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»

Варіант 18

**Виконала:**

ст. групи КІ-308

Продан К.О.

**Перевірив:**

старший викладач кафедри ЕОМ

Козак Н.Б.

Львів – 2024

**ЗАВДАННЯ НА КУРСОВЕ ПРОЄКТУВАННЯ**

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 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.

В моєму випадку це p18

**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.

3. За допомогою оператора вводу можна зчитати з клавіатури значення змінної; за допомогою оператора виводу можна вивести на екран значення змінної, виразу чи цілої константи.

4. В кожному завданні обов’язковим є оператор присвоєння за допомого 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.

5. В кожному завданні обов’язковим є оператор типу “блок” (складений оператор), його вигляд має бути таким, як і блок тіла програми

6. 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.

7. Оператори можуть бути довільної вкладеності і в будь-якій послідовності.

8. Для перевірки роботи розробленого транслятора, необхідно написати три тестові програми на вхідній мові програмування.

**Деталізований опис власної мови програмування:**

Блок тіла програми: **StartProgram;**

**Int32 … ;**

**EndBlok;**

Оператори вводу-виводу: **Scan, Print**

Оператор присвоєння: **==>**

Умовний оператор: **If-Else**

Оператор безумовного переходу: **Goto**

Оператори циклу: **For-To-Do; For-DownTo-Do; While; Repeat-Until**

Регістр ідентифікаторів: **\_LowUp6**

Операції:

- арифметичні: **++ , --, \*\*, Div**

- порівняння: **Ls, Gr, Eq, Neq**

- логічні: **Not, And, Or**

Тип даних: **Int32**

Коментар: **{ ... }**

Програма на вхідній мові програмування має починатись з ключового слова StartProgram та назвою програми, далі має йти розділ опису змінних Variable. Між розділом Variable і ключовим словом EndBlok розміщуються оператори програми. Операторів є 5: оператор вводу даних Scan, оператор виводу даних Print, оператор присвоєння ==>, умовний оператор If-Goto та оператор циклу For-To-Next. Також програма може містити складені оператори які починаються ключовим словом StartBlok та закінчуються ключовим словом EndBlok. Кожен оператор має завершуватись символом крапка з комою ;. Оператор присвоєння дозволяє присвоїти деякій змінній значення арифметичного виразу. Допустимі арифметичні операції ++, --, \*\*, Div, Mod. Операндами можуть бути змінні, цілі додатні константи і інші вирази, взяті в дужки. В умовному операторі використовуються логічні вирази, допустимі такі операції порівняння Eq, Neq, Ls, Gr і такі логічні операції Not, And, Or. Ідентифікатори (імена змінних) починаються лише символом нижнього підкреслення «\_» і латинською літерою, далі йдуть букви або цифри, довжиною до 6-ти символів. Тип даних лише один – Int32, при оголошення декількох змінних вони записуються через кому, вкінці опису змінних ставиться символ крапка з комою ;. Коментарі починаються з символів "{ " та завершуються ними ж – «{ ... }» .

Приклади оголошення змінних:

Int32 \_a;

Int32 \_a, \_b, \_c;

Int32 \_a, \_Max;

Приклади ідентифікаторів:

\_a, \_a1, \_a20, \_Min, \_Max

Приклади арифметичних виразів:

0 -- 75

\_y ++ 68

\_a ++ \_b \*\* \_c ++ 76

\_c \*\* (\_a ++ \_b) -- 50 Div \_b

Приклади логічних виразів:

\_a Ls \_b

\_a Ls \_b And \_a Ls \_c

Not (\_a Gr \_c)

\_a Neq \_b

\_Min Gr 0 -- 100

**АНОТАЦІЯ**

У даному курсовому проєкті 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**

ВСТУП8

1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ9

2. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ13

**2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура13**

2.2. Опис термінальних символів та ключових слів16

**3. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ18**

**3.1. Вибір технології програмування18**

3.2. Проєктування таблиць транслятора та вибір структур даних18

**3.3. Розробка лексичного аналізатора20**

**3.3.1. Розробка алгоритму роботи лексичного аналізатора21**

3.3.2. Опис програми реалізації лексичного аналізатора24

**3.4. Розробка синтаксичного та семантичного аналізатора25**

**3.4.1. Розробка алгоритму роботи синтаксичного і семантичного аналізатора25**

**3.4.2. Опис програми реалізації синтаксичного та семантичного аналізатора32**

**3.5. Розробка генератора коду33**

**3.5.1. Розробка алгоритму роботи генератора коду33**

**3.5.2. Опис програми реалізації генератора коду34**

**4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА36**

**4.1. Опис інтерфейсу та інструкції користувачу36**

**4.2. Виявлення лексичних і синтаксичних помилок37**

**4.3. Перевірка роботи транслятора за допомогою тестових задач39**

**ВИСНОВКИ**

**СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ**

**ДОДАТКИ**

**ВСТУП**

Термін "транслятор" визначає програму, яка виконує переклад (трансляцію) початкової програми, написаної на вхідній мові, у еквівалентну їй об'єктну програму. У випадку, коли мова високого рівня є вхідною, а мова асемблера або машинна – вихідною, такий транслятор отримує назву компілятора.

Транслятори можуть бути розділені на два основних типи: компілятори та інтерпретатори. Процес компіляції включає дві основні фази: аналіз та синтез. Під час аналізу вхідну програму розбивають на окремі елементи (лексеми), перевіряють її відповідність граматичним правилам і створюють проміжне представлення програми. На етапі синтезу з проміжного представлення формується програма в машинних кодах, яку називають об'єктною програмою. Останню можна виконати на комп'ютері без додаткової трансляції.

У відміну від компіляторів, інтерпретатор не створює нову програму; він лише виконує – інтерпретує – кожну інструкцію вхідної мови програмування. Подібно компілятору, інтерпретатор аналізує вхідну програму, створює проміжне представлення, але не формує об'єктну програму, а негайно виконує команди, передбачені вхідною програмою.

Компілятор виконує переклад програми з однієї мови програмування в іншу. На вхід компілятора надходить ланцюг символів, який представляє вхідну програму на певній мові програмування. На виході компілятора (об'єктна програма) також представляє собою ланцюг символів, що вже відповідає іншій мові програмування, наприклад, машинній мові конкретного комп'ютера. При цьому сам компілятор може бути написаний на третій мові.

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

Термін "транслятор" визначає обслуговуючу програму, що проводить трансляцію вихідної програми, представленої на вхідній мові програмування, у робочу програму, яка відображена на об'єктній мові. Наведене визначення застосовне до різноманітних транслюють програм. Однак кожна з таких програм може виявляти свої особливості в організації процесу трансляції. В сучасному контексті транслятори поділяються на три основні групи: асемблери, компілятори та інтерпретатори.

Асемблер - це системна обслуговуюча програма, яка перетворює символічні конструкції в команди машинної мови. Типовою особливістю асемблерів є дослівна трансляція однієї символічної команди в одну машинну.

Компілятор - обслуговуюча програма, яка виконує трансляцію програми, написаної мовою оригіналу програмування, в машинну мову. Схоже до асемблера, компілятор виконує перетворення програми з однієї мови в іншу, найчастіше - у мову конкретного комп'ютера.

Інтерпретатор - це програма чи пристрій, що виконує пооператорну трансляцію та виконання вихідної програми. Відмінно від компілятора, інтерпретатор не створює на виході програму на машинній мові. Розпізнавши команду вихідної мови, він негайно її виконує, забезпечуючи більшу гнучкість у процесі розробки та налагодження програм.

Процес трансляції включає фази лексичного аналізу, синтаксичного та семантичного аналізу, оптимізації коду та генерації коду. Лексичний аналіз розбиває вхідну програму на лексеми, що представляють слова відповідно до визначень мови. Синтаксичний аналіз визначає структуру програми, створюючи синтаксичне дерево. Семантичний аналіз виявляє залежності між частинами програми, недосяжні контекстно-вільним синтаксисом. Оптимізація коду та генерація коду спрямовані на оптимізацію та створення машинно-залежного коду відповідно.

Зазначені фази можуть об'єднуватися або відсутні у трансляторах в залежності від їхньої реалізації. Наприклад, у простих однопрохідних трансляторах може відсутні фаза генерації проміжного представлення та оптимізації, а інші фази можуть об'єднуватися.

Під час процесу виділення лексем лексичний аналізатор може виконувати дві основні функції: автоматично побудову таблиць об'єктів (таких як ідентифікатори, рядки, числа і т. д.) і видачу значень для кожної лексеми при кожному новому зверненні до нього. У цьому контексті таблиці об'єктів формуються в подальших етапах, наприклад, під час синтаксичного аналізу.

