

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-:

Перевірів:

Львів-2024

## ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ

1. Цільова мова транслятора – мова програмування C 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 C або асемблера;*
  - об'єктний файл;*
  - виконуваний 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 проектування:

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 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.
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:     **start**  
                                   **var ... ;**  
                                   **stop**

Оператори вводу-виводу: **input, output**

Оператор присвоєння: **:=**

Оператор: **if – then [- else ]**

Регістр ідентифікаторів: **low4**

Операції:

- арифметичні: **+** , **-** , **\*** , **/**
- порівняння: **>** , **<** , **=** , **<>**
- логічні: **!** , **&** , **|**

Тип даних: **integer**

Коментар: **// ...**

Програма на вхідній мові програмування має починатись з ключового слова **start**, далі має йти розділ опису змінних **var**. Між розділом **var** і ключовим словом **stop** розміщуються оператори програми. Операторів є 4: оператор вводу даних **input**, оператор виводу даних **output**, оператор присвоєння **:=** і умовний оператор **if – then [- else ]**. Кожен оператор має завершуватись символом крапка з комою **;**. Оператор присвоєння дозволяє присвоїти деякій змінній значення арифметичного виразу. Допустимі арифметичні операції - **+** , **-** , **\*** , **/**. Операндами можуть бути змінні, цілі додатні константи і інші вирази, взяті в дужки. В умовному операторі використовуються логічні вирази, допустимі такі операції порівняння **>** , **<** , **=** , **<>** і такі логічні операції **!** , **&** , **|**. Ідентифікатори (імена змінних) можуть бути довжиною до 4-х символів і складаються лише з маленьких латинських літер або цифр. Перший символ ідентифікатора завжди літера. Тип даних лише один – **integer**, при оголошенні декількох змінних вони записуються через кому, вкінці опису змінних ставиться символ крапка з комою **;**. Коментарі починаються з двох косих ліній **// ...**.

Приклади оголошення змінних:

integer a;

integer a, b, c;

integer a, max;

Приклади ідентифікаторів:

a, a1, a10, min, max, min1

Приклади арифметичних виразів:

$0 - 75$

$y + 68$

$a + b * c + 76$

$c * (a + b) - 50 / b$

Приклади логічних виразів:

$a > b$

$a > b \ \& \ a > c$

$! ( a < c )$

$a <> b$

$\text{min} < 0 - 100$

## АНОТАЦІЯ

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

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

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

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

Розроблене програмне забезпечення налаштоване і протестоване на тестових прикладах.

## ЗМІСТ

ВСТУП.....	8
1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ.....	9
2. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ .....	10
2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура. .....	10
2.2. Опис термінальних символів та ключових слів. ....	12
3. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ.....	13
3.1. Вибір технології програмування. ....	13
3.2. Проектування таблиць транслятора та вибір структур даних. ....	13
3.3. Розробка лексичного аналізатора.....	14
3.3.1. Розробка алгоритму роботи лексичного аналізатора. ....	16
3.3.2. Опис програми реалізації лексичного аналізатора. ....	18
3.4. Розробка синтаксичного та семантичного аналізатора. ....	19
3.4.1. Розробка дерева граматичного розбору. ....	21
3.4.2. Розробка алгоритму роботи синтаксичного і семантичного аналізатора. ....	22
3.4.3. Опис програми реалізації синтаксичного та семантичного аналізатора. ....	26
3.5. Розробка генератора коду. ....	27
3.5.1. Розробка алгоритму роботи генератора коду.....	28
3.5.2. Опис програми реалізації генератора коду.....	31
4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА ...	36
4.1. Опис інтерфейсу та інструкції користувачу. ....	36
4.2. Виявлення лексичних і синтаксичних помилок. ....	37
4.3. Перевірка роботи транслятора за допомогою тестових задач. ....	39
ВИСНОВКИ .....	43
СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ.....	44
ДОДАТКИ.....	45

## **ВСТУП**



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

## 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 рисою |, 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> = **'start'** **'var'** <оголошення змінних> **';** <тіло програми> **'stop'**  
 <оголошення змінних> = [<тип даних> <список змінних>]  
 <тип даних> = **'integer'**  
 <список змінних> = <ідентифікатор> { **' , '** <ідентифікатор> }  
 <ідентифікатор> = <буква> { <буква або цифра> }  
 <буква або цифра> = <буква> | <цифра>  
 <буква> = **'a' | 'b' | 'c' | 'd' | ... | 'z'**  
 <цифра> = **'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'**  
 <тіло програми> = <оператор> **' ; '** { <оператор> **' ; '** }  
 <оператор> = <присвоєння> | <ввід> | <вивід> | <умовний оператор> |  
 <складений оператор>  
 <присвоєння> = <ідентифікатор> **' : = '** <арифметичний вираз>  
 <арифметичний вираз> = <доданок> { **' + '** <доданок> | **' - '** <доданок> }  
 <доданок> = <множник> { **' \* '** <множник> | **' / '** <множник> }  
 <множник> = <ідентифікатор> | <число> | **' ( '** <арифметичний вираз> **' ) '**  
 <число> = <цифра> { <цифра> }  
 <ввід> = **'input'** <ідентифікатор>  
 <вивід> = **'output'** <ідентифікатор>  
 <умовний оператор> = **'if'** <логічний вираз> **'then'** <оператор> [ **'else'** <оператор> ]  
 <логічний вираз> = <вираз l> { **' | '** <вираз l> }  
 <вираз l> = <порівняння> { **' & '** <порівняння> }  
 <порівняння> = <операція порівняння> | **' ! '** **' ( '** <логічний вираз> **' ) '**  
 | **' ( '** <логічний вираз> **' ) '**  
 <операція порівняння> = <арифметичний вираз> <менше-більше> <арифметичний вираз>  
 <менше-більше> = **' > ' | ' < ' | ' = ' | ' < > '**  
 <складений оператор> = **'start'** <тіло програми> **'stop'**

## 2.2. Опис термінальних символів та ключових слів.

Визначаємо термінальні символи і ключові слова:

- **start** – початок програми
- **var** – оголошення змінних
- **stop** – кінець програми
- **integer** – тип даних
- **input** – оператор вводу
- **output** – оператор виводу
- **if, then, else** – умовний оператор
- **:=** – оператор присвоєння
- **+** – додавання
- **-** – віднімання
- **\*** – множення
- **/** – ділення
- **>** – більше
- **<** – менше
- **=** – рівність
- **<>** – нерівність
- **!** – заперечення
- **&** – логічне І
- **|** – логічне АБО
- **;** – кінець оператора
- **,** – розділювач змінних
- **(** – відкрита дужка
- **)** – закрита дужка
- **//** – початок коментаря
- **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/C++.

