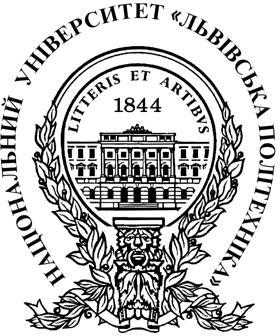
Міністерство освіти і науки України Національний університет "Львівська Політехніка"

Кафедра ЕОМ



## Пояснювальна записка

до курсового проєкту "СИСТЕМНЕ ПРОГРАМУВАННЯ"

на тему: “РОЗРОБКА СИСТЕМНИХ ПРОГРАМНИХ МОДУЛІВ ТА КОМПОНЕНТ СИСТЕМ ПРОГРАМУВАННЯ”

Індивідуальне завдання

“РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ”

Виконав студент групи КІ-307:

Затварницький Віталій

Перевірив:

Козак Назар

Львів 2025

# ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ

1. Цільова мова транслятора – мова програмування С або асемблер для 32/64 розрядного процесора.
2. Для отримання виконуваного файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio або будь-яким іншим.
3. Мова розробки транслятора: C/C++.
4. Реалізувати графічну оболонку або інтерфейс з командного рядка.
5. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.
6. На виході розробленого транслятора мають створюватись такі файли:

*файл з лексемами;*

*файл з повідомленнями про помилки (або про їх відсутність); файл на мові С або асемблера;*

*об’єктний файл; виконуваний файл.*

1. Назва вхідної мови програмування утворюється від першої букви у прізвищі студента та останніх двох цифр номера його варіанту. Саме таке розширення повинні мати текстові файли, написані на цій мові програмування.

**Деталізація завдання на проєктування:**

1. В кожному завданні передбачається блок оголошення змінних; змінні зберігають значення цілих чисел і, в залежності від варіанту, можуть бути 16/32 розрядними. За потребою можна реалізувати логічний тип даних.
2. Необхідно реалізувати арифметичні операції — додавання, віднімання, множення, ділення, залишок від ділення; операції порівняння — перевірка на рівність і нерівність, більше і менше; логічні операції — заперечення, “логічне І” і “логічне АБО”.

Пріоритет операцій наступний — круглі дужки (), логічне заперечення, мультиплікативні (множення, ділення, залишок від ділення), адитивні (додавання, віднімання), відношення (більше, менше), перевірка на рівність і нерівність, логічне І, логічне АБО.

1. За допомогою оператора вводу можна зчитати з клавіатури значення змінної; за допомогою оператора виводу можна вивести на екран значення змінної, виразу чи цілої константи.
2. В кожному завданні обов’язковим є оператор присвоєння, за допомогою якого можна реалізувати обчислення виразів з використанням заданих операцій і операцій круглі дужки (). У якості операндів можуть бути цілі константи, змінні, значення виразу.
3. В кожному завданні обов’язковим є оператор типу “блок” (вкладеність операторів), в якому мають бути вирази з тілом типу програми.
4. Необхідно реалізувати синтаксис вихідної мови, забезпечити реалізацію обчислення значень змінних, написати алгоритм з розгалуженням і циклічних обчислень.
5. Оператори різного виду допускаються і в будь-якій послідовності.
6. Для перевірки роботи розробленого транслятора необхідно написати три тестові програми на вихідній мові програмування.

# АНОТАЦІЯ

У даному курсовому проекті розроблено програмне забезпечення – транслятор з вхідної мови програмування.

Для реалізації транслятора визначено граматику вхідної мови програмування у термінах розширеної нотації Бекуса-Наура.

Реалізовано лексичний, синтаксичний, семантичний аналізатор. На етапі синтаксичного і семантичного аналізу відбувається перевірка програми на вхідній мові програмування на наявність помилок.

Перед генеруванням вихідного коду програма на вхідній мові програмування перетворюється у двійкове абстрактне синтаксичне дерево, обходячи яке генератор коду будує вихідний код на мові програмування С.

Розроблене програмне забезпечення налаштоване і протестоване на тестових прикладах.

# ЗМІСТ

[Пояснювальна записка 1](#_Toc187775564)

[ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ 2](#_Toc187775565)

[АНОТАЦІЯ 4](#_Toc187775566)

[ЗМІСТ 5](#_Toc187775567)

[ВСТУП 6](#_Toc187775568)

[ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ 9](#_Toc187775569)

[1. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ 12](#_Toc187775570)

[1.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура. 12](#_Toc187775571)

[Опис вхідної мови програмування у термінах розширеної форми Бекуса- Наура: 13](#_Toc187775572)

[1.2. Опис термінальних символів та ключових слів. 17](#_Toc187775573)

[2. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ 19](#_Toc187775574)

[2.1. Вибір технології програмування. 19](#_Toc187775575)

[2.2. Проектування таблиць транслятора та вибір структур даних. 19](#_Toc187775576)

[2.3. Розробка лексичного аналізатора. 22](#_Toc187775577)

[2.4. Розробка синтаксичного та семантичного аналізатора. 31](#_Toc187775578)

[2.5. Розробка генератора коду. 40](#_Toc187775579)

[3. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА 49](#_Toc187775580)

[3.1. Опис інтерфейсу та інструкції користувачу. 49](#_Toc187775581)

[3.2. Виявлення лексичних і синтаксичних помилок. 51](#_Toc187775582)

[3.3. Перевірка роботи транслятора за допомогою тестових задач. 53](#_Toc187775583)

[ВИСНОВКИ 57](#_Toc187775584)

[СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ 58](#_Toc187775585)

[ДОДАТКИ 59](#_Toc187775586)

# ВСТУП

**Транслятор**

Транслятор – це програма чи комплекс програм, що здійснюють переклад тексту, написаного однією мовою програмування (вхідна мова), в текст, поданий іншою мовою (вихідна мова).

**Види трансляторів**

Розрізняють транслятори двох видів:

1. **Компілюючого типу**
2. **Інтерпретуючого типу**

**Компілятор**

Компілятор – це транслятор, для якого:

* **Вхідна мова:** мова високого рівня (наприклад, C, Pascal, Algol).
* **Вихідна мова:** мова асемблера чи мова машинних команд.

Особливості:

* Переклад вхідної програми на вихідну мову виконується одразу цілком.
* Вхідна та вихідна програми завжди подаються у вигляді тексту.

**Асемблер**

Асемблер – це компілятор, у якому:

* **Вхідна мова:** мова асемблера.
* **Вихідна мова:** мова машинних команд.

**Інтерпретатор**

Інтерпретатор – це транслятор, що:

* Здійснює пооператорний переклад тексту програми на вихідну мову.
* Одночасно виконує ці оператори.

**Результат:** на виході інтерпретатора отримуємо результат роботи вхідної програми.

**Структура транслятора (компілятора)**

Загальна структура транслятора (компілятора) показана на рис.1.

**Лексичний аналізатор (scanner, сканер)**

* Здійснює перетворення вхідного тексту програми (рядок символів) у рядок лексем, поданий у цифровій формі.
* Виявляє лексичні помилки.

**Лексема** – це найменша одиниця інформації, яка обробляється синтаксичним аналізатором.

**Приклади лексем:**

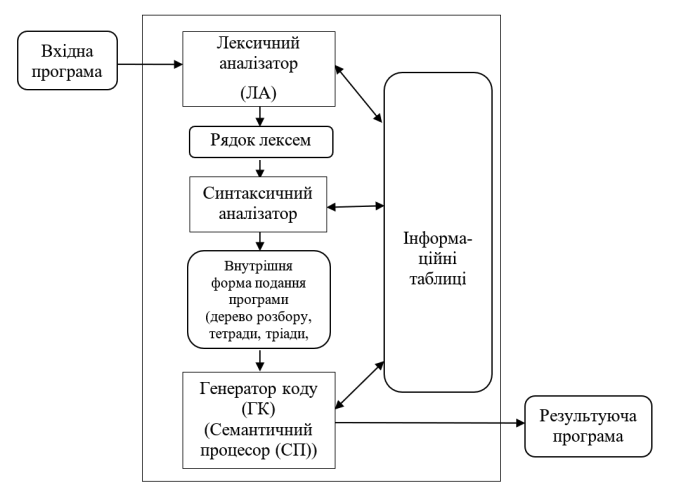
1. **Односимвольні роздільники:** ,, ;, .
2. **Знаки операцій:** +, -, \*, /
3. **Багатосимвольні роздільники:** <=, <>
4. **Ідентифікатори**
5. **Константи**
6. **Ключові слова:** for, while тощо.

Рис.1. Структура транслятора (компілятора)

**Синтаксичний аналізатор (parser, парсер)**

Синтаксичний аналізатор:

* Здійснює декомпозицію вхідної програми (рядок лексем) у структурні одиниці мови:
  + Оператори
  + Описання
  + Декларації
* Перевіряє відповідність граматиці вхідної мови.
* Виявляє синтаксичні помилки.

**Дерево розбору**

Це внутрішня форма подання вхідної програми, яка:

* Містить структурні одиниці мови.
* Відображає зв’язки між ними.

**Генератор коду (семантичний процесор)**

Генератор коду перетворює:

* Вхідну програму, подану у внутрішній формі.
* Вихідний текст у вигляді операторів або команд вихідною мовою.

Це відбувається на основі семантики вхідної мови.  
Семантичний процесор – це інша назва генератора коду.

**Метамови**

Для опису мов програмування використовуються спеціальні мови – метамови:

* Метасинтаксична мова для опису синтаксису.
* Метасемантична мова для опису семантики.

**Теоретична основа трансляторів**

1. Лексичний і синтаксичний аналізатори
   * Базуються на теорії формальних граматик.
2. Генератор коду
   * Використовує мови та методи опису семантики мов програмування.

# ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ

**1. Огляд методів та способів проєктування трансляторів**

Проєктування трансляторів є багатоступеневим процесом, що базується на використанні формальних методів опису мов програмування та побудови алгоритмів. Основні методи та способи проєктування трансляторів поділяються на такі етапи:

**Формальний опис мови**

Для опису мов програмування використовуються спеціальні засоби:

* Граматики (контекстно-вільні, регулярні) для формалізації синтаксису
* Семантичні правила для визначення поведінки програми
* Метамови:
  + Метасинтаксична мова для опису структури програми
  + Метасемантична мова для визначення значення та дій

**Розробка лексичного аналізатора**

* Виділення лексем із тексту програми
* Перетворення вхідного тексту на послідовність символів, що обробляються синтаксичним аналізатором

**Синтаксичний аналіз**

* Перетворення послідовності лексем у дерево розбору
* Виявлення синтаксичних помилок

**Семантичний аналіз**

* Перевірка семантичної коректності програми (типи даних, області видимості)
* Генерація проміжного представлення програми

**Генерація коду**

* Перетворення програми у вихідний код цільової мови (мови асемблера чи машинних команд)
* Оптимізація отриманого коду

**Тестування та верифікація**

* Перевірка коректності роботи транслятора на тестових програмах
* Виявлення та виправлення помилок у реалізації

**Види підходів до проєктування**

Ручне проєктування: використання алгоритмів та структур даних для реалізації транслятора  
Автоматизоване проєктування: застосування генераторів аналізаторів, таких як Lex і Yacc  
Комбіноване проєктування: поєднання ручного та автоматизованого підходів

Проєктування трансляторів є складною інженерною задачею, яка потребує ґрунтовних знань теорії формальних мов і алгоритмів, а також практичних навичок у розробці програмного забезпечення.

**1. Огляд методів та способів проєктування трансляторів**

Проєктування трансляторів є багатоступеневим процесом, що включає формальний опис мов, створення алгоритмів та інструментів для їх реалізації. Основні методи та способи проєктування поділяються на три категорії: ручне, автоматизоване та комбіноване проєктування.

**Ручне проєктування**

Цей метод передбачає розробку транслятора без використання спеціалізованих інструментів автоматизації. Основні етапи:

1. **Аналіз вимог**
   * Визначення вхідної та вихідної мов.
   * Опис граматики вхідної мови.
2. **Розробка алгоритмів**
   * Лексичний аналізатор для виділення лексем.
   * Синтаксичний аналізатор для побудови дерева розбору.
   * Семантичний аналізатор для перевірки логічної коректності програми.
   * Генератор коду для створення вихідного тексту цільовою мовою.
3. **Реалізація**
   * Написання програмного коду аналізаторів та генератора вручну.
   * Оптимізація реалізації для покращення продуктивності.
4. **Переваги**
   * Гнучкість у реалізації.
   * Можливість адаптації під специфічні вимоги.
5. **Недоліки**
   * Значний час розробки.
   * Велика ймовірність помилок через людський фактор.

