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:

Луцик Н.І.

Перевірив:

старший викладач каф. ЕОМ

Козак Н.Б.

Львів-2024

**ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ**

1. Цільова мова транслятора – мова програмування С або асемблер для 32/64 розрядного процесора.

2. Для отримання виконавчого файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio бо будь-яким іншим.

3. Мова розробки транслятора: С/C++.

4. Реалізувати 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 асемблера;*
* *об’єктний файл;*
* *виконавчий файл.*

7. 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 операції круглі дужки (); у якості операндів можуть бути цілі константи, змінні, а також інші вирази.
3. В кожному завданні обов’язковим є оператор типу “блок” (складений оператор), його вигляд має бути таким, як і блок тіла програми.
4. Необхідно реалізувати задані варіантом оператори, синтаксис операторів наведено у таблиці 1.1. Синтаксис вхідної мови має забезпечити реалізацію обчислень лінійних алгоритмів, алгоритмів з розгалуженням і циклічних алгоритмів. Опис формальної мови студент погоджує з викладачем.
5. Оператори можуть бути довільної вкладеності і в будь-якій послідовності.
6. Для перевірки роботи розробленого транслятора, необхідно написати три тестові програми на вхідній мові програмування.

**Деталізований опис власної мови програмування:**

* Тип даних: Int32
* Блок тіла програми: StartProgram; StartBlok Variable…; EndBlok
* Оператор вводу: Scan ()
* Оператор виводу: Print ()
* Оператори: If-else (C)

Goto (C)

For-To (Паскаль)

For-Downto (Паскаль)

While (Бейсік)

Repeat-Until (Паскаль)

* Регістр ключових слів: Up-Low перший символ Up
* Регістр ідентифікаторів: Low-Up2 перший символ \_
* Операції арифметичні: ++, --, \*\*, Div, Mod
* Операції порівняння: Eq, Neq, Ls, Gr
* Операції логічні: Not, And, Or
* Коментар: !! …!!
* Ідентифікатори змінних, числові константи
* Оператор присвоєння: <-

АНОТАЦІЯ

У цьому курсовому проєкті створено транслятор, який здійснює перетворення вхідної мови, визначеної варіантом завдання, у мову асемблера. Трансляція включає три ключові етапи: лексичний аналіз, синтаксичний аналіз та генерацію коду.

Під час лексичного аналізу вхідний потік символів розбивається на лексеми, що заносяться до спеціальної таблиці. Для кожної лексеми генерується унікальний числовий ідентифікатор, що спрощує подальшу обробку. До таблиці також додається додаткова інформація: номер рядка, значення (у разі числової лексеми) та інші важливі параметри.

Синтаксичний аналіз здійснюється за допомогою висхідного методу без повернення, що дозволяє побудувати дерево розбору, просуваючись від листків до кореня. Цей етап забезпечує правильне структурування даних відповідно до синтаксичних правил мови.

На етапі генерації коду обробляється таблиця лексем для створення асемблерного коду, що відповідає кожному блоку. Згенерований код записується у вихідний файл і готовий до компіляції.

Отриманий асемблерний код можна скомпілювати та виконати за допомогою інструментів, таких як LINK, ML тощо.

**ЗМІСТ**

**ВСТУП**

Термін «транслятор» відноситься до програмного забезпечення, яке здійснює перетворення вихідного коду, написаного однією мовою програмування, у еквівалентний код іншої. Якщо вхідна мова є мовою високого рівня, а вихідна — мова асемблера або машинний код, такий транслятор називається компілятором.

Транслятори поділяються на два основних види: компілятори та інтерпретатори. Процес компіляції включає два основні етапи: аналіз і синтез. На етапі аналізу вихідний код розбивається на лексеми, перевіряється його синтаксична правильність і створюється проміжне представлення програми. Під час синтезу це представлення перетворюється на об'єктний код, який складається з машинних інструкцій і може виконуватися безпосередньо на комп’ютері.

Інтерпретатори, на відміну від компіляторів, не створюють окремого виконуваного файлу. Вони аналізують програму, формують її проміжне представлення, але не генерують об'єктного коду. Інтерпретатор виконує команди програми безпосередньо під час її обробки.

Компілятор дозволяє перетворювати вихідний код з однієї мови програмування на іншу. Вхідними даними для компілятора є текст вихідної програми, а результатом — об'єктний код, адаптований для конкретної апаратної платформи. Варто зазначити, що сам компілятор може бути створений на третій мові програмування, що демонструє його універсальність і гнучкість у розробці програмного забезпечення.

1. **ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ**

Транслятор — це програмне забезпечення, яке перетворює вихідний код, написаний на одній мові програмування, у робочий код, представлений об’єктною мовою. До трансляторів належать різні види програм, кожна з яких має свої особливості у процесі трансляції. Сучасні транслятори поділяються на три основні категорії: асемблери, компілятори та інтерпретатори.