На етапі лексичного аналізу виявляються деякі прості помилки, такі як неприпустимі символи або невірний формат чисел та ідентифікаторів.

Основним завданням синтаксичного аналізу є розбір структури програми. Зазвичай під структурою розуміється дерево, яке відповідає розбору в контекстно-вільній граматиці мови програмування. У сучасній практиці найчастіше використовуються методи аналізу, такі як LL (1) або LR (1) та їхні варіанти (рекурсивний спуск для LL (1) або LR (1), LR (0), SLR (1), LALR (1) та інші для LR (1)). Рекурсивний спуск застосовується частіше при ручному програмуванні синтаксичного аналізатора, тоді як LR (1) використовується при автоматичній генерації синтаксичних аналізаторів.

Результатом синтаксичного аналізу є синтаксичне дерево з посиланнями на таблиці об'єктів. Під час синтаксичного аналізу також виявляються помилки, пов'язані зі структурою програми.

На етапі контекстного аналізу виявляються взаємозалежності між різними частинами програми, які не можуть бути адекватно описані за допомогою контекстно-вільної граматики. Ці взаємозалежності, зокрема, включають аналіз типів об'єктів, областей видимості, відповідності параметрів, міток та інших аспектів "опис-використання". У ході контекстного аналізу таблиці об'єктів доповнюються інформацією, пов'язаною з описами (властивостями) об'єктів.

В основі контекстного аналізу лежить апарат атрибутних граматик. Результатом цього аналізу є створення атрибутованого дерева програми, де інформація про об'єкти може бути розсіяна в самому дереві чи сконцентрована в окремих таблицях об'єктів. Під час контекстного аналізу також можуть бути виявлені помилки, пов'язані з неправильним використанням об'єктів.

Після завершення контекстного аналізу програма може бути перетворена во внутрішнє представлення. Це здійснюється з метою оптимізації та/або для полегшення генерації коду. Крім того, перетворення програми у внутрішнє представлення може бути використано для створення переносимого компілятора. У цьому випадку, тільки остання фаза (генерація коду) є залежною від конкретної архітектури. В якості внутрішнього представлення може використовуватися префіксний або постфіксний запис, орієнтований граф, трійки, четвірки та інші формати.

Фаза оптимізації транслятора може включати декілька етапів, які спрямовані на покращення якості та ефективності згенерованого коду. Ці оптимізації часто розподіляються за двома головними критеріями: машинно-залежні та машинно-незалежні, а також локальні та глобальні.

Машинно-залежні оптимізації, як правило, проводяться на етапі генерації коду, і вони орієнтовані на конкретну архітектуру машини. Ці оптимізації можуть включати розподіл регістрів, вибір довгих або коротких переходів та оптимізацію вартості команд для конкретних послідовностей команд.

Глобальна оптимізація спрямована на поліпшення ефективності всієї програми і базується на глобальному потоковому аналізі, який виконується на графі програми. Цей аналіз враховує властивості програми, такі як міжпроцедурний аналіз, міжмодульний аналіз та аналіз галузей життя змінних.

Фінальна фаза трансляції - генерація коду, результатом якої є або асемблерний модуль, або об'єктний (або завантажувальний) модуль. На цьому етапі можуть застосовуватися деякі локальні оптимізації для полегшення генерації вартісного та ефективного коду.

Важливо відзначити, що фази транслятора можуть бути відсутніми або об'єднаними в залежності від конкретної реалізації. В простіших випадках, таких як у випадку однопроходових трансляторів, може відсутній окремий етап генерації проміжного представлення та оптимізації, а інші фази можуть бути об'єднані в одну, при цьому не створюється явно побудованого синтаксичного дерева.

**2. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ**

**2.1. 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:**

**<програма>** = ‘StartProgram’ ‘;’ ‘Variable’ <оголошення змінних> ‘;’ <тіло програми> ‘EndBlok’ ';'

**<оголошення змінних>** = [<тип даних> <список змінних>]

**<тип даних>** = ‘Int32’

**<список змінних>** = <ідентифікатор> ['==>' <арифметичний вираз>] { ‘,’ <ідентифікатор> ['==>' <арифметичний вираз>]}

**<ідентифікатор>** = '\_' <буква> { <буква або цифра> }

**<буква або цифра>** = <буква> | <цифра>

**<буква>** = <велика буква> | <мала буква>

**<велика буква>** = ‘A’ | ‘B’ | ‘C’ | ‘D’ | ... | ‘Z’

**<мала буква>** = ‘a’ | ‘b’ | ‘c’ | ‘d’ | ... | ‘z’

**<цифра>** = ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ | ‘0’

**<арифметичний вираз>** = <доданок> { ‘++’ <доданок> | ‘--’ <доданок> }

**<доданок>** = <множник> { ‘\*\*’ <доданок> | ‘Div’ <доданок> | ‘Mod’ <доданок>}

**<множник>** = <ідентифікатор> | <число> | ‘(’ <арифметичний вираз> ‘)’

**<число>** = ['-' | '+'] <цифра> { <цифра> }

**<тіло програми>** = { <оператор> }

**<оператор>** = <мітка> | <присвоєння> | <ввід> | <вивід> | <умовний оператор> | <безумовний перехід> | <оператор циклу forTo> | <оператор циклу forDownTo> | <оператор циклу while> | <оператор циклу repeat> | <складений оператор>

**<мітка>** = <ідентифікатор> ':'

**<присвоєння>** = <ідентифікатор> ‘==>’ <арифметичний вираз> ';'

**<ввід>** = ‘Scan’ <ідентифікатор> ';'

**<вивід>** = ‘Print’ <арифметичний вираз> ';'

**<умовний оператор>** = 'If’ <логічний вираз> ‘Goto' <мітка>

**<логічний вираз>** = <вираз І> { ‘Or’ <вираз І> }

**<вираз І>** = <порівняння> { ‘And’ <вираз І> }

**<порівняння>** = <операція порівняння> | ‘Not’ ‘(’ <логічний вираз> ‘)’ | ‘(’ <логічний вираз> ‘)’

**<операція порівняння>** = <арифметичний вираз> <менше-більше> <арифметичний вираз>

**<менше-більше>** = ‘Eq’ | ‘Neq’ | ‘Ls’ | ‘Gr’

**<оператор циклу forTo>** ==> 'For' <ідентифікатор> '==>' <арифметичний вираз> 'To' <арифметичний вираз> 'Do' <оператор>

**<оператор циклу forDownTo>** ==> 'For' <ідентифікатор> '==>' <арифметичний вираз> 'DownTo' <арифметичний вираз> 'Do' <оператор>

**<оператор циклу while>** ==> 'While' '(' <логічний вираз> ')' 'Do' <оператор>

**<оператор циклу repeat>** ==> 'Repeat' <оператор> 'Until' '(' <логічний вираз> ')' ';'

**<безумовний перехід>** = 'Goto' <ідентифікатор> ';'

**<складений оператор>** = ‘StartBlok’ <тіло програми> ‘EndBlok’ ';'

**2.2. Опис термінальних символів та ключових слів**

Визначаємо термінальні символи і ключові слова:

• **StartProgram** – початок програми

• **Variable** – оголошення змінних

• **StartBlok** – початок складеного оператора

• **EndBlok** – кінець програми

• **Int32** – тип даних

• **Scan** – оператор вводу

• **Print** – оператор виводу

• **If,Else** – умовний оператор

• **Goto** – безумовний перехід

• **For,To,DownTo,Do,While,Repeat,Until** – оператор циклу

• **==>** – оператор присвоєння

• **++** – додавання

• **--** – віднімання

• **\*\*** – множення

• **Div** – ділення

• **Mod** – остача від ділення

• **Eq** – рівність

• **Neq** – нерівність

• **Ls** – більше

• **Gr** – менше

• **Not** – логічне НЕ

• **And** – логічне І

• **Or** – логічне АБО

• **;** – кінець оператора

• **,** – розділювач змінних

• **(** – відкрита дужка

• **)** – закрита дужка

• **{ ... }** – початок та кінець коментаря

• **A … Z** – великі латинські букви

• **a … z** – маленькі латинські букви

• **0 … 9** – цифри

• символи табуляції, переходу на новий рядок, пробіл

**3. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ**

**3.1. 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.

**3.2. Проєктування таблиць транслятора та вибір структур даних**

Використання таблиць значно полегшує створення трансляторів, а тому

створимо необхідні структури даних для зберігання інформації про лексеми

// перерахування, яке описує всі можливі типи лексем

enum TypeOfTokens

{

StartProgram,// StartProgram

StartBody, // StartBlok

Variable, // Variable

Type, // Int32

End, // EndBlok

Input, // Scan

Output, // Print

If, // If

Else, // Else

Goto, // Goto

For, // For

To, // To

DownTo, // DownTo

Do, // Do

While, // While

Repeat, // Repeat

Until, // Until

Label, // Label

Identifier, // Identifier

Number, // Number

Assign, // ==>

Add, // ++

Sub, // --

Mul, // \*\*

Div, // Div

Mod, // Mod

Equality, // Eq

NotEquality, // Neq

Greate, // Ls

Less, // Gr

Not, // Not

And, // And

Or, // Or

LBraket, // (

RBraket, // )