**Автоматизоване проєктування**

Цей метод використовує спеціалізовані інструменти для автоматичної генерації частин транслятора.

1. **Генерація лексичного аналізатора**
   * Використовуються інструменти, такі як Lex або Flex, для автоматичного створення сканера.
2. **Генерація синтаксичного аналізатора**
   * Інструменти, наприклад, Yacc або Bison, забезпечують автоматичну генерацію парсера на основі граматики.
3. **Автоматизоване створення генератора коду**
   * Використовуються генератори шаблонів або середовища з підтримкою метасемантичних мов.
4. **Переваги**
   * Швидкість розробки.
   * Зменшення ймовірності помилок завдяки перевіреним інструментам.
5. **Недоліки**
   * Обмежена гнучкість.
   * Залежність від специфіки інструментів.

**Комбіноване проєктування**

Цей підхід поєднує ручне та автоматизоване проєктування, щоб використати переваги обох методів.

1. **Основні етапи**
   * Лексичний і синтаксичний аналізатор генеруються автоматично.
   * Семантичний аналізатор та генератор коду створюються вручну для забезпечення максимальної гнучкості.
2. **Переваги**
   * Баланс між швидкістю розробки та гнучкістю.
   * Можливість точного налаштування критичних компонентів.
3. **Недоліки**
   * Необхідність знань як ручного, так і автоматизованого підходу.
   * Може зайняти більше часу, ніж повністю автоматизований метод.

# ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ

## Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура.

Для задання синтаксису мов програмування використовують форму Бекуса- Наура або розширену форму Бекуса-Наура — це спосіб запису правил контекстно- вільної граматики, тобто форма опису формальної мови. Саме її типово використовують для запису правил мов програмування та протоколів комунікації.

БНФ визначає скінченну кількість символів (нетерміналів). Крім того, вона визначає правила заміни символу на якусь послідовність букв (терміналів) і символів. Процес отримання ланцюжка букв можна визначити поетапно: спочатку є один символ (символи зазвичай знаходяться у кутових дужках, а їх назва не несе жодної інформації). Потім цей символ замінюється на деяку послідовність букв і символів, відповідно до одного з правил. Потім процес повторюється (на кожному кроці один із символів замінюється на послідовність, згідно з правилом). Зрештою , виходить ланцюжок , що складається з букв і не містить символів. Це означає , що отриманий ланцюжок може бути виведений з початкового символу .

Нотація БНФ є набором «продукцій», кожна з яких відповідає зразку:

символ = <вираз, що містить символи>

де вираз, що містить символи це послідовність символів або послідовності символів, розділених вертикальною рискою |, що повністю перелічують можливий вибір символ з лівої частини формули.

У розширеній формі нотації Бекуса — Наура вирази, що можна пропускати або які можуть повторятись слід записувати у фігурних дужках { … }:, а можлива поява може відображатися застосуванням квадратних дужок [ … ]:.

## Опис вхідної мови програмування у термінах розширеної форми Бекуса- Наура:

labeled\_point = label , ":"

goto\_label = tokenGOTO, label, ";"

program\_name = ident,";"

value\_type = tokenINTEGER16

other\_declaration\_ident = tokenCOMMA , ident

declaration = value\_type , ident , {other\_declaration\_ident}

unary\_operator = tokenNOT | tokenMINUS | tokenPLUS

unary\_operation = unary\_operator , expression

binary\_operator = tokenAND | tokenOR | tokenEQUAL | tokenNOTEQUAL | tokenLESSOREQUAL | tokenGREATEROREQUAL | tokenPLUS | tokenMINUS | tokenMUL | tokenDIV | tokenMOD

binary\_action = binary\_operator , expression

left\_expression = group\_expression | unary\_operation | ident | value

expression = left\_expression , {binary\_action}

group\_expression = tokenGROUPEXPRESSIONBEGIN , expression , tokenGROUPEXPRESSIONEND

//

bind\_right\_to\_left = ident , tokenRLBIND , expression

bind\_left\_to\_right = expression , tokenLRBIND , ident

//

if\_expression = expression

body\_for\_true = {statement} , ";"

body\_for\_false = tokenELSE , {statement} , ";"

cond\_block = tokenIF , tokenGROUPEXPRESSIONBEGIN , if\_expression , tokenGROUPEXPRESSIONEND , body\_for\_true , [body\_for\_false];

//

cycle\_begin\_expression = expression

cycle\_counter = ident

cycle\_counter\_rl\_init = cycle\_counter , tokenRLBIND , cycle\_begin\_expression

cycle\_counter\_lr\_init = cycle\_begin\_expression , tokenLRBIND , cycle\_counter

cycle\_counter\_init = cycle\_counter\_rl\_init | cycle\_counter\_lr\_init

cycle\_counter\_last\_value = value

cycle\_body = tokenDO , statement , {statement}

forto\_cycle = tokenFOR , cycle\_counter\_init , tokenTO , cycle\_counter\_last\_value , cycle\_body , ";"

continue\_while = tokenCONTINUE , tokenWHILE

exit\_while = tokenEXIT , tokenWHILE

statement\_in\_while\_body = statement | continue\_while | exit\_while

while\_cycle\_head\_expression = expression

while\_cycle = tokenWHILE , while\_cycle\_head\_expression , {statement\_in\_while\_body} , tokenEND , tokenWHILE

//

repeat\_until\_cycle\_cond = group\_expression

repeat\_until\_cycle = tokenREPEAT , {statement} , tokenUNTIL , repeat\_until\_cycle\_cond

input = tokenGET , tokenGROUPEXPRESSIONBEGIN , ident , tokenGROUPEXPRESSIONEND

output = tokenPUT , tokenGROUPEXPRESSIONBEGIN , expression , tokenGROUPEXPRESSIONEND

statement = bind\_right\_to\_left | bind\_left\_to\_right | cond\_block | forto\_cycle | while\_cycle | repeat\_until\_cycle | labeled\_point | goto\_label | input | output

program = tokenNAME , program\_name , tokenSEMICOLON , tokenBODY , tokenDATA , [declaration] , tokenSEMICOLON , {statement} , tokenEND

//

digit = digit\_0 | digit\_1 | digit\_2 | digit\_3 | digit\_4 | digit\_5 | digit\_6 | digit\_7 | digit\_8 | digit\_9

non\_zero\_digit = digit\_1 | digit\_2 | digit\_3 | digit\_4 | digit\_5 | digit\_6 | digit\_7 | digit\_8 | digit\_9

unsigned\_value = ((non\_zero\_digit , {digit}) | digit\_0)

value = [sign] , unsigned\_value

// -- hello wolrd

letter\_in\_lower\_case = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

letter\_in\_upper\_case = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

ident = tokenUNDERSCORE , letter\_in\_upper\_case , letter\_in\_upper\_case , letter\_in\_upper\_case , letter\_in\_upper\_case , letter\_in\_upper\_case , letter\_in\_upper\_case , letter\_in\_upper\_case

label = letter\_in\_lower\_case , {letter\_in\_lower\_case}

//

sign = sign\_plus | sign\_minus

sign\_plus = '-'

sign\_minus = '+'

//

digit\_0 = '0'

digit\_1 = '1'

digit\_2 = '2'

digit\_3 = '3'

digit\_4 = '4'

digit\_5 = '5'

digit\_6 = '6'

digit\_7 = '7'

digit\_8 = '8'

digit\_9 = '9'

//

tokenCOLON = ":"

tokenGOTO = "goto"

tokenINTEGER16 = "integer16"

tokenCOMMA = ","

tokenNOT = "not"

tokenAND = "and"

tokenOR = "or"

tokenEQUAL = "="

tokenNOTEQUAL = "<>"

tokenLESSOREQUAL = "<"

tokenGREATEROREQUAL = ">"

tokenPLUS = "add"

tokenMINUS = "sub"

tokenMUL = "\*"

tokenDIV = "/"

tokenMOD = "%"

tokenGROUPEXPRESSIONBEGIN = "("

tokenGROUPEXPRESSIONEND = ")"

tokenRLBIND = "<-"

tokenLRBIND = ","

tokenELSE = "else"

tokenIF = "if"

tokenDO = "do"

tokenFOR = "for"

tokenTO = "to"

tokenWHILE = "while"

tokenCONTINUE = "continue"

tokenEXIT = "exit"

tokenREPEAT = "repeat"

tokenUNTIL = "until"

tokenGET = "scan"

tokenPUT = "print"

tokenNAME = "program"

tokenBODY = "start"

tokenDATA = "var"

tokenEND = "finish"

tokenSEMICOLON = ""

//

tokenUNDERSCORE = "\_"

//

A = "A"

B = "B"

C = "C"

D = "D"

E = "E"

F = "F"

G = "G"

H = "H"

I = "I"

J = "J"

K = "K"

L = "L"

M = "M"

N = "N"

O = "O"

P = "P"

Q = "Q"

R = "R"

S = "S"

T = "T"

U = "U"

V = "V"

W = "W"

X = "X"

Y = "Y"

Z = "Z"

//

a = "a"

b = "b"

c = "c"

d = "d"

e = "e"

f = "f"

g = "g"

h = "h"

i = "i"

j = "j"

k = "k"

l = "l"

m = "m"

n = "n"

o = "o"

p = "p"

q = "q"

r = "r"

s = "s"

t = "t"

u = "u"

v = "v"

w = "w"

x = "x"

y = "y"

z = "z"

//

## Опис термінальних символів та ключових слів.

Визначаємо термінальні символи і ключові слова:

* **start** – початок програми
* **var** – оголошення змінних
* **finish** – кінець програми
* **integer16** – тип даних
* **scan**– оператор вводу
* **print** – оператор виводу
* **if, else** – умовний оператор
* **<-** –оператор присвоєння
* **goto –** оператор переходу
* **for (**to – downto) – оператор циклу
* **repeat-until –** оператор циклу з постумовою

• **add** – додавання

* **sub** – віднімання
* **\*** – множення
* **/** – ділення
* **%** - додавання за модулем 2
* **>** – більше
* **<** – менше

• **=** – рівність

* **<>** – нерівність
* **not** – заперечення
* **and** – логічне І
* **or** – логічне АБО
* **;** – кінець оператора
* **,** – розділювач змінних
* **(**– відкрита дужка
* **)** – закрита дужка
* **!!** – початок коментаря
* **[a…z][A…Z]** – маленькі і великі латинські букви
* **0…9** – цифри
* символи табуляції, переходу на новий рядок, пробіл

Програма на вихідній мові програмування має починатись з ключового слова start, далі має йти розділ опису змінних var. Між розділом var і ключовим словом finish розміщуються оператори програми. Оператори є 4: оператор вводу даних scan, оператор виводу даних print, оператор присвоєння <- і умовний оператор if – [- else ]. Кожен оператор має завершуватись символом крапка з комою ;.

Оператор присвоєння дозволяє присвоїти деякій змінній значення арифметичного виразу. Допустимі арифметичні операції: add, -, sub, /. Операндами можуть бути змінні, цілі додатні константи і інші вирази, взяті в дужки.

В умовному операторі використовуються логічні вирази, допустимі такі операції порівняння >, <, =, <> і такі логічні операції not and or. Ідентифікаторами (імена змінних) можуть бути довжиною 6-х символів і складатись з 6 латинських літер. Перший символ ідентифікатора завжди велика літера наступні 5 завжди мала літера. Тип даних лише один – integer16, при оголошенні декількох змінних вони записуються через кому, вкінці опису змінних ставиться символ крапка з комою ;. Коментарі починаються з !!.

Приклади оголошення змінних:

integer16 aINTEG;

integer16 aINTEG, hINTEG, cINTEG;

Приклади арифметичних виразів:

1000 sub 7

aINTEG add 69

666 add 666 \* aINTEG add 4308

cNIGEG \* (14 add 88) sub 50 / b

Приклади логічних виразів:

aINTEG > bINTEG

aINTEG > bINTEG and aINTEG > 1

not(aINTEG < cINTEG)

aINTEG <> bINTEG

aINTEG <> 0 sub 100

# РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ

## Вибір технології програмування.

Перед тим як розпочинати створювати програму, для більш швидкого і ефективного її написання, необхідно розробити алгоритм її функціонування, та вибрати технологію програмування, середовище програмування.