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

#### 3.2. Проектування 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,    // start
    Variable,         // var
    Type,             // integer
    EndProgram,       // stop
    Input,            // input
    Output,           // output
    If,               // if
    Then,             // then
    Else,             // else
    Identifier,       // Identifier
    Number,           // number
    Assign,           // :=
    Add,              // +
    Sub,              // -
    Mul,              // *
    Div,              // /
    Equality,         // =
}
```

```

    NotEquality,    // <>
    Greate,        // >
    Less,          // <
    Not,           // !
    And,           // &
    Or,            // |
    LBraket,       // (
    RBraket,       // )
    Semicolon,     // ;
    Comma,         // ,
    Unknown
};

// структура для зберігання інформації про лексему
struct Token
{
    char name[16];    // ім'я лексеми
    int value;        // значення лексеми (для цілих констант)
    int line;         // номер рядка
    TypeOfTokens type; // тип лексеми
};

// структура для зберігання інформації про ідентифікатор
struct Id
{
    char name[16];
};

```

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. Вхідна програма проглядається послідовно з початку до кінця. Базові елементи, або лексичні одиниці, розділяються пробілами, знаками операцій і спеціальними символами (новий рядок, знак табуляції), і таким чином виділяються та розпізнаються ідентифікатори, літерали і термінальні символи (операції, 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 компіляції звертатись лексеми не як до послідовності символів, а як до унікального номера лексеми, що значно спрощує роботу синтаксичного аналізатора: легко перевірити 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 відкидає коментарі, оскільки вони не мають ніякого впливу на виконання програми, отже ж й на синтаксичний розбір та генерацію коду.

Розділимо лексеми на типи або лексичні класи:

- Ключові слова (0 – **start**, 1 – **var**, 2 – **stop**, 3 – **input**, 4 – **output**, 5 – **integer**, 6 – **if**, 7 – **then**, 8 – **else**)
- Ідентифікатори (9 – починається з літери, далі або маленька літера або цифра, максимум 4 символи)
- Числові константи (10 - ціле число без знаку)
- Оператор присвоєння (11 – **:=**)
- Знаки операції (12 – **+**, 13 – **-**, 14 – **\***, 15 – **/**, 16 – **>**, 17 – **<**, 18 – **=**, 19 – **<>**, 20 – **!**, 21 – **&**, 22 – **|**)
- Розділювачі (25 – **;**, 26 – **,**)
- Дужки (23 – **(**, 24 – **)**)

- Невідома лексема (27 – 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;
- **Letter** – перший символ буква, розпізнавання слів (ключові слова і ідентифікатори);
- **Digit** – перший символ цифра, розпізнавання числових констант;
- **Separators** – видалення пробілів, символів табуляції і переходу на новий рядок;
- **Scomment** – перший символ “/”, можливо далі йде коментар;
- **Comment** – видалення тексту коментаря;
- **Another** – опрацювання інших символів.

У стані **Letter** 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, вважаємо його ідентифікатором при умові, що довжина ланцюжка не більше 4-х символів, 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 **Scomment** читаємо наступний символ, якщо це знак “/”, то далі до кінця рядка йде коментар, який можна проігнорувати, переходимо до стану **Comment**. Якщо ж наступний символ не знак “/”, то вважаємо що поточна лексема це знак операції ділення, читаємо наступний символ і переходимо до стану **Finish**.

У стані **Comment** читаємо символи, поки не зустрінеється символ переходу на новий рядок, після цього 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 дві лексеми, які вимагають ще читання наступного символу з файлу – це оператор присвоєння “:=” і операція перевірки на нерівність “ $\diamond$ ”. 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.

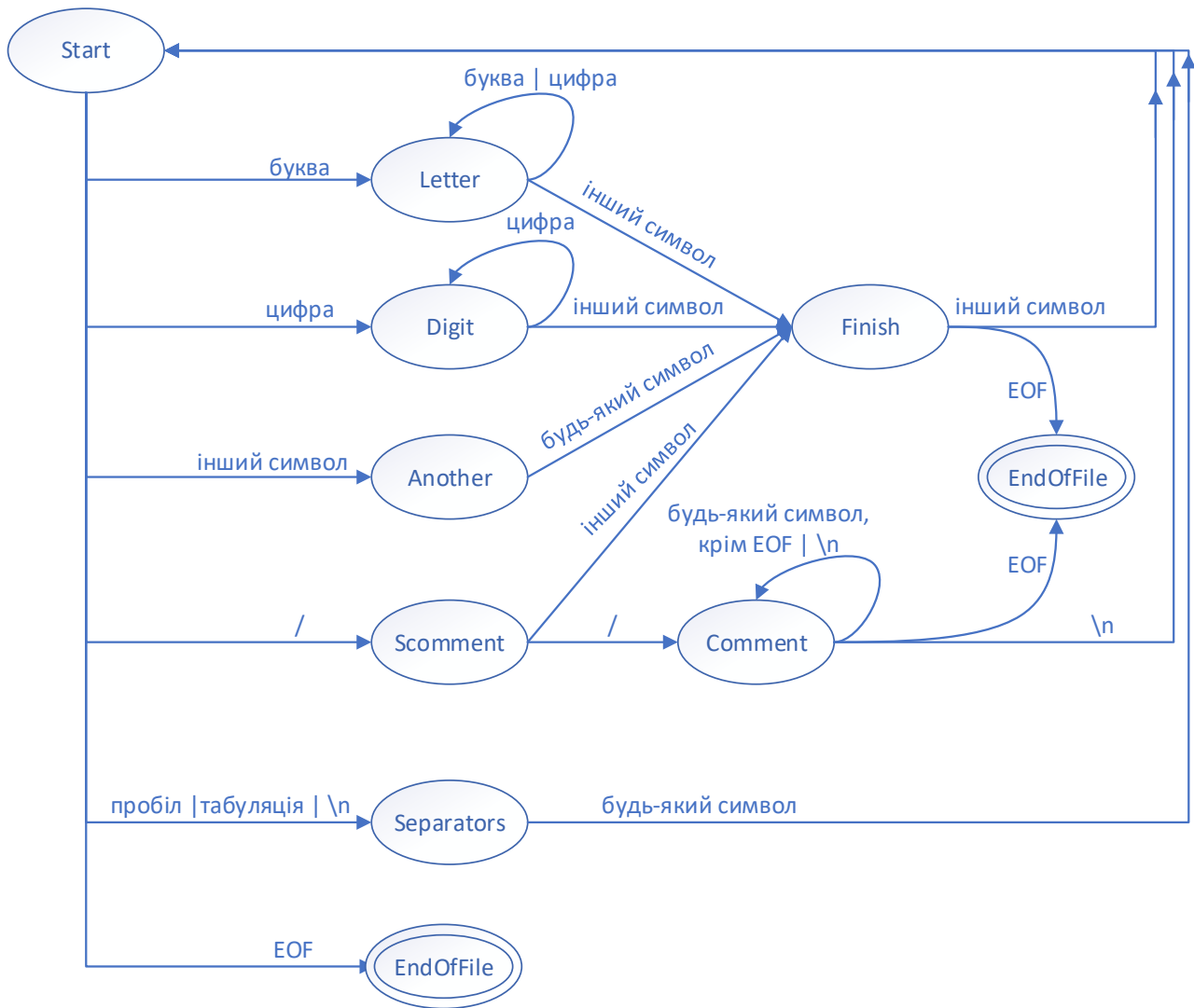


Рис. 3.1. Граф-схема алгоритму роботи лексичного аналізатора.

### 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
    Letter,         // опрацювання слів (ключові слова і
    ідентифікатори)
    Digit,          // опрацювання цифри
    Separators,     // видалення пробілів, символів табуляції і
    переходу на новий рядок
    Another,        // опрацювання інших символів
    EndOfFile,      // кінець файлу
    SComment,       // початок коментаря
    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  $O(n^3)$  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 дерева граматичного розбору.

Схема дерева розбору виглядає наступним чином:



Рис. 3.2. Дерево граматичного розбору.

### 3.4.2. 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> = 'start' 'var' <оголошення змінних> ';' <тіло програми> 'stop'
void program();

<оголошення змінних> = [<тип даних> <список змінних>]
void variable_declaration();

<тип даних> = 'integer'
<список змінних> = <ідентифікатор> { ',' <ідентифікатор> }
void variable_list();

<тіло програми> = <оператор> ';' { <оператор> ';' }
void program_body();

<оператор> = <присвоєння> | <ввід> | <вивід> | <умовний оператор> | <складений
оператор>
void statement();

<присвоєння> = <ідентифікатор> ':=' <арифметичний вираз>
void assignment();

<арифметичний вираз> = <доданок> { '+' <доданок> | '-' <доданок> }
void arithmetic_expression();

<доданок> = <множник> { '*' <множник> | '/' <множник> }
void term();

<множник> = <ідентифікатор> | <число> | '(' <арифметичний вираз> ')'
void factor();

<ввід> = 'input' <ідентифікатор>
void input();

<вивід> = 'output' <ідентифікатор>
void output();

<умовний оператор> = 'if' <логічний вираз> 'then' <оператор> [ 'else' <оператор> ]
void conditional();

