

Міністерство освіти і науки України
Національний університет “Львівська політехніка”



Курсовий проект

З дисципліни «Системне програмування»
на тему: "Розробка системних програмних модулів
та компонент систем програмування."
Розробка транслятора з вхідної мови програмування"
Варіант №18

Виконав: ст. гр. КІ-309
Смаль М. Ю.

Перевірив:
ст. викладач
Козак Н. Б.

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 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: **mainprogram**
 start
 data ...;
 end

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

Оператор присвоєння: **==>**

Оператори:

if [- else]

goto

for – to

for-downto

while

repeat - until

Регістр ідентифікаторів: **Up-Low12**, перший символ **_**

Операції:

- арифметичні: **+** , **-** , ***** , **/** , **%**
- порівняння: **eq** , **ne** , **le** , **ge**
- логічні: **!!** , **&&** , **||**

Тип даних: **integer32**

Коментар: **/* ... */**

Програма на вхідній мові програмування має починатись з ключового слова **mainprogram**, далі має відкриватись блок коду програми з ключового слова **start**, опісля має йти розділ опису змінних **data**. Між розділом **data** і ключовим словом **end** розміщуються оператори програми. Операторів є 9: оператор вводу даних **input**, оператор виводу даних **output**, оператор присвоєння **==>**, умовний оператор **if [- else]**, оператор безумовного переходу **goto** та оператори циклів: **for-to**, **for-downto**, **while**, **repeat-until**. 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 **le** , **ge** , **eq** , **ne** і такі логічні операції **!!** , **&&** , **||**. Ідентифікатори (імена змінних) можуть бути довжиною до 13-х символів, складаються лише з малих та великих латинських літер або цифр та починаються з символу **_**. Тип даних лише один – **integer32**, при оголошенні декількох змінних вони записуються через кому, вкінці опису змінних ставиться символ крапка з комою **;**. Коментарі починаються з **/*** і завершуються ***/**.

Приклади оголошення змінних:

```
data integer32 _Aaaaaaaaaaaaaa;  
data integer32 _Aaaaaaaaaaaaaa, _Bbbbbbbbbbbbbbb, _Cccccccccccccc;  
data integer32 _Aaaaaaaaaaaaaa, _Maaaaaaxxxxxx, _Miiiiiiinnnnnnn;
```

Приклади ідентифікаторів:

```
_Aaaaaaaaaaaaaa, _Maaaaaaxxxxxx, _Miiiiiiinnnnnnn
```

Приклади арифметичних виразів:

_Yuuuuuuuuuuuu + 68
_Aaaaaaaaaaaaaa + _Bbbbbbbbbbbbbbb * _Cccccccccccc + 76
_Cccccccccccc * (_Aaaaaaaaaaaaaa + _Bbbbbbbbbbbbbbb) - 50 /
_Bbbbbbbbbbbbbbb

Приклади логічних виразів:

_Aaaaaaaaaaaaaa ge _Bbbbbbbbbbbbbbb
_Aaaaaaaaaaaaaa le _Bbbbbbbbbbbbbbb && _Aaaaaaaaaaaaaa le
_Cccccccccccc
!! (_Aaaaaaaaaaaaaa le _Cccccccccccc)
_Aaaaaaaaaaaaaa ne _Bbbbb
_Miiiiiiinnnnnnn ge 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

Анотація	Error! Bookmark not defined.
Завдання до курсового проекту	8
Вступ	Error! Bookmark not defined.
1. Огляд методів та способів проектування трансляторів	Error! Bookmark not defined.
2. Формальний опис вхідної мови програмування	Error! Bookmark not defined.
2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура	Error! Bookmark not defined.
2.2. Опис термінальних символів та ключових слів	Error! Bookmark not defined.
3. Розробка транслятора вхідної мови програмування	9
3.1. Вибір технології програмування	Error! Bookmark not defined.
3.2. Проектування таблиць транслятора	Error! Bookmark not defined.
3.3. Розробка лексичного аналізатора	Error! Bookmark not defined.
3.3.1. Розробка блок-схеми алгоритму	Error! Bookmark not defined.
3.3.2. Опис програми реалізації лексичного аналізатора	Error! Bookmark not defined.
3.4. Розробка синтаксичного та семантичного аналізатора	Error! Bookmark not defined.
3.4.1. Опис програми реалізації синтаксичного та семантичного аналізатора ...	Error! Bookmark not defined.
3.4.2. Розробка граф-схеми алгоритму	Error! Bookmark not defined.
3.5. Розробка генератора коду	Error! Bookmark not defined.
3.5.1. Розробка граф-схеми алгоритму	Error! Bookmark not defined.
3.5.2. Опис програми реалізації генератора коду	Error! Bookmark not defined.
4. Опис програми	Error! Bookmark not defined.
4.1. Опис інтерфейсу та інструкція користувачеві	Error! Bookmark not defined.
5. Відлагодження та тестування програми	Error! Bookmark not defined.
5.1. Виявлення лексичних та синтаксичних помилок	49
5.2. Виявлення семантичних помилок	50
5.3. Загальна перевірка коректності роботи транслятора	50
5.4. Тестова програма №1	52
5.5. Тестова програма №2	53
5.6. Тестова програма №3	55
Висновки	Error! Bookmark not defined.
Список використаної літератури	49
Додатки	61

Завдання до курсового проекту

Варіант 18

Завдання на курсовий проект

1. Цільова мова транслятора – асемблер для 32-розрядного процесора.
2. Для отримання виконавчого файлу на виході розробленого транслятора скористатися програмами ml.exe і link.exe.
3. Мова розробки транслятора: C++.
4. Реалізувати оболонку або інтерфейс з командного рядка.
5. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.
6. На виході розробленого транслятора мають створюватись такі файли:
 - *файл з лексемами;*
 - *файл з повідомленнями про помилки (або про їх відсутність);*
 - *файл на мові асемблера;*
 - *об'єктний файл;*
 - *виконавчий файл.*
7. Назва вхідної мови програмування утворюється від першої букви у прізвищі студента та останніх двох цифр номера його варіанту. Саме таке розширення повинні мати текстові файли, написані на цій мові програмування.

В моєму випадку це .s18

Опис вхідної мови програмування:

- Тип даних: integer32
- Блок тіла програми: mainprogram data...; start end
- Оператор вводу: input ()
- Оператор виводу: output ()
- Оператори: if else (C)
 - goto (C)
 - for-to(Паскаль)
 - for-downto-do (Паскаль)
 - while (Бейсік)
 - repeat-until (Паскаль)
- Регістр ключових слів: Low
- Регістр ідентифікаторів: Up-Low12 перший символ _
- Операції арифметичні: +, -, *, /, %
- Операції порівняння: eq, ne, ge, le
- Операції логічні: !, &&, !!
- Коментар: /*... */
- Ідентифікатори змінних, числові константи
- Оператор присвоєння: <==

ВСТУП

У сучасному світі програмне забезпечення відіграє ключову роль у бізнесі, освіті та медицині. Розробка системних модулів і компонентів систем програмування є важливою частиною цієї галузі. Транслятори, як один із основних інструментів, дозволяють перетворювати код з однієї мови програмування в іншу, забезпечуючи виконання програм на різних платформах.

Цей курсовий проєкт присвячений розробці транслятора вхідної мови програмування, створеної для цього завдання. Його мета — розробити програму, яка перетворює код цієї мови у код на C, реалізуючи лексичний, синтаксичний і семантичний аналізатори та генератор коду.

Актуальність

Попит на ефективні інструменти розробки зростає. Транслятори оптимізують процес створення й тестування програм, забезпечують сумісність із різними платформами, що важливо у швидкому розвитку технологій.

Мета проєкту

Розробити транслятор, який перетворює код вхідної мови у мову C. Для цього буде реалізовано лексичний, синтаксичний і семантичний аналізатори, а також генератор коду. Результатом стане програма, здатна створювати виконувані файли із текстових.

1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ

1.1. Методи побудови трансляторів

1.1.1. Лексичний аналіз

На цьому етапі текст програми розбивається на окремі елементи (лексеми): ключові слова, ідентифікатори, літерали, роздільники та оператори. Для реалізації лексичного аналізу зазвичай застосовують:

Регулярні вирази — для опису шаблонів лексем.

Скінченні автомати — для автоматизованого розпізнавання лексем.

1.2. Синтаксичний аналіз

Синтаксичний аналізатор перевіряє структуру програми відповідно до граматики мови та формує дерево розбору. Основні методи:

Зверху вниз (top-down parsing):

Рекурсивний спуск (Recursive Descent Parsing).

LL-аналізатори (табличні методи).

Знизу вгору (bottom-up parsing):

LR-аналізатори (типи LR(0), SLR, LALR, Canonical LR).

1.3. Семантичний аналіз

Цей етап спрямований на перевірку логічної коректності програми: відповідність типів даних, області видимості змінних, коректність викликів функцій. Для семантичного аналізу використовують:

Атрибутні граматики (Attribute Grammars).

Таблиці символів — структури для зберігання інформації про ідентифікатори.

1.4. Генерація коду

Генерація коду полягає у створенні проміжного або машинного коду. Методи включають:

Однопрохідну генерацію — для простих мов.

Двопрохідну генерацію — спочатку створюється проміжний код, який оптимізується перед перетворенням у машинний.

Багатопохідну генерацію — для складних мов із додатковими етапами оптимізації.