Для виконання поставленого завдання найбільш доцільно буде використати середовище програмування Microsoft Visual Studio 2022, та мову програмування C/С++.

Для якісного і зручного використання розробленої програми користувачем, було прийнято рішення створення консольного інтерфейсу.

## Проектування таблиць транслятора та вибір структур даних.

Використання таблиць значно полегшує створення трансляторів, а тому створимо необхідні структури даних для зберігання інформації про лексеми:

struct LexemInfo {public:

char lexemStr[MAX\_LEXEM\_SIZE];

unsigned long long int lexemId;

unsigned long long int tokenType;

unsigned long long int ifvalue;

unsigned long long int row;

unsigned long long int col;

LexemInfo();

LexemInfo(const char\* lexemStr, unsigned long long int lexemId, unsigned long long int tokenType, unsigned long long int ifvalue, unsigned long long int row, unsigned long long int col);

LexemInfo(const NonContainedLexemInfo& nonContainedLexemInfo);

};

**Опис структури LexemInfo**

LexemInfo — це структура, яка використовується для зберігання інформації про окрему лексему, отриману під час лексичного аналізу. Вона має публічний доступ до своїх членів і призначена для забезпечення зручного доступу до атрибутів лексеми. Нижче детально описані її елементи та функції:

**Члени структури:**

1. **char lexemStr[MAX\_LEXEM\_SIZE]**  
   Массив символів, що містить саму лексему у вигляді рядка.  
   MAX\_LEXEM\_SIZE — це максимальний розмір лексеми, зазвичай визначений як константа.
2. **unsigned long long int lexemId**  
   Унікальний ідентифікатор лексеми. Він дозволяє відрізняти лексеми між собою.
3. **unsigned long long int tokenType**  
   Тип токена, який відповідає лексемі. Наприклад, це може бути константа, оператор, ключове слово тощо.
4. **unsigned long long int ifvalue**  
   Додаткове значення, яке використовується для обробки умовних виразів або контексту лексеми. Наприклад, це може бути значення для порівняння чи виконання умов.
5. **unsigned long long int row**  
   Номер рядка, де знаходиться лексема в коді. Це корисно для відлагодження або повідомлень про помилки.
6. **unsigned long long int col**  
   Номер колонки в рядку, де розташована лексема.
7. **// TODO: ...**  
   Коментар, який вказує, що до структури можуть бути додані нові члени або властивості для розширення її функціональності.

**Конструктори:**

1. **Конструктор за замовчуванням: LexemInfo()**  
   Ініціалізує структуру з початковими значеннями. Зазвичай це нульові або порожні значення для членів структури.
2. **Параметризований конструктор: LexemInfo(const char\* lexemStr, unsigned long long int lexemId, unsigned long long int tokenType, unsigned long long int ifvalue, unsigned long long int row, unsigned long long int col)**  
   Ініціалізує структуру з заданими значеннями.
   * **lexemStr**: рядок лексеми.
   * **lexemId**: унікальний ідентифікатор.
   * **tokenType**: тип токена.
   * **ifvalue**: додаткове значення.
   * **row**: номер рядка.
   * **col**: номер колонки.
3. **Конструктор копіювання: LexemInfo(const NonContainedLexemInfo& nonContainedLexemInfo)**  
   Ініціалізує LexemInfo на основі іншої структури NonContainedLexemInfo. Це дозволяє створити об'єкт на основі схожої структури.

**Призначення:**

Ця структура є корисною для:

* Лексичного аналізу (збереження інформації про токени у процесі аналізу вхідного коду).
* Збереження позицій (рядок і колонка) для генерації повідомлень про помилки.
* Структурування даних про лексеми, необхідних для побудови синтаксичного дерева.
* Розширення можливостей за допомогою додавання нових полів, наприклад, для семантичного аналізу.

## Розробка лексичного аналізатора.

Основна задача лексичного аналізу – розбити вихідний текст, що складається з послідовності символів, на послідовність слів, або лексем, тобто виділити ці слова з безперервної послідовності символів. Всі символи вхідної послідовності з цієї точки зору розділяються на символи, що належать яким-небудь лексемам, і символи, що

розділяють лексеми. В цьому випадку використовуються звичайні засоби обробки рядків. Вхiдна програма проглядається послідовно з початку до кінця. Базовi елементи, або лексичнi одиницi, роздiляються пробілами, знаками операцiй i спецiальними символами (новий рядок, знак табуляції), i таким чином видiляються та розпізнаються iдентифiкатори, лiтерали i термiнальнi символи (операцiї, ключові слова).

При виділенні лексеми вона розпізнається та записується у таблицю лексем за допомогою відповідного номера лексеми, що є унікальним для кожної лексеми із усього можливого їх набору. Це дає можливість наступним фазам компiляції звертатись лексеми не як до послідовності символів, а як до унікального номера лексеми, що значно спрощує роботу синтаксичного аналізатора: легко перевіряти належність лексеми до відповідної синтаксичної конструкції та є можливість легкого перегляду програми, як вгору, так і вниз, від текучої позиції аналізу. Також в таблиці лексем ведуться записи, щодо рядка відповідної лексеми – для місця помилки – та додаткова інформація.

Лексична фаза вiдкидає коментарi, оскiльки вони не мають нiякого впливу на виконання програми, отже ж й на синтаксичний розбір та генерацію коду.

Розділимо лексеми на типи або лексичні класи:

* Ключові слова (**start**, **var**, **finish**, **scan**, **print**, **integer**16, **if**, **else**, **for , goto ,downto ,repeat,until,while** )
* Ідентифікатори
* Числові константи (ціле число без знаку)
* Оператор присвоєння ( **<-**)
* Знаки операції (**add** ,**sub , \***, **/**, **>**, **<**, **=**, **<>**, **not**, **and**, **or**)
* Розділювачі (**;** ,)
* Дужки ( **(**, **)** )
  + 1. Розробка алгоритму роботи лексичного аналізатора.

Даний лексичний аналізатор — це програмний модуль, який розбиває вхідний текст на лексеми (основні синтаксичні одиниці) і класифікує їх за певними типами. Його основна мета — підготовка тексту до подальшого синтаксичного або семантичного аналізу. У цьому коді реалізовано багато функцій, які забезпечують ідентифікацію ключових слів, значень, ідентифікаторів, а також обробку коментарів.

Ось як працює цей аналізатор:

**1. Основні структури даних**

**LexemInfo**

Містить інформацію про кожну лексему:

* **lexemStr** — текстовий рядок лексеми.
* **lexemId** — унікальний ідентифікатор лексеми.
* **tokenType** — тип токена (ключове слово, ідентифікатор, значення тощо).
* **ifvalue** — додаткова інформація для значень.
* **row і col** — позиція лексеми в тексті (номер рядка та стовпця).

**NonContainedLexemInfo**

Служить для тимчасового зберігання лексем, забезпечуючи використання буфера (tempStrFor\_123).

**2. Основні масиви**

* **lexemesInfoTable** — таблиця, де зберігаються всі знайдені лексеми.
* **identifierIdsTable** — таблиця для збереження ідентифікаторів, яка запобігає дублюванню.

**3. Алгоритм лексичного аналізу**

**3.1. Токенізація (tokenize)**

Ця функція розбиває текст на токени відповідно до регулярного виразу:

* Регулярний вираз (TOKENS\_RE) визначає, які символи формують токен (ідентифікатори, ключові слова, числа тощо).
* За допомогою ітератора (std::sregex\_token\_iterator) текст обробляється токен за токеном.

**3.2. Ідентифікація токена (lexicalAnalyze)**

Для кожного токена викликаються функції:

1. **tryToGetKeyWord** — перевіряє, чи є токен ключовим словом.
2. **tryToGetIdentifier** — перевіряє, чи є токен ідентифікатором.
3. **tryToGetUnsignedValue** — перевіряє, чи є токен числовим значенням.

Якщо жоден із цих тестів не вдається, токен помічається як "непередбачувана лексема" (UNEXPECTED\_LEXEME\_TYPE).

**4. Обробка ключових слів, ідентифікаторів та значень**

**Ключові слова**

Ключові слова перевіряються за допомогою регулярного виразу (KEYWORDS\_RE) і отримують унікальний lexemId.

**Ідентифікатори**

* Перевіряються регулярним виразом (IDENTIFIERS\_RE).
* Заноситься до таблиці identifierIdsTable.

**Значення**

* Перевіряються регулярним виразом (UNSIGNEDVALUES\_RE).
* Зберігаються у поле ifvalue.

**5. Обробка коментарів (commentRemover)**

Функція видаляє коментарі з тексту. Вона підтримує:

* Однорядкові коментарі (наприклад, //).
* Багаторядкові коментарі (наприклад, /\* ... \*/). Після видалення коментарі замінюються пробілами, зберігаючи структуру тексту.

**6. Збереження позицій (setPositions)**

Функція встановлює номер рядка та стовпця кожної лексеми у вхідному тексті. Це дозволяє вказувати точне місце розташування помилок у тексті.

**7. Друк результатів (printLexemes)**

Результати аналізу виводяться у вигляді таблиці, де показано:

* Індекс лексеми.
* Її текст.
* Ідентифікатор.
* Тип.
* Значення (для чисел).
* Рядок і стовпець у тексті.

**Структура та поля результатів лексичного аналізатора**

Результати роботи лексичного аналізатора подаються у вигляді таблиці. Кожен рядок цієї таблиці представляє одну лексему та містить наступну інформацію:

**Поля таблиці:**

1. **Індекс лексеми (index)**  
   Це порядковий номер лексеми у загальному списку. Використовується для нумерації та швидкого доступу до конкретної лексеми.
2. **Текст лексеми (lexemStr)**  
   Текстовий вигляд лексеми, зчитаний з вихідного тексту програми. Наприклад, це може бути слово, число, символ або оператор.
3. **Ідентифікатор лексеми (lexemId)**  
   Унікальний ідентифікатор, який присвоюється кожній лексемі залежно від її типу. Наприклад:
   * Ідентифікатори для ключових слів.
   * Ідентифікатори для змінних.
   * Унікальні номери для інших лексем.
4. **Тип лексеми (tokenType)**  
   Визначає тип лексеми, наприклад:
   * **Ключове слово (keyword)**.
   * **Ідентифікатор (identifier)**.
   * **Числове значення (value)**.
   * **Неочікувана лексема (unexpected lexeme)**.
5. **Значення (ifvalue)**  
   Актуальне значення для числових лексем. Наприклад, якщо лексема — це число 123, то його значення буде 123. Для інших типів лексем це поле може бути неактивним.
6. **Рядок (row)**  
   Номер рядка у вихідному тексті, де знаходиться лексема. Це полегшує ідентифікацію її місця у програмному коді.
7. **Стовпець (col)**  
   Номер символу у рядку, з якого починається лексема. Це додатково уточнює її позицію у вихідному коді.

**Стани під час аналізу**

Лексичний аналізатор проходить кілька основних станів:

1. **Ініціалізація**  
   Підготовка таблиць і структур, зокрема:
   * Таблиці лексем (lexemesInfoTable).
   * Таблиці ідентифікаторів (identifierIdsTable).
2. **Обробка тексту**
   * Видалення коментарів.
   * Розбиття тексту на токени.
3. **Класифікація лексем**  
   Для кожної лексеми визначають:
   * Чи є вона ключовим словом.
   * Чи є вона ідентифікатором.
   * Чи є вона числовим значенням.
   * Чи є вона несподіваною або помилковою.
4. **Формування таблиці результатів**  
   Для кожної лексеми записується відповідна інформація: індекс, текст, ідентифікатор, тип, значення, позиція в тексті.
5. **Виведення результатів**  
   Таблиця лексем друкується у форматі зручному для перегляду, де відображаються всі згадані поля.

