#. Ця мова надає широкі можливості для написання об’єктно-орієнтованого коду та є досить простою у використанні. Завдяки технології .NET Core 2.2, програми, написані цією мовою на одній платформі, можуть бути запущені на іншій платформі з мінімальними змінами (або повною їх відсутністю) в коді. Так, наприклад, основною платформою розробки є операційна система Manjaro Linux, але програмний продукт без труднощів та жодних змін можливо запустити на операційних системах Windows та Mac. Розробка буде проводитись в інтегрованому середовищі розробки JetBrains © Rider 2019.1.

Проект було поділено шість частин (модулів) та шістнадцять підпроектів. Кожна частина надає одну або кілька реалізацій відповідного їй кроку трансляції.

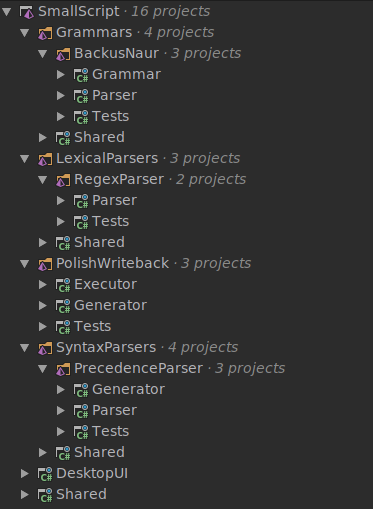


Рисунок 1. Структура рішення

Кожен підпроект має у супроводі проект з юніт-тестами. Юніт-тестування - процес у програмуванні, що забезпечує перевірку виконання окремих модулів (класів, їх нетривіальних методів) на коректність. Перед збіркою продукту необхідно запустити всі тести, щоб переконатись, що всі файли було скопійовано вірно, реалізації модулів виконують необхідні до специфікації дії.

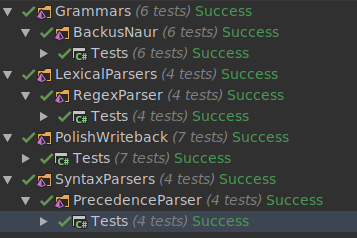


Рисунок 2. Успішний запуск юніт-тестів

Більша частина програмного продукту була розроблена за схемою TDD - Test Driven Development, тобто спочатку пишуться тести, що покривають майбутні реалізації, а потім вже самі реалізації. Це забезпечує швидкість та надійність розробки, просте тестування коду, як перед використанням, так і під час відлагодження.

# 3. Технічне завдання

**Задача:** розробити власну мову програмування та супутній до неї транслятор. Описати мову за граматикою Бекуса-Наура, виконати необхідні її перетворення (за потреби). Розробити транслятор із зручним користувацьким інтерфейсом, що дозволяє редагувати, компілювати та виконувати написаний код.

**Індивідуальне завдання:** мова програмування повинна містити в собі оператор присвоювання, оператори вводу та виводу, структура яких пропонується самим студентом, а також оператори умовного переходу та циклу, що надаються згідно з варіантом:

* умовний оператор

if <relation> then

<operators>

else

<operators>

fi

* оператор циклу

for @VAR = <math> by step to bound do

<operators>

rof

* цілі числа, математичні та логічні операції, піднесення до степені
* оператори вводу/виводу

stdin >> @VAR

stdout << @VAR

* лексичний аналізатор на основі розбору до роздільника
* синтаксичний аналізатор на основі таблиці простого передування
* роздільник - перенос лінії (\n, \r, \r\n в залежності від кодування файлу)

# 4. Граматика мови

Дана граматика є граматикою Бекуса-Наура та вже є зведеною до граматики простого передування, тобто не має конфліктів. Нетермінали огорнуті у лапки, термінали подаються у вигляді просто тексту, спеціальні символи починаються з ‘@’, альтернативи розділяються символом ‘|’.

Крім того, граматика потребує дотримання чіткого формату, оскільки вона є не тільки зручним для людини представленням, але й безпосередньо приймає участь у роботі транслятора. Для цього виділено окремий модуль “*SmallScript.Grammars.BackusNaur.Grammar*” для представлення граматики, а також “*SmallScript.Grammars.BackusNaur.Parser*” для розбору тексту граматики та подальшого її завантаження у оперативну пам’ять. Слід зазначити, що ці модулі реалізують загальний інтерфейс, що є безсумнівним плюсом в разі, якщо реалізацію граматики необхідно замінити. Інші модулі взаємодіють с граматикою виключно через інтерфейс, отже вони заздалегідь сумісні с майбутніми реалізаціями. Так, наприклад, можна реалізувати таку граматику, що буде зберігатися прямо у коді програми. Надана реалізація у проекті зчитує граматику з файлу.