Semicolon, // ;

Comma, // ,

Unknown

};

У програмі будемо зберігати таблицю лексем і таблицю ідентифікаторів, а

також кількість лексем і ідентифікаторів:

// таблиця лексем

Token\* TokenTable;

// кількість лексем

unsigned int TokensNum;

// таблиця ідентифікаторів

Id\* IdTable;

// кількість ідентифікаторів

unsigned int IdNum;

**3.3. 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 (0 - StartProgram, 1 - StartBlok, 2 - Variable, 3 - EndBlok, 4 - Scan, 5 - Print, 6 - Int32, 7 - If, 8 - Goto, 9 - For, 10 - To, 11 - Next, 12 - Div, 13 - Mod, 14 - Eq, 15 - Neq, 16 - Ls, 17 - Gr, 18 - Not, 19 - And, 20 - Or)

• Ідентифікатори (21 - починаються лише символом нижнього підкреслення «\_» і латинською літерою, далі йдуть букви або цифри, довжиною до 6-ти символів)

• Числові константи (22 - ціле число без знаку)

• Оператор присвоєння (23 – ==>)

• Знаки операції (24 - ++, 25 - --, 26 - \*\*)

• Дужки (27 – (, 28 – ))

• Розділювачі (29 –;, 30 –,)

• Невідома лексема (31 – символи і ланцюжки символів, які не підпадають під вищеописані правила).

**3.3.1. Розробка алгоритму роботи лексичного аналізатора.**

Розробимо алгоритм роботи лексичного аналізатора на основі скінченного автомату. Лексичний аналізатор працює за принципом скінченного автомату з такими станами:

**• Start** – початок виділення чергової лексеми;

**• Finish** – кінець виділення чергової лексеми;

**• EndOfFile** – кінець файлу, завершення розпізнавання лексем;

• **KeyWords** – перший символ буква, розпізнавання слів (ключові слова);

• **Identifiers** – перший символ буква, розпізнавання слів (ідентифікатори);

• **Digit** – перший символ цифра, розпізнавання числових констант;

**• Separators** – видалення пробілів, символів табуляції і переходу на новий рядок;

**• Comment** – видалення тексту коментаря;

**• Another** – опрацювання інших символів.

У стані KeyWords читаємо по одному символи з файлу і виділяємо ланцюжок символів, який починається з букви, а далі можуть слідувати букви або цифри. Кінець ланцюжка – якщо прочитаний символ відмінний від букви чи цифри. Виділений ланцюжок порівнюємо з ключовими словами, якщо співпадінь немає то він є невизначеним. Переходимо до стану Finish.

У стані Identifiers читаємо по одному символи з файлу і виділяємо ланцюжок символів, який починається з символу підкреслення, далі може бути велика або мала буква, а далі букви або цифри. Кінець ланцюжка – якщо прочитаний символ відмінний від букви чи цифри. Якщо довжина ланцюжка не більше 6-х символів вважаємо його ідентифікатором, інакше це невизначена лексема.  
Переходимо до стану Finish.

У стані Digit читаємо по одному символи з файлу і виділяємо ланцюжок символів, який складається лише з цифр, вважаємо цей ланцюжок числовою константою. Кінець ланцюжка – якщо прочитаний символ відмінний від цифри. Переходимо до стану Finish.

У стані Comment читаємо символи, поки не зустрінеться кінець файлу або символ закриття коментаря(‘**}**’). Якщо зустрівся символ кінця файлу то повертаємось на початок та позначаємо символ ‘**{**’, як невідомий. Якщо зустрівся символ закриття коментаря(‘**}**’) то переходимо до виділення нової лексеми – до стану Start.

У стані Separators читаємо наступний символ і переходимо до виділення нової лексеми – до стану Start. Тобто пропускаємо усі пробіли, символи табуляції і переходу на новий рядок.

У стані Another порівнюємо поточний прочитаний символ з символами, що позначають знаки операцій, розділювачі і круглі дужки і визначаємо одну з лексем. Є три лексеми, які вимагають ще читання наступного символу з файлу – це операція додавання “++”, операція віднімання “--” та операція множення “\*\*”. А також одна лексема яка вимагає третього читання наступного символу з файлу – це оператор присвоєння “==>”. Якщо співпадіння не виявлено, то поточний символ – невідома лексема, читаємо наступний символ і переходимо до стану Finish.

У стані Finish записуємо поточну лексему у таблицю лексем і переходимо до виділення нової лексеми, до стану Start.

У стані EndOfFile завершуємо обробку вхідного файлу, усі символи з файлу прочитані, усі лексеми записані у таблицю лексем.

Алгоритм роботи лексичного аналізатора можна зобразити у вигляді граф-схеми.

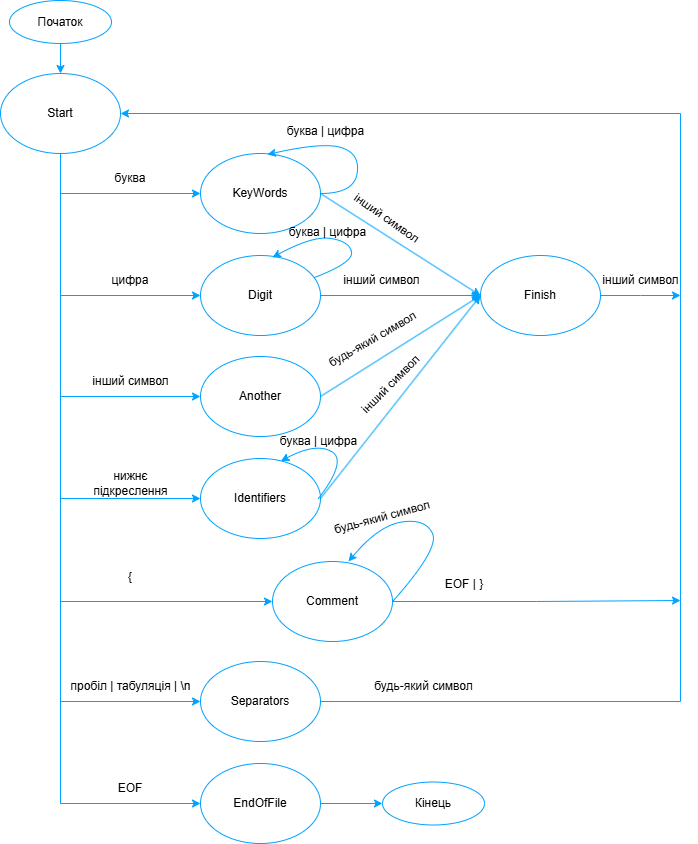


Рис. 3.1 Граф-схема алгоритму роботи лексичного аналізатора.

**3.3.2. Опис програми реалізації лексичного аналізатора.**

В даному курсовому проєкті реалізовано прямий лексичний аналізатор, який виділяє з вхідного тексту програми окремі лексеми і на основі цього формує таблицю лексем.

Створимо структуру даних для зберігання стану аналізатора:

// перерахування, яке описує стани лексичного аналізатора

enum States

{

Start, // початок виділення чергової лексеми

Finish, // кінець виділення чергової лексеми

KeyWords, // опрацювання слів (ключові слова)

Identifiers,// опрацювання слів (ідентифікатори)

Digit, // опрацювання цифри

Separators, // видалення пробілів, символів табуляції і переходу на новий рядок

Another, // опрацювання інших символів

EndOfFile, // кінець файлу

Comment // видалення коментаря

};

Напишемо функцію, яка реалізує лексичний аналіз:  
// функція отримує лексеми з вхідного файлу F і записує їх у таблицю лексем TokenTable

// результат функції - кількість лексем

unsigned int GetTokens(FILE\* F, Token TokenTable[]);

І функції, які друкують список лексем:

// функція друкує таблицю лексем на екран

void PrintTokens(Token TokenTable[], unsigned int TokensNum);

// функція друкує таблицю лексем у файл

void PrintTokensToFile(char\* FileName, Token TokenTable[], unsigned int TokensNum);

**3.4. 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, але в більшості випадків для заданої мови програмування ми можемо побудувати таку граматику, що дозволить сконструювати і більш швидкий аналізатор.