**Асемблер** — це інструмент, який перетворює символічний код на машинні інструкції. Його ключова особливість полягає в тому, що кожна команда символічного коду безпосередньо відповідає одній машинній інструкції.

**Компілятор** — це програма, яка транслює вихідний код з мови високого рівня у машинний код. У порівнянні з асемблером, компілятор виконує складніше перетворення, адаптуючи код для конкретної архітектури.

**Інтерпретатор** — це програма, що послідовно виконує команди вихідного коду. На відміну від компілятора, інтерпретатор не створює окремого виконуваного файлу, а обробляє інструкції під час виконання, що полегшує процес тестування і налагодження.

Процес трансляції включає кілька етапів:

* **Лексичний аналіз**. Вхідний код розбивається на лексеми, що представляють окремі слова та символи мови програмування. На цьому етапі також виявляються помилки, наприклад, некоректні символи чи неправильний формат ідентифікаторів.
* **Синтаксичний аналіз**. Створюється синтаксичне дерево, яке відображає структуру програми відповідно до граматики мови. Для цього використовуються методи аналізу, такі як LL(1) чи LR(1).
* **Семантичний аналіз**. Перевіряються логічні залежності між елементами програми, які не описуються синтаксисом. Це включає перевірку типів даних, областей видимості та правильність параметрів функцій.
* **Оптимізація коду**. Поліпшується ефективність виконання програми. Оптимізація може бути локальною чи глобальною, а також залежною чи незалежною від конкретної апаратної архітектури.
* **Генерація коду**. Завершується створенням об’єктного або асемблерного коду, готового для подальшої компіляції чи виконання.

Фази трансляції можуть поєднуватися або бути скороченими залежно від особливостей реалізації транслятора. Наприклад, у простих однопрохідних трансляторах проміжне представлення та оптимізація можуть бути відсутні, а інші етапи часто об'єднуються.

На етапі лексичного аналізу формується таблиця, яка містить ідентифікатори, рядки та числові значення. Синтаксичний аналіз генерує дерево розбору, що використовується для оптимізації та генерації коду. Контекстний аналіз забезпечує коректність типів, контроль областей видимості та відповідність параметрів функцій.

Результатом всіх етапів трансляції є об’єктний код, оптимізований для конкретної апаратної архітектури, який може бути скомпільований і виконаний на цільовій платформі. Кожен етап процесу забезпечує послідовне перетворення вихідного коду у робочий виконуваний файл, готовий до запуску.

1. Формальний опис вхідної мови програмування
   1. Деталізований опис вхідної мови в термінах розширеної нотації 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 = "Int32"

tokenCOMMA = ","

tokenNOT = "Not"

tokenAND = "And"

tokenOR = "Or"

tokenEQUAL = "Eq"

tokenNOTEQUAL = "Neq"

tokenPLUS = "++"

tokenMINUS = "--"

tokenMUL = "\*\*"

tokenDIV = "Div"

tokenMOD = "Mod"

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 = "StartProgram"

tokenBODY = "StartBlok"

tokenDATA = "Variable"

tokenEND = "EndBlok"

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"

1. Розробка транслятора з вхідної мови програмування
   1. 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.

* 1. Проектування таблиць транслятора та вибір структур даних.

**Опис структури LexemInfo**

LexemInfo — це структура, яка використовується для зберігання інформації про лексеми, отримані під час лексичного аналізу. Вона надає публічний доступ до своїх членів і створена для зручного доступу до властивостей кожної лексеми. Детальний опис її елементів та функцій наведено нижче.

**Члени структури:**

1. **char lexemStr[MAX\_LEXEM\_SIZE]**  
   Масив символів, у якому зберігається лексема у вигляді рядка.  
   MAX\_LEXEM\_SIZE визначає максимальну довжину лексеми, зазвичай задається як константа.
2. **unsigned long long int lexemId**  
   Унікальний ідентифікатор лексеми, який дозволяє відрізняти її від інших.
3. **unsigned long long int tokenType**  
   Тип токена, до якого належить лексема (наприклад, константа, оператор чи ключове слово).
4. **unsigned long long int ifvalue**  
   Додаткове значення, яке використовується в умовних виразах або для контекстуального аналізу.  
   Наприклад, це може бути значення для порівнянь чи виконання умов.
5. **unsigned long long int row**  
   Номер рядка у вихідному коді, де знаходиться лексема. Це корисно для діагностики або виведення повідомлень про помилки.
6. **unsigned long long int col**  
   Номер колонки у рядку, де розташована лексема.
7. **Коментар або місце для розширення**  
   Позначка // TODO: ... передбачає можливість додавання нових полів для розширення функціональності структури.