<SYNTAX> ::= <IMPL\_LIST\_1>

<IMPL\_LIST> ::= <IMPL> <EOL> <IMPL\_LIST> | <IMPL> <EOL>

<IMPL> ::= <DECL> | <ASSIGN> | <FOR> | <COND\_IF> | <READ> | <WRITE>

<DECL> ::= declare @VAR = <T1>

<ASSIGN> ::= let @VAR = <T1>

<FOR> ::= for @VAR = <T1> by <T1> to <T1> do <IMPL\_LIST\_1> rof

<REL> ::= != | == | < | > | <= | >=

<BOOL> ::= <T1> <REL> <T1>

<COND\_IF> ::= if <BOOL> then <IMPL\_LIST\_1> else <IMPL\_LIST\_1> fi

<READ> ::= stdin >> @VAR

<WRITE> ::= stdout << <T1> | stdout << @STRING

<T> ::= <T> + <E1> | <T> - <E1> | <E1>

<E> ::= <E> \* <F1> | <E> / <F1> | <F1>

<F> ::= <F> \*\* <R> | <R>

<R> ::= ( <T1> ) | <K>

<K> ::= @VAR | @CONST

<IMPL\_LIST\_1> ::= <IMPL\_LIST>

<DECL\_LIST\_1> ::= <DECL\_LIST>

<T1> ::= <T>

<E1> ::= <E>

<F1> ::= <F>

<EOL> ::= \n | \r | \r\n

# 5. Структура транслятора

Як вже обумовлено, транслятор складається з трьох основних частин: лексичний аналізатор, синтаксичний аналізатор та генератор машинного коду. В залежності від того, яким чином представлення передається між модулями, транслятори поділяють на дві категорії: однопрохідні та багатопрохідні. Різниця між ними полягає в тому, яким чином транслятор зчитує вхідні дані - однопрохідний транслятор генерує вихідний код за одне послідовне зчитування вхідного коду; багатопрохідний зчитує код багаторазово. Реалізований у даній курсовій роботі транслятор є багатопрохідним, оскільки список лексем обробляється декілька разів у різних модулях транслятора.

У проекті за кожен крок відповідає окремий модуль. Кожен модуль реалізує певний інтерфейс, що зобов’язує реалізацію виконувати поставлену задачу. Результат роботи попереднього модуля використовується наступним модулем. Загалом, проект містить шість модулів, шістнадцять підпроектів, включаючи реалізації та тести. Підпроект “*SmallScript.DesktopUI*” містить лише графічний інтерфейс користувача та керує іншими модулями транслятора. Підпроект “*SmallScript.Shared*” містить класи та розширення, загальні для всіх інших модулів. Інші модулі буде детально розглянуто у наступних підрозділах.

## 5.1 Лексичний аналізатор

Лексичний аналіз (токенізація) - процес розбору вхідної послідовності символів, їх розпізнавання та поділ на групи - лексеми (токени). Граматика мови задає наперед визначений набір лексем, які можливо розпізнати у процесі токенізації. Якщо у вхідній послідовності зустрічається така послідовність, що не може бути розпізнана, аналізатор має повідомити про помилку та припинити аналіз. Зазвичай, поняття лексеми та токена є еквівалентними, але у більш складних лексичних аналізаторах на лексеми накладають додаткові властивості. Надалі, будемо вважати ці поняття рівнозначними.

У даній роботі за лексичний аналіз відповідає підпроект “*SmallScript.LexicalParsers.RegexParser.Parser*”. Принцип його роботи базується на розбитті вхідної послідовності на токени за допомогою регулярних виразів. На основі даної граматики будується регулярний вираз, що містить у собі всі ключові слова, роздільники та спеціальні символи. Після розбиття маємо “сирий” список токенів, тобто вони ще не є класифіковані. Для класифікації токенів знову використовуються регулярні вирази, тобто кожному класу токенів відповідає свій регулярний вираз. Якщо не було знайдено жодного збігу, аналізатор виділяє спеціальну лексему. Запропонована наступна класифікація лексем (відповідні класи позначені у дужках):

* ключове слово (KeywordToken)
* роздільник (DelimiterToken)
* константа (ConstantToken)
* змінна (VariableToken)
* нерозпізнана лексема (InvalidToken)

Всі класи реалізують загальний інтерфейс IToken, що містить у собі значення токену, його позицію у файлі та відповідний граматичний запис, що відповідає такому у граматиці. Також це дозволяє динамічно додавати класи лексем, не змінюючи вже існуючий код.

Результатом роботи лексичного аналізатора є результативний набір, що містить у собі статус (успіх або помилка), текстове повідомлення, набір токенів, помилку та її положення у вхідній послідовності (якщо наявна). Список можливих лексем наведено у наступній таблиці:

|  |  |  |  |
| --- | --- | --- | --- |
| Позначення в коді | Значення | Помітка | Приклад |
| String | @STRING | будь-які символи, огорнуті в подвійні лапки | “string”  “text” |
| Var | @VAR | змінна | var  arg |
| Const | @CONST | константа (ціле число) | 32  15 |
| BoolValue | @BOOL | логічне значення  (службова) | true  false |
| IntValue | @INT | (службова) |  |
| OperationDelimiter | <EOL> | кінець виразу | \n  \r  \r\n |
| Label | @LBL | мітка  (службова) |  |
| LabelDeclaration | @LBL\_DECL | об’ява мітки  (службова) |  |
| JumpByNotEquality | @JNE | перехід на мітку за нерівністю  (службова) |  |
| Jump | @JMP | безумовний перехід  (службова) |  |
| Declare | declare | об’ява змінної | declare var = 1 |
| Let | let | присвоєння значення | let var = 2 |
| OpenLoop | for | цикл | for i = 0 by 1 to 10 do  ...  rof |
| To | to | границя циклу |
| By | by | крок циклу |
| Do | do | початок списку операторів циклу |
| CloseLoop | rof | кінець списку операторів циклу |
| OpenCondition | if | початок умови | if ... then ... else ... fi |
| Then | then | початок списку операторів за правдивої умови |
| Else | else | початок списку операторів за хибної умови |
| CloseCondition | fi | кінець умовного блоку |
| StandartInput | stdin | оператор вводу | stdin >> var |
| StreamReading | >> | зчитування зі стандартного потоку вводу |
| StandartOutput | stdout | оператор виводу | stdout << var |
| StreamWriting | << | запис у стандартний потік виводу |
| Assign | = | знак присвоєння |  |
| OpenCurlyBrace | { | не використовується |  |
| CloseCurlyBrace | } | не використовується |  |
| OpenParenthesis | ( | відкриваюча дужка | (5 + 5) |
| CloseParenthesis | ) | закриваюча дужка |
| Less | < | знак менше | 1 < 2 |
| Greater | > | знак більше | 2 > 1 |
| GreaterEqual | >= | знак більше або рівне | 5 >= 5 |
| LessEqual | <= | знак менше або рівне | 4 <= 6 |
| Equal | == | знак рівності | 7 == 7 |
| NotEqual | != | знак нерівності | 7 != 8 |
| Plus | + | знак додавання | 1 + 2 |
| Minus | - | знак віднімання | 2 - 1 |
| Multiplication | \* | знак множення | 2 \* 2 |
| Division | / | знак ділення | 4 / 2 |
| Power | \*\* | знак піднесення до степені | 2 \*\* 2 |