```

```

<логічний вираз> = <вираз l> { '|' <вираз l> }
void logical_expression();

<вираз l> = <порівняння> { '&' <порівняння> }
void and_expression();

<порівняння> = <операція порівняння> | '!' '(' <логічний вираз> ')' | '(' <логічний вираз> ')'
<операція порівняння> = <арифметичний вираз> <менше-більше> <арифметичний вираз>
<менше-більше> = '>' | '<' | '=' | '<>'
void comparison();

<складений оператор> = 'start' <тіло програми> 'stop'
void compound_statement();

```

Блок-схема алгоритму роботи синтаксичного аналізатора виглядатиме наступним чином:

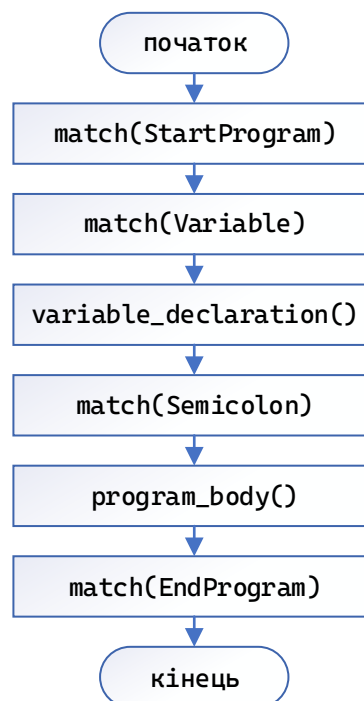


Рис. 3.3. Блок-схема алгоритму роботи синтаксичного аналізатора.

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:

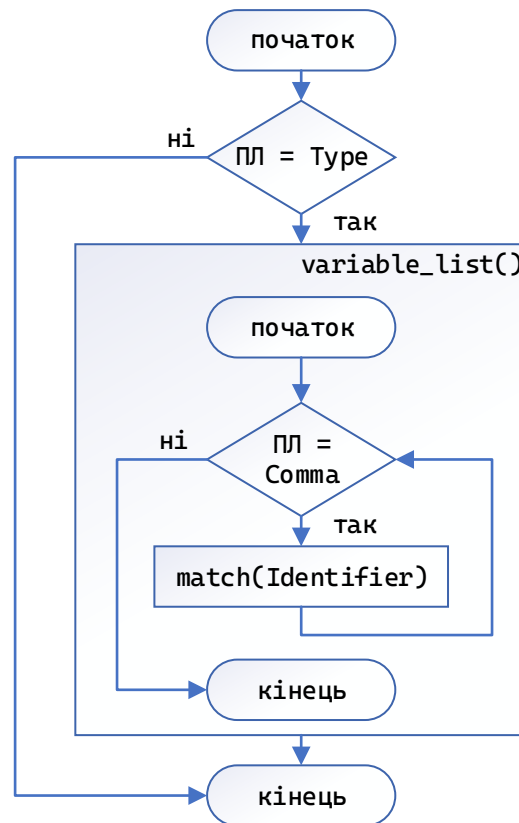


Рис. 3.4. Блок-схема алгоритму роботи функції `variable_declaration()`.

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.