Аналізатори реальних мов зазвичай мають лінійну складність; це досягається за рахунок перегляду вхідної програми зліва направо із загляданням уперед на один термінальний символ (лексичний клас).

Вхід синтаксичного аналізатора – це послідовність лексем і таблиці представлень, які є виходом лексичного аналізатора.

На виході синтаксичного аналізатора отримуємо дерево граматичного розбору і таблиці ідентифікаторів та типів, які є входом для наступного перегляду компілятора (наприклад, це може бути перегляд, який здійснює контроль типів – семантичний аналіз)

**3.4.1. Розробка алгоритму роботи синтаксичного і 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.

Визначимо назви процедур, що відповідають нетерміналам граматики таким чином :

<програма> = ‘StartProgram’ ‘;’ ‘Variable’ <оголошення змінних> ‘; ’ <тіло програми> ‘EndBlok’ ';'

void program();

<оголошення змінних> = [<тип даних> <список змінних>]

void variable\_declaration();

<список змінних> = <ідентифікатор> ['==>' <арифметичний вираз> ] { ‘,’ <ідентифікатор>['==>' <арифметичний вираз >]}

void variable\_list();

<тіло програми> = { <оператор> }

void program\_body();

<оператор> = <присвоєння> | <ввід> | <вивід> | <умовний оператор> | <умовний перехід> | <мітка> | <оператор циклу> | <складений оператор>

void statement();

<мітка> = <ідентифікатор> ':'

void label();

<присвоєння> = <ідентифікатор> '==>' <арифметичний вираз> ';'

void assignment();

<арифметичний вираз> = <доданок> { ‘++’ <доданок> | ‘--’ <доданок> }

void arithmetic\_expression();

<доданок> = <множник>{ ‘\*\*’ <доданок> | ‘Div’ <доданок> | ‘Mod’ <доданок> }

void term();

<множник> = <ідентифікатор> | <число> | ‘(’ <арифметичний вираз> ‘)’

void factor();

<ввід> = 'Scan' <ідентифікатор> ';'

void input();

<вивід> = ‘Print’ <арифметичний вираз> ';'

void output();

<умовний оператор> = 'If’ <логічний вираз> ‘Goto' <мітка>

void conditional();

<логічний вираз> = <вираз І> { ‘Or’ <вираз І> }

void logical\_expression();

<вираз І> = <порівняння> { ‘And’ <вираз І> }

void and\_expression();

<порівняння> = <операція порівняння> | ‘Not’ ‘(’ <логічний вираз> ‘)’ | ‘(’ <логічний вираз> ‘)’

<операція порівняння> = <арифметичний вираз> <менше - більше> <арифметичний вираз>

<менше - більше> = ‘Eq’ | ‘Neq’ | ‘Ls’ | ‘Gr’

void comparison();

<оператор циклу> ==> <оператор циклу forTo> | <оператор циклу forDownTo>

void for\_statement();  
<оператор циклу forTo> ==> 'For' <ідентифікатор> '==>' <арифметичний вираз> 'To' <арифметичний вираз> 'Do' <оператор>

void forTo\_statement();

<оператор циклу forDownTo> ==> 'For' <ідентифікатор> '==>' <арифметичний вираз> 'DownTo' <арифметичний вираз> 'Do' <оператор>

void forDownTo\_statement();

<оператор циклу while> ==> 'While' '(' <логічний вираз> ')' 'Do' <оператор>

void while\_statement();

<оператор циклу repeat> ==> 'Repeat' <оператор> 'Until' '(' <логічний вираз> ')' ';'

void repeat\_statement();

<безумовний перехід> = 'Goto' <ідентифікатор> ';'

void goto\_statement();

<складений оператор> = ‘StartBlok’ <тіло програми> ‘EndBlok’ ';'

void compound\_statement();

Блок-схема алгоритму роботи синтаксичного аналізатора виглядатиме наступним чином:

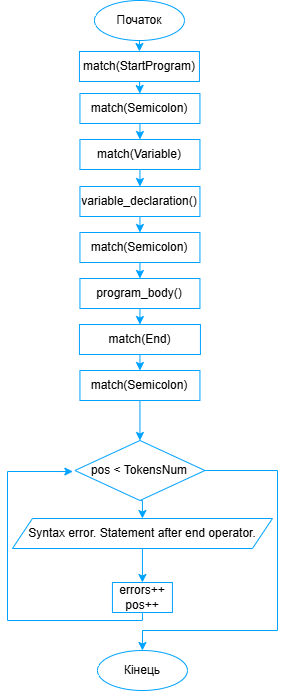


Рис. 3.3. Блок-сема алгоритму роботи синтаксичного аналізатора.

Синтаксичний аналізатор буде читати лексеми з таблиці лексем і аналізувати їх. Нам знадобиться допоміжна функція

void match(TypeOfTokens expectedType)

яка перевіряє, чи поточна лексема збігається з очікуваною.

Блок-схема алгоритму функції variable\_declaration(), яка перевіряє чи

правильно описані змінні виглядає наступним чином:

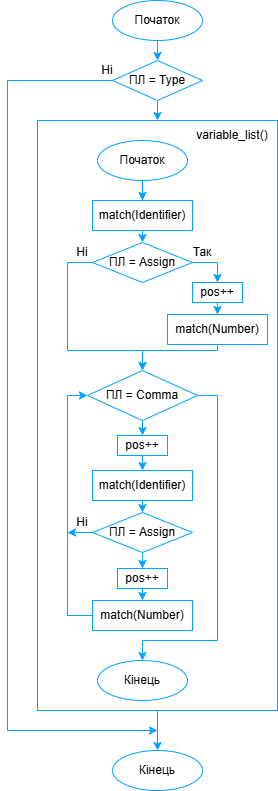


Рис. 3.4. Блок-сема алгоритму роботи функції variable\_declaration().

Блок-схема алгоритму функції program\_body(), яка перевіряє чи правильно написані оператори вхідної мови програмування зображена на рисунку 3.5.

У наведених блок-схемах використано позначення ПЛ, яке позначає поточну лексему. У процесі перегляду таблиці лексем, після аналізу поточної лексеми, необхідно переміщатися на наступну лексему.