**Конструктори:**

1. **Конструктор за замовчуванням: LexemInfo()**  
   Ініціалізує всі поля структури стандартними значеннями (наприклад, нулями чи порожніми рядками).
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)**  
   Створює об’єкт з переданими значеннями:
   * lexemStr — рядок лексеми.
   * lexemId — унікальний ідентифікатор.
   * tokenType — тип токена.
   * ifvalue — додаткове значення.
   * row і col — позиція лексеми у вихідному коді.
3. **Конструктор копіювання: LexemInfo(const NonContainedLexemInfo& nonContainedLexemInfo)**  
   Ініціалізує об'єкт на основі іншої структури NonContainedLexemInfo, дозволяючи зберегти сумісність з іншими форматами даних.

**Призначення:**

Структура LexemInfo забезпечує:

* Збереження інформації про токени під час лексичного аналізу.
* Легкий доступ до позиції лексеми (рядок і колонка) для генерації звітів чи повідомлень про помилки.
* Структурування даних, необхідних для побудови синтаксичного дерева.
* Розширення для семантичного аналізу чи інших етапів компіляції.
  1. Розробка лексичного аналізатора.

Метою лексичного аналізу є розбиття вхідного тексту, який складається з послідовності символів, на окремі слова або лексеми. Це передбачає виокремлення слів із суцільної послідовності символів. Усі символи вхідного тексту поділяються на дві категорії: ті, що є частинами лексем, і ті, що розділяють лексеми.

Для цього зазвичай використовуються стандартні методи обробки рядків. Програма аналізується послідовно від початку до кінця. Основні елементи тексту, або лексичні одиниці, розділяються пробілами, знаками операцій та спеціальними символами (наприклад, переведення рядка чи табуляція). Таким чином визначаються та розпізнаються:

* **Ідентифікатори**
* **Літерали**
* **Термінальні символи** (зокрема операції та ключові слова).

Після виділення лексеми вона розпізнається та заноситься до таблиці лексем за допомогою унікального номера, який відповідає кожній лексемі із загального набору. Це дозволяє наступним фазам компіляції працювати з лексемами не як із послідовностями символів, а як із їх унікальними номерами. Такий підхід значно спрощує роботу синтаксичного аналізатора, зокрема:

* Полегшує перевірку належності лексеми до синтаксичної конструкції.
* Дозволяє зручно переглядати програму як вперед, так і назад від поточної позиції аналізу.

Окрім унікального номера, таблиця лексем також містить інформацію про рядок, у якому знаходиться відповідна лексема (це корисно для вказівки місця помилки), а також інші додаткові дані.

Під час лексичного аналізу коментарі відкидаються, оскільки вони не впливають на виконання програми, синтаксичний розбір чи генерацію коду.

**Типи лексем (лексичні класи):**

1. **Ключові слова**:  
   StartProgram, Variable, EndBlok, Scan, Print, Int32, If, else, For, Goto, Downto, Repeat, Until, While.
2. **Ідентифікатори**.
3. **Числові константи**:  
   Цілі числа без знаку.
4. **Оператор присвоєння**:  
   <-.
5. **Знаки операцій**:  
   ++, --, \*\*, Div, Gr, Ls, Eq, Neq, Not, And, Or.
6. **Роздільники**:  
   ; ,
7. **Дужки**:  
   (, )
   * 1. **Розробка алгоритму роботи лексичного аналізатора.**

Даний лексичний аналізатор — це програмний модуль, який перетворює вхідний текст на послідовність лексем (основних синтаксичних одиниць) і класифікує їх за типами. Основна мета аналізатора — підготувати текст для подальших етапів аналізу, таких як синтаксичний або семантичний. У коді реалізовано функціонал для розпізнавання ключових слів, значень, ідентифікаторів, а також для обробки коментарів.

Як працює 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.

Цей лексичний аналізатор — програмний модуль, який розбиває вхідний текст на лексеми (синтаксичні одиниці) та класифікує їх за типами. Його основна задача — підготувати текст для подальшого синтаксичного та семантичного аналізу. Аналізатор має низку функцій для розпізнавання ключових слів, значень, ідентифікаторів, а також для обробки коментарів. Ось як він працює:

**Структури даних**

1. **LexemInfo** Зберігає інформацію про кожну лексему:
   * lexemStr — текст лексеми.
   * lexemId — унікальний ідентифікатор.
   * tokenType — тип токена (ключове слово, ідентифікатор, значення тощо).
   * ifvalue — додаткова інформація для числових значень.
   * row і col — позиція лексеми в тексті (рядок і стовпець).
2. **NonContainedLexemInfo** Використовується для тимчасового збереження лексем у буфері (наприклад, tempStrFor\_123).