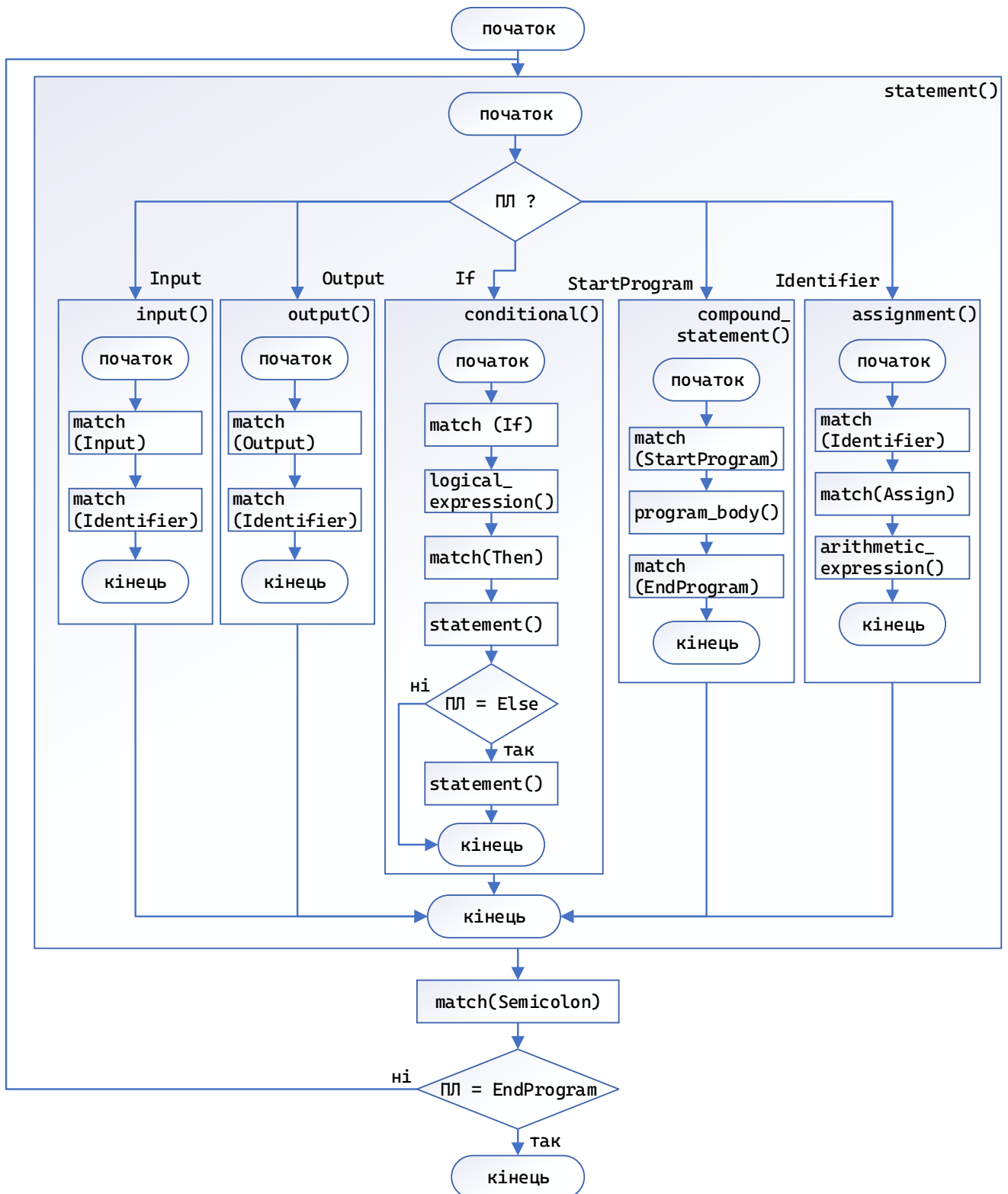


Рис. 3.5. Блок-схема алгоритму роботи функції *program\_body()*.

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.

### 3.4.3. Опис програми реалізації синтаксичного та семантичного аналізатора.

Структура синтаксичного аналізатора буде такою:

```
// Вхідна таблиця лексем
extern Token* TokenTable;
int pos = 0;

void parser()
{
    program();
    printf("\nThe program is syntax correct.\n");
}
```

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 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.

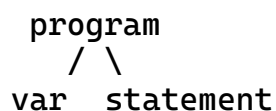
Кожен вузол відповідає певному типу конструкції коду (наприклад, оператору додавання, виклику функції, оголошенню змінної).

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).

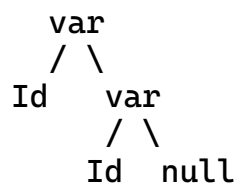
### 3.5.1. Розробка алгоритму роботи генератора коду.

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

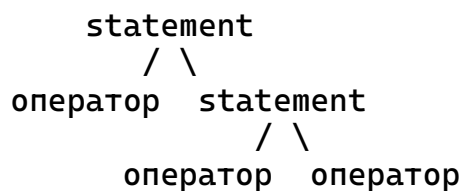
Програма має вигляд:



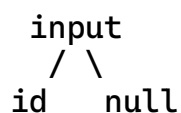
Оголошення змінних:



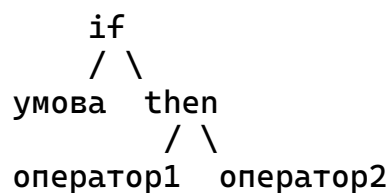
Тіло програми:



Оператор вводу (виводу):



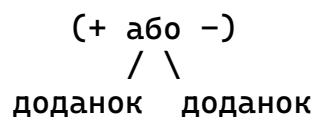
Умовний оператор:



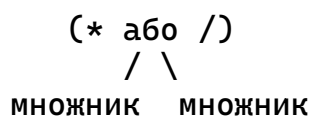
Оператор присвоєння:



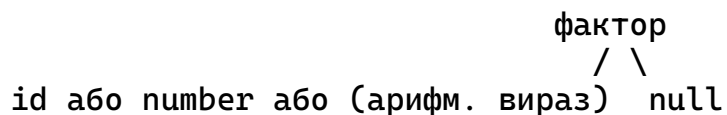
Арифметичний вираз:



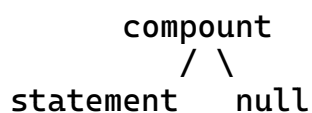
Доданок:



Множник:



Складений оператор:



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.