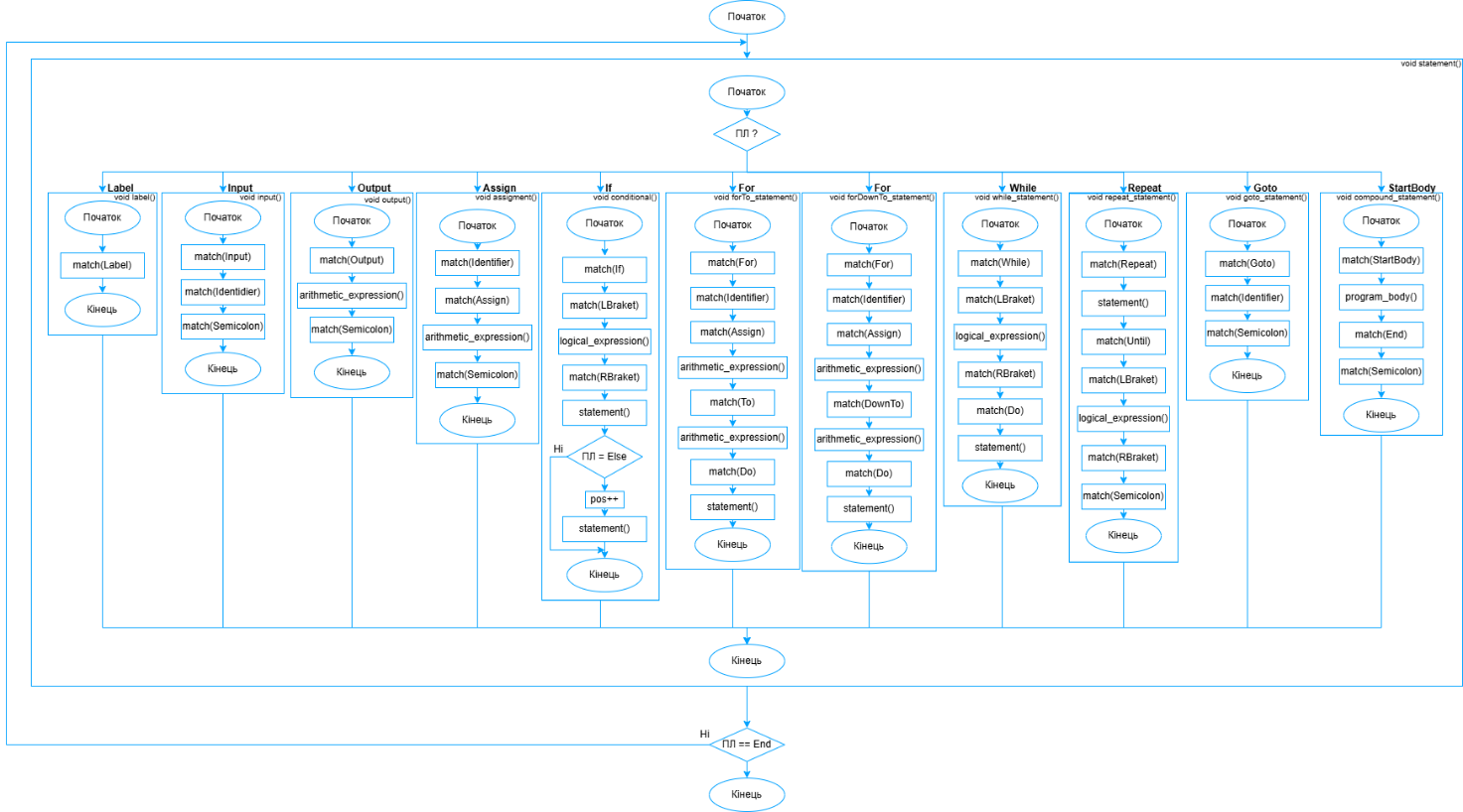


Рис. 3.5. Блок-сема алгоритму роботи функції program\_body().

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.

Опрацювання оголошення ідентифікатора. 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.

**3.4.2. Опис програми реалізації синтаксичного та семантичного аналізатора**

Структура синтаксичного аналізатора буде такою:

extern unsigned int errors;

extern char\* ErrorsFileName;

extern FILE\* ErrorsFile;

// Вхідна таблиця лексем

extern Token\* TokenTable;

// кількість лексем

extern unsigned int TokensNum;

void parser()

{

if ((fopen\_s(&ErrorsFile, ErrorsFileName, "at")) != 0)

{

printf("Error: Can not create file: %s\n", ErrorsFileName);

return;

}

program();

fclose(ErrorsFile);

}

Синтаксичний аналізатор працює за методом рекурсивного спуску, а отже функція parser() викликає функцію program(), яка в свою чергу викликає інші функції.

Семантичний аналіз у нашому випадку буде реалізований у функції, яка опрацьовує оголошення і використання ідентифікаторів:

// функція записує оголошені ідентифікатори в таблицю ідентифікаторів IdTable

// повертає кількість ідентифікаторів

// перевіряє чи усі використані ідентифікатори оголошені

unsigned int IdIdentification(Id IdTable[], Token TokenTable[], unsigned int tokenCount)

**3.5. Розробка генератора коду**

Генерація коду є однією з основних фаз у процесі компіляції програмного коду. Ця фаза відповідає за перетворення проміжного представлення коду (такого як абстрактне синтаксичне дерево або проміжний код) у машинний код або код низького рівня, який може бути безпосередньо виконаний апаратним забезпеченням або віртуальною машиною.

Генерацію коду можна розглядати як переклад програми на одній мові програмування у програму на іншій мові програмування. Для того, щоб замінити синтаксичну конструкцію однієї мови на синтаксичну конструкцію іншою мовою, необхідно встановити семантичні відповідності.

**3.5.1. Розробка блок-схеми алгоритму**

Розглянемо мову програмування, описану вище. Вважаємо, що програма синтаксично і семантично правильна, а отже її можна перекласти на іншу мову програмування, наприклад на мову С.

Для перекладу програми на нашій мові програмування у програму на мові С можемо скористатись методом рекурсивного спуску, де певна функція опрацьовує певне правило за встановленим шаблоном.

Блок-схема алгоритму роботи генератора коду виглядатиме наступним чином:

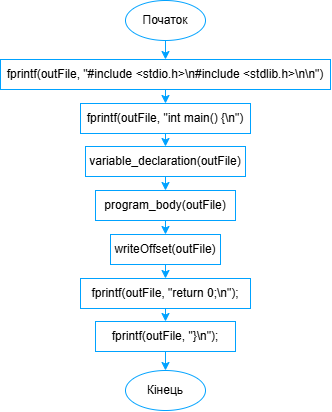


Рис. 3.6. Блок-сема алгоритму роботи генератора коду.

Генератор коду буде читати лексеми з таблиці лексем генерувати код у файл. Для кращої читабельності згенерованого коду скористуюсь функцією

void writeOffset(FILE\* outFile) {

for (int i = 0; i < offset; i++)

fprintf(outFile, "\t");

}

яка буде робити відступи від початку рядка.

**3.5.2. Опис програми реалізації генератора коду**

Структура генератора коду буде такою:

// індекс поточної лексеми у таблиці лексем

static int pos = 4;

// відступ поточної лексеми для кращої читабельності отриманого коду

static int offset = 0;

// таблиця лексем

extern Token\* TokenTable;

// кількість лексем

extern unsigned int TokensNum;

void generateCCode(FILE\* outFile)

{

fprintf(outFile, "#include <stdio.h>\n#include <stdlib.h>\n\n");

fprintf(outFile, "int main() {\n");

offset++;

variable\_declaration(outFile);

program\_body(outFile);

writeOffset(outFile);

fprintf(outFile, "return 0;\n");

fprintf(outFile, "}\n");

}

Генератор коду працює за методом рекурсивного спуску, а отже функція generateCCode() викликає функцію program\_body(), яка в свою чергу викликає інші функції.

**4. 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.

**4.1. Опис інтерфейсу та інструкції користувачу**

Розроблений транслятор має простий консольний інтерфейс. При запуску програми необхідно ввести ім’я файлу з текстом програми на вхідній мові програмування:

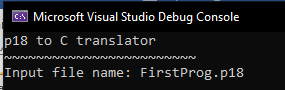


Рис. 4.1. Інтерфейс розробленого транслятора.

Після вводу імені програми отримуємо результати роботи транслятора:

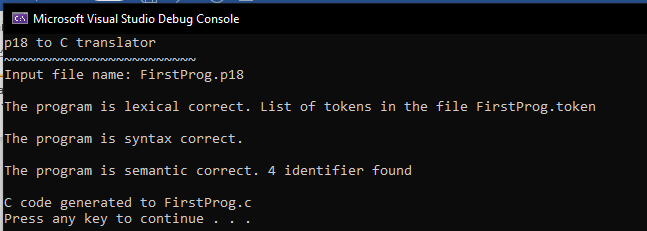


Рис. 4.2. Результати роботи розробленого транслятора.

**4.2. Виявлення лексичних і синтаксичних помилок**

Помилки у вхідній програмі виявляються на етапі лексичного аналізу.

Наприклад, у програмі зробимо лексичну помилку – у 13-му рядку неправильно вкажемо оператор вводу:

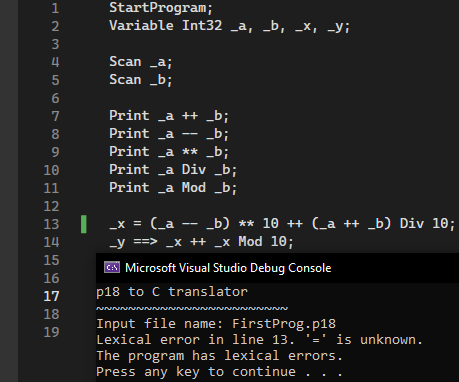


Рис. 4.3. Вивід інформації про лексичну помилку.

Також, помилки у вхідній програмі виявляються й на етапі синтаксичного і семантичного аналізу.

Зробимо синтаксичну помилку – у 7-му рядку, пропущу крапку з комою:

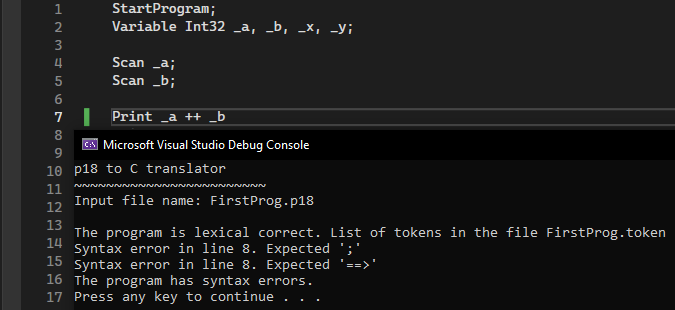


Рис. 4.4. Вивід інформації про синтаксичну помилку.