Лексичний аналізатор може повернути помилку лише в одному випадку - якщо не вдалося виділити лексему, тобто не вдалося зіставити її із жодним правилом граматики. Наприклад:

declare i = &

for t ^ by 10 to 20 do

...

rof

В цьому випадку лексичний аналізатор повідомить про помилки, а помилкові лексеми будуть класифіковані як InvalidToken. В цьому випадку транслятор припиняє свою роботу і не передає результат синтаксичному аналізатору. Результатом компіляції буде наступне повідомлення:

[14:56:20] Unexpected token [InvalidToken] & at 1;1

[14:56:20] Unexpected token [InvalidToken] ^ at 2;1

В разі успішного лексичного аналізу повідомлення будуть відсутні, а робота транслятора продовжиться в нормальному режимі.

## 5.2 Синтаксичний аналізатор

Синтаксичний аналізатор (парсинг) - процес зіставлення отриманої послідовності лексем формальної мови з її граматикою. Зазвичай використовується або після лексичного аналізатора, або разом з ним. Перед синтаксичним аналізатором стоїть дві основні задачі - розпізнавати послідовності токенів та повідомляти про неправильну послідовність, вказувати на її положення у вхідній послідовності.

У курсовій роботі було реалізовано синтаксичний аналізатор на основі простого передування. Парсер простого передування є висхідним аналізатором, тобто необхідно від вхідного ланцюжка необхідно прийти до аксіоми мови. Розбір починається з набору терміналів, отриманого лексичним аналізатором, та закінчується виділенням аксіоми. Якщо не вдається виділити аксіому або будь-який проміжний ланцюжок не відповідає жодному з правил граматики, синтаксичний аналізатор повідомляє про помилку та припиняє роботу.

Аналіз проходить у два етапи - генерація таблиці передування на основі даної граматики та висхідний розбір з використанням отриманої таблиці та списку лексем. За генерацію таблиці відповідає підпроект “*SmallScript.SyntaxParsers.PrecedenceParser.Generator*”, за розбір - “*SmallScript.SyntaxParsers.PrecedenceParser.Parser*”.

Генерація таблиці передування проходить у декілька етапів. Задана граматика розбивається на пари - декартовий добуток множини всіх терміналів і нетерміналів із самою собою. Позначимо перший елемент пари “Left”, а другий “Right”. Далі реалізований наступний покроковий алгоритм:

1. знаходяться всі альтернативи правил, що містять однакові праві частини. Така граматика не є детермінованою, а отже немає сенсу продовжувати генерацію таблиці.
2. встановлюються всі відношення “=” між всіма парами, другий елемент якої слідує за першим. Важливо пам’ятати, що пара a = b не рівнозначна парі b = a. Наприклад, послідовність Z ::= A B C D утворює три пари: A = B, B = C, C = D.
3. Встановлюються множина First+ для кожного нетерміналу граматики. Така множина включає в себе перші входження кожної альтернативи цього нетерміналу.
4. Встановлюються множина Last+ для кожного нетерміналу граматики. Така множина включає в себе останні входження кожної альтернативи цього нетерміналу.
5. Знаходяться такі пари, що мають відношення “=”.
   1. Якщо обидва входження - нетермінали, тоді Last+(Left) > First+(Right)
   2. Якщо лише ліве входження - нетермінал, тоді Last+(Left) > Right
   3. Якщо лише праве входження - нетермінал, тоді Left < First+(Right)
6. Якщо хоча б одна пара містить більше одного відношення, то генератор повідомляє про помилку, адже граматика містить конфлікти. Частіше всього це зумовлено рекурсіями та їх викликами. В такому випадку виклик рекурсії замінюють на “аліас” - підміна рекурсивного виклику на нерекурсивний.

Згенерована таблиця використовується вже самим синтаксичним аналізатором простого передування. Висхідний розбір виконується скінченним автоматом. Принцип його роботи полягає в тому, що він на основі таблиці передування набирає основу та шукає її визначення серед всіх альтернатив граматики. Якщо альтернатива не знайдена, аналізатор припиняє свою роботу та повідомляє про помилку. Знайдена основа замінюється на правило граматики. Цей процес повторюється, доки не буде виділена аксіома. Наприклад, нехай у коді помилково двічі підряд було використане ключове слово “then”:

declare i = 0

if i == 0 then then

stdout << i

else

stdout << 0

fi

Тоді буде згенероване наступне повідомлення про помилку:

[15:35:37] Unexpected token “then” at 2;1

## 5.3 Генерація польського інверсного запису

Польський інверсний запис (далі ПОЛІЗ) - спеціальна форма запису виразів, у якій операнди розташовані перед операціями. Перевага ПОЛІЗу полягає в тому, що такий запис не містить у собі дужок пріоритету, що значно спрощує обробку виразу. Також важливою особливістю є те, що під час виконання будь-якої операції, її операнди вже відомо заздалегідь. Для зберігання операндів зазвичай використовують стек, по мірі зчитування вхідних лексем стек наповнюється / звільняється ними. ПОЛІЗ дуже просто розширювати, адже всі необхідні дані для будь-якої операції наперед відомі та зберігаються на стеку. Варто наголосити, що даний алгоритм не передбачає додаткових перевірок лексем та їх порядку, адже цю роботу вже виконали лексичний та синтаксичний аналізатори.