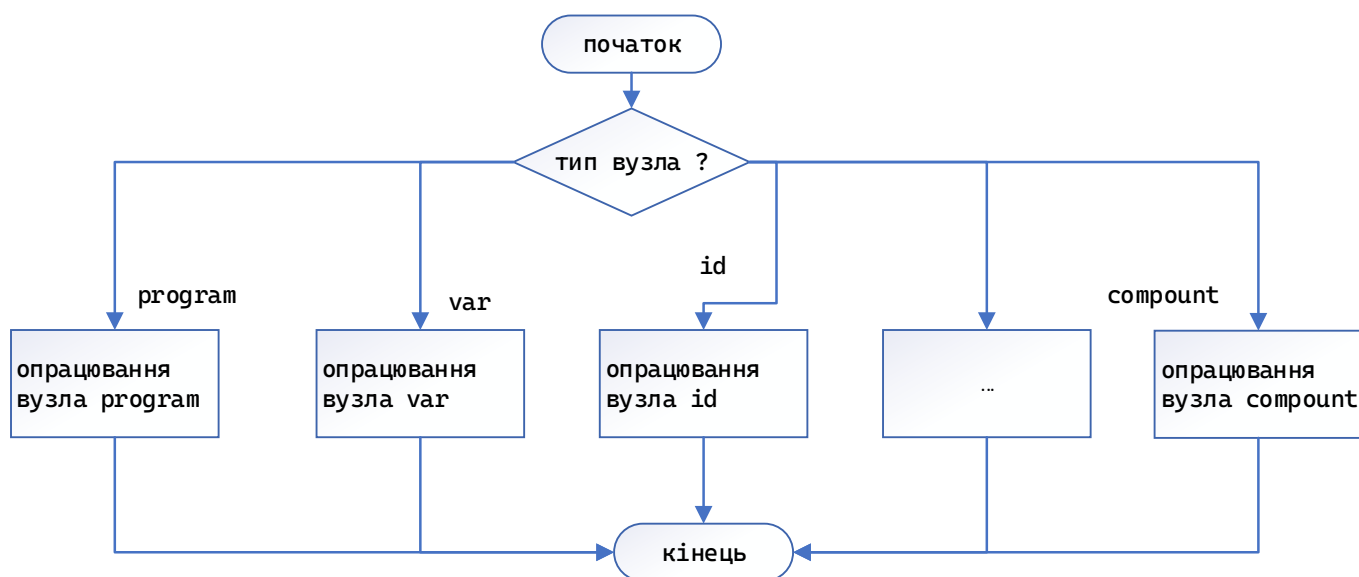


Рис. 3.6. Блок-схема алгоритму роботи генератора коду.

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

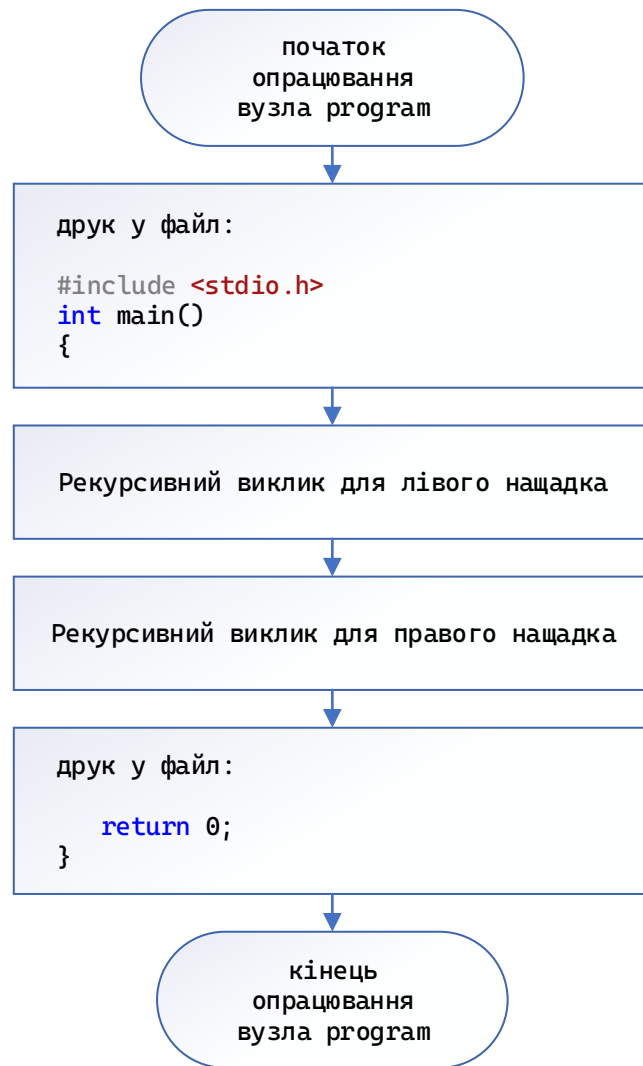


Рис. 3.7. Блок-схема алгоритму опрацювання вузла *program*.

### 3.5.2. Опис програми реалізації генератора коду.

Генерувати вихідний код будемо з абстрактного синтаксичного дерева.

Створимо таку структуру даних для зберігання вузлів дерева:

```

// структура, яка описує вузол абстрактного синтаксичного дерева (AST)
struct ASTNode
{
    TypeOfNodes nodetype; // Тип вузла
    char name[16];         // Ім'я вузла
    struct ASTNode* left;  // Лівий нащадок
    struct ASTNode* right; // Правий нащадок
};
  
```

// перерахування, яке описує всі можливі вузли абстрактного синтаксичного дерева

```
enum TypeOfNodes
{
    program_node,
    var_node,
    input_node,
    output_node,
    if_node,
    then_node,
    id_node,
    num_node,
    assign_node,
    add_node,
    sub_node,
    mul_node,
    div_node,
    or_node,
    and_node,
    not_node,
    cmp_node,
    statement_node,
    compount_node
};
```

Функція створення вузла дерева:

```
// функція створення вузла AST
ASTNode* createNode(TypeOfNodes type, const char* name, ASTNode*
left, ASTNode* right)
```

Функція створення абстрактного синтаксичного дерева реалізована методом рекурсивного спуску:

```
// функція синтаксичного аналізу і створення абстрактного
синтаксичного дерева
ASTNode* ParserAST()
{
    ASTNode* tree = program();

    printf("\nParsing completed. AST created.\n");

    return tree;
}
```



Також напишемо функції для друку абстрактного синтаксичного дерева:

```
// функція для друку AST у вигляді дерева на екран
void PrintAST(ASTNode* node, int level);

// функція для друку AST у вигляді дерева у файл
void PrintASTToFile(ASTNode* node, int level, FILE* outFile);
```

Напишемо рекурсивну функцію, яка буде генерувати код на мові програмування C з абстрактного синтаксичного дерева:

```
// Рекурсивна функція для генерації коду з AST
void generateCodefromAST(ASTNode* node, FILE* outFile)
{
    if (node == NULL)
        return;

    switch (node->nodetype)
    {
        case ...
        ...
        case ...
        default:
            fprintf(stderr, "Unknown node type: %d\n", node->nodetype);
            break;
    }
}
```

Тепер розглянемо варіанти генерації коду для можливих вузлів дерева.