Табл 1.1

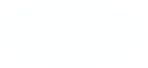
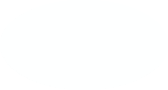
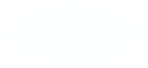
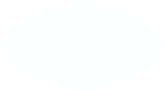
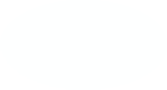
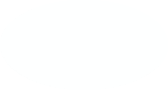
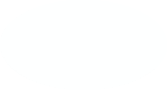
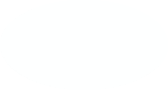
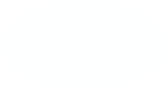
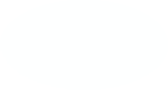
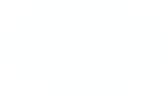
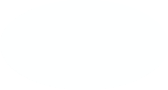
| **Індекс** | **Текст лексеми** | **Ідентифікатор** | **Тип** | **Значення** | **Рядок** | **Стовпець** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | program | 101 | Ключове слово | - | 1 | 1 |
| 1 | mЗrgty | 1 | Ідентифікатор | - | 1 | 9 |
| 2 | start | 102 | Ключове слово | - | 2 | 1 |
| 3 | 123 | 1001 | Значення | 123 | 3 | 5 |
| 4 | finish | 103 | Ключове слово | - | 4 | 1 |

**Преваги такої структури:**

* **Простота аналізу**: Користувач легко знаходить помилки або несподівані лексеми завдяки вказаним рядкам і стовпцям.
* **Гнучкість**: Додавання нових типів лексем або розширення можливостей аналізатора не потребує значних змін.
* **Уніфікованість**: Усі дані про лексеми представлені в одній структурованій формі.

**8. Особливості**

1. **Буферизація**:
   * Для тимчасового збереження рядків використовується буфер tempStrFor\_123, що дозволяє ефективно управляти пам'яттю.
2. **Гнучкість**:
   * Регулярні вирази (TOKENS\_RE, IDENTIFIERS\_RE, KEYWORDS\_RE, UNSIGNEDVALUES\_RE) можна налаштовувати під конкретні вимоги.
3. **Обробка помилок**:
   * Якщо лексема не відповідає жодному з шаблонів, вона позначається як помилкова.



Start

буква | цифра

буква

Letter

цифра

цифра

інший символ

інший символ

Digit Finish

EOF

інший символ

Another

EnSdtOafrFtile

будь-який символ,

крім EOF | \n

EOF

/

Scomment

/

Comment

\n

пробіл |табуляція | \n

Separators

будь-який символ

EOF

EnSdtOafrFtile

*Рис. 3.1. Граф-схема алгоритму роботи лексичного аналізатора.*

* + 1. Опис програми реалізації лексичного аналізатора.

Основна задача лексичного аналізатора – розділити вхідний текст програми, що складається з послідовності символів, на окремі лексеми, тобто слова, які мають змістовне значення для подальшого аналізу. Усі символи вхідної послідовності поділяються на ті, що належать лексемам, і ті, що виконують функцію роздільників. У процесі аналізу використовуються стандартні методи обробки рядків. Вхідний текст програми переглядається послідовно від початку до кінця, а базові елементи (лексичні одиниці) виділяються на основі пробілів, знаків операцій, спеціальних символів (таких як новий рядок або табуляція). У результаті розпізнаються ідентифікатори, літерали та термінальні символи (наприклад, операції або ключові слова).

Програма аналізує файл доти, доки не досягне його кінця. Для обробки вхідного файлу викликається функція tokenize(). Ця функція читає вміст файлу, виділяє лексеми та порівнює їх із зарезервованими словами. У разі збігу лексемі присвоюється відповідний тип або значення (якщо це числова константа).

Кожна виділена лексема додається до списку m\_tokens з використанням унікального типу лексеми. Це дозволяє наступним фазам компіляції оперувати лексемами не як послідовностями символів, а як конкретними типами, що значно полегшує синтаксичний аналіз. Наприклад, перевірка належності лексеми до певної синтаксичної конструкції або навігація текстом програми (вперед і назад від поточної позиції) стають більш зручними. У таблиці лексем також зберігається інформація про рядок і стовпець кожної лексеми, що спрощує пошук місця помилки. Додатково зберігається метаінформація, корисна для подальших етапів аналізу.

Під час лексичного аналізу виявляються та відзначаються лексичні помилки, наприклад, некоректні символи або невірні ідентифікатори ігноруються, оскільки вони не впливають ні на синтаксичний розбір, ні на генерацію коду.

У межах цього проєкту реалізовано прямий лексичний аналізатор, який виділяє лексеми з вхідного тексту програми та формує таблицю лексем для подальшої обробки.

## Розробка синтаксичного та семантичного аналізатора.

Синтаксичний аналіз – це процес, що визначає, чи належить деяка послідовність лексем граматиці мови програмування. В принципі, для будь-якої граматики можна побудувати синтаксичний аналізатор, але граматики, які використовуються на практиці, мають спеціальну форму. Наприклад, відомо, що для будь-якої контекстно-вільної граматики може бути побудований аналізатор, складність якого не перевищує O(n3) для вхідного рядка довжиною n.

Код реалізує лексичний і синтаксичний аналізатор із побудовою абстрактного синтаксичного дерева (AST) на основі методу Кока-Янгера-Касамі (CYK) та рекурсивного спуску. Розглянемо основні етапи роботи:

**1. Лексичний аналіз**

Лексичний аналізатор розбиває вхідний текст на лексеми (мінімальні значущі одиниці мови, такі як ідентифікатори, ключові слова, константи тощо) та зберігає їх у таблиці LexemInfo.

**2. Метод CYK для синтаксичного аналізу**

* **Ініціалізація**: створюється таблиця parseInfoTable, де кожна комірка містить множину символів граматики.
* **Заповнення таблиці**: використовується двовимірний підхід, де кожна комірка заповнюється на основі правил граматики:
  + Якщо правило має один елемент справа, перевіряється відповідність лексеми цьому правилу.
  + Якщо правило має два елементи, шукається розбиття, яке дозволяє побудувати комбінацію двох піддерев.
* Після завершення побудови таблиці перевіряється наявність стартового символу граматики у верхньому правому куті таблиці. Якщо символ є, аналіз вважається успішним.

**3. Рекурсивний спуск**

Якщо метод CYK не успішний або обрано режим рекурсивного спуску, запускається рекурсивний аналізатор:

* Кожне правило граматики перевіряється на відповідність лексемам у поточній позиції.
* Якщо знайдено відповідність, індекс лексем збільшується, і аналіз продовжується для наступних правил.
* У разі помилки повертається інформація про невідповідність лексеми.

**4. Побудова абстрактного синтаксичного дерева (AST)**

* Дерево будується функцією buildAST. Кожен вузол представляє або термінальний, або нетермінальний символ.
* Для кожного правила створюються дочірні вузли, які відповідають його елементам.
* Якщо правило має два елементи справа, дерево будується рекурсивно для обох піддерев.

**5. Виведення AST**

Для візуалізації AST використовуються функції:

* printAST: виводить дерево в консоль у вигляді ієрархічної структури.
* printASTToFile: записує дерево у файл.

**6. Збереження таблиці CYK**

Таблиця результатів CYK може бути виведена або збережена у файл за допомогою функцій displayParseInfoTable та saveParseInfoTableToFile.

**7. Основна функція синтаксичного аналізу**

Функція syntaxAnalyze координує процес:

* Спочатку викликається метод CYK.
* Якщо CYK не успішний, виконується рекурсивний спуск.
* У разі помилки виводиться інформація про невідповідність та позицію помилки у вхідному коді.

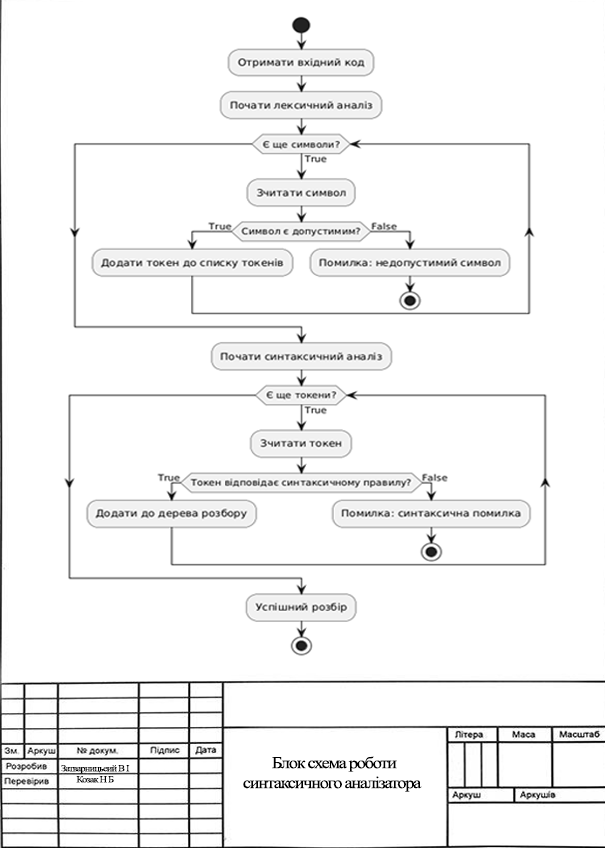


Рис 3. Графічне представлення роботи синтасичного аналізатора

* + 1. Розробка дерева граматичного розбору.

**Схема дерева розбору** виглядає наступним чином:

<програма>

│

├── 'program'

├── <назва програми>

│ └── <ідентифікатор>

├── 'start'

├── <оголошення змінних>

│ ├── <тип даних>

│ │ └── 'integer16'

│ └── <список змінних>

│ ├── <ідентифікатор>

│ └── { ',' <ідентифікатор> }

├── <тіло програми>

│ ├── <оператор> ';'

│ └── { <оператор> ';' }

│ │

│ ├── <присвоєння>

│ │ ├── <ідентифікатор>

│ │ ├── '<-'

│ │ └── <арифметичний вираз>

│ │ ├── <доданок>

│ │ │ ├── <множник>

│ │ │ │ └── (<ідентифікатор> | <число> | '(' <арифметичний вираз>

')')

│ │ │ └── { ('\*' | '/' | '%') <доданок> }

│ │ └── { ('+' | '-') <доданок> }

│ │

│ ├── <ввід>

│ │ ├── 'scan'

│ │ └── <ідентифікатор>

│ │

│ ├── <вивід>

│ │ ├── 'print'

│ │ └── <ідентифікатор>

│ │

│ ├── <умовний оператор>

│ │ ├── 'if'

│ │ ├── <логічний вираз>

│ │ │ ├── <вираз І>

│ │ │ │ ├── <порівняння>

│ │ │ │ │ └── (<операція порівняння> | 'not' '(' <логічний вираз> ')' | '(' <логічний вираз> ')')

│ │ │ │ │ ├── <арифметичний вираз>

│ │ │ │ │ ├── <оператор порівняння> ('=' | '<>' | '>' | '<')

│ │ │ │ │ └── <арифметичний вираз>

│ │ │ │ └── { 'and' <порівняння> }

│ │ │ └── { 'or' <вираз І> }

│ │ ├── <оператор>

│ │ └── ['else' <оператор>]

│ │

│ ├── <оператор циклу>

│ │ ├── 'while'

│ │ ├── <логічний вираз>

│ │ ├── <тіло циклу>

│ │ │ ├── <оператор>

│ │ │ └── { <оператор> ';' }

│ │ ├── ['exit'] (опціонально)

│ │ ├── ['next'] (опціонально)

│ │ └── 'end'

│ │

│ └── <складений оператор>

│ ├── 'start'

│ ├── <тіло програми>

│ └── 'finish'

│

└── 'finish'

*Рис. 3.2. Дерево граматичного розбору.*

* + 1. Розробка алгоритму роботи семантичного аналізазатора

На етапі семантичного аналізу нам необхідно вирішити задачу ідентифікації ідентифікаторів. Алгоритм ідентифікації складається з двох частин:

* перша частина алгоритму опрацьовує оголошення ідентифікаторів;
* друга частина алгоритму опрацьовує використання ідентифікаторів.

Нехай лексичний аналізатор видав чергову лексему, що є ідентифікатором. Лексичний аналізатор сформував структуру, що містить атрибути виділеної лексеми, такі як ім’я ідентифікатора, його тип і лексичний клас. Далі вся ця інформація передається семантичному аналізатору. Припустимо, що в даний момент опрацьовується оголошення ідентифікатора. Основна семантична дія в цьому випадку полягає в занесенні інформації про ідентифікатор у таблицю ідентифікаторів.