1.5. Оптимізація коду

Мета оптимізації — зменшити розмір і підвищити ефективність виконання програми. Популярні техніки:

- Локальні оптимізації — у межах одного блоку коду.
- Глобальні оптимізації — враховують взаємодію між кількома блоками.
- Машинно-незалежна оптимізація — спрощення обчислень.
- Машинно-залежна оптимізація — призначення регістрів, налаштування інструкцій.

2. Класифікація трансляторів

Транслятори можна класифікувати за різними критеріями:

- За способом роботи: компілятори, інтерпретатори, асемблери.
- За кількістю проходів: однопрохідні, багатопохідні.
- За цільовим кодом: машинний код, проміжний код (LLVM IR, байт-код).

3. Основні підходи до проєктування

Модульний підхід: розбиття транслятора на окремі компоненти (лексичний і синтаксичний аналізатори, генератор коду тощо).

Фазовий підхід: чітке розділення фаз із обміном результатами між ними.

Однопрохідний дизайн: підходить для невеликих або простих мов.

Багатопохідний дизайн: забезпечує глибоку оптимізацію та підтримує складні мови.

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:

```
program = "mainprogram", "start", {"data", variable_declaration, ";"},
{statement}, "end";

variable_declaration = "integer32", variable_list;

variable_list = identifier, {"", identifier};

identifier = "_", up, low, low, low, low, low, low, low, low, low, low, low,
low;

up_low = up | low | digit;

up = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" |
"M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" |
"Z";

low = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" |
"m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" |
"z" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

statement = input_statement | output_statement | assign_statement |
if_else_statement | goto_statement | label_point | for_statement |
while_statement | repeat_until_statement | compound_statement;

input_statement = "input", identifier;

output_statement = "output", arithmetic_expression;

arithmetic_expression = low_priority_expression {low_priority_operator,
low_priority_expression};

low_priority_operator = "+" | "-";

low_priority_expression = middle_priority_expression {middle_priority_operator,
middle_priority_expression};

middle_priority_operator = "*" | "/" | "%";

middle_priority_expression = identifier | number | "(", arithmetic_expression,
");";

number = ["-"], (nonzero_digit, {digit} | "0") ;

nonzero_digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

assign_statement = arithmetic_expression, "==>", identifier;

if_else_statement = "if", "(", logical_expression, ")", statement, [ "else",
statement];

logical_expression = and_expression {or_operator, and_expression};

or_operator = "||";

and_expression = comparison {and_operator, and_expression};
```

```
and_operator = "&&";
comparison = comparison_expression | [not_operator] "(", logical_expression,
");";
not_operator = "!!";
comparison_expression = arithmetic_expression comparison_operator
arithmetic_expression;
comparison_operator = "eq" | "ne" | "le" | "ge";
goto_statement = "GOTO", identifier;
label_point = identifier, ":";
for_to_statement = "for", assign_statement, "to" | "downto",
arithmetic_expression, "do", statement;
statement_in_while = statement | ("CONTINUE", "WHILE") | ("EXIT", "WHILE");
while_statement = "while", logical_expression, {statement_in_while}, "end",
"WHILE";
repeat_until_statement = "repeat", {statement}, "until", "(",
logical_expression, ")";
compoundStatement = "start", {statement}, "end";
```

Опис термінальних символів та ключових слів.

Визначаємо термінальні символи і ключові слова:

- **mainprogram** - початок програми
- **start** - початок блоку
- **data** - оголошення змінних
- **end** - кінець програми
- **integer32** - тип даних
- **input** - оператор вводу
- **output** - оператор виводу
- **if, else** - умовний оператор
- **got** - оператор безумного переходу
- **for, to, do** - оператор циклу for
- **for, downto, do** - оператор циклу for
- **while** - оператор циклу while
- **repeat, UNTIL** - оператор циклу repeat
- **==>** - оператор присвоєння
- **+** - додавання
- **-** - віднімання
- ***** - множення
- **/** - ділення
- ***** - остача від ділення
- **ge** - більше
- **le** - менше
- **eq** - рівність
- **ne** - нерівність
- **!!** - заперечення
- **&&** - логічне І
- **||** - логічне АБО
- **;** - кінець оператора
- **,** - розділювач змінних
- **(** - відкрита дужка
- **)** - закрита дужка
- **/*** - початок коментаря
- ***/** - кінець коментаря
- **a...z** - маленькі латинські букви
- **0...9** - цифри
- символи табуляції, переходу на новий рядок, пробіл

3. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ 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.

Використання таблиць значно полегшує створення трансляторів, а тому створимо рігання інформації про лексеми:

```
enum TypeOfToken {  
    StartProgram, // mainprogram  
    StartBlock,   // start  
    Variable,     // data  
    Type,         // integer32  
    EndBlock,     // end  
    Input,        // input  
    Output,       // output  
    If,           // if  
    Else,         // else  
    Goto,         // goto  
    For,          // for  
    To,           // to  
    Downto,       // downto  
    Do,           // do  
    While,        // while  
    Continue,  
    Exit,
```


End, // end

Repeat, // repet

Until, // until

Identifier, // Identifier

Number, // number

Float, // float (incorrect)

Assign, // ==>

Add, // +

Sub, // -

Mul, // *

Div, // /

Mod, // %

Equality, // eq

NotEquality, // ne

Greater, // ge

Less, // le

Not, // !!

And, // &&

Or, // ||

LBracket, // (

RBracket, //)

Semicolon, // ;

Colon, // :

```

    Comma,      // ,
    Unknown_
};

struct Token {
    char name[16];    // ім'я лексеми
    int value;        // значення (для констант)
    int line;         // номер рядка
    enum TypeOfToken type; // тип лексеми
};

```

```

struct id {
    char name[16];
    //unsigned int pos; //for labels
};

```

У програмі будемо зберігати таблицю лексем і таблицю та кількість лексем, ідентифікаторів та міток:

```

struct Token* TokenTable; // Таблиця лексем
unsigned int TokensNum;    // Кількість лексем

struct id* idTable;        // Таблиця ідентифікаторів
unsigned int idNum;        // кількість ідентифікаторів

struct id* labelTable;     // Таблиця міток
unsigned int labelNum;     // кількість міток

```

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 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 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 (mainprogram, start, data, end, integer32, input, output, if, else, for, to, downto, do, while, end, repeat, until, div, %)
- Ідентифікатори (починається з символу _, далі велика або маленька літера або цифра, максимум 6 символи)
- Числові константи (ціле число)
- Оператор присвоєння (==>)
- Знаки операції (+, -, *, <, >, <=, >=, !=, &&, ||)
- Розділювачі (., ;)
- Дужки ((,))
- Невідома лексема (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 чи символу `_`, 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 6-х символів, інакше це невизначена лексема. Переходимо до стану **Finish**.

У стані **Digit** читаємо по одному символи з файлу і виділяємо ланцюжок символів, який починається з крапки, мінуса або ж цифри, далі ж йдуть лише цифри або крапки, 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 – невідома, читаємо наступний символ і переходимо до стану **Finish**.

У стані **Comment** читаємо символи, поки не зустрінеться 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 кілька лексем, які вимагають ще читання наступного символу з файлу - це оператор присвоєння “==>” і операцій “!=”, “!!”, “||”, “&&”, “<<”, “>>”. 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.