Отже опрацювання вузла `program_node` буде виглядати таким чином:

```
fprintf(outFile, "#include <stdio.h>\n\nint main() \n{\n");
generateCodefromAST(node->left, outFile); // Оголошення змінних
generateCodefromAST(node->right, outFile); // Тіло програми
fprintf(outFile, "    return 0;\n}\n");
```

При генерації вихідного коду для блоку оголошення змінних будемо опрацьовувати вузли `var_node`:

```
// Якщо є права частина (інші змінні), то генеруємо для них код
if (node->right != NULL)
    generateCodefromAST(node->right, outFile);
// Виводимо тип змінних (в даному випадку int)
fprintf(outFile, "    int ");
generateCodefromAST(node->left, outFile);
fprintf(outFile, ";\n"); // Завершуємо оголошення змінних
```

Опрацювання вузлів `id_node` і `num_node` буде полягати у друці імені вузла (ім'я ідентифікатора):

```
fprintf(outFile, "%s", node->name);
```

При генерації вихідного коду для блоку тіло програми будемо опрацьовувати вузли `statement_node`:

```
generateCodefromAST(node->left, outFile);
if (node->right != NULL)
    generateCodefromAST(node->right, outFile);
```

Опрацювання вузла `input_node` буде виглядати таким чином:

```
fprintf(outFile, "    printf(\"Enter \");
generateCodefromAST(node->left, outFile);
fprintf(outFile, ":\");\n");
fprintf(outFile, "    scanf_s(\"%d\", &);
generateCodefromAST(node->left, outFile);
fprintf(outFile, ");\n");
```

А вузла `output_node` буде виглядати таким чином:

```
fprintf(outFile, "    printf(\"%d\\n\", ");
generateCodefromAST(node->left, outFile);
fprintf(outFile, ");\n");
```

Опрацювання вузла `if_node` буде виглядати таким чином:

```
fprintf(outFile, "    if (");
generateCodefromAST(node->left, outFile); // Умова
fprintf(outFile, ") \n");
generateCodefromAST(node->right->left, outFile); // Тіло if
if (node->right->right != NULL)
{ // Else-гілка
    fprintf(outFile, "    else\n");
    generateCodefromAST(node->right->right, outFile);
}
```

Отже опрацювання вузла `assign_node` буде виглядати таким чином:

```
fprintf(outFile, "    ");
generateCodefromAST(node->left, outFile);
fprintf(outFile, " = ");
generateCodefromAST(node->right, outFile);
fprintf(outFile, ";\n");
```

Опрацювання вузлів `or_node`, `and_node`, `cmp_node`, а також вузлів `add_node`, `sub_node`, `mul_node`, `div_node` буде полягати у друці знаку операції і круглих дужок справа і зліва від знаку операції.

```
fprintf(outFile, "(");
generateCodefromAST(node->left, outFile);
fprintf(outFile, " необхідний знак операції ");
generateCodefromAST(node->right, outFile);
fprintf(outFile, ");
```

Опрацювання вузла `not_node` буде виглядати таким чином:

```
fprintf(outFile, "!(");
generateCodefromAST(node->left, outFile);
fprintf(outFile, ");
```

Опрацювання вузла `compound_node` буде виглядати таким чином:

```
fprintf(outFile, "    {\n");
generateCodefromAST(node->left, outFile);
fprintf(outFile, "    }\n");
```

## 4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА

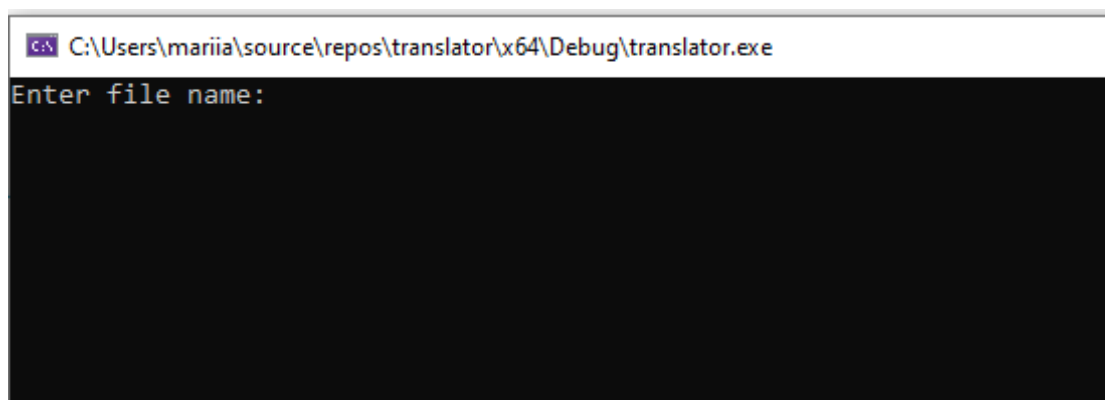
Будь-яке програмне забезпечення необхідно протестувати і налагодити. Після опрацювання синтаксичних і семантичних помилок необхідно переконатися, що розроблене програмне забезпечення функціонує так, як очікувалось.

Для перевірки коректності роботи розробленого транслятора необхідно буде написати тестові задачі на вхідній мові програмування, отримати код на мові програмування C 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:

```

Microsoft Visual Studio Debug Console
Enter file name: test
Lexical analysis completed: 103 tokens. List of tokens in the file test.token
The program is syntax correct.
3 identifier found.
Parsing completed. AST created.
AST has been created and written to test.ast.
C code has been generated and written to test.c.
C code has been generated and written to test_fromAST.c.
C:\Users\mariaa\source\repos\translator\x64\Debug\translator.exe (process 9228) exit
Press any key to close this window . . .

```

Рис. 4.2. Результати роботи розробленого транслятора.

## 4.2. Виявлення лексичних і синтаксичних помилок.

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

Наприклад, у програмі зробимо синтаксичну помилку – у 5-му рядку неправильно вкажемо оператор присвоєння :

```

test
1 start
2 var integer x, y, z;
3 input x;
4 input y;
5 z = x + 8 * y / (x * x - 9);
6
Microsoft Visual Studio Debug Console
8 Enter file name: test
9
10 Lexical analysis completed: 103 tokens. List of tokens in the file test.token
11
12 Syntax error in line 5 : another type of lexeme was expected.
13
14 C:\Users\mariaa\source\repos\translator\x64\Debug\translator.exe (process 14964
15 Press any key to close this window . . .

```

Рис. 4.3. Вивід інформації про синтаксичну помилку.

Зробимо семантичну помилку – не оголосимо змінну “z”:

The screenshot shows a Visual Studio editor window with a file named 'test.cpp'. The code in the editor is as follows:

```

1  start
2  var integer x, y;
3  input x;
4  input y;
5  z := x + 8 * y / (x * x - 9);
6  if x = 0 then
7      if ! (x > 0) & y <> 0 then
8          start
9          x := x * 6;
10         z := z + x / 10;
11         stop
12     else
13         start
14         z := (z - 10) / 100;
15         stop;
16     start
17     output z;
18     stop;
19     if x < 0 then
20         z := z*z
21     else
22         z := 1000;
23     output z;

```

Below the editor, the Microsoft Visual Studio Debug Console is open, displaying the following output:

```

Enter file name: test
Lexical analysis completed: 101 tokens. List of tokens in the file test.token
The program is syntax correct.
In line 5, an undeclared identifier "z"!
In line 10, an undeclared identifier "z"!
In line 10, an undeclared identifier "z"!
In line 14, an undeclared identifier "z"!
In line 14, an undeclared identifier "z"!
In line 17, an undeclared identifier "z"!
In line 20, an undeclared identifier "z"!
In line 20, an undeclared identifier "z"!
In line 20, an undeclared identifier "z"!
In line 22, an undeclared identifier "z"!
In line 23, an undeclared identifier "z"!
2 identifier found.