**Основні масиви**

* lexemesInfoTable — містить усі знайдені лексеми.
* identifierIdsTable — зберігає ідентифікатори, запобігаючи дублюванню.

Етапи роботи

**1. Токенізація (tokenize)**

Функція розбиває текст на токени за допомогою регулярного виразу:

* Регулярний вираз (TOKENS\_RE) визначає правила формування токенів (ключові слова, числа, ідентифікатори тощо).
* Знайдені токени обробляються по черзі за допомогою ітератора (std::sregex\_token\_iterator).

**2. Ідентифікація токенів**

Для кожного токена викликаються функції:

* tryToGetKeyWord — перевіряє, чи є токен ключовим словом.
* tryToGetIdentifier — визначає, чи це ідентифікатор.
* tryToGetUnsignedValue — перевіряє, чи це числове значення.

Токени, які не відповідають жодному з цих шаблонів, позначаються як "непередбачувані лексеми".

**3. Обробка лексем**

* **Ключові слова** розпізнаються за допомогою регулярного виразу (KEYWORDS\_RE) та отримують унікальний lexemId.
* **Ідентифікатори** перевіряються за шаблоном (IDENTIFIERS\_RE) і додаються до identifierIdsTable.
* **Числові значення** розпізнаються регулярним виразом (UNSIGNEDVALUES\_RE) і зберігаються в полі ifvalue.

**4. Видалення коментарів**

Функція commentRemover видаляє коментарі:

* Однорядкові (//).
* Багаторядкові (/\* ... \*/). Коментарі замінюються пробілами, щоб зберегти структуру тексту.

**5. Визначення позицій лексем**

Функція setPositions встановлює рядок і стовпець для кожної лексеми, дозволяючи визначати точне місце розташування помилок у тексті.

**6. Формування результатів**

Результати аналізу подаються у вигляді таблиці(табл.1), яка включає:

* **Індекс лексеми** — номер у загальному списку.
* **Текст лексеми** — її оригінальний вигляд.
* **Ідентифікатор** — унікальний номер для кожного типу.
* **Тип лексеми** — ключове слово, ідентифікатор, значення тощо.
* **Значення** — числове значення для числових лексем.
* **Рядок і стовпець** — позиція у тексті.

**Табл.1.**

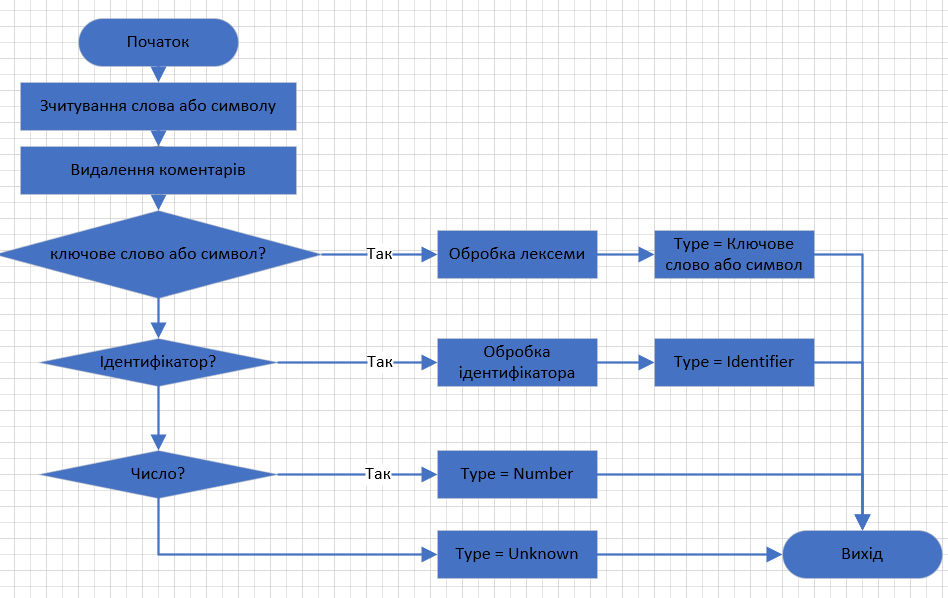
| **Індекс** | **Текст лексеми** | **Ідентифікатор** | **Тип** | **Значення** | **Рядок** | **Стовпець** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | StartProgram | 101 | Ключове слово | - | 1 | 1 |
| 1 | \_p | 1 | Ідентифікатор | - | 1 | 9 |
| 2 | StartBlok | 102 | Ключове слово | - | 2 | 1 |
| 3 | 123 | 1001 | Значення | 123 | 3 | 5 |
| 4 | EndBlok | 103 | Ключове слово | - | 4 | 1 |

**Особливості**

1. **Буферизація**: Використання буфера (tempStrFor\_123) для оптимізації пам’яті.
2. **Гнучкість**: Регулярні вирази (TOKENS\_RE, IDENTIFIERS\_RE тощо) можна легко адаптувати.
3. **Обробка помилок**: Некоректні токени автоматично позначаються як помилкові, не впливаючи на решту аналізу.

