НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«Київський політехнічний інститут імені Ігоря Сікорського»

**КУРСОВА РОБОТА**

**з теми**

**“Основи розробки трансляторів - 2”**

Виконав студент ІІІ курсу ТЕФ

Кафедри АПЕПС

Групи ТВ-61

Калашников-Травін Владислав Володимирович

Перевірив: доцент Тарнавсьий Ю. А.

Оцінка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Київ - 2019

# 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 відлагоджений та протестований задля відсутності неочікуваного завершення та забезпечення стабільної роботи програми.

# ЗМІСТ

[АНОТАЦІЯ 2](#_Toc1)

[ЗМІСТ 3](#_Toc2)

[1. ВСТУП 4](#_Toc3)

[2. СТРУКТУРА ПРОЕКТУ 5](#_Toc4)

[3. ТЕХНІЧНЕ ЗАВДАННЯ 7](#_Toc5)

[4. ГРАМАТИКА МОВИ 8](#_Toc6)

[5. СТРУКТУРА ТРАНСЛЯТОРА 9](#_Toc7)

[5.1 ЛЕКСИЧНИЙ АНАЛІЗАТОР 10](#_Toc8)

[5.2 СИНТАКСИЧНИЙ АНАЛІЗАТОР 11](#_Toc9)

[5.3 ГЕНЕРАЦІЯ ПОЛЬСЬКОГО ІНВЕРСНОГО ЗАПИСУ 12](#_Toc10)

[5.4 ІНТЕРПРЕТАЦІЯ ПОЛЬСЬКОГО ІНВЕРСНОГО ЗАПИСУ 13](#_Toc11)

[6. ІНСТРУКЦІЯ КОРИСТУВАЧА 14](#_Toc12)

[7. ВИСНОВОК 15](#_Toc13)

[СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 16](#_Toc14)

# 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.

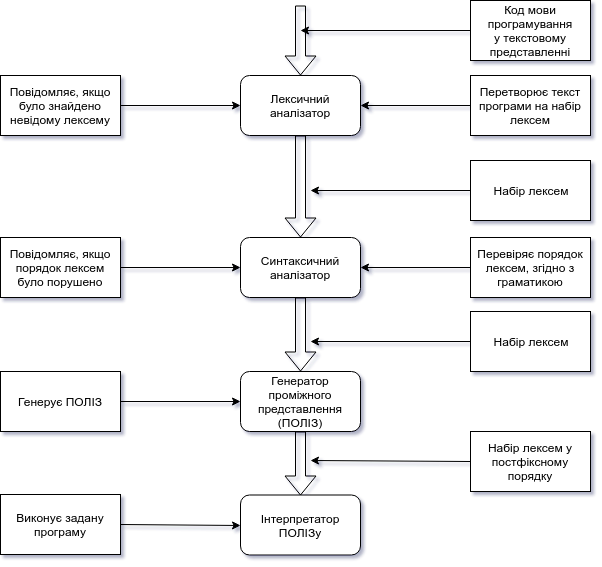


Схема 1. Процес трансляції коду

# 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, програми, написані цією мовою на одній платформі, можуть бути запущені на іншій платформі з мінімальними змінами (або повною їх відсутністю) в коді. Так, наприклад, поточною основною платформою розробки є операційна система Manjaro Linux, але програмний продукт без труднощів та змін можливо запустити на операційних системах Windows та Mac.

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

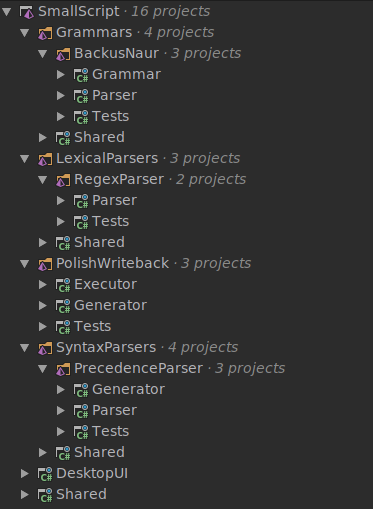


Рисунок 1. Структура рішення

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.

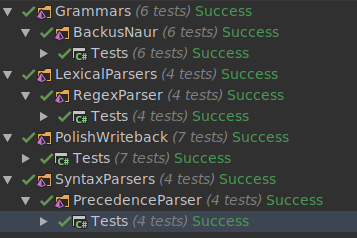


Рисунок 2. Успішний запуск юніт-тестів

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 \n

# 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 “Grammars.BackusNaur.Grammar” для представлення граматики, а також “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

## 5.1 ЛЕКСИЧНИЙ АНАЛІЗАТОР

## 5.2 СИНТАКСИЧНИЙ АНАЛІЗАТОР

## 5.3 ГЕНЕРАЦІЯ ПОЛЬСЬКОГО ІНВЕРСНОГО ЗАПИСУ

## 5.4 ІНТЕРПРЕТАЦІЯ ПОЛЬСЬКОГО ІНВЕРСНОГО ЗАПИСУ

# 6. ІНСТРУКЦІЯ КОРИСТУВАЧА

# 7. ВИСНОВОК

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