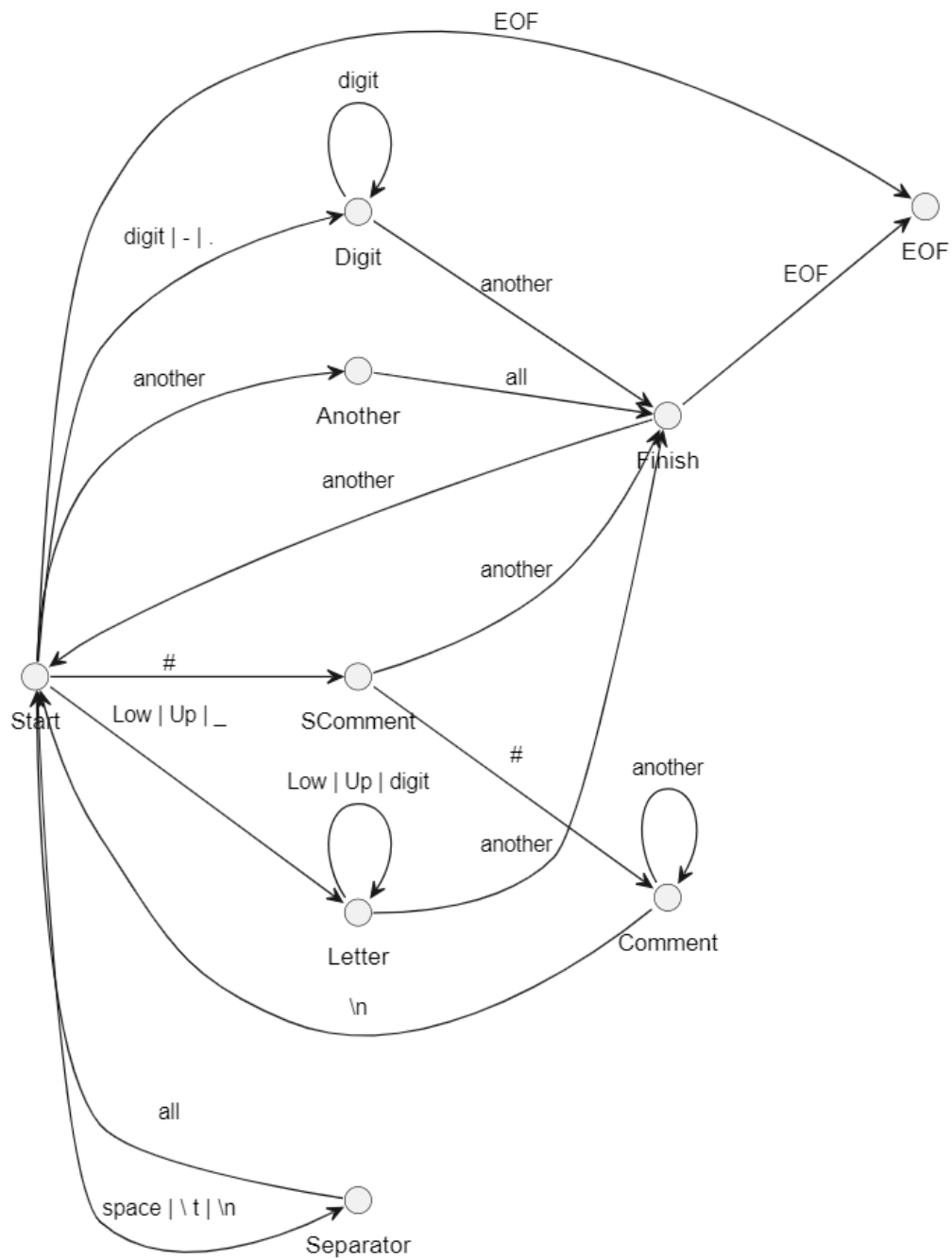


Рис. 3.1. Граф-схема алгоритму роботи лексичного аналізатора.

3.3.2. Опис програми реалізації лексичного аналізатора.

В даному курсовому проекті 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:

```
enum States {  
    Start,    // початковий стан  
    Finish,   // фінальний стан  
    Letter,   // опрацювання слів (ключові слова та ідентифікаторів)  
    Digit,    // опрацювання цифр  
    Separator, // опрацювання роздільників  
    Another,  // опрацювання інших символів  
    EndOfFile, // кінець файлу  
    SComment, // початок коментаря  
    Comment   // ігнорування коментаря  
};
```

Напишемо функцію, яка реалізує лексичний аналіз:

```
unsigned int getTokens(FILE* F);
```

І функції, які друкують список лексем:

```
void printTokens(void);
```

```
void fprintfTokens(FILE* F);
```

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.

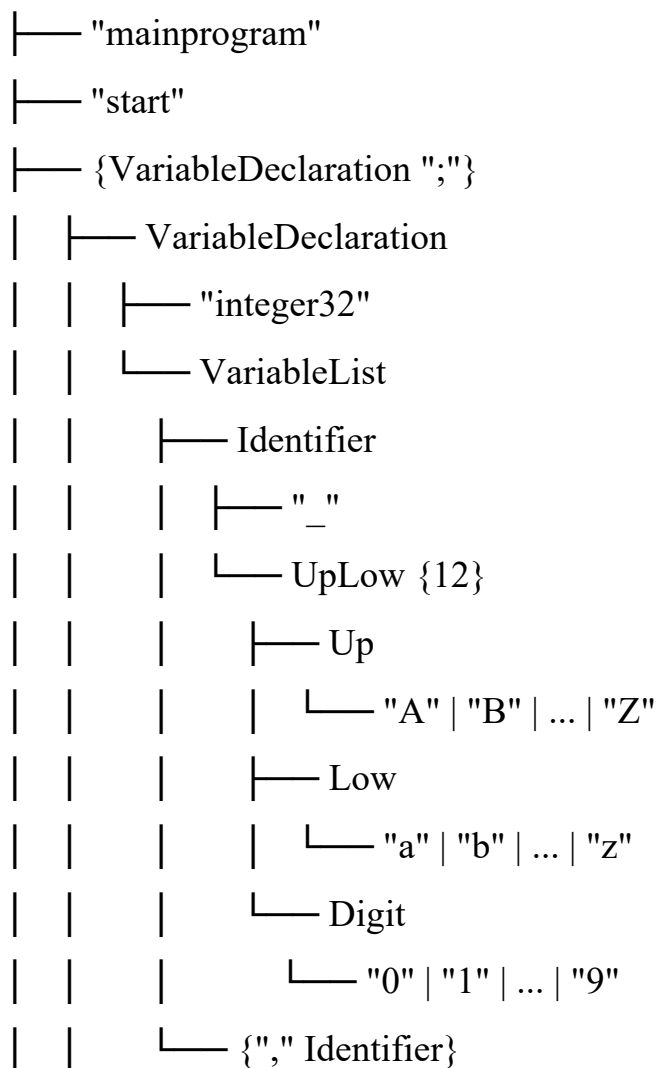
На виході синтаксичного аналізатора отримуємо дерево граматичного розбору і таблиці ідентифікаторів та типів, які є входом для наступного перегляду компілятора.

Семантичний аналіз перевіряє змістовну коректність програми, враховуючи правила мови програмування. Він визначає, чи відповідають операції типам даних та чи виконуються всі обмеження мови. Результатом є анотація дерева розбору додатковою інформацією для генерації коду.

3.4.1. Розробка дерева граматичного розбору.

Схема дерева розбору виглядає наступним чином:

Program




```

└── {Statement}
|   ├── Statement
|   |   ├── InputStatement
|   |   |   ├── "input"
|   |   |   └── Identifier
|   |   ├── OutputStatement
|   |   |   ├── "output"
|   |   |   └── ArithmeticExpression
|   |   |       ├── LowPriorityExpression
|   |   |       |   ├── MiddlePriorityExpression
|   |   |       |   |   ├── Identifier
|   |   |       |   |   ├── Number
|   |   |       |   |   ├── ["-"]
|   |   |       |   |   └── Digit {5}
|   |   |       |   └── "(" ArithmeticExpression ")"
|   |   |       └── {MiddlePriorityOperator MiddlePriorityExpression}
|   |   |           └── {LowPriorityOperator LowPriorityExpression}
|   |   └── AssignStatement
|   |       ├── ArithmeticExpression
|   |       └── "==" Identifier
|   └── IfElseStatement
|       ├── "if"
|       ├── "(" LogicalExpression ")"
|       |   ├── AndExpression
|       |   |   ├── Comparison
|       |   |   |   ├── ComparisonExpression
|       |   |   |   |   ├── ArithmeticExpression
|       |   |   |   |   ├── ComparisonOperator
|       |   |   |   └── ArithmeticExpression

```

						└─ [NotOperator] "(" LogicalExpression ")"
						└─ {AndOperator AndExpression}
						└─ {OrOperator AndExpression}
						└─ Statement
						└─ ["else" Statement]
						└─ GotoStatement
						└─ "goto"
						└─ Identifier
						└─ LabelPoint
						└─ Identifier
						└─ ":"
						└─ ForToStatement
						└─ "for"
						└─ AssignStatement
						└─ "to" "downto"
						└─ ArithmeticExpression
						└─ "do"
						└─ Statement
						└─ WhileStatement
						└─ "while"
						└─ LogicalExpression
						└─ {Statement}
						└─ "end" "while"
						└─ RepeatUntilStatement
						└─ "repeat"
						└─ {Statement}
						└─ "until" "(" LogicalExpression ")"
						└─ CompoundStatement
						└─ "start"

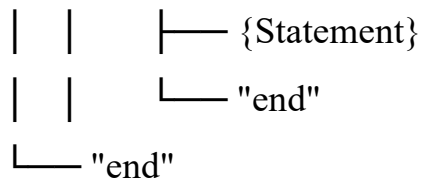


Рис. 3.2. Дерево граматичного розбору.

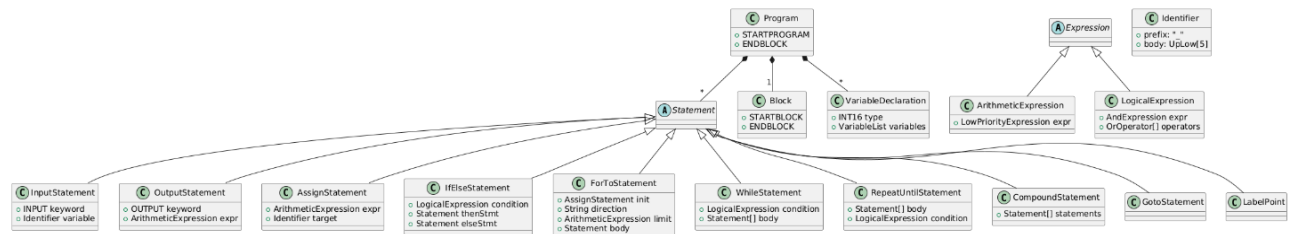


Рис. 3.3. Дерево граматичного розбору.