Алгоритм генерації ПОЛІЗу потребує введення додаткової властивості лексеми - пріоритету. Сам алгоритм передбачає такі правила:

* константи та змінні ніколи не потрапляють у стек, а одразу додаються до вихідної колекції лексем
* кожна операція має свій пріоритет, від нього залежить, куди саме варто додати лексему операції на поточному кроці. Якщо поточна операція має пріоритет більший або рівний пріоритету операції на вершині стеку, то поточна лексема додається до стеку. В зворотньому випадку зі стеку по черзі дістаються лексеми та додаються до вихідної колекції, доки не виконається попередня умова.
* якщо список вхідних лексем закінчився, а у стеку залишилися лексеми, вони по черзі додаються до вихідної колекції

Нижче приведено таблицю пріоритетів для мови *SmallScript.* Всі лексеми, що відсутні у таблиці ніколи не потрапляють у стек, а одразу додаються до вихідної колекції або просто вилучаються.

|  |  |
| --- | --- |
| Лексема | Пріоритет |
| OperationDelimiter | 75 |
| Declare | 100 |
| Let |
| StandartInput |
| StandartOutput |
| Assign | 200 |
| StreamReading |
| StreamWriting |
| Greater | 275 |
| GreaterEqual |
| Less |
| LessEqual |
| Equal |
| NotEqual |
| OpenParenthesis | 300 |
| CloseParenthesis |
| Plus | 400 |
| Minus |
| Multiplication | 401 |
| Division |
| Power | 402 |

Для того щоб у майбутньому було легко додавати нові операції та не змінювати вже існуючі пріоритети, було обрано трьохзначний порядок чисел пріоритетів з великою різницею між сусідніми пріоритетами.

За реалізацію генерації ПОЛІЗу відповідає підпроект “*SmallScript.PolishWriteback.Generator*”. Принцип його роботи полягає у декларації операцій як окремих класів. Коли поточною лексемою є операція, викликається метод Consume відповідного операції класу. Всі операції реалізують інтерфейс IOperation, що дає можливість додавати нові операції без необхідності редагування вже існуючого коду.

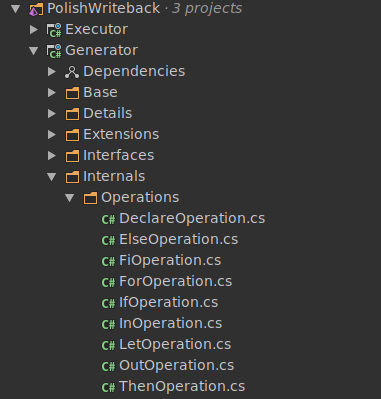


Рисунок 3. Декларації класів операцій

### 5.3.1 Генерація ПОЛІЗу інверсного оператора

Розглянемо детально генерацію умовного оператора. Згідно з граматикою мови, він має наступний вигляд:

if <relation> then

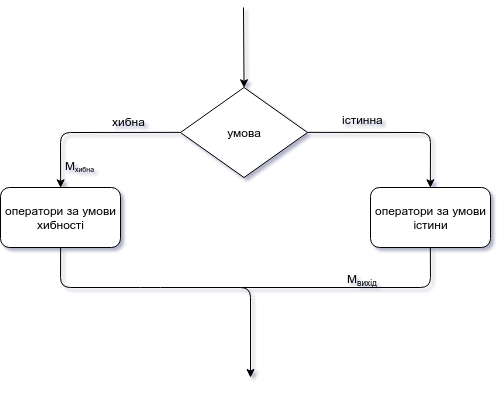
<on\_true>

else

<on\_false>

fi

Блок-схема виконання такої операції має наступний вигляд:



Блок-схема 2. Виконання умовної переходу

Блок-схема містить у собі додаткові позначки - мітки. Після виконання операторного блоку має слідувати перехід на мітку. Маємо відповідний блок-схемі ПОЛІЗ:

RPN(relation) Mfalse @JNE RPN(on\_true) Mout @JMP Mfalse: RPN(on\_false) Mout:

Тут RPN - ПОЛІЗ операнду, JNE - перехід за хибністю, JMP - безумовний перехід, M - цільова мітка переходу, M: - декларація мітки.

Алгоритм генерації ПОЛІЗу не передбачає ускладену генерацію, тобто додавання нових лексем, але реалізація операцій у коді дозволяє додавати семантичні підпрограми, всередині яких можна описати необхідні додаткові дії.

### 5.3.2 Генерація ПОЛІЗу оператора циклу

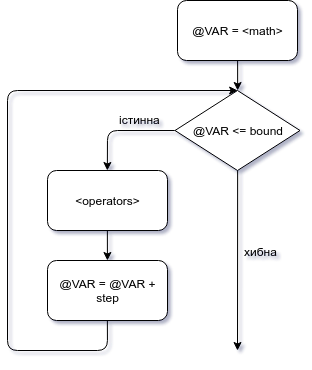
Розглянемо детально генерацію оператора циклу. Згідно з граматикою мови, він має наступний вигляд:

for @VAR = <math> by step to bound do

<operators>

rof

Блок-схема виконання такої операції має наступний вигляд:



Блок-схема 3. Виконання операції циклу

Використовуючи семантичні підпрограми, згенеруємо наступний ПОЛІЗ:

@VAR RPN(math) let Min: @VAR bound <= Mout @JNE RPN(operators) @VAR @VAR step + let Min @JMP Mout:

### 5.3.3 Генерація ПОЛІЗу операторів вводу/виводу

Розглянемо детально генерацію операторів вводу/виводу. Згідно з граматикою мови, вони мають наступний вигляд:

stdout << @VAR

stdin >> @VAR

Оскільки такі операції є одномісними, їх ПОЛІЗ є достатньо примітивним:

@VAR stdout

@VAR stdin

## 5.4 Інтерпретація польського інверсного запису

Інтерпретація - покомандне виконання програми. На відміну від компіляції, інтерпретація не потребує генерації машинного коду, достатньо лише проміжного представлення програми.