Опрацювання використання ідентифікатора. Припустимо, що уже побудовано (цілком чи частково) таблицю ідентифікаторів. Далі вся ця інформація передається фазі використання ідентифікаторів. Таким чином, відомо, що опрацьовується використання ідентифікатора. Для того, щоб одержати інформацію про тип ідентифікатора нам достатньо прочитати певне поле таблиці ідентифікаторів.

* + 1. Опис програми реалізації семантичного аналізатора.

Семантичний аналізатор виконує перевірку правильності структур та логіки програми на основі аналізу лексем та граматики. У цьому коді реалізовано кілька функцій, які відповідають за різні аспекти семантичного аналізу.

**Основні функції семантичного аналізатора**

1. **getLastDataSectionLexemIndex**  
   Ця функція знаходить індекс останньої лексеми у секції даних.
   * Використовує функцію парсера recursiveDescentParserRuleWithDebug, щоб пройти по граматиці секції даних ("program\_\_\_\_part1").
   * Якщо лексема знайдена, повертається її індекс; якщо ні – повертається помилка (~0).
2. **checkingInternalCollisionInDeclarations**  
   Перевіряє внутрішні колізії у деклараціях змінних і міток:
   * **Колізії identifier/identifier:** Виявляється, якщо ідентифікатор задекларовано кілька разів у тій самій області.
   * **Колізії label/label:** Виявляється при дублюванні міток.
   * **Колізії identifier/label:** Виявляється, якщо ідентифікатор використовується і як змінна, і як мітка.
   * Якщо ідентифікатор або мітка не були задекларовані, виводиться помилка.
3. **checkingVariableInitialization**  
   Перевіряє, чи ініціалізовано всі змінні перед використанням:
   * Аналізує ділянку коду після секції даних.
   * Визначає, чи були змінні ініціалізовані (перевіряє наявність операцій присвоєння, введення чи виклику функцій, що ініціалізують значення).
4. **checkingCollisionInDeclarationsByKeyWords**  
   Перевіряє, чи збігаються імена декларацій з ключовими словами:
   * Використовує регулярний вираз для виявлення збігів.
   * Якщо ідентифікатор відповідає ключовому слову, генерується помилка (COLLISION\_IK\_STATE).
5. **semantixAnalyze**  
   Головна функція, що викликає всі попередні модулі аналізу:
   * Перевіряє колізії в деклараціях.
   * Аналізує ініціалізацію змінних.
   * Перевіряє збіг імен з ключовими словами.
   * Якщо хоча б одна перевірка не проходить, повертається відповідний код помилки.

**Ключові аспекти реалізації**

1. **Лексеми та граматика:**
   * Семантичний аналізатор працює з таблицею лексем (lexemInfoTable) та граматикою (Grammar), які є результатами попередніх етапів аналізу (лексичного та синтаксичного).
   * Типи лексем визначаються полем tokenType.
2. **Перевірка колізій:**  
   Семантичний аналізатор знаходить конфлікти в ідентифікаторах, щоб уникнути неоднозначності або помилок у виконанні програми.
3. **Робота з регулярними виразами:**  
   Для перевірки ідентифікаторів на збіг із зарезервованими словами використовуються регулярні вирази (std::regex).
4. **Повідомлення про помилки:**  
   Усі помилки виводяться у консоль із деталізацією, наприклад:
5. Collision(identifier/identifier): myVariable
6. Uninitialized: myVariable
7. **Коди стану:**  
   Кожна функція повертає код стану (наприклад, SUCCESS\_STATE, COLLISION\_II\_STATE), що дозволяє головній функції визначити, чи є помилки.

**Типовий процес роботи**

1. Виклик функції semantixAnalyze, яка:
   * Перевіряє декларації та їх колізії.
   * Аналізує ініціалізацію змінних.
   * Виявляє невірне використання ключових слів.
2. У разі помилки повертається відповідний код, і програма виводить інформацію про проблему.

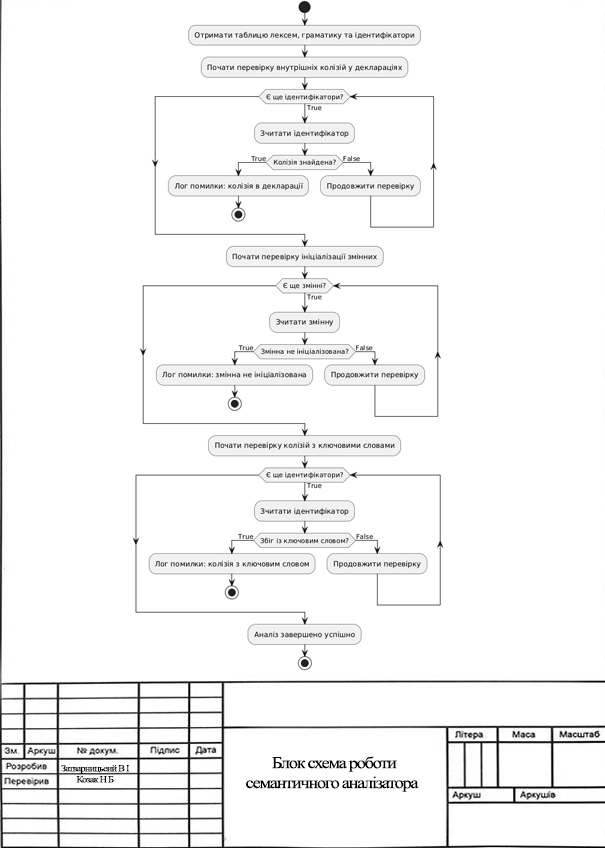


Рис4 графічне представлення роботи семантичного аналізатора

## Розробка генератора коду.

Генерація вихідного коду передбачає спочатку перетворення програми у якесь проміжне представлення, а тоді вже генерацію з проміжного представлення у вихідний код. У якості проміжного представлення виберемо абстрактне синтаксичне дерево.

Абстрактне синтаксичне дерево (AST) — це структура даних, яка представляє синтаксичну структуру вихідного коду програми у вигляді дерева. AST використовується в компіляторах, інтерпретаторах та інструментах статичного аналізу для обробки коду.

AST представляє тільки важливу для аналізу і виконання інформацію, ігноруючи зайві деталі (наприклад, круглі дужки чи крапки з комою). Це спрощений, але точний опис логіки програми.

Вузли дерева представляють конструкції мови програмування (оператори, вирази, змінні, функції тощо). Гілки відповідають підконструкціям або елементам цих конструкцій.

Кожен вузол відповідає певному типу конструкції коду (наприклад, оператору додавання, виклику функції, оголошенню змінної).

AST є спрощеною версією синтаксичного дерева. Воно не включає зайві вузли, що відповідають елементам, які не впливають на логіку програми (наприклад, дужки чи крапки з комою).

* + 1. Розробка алгоритму роботи генератора коду.

Будемо використовувати бінарні дерева, а отже вузол у нас має два нащадки, відповідно нарисуємо типові варіанти побудови дерева.

Програма має вигляд:

program

/ \

var statement

Оголошення змінних:

var

/ \

Id var

/ \

Id null

Тіло програми:

statement

/ \

оператор statement

/ \

оператор оператор

Оператор вводу (виводу):

input

/ \

id null

Умовний оператор:

if

/ \

умова else

/ \

оператор1 оператор2

Оператор присвоєння:

<-

/ \

id арфметчнй враз

Арифметичний вираз:

Доданок:

(add або syb)

/ \

доданок доданок

(\* або /)

/ \

множнк множнк

Множник:

фактор

/ \

id або number або (арфм. враз) null

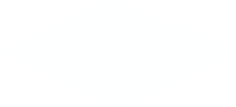
Складений оператор:

compount

/ \ statement null

Генератор коду буде обходити створене дерево і, маючи усію необхідну інформацію, генерувати вихідний код на мові програмування С у текстовий файл. Кожен вузол у дереві буде позначати якусь конструкцію, для якої генерується певний код на мові програмування С. Опрацювання кожного з вузлів дерева передбачає рекурсивний виклик функції генерування коду для лівого і правого нащадків.

Блок-схема алгоритму роботи генератора коду зображена на рисунку 3.6.



початок

т п вуз а o

id

program

var

compount

опрацювання

вуз а program

опрацювання

вуз а var

опрацювання

вуз а id

...

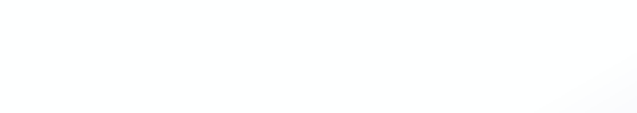
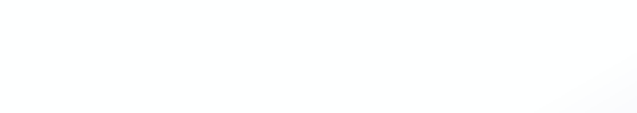
опрацювання

вуз а compount

кінець

*Рис. 3.6. Блок-сема алгоритму роботи генератора коду.*

Розглянемо на прикладі вузла program детальніше алгоритм обходу дерева, який зображено на рисунку 3.7. Вузол позначає програму, зліва будемо зберігати інформацію про оголошені змінні, справа про оператори програми. Опрацювання вузла полягає у друці у файл необхідних шаблонів на мові програмування С, а також рекурсивного виклику для опрацювання лівого і правого нащадків. Лівий нащадок – оголошення змінних (вузол var), правий – тіло програми (вузол statement).



початок опрацювання вуз а program

друк у фай :

include <stdio.h>

int main()

{

кінець опрацювання вуз а program

друк у фай :

return 0;

}

Рекурс вн й в к к д я правого нащадка

Рекурс вн й в к к д я івого нащадка

*Рис. 3.7. Блок-сема алгоритму опрацювання вузла program.*

* + 1. Опис програми реалізації генератора коду.

. Основні функції і макроси забезпечують різні етапи генерації коду: створення секцій даних, секцій коду, ініціалізації змінних і структурування команд. Давайте розглянемо основні компоненти і їх призначення:

**1. Макроси та константи**

* **MAX\_TEXT\_SIZE, MAX\_GENERATED\_TEXT\_SIZE**: Визначають максимальний розмір тексту та згенерованого коду.
* **SUCCESS\_STATE**: Статус для успішного виконання.
* **MAX\_OUTTEXT\_SIZE**: Буфер для вихідного тексту.
* **MAX\_LEXEM\_SIZE**: Максимальний розмір однієї лексеми.
* **MAX\_WORD\_COUNT**: Максимальна кількість слів/лексем, які обробляються.

**2. Структури даних**

* **LabelOffsetInfo**:
  + Зберігає інформацію про мітки (label) та їх позиції в коді.
  + Використовується для управління стрибками (goto) в асемблерному коді.
* **GotoPositionInfo**:
  + Інформація про позиції інструкцій стрибків, які мають бути пов'язані з відповідними мітками.
* **tokenStruct**:
  + Таблиця, що описує багатокомпонентні токени, такі як IF ... THEN, FOR ... TO ..., WHILE, тощо.

**3. Генерація коду**

* **makeCode**:
  + Основна функція для генерації коду. Вона викликає кілька інших функцій для побудови різних секцій:
    - **makeTitle**: Генерує заголовок (наприклад, визначення моделі процесора та архітектури).
    - **makeDependenciesDeclaration**: Додає оголошення необхідних функцій і констант.
    - **makeDataSection**: Створює секцію даних.
    - **makeBeginProgramCode**: Починає секцію коду.
    - **makeInitCode**, **initMake**: Виконує ініціалізацію змінних.
    - **makeSaveHWStack**, **makeResetHWStack**: Зберігає та відновлює стек на апаратному рівні.
    - **makeEndProgramCode**: Додає фінальні інструкції (наприклад, ret для завершення програми).

**4. Маніпуляція з токенами**

* **detectMultiToken**:
  + Перевіряє, чи відповідає поточна лексема багатокомпонентному токену з таблиці tokenStruct.
* **createMultiToken**:
  + Створює багатокомпонентний токен і зберігає його у структурі LexemInfo.