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:

```
// program ::= "mainprogram", "start", { "data", variable_declaration, ";" }, {
statement, ";" }, "end";

void program();

// programBody ::= { statement, ";" };

void programBody();

// variable_declaration ::= "integer32", variable_list;

void variableDeclaration();

// variable_list ::= identifier, { ",", identifier };

void variableList();

// statement ::= input_statement | output_statement | assign_statement |
if_else_statement | goto_statement | label_point | for_statement | while_statement |
repeat_until_statement | compound_statement;
```

```

void statement();
// input_statement ::= "input", identifier;
void inputStatement();
// output_statement ::= "output", arithmetic_expression;
void outputStatement();
// arithmetic_expression ::= low_priority_expression { low_priority_operator,
low_priority_expression };
void arithmeticExpression();
// low_priority_expression ::= middle_priority_expression {
middle_priority_operator, middle_priority_expression };
void lowPriorityExpression();
// low_priority_operator ::= "+" | "-";
//void lowPriorityOperator();

// middle_priority_expression ::= identifier | number | "(", arithmetic_expression,
")";
void middlePriorityExpression();
// middle_priority_operator ::= "*" | "/" | "%";
//void middlePriorityOperator();

// assign_statement ::= arithmetic_expression, "==>", identifier;
void assignStatement();
// if_else_statement ::= "if", "(", logical_expression, ")", statement, [";", "else",
statement];
void ifStatement();
// logical_expression ::= and_expression { or_operator, and_expression };
void logicalExpression();

// or_operator ::= "||";
//void orOperator();

```

```

// and_expression ::= comparison{ and_operator, and_expression };
void andExpression();

// and_operator ::= "&&";
// void andOperator();

// comparison ::= comparison_expression | [not_operator] "(", logical_expression,
// ");";
void comparison();

// not_operator ::= "!!";
// void notOperator();

// comparison_expression ::= arithmetic_expression comparison_operator
// arithmetic_expression;
void comparisonExpression();

// comparison_operator ::= "eq" | "ne" | "le" | "ge";
// void comparisonOperator();

// goto_statement ::= "goto", identifier;
void gotoStatement();

// label_point ::= identifier, ":";
void labelPoint();

// for_to_statement ::= "for", assign_statement, "to" | "downto",
// arithmetic_expression, "do", statement;
void forStatement();

// while_statement ::= "while", logical_expression, { statement_in_while, ";" },
// "end";
void whileStatement();

```

```

// statement_in_while ::= statement | "CONTINUE WHILE" | "EXIT WHILE";
void statementInWhile();

// repeat_until_statement ::= "repeat", { statement, ";" }, "until", "(",
logical_expression, ")";
void repeatStatement();

// compoundStatement ::= "start", { statement, ";" }, "end";
void compoundStatement();

```

Блок-схема алгоритму роботи синтаксичного аналізатора виглядатиме наступним чином:

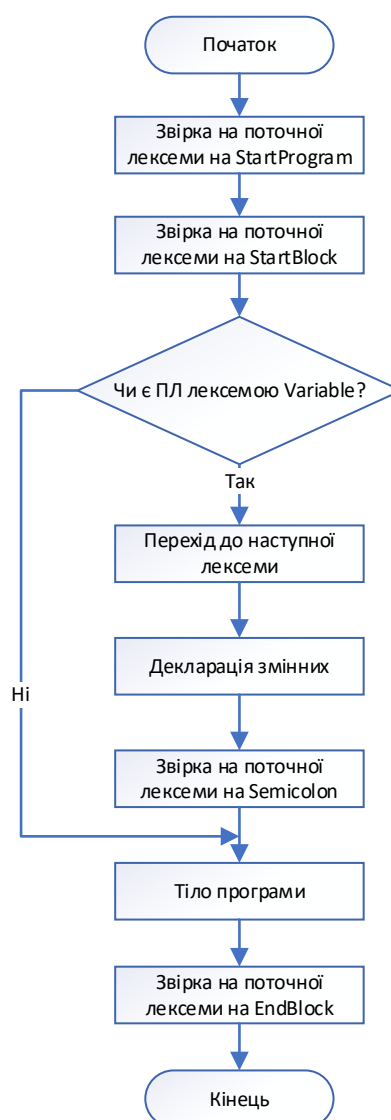


Рис. 3.3. Блок-схема алгоритму роботи синтаксичного аналізатора.

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

```
void match(enum TokenType expectedType);
```

яка перевіряє, чи поточна лексема збігається з очікуваною. Блок-схема алгоритму функції `variable_declaration()`, яка перевіряє чи правильно описані змінні виглядає наступним чином:

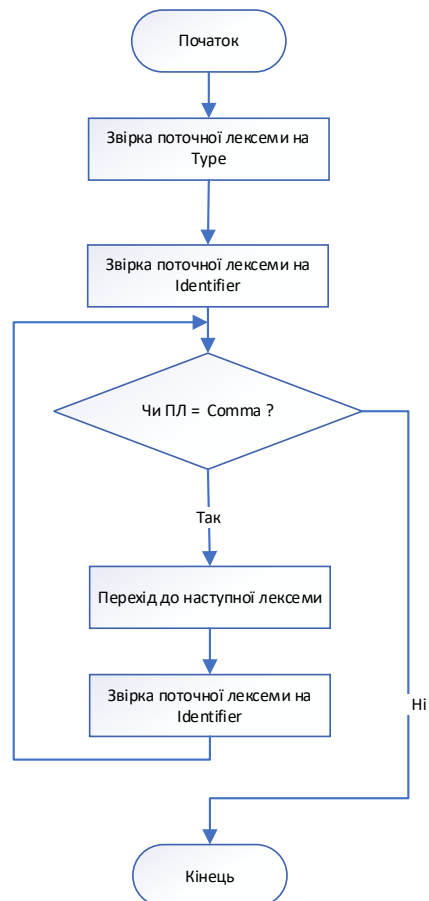


Рис. 3.4. Блок-схема алгоритму роботи функції `variableDeclaration()`.

Блок-схема алгоритму функції `programBody()`, 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 та НЛ, що позначає наступну лексему. У процесі перегляду таблиці лексем, після аналізу поточної лексеми, необхідно переміщатися на наступну лексему.



Рис. 3.5. Блок-схема алгоритму роботи функції *programBody()*.

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");
```

```
    fprintf(errorFile, "\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 викликає функції, що отримують списки ідентифікаторів та міток:

```
void Semantic() {
```

```
    idNum = IdIdentification(idTable, TokenTable, TokensNum);
```

```
    labelNum = LabelIdentification(labelTable, TokenTable, TokensNum);
```

```
    printf("\nThe program is semantic correct.\n");
```

```
    printf("\n%d labels found\n", labelNum);
```

```
    fprintf(errorFile, "\nThe program is semantic correct.\n");
```

```
    fprintf(errorFile, "\n%d labels found\n", labelNum);
```

```
    printIdentifiers(labelNum, labelTable);
```

```

fprintIdentifiers(errorFile, labelNum, labelTable);

printf("\n%d identifiers found\n", idNum);

fprintf(errorFile, "\n%d identifiers found\n", idNum);

printIdentifiers(idNum, idTable);

fprintIdentifiers(errorFile, idNum, idTable);

}

```

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:

```

      Program
      /    \
var    statement

```

Оголошення змінних:

var
/
Id var
/
Id null

Тіло програми:

Statement
/
Statement Оператор
/
statement Оператор

Оператор вводу:

Input
/
Id null

Оператор виводу:

Output
/
Id null

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

```
      Output
    /      \
  Add  null
    /      \
  Id   num
```

Умовний оператор (IF() оператор;):

```
      If
    /      \
  Умова   оператор
```

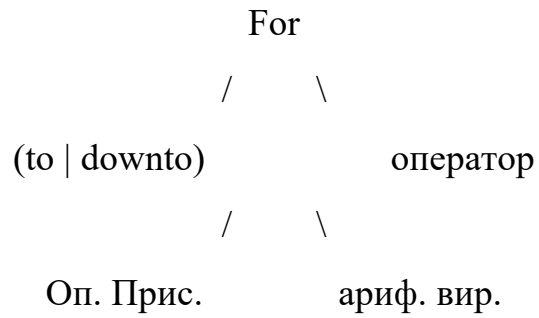
Умовний оператор (IF() оператор1; else оператор2;):

```
      If
    /      \
  Умова   else
    /      \
Оператор1 оператор2
```

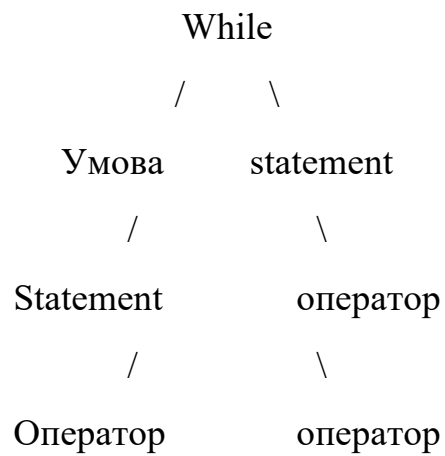
Оператор безумовного переходу:

```
      Goto
    /      \
  Id   null
```

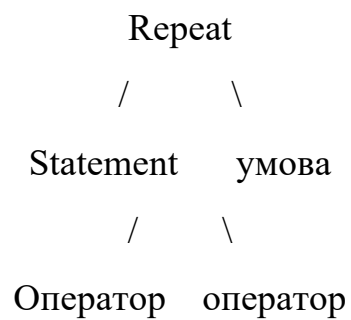
Оператор циклу for:



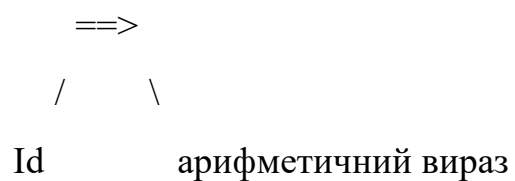
Оператор циклу while:



Оператор циклу repeat:



Оператор присвоєння:



Арифметичний вираз:

(+ або -)

/ \

Id id

Доданок:

(* , / або %)

/ \

МНОЖНИК МНОЖНИК

Множник:

фактор

/ \

id або number або (арифм. вираз) null

Складений оператор:

compound

/ \

statement null

Генератор коду буде обходити створене дерево і, маючи усю 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.