```

Рис. 4.4. Вивід інформації про семантичну помилку.

### 4.3. Перевірка роботи транслятора за допомогою тестових задач.

#### Тестова програма «Лінійний алгоритм»

1. Ввести два числа  $A$  і  $B$  (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання).

2. Вивести на екран:

$A + B$  (результат операції додавання);

$A - B$  (результат операції віднімання);

$A * B$  (результат операції множення);

$A / B$  (результат операції ділення).

3. Обрахувати значення виразу

$$X = (A - B) * 10 + (A + B) / 10$$

4. Вивести значення  $X$  на екран.

Напишемо програму на вхідній мові програмування:

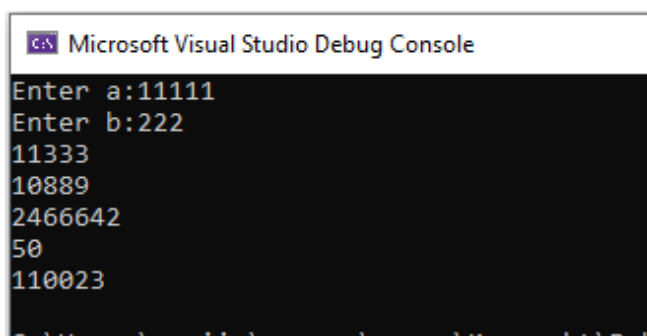
```
start
var integer a, b, x;
input a;
input b;
x := a + b;
output x;
x := a - b;
output x;
x := a * b;
output x;
x := a / b;
output x;
x := (a - b) * 10 + (a + b) / 10;
output x;
stop;
```

В результаті отримаємо файл з програмою на мові програмування C:

```
#include <stdio.h>

int main()
{
    int a;
    int b;
    int x;
    printf("Enter a:");
    scanf_s("%d", &a);
    printf("Enter b:");
    scanf_s("%d", &b);
    x = (a + b);
    printf("%d\n", x);
    x = (a - b);
    printf("%d\n", x);
    x = (a * b);
    printf("%d\n", x);
    x = (a / b);
    printf("%d\n", x);
    x = (((a - b) * 10) + ((a + b) / 10));
    printf("%d\n", x);
    return 0;
}
```

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



```
Microsoft Visual Studio Debug Console
Enter a:11111
Enter b:222
11333
10889
2466642
50
110023
```

Рис. 4.5. Результати виконання тестової задачі 1.



### Тестова програма «Алгоритм з розгалуженням»

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

Використання вкладеного умовного оператора:

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

Використання простого умовного оператора:

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

Напишемо програму на вхідній мові програмування:

```
start
var integer a, b, c, max, x;

input a;
input b;
input c;

if a > b then
    if a > c then
        max := a
    else
        max := c
else
    if b > c then
        max := b
    else
        max := c;
output max;

if a = b & a = c & b = c then
    x := 1
else
    x := 0;
output x;

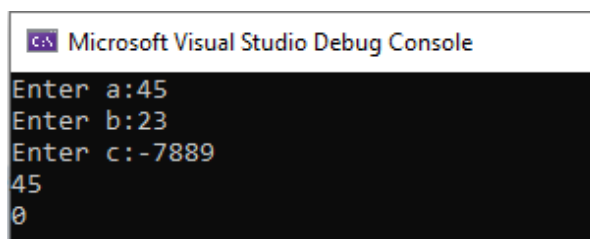
stop
```

В результаті отримаємо файл з програмою на мові програмування C:

```
#include <stdio.h>

int main()
{
    int a;
    int b;
    int c;
    int max;
    int x;
    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 > c)
            max = a;
        else
            max = c;
    else
        if (b > c)
            max = b;
        else
            max = c;
    printf("%d\n", max);
    if ((a == b && a == c) && b == c)
        x = 1;
    else
        x = 0;
    printf("%d\n", x);
    return 0;
}
```

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



```
Microsoft Visual Studio Debug Console
Enter a:45
Enter b:23
Enter c:-7889
45
0
```

Рис. 4.6. Результати виконання тестової задачі 2.

## **ВИСНОВКИ**

## 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 с.
4. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 с.
5. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. – 1038 с.
6. Системне програмування (курсний 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-spring-2010>.

## ДОДАТКИ

### А. Таблиці лексем для тестових прикладів

#### Тестова програма «Лінійний алгоритм»

TOKEN TABLE					
line number	token	value	token code	type of token	
1	start	0	0	StartProgram	
2	var	0	1	Variable	
2	integer	0	2	Integer	
2	a	0	9	Identifier	
2	,	0	26	Comma	
2	b	0	9	Identifier	
2	,	0	26	Comma	
2	x	0	9	Identifier	
2	;	0	25	Semicolon	
3	input	0	4	Input	
3	a	0	9	Identifier	
3	;	0	25	Semicolon	
4	input	0	4	Input	
4	b	0	9	Identifier	
4	;	0	25	Semicolon	
5	x	0	9	Identifier	
5	:=	0	11	Assign	
5	a	0	9	Identifier	
5	+	0	12	Add	
5	b	0	9	Identifier	
5	;	0	25	Semicolon	
6	output	0	5	Output	
6	x	0	9	Identifier	
6	;	0	25	Semicolon	

	7		x		0		9		Identifier	
	7		:=		0		11		Assign	
	7		a		0		9		Identifier	
	7		-		0		13		Sub	
	7		b		0		9		Identifier	
	7		;		0		25		Semicolon	
	8		output		0		5		Output	
	8		x		0		9		Identifier	
	8		;		0		25		Semicolon	
	9		x		0		9		Identifier	
	9		:=		0		11		Assign	
	9		a		0		9		Identifier	
	9		*		0		14		Mul	
	9		b		0		9		Identifier	
	9		;		0		25		Semicolon	
	10		output		0		5		Output	
	10		x		0		9		Identifier	
	10		;		0		25		Semicolon	
	11		x		0		9		Identifier	
	11		:=		0		11		Assign	
	11		a		0		9		Identifier	
	11		/		0		15		Div	
	11		b		0		9		Identifier	
	11		;		0		25		Semicolon	
	12		output		0		5		Output	
	12		x		0		9		Identifier	
	12		;		0		25		Semicolon	
	13		x		0		9		Identifier	
	13		:=		0		11		Assign	
	13		(		0		23		LBracket	