**5. Генерація машинного коду**

* **outBytes2Code**:
  + Копіює байти з одного буфера до іншого, формуючи машинний код.
* **Пример генерації команд:**
  + **makeSaveHWStack**:
    - Генерує інструкцію mov ebp, esp для збереження стека.
  + **makeResetHWStack**:
    - Генерує інструкцію mov esp, ebp для відновлення стека.

**Як працює генерація коду в функції makeCode**

Функція makeCode поступово трансформує лексеми з таблиці LexemInfo у машинний код або інший низькорівневий формат. У цьому поясненні з кодовими вставками розглянемо, як саме це реалізовано.

**1. Ініціалізація**

На початку функція викликає кілька підфункцій для створення основних секцій коду:

currBytePtr = makeTitle(lastLexemInfoInTable, currBytePtr);

currBytePtr = makeDependenciesDeclaration(lastLexemInfoInTable, currBytePtr);

currBytePtr = makeDataSection(lastLexemInfoInTable, currBytePtr);

currBytePtr = makeBeginProgramCode(lastLexemInfoInTable, currBytePtr);

* **makeTitle**: Генерує заголовок програми
* **makeDependenciesDeclaration**: Додає секцію залежностей (наприклад, бібліотеки або модулі).
* **makeDataSection**: Додає секцію даних (глобальні змінні, константи тощо).
* **makeBeginProgramCode**: Додає інструкції для ініціалізації, наприклад, налаштування стеку чи регістрів.

**2. Ініціалізація стеку**

Перед початком основної генерації коду функція скидає тимчасовий стек і генерує інструкції для ініціалізації:

lexemInfoTransformationTempStackSize = 0;

currBytePtr = makeInitCode(lastLexemInfoInTable, currBytePtr);

currBytePtr = initMake(lastLexemInfoInTable, currBytePtr);

currBytePtr = makeSaveHWStack(lastLexemInfoInTable, currBytePtr);

* **makeInitCode**: Генерує код для ініціалізації змінних.

**3. Обробка лексем у циклі**

Основна логіка генерації знаходиться в циклі for, де кожна лексема обробляється залежно від її типу:

for (struct LexemInfo\* lastLexemInfoInTable\_;

lastLexemInfoInTable\_ = \*lastLexemInfoInTable,

(\*lastLexemInfoInTable)->lexemStr[0] != '\0'; ) {

Цей цикл ітерує через таблицю лексем, поки не зустріне лексему з порожнім рядком (lexemStr[0] == '\0').

**4. Генерація коду для конструкцій**

В залежності від лексеми, викликаються функції-генератори. Наприклад:

**Умовні оператори:**

IF\_THEN\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

ELSE\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

* **IF\_THEN\_CODER**: Додає інструкції для умовного оператора if.
* **ELSE\_CODER**: Генерує код для гілки else.

**Цикли:**

FOR\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

WHILE\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

REPEAT\_UNTIL\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

* **FOR\_CODER**: Генерує код для циклу for.
* **WHILE\_CODER**: Генерує інструкції для циклу while.
* **REPEAT\_UNTIL\_CODER**: Обробляє конструкцію циклу repeat until.

**Операції та оператори:**

ADD\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

SUB\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

MUL\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

DIV\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

MOD\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

* Генерація арифметичних операцій (+, -, \*, /, %).

**Логічні оператори:**

AND\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

OR\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

NOT\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

* Логічні оператори &&, ||, !.

**Інші оператори:**

INPUT\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

OUTPUT\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

* **INPUT\_CODER**: Обробляє введення.
* **OUTPUT\_CODER**: Обробляє виведення.

**5. Обробка помилок**

Якщо лексема не була оброблена жодною з функцій-генераторів, генерується помилка:

if (lastLexemInfoInTable\_ == \*lastLexemInfoInTable) {

printf("\r\nError in the code generator! \"%s\" - unexpected token!\r\n", (\*lastLexemInfoInTable)->lexemStr);

exit(0);

}

Це простий механізм обробки помилок, який завершує програму з повідомленням про неочікувану лексему.

**6. Завершення програми**

Після обробки всіх лексем функція генерує завершальні інструкції:

currBytePtr = makeResetHWStack(lastLexemInfoInTable, currBytePtr);

currBytePtr = makeEndProgramCode(lastLexemInfoInTable, currBytePtr);

* **makeResetHWStack**: Відновлює стан стеку.
* **makeEndProgramCode**: Додає фінальні інструкції, наприклад, завершення виконання.

**7. Виведення коду**

Функція viewCode виводить згенерований код форматі (шістнадцяткові):

void viewCode(unsigned char\* outCodePtr, size\_t outCodePrintSize, unsigned char align) {

printf("\r\n; +0x0 +0x1 +0x2 +0x3 +0x4 +0x5 +0x6 +0x7 +0x8 +0x9 +0xA +0xB +0xC +0xD +0xE +0xF ");

printf("\r\n;0x00000000: ");

// Вивід кожного байта

# НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА

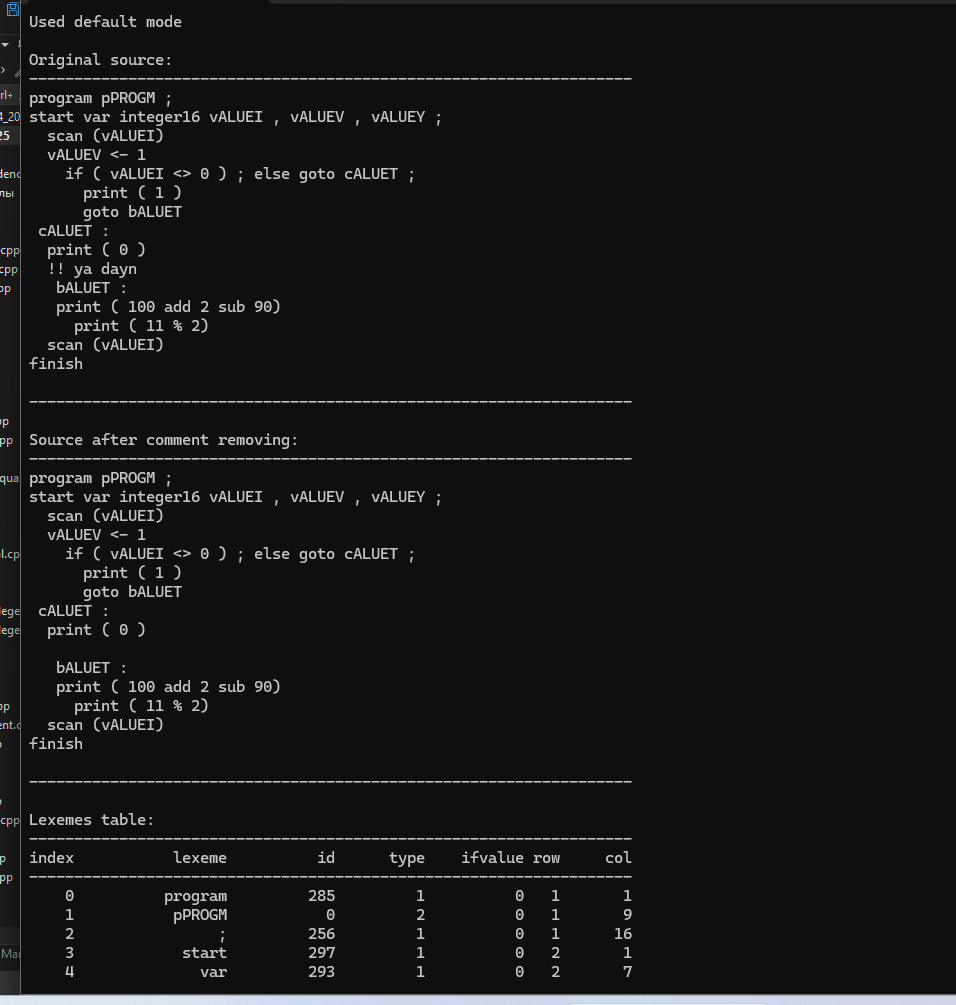
Будь-яке програмне забезпечення необхідно протестувати і налагодити. Після опрацювання синтаксичних і семантичних помилок необхідно переконатися, що розроблене програмне забезпечення функціонує так, як очікувалось.

Для перевірки коректності роботи розробленого транслятора необхідно буде написати тестові задачі на вхідній мові програмування, отримати код на мові програмування С і переконатись, що він працює правильно.

## Опис інтерфейсу та інструкції користувачу.

Розроблений транслятор має простий консольний інтерфейс.

При запуску програми обирається базовий файл fille1.z10:

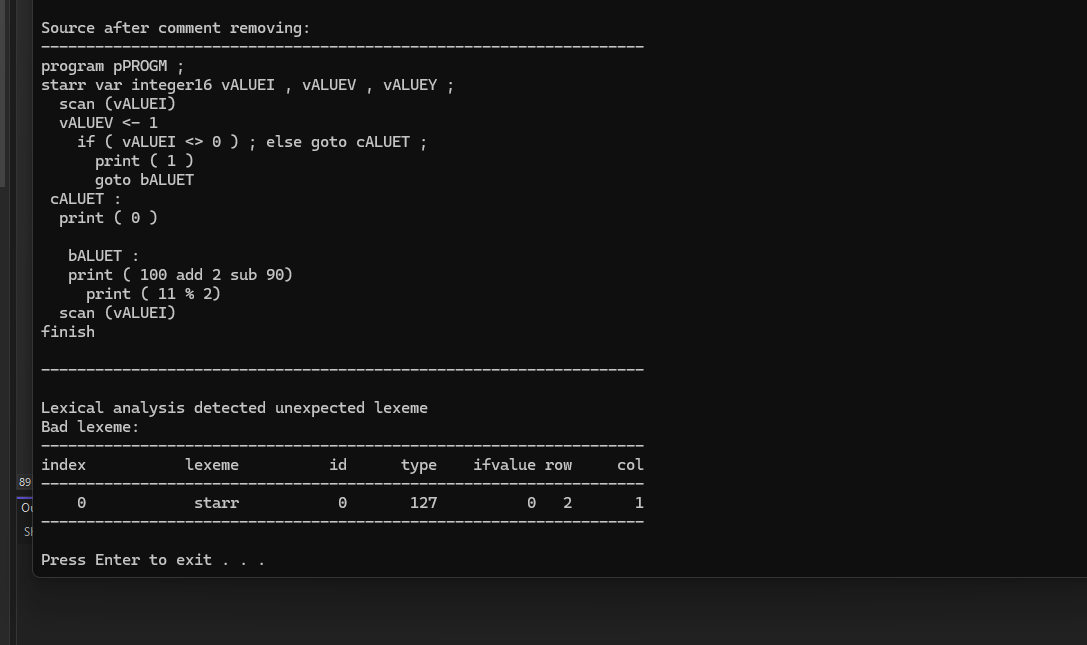


*Рис. 4.2.Результати роботи розробленого транслятора.*

## Виявлення лексичних і синтаксичних помилок.

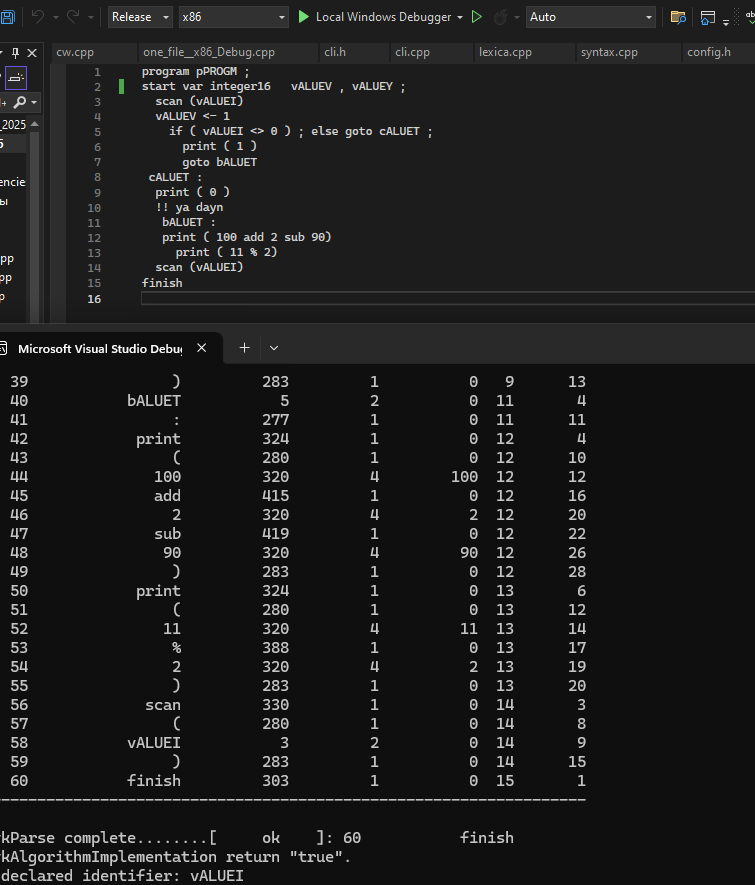
Помилки у вхідній програмі виявляються на етапі синтаксичного і семантичного аналізу.