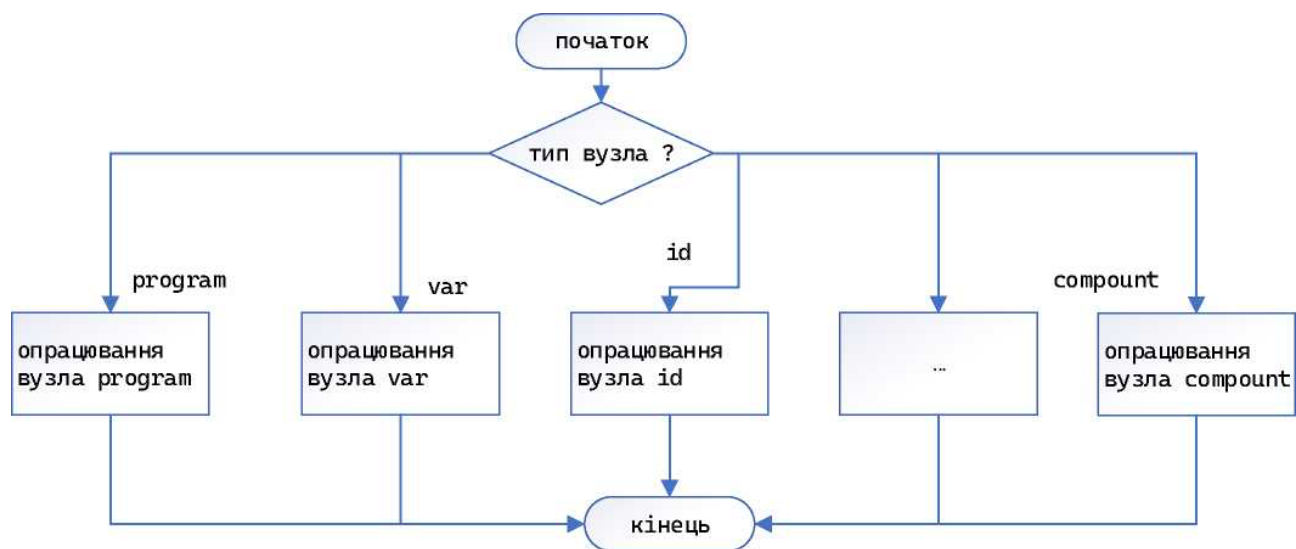


Рис. 3.6. Блок-схема алгоритму роботи генератора коду.

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`).

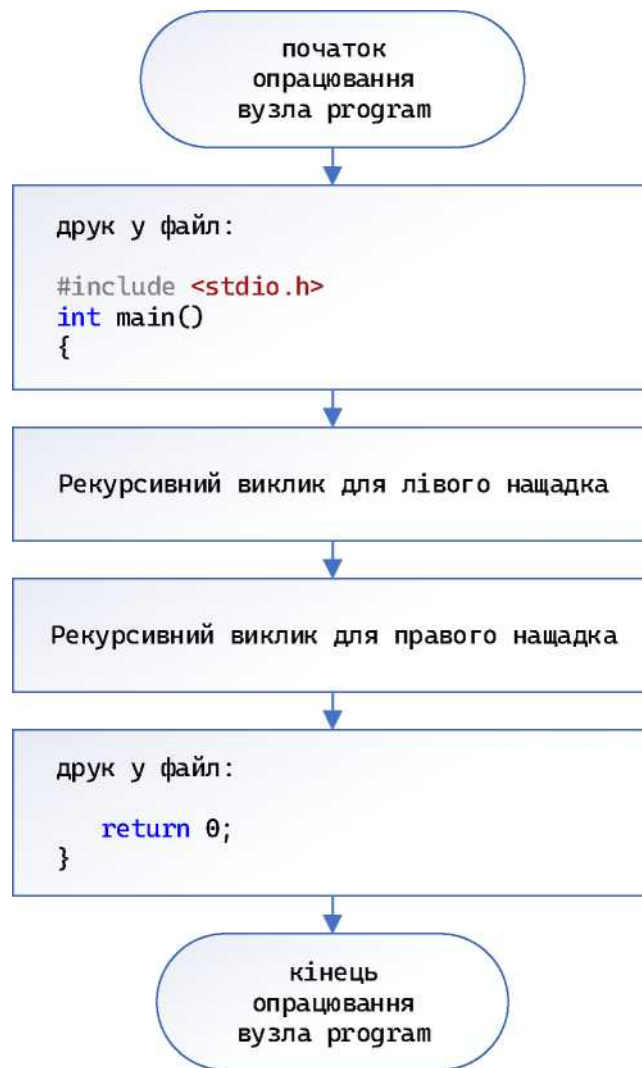


Рис. 3.7. Блок-схема алгоритму опрацювання вузла *program*.

3.5.2. Опис програми реалізації генератора коду.

Генерувати вихідний код будемо з абстрактного синтаксичного дерева.

Створимо таку структуру даних для зберігання вузлів дерева:

```
enum TypeOfNode {  
    program_node,  
    id_node,  
    var_node,  
    statement_node,  
    input_node,  
    output_node,
```



```
    add_node,  
    sub_node,  
    mul_node,  
    div_node,  
    mod_node,  
    number_node,  
    assign_node,  
    if_node,  
    else_node,  
    or_node,  
    and_node,  
    not_node,  
    eq_node,  
    neq_node,  
    gr_node,  
    ls_node,  
    goto_node,  
    label_node,  
    for_node,  
    to_node,  
    downto_node,  
    while_node,  
    continue_node,  
    exit_node,  
    repeat_node,  
    compound_node  
};  
  
struct astNode {  
    enum TypeOfNode type;
```

```
char name[16];  
struct astNode* left;  
struct astNode* right;  
};
```

Функція створення вузла дерева:

// функція створення вузла AST

```
struct astNode* createNode(enum TypeOfNode type, const char* name, struct  
astNode* left, struct astNode* right);
```

Функція створення абстрактного синтаксичного дерева реалізована методом рекурсивного спуску:

// функція синтаксичного аналізу і створення абстрактного синтаксичного дерева

```
struct astNode* astParser() {  
    pos = 0;  
    struct astNode* tree = program();  
  
    printf("AST created.\n");  
  
    return tree;  
}
```

Також напишемо функції для друку абстрактного синтаксичного дерева:

// функція для друку AST у вигляді дерева на екран

```
void printAST(struct astNode* node, int level)
```

// функція для друку AST у вигляді дерева у файл

```
void fPrintAST(FILE* outFile, struct astNode* node, int level);
```

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

// Рекурсивна функція для генерації коду з AST

```

void codegen(FILE* outFile, struct astNode* node){
if (node == NULL) return;
switch (node->type) {
case ...
case ...
default: {
    exit(1);
    printf("Undescribed node type: %d\n", node->type);
    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");
    fprintf(outFile, "int main() {\n");
    codegen(outFile, node->left); // for declaration
    fprintf(outFile, "\n");
    codegen(outFile, node->right); // for statements
    fprintf(outFile, "return 0;\n}\n");
    break;
}

```

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

```

{
    fprintf(outFile, "int ");
    codegen(outFile, node->left);
}

```

```

        fprintf(outFile, ";\n");
        codegen(outFile, node->right);
        break;
    }

```

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

```

{
    fprintf(outFile, "%s", node->name);
    break;
}

```

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

```

{
    codegen(outFile, node->left);
    codegen(outFile, node->right);
    break;
}

```

Опрацювання вузла `input_node` буде виглядати таким чином:

```

{
    fprintf(outFile, "printf(\"Enter ");
    codegen(outFile, node->left);
    fprintf(outFile, ": \");\n");
    fprintf(outFile, "scanf(\"%d\", &");
    codegen(outFile, node->left);
    fprintf(outFile, ");\n");
    break;
}

```

```
}
```

А вузла `output_node` буде виглядати таким чином:

```
{  
    fprintf(outFile, "printf(\"%%d\\n\", ");  
    codegen(outFile, node->left);  
    fprintf(outFile, ");\\n");  
    break;  
}
```

Опрацювання вузла `if_node` буде виглядати таким чином:

```
{  
    fprintf(outFile, "if (");  
    codegen(outFile, node->left);  
    fprintf(outFile, ") ");  
    codegen (outFile, node->right);  
    break;  
}
```

Опрацювання вузла `else_node` буде виглядати таким чином:

```
{  
    codegen(outFile, node->left);  
    fprintf(outFile, "else ");  
    codegen(outFile, node->right);  
    break;  
}
```

Отже опрацювання вузла `assign_node` буде виглядати таким чином:

```
{
```

```

        codegen(outFile, node->left);
        fprintf(outFile, " = ");
        codegen(outFile, node->right);
        fprintf(outFile, ";\n");
        break;
    }

```

Отже опрацювання вузла `not_node` буде виглядати таким чином:

```

{
    fprintf(outFile, "!(");
    codegen(outFile, node->left);
    fprintf(outFile, ")");
    break;
}