Переваги

* **Простота аналізу**: Легко виявляти помилки та несподівані лексеми завдяки координатам у тексті.
* **Універсальність**: Структурована таблиця полегшує додавання нових типів лексем.
* **Зручність**: Вся інформація про лексеми зберігається в єдиній систематизованій формі.



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

* + 1. **Опис програми реалізації лексичного аналізатора.**

Основна мета лексичного аналізатора — розділити вхідний текст програми на лексеми, тобто окремі смислові одиниці, які використовуються для подальшого аналізу. Символи у тексті поділяються на ті, що належать до лексем, і роздільники. Аналіз виконується послідовно, від початку до кінця тексту, з використанням стандартних методів обробки рядків. Базові елементи (лексичні одиниці) визначаються за допомогою пробілів, знаків операцій та спеціальних символів (наприклад, переведення рядка або табуляція). У результаті аналізатор розпізнає ідентифікатори, літерали та термінальні символи (наприклад, оператори чи ключові слова).

Робота програми триває доти, доки не буде оброблено весь текст. Для цього використовується функція tokenize(), яка зчитує вміст файлу, виділяє лексеми та порівнює їх із зарезервованими словами. У разі збігу лексема отримує відповідний тип або значення (наприклад, для числових констант).

Всі розпізнані лексеми додаються до списку m\_tokens з унікальним типом. Це дає можливість подальшим фазам компіляції працювати з лексемами як із конкретними типами, а не послідовностями символів, що значно полегшує синтаксичний аналіз. Такий підхід дозволяє легко перевіряти відповідність лексеми синтаксичним конструкціям або здійснювати навігацію текстом програми (вперед і назад від поточної позиції). У таблиці лексем також зберігаються координати лексем (номер рядка і стовпця), що спрощує пошук місця помилки. Крім того, додається метаінформація для подальших етапів аналізу.

Під час аналізу також виявляються лексичні помилки, наприклад, некоректні символи або неправильні ідентифікатори. Вони ігноруються, оскільки не впливають на синтаксичний розбір чи генерацію коду.

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

* 1. Розробка синтаксичного та семантичного аналізатора.
     1. **Розробка дерева граматичного розбору.**

Program

├── "StartProgram"

├── "StartBlok"

├── {VariableDeclaration ";"}

│ ├── VariableDeclaration

│ │ ├── "Int32"

│ │ └── VariableList

│ │ ├── Identifier

│ │ │ ├── "\_"

│ │ │ └── LowUp {2}

│ │ │ ├── Up

│ │ │ │ └── "A" | "B" | ... | "Z"

│ │ │ ├── Low

│ │ │ │ └── "a" | "b" | ... | "z"

│ │ │ └── Digit

│ │ │ └── "0" | "1" | ... | "9"

│ │ └── {"," Identifier}

├── {Statement}

│ ├── Statement

│ │ ├── InputStatement

│ │ │ ├── "Scan"

│ │ │ └── Identifier

│ │ ├── OutputStatement

│ │ │ ├── "Print"

│ │ │ └── ArithmeticExpression

│ │ │ ├── LowPriorityExpression

│ │ │ │ ├── MiddlePriorityExpression

│ │ │ │ │ ├── Identifier

│ │ │ │ │ ├── Number

│ │ │ │ │ │ ├── ["-"]

│ │ │ │ │ │ └── Digit {5}

│ │ │ │ │ └── "(" ArithmeticExpression ")"

│ │ │ │ └── {MiddlePriorityOperator MiddlePriorityExpression}

│ │ │ └── {LowPriorityOperator LowPriorityExpression}

│ │ ├── AssignStatement

│ │ │ ├── ArithmeticExpression

│ │ │ └── "==>" Identifier

│ │ ├── IfElseStatement

│ │ │ ├── "If"

│ │ │ ├── "(" LogicalExpression ")"

│ │ │ │ ├── AndExpression

│ │ │ │ │ ├── Comparison

│ │ │ │ │ │ ├── ComparisonExpression

│ │ │ │ │ │ │ ├── ArithmeticExpression

│ │ │ │ │ │ │ ├── ComparisonOperator

│ │ │ │ │ │ │ └── ArithmeticExpression

│ │ │ │ │ │ └── [NotOperator] "(" LogicalExpression ")"

│ │ │ │ │ └── {AndOperator AndExpression}

│ │ │ │ └── {OrOperator AndExpression}

│ │ │ ├── Statement

│ │ │ └── ["else" Statement]

│ │ ├── GotoStatement

│ │ │ ├── "Goto"