Проміжним представленням програми у даній роботі є польський інверсний запис. Інтерпретатор зчитує лексеми, заздалегідь впорядковані генератором ПОЛІЗу і якщо зустрінута лексема є командою (або оператором), виконується її семантична підпрограма.

За реалізацію інтерпретатора відповідає підпроект “*SmallScript.PolishWriteback.Executor*”. Принцип його роботи майже ідентичний з генератором ПОЛІЗу - тут також використовується стек, всі оператори представляються у вигляді окремих класів. Кожен оператор реалізує інтерфейс IOperator, що надає можливість додавати нові оператори без змін в існуючому коді. Крім того, цей модуль працює з потоками вводу/виводу також через інтерфейси, що надає можливість динамічно змінювати стандартні потоки вводу/виводу.

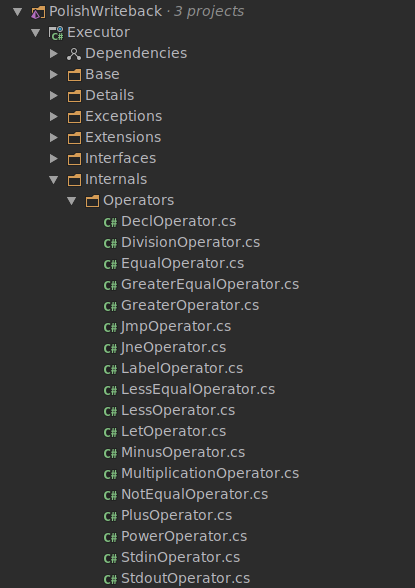


Рисунок 4. Декларації класів операторів

Принцип інтерпретації ПОЛІЗу у даній роботі впроваджує наступне правило виконання: якщо зустрінута лексема є оператором, необхідно викликати метод Execute класу, що відповідає оператору лексеми. Цей метод має доступ до стеку та може дістати звідти необхідні йому аргументи. Якщо ж лексема не є оператором, її необхідно додати на вершину стека.

Варто зазначити, що цей підпроект декларує два додаткових типи токенів: BoolValueToken та IntValueToken. Вони є службовими та призначені для зберігання тимчасових логічних та цілочислових значень. Вони реалізують інтерфейс IToken, а отже робота з ними відбувається так само, як і з іншими токенами.

# 6. Інструкція користувача

Кінцевий програмний продукт являє собою графічний додаток, що дозволяє редагувати, компілювати та виконувати код. Крім того, продукт підтримує роботу з файлами, а саме: зчитування, запис та збереження. Основне вікно додатка складається з п’яти вкладок:

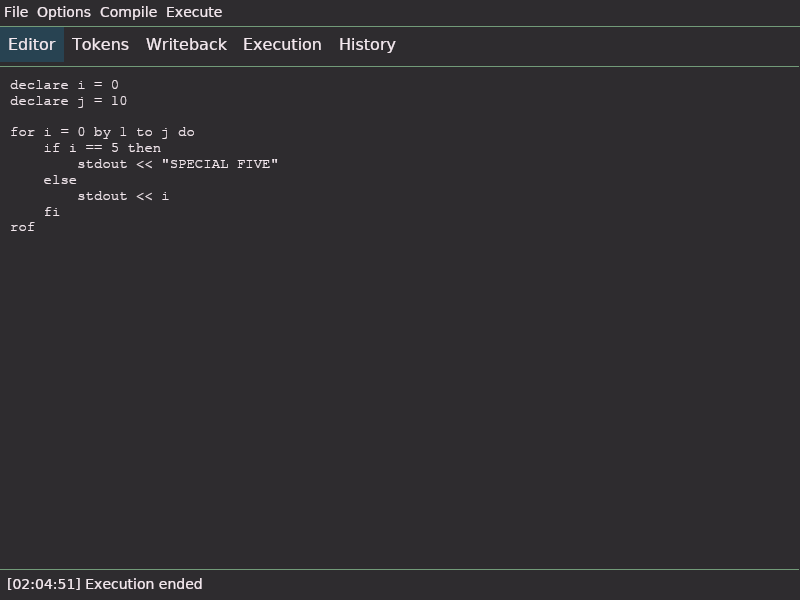


Рисунок 5. Editor - область редагування коду

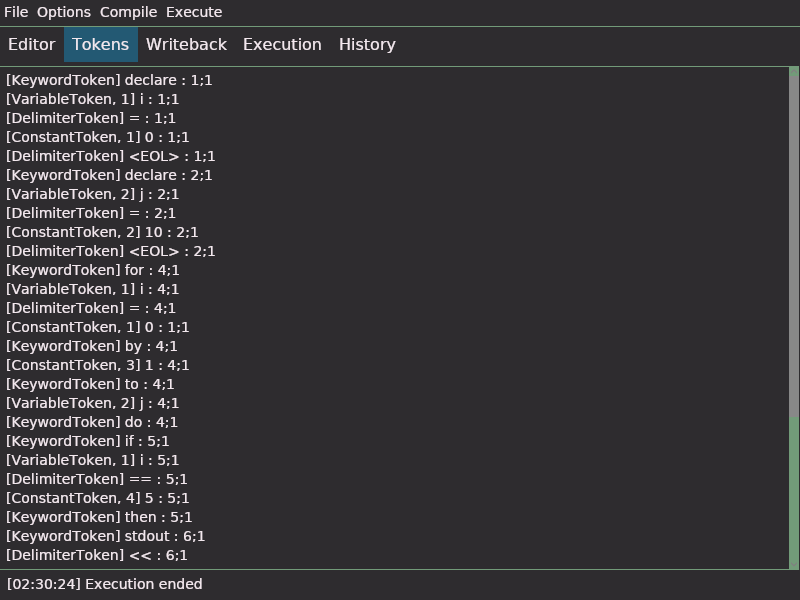


Рисунок 6. Tokens - лістинг лексем, отриманих під час лексичного аналізу

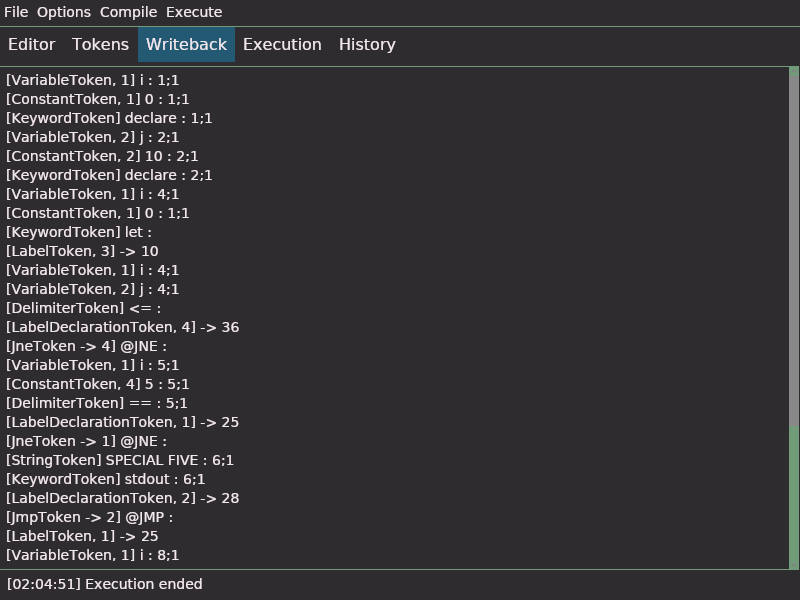


Рисунок 7. Writeback - лістинг лексем, отриманих під час генерації ПОЛІЗу

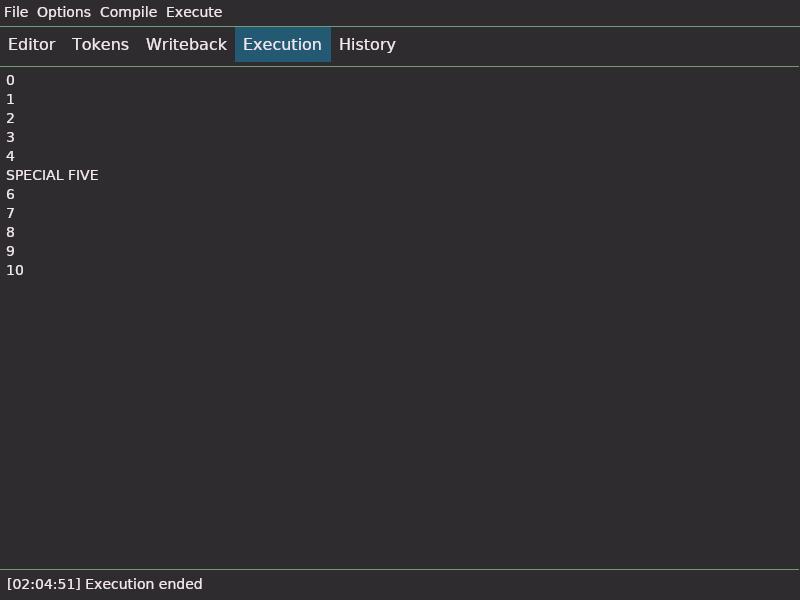


Рисунок 8. Execution - область виконання програми (емуляція вводу/виводу)

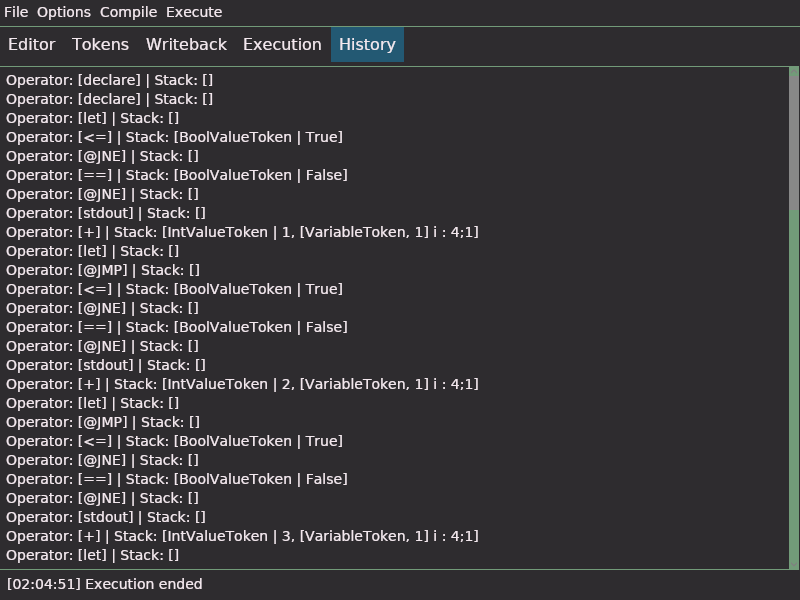


Рисунок 9. History - лістинг історії виконання обробки коду різними модулями програми

Вкладка *File* містить необхідні кнопки для роботи з файлами.

Вкладка *Options* містить параметри редактору коду, зокрема розмір шрифту.

Кнопка *Compile* виконує компіляцію: лексичний та синтаксичний аналіз.