```

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

```

{
    fprintf(outFile, "(");
    codegen(outFile, node->left);
    fprintf(outFile, <необхідний знак операції>);
    codegen(outFile, node->right);
    fprintf(outFile, ")");
    break;
}

```

Опрацювання вузла `for_node` буде виглядати таким чином:

```

{

```

```

    fprintf(outFile, "for(\n");
    codegen(outFile, node->left);
    fprintf(outFile, "\n) ");
    codegen(outFile, node->right);
    break;
}

```

Опрацювання вузла to_node буде виглядати таким чином:

```

{
    codegen(outFile, node->left);
    codegen(outFile, node->left->left);
    fprintf(outFile, " <= ");
    codegen(outFile, node->right);
    fprintf(outFile, ";\n++");
    codegen(outFile, node->left->left);
    break;

}

```

Опрацювання вузла downto_node буде виглядати таким чином:

```

{
    codegen(outFile, node->left);
    codegen(outFile, node->left->left);
    fprintf(outFile, " >= ");
    codegen(outFile, node->right);
    fprintf(outFile, ";\n--");
    codegen(outFile, node->left->left);
    break;

}

```

Опрацювання вузла `while_node` буде виглядати таким чином:

```
{  
    fprintf(outFile, "while");  
    codegen(outFile, node->left);  
    fprintf(outFile, ") {\n");  
    codegen(outFile, node->right);  
    fprintf(outFile, "}\n");  
    break;  
}
```

Опрацювання вузла `repeat_node` буде виглядати таким чином:

```
{  
    fprintf(outFile, "do {\n");  
    codegen(outFile, node->left);  
    fprintf(outFile, "} while");  
    codegen(outFile, node->right);  
    fprintf(outFile, ");\n");  
    break;  
}
```

Опрацювання вузла `compound_node` буде виглядати таким чином:

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


Тестування програмного забезпечення є важливим етапом розробки продукту. На цьому етапі знаходяться помилки допущені на попередніх етапах. Цей етап дозволяє покращити певні характеристики продукту, наприклад – інтерфейс. Дає можливість знайти та вподальшому виправити слабкі сторони, якщо вони є.

Відлагодження даної програми здійснюється за допомогою набору кількох програм, які відповідають заданій граматиці. Та перевірка коректності коду, що генерується, коректності знаходження помилок та розбивки на лексеми.

4. Виявлення лексичних та синтаксичних помилок

Виявлення лексичних помилок відбувається на стадії лексичного аналізу. Під час розбиття вхідної програми на окремі лексеми відбувається перевірка чи відповідає вхідна лексема граматиці. Якщо ця лексема є в граматиці то вона ідентифікується і в таблиці лексем визначається. У випадку неспівпадіння лексемі присвоюється тип "невпізнаної лексеми". Повідомлення про такі помилки можна побачити лише після виконання процедури перевірки таблиці лексем, яка знаходиться в файлі.

Виявлення синтаксичних помилок відбувається на стадії перевірки програми на коректність окремо від синтаксичного аналізу. При цьому перевіряється окремо кожне твердження яке може бути або виразом, або оператором (циклу, вводу/виводу), або оголошенням, та перевіряється структура програми в цілому.

Приклад виявлення:

Текст програми з помилками

```
/* Prog1 */  
mainprogram  
start  
  
data integer32 _ Aaaaaaaaaa,_ Bbbbbbbbbbbbbb,_ Xxxxxxxxxxxxxx,_ Yyyyyyyyyyyyyy;  
input _ Aaaaaaaaaa;  
input _ Bbbbbbbbbbbbbb;  
output _ Aaaaaaaaaa + _ Bbbbbbbbbbbbbb;  
output _ Aaaaaaaaaa - _ Bbbbbbbbbbbbbb;  
output _ Aaaaaaaaaa * _ Bbbbbbbbbbbbbb;  
output _ Aaaaaaaaaa / _ Bbbbbbbbbbbbbb;  
output _ Aaaaaaaaaa % _ Bbbbbbbbbbbbbb;
```

```

_Xxxxxxxxxxxxx<==(_Aaaaaaaaaaaaa - _Bbbbbbbbbbbbbb) * 10 + (_Aaaaaaaaaaaaa +
_Bbbbbbbbbbbbbb) / 10;
_Yyyyyyyyyyyyy<==_Xxxxxxxxxxxxx + (_Xxxxxxxxxxxxx % 10)
output _Xxxxxxxxxxxxx
output2 _Yyyyyyyyyyyyy
end

```

Текст файлу з повідомленнями про помилки

Lexical Error: line 5, lexem _Aaaaaaaaaaaaa is Unknown

Lexical Error: line 17, lexem output2 is Unknown

Syntax error in line 5 : another type of lexeme was expected.

Syntax error: type Unknown

Expected Type: Identifier

4.1 Виявлення семантичних помилок

Суттю виявлення семантичних помилок є перевірка числових констант на відповідність типу integer32, тобто знаковому цілому числу з відповідним діапазоном значень і перевірку на коректність використання змінних integer32 у цілочисельних і логічних виразах.

4.2 Загальна перевірка коректності роботи транслятора

Для того щоб здійснити перевірку коректності роботи транслятора необхідно завантажити коректну до заданої вхідної мови програму.

Текст коректної програми

```

/* Prog1 */
mainprogram
start

data integer32 _Aaaaaaaaaaaaa, _Bbbbbbbbbbbbbb, _Xxxxxxxxxxxxx, _Yyyyyyyyyyyyy;
input _Aaaaaaaaaaaaa
input _Bbbbbbbbbbbbbb
output _Aaaaaaaaaaaaa + _Bbbbbbbbbbbbbb
output _Aaaaaaaaaaaaa - _Bbbbbbbbbbbbbb
output _Aaaaaaaaaaaaa * _Bbbbbbbbbbbbbb

```

```

output _Aaaaaaaaaaaaaa / _Bbbbbbbbbbbbbbb
output _Aaaaaaaaaaaaaa % _Bbbbbbbbbbbbbbb

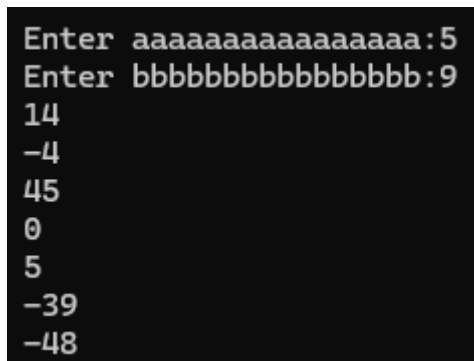
_Xxxxxxxxxxxxxx<==(_Aaaaaaaaaaaaaa - _Bbbbbbbbbbbbbbb) * 10 + (_Aaaaaaaaaaaaaa +
_Bbbbbbbbbbbbbbb) / 10
_Yyyyyyyyyyyyyy<==_Xxxxxxxxxxxxxx + (_Xxxxxxxxxxxxxx % 10)
output _Xxxxxxxxxxxxxx
output _Yyyyyyyyyyyyyy
end

```

Оскільки дана програма відповідає граматиці то результати виконання лексичного, синтаксичного аналізів, а також генератора коду будуть позитивними.

В результаті буде отримано с файл, який є результатом виконання трансляції з заданої вхідної мови на мову C даної програми (його вміст наведений в Додатку А).

Після виконання компіляції даного файлу на виході отримаєм наступний результат роботи програми:



```

Enter aaaaaaaaaaaaaa:5
Enter bbbbbbbbbbbbbbbb:9
14
-4
45
0
5
-39
-48

```

Рис. 5.1 Результат виконання коректної програми

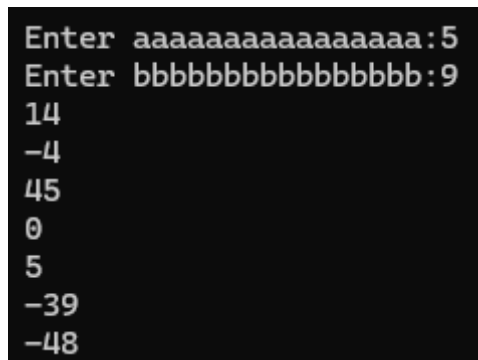
При перевірці отриманого результату, можна зробити висновок про правильність роботи програми, а отже і про правильність роботи транслятора.