Зробимо семантичну помилку – не оголосимо змінну “x”:

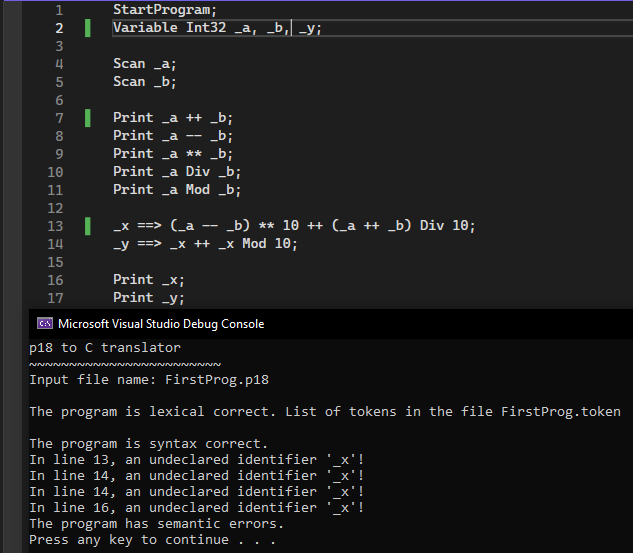


Рис. 4.5. Вивід інформації про семантичну помилку.

**4.3. 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. Вивести на екран:

А + В (результат операції додавання);

А - В (результат операції віднімання);

А \* В (результат операції множення);

А / В (результат операції ділення);

А % В (результат операції отримання залишку від ділення).

3. Обрахувати значення виразів

Х = (А - В) \* 10 + (А + В) / 10

У = Х + Х % 10

4. Вивести значення Х і У на екран

Напишемо програму на вхідній мові програмування(p18):

StartProgram;

Variable Int32 \_a, \_b, \_x, \_y;

Scan \_a;

Scan \_b;

Print \_a ++ \_b;

Print \_a -- \_b;

Print \_a \*\* \_b;

Print \_a Div \_b;

Print \_a Mod \_b;

\_x ==> (\_a -- \_b) \*\* 10 ++ (\_a ++ \_b) Div 10;

\_y ==> \_x ++ \_x Mod 10;

Print \_x;

Print \_y;

EndBlok;

В результаті отримаємо файл з програмою на мові програмування С:

#include <stdio.h>

#include <stdlib.h>

int main() {

int \_a = 0, \_b = 0, \_x = 0, \_y = 0;

printf("Enter \_a:");

scanf\_s("%d", &\_a);

printf("Enter \_b:");

scanf\_s("%d", &\_b);

printf("%d\n", \_a + \_b);

printf("%d\n", \_a - \_b);

printf("%d\n", \_a \* \_b);

printf("%d\n", \_a / (\_b == 0 ? (printf("Error: Division by zero\n"), exit(1), 0) : \_b));

printf("%d\n", \_a % (\_b == 0 ? (printf("Error: Division by zero\n"), exit(1), 0) : \_b));

\_x = (\_a - \_b) \* 10 + (\_a + \_b) / (10 == 0 ? (printf("Error: Division by zero\n"), exit(1), 0) : 10);

\_y = \_x + \_x % (10 == 0 ? (printf("Error: Division by zero\n"), exit(1), 0) : 10);

printf("%d\n", \_x);

printf("%d\n", \_y);

return 0;

}

Отриманий код на мові С протестуємо у новому проєкті у середовищі Visual

Studio 2022 і отримаємо такі результати:

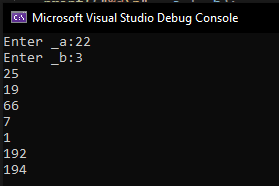


Рис. 4.6. Результати виконання тестової задачі 1.

**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).

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

3. Вивести на екран число 1, якщо усі числа однакові (логічний вираз в умовному операторі має виглядати так: «(А=В) і (А=С) і (В=С)»), інакше вивести 0.

4. Вивести на екран число -1, якщо хоча б одне з чисел від’ємне (логічний вираз в умовному операторі має виглядати так: «(А<0) або (В<0) або (С<0)»), інакше вивести 0.

5. Вивести на екран число 10, якщо число А більше за суму чисел В і С (логічний вираз в умовному операторі має виглядати так: «!(А<(В+С))»), інакше вивести 0.

Напишемо програму на вхідній мові програмування(p18):

StartProgram;

Variable Int32 \_a, \_b, \_c;

Scan \_a;

Scan \_b;

Scan \_c;

If(\_a Gr \_b)

If(\_a Gr \_b)

Print \_a;

Else

Print \_c;

Else

If(\_b Gr \_c)

Print \_b;

Else

Print \_c;

If ((\_a Eq \_b) And (\_a Eq \_c) And (\_b Eq \_c))

Print 1;

Else

Print 0;

If ((\_a Ls 0) Or (\_b Ls 0) Or (\_c Ls 0))

Print -1;

Else

Print 0;

If (Not (\_a Gr (\_b ++ \_c)))

Print 10;

Else

Print 0;

EndBlok;

В результаті отримаємо файл з програмою на мові програмування С:

#include <stdio.h>

#include <stdlib.h>

int main() {

int \_a = 0, \_b = 0, \_c = 0;

printf("Enter \_a:");

scanf\_s("%d", &\_a);

printf("Enter \_b:");

scanf\_s("%d", &\_b);

printf("Enter \_c:");

scanf\_s("%d", &\_c);

if (\_a > \_b) {

if (\_a > \_b) {

printf("%d\n", \_a);

}

else {

printf("%d\n", \_c);

}

}

else {

if (\_b > \_c) {

printf("%d\n", \_b);

}

else {

printf("%d\n", \_c);

}

}

if ((\_a == \_b) && (\_a == \_c) && (\_b == \_c)) {

printf("%d\n", 1);

}

else {

printf("%d\n", 0);

}

if ((\_a < 0) || (\_b < 0) || (\_c < 0)) {

printf("%d\n", -1);

}

else {

printf("%d\n", 0);

}

if (!(\_a < (\_b + \_c))) {

printf("%d\n", 10);

}

else {

printf("%d\n", 0);

}

return 0;

}

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

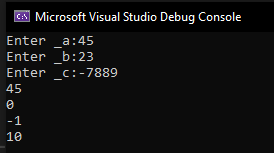


Рис. 4.7. Результати виконання тестової задачі 2.

**Тестова програма «*Циклічний алгоритм*»**

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

2. Вивести на екран квадрати чисел від А до В включно.

3. Обрахувати Х=А\*В за наступним алгоритмом:

Х = 0

Цикл від 1 до А з кроком 1

Цикл від 1 до В з кроком 1

Х = Х + 1

4. Вивести значення Х на екран.

Напишемо програму на вхідній мові програмування(p18):

StartProgram;

Variable Int32 \_a, \_b, \_x, \_i, \_j;

Scan \_a;

Scan \_b;

For \_i ==> \_a To \_b Do

Print \_i \*\* \_i;

\_x ==> 0;

For \_i ==> 1 To \_a Do

For \_j ==> 1 To \_b Do

\_x ==> \_x ++ 1;

Print \_x;

EndBlok;

В результаті отримаємо файл з програмою на мові програмування С:

#include <stdio.h>

#include <stdlib.h>

int main() {

int \_a = 0, \_b = 0, \_x = 0, \_i = 0, \_j = 0;

printf("Enter \_a:");

scanf\_s("%d", &\_a);

printf("Enter \_b:");

scanf\_s("%d", &\_b);

for (int \_i = \_a; \_i < \_b + 1; \_i++) {

printf("%d\n", \_i \* \_i);

}

\_x = 0;

for (int \_i = 1; \_i < \_a; \_i++) {

for (int \_j = 1; \_j < \_b; \_j++) {

\_x = \_x + 1;

}

}

printf("%d\n", \_x);

return 0;

}

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

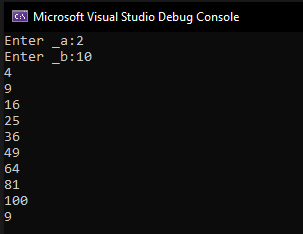


Рис. 4.8. Результати виконання тестової задачі 3.