Наприклад, у програмі зробимо синтаксичну помилку:



*Рис. 4.3. Вивід інформації про синтаксичну помилку.*

Зробимо семантичну помилку – не оголосимо змінну “ vALUEI”:



*Рис. 4.4. Вивід інформації про семантичну помилку.*

## Перевірка роботи транслятора за допомогою тестових задач.

**Тестова програма «*Лінійний алгоритм*»**

1. Ввести два числа А і В (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання).
2. Обрахувати значення виразу

Х = (А - В) \* 10 + (А + В) / 10

1. Вивести значення Х на екран.

Напишемо програму на вхідній мові програмування:

program pPROGM ;

start var integer16 aVVVVV , bVVVVV , xVVVVV ;

scan (aVVVVV)

scan (bVVVVV)

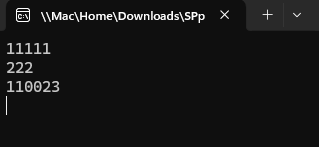
xVVVVV <- 10 \* (aVVVVV sub bVVVVV) add (aVVVVV add bVVVVV) / 10

print ( xVVVVV )

scan (aVVVVV)

finish

На мові Assembler протестуємо у новому проекті у середовищі Visual Studio 2022 і отримаємо такі результати:



*Рис. 4.5. Результати виконання тестової задачі 1.*

**Тестова програма «*Алгоритм з розгалуженням*»**

1. Ввести три числа А, В, С (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання).

Використання вкладеного умовного оператора:

1. Знайти найбільше з них і вивести його на екран.

Використання простого умовного оператора:

1. Вивести на екран число 1, якщо усі числа однакові інакше вивести 0.

Напишемо програму на вхідній мові програмування:

program pPROGM ;

start var integer16 aVVVVV , bVVVVV , cVVVVV ;

scan (aVVVVV)

scan (bVVVVV)

scan (cVVVVV)

if ( aVVVVV = bVVVVV ) ; else goto cALUET ;

goto bALUET

cALUET :

print ( 0 )

goto eNASDF

bALUET :

if ( aVVVVV = cVVVVV ) ; else goto cALUET ;

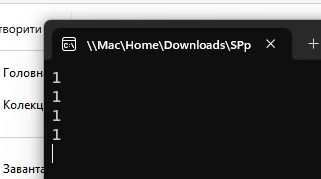
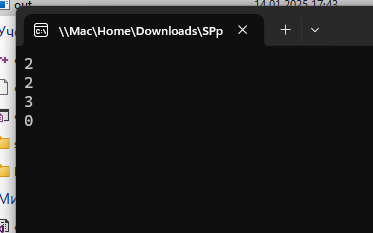
print ( 1 )

eNASDF:

scan (aVVVVV)

finish

Отриманий код на мові assembler протестуємо у новому проекті у середовищі Visual Studio 2022 і отримаємо такі результати:



*Рис. 4.6. Результати виконання тестової задачі 2.*

# ВИСНОВКИ

В процесі виконання курсового проекту було виконано наступне:

1.​Складено формальний опис мови програмування z10, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2.​Створено, а саме:

2.1.​Розроблено прямий лексичний аналізатор, орієнтований на розпізнавання лексем, що є заявлені в формальному описі мови програмування.

2.2.​Розроблено синтаксичний аналізатор на основі низхідного методу. Складено деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура

2.3.​Розроблено генератор коду, відповідні процедури якого викликаються після перевірки синтаксичним аналізатором коректності запису чергового оператора, мови програмування p24. Вихідним кодом генератора є програма на мові Assembler(x86).

3.​Проведене тестування компілятора на тестових програмах за наступними пунктами:

3.1.​На виявлення лексичних помилок.

3.2.​На виявлення синтаксичних помилок.

3.3.​Загальна перевірка роботи компілятора.

В результаті виконання даної курсового проекту було  засвоєно методи розробки та реалізації компонент систем програмування.

# СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс]

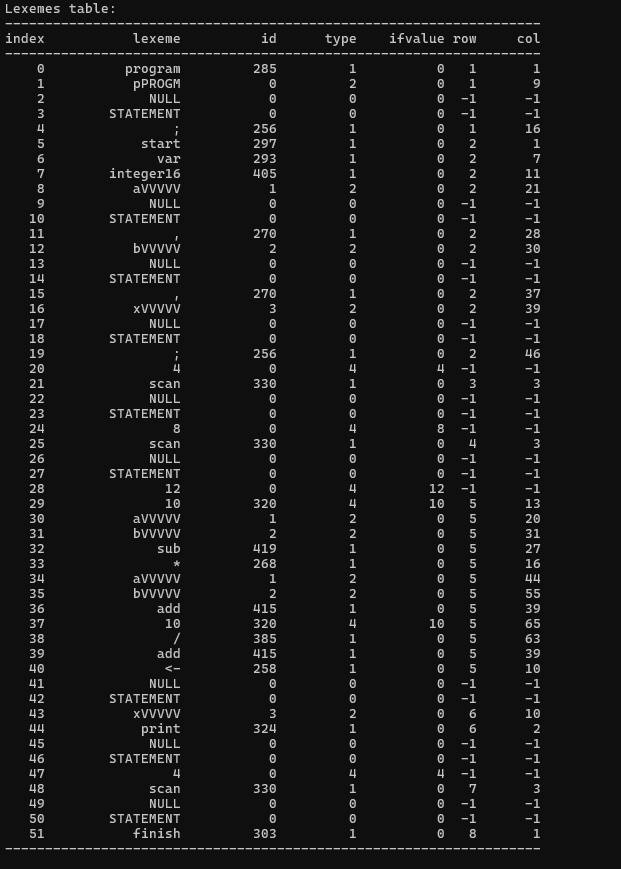
: навч. посіб. для студ. спеціальності 123 – «Комп’ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. – Київ: КПІ ім. Ігоря Сікорського, 2021. – 108 с.

1. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. – Харків: НТУ «ХПІ», 2021. – 133 с.
2. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії формальних мов: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 c.
3. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 c.
4. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. – 1038 c.
5. Системне програмування (курсовий проект) [Електронний ресурс] – Режим доступу до ресурсу: https://vns.lpnu.ua/course/view.php?id=11685.
6. MIT OpenCourseWare. Computer Language Engineering [Електронний ресурс] – Режим доступу до ресурсу: https://ocw.mit.edu/courses/6-035-computer- language-engineering-spring-2010.

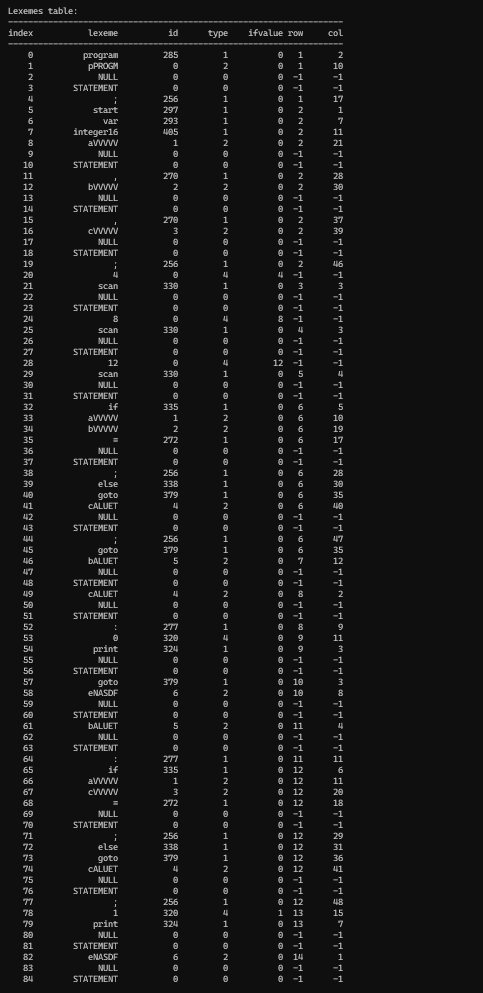
# ДОДАТКИ

А. Таблиці лексем для тестових прикладів

Тестова програма «лінійного алгоритму»



Тестова програма «Алгоритм з розгалуженням»



В. С код (або код на асемблері), отриманий на виході транслятора для тестових прикладів

Тестова програма «Лінійний алгоритм»

.model flat,stdcall

option casemap:none

GetStdHandle proto STDCALL,nStdHandle:DWORD

ExitProcess proto STDCALL,uExitCode:DWORD

ReadConsoleA proto STDCALL,hConsoleInput:DWORD,lpBuffer:DWORD,nNumberOfCharsToRead:DWORD,lpNumberOfCharsRead:DWORD,lpReserved:DWORD

WriteConsoleA proto STDCALL,hConsoleOutput:DWORD,lpBuffert:DWORD,nNumberOfCharsToWrite:DWORD,lpNumberOfCharsWritten:DWORD,lpReserved:DWORD

wsprintfA PROTO C:VARARG

GetConsoleMode PROTO STDCALL,hConsoleHandle:DWORD,lpMode:DWORD

SetConsoleMode PROTO STDCALL,hConsoleHandle:DWORD,dwMode:DWORD

ENABLE\_LINE\_INPUT EQU 0002h

ENABLE\_ECHO\_INPUT EQU 0004h

.data

data\_start db 8192 dup(0)

valueTemp\_msg db 256 dup(0)

valueTemp\_fmt db "%d",10,13,0

hConsoleInput dd 0

hConsoleOutput dd 0

buffer db 128 dup(0)

readOutCount dd ?

.code

start:

db 0E8h,00h,00h,00h,00h;call NexInstruction

pop esi

sub esi,5

mov edi,esi

add edi,000004000h

mov ecx,edi

add ecx,512

jmp initConsole

putProc PROC

push eax

push offset valueTemp\_fmt

push offset valueTemp\_msg

call wsprintfA

add esp,12

push 0

push 0

push eax

push offset valueTemp\_msg

push hConsoleOutput

call WriteConsoleA

ret

putProc ENDP

getProc PROC

push ebp

mov ebp,esp

push 0

push offset readOutCount

push 15

push offset buffer+1

push hConsoleInput

call ReadConsoleA

lea esi,offset buffer

add esi,readOutCount

sub esi,2

call string\_to\_int

mov esp,ebp

pop ebp

ret

getProc ENDP

string\_to\_int PROC

xor eax,eax

mov ebx,1

xor ecx,ecx

convert\_loop:

movzx ecx,byte ptr[esi]

test ecx,ecx

jz done

sub ecx,'0'

imul ecx,ebx

add eax,ecx

imul ebx,ebx,10

dec esi

jmp convert\_loop

done:

ret

string\_to\_int ENDP

initConsole:

push -10

call GetStdHandle

mov hConsoleInput,eax

push -11

call GetStdHandle

mov hConsoleOutput,eax

mov ebp,esp

mov eax,edi

add eax,000000000h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,edi

add eax,000000004h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,edi

add eax,000000008h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,edi

add eax,00000000Ch

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

add ecx,4

mov eax,000000004h

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx]

mov edx,000000044h

add edx,esi

push ecx

push esi

push edi

call edx

pop edi

pop esi

mov ebx,dword ptr[ecx]

sub ecx,4

add ebx,edi

mov dword ptr[ebx],eax

mov ecx,edi

add ecx,512

add ecx,4

mov eax,000000008h

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx]

mov edx,000000044h

add edx,esi

push ecx

push esi

push edi

call edx

pop edi

pop esi

mov ebx,dword ptr[ecx]

sub ecx,4

add ebx,edi

mov dword ptr[ebx],eax

mov ecx,edi

add ecx,512

add ecx,4

mov eax,00000000Ch

mov dword ptr[ecx],eax

add ecx,4

mov eax,00000000Ah

mov dword ptr[ecx],eax

mov eax,edi

add eax,000000004h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,edi

add eax,000000008h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx]

sub ecx,4

sub dword ptr[ecx],eax

mov eax,dword ptr[ecx]

mov eax,dword ptr[ecx-4]

imul dword ptr[ecx]

sub ecx,4

mov dword ptr[ecx],eax

mov eax,edi

add eax,000000004h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,edi

add eax,000000008h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx]

sub ecx,4

add dword ptr[ecx],eax

mov eax,dword ptr[ecx]

add ecx,4

mov eax,00000000Ah

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx-4]

cdq

idiv dword ptr[ecx]

sub ecx,4

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx]

sub ecx,4

add dword ptr[ecx],eax

mov eax,dword ptr[ecx]

mov eax,dword ptr[ecx]

mov ebx,dword ptr[ecx-4]

sub ecx,8

add ebx,edi

mov dword ptr[ebx],eax

mov ecx,edi

add ecx,512

mov eax,edi

add eax,00000000Ch

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx]

mov edx,00000001Bh

add edx,esi

push esi

push edi

call edx

pop edi

pop esi

mov ecx,edi

add ecx,512

add ecx,4

mov eax,000000004h

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx]

mov edx,000000044h

add edx,esi

push ecx

push esi

push edi

call edx

pop edi

pop esi

mov ebx,dword ptr[ecx]

sub ecx,4

add ebx,edi

mov dword ptr[ebx],eax

mov ecx,edi

add ecx,512

mov esp,ebp

xor eax,eax

ret

end startа

Тестова програма «Алгоритм з розгалуженням»

