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.

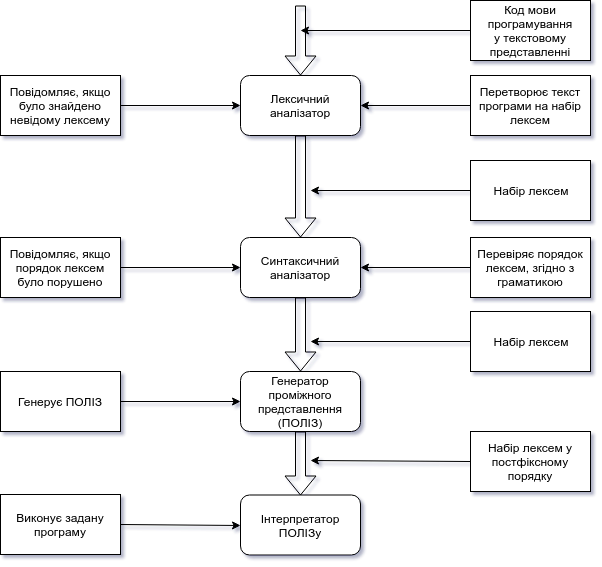


Схема 1. Процес трансляції коду

# 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.

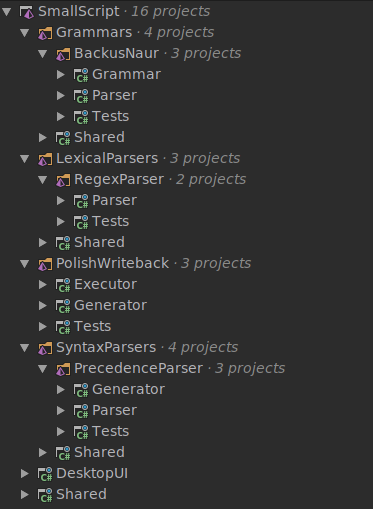


Рисунок 1. Структура рішення

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.

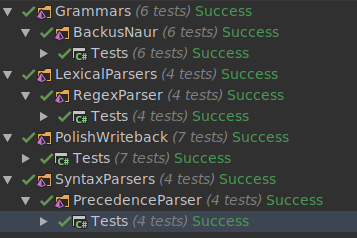


Рисунок 2. Успішний запуск юніт-тестів

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 - 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)
* невідома 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).

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

Таблиця 1. Список визначених лексем

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

Приклад 1. Програмний код з помилками

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

Приклад 2. Повідомлення про лексичні помилки

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.

Генерація таблиці передування проходить у декілька етапів. Задана граматика розбивається на пари - декартовий добуток множини всіх терміналів і нетерміналів із самою собою. 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.
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.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| a | b | c | d | e | f | g |
| < | = | = | = | = | = | > |

Приклад 3. Знайдена основа (**bcdef**) на базі простого передування

declare i = 0

if i == 0 then then

stdout << i

else

stdout << 0

fi

Приклад 4. Код, в якому помилково двічі використано ключове слово “then”

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

Приклад 5. Повідомлення про помилку в ході синтаксичного аналізу

## 5.3 ГЕНЕРАЦІЯ ПОЛЬСЬКОГО ІНВЕРСНОГО ЗАПИСУ

## 5.4 ІНТЕРПРЕТАЦІЯ ПОЛЬСЬКОГО ІНВЕРСНОГО ЗАПИСУ

# 6. ІНСТРУКЦІЯ КОРИСТУВАЧА

# 7. ВИСНОВОК

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