**ВИСНОВКИ**

В процесі виконання курсового проєкту було виконано наступне:

1. Складено формальний опис мови програмування p18, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.
2. Створено транслятор мови програмування p18, а саме:
   1. Розроблено прямий лексичний аналізатор, орієнтований на розпізнавання лексем, що є заявлені в формальному описі мови програмування.
   2. Розроблено синтаксичний аналізатор на основі низхідного методу. Складено деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура
   3. Розроблено генератор коду, відповідні процедури якого викликаються після перевірки синтаксичним аналізатором коректності запису лексем, мови програмування p18. Вихідним кодом генератора є програма на мові C.
3. Проведене тестування транслятора на тестових програмах за наступними пунктами:
   1. На виявлення лексичних помилок.
   2. На виявлення синтаксичних помилок.
   3. Загальна перевірка роботи транслятора.

Тестування не виявило помилок в роботі транслятора, і всі помилки в тестових програмах на мові p18 були успішно виявлені і відповідно оброблені.

В результаті виконання даної курсового проєкту 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.

2. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. –Харків: НТУ «ХПІ», 2021. – 133 с.

3. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії формальних мов: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 c.

4. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 c.

5. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. – 1038 c.

6. Системне програмування (курсовий проект) [Електронний ресурс] – Режим доступу до ресурсу: <https://vns.lpnu.ua/course/view.php?id=11685>.

7. MIT OpenCourseWare. Computer Language Engineering [Електронний ресурс] – Режим доступу до ресурсу: https://ocw.mit.edu/courses/6-035-computer-language-engineering-spring2010.

8. Рисований О.М. Системне програмування: підручник для студентів напрямку

“Комп’ютерна інженерія” вищих навчальних закладів в 2-х томах. Том 1. – Видання четверте: виправлено та доповнено – Х.: “Слово”, 2015. – 576 с.

9. Рисований О.М. Системне програмування: підручник для студентів напрямку

“Комп’ютерна інженерія” вищих навчальних закладів в 2-х томах. Том 2. – Видання четверте: виправлено та доповнено – Х.: “Слово”, 2015. – 378 с.

**ДОДАТКИ**

**ДОДАТОК А. Таблиці лексем для тестових прикладів**

Тестова програма «Лінійний алгоритм»

**---------------------------------------------------------------------------**

**| TOKEN TABLE |**

**---------------------------------------------------------------------------**

**| line number | token | value | token code | type of token |**

**---------------------------------------------------------------------------**

**| 1 | StartProgram | 0 | 0 | StartProgram |**

**---------------------------------------------------------------------------**

**| 1 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 2 | Variable | 0 | 2 | Variable |**

**---------------------------------------------------------------------------**

**| 2 | Int32 | 0 | 3 | Integer |**

**---------------------------------------------------------------------------**

**| 2 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | , | 0 | 36 | Comma |**

**---------------------------------------------------------------------------**

**| 2 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | , | 0 | 36 | Comma |**

**---------------------------------------------------------------------------**

**| 2 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | , | 0 | 36 | Comma |**

**---------------------------------------------------------------------------**

**| 2 | \_y | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 4 | Scan | 0 | 5 | Input |**

**---------------------------------------------------------------------------**

**| 4 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 4 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 5 | Scan | 0 | 5 | Input |**

**---------------------------------------------------------------------------**

**| 5 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 5 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 7 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 7 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 7 | ++ | 0 | 21 | Add |**

**---------------------------------------------------------------------------**

**| 7 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 7 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 8 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 8 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 8 | -- | 0 | 22 | Sub |**

**---------------------------------------------------------------------------**

**| 8 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 8 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 9 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 9 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 9 | \*\* | 0 | 23 | Mul |**

**---------------------------------------------------------------------------**

**| 9 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 9 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 10 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 10 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 10 | Div | 0 | 24 | Div |**

**---------------------------------------------------------------------------**

**| 10 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 10 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 11 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 11 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 11 | Mod | 0 | 25 | Mod |**

**---------------------------------------------------------------------------**

**| 11 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 11 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 13 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 13 | ==> | 0 | 20 | Assign |**

**---------------------------------------------------------------------------**

**| 13 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 13 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 13 | -- | 0 | 22 | Sub |**

**---------------------------------------------------------------------------**

**| 13 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 13 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 13 | \*\* | 0 | 23 | Mul |**

**---------------------------------------------------------------------------**

**| 13 | 10 | 10 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 13 | ++ | 0 | 21 | Add |**

**---------------------------------------------------------------------------**

**| 13 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 13 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 13 | ++ | 0 | 21 | Add |**

**---------------------------------------------------------------------------**

**| 13 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 13 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 13 | Div | 0 | 24 | Div |**

**---------------------------------------------------------------------------**

**| 13 | 10 | 10 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 13 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 14 | \_y | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 14 | ==> | 0 | 20 | Assign |**

**---------------------------------------------------------------------------**

**| 14 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 14 | ++ | 0 | 21 | Add |**

**---------------------------------------------------------------------------**

**| 14 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 14 | Mod | 0 | 25 | Mod |**

**---------------------------------------------------------------------------**

**| 14 | 10 | 10 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 14 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 16 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 16 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 16 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 17 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 17 | \_y | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 17 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 19 | EndBlok | 0 | 4 | End |**

**---------------------------------------------------------------------------**

**| 19 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

Тестова програма «Алгоритм з розгалуженням»

**---------------------------------------------------------------------------**

**| TOKEN TABLE |**

**---------------------------------------------------------------------------**

**| line number | token | value | token code | type of token |**

**---------------------------------------------------------------------------**

**| 1 | StartProgram | 0 | 0 | StartProgram |**

**---------------------------------------------------------------------------**

**| 1 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 2 | Variable | 0 | 2 | Variable |**

**---------------------------------------------------------------------------**

**| 2 | Int32 | 0 | 3 | Integer |**

**---------------------------------------------------------------------------**

**| 2 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | , | 0 | 36 | Comma |**

**---------------------------------------------------------------------------**

**| 2 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | , | 0 | 36 | Comma |**

**---------------------------------------------------------------------------**

**| 2 | \_c | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 4 | Scan | 0 | 5 | Input |**

**---------------------------------------------------------------------------**

**| 4 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 4 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 5 | Scan | 0 | 5 | Input |**

**---------------------------------------------------------------------------**

**| 5 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 5 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 6 | Scan | 0 | 5 | Input |**

**---------------------------------------------------------------------------**

**| 6 | \_c | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 6 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 8 | If | 0 | 7 | If |**

**---------------------------------------------------------------------------**

**| 8 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 8 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 8 | Gr | 0 | 28 | Greate |**

**---------------------------------------------------------------------------**

**| 8 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 8 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 9 | If | 0 | 7 | If |**

**---------------------------------------------------------------------------**

**| 9 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 9 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 9 | Gr | 0 | 28 | Greate |**

**---------------------------------------------------------------------------**

**| 9 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 9 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 10 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 10 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 10 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 11 | Else | 0 | 8 | Else |**

**---------------------------------------------------------------------------**

**| 12 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 12 | \_c | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 12 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 13 | Else | 0 | 8 | Else |**

**---------------------------------------------------------------------------**

**| 14 | If | 0 | 7 | If |**

**---------------------------------------------------------------------------**

**| 14 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 14 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 14 | Gr | 0 | 28 | Greate |**

**---------------------------------------------------------------------------**

**| 14 | \_c | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 14 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 15 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 15 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 15 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 16 | Else | 0 | 8 | Else |**

**---------------------------------------------------------------------------**

**| 17 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 17 | \_c | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 17 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 19 | If | 0 | 7 | If |**

**---------------------------------------------------------------------------**

**| 19 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 19 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 19 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 19 | Eq | 0 | 26 | Equality |**

**---------------------------------------------------------------------------**

**| 19 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 19 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 19 | And | 0 | 31 | And |**

**---------------------------------------------------------------------------**

**| 19 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 19 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 19 | Eq | 0 | 26 | Equality |**

**---------------------------------------------------------------------------**

**| 19 | \_c | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 19 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 19 | And | 0 | 31 | And |**

**---------------------------------------------------------------------------**

**| 19 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 19 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 19 | Eq | 0 | 26 | Equality |**

**---------------------------------------------------------------------------**

**| 19 | \_c | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 19 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 19 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 20 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 20 | 1 | 1 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 20 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 21 | Else | 0 | 8 | Else |**

**---------------------------------------------------------------------------**

**| 22 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 22 | 0 | 0 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 22 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 24 | If | 0 | 7 | If |**