Код на мові assembler:

.686

.model flat, stdcall

option casemap : none

GetStdHandle proto STDCALL, nStdHandle : DWORD

ExitProcess proto STDCALL, uExitCode : DWORD

;MessageBoxA PROTO hwnd : DWORD, lpText : DWORD, lpCaption : DWORD, uType : DWORD

ReadConsoleA proto STDCALL, hConsoleInput : DWORD, lpBuffer : DWORD, nNumberOfCharsToRead : DWORD, lpNumberOfCharsRead : DWORD, lpReserved : DWORD

WriteConsoleA proto STDCALL, hConsoleOutput : DWORD, lpBuffert : DWORD, nNumberOfCharsToWrite : DWORD, lpNumberOfCharsWritten : DWORD, lpReserved : DWORD

wsprintfA PROTO C : VARARG

GetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, lpMode : DWORD

SetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, dwMode : DWORD

ENABLE\_LINE\_INPUT EQU 0002h

ENABLE\_ECHO\_INPUT EQU 0004h

.data

data\_start db 8192 dup (0)

;title\_msg db "Output:", 0

valueTemp\_msg db 256 dup(0)

valueTemp\_fmt db "%d", 10, 13, 0

;NumberOfCharsWritten dd 0

hConsoleInput dd 0

hConsoleOutput dd 0

buffer db 128 dup(0)

readOutCount dd ?

.code

start:

db 0E8h, 00h, 00h, 00h, 00h; call NexInstruction

;NexInstruction:

pop esi

sub esi, 5

mov edi, esi

add edi, 000004000h

mov ecx, edi

add ecx, 512

jmp initConsole

putProc PROC

push eax

push offset valueTemp\_fmt

push offset valueTemp\_msg

call wsprintfA

add esp, 12

;push 40h

;push offset title\_msg

;push offset valueTemp\_msg;

;push 0

;call MessageBoxA

push 0

push 0; offset NumberOfCharsWritten

push eax; NumberOfCharsToWrite

push offset valueTemp\_msg

push hConsoleOutput

call WriteConsoleA

ret

putProc ENDP

getProc PROC

push ebp

mov ebp, esp

push 0

push offset readOutCount

push 15

push offset buffer + 1

push hConsoleInput

call ReadConsoleA

lea esi, offset buffer

add esi, readOutCount

sub esi, 2

call string\_to\_int

mov esp, ebp

pop ebp

ret

getProc ENDP

string\_to\_int PROC

; input: ESI - string

; output: EAX - value

xor eax, eax

mov ebx, 1

xor ecx, ecx

convert\_loop :

movzx ecx, byte ptr[esi]

test ecx, ecx

jz done

sub ecx, '0'

imul ecx, ebx

add eax, ecx

imul ebx, ebx, 10

dec esi

jmp convert\_loop

done:

ret

string\_to\_int ENDP

initConsole:

push -10

call GetStdHandle

mov hConsoleInput, eax

push -11

call GetStdHandle

mov hConsoleOutput, eax

;push ecx

;push ebx

;push esi

;push edi

;push offset mode

;push hConsoleInput

;call GetConsoleMode

;mov ebx, eax

;or ebx, ENABLE\_LINE\_INPUT

;or ebx, ENABLE\_ECHO\_INPUT

;push ebx

;push hConsoleInput

;call SetConsoleMode

;pop edi

;pop esi

;pop ebx

;pop ecx

;hw stack save(save esp)

mov ebp, esp

;"pPROGM"

mov eax, edi

add eax, 000000000h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;null statement (non-context)

;";"

;"aVVVVV"

mov eax, edi

add eax, 000000004h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;null statement (non-context)

;"bVVVVV"

mov eax, edi

add eax, 000000008h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;null statement (non-context)

;"cVVVVV"

mov eax, edi

add eax, 00000000Ch

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;null statement (non-context)

;";"

;"4"

add ecx, 4

mov eax, 000000004h

mov dword ptr [ecx], eax

;"scan"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"8"

add ecx, 4

mov eax, 000000008h

mov dword ptr [ecx], eax

;"scan"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"12"

add ecx, 4

mov eax, 00000000Ch

mov dword ptr [ecx], eax

;"scan"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"if"

;"aVVVVV"

mov eax, edi

add eax, 000000004h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"bVVVVV"

mov eax, edi

add eax, 000000008h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"="

mov eax, dword ptr[ecx]

sub ecx, 4

cmp dword ptr[ecx], eax

sete al

and eax, 1

mov dword ptr[ecx], eax

;after cond expresion (after "if")

cmp eax, 0

jz LABEL@AFTER\_THEN\_000000002193DDC8

;";" (after "then"-part of if-operator)

mov eax, 1

LABEL@AFTER\_THEN\_000000002193DDC8:

;"else"

cmp eax, 0

jnz LABEL@AFTER\_ELSE\_000000002193EA40

;"goto" previous ident "cALUET"(as label)

jmp LABEL@00000000223C7B50

;null statement (non-context)

;";" (after "else")

LABEL@AFTER\_ELSE\_000000002193EA40:

;"goto" previous ident "bALUET"(as label)

jmp LABEL@00000000223C0E08

;null statement (non-context)

;ident "cALUET"(as label) previous ":"

LABEL@00000000223C7B50:

;"0"

add ecx, 4

mov eax, 000000000h

mov dword ptr [ecx], eax

;"print"

mov eax, dword ptr[ecx]

mov edx, 00000001Bh

add edx, esi

;push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

;pop ecx

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"goto" previous ident "eNASDF"(as label)

jmp LABEL@00000000223C0E58

;null statement (non-context)

;ident "bALUET"(as label) previous ":"

LABEL@00000000223C0E08:

;"if"

;"aVVVVV"

mov eax, edi

add eax, 000000004h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"cVVVVV"

mov eax, edi

add eax, 00000000Ch

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"="

mov eax, dword ptr[ecx]

sub ecx, 4

cmp dword ptr[ecx], eax

sete al

and eax, 1

mov dword ptr[ecx], eax

;after cond expresion (after "if")

cmp eax, 0

jz LABEL@AFTER\_THEN\_00000000219466F0

;";" (after "then"-part of if-operator)

mov eax, 1

LABEL@AFTER\_THEN\_00000000219466F0:

;"else"

cmp eax, 0

jnz LABEL@AFTER\_ELSE\_0000000021947368

;"goto" previous ident "cALUET"(as label)

jmp LABEL@00000000223C7B50

;null statement (non-context)

;";" (after "else")

LABEL@AFTER\_ELSE\_0000000021947368:

;"1"

add ecx, 4

mov eax, 000000001h

mov dword ptr [ecx], eax

;"print"

mov eax, dword ptr[ecx]

mov edx, 00000001Bh

add edx, esi

;push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

;pop ecx

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;ident "eNASDF"(as label) previous ":"

LABEL@00000000223C0E58:

;"4"

add ecx, 4

mov eax, 000000004h

mov dword ptr [ecx], eax

;"scan"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;hw stack reset(restore esp)

mov esp, ebp

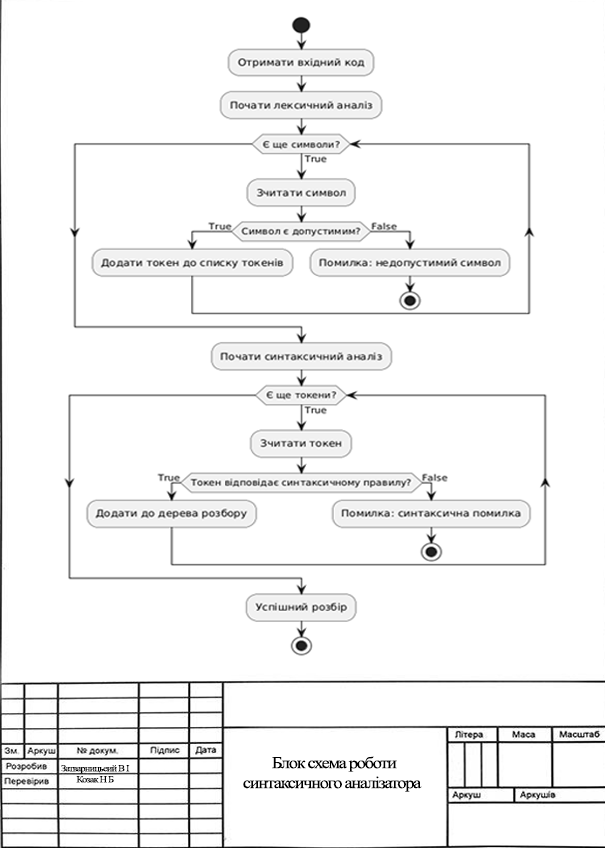
xor eax, eax

ret

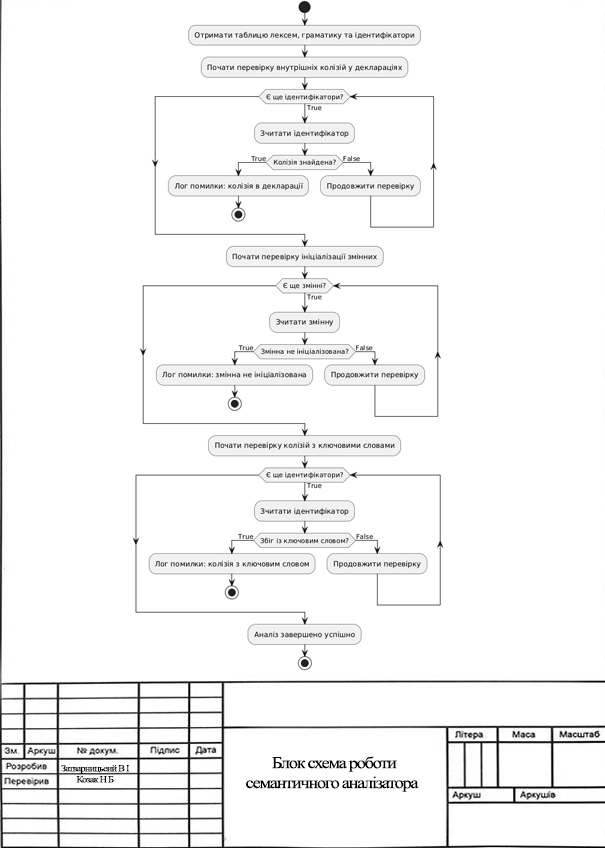
end start

С. графічні представлення алгоритмів роботи

Блок схема роботи синтаксичного аналізатора



Блок схема роботи семантичного аналізатора

C. дерево розбору