Кнопка *Execute* виконує компіляцію, генерацію та виконання ПОЛІЗу.

У нижній частині графічного інтерфейсу розташована стрічка статусу - саме сюди виводяться всі інформаційні повідомлення та повідомлення про помилки. Для зручності, кожне повідомлення помічене поточним часом.

# 7. Висновок

У ході виконання даної курсової роботи було розроблено готовий продукт - транслятор зі зручним користувацьким інтерфейсом. Було вдосконалено навички розробки трансляторів, була розроблена власна мова, поліпшене розуміння різних видів аналізу та генерації. Крім того, було поглиблено знання розробки графічних інтерфейсів. Була досягнута повна незалежність реалізацій модулів однієї від інших.

Готовий продукт містить у собі:

* інтерфейси лексичного аналізатора, його реалізація методом розбору до роздільника, покриття юніт-тестами
* інтерфейси синтаксичного аналізатора, його реалізація методом простого передування, покриття юніт-тестами
* генератор польського інверсного запису, покриття юніт-тестами
* інтерпретатор польського інверсного запису
* графічний інтерфейс користувача

Крім того, було вдосконалено володіння інтегрованим середовищем розробки JetBrains © Rider, що значно пришвидшить темпи розробки майбутніх проектів.

# Список використаної літератури

1. Медведєва В. М. Транслятори: внутрішнє подання програм та інтерпретація / В. М. Медведєва, В. А. Третяк. - Бібліографія, 2015. - 144 с.
2. Медведєва, В.М. Транслятори: лексичний та синтаксичний аналізатори : Навчальний посібник / В.М. Медведєва, В.А. Третяк. - Політехніка, 2012. – 148 с.
3. Карпов Ю. Теория и технология программирования. Основы построения трансляторов / Ю. Карпов - БХВ-Петербург, 2005 - 272 с.
4. Компаниец Р. Системное программирование: Основы построения трансляторов / Р. Компаниец, Е. Маньков, Н. Филатов. - Мир Книг, 2004. - 256 с.
5. Грис Д. Построение компиляторов для цифровых вычислительных машин / Д. Грис - Мир, 1975. - 545 с.

# **Додаток 1**

**Специфікація**

**УКР.НТУУ «КПІ»\_ТЕФ\_АПЕПС\_ТВ\_61**

**Листів 2**

|  |  |
| --- | --- |
| Позначення | Найменування |
| Документація | |
| doc.docx | Пояснювальна записка |
| Програмний комплекс | |
| SmallScript.LexicalParsers.RegexParser | Лексичний аналізатор |
| SmallScript.Grammars.BackusNaur | Цифрове представлення граматики |
| SmallScript.SyntaxParsers.PrecedenceParser | Синтаксичний аналізатор |
| SmallScript.PolishWriteback.Generator | Генератор ПОЛІЗу |
| SmallScript.PolishWriteback.Executor | Інтерпретатор ПОЛІЗу |
| SmallScript.DesktopUI | Графічний інтерфейс користувача |