│ │ │ └── Identifier

│ │ ├── LabelPoint

│ │ │ ├── Identifier

│ │ │ └── ":"

│ │ ├── ForToStatement

│ │ │ ├── "For"

│ │ │ ├── AssignStatement

│ │ │ ├── "To" | "DOWNTO"

│ │ │ ├── ArithmeticExpression

│ │ │ ├── "Do"

│ │ │ └── Statement

│ │ ├── WhileStatement

│ │ │ ├── "While"

│ │ │ ├── LogicalExpression

│ │ │ ├── {Statement}

│ │ │ └── "End" "While"

│ │ ├── RepeatUntilStatement

│ │ │ ├── "Repeat"

│ │ │ ├── {Statement}

│ │ │ └── "Until" "(" LogicalExpression ")"

│ │ └── CompoundStatement

│ │ ├── "StartBlok"

│ │ ├── {Statement}

│ │ └── "EndBlok"

└── "EndBlok"

*Рис. 3.2. Дерево граматичного розбору.*

* + 1. **Розробка алгоритму роботи синтаксичного і семантичного аналізатора.**

На етапі семантичного аналізу необхідно виконати ідентифікацію ідентифікаторів, що складається з двох основних етапів:

* **Оголошення ідентифікаторів**: обробка інформації про нові ідентифікатори.
* **Використання ідентифікаторів**: перевірка коректності використання ідентифікаторів у коді.

Коли лексичний аналізатор знаходить ідентифікатор, він створює структуру, яка містить атрибути лексеми, зокрема її ім'я, тип і лексичний клас. Ця інформація передається семантичному аналізатору.

Якщо аналізується **оголошення ідентифікатора**, основним завданням семантичного аналізатора є додавання даних про ідентифікатор до таблиці ідентифікаторів.

Під час **використання ідентифікатора**, інформація з таблиці ідентифікаторів перевіряється на відповідність. Для визначення типу ідентифікатора достатньо звернутися до відповідного поля у цій таблиці.

* + 1. **Опис програми реалізації синтаксичного та семантичного аналізатора.**

Одним з найбільш простих і найбільш популярних методів низхідного синтаксичного аналізу є метод рекурсивного спуску (recursive descent method).

Метод заснований на тому, що в склад синтаксичного аналізатора входить множина рекурсивних процедур граматичного розбору, по одній для кожного правила граматики.

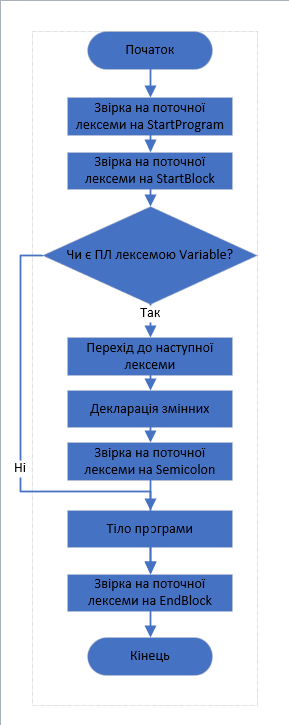


Рис. 3.3. Блок-сема алгоритму роботи синтаксичного аналізатора.

Семантичний аналіз і алгоритм ідентифікації ідентифікаторів

На етапі семантичного аналізу вирішується задача ідентифікації ідентифікаторів, що складається з двох частин:

* **Обробка оголошень ідентифікаторів:** додається інформація про ідентифікатори в таблицю ідентифікаторів.
* **Обробка використання ідентифікаторів:** перевіряється відповідність використання ідентифікаторів за інформацією з таблиці.

Лексичний аналізатор формує структуру, яка містить атрибути виділеної лексеми (ім’я, тип, лексичний клас) і передає її семантичному аналізатору. Семантична дія на цьому етапі залежить від того, чи йде мова про оголошення чи використання ідентифікатора.

Опис програми семантичного аналізатора

Семантичний аналізатор перевіряє логіку програми та коректність її структур. У коді реалізовано кілька ключових функцій:

Основні функції семантичного аналізатора

1. **getLastDataSectionLexemIndex**
   * Визначає індекс останньої лексеми в секції даних.
   * Використовує парсер recursiveDescentParserRuleWithDebug для перевірки секції "program\_\_\_\_part1".
   * Якщо лексема знайдена, повертається її індекс, інакше — код помилки.
2. **checkingInternalCollisionInDeclarations**
   * Перевіряє колізії в деклараціях ідентифікаторів та міток:
     + **Колізія ідентифікатор/ідентифікатор:** виявляється, якщо ідентифікатор задекларовано кілька разів у межах однієї області.
     + **Колізія мітка/мітка:** виявляється при дублюванні міток.
     + **Колізія ідентифікатор/мітка:** виявляється, якщо одне ім’я використовується і для змінної, і для мітки.
   * У разі недекларованого ідентифікатора чи мітки генерується помилка.
3. **checkingVariableInitialization**
   * Перевіряє, чи всі змінні ініціалізовані перед використанням.
   * Аналізує код після секції даних, перевіряючи операції присвоєння, виклики функцій чи введення даних.
4. **checkingCollisionInDeclarationsByKeyWords**
   * Виявляє збіги між іменами ідентифікаторів і зарезервованими словами:
     + Використовує регулярні вирази для аналізу.
     + У разі збігу повертає помилку (наприклад, COLLISION\_IK\_STATE).
5. **semantixAnalyze**
   * Координує всі перевірки:
     + Виявляє колізії в деклараціях.
     + Аналізує ініціалізацію змінних.
     + Перевіряє збіги з ключовими словами.
   * У разі помилок повертає відповідні коди стану.

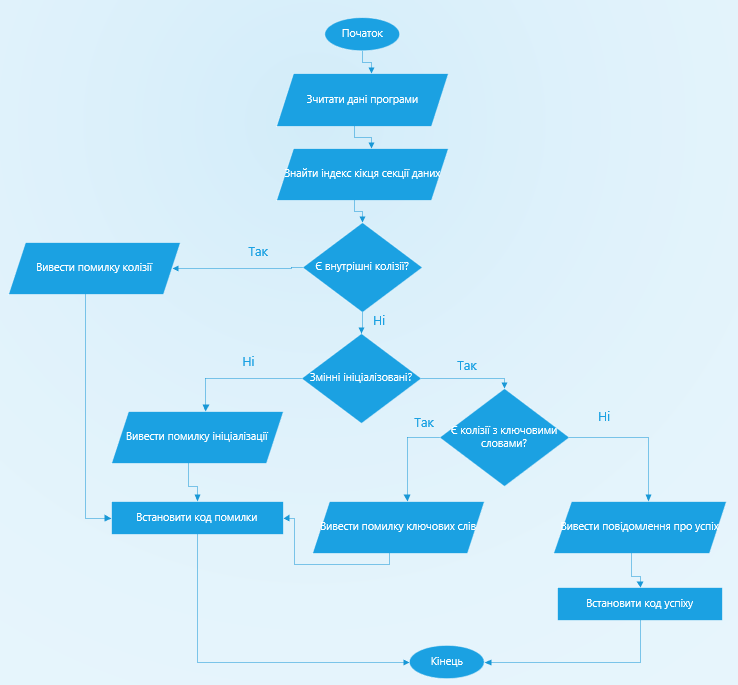
Основні аспекти реалізації

1. **Робота з лексемами та граматикою:**
   * Таблиця лексем (lexemInfoTable) та граматика (Grammar) використовуються для аналізу на основі попередніх етапів.
   * Типи лексем визначаються через поле tokenType.
2. **Перевірка колізій:**
   * Виявляються конфлікти, щоб уникнути неоднозначності в коді.
3. **Використання регулярних виразів:**
   * Перевірки з ключовими словами виконуються за допомогою std::regex.
4. **Обробка помилок:**
   * Помилки відображаються у консоль із деталями
5. **Коди стану:**
   * Функції повертають коди стану для сигналізації про успішність аналізу чи виявлені помилки.

Типовий процес роботи

1. **Виклик функції semantixAnalyze:**
   * Виконується перевірка декларацій і виявлення колізій.
   * Аналізується ініціалізація змінних.
   * Перевіряються збіги з ключовими словами.
2. **У разі помилок:**
   * Генерується відповідний код стану.
   * Виводиться детальна інформація про проблему.

Цей алгоритм забезпечує надійний семантичний аналіз програми, дозволяючи виявляти помилки в логіці та деклараціях ще до виконання коду.



*Рис. 3.4. Графічне представлення роботи семантичного аналізатора*

* 1. 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.**
    2. 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

/ \

Statement Оператор

/ \

statement Оператор

Оператор вводу:

Input

/ \

Id null

Оператор виводу:

Output

/ \

Id null

Також оператор виводу може мати за лівого нащадка різні арифметичні вирази, наприклад:

Output

/ \

Add null

/ \

Id num

Умовний оператор (IF() оператор;):

If

/ \

Умова оператор

Умовний оператор (IF() оператор1; else оператор2;):

If

/ \

Умова else

/ \

Оператор1 оператор2

Оператор безумовного переходу:

Goto

/ \

Id null

Оператор циклу for:

For

/ \

(to | downto) оператор

/ \

Оп. Прис. ариф. вир.