	13		a		0		9		Identifier	
	13		-		0		13		Sub	
	13		b		0		9		Identifier	
	13		)		0		24		RBracket	
	13		*		0		14		Mul	
	13		10		10		10		Number	
	13		+		0		12		Add	
	13		(		0		23		LBracket	
	13		a		0		9		Identifier	
	13		+		0		12		Add	
	13		b		0		9		Identifier	
	13		)		0		24		RBracket	
	13		/		0		15		Div	
	13		10		10		10		Number	
	13		;		0		25		Semicolon	
	14		output		0		5		Output	
	14		x		0		9		Identifier	
	14		;		0		25		Semicolon	
	15		stop		0		3		EndProgram	
	15		;		0		25		Semicolon	

## Тестова програма «Алгоритм з розгалуженням»

TOKEN TABLE					
line number	token	value	token code	type of token	
1	start	0	0	StartProgram	
2	var	0	1	Variable	
2	integer	0	2	Integer	
2	a	0	9	Identifier	
2	,	0	26	Comma	
2	b	0	9	Identifier	
2	,	0	26	Comma	
2	c	0	9	Identifier	
2	,	0	26	Comma	
2	max	0	9	Identifier	
2	,	0	26	Comma	
2	x	0	9	Identifier	
2	;	0	25	Semicolon	
4	input	0	4	Input	
4	a	0	9	Identifier	
4	;	0	25	Semicolon	
5	input	0	4	Input	
5	b	0	9	Identifier	
5	;	0	25	Semicolon	
6	input	0	4	Input	
6	c	0	9	Identifier	
6	;	0	25	Semicolon	
8	if	0	6	If	
8	a	0	9	Identifier	
8	>	0	18	Greate	
8	b	0	9	Identifier	
8	then	0	7	Then	



9	if	0	6	If
9	a	0	9	Identifier
9	>	0	18	Greater
9	c	0	9	Identifier
9	then	0	7	Then
10	max	0	9	Identifier
10	:=	0	11	Assign
10	a	0	9	Identifier
11	else	0	8	Else
12	max	0	9	Identifier
12	:=	0	11	Assign
12	c	0	9	Identifier
13	else	0	8	Else
14	if	0	6	If
14	b	0	9	Identifier
14	>	0	18	Greater
14	c	0	9	Identifier
14	then	0	7	Then
15	max	0	9	Identifier
15	:=	0	11	Assign
15	b	0	9	Identifier
16	else	0	8	Else
17	max	0	9	Identifier
17	:=	0	11	Assign
17	c	0	9	Identifier
17	;	0	25	Semicolon
18	output	0	5	Output
18	max	0	9	Identifier
18	;	0	25	Semicolon

	20		if		0		6		If	
	20		a		0		9		Identifier	
	20		=		0		16		Equality	
	20		b		0		9		Identifier	
	20		&		0		21		And	
	20		a		0		9		Identifier	
	20		=		0		16		Equality	
	20		c		0		9		Identifier	
	20		&		0		21		And	
	20		b		0		9		Identifier	
	20		=		0		16		Equality	
	20		c		0		9		Identifier	
	20		then		0		7		Then	
	21		x		0		9		Identifier	
	21		:=		0		11		Assign	
	21		1		1		10		Number	
	22		else		0		8		Else	
	23		x		0		9		Identifier	
	23		:=		0		11		Assign	
	23		0		0		10		Number	
	23		;		0		25		Semicolon	
	24		output		0		5		Output	
	24		x		0		9		Identifier	
	24		;		0		25		Semicolon	
	26		stop		0		3		EndProgram	

## Б. Абстрактне синтаксичне дерево для тестових прикладів

Тестова програма «Лінійний алгоритм»

```

-- program
-- var
--   -- x
--   -- var
--     -- b
--     -- var
--       -- a
--   -- statement
--     -- statement
--       -- statement
--         -- statement
--           -- statement
--             -- statement
--               -- statement
--                 -- statement
--                   -- statement
--                     -- input
--                       -- a
--                     -- input
--                       -- b
--                   -- :=
--                     -- x
--                     -- +
--                       -- a
--                       -- b
--                   -- output
--                     -- x
--                   -- :=
--                     -- x
--                     -- -
--                       -- a
--                       -- b
--                   -- output
--                     -- x
--                   -- :=
--                     -- x
--                     -- *
--                       -- a
--                       -- b
--                   -- output
--                     -- x
--                   -- :=
--                     -- x
--                     -- /
--                       -- a
--                       -- b
--                   -- output
--                     -- x
--                   -- :=
--                     -- x
--                     -- +
--                       -- *
--                         -- -
--                           -- a

```

```

-- b
-- 10
-- /
-- +
-- a
-- b
-- 10
-- output
-- x

```

### Тестова програма «Алгоритм з розгалуженням»

```

-- program
-- var
--   x
--   var
--     max
--     var
--       c
--       var
--         b
--         var
--           a
-- statement
--   statement
--     statement
--       statement
--         statement
--           statement
--             input
--               a
--             input
--               b
--             input
--               c
--           if
--             >
--               a
--               b
--             then
--               if
--                 >
--                   a
--                   c
--                 then
--                   :=
--                     max
--                     a
--                   :=
--                     max
--                     c
--               if
--                 >
--                   b
--                   c
--                 then
--                   :=
--                     max
--                     b
--                   :=

```



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

Тестова програма «Лінійний алгоритм»

```
#include <stdio.h>

int main()
{
    int a;
    int b;
    int x;
    printf("Enter a:");
    scanf_s("%d", &a);
    printf("Enter b:");
    scanf_s("%d", &b);
    x = (a + b);
    printf("%d\n", x);
    x = (a - b);
    printf("%d\n", x);
    x = (a * b);
    printf("%d\n", x);
    x = (a / b);
    printf("%d\n", x);
    x = (((a - b) * 10) + ((a + b) / 10));
    printf("%d\n", x);
    return 0;
}
```

Тестова програма «Алгоритм з розгалуженням»

```
#include <stdio.h>

int main()
{
    int a;
    int b;
    int c;
    int max;
    int x;
    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 > c)
        max = a;
    else
        max = c;
    else
    if (b > c)
        max = b;
    else
        max = c;
    printf("%d\n", max);
    if ((a == b && a == c) && b == c))
        x = 1;
    else
        x = 0;
    printf("%d\n", x);
    return 0;
}
```