# **Додаток 2**

**Текст програмного модуля “Інтерпретатор ПОЛІЗу”**

**УКР.НТУУ «КПІ»\_ТЕФ\_АПЕПС\_ТВ\_61**

**Листів 5**

**using System;**

**using SmallScript.Grammars.Shared.Interfaces;**

**using SmallScript.PolishWriteback.Executor.Details;**

**using SmallScript.PolishWriteback.Executor.Interfaces;**

**using SmallScript.PolishWriteback.Executor.Internals;**

**namespace SmallScript.PolishWriteback.Executor.Base**

**{**

**internal abstract class OperatorBase : IOperator**

**{**

**public abstract IGrammarEntry GrammarEntry { get; }**

**public event EventHandler<HistoryPoint> OnExecution;**

**public void Execute(Runtime runtime)**

**{**

**ExecuteImpl(runtime);**

**var args = new HistoryPoint(GrammarEntry, runtime.Stack);**

**OnExecution?.Invoke(this, args);**

**}**

**protected abstract void ExecuteImpl(Runtime runtime);**

**}**

**}**

**using System.Collections.Generic;**

**using System.Linq;**

**using System.Runtime.CompilerServices;**

**using SmallScript.Grammars.Shared.Interfaces;**

**using SmallScript.LexicalParsers.Shared.Interfaces;**

**using SmallScript.PolishWriteback.Executor.Exceptions;**

**using SmallScript.PolishWriteback.Executor.Interfaces;**

**using SmallScript.PolishWriteback.Executor.Internals;**

**using SmallScript.PolishWriteback.Generator.Details;**

**using SmallScript.Shared.Details.Auxiliary;**

**[assembly: InternalsVisibleTo("DynamicProxyGenAssembly2")]**

**[assembly: InternalsVisibleTo("SmallScript.PolishWriteback.Tests")]**

**namespace SmallScript.PolishWriteback.Executor.Details**

**{**

**public class WritebackExecutor**

**{**

**private readonly IDictionary<IGrammarEntry, IOperator> \_operators;**

**private IInput  \_input;**

**private IOutput \_output;**

**public IInput Input**

**{**

**get => \_input;**

**set => \_input = Require.NotNull(value, nameof(value));**

**}**

**public IOutput Output**

**{**

**get => \_output;**

**set => \_output = Require.NotNull(value, nameof(value));**

**}**

**private readonly IList<HistoryPoint> \_history;**

**public WritebackExecutor(IInput input, IOutput output)**

**{**

**\_history = new List<HistoryPoint>();**

**\_operators = new OperatorProvider().Operators**

**.ToDictionary(o =>**

**{**

**o.OnExecution += (sender, e) => \_history.Add(e);**

**return o.GrammarEntry;**

**});**

**Input  = input;**

**Output = output;**

**}**

**public WritebackExecutionResult Execute(IEnumerable<IToken> tokens)**

**{**

**\_history.Clear();**

**var runtime = BuildRuntime(tokens);**

**try**

**{**

**Execute(runtime);**

**return WritebackExecutionResult.FromHistory(\_history);**

**}**

**catch (RuntimeException exception)**

**{**

**return WritebackExecutionResult.FromException(exception, \_history);**

**}**

**}**

**private Runtime BuildRuntime(IEnumerable<IToken> tokens)**

**{**

**return Runtime.Builder**

**.UseIterator(new TokenIterator(tokens))**

**.UseInput(\_input)**

**.UseOutput(\_output)**

**.UseStack(new Stack<IToken>())**

**.UseVariableTable(new VariablesData())**

**.Build();**

**}**

**private void Execute(Runtime runtime)**

**{**

**var iterator = runtime.Iterator;**

**var stack    = runtime.Stack;**

**while (iterator.IsValid)**

**{**

**var token = iterator.Current;**

**if (IsOperator(token, out var @operator))**

**{**

**@operator.Execute(runtime);**

**}**

**else**

**{**

**stack.Push(token);**

**}**

**iterator.MoveNext();**

**}**

**}**

**private bool IsOperator(IToken token, out IOperator @operator)**

**{**

**return \_operators.TryGetValue(token.GrammarEntry, out @operator);**

**}**

**}**

**}**

**using System;**

**using System.Collections.Generic;**

**namespace SmallScript.PolishWriteback.Executor.Details**

**{**

**public sealed class WritebackExecutionResult**

**{**

**public bool   Ok      { get; private set; }**

**public string Message { get; private set; }**

**public IEnumerable<HistoryPoint> History { get; private set; }**

**private WritebackExecutionResult()**

**{**

**}**

**public static WritebackExecutionResult FromHistory(IEnumerable<HistoryPoint> history)**

**{**

**return new WritebackExecutionResult**

**{**

**Ok      = true,**

**Message = "Ok",**

**History = history**

**};**

**}**

**public static WritebackExecutionResult FromException(Exception exception, IEnumerable<HistoryPoint> history)**

**{**

**return new WritebackExecutionResult**

**{**

**Ok      = false,**

**Message = exception.Message,**

**History = history**

**};**

**}**

**}**

**}**

**using System;**

**using SmallScript.Grammars.Shared.Interfaces;**

**using SmallScript.PolishWriteback.Executor.Details;**

**using SmallScript.PolishWriteback.Executor.Internals;**

**namespace SmallScript.PolishWriteback.Executor.Interfaces**

**{**

**internal interface IOperator**

**{**

**IGrammarEntry GrammarEntry { get; }**

**void Execute(Runtime runtime);**

**event EventHandler<HistoryPoint> OnExecution;**

**}**

**}**

**using System.Collections.Generic;**

**using SmallScript.LexicalParsers.Shared.Interfaces;**

**using SmallScript.PolishWriteback.Executor.Interfaces;**

**using SmallScript.PolishWriteback.Generator.Details;**

**using SmallScript.Shared.Details.Auxiliary;**

**namespace SmallScript.PolishWriteback.Executor.Internals**

**{**

**internal class Runtime**

**{**

**public Stack<IToken> Stack    { get; private set; }**

**public TokenIterator Iterator { get; private set; }**

**public IInput        Input     { get; private set; }**

**public IOutput       Output    { get; private set; }**

**public VariablesData Variables { get; private set; }**

**public static RuntimeBuilder Builder => new RuntimeBuilder();**

**internal class RuntimeBuilder**

**{**

**private Runtime \_instance;**

**public RuntimeBuilder()**

**{**

**\_instance = new Runtime();**

**}**

**public RuntimeBuilder UseIterator(TokenIterator iterator)**

**{**

**\_instance.Iterator = iterator;**

**return this;**

**}**

**public RuntimeBuilder UseStack(Stack<IToken> stack)**

**{**

**\_instance.Stack = stack;**

**return this;**

**}**

**public RuntimeBuilder UseInput(IInput input)**

**{**

**\_instance.Input = input;**

**return this;**

**}**

**public RuntimeBuilder UseOutput(IOutput output)**

**{**

**\_instance.Output = output;**

**return this;**

**}**

**public RuntimeBuilder UseVariableTable(VariablesData variables)**

**{**

**\_instance.Variables = variables;**

**return this;**

**}**

**public Runtime Build()**

**{**

**var instance = \_instance;**

**\_instance = new Runtime();**

**return instance;**

**}**

**}**

**}**

**}**

# **Додаток 3**

**Опис програмного модуля**

**УКР.НТУУ «КПІ»\_ТЕФ\_АПЕПС\_ТВ\_61**

**Листів 2**

Програмний модуль “*SmallScript.PolishWriteback.Executor*” відповідає за виконання ПОЛІЗу. Модуль складається з множини класів, основними з них є:

* WritebackExecutor - основний клас модуля, запускає ітерацію по колекції токенів та викликає необхідні оператори
* WritebackExecutorResult - представляє результат виконання роботи користувацької програми
* IOperator - надає загальний інтерфейс для всіх операторів мови
* IInput - надає інтерфейс для вводу даних
* IOutput - надає інтерфейс для виводу даних
* BoolValueToken - службовий токен, використовується для зберігання булевих значень під час виконання користувацької програми
* IntValueToken - службовий токен, використовується для зберігання цілочислових значень під час виконання користувацької програми

Модуль не є прив’язаним до графічного інтерфейсу та може бути використаним як окрема бібліотека класів.

Модуль розроблено мовою C# з використанням технології NET Core. Для його використання необхідно встановити dotnet-sdk, dotnet-host та dotnet-runtime версії не нижче 2.2.