**---------------------------------------------------------------------------**

**| 24 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 24 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 24 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 24 | Ls | 0 | 29 | Less |**

**---------------------------------------------------------------------------**

**| 24 | 0 | 0 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 24 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 24 | Or | 0 | 32 | Or |**

**---------------------------------------------------------------------------**

**| 24 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 24 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 24 | Ls | 0 | 29 | Less |**

**---------------------------------------------------------------------------**

**| 24 | 0 | 0 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 24 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 24 | Or | 0 | 32 | Or |**

**---------------------------------------------------------------------------**

**| 24 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 24 | \_c | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 24 | Ls | 0 | 29 | Less |**

**---------------------------------------------------------------------------**

**| 24 | 0 | 0 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 24 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 24 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 25 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 25 | -1 | -1 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 25 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 26 | Else | 0 | 8 | Else |**

**---------------------------------------------------------------------------**

**| 27 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 27 | 0 | 0 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 27 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 29 | If | 0 | 7 | If |**

**---------------------------------------------------------------------------**

**| 29 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 29 | Not | 0 | 30 | Not |**

**---------------------------------------------------------------------------**

**| 29 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 29 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 29 | Ls | 0 | 29 | Less |**

**---------------------------------------------------------------------------**

**| 29 | ( | 0 | 33 | LBraket |**

**---------------------------------------------------------------------------**

**| 29 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 29 | ++ | 0 | 21 | Add |**

**---------------------------------------------------------------------------**

**| 29 | \_c | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 29 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 29 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 29 | ) | 0 | 34 | RBraket |**

**---------------------------------------------------------------------------**

**| 30 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 30 | 10 | 10 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 30 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 31 | Else | 0 | 8 | Else |**

**---------------------------------------------------------------------------**

**| 32 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 32 | 0 | 0 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 32 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 34 | EndBlok | 0 | 4 | End |**

**---------------------------------------------------------------------------**

**| 34 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

Тестова програма «Циклічний алгоритм»

**---------------------------------------------------------------------------**

**| TOKEN TABLE |**

**---------------------------------------------------------------------------**

**| line number | token | value | token code | type of token |**

**---------------------------------------------------------------------------**

**| 1 | StartProgram | 0 | 0 | StartProgram |**

**---------------------------------------------------------------------------**

**| 1 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 2 | Variable | 0 | 2 | Variable |**

**---------------------------------------------------------------------------**

**| 2 | Int32 | 0 | 3 | Integer |**

**---------------------------------------------------------------------------**

**| 2 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | , | 0 | 36 | Comma |**

**---------------------------------------------------------------------------**

**| 2 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | , | 0 | 36 | Comma |**

**---------------------------------------------------------------------------**

**| 2 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | , | 0 | 36 | Comma |**

**---------------------------------------------------------------------------**

**| 2 | \_i | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | , | 0 | 36 | Comma |**

**---------------------------------------------------------------------------**

**| 2 | \_j | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 2 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 4 | Scan | 0 | 5 | Input |**

**---------------------------------------------------------------------------**

**| 4 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 4 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 5 | Scan | 0 | 5 | Input |**

**---------------------------------------------------------------------------**

**| 5 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 5 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 7 | For | 0 | 10 | For |**

**---------------------------------------------------------------------------**

**| 7 | \_i | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 7 | ==> | 0 | 20 | Assign |**

**---------------------------------------------------------------------------**

**| 7 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 7 | To | 0 | 11 | To |**

**---------------------------------------------------------------------------**

**| 7 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 7 | ++ | 0 | 21 | Add |**

**---------------------------------------------------------------------------**

**| 7 | 1 | 1 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 7 | Do | 0 | 13 | Number |**

**---------------------------------------------------------------------------**

**| 8 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 8 | \_i | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 8 | \*\* | 0 | 23 | Mul |**

**---------------------------------------------------------------------------**

**| 8 | \_i | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 8 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 10 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 10 | ==> | 0 | 20 | Assign |**

**---------------------------------------------------------------------------**

**| 10 | 0 | 0 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 10 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 11 | For | 0 | 10 | For |**

**---------------------------------------------------------------------------**

**| 11 | \_i | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 11 | ==> | 0 | 20 | Assign |**

**---------------------------------------------------------------------------**

**| 11 | 1 | 1 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 11 | To | 0 | 11 | To |**

**---------------------------------------------------------------------------**

**| 11 | \_a | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 11 | Do | 0 | 13 | Identifier |**

**---------------------------------------------------------------------------**

**| 12 | For | 0 | 10 | For |**

**---------------------------------------------------------------------------**

**| 12 | \_j | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 12 | ==> | 0 | 20 | Assign |**

**---------------------------------------------------------------------------**

**| 12 | 1 | 1 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 12 | To | 0 | 11 | To |**

**---------------------------------------------------------------------------**

**| 12 | \_b | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 12 | Do | 0 | 13 | Identifier |**

**---------------------------------------------------------------------------**

**| 13 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 13 | ==> | 0 | 20 | Assign |**

**---------------------------------------------------------------------------**

**| 13 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 13 | ++ | 0 | 21 | Add |**

**---------------------------------------------------------------------------**

**| 13 | 1 | 1 | 19 | Number |**

**---------------------------------------------------------------------------**

**| 13 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 15 | Print | 0 | 6 | Output |**

**---------------------------------------------------------------------------**

**| 15 | \_x | 0 | 18 | Identifier |**

**---------------------------------------------------------------------------**

**| 15 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**| 17 | EndBlok | 0 | 4 | End |**

**---------------------------------------------------------------------------**

**| 17 | ; | 0 | 35 | Semicolon |**

**---------------------------------------------------------------------------**

**ДОДАТОК Б. С код, отриманий на виході транслятора для тестових прикладів**

Тестова програма «Лінійний алгоритм»

#include <stdio.h>

#include <stdlib.h>

int main() {

int \_a = 0, \_b = 0, \_x = 0, \_y = 0;

printf("Enter \_a:");

scanf\_s("%d", &\_a);

printf("Enter \_b:");

scanf\_s("%d", &\_b);

printf("%d\n", \_a + \_b);

printf("%d\n", \_a - \_b);

printf("%d\n", \_a \* \_b);

printf("%d\n", \_a / (\_b == 0 ? (printf("Error: Division by zero\n"), exit(1), 0) : \_b));

printf("%d\n", \_a % (\_b == 0 ? (printf("Error: Division by zero\n"), exit(1), 0) : \_b));

\_x = (\_a - \_b) \* 10 + (\_a + \_b) / (10 == 0 ? (printf("Error: Division by zero\n"), exit(1), 0) : 10);

\_y = \_x + \_x % (10 == 0 ? (printf("Error: Division by zero\n"), exit(1), 0) : 10);

printf("%d\n", \_x);

printf("%d\n", \_y);

return 0;

}

Тестова програма «Алгоритм з розгалуженням»

#include <stdio.h>

#include <stdlib.h>

int main() {

int \_a = 0, \_b = 0, \_c = 0;

printf("Enter \_a:");

scanf\_s("%d", &\_a);

printf("Enter \_b:");

scanf\_s("%d", &\_b);

printf("Enter \_c:");

scanf\_s("%d", &\_c);

if (\_a > \_b) {

if (\_a > \_b) {

printf("%d\n", \_a);

}

else {

printf("%d\n", \_c);

}

}

else {

if (\_b > \_c) {

printf("%d\n", \_b);

}

else {

printf("%d\n", \_c);

}

}

if ((\_a == \_b) && (\_a == \_c) && (\_b == \_c)) {

printf("%d\n", 1);

}

else {

printf("%d\n", 0);

}

if ((\_a < 0) || (\_b < 0) || (\_c < 0)) {

printf("%d\n", -1);

}

else {

printf("%d\n", 0);

}

if (!(\_a < (\_b + \_c))) {

printf("%d\n", 10);

}

else {

printf("%d\n", 0);

}

return 0;

}

Тестова програма «Циклічний алгоритм»

#include <stdio.h>

#include <stdlib.h>

int main() {

int \_a = 0, \_b = 0, \_x = 0, \_i = 0, \_j = 0;

printf("Enter \_a:");

scanf\_s("%d", &\_a);

printf("Enter \_b:");

scanf\_s("%d", &\_b);

for (int \_i = \_a; \_i < \_b + 1; \_i++) {

printf("%d\n", \_i \* \_i);

}

\_x = 0;

for (int \_i = 1; \_i < \_a; \_i++) {

for (int \_j = 1; \_j < \_b; \_j++) {

\_x = \_x + 1;

}

}

printf("%d\n", \_x);

return 0;

}