Оператор циклу while:

While

/ \

Умова statement

/ \

Statement оператор

/ \

Оператор оператор

Оператор циклу repeat:

Repeat

/ \

Statement умова

/ \

Оператор оператор

Оператор присвоєння:

==>

/ \

Id арифметичний вираз

Арифметичний вираз:

(+ або -)

/ \

Id id

Доданок:

(\*, DIV або MOD)  
/ \  
множник множник

Множник:

фактор  
/ \

id або number або (арифм. вираз) null

Складений оператор:

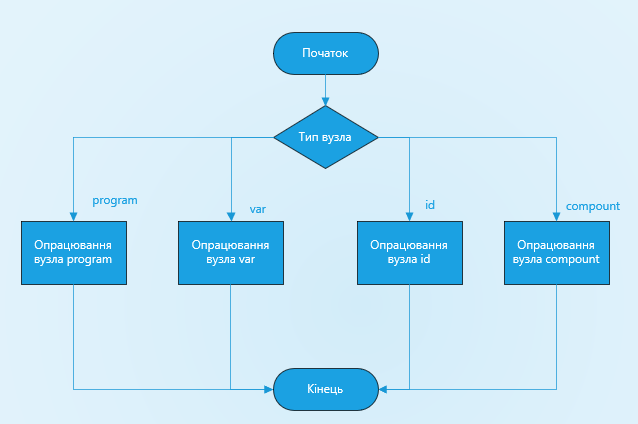
compount

/ \

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.

  
Рис. 3.5. Блок-сема алгоритму роботи генератора коду.

* + 1. **Опис програми реалізації генератора коду.**

Генерація коду в функції makeCode

Функція makeCode перетворює лексеми, що були зібрані на попередньому етапі, у машинний код або інший низькорівневий формат, поступово створюючи різні частини програми. Ось детальніше пояснення її роботи:

1. **Ініціалізація:**

Спочатку функція викликає кілька допоміжних функцій, щоб сформувати основні частини коду програми:

currBytePtr = makeTitle(lastLexemInfoInTable, currBytePtr);

currBytePtr = makeDependenciesDeclaration(lastLexemInfoInTable, currBytePtr);

currBytePtr = makeDataSection(lastLexemInfoInTable, currBytePtr);

currBytePtr = makeBeginProgramCode(lastLexemInfoInTable, currBytePtr);

* **makeTitle**: Формує заголовок програми, в якому можуть бути визначення архітектури або параметрів.
* **makeDependenciesDeclaration**: Додає оголошення для необхідних залежностей, таких як бібліотеки або модулі.
* **makeDataSection**: Формує секцію даних, яка містить глобальні змінні та константи.
* **makeBeginProgramCode**: Початкова частина коду програми, зокрема налаштування стеку чи регістрів.

2. **Ініціалізація стеку:**

Після створення базових частин програми функція переходить до ініціалізації стеку та зберігання його поточного стану:

lexemInfoTransformationTempStackSize = 0;

currBytePtr = makeInitCode(lastLexemInfoInTable, currBytePtr);

currBytePtr = initMake(lastLexemInfoInTable, currBytePtr);

currBytePtr = makeSaveHWStack(lastLexemInfoInTable, currBytePtr);

* **makeInitCode**: Створює код для ініціалізації змінних.
* **initMake**: Додає додаткові налаштування для ініціалізації, включаючи стек.
* **makeSaveHWStack**: Зберігає поточний стан апаратного стеку.

3. **Обробка лексем у циклі:**

Основний процес генерації коду відбувається в циклі, де кожна лексема аналізується і обробляється в залежності від її типу:

for (struct LexemInfo\* lastLexemInfoInTable\_;

lastLexemInfoInTable\_ = \*lastLexemInfoInTable,

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

Цей цикл продовжується, поки не буде досягнута лексема з порожнім рядком.

4. **Генерація коду для конструкцій:**

В залежності від лексеми викликаються спеціалізовані функції для генерації конкретних частин коду. Наприклад:

* **Умовні оператори:**
* IF\_THEN\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
* ELSE\_CODER(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
  + **IF\_THEN\_CODER**: Генерує код для умовного оператора if.
  + **ELSE\_CODER**: Генерує код для блоку 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 код для циклу while.
  + **REPEAT\_UNTIL\_CODER**: Обробляє конструкцію циклу repeat until.

1. 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.

* 1. Опис інтерфейсу та інструкції користувачу.

Розроблений транслятор має простий консольний інтерфейс.

При запуску програми обирається базовий файл file1.cwl:

* 1. Виявлення лексичних і синтаксичних помилок.
  2. Перевірка роботи транслятора за допомогою тестових задач.

Висновки

Список літературних джерел

Додатки

А. Таблиці лексем для тестових прикладів

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

В. Абстрактне синтаксичне дерево для тестових прикладів (за наявності).

Г. Документований текст програмних модулів (лістинги)