4.3 Тестова програма №1

Текст програми

```
/* Prog1 */  
mainprogram  
start  
  
data integer32 _Aaaaaaaaaaaa, _Bbbbbbbbbbbbbb, _XXXXXXXXXXXX, _Yyyyyyyyyyyyyy;  
input _Aaaaaaaaaaaa  
input _Bbbbbbbbbbbbbb  
output _Aaaaaaaaaaaa + _Bbbbbbbbbbbbbb  
output _Aaaaaaaaaaaa - _Bbbbbbbbbbbbbb  
output _Aaaaaaaaaaaa * _Bbbbbbbbbbbbbb  
output _Aaaaaaaaaaaa / _Bbbbbbbbbbbbbb  
output _Aaaaaaaaaaaa % _Bbbbbbbbbbbbbb  
  
_XXXXXXXXXXXX<==(_Aaaaaaaaaaaa - _Bbbbbbbbbbbbbb) * 10 + (_Aaaaaaaaaaaa +  
_Bbbbbbbbbbbbbb) / 10  
_Yyyyyyyyyyyyyy<==_XXXXXXXXXXXX + (_XXXXXXXXXXXX % 10)  
output _XXXXXXXXXXXX  
output _Yyyyyyyyyyyyyy  
end
```

Результат виконання



```
Enter aaaaaaaaaaaaaa:5  
Enter bbbbbbbbbbbbbb:9  
14  
-4  
45  
0  
5  
-39  
-48
```

Рис. 5.2 Результат виконання тестової програми №1

4.4 Тестова програма №2

Текст програми

```
/* Prog2 */
mainprogram
start
data integer32 _Aaaaaaaaaaaa, _Bbbbbbbbbbbbbb, _Cccccccccccc, _Tttttttttt;

input _Aaaaaaaaaaaa
input _Bbbbbbbbbbbbbb
input _Cccccccccccc

if (_Aaaaaaaaaaaa ge _Bbbbbbbbbbbbbb) goto Abigger;

else

    _Tttttttttt <== _Aaaaaaaaaaaa;

Outofib:

if (_Cccccccccccc ge _Bbbbbbbbbbbbbb && _Cccccccccccc ge _Aaaaaaaaaaaa) goto Outofic;
else goto Outofif;

Abigger:
    _Tttttttttt <== _Bbbbbbbbbbbbbb
    goto Outofib
Outofic:
    _Tttttttttt <== _Cccccccccccc
    goto Outofif

Outofif:
    output _Tttttttttt
```

```
if((_Aaaaaaaaaaaa eq _Bbbbbbbbbbbb) && (_Aaaaaaaaaaaa eq _Cccccccccccc) &&  
(_Bbbbbbbbbbbb eq _Cccccccccccc))
```

```
    output 1;
```

```
else
```

```
    output 0;
```

```
if((_Aaaaaaaaaaaa le 0) || (_Bbbbbbbbbbbb le 0) || (_Cccccccccccc le 0))
```

```
    output -1;
```

```
else
```

```
    output 0;
```

```
if(!_Aaaaaaaaaaaa le (_Bbbbbbbbbbbb + _Cccccccccccc)))
```

```
    output 10
```

```
;
```

```
else
```

```
    output 0
```

```
;
```

```
end
```

Результат виконання

```

Enter aaaaaaaaaaaaaaaaaa:5
Enter bbbbbbbbbbbbbbbbbb:9
Enter cccccccccccccccc:-10
9
0
-1
10

```

Рис. 5.3 Результат виконання тестової програми №2

4.5 Тестова програма №3

Текст програми

```

/* Prog3 */
mainprogram
start
data integer32
_Aaaaaaaaaaaaaa,_Aaaaaaaaaaaa2,_Bbbbbbbbbbbbbb,_Xxxxxxxxxxxxxx,_Cccccccccccc,_Cccccccc
ccca;
input _Aaaaaaaaaaaaaa
input _Bbbbbbbbbbbbbb

for _Aaaaaaaaaaaa2<==_Aaaaaaaaaaaaaa to _Bbbbbbbbbbbbbb do
    output _Aaaaaaaaaaaaaa2 * _Aaaaaaaaaaaaaa2 ;

for _Aaaaaaaaaaaa2<==_Bbbbbbbbbbbbbb to _Aaaaaaaaaaaaaa do
    output _Aaaaaaaaaaaaaa2 * _Aaaaaaaaaaaaaa2;

_Xxxxxxxxxxxxxx<==0
_Cccccccccccc<==0

while _Cccccccccccc le _Aaaaaaaaaaaaaa
    _Ccccccccccca<==0
    while _Ccccccccccca le _Bbbbbbbbbbbbbb

```

```

        _Xxxxxxxxxxxxxx<==_Xxxxxxxxxxxxxx + 1
        _Ccccccccccca<==_Ccccccccccca + 1
    end while

```

```

    _Cccccccccccc<==_Cccccccccccc + 1

```

```

end while

```

```

output _Xxxxxxxxxxxxxx

```

```

_Xxxxxxxxxxxxxx<==0

```

```

_Cccccccccccc<==1

```

```

repeat

```

```

    _Ccccccccccca<==1

```

```

    repeat

```

```

        _Xxxxxxxxxxxxxx<==_Xxxxxxxxxxxxxx + 1

```

```

        _Ccccccccccca<==_Ccccccccccca + 1

```

```

    until(!_Ccccccccccca ge _Bbbbbbbbbbbbbb))

```

```

    _Cccccccccccc<==_Cccccccccccc + 1

```

```

until(!_Cccccccccccc ge _Aaaaaaaaaaaaaa))

```

```

output _Xxxxxxxxxxxxxx

```

```

end

```

Результат виконання


```
Enter aaaaaaaaaaaaaaaaaa:5  
Enter bbbbbbbbbbbbbbbbbb:9  
25  
36  
49  
64  
81  
45  
45
```

Рис. 5.4 Результат виконання тестової програми №3

ВИСНОВКИ

У рамках курсового проекту було створено транслятор з вхідної мови програмування, який виконує такі завдання:

1. Лексичний аналіз:

- Алгоритм лексичного аналізу розбиває текст на лексеми, формуючи таблицю з інформацією про їх тип, значення та рядок.
- Лексичний аналізатор працює за принципом скінченного автомату, розпізнаючи ключові слова, ідентифікатори, константи, оператори та розділювачі.

2. Синтаксичний і семантичний аналіз:

- Синтаксичний аналізатор перевіряє структуру програми відповідно до граматики, формуючи дерево розбору й таблиці ідентифікаторів та типів.
- Семантичний аналізатор перевіряє логічну коректність програми: відповідність типів даних, області видимості змінних і коректність викликів функцій.

3. Генерація коду:

- Генератор коду перетворює абстрактне дерево у вихідний код на мові C, обходячи дерево та створюючи код для кожного вузла.

4. Тестування:

- Проведено тестування на різних програмах (лінійні, з розгалуженням, циклічні), виявлено та усунуто лексичні, синтаксичні й семантичні помилки.

- Транслятор генерує правильний код на основі вхідних програм.

Переваги проекту:

- Послідовна реалізація всіх етапів трансляції.
- Модульна структура, яка спрощує розширення та модифікацію.
- Ретельне тестування підтвердило надійність роботи.

Недоліки проекту:

- Підтримка лише базових конструкцій мови.
- Відсутність графічного інтерфейсу, що знижує зручність використання.

Проект демонструє базову функціональність транслятора й є основою для подальшого розвитку.

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

Додаток А (Таблиці лексем для тестових програм)

TOKEN TABLE				
line number	token	value	token code	type of token
2	mainprogram	0	0	MainProgram
3	start	0	3	StartProgram
4	data	0	2	Variable
4	integer32	0	4	Integer
4	_Aaaaaaaaaaaaa	0	23	Identifier
4	,	0	44	Comma
4	_Bbbbbbbbbbbbbbb	0	23	Identifier
4	,	0	44	Comma
4	_XXXXXXXXXXXXX	0	23	Identifier
4	,	0	44	Comma
4	_Yyyyyyyyyyyyyyy	0	23	Identifier
4	;	0	42	Semicolon
5	input	0	6	Input
5	_Aaaaaaaaaaaaa	0	23	Identifier
6	input	0	6	Input
6	_Bbbbbbbbbbbbbbb	0	23	Identifier
7	output	0	7	Output
7	_Aaaaaaaaaaaaa	0	23	Identifier

7	+	0	26	Add
7	_Bbbbbbbbbbbbbbb	0	23	Identifier
8	output	0	7	Output
8	_Aaaaaaaaaaaaaa	0	23	Identifier
8	-	0	27	Sub
8	_Bbbbbbbbbbbbbbb	0	23	Identifier
9	output	0	7	Output
9	_Aaaaaaaaaaaaaa	0	23	Identifier
9	*	0	28	Mul
9	_Bbbbbbbbbbbbbbb	0	23	Identifier
10	output	0	7	Output
10	_Aaaaaaaaaaaaaa	0	23	Identifier
10	/	0	29	Div
10	_Bbbbbbbbbbbbbbb	0	23	Identifier
11	output	0	7	Output
11	_Aaaaaaaaaaaaaa	0	23	Identifier
11	%	0	30	Mod
11	_Bbbbbbbbbbbbbbb	0	23	Identifier
13	_XXXXXXXXXXXXX	0	23	Identifier
13	<==	0	25	Assign
13	(0	40	LBraket
13	_Aaaaaaaaaaaaaa	0	23	Identifier
13	-	0	27	Sub

	13	_Bbbbbbbbbbbbbb	0	23 Identifier

	13)	0	41 RBracket

	13	*	0	28 Mul

	13	10	10	24 Number

	13	+	0	26 Add

	13	(0	40 LBracket

	13	_Aaaaaaaaaaaaaa	0	23 Identifier

	13	+	0	26 Add

	13	_Bbbbbbbbbbbbbb	0	23 Identifier

	13)	0	41 RBracket

	13	/	0	29 Div

	13	10	10	24 Number

	14	_Yyyyyyyyyyyyyy	0	23 Identifier

	14	<==	0	25 Assign

	14	_XXXXXXXXXXXXX	0	23 Identifier

	14	+	0	26 Add

	14	_XXXXXXXXXXXXX	0	23 Identifier

	14	%	0	30 Mod

	14	10	10	24 Number

	15	output	0	7 Output

	15	_XXXXXXXXXXXXX	0	23 Identifier

	16	output	0	7 Output

16		_Yyyyyyyyyyyyy		0		23		Identifier	
----	--	----------------	--	---	--	----	--	------------	--

17		end		0		5		EndProgram	
----	--	-----	--	---	--	---	--	------------	--

TOKEN TABLE									
-------------	--	--	--	--	--	--	--	--	--

line number		token		value		token code		type of token	
-------------	--	-------	--	-------	--	------------	--	---------------	--

2		mainprogram		0		0		MainProgram	
---	--	-------------	--	---	--	---	--	-------------	--

3		start		0		3		StartProgram	
---	--	-------	--	---	--	---	--	--------------	--

4		data		0		2		Variable	
---	--	------	--	---	--	---	--	----------	--

4		integer32		0		4		Integer	
---	--	-----------	--	---	--	---	--	---------	--

4		_Aaaaaaaaaaaaa		0		23		Identifier	
---	--	----------------	--	---	--	----	--	------------	--

4		,		0		44		Comma	
---	--	---	--	---	--	----	--	-------	--

4		_Bbbbbbbbbbbbbb		0		23		Identifier	
---	--	-----------------	--	---	--	----	--	------------	--

4		,		0		44		Comma	
---	--	---	--	---	--	----	--	-------	--

4		_Ccccccccccccc		0		23		Identifier	
---	--	----------------	--	---	--	----	--	------------	--

4		,		0		44		Comma	
---	--	---	--	---	--	----	--	-------	--

4		_Ttttttttttt		0		23		Identifier	
---	--	--------------	--	---	--	----	--	------------	--

4		;		0		42		Semicolon	
---	--	---	--	---	--	----	--	-----------	--

6		input		0		6		Input	
---	--	-------	--	---	--	---	--	-------	--

6		_Aaaaaaaaaaaaa		0		23		Identifier	
---	--	----------------	--	---	--	----	--	------------	--

7		input		0		6		Input	
---	--	-------	--	---	--	---	--	-------	--

7		_Bbbbbbbbbbbbbb		0		23		Identifier	
---	--	-----------------	--	---	--	----	--	------------	--

8		input		0		6		Input	
---	--	-------	--	---	--	---	--	-------	--

8		_Ccccccccccccc		0		23		Identifier	
---	--	----------------	--	---	--	----	--	------------	--

10		if		0		8		If	
----	--	----	--	---	--	---	--	----	--

	10		(0		40		LBracket	
	10		_Aaaaaaaaaaaaa		0		23		Identifier	
	10		ge		0		35		Greater	
	10		_Bbbbbbbbbbbbbb		0		23		Identifier	
	10)		0		41		RBracket	
	10		goto		0		11		Goto	
	10		Abigger		0		23		Identifier	
	10		;		0		42		Semicolon	
	12		else		0		10		Else	
	14		_Ttttttttttt		0		23		Identifier	
	14		<=		0		25		Assign	
	14		_Aaaaaaaaaaaaa		0		23		Identifier	
	16		Outofib		0		12		Label	
	18		if		0		8		If	
	18		(0		40		LBracket	
	18		_Ccccccccccccc		0		23		Identifier	
	18		ge		0		35		Greater	
	18		_Bbbbbbbbbbbbbb		0		23		Identifier	
	18		&&		0		38		And	
	18		_Ccccccccccccc		0		23		Identifier	
	18		ge		0		35		Greater	
	18		_Aaaaaaaaaaaaa		0		23		Identifier	

	18)		0		41		RBracket	

	18		goto		0		11		Goto	

	18		Outofic		0		23		Identifier	

	18		;		0		42		Semicolon	

	19		else		0		10		Else	

	19		goto		0		11		Goto	

	19		Outofif		0		23		Identifier	

	19		;		0		42		Semicolon	

	21		Abigger		0		12		Label	

	22		_Tttttttttt		0		23		Identifier	

	22		<==		0		25		Assign	

	22		_Bbbbbbbbbbbb		0		23		Identifier	

	23		goto		0		11		Goto	

	23		Outofib		0		23		Identifier	

	24		Outofic		0		12		Label	

	25		_Tttttttttt		0		23		Identifier	

	25		<==		0		25		Assign	

	25		_Cccccccccccc		0		23		Identifier	

	26		goto		0		11		Goto	

	26		Outofif		0		23		Identifier	

	28		Outofif		0		12		Label	

	29		output		0		7		Output	

	29		_Tttttttttt		0		23		Identifier	

	32	if	0	8 If

	32	(0	40 LBraket

	32	(0	40 LBraket

	32	_Aaaaaaaaaaaaa	0	23 Identifier

	32	eq	0	33 Equality

	32	_Bbbbbbbbbbbbbb	0	23 Identifier

	32)	0	41 RBraket

	32	&&	0	38 And

	32	(0	40 LBraket

	32	_Aaaaaaaaaaaaa	0	23 Identifier

	32	eq	0	33 Equality

	32	_Ccccccccccccc	0	23 Identifier

	32)	0	41 RBraket

	32	&&	0	38 And

	32	(0	40 LBraket

	32	_Bbbbbbbbbbbbbb	0	23 Identifier

	32	eq	0	33 Equality

	32	_Ccccccccccccc	0	23 Identifier

	32)	0	41 RBraket

	32)	0	41 RBraket

	34	output	0	7 Output

	34	1	1	24 Number

35	;	0	42	Semicolon
37	else	0	10	Else
39	output	0	7	Output
39	0	0	24	Number
40	;	0	42	Semicolon
42	if	0	8	If
42	(0	40	LBracket
42	(0	40	LBracket
42	_Aaaaaaaaaaaaa	0	23	Identifier
42	le	0	36	Less
42	0	0	24	Number
42)	0	41	RBracket
42		0	39	Or
42	(0	40	LBracket
42	_Bbbbbbbbbbbbb	0	23	Identifier
42	le	0	36	Less
42	0	0	24	Number
42)	0	41	RBracket
42		0	39	Or
42	(0	40	LBracket
42	_Ccccccccccccc	0	23	Identifier
42	le	0	36	Less
42	0	0	24	Number

	42)		0		41		RBracket	
	42)		0		41		RBracket	
	44		output		0		7		Output	
	44		-		0		27		Sub	
	44		1		1		24		Number	
	45		;		0		42		Semicolon	
	47		else		0		10		Else	
	49		output		0		7		Output	
	49		0		0		24		Number	
	49		;		0		42		Semicolon	
	51		if		0		8		If	
	51		(0		40		LBracket	
	51		!!		0		37		Not	
	51		(0		40		LBracket	
	51		_Aaaaaaaaaaaaa		0		23		Identifier	
	51		le		0		36		Less	
	51		(0		40		LBracket	
	51		_Bbbbbbbbbbbbbb		0		23		Identifier	
	51		+		0		26		Add	
	51		_Ccccccccccccc		0		23		Identifier	
	51)		0		41		RBracket	
	51)		0		41		RBracket	

	51)		0		41		RBracket	

	53		output		0		7		Output	

	53		10		10		24		Number	

	53		;		0		42		Semicolon	

	54		else		0		10		Else	

	55		output		0		7		Output	

	55		0		0		24		Number	

	55		;		0		42		Semicolon	

	56		end		0		5		EndProgram	

TOKEN TABLE				
line number	token	value	token code	type of token
2	mainprogram	0	0	MainProgram
3	start	0	3	StartProgram
4	data	0	2	Variable
4	integer32	0	4	Integer
4	_Aaaaaaaaaaaaaa	0	23	Identifier
4	,	0	44	Comma
4	_Aaaaaaaaaaaaaa2	0	23	Identifier
4	,	0	44	Comma
4	_Bbbbbbbbbbbbbbbb	0	23	Identifier
4	,	0	44	Comma
4	_XXXXXXXXXXXXXX	0	23	Identifier
4	,	0	44	Comma

4		_Cccccccccccc		0		23		Identifier	
4		,		0		44		Comma	
4		_Ccccccccccca		0		23		Identifier	
4		;		0		42		Semicolon	
5		input		0		6		Input	
5		_Aaaaaaaaaaaa		0		23		Identifier	
6		input		0		6		Input	
6		_Bbbbbbbbbbbb		0		23		Identifier	
8		for		0		13		For	
8		_Aaaaaaaaaaaa2		0		23		Identifier	
8		<=		0		25		Assign	
8		_Aaaaaaaaaaaa		0		23		Identifier	
8		to		0		14		To	
8		_Bbbbbbbbbbbb		0		23		Identifier	
8		do		0		16		Do	
9		output		0		7		Output	
9		_Aaaaaaaaaaaa2		0		23		Identifier	
9		*		0		28		Mul	
9		_Aaaaaaaaaaaa2		0		23		Identifier	
9		;		0		42		Semicolon	
11		for		0		13		For	
11		_Aaaaaaaaaaaa2		0		23		Identifier	

11	<==	0	25	Assign
11	_Bbbbbbbbbbbbbb	0	23	Identifier
11	to	0	14	To
11	_Aaaaaaaaaaaa	0	23	Identifier
11	do	0	16	Do
12	output	0	7	Output
12	_Aaaaaaaaaaaa2	0	23	Identifier
12	*	0	28	Mul
12	_Aaaaaaaaaaaa2	0	23	Identifier
12	;	0	42	Semicolon
14	_XXXXXXXXXXXXX	0	23	Identifier
14	<==	0	25	Assign
14	0	0	24	Number
15	_Cccccccccccc	0	23	Identifier
15	<==	0	25	Assign
15	0	0	24	Number
17	while	0	17	While
17	_Cccccccccccc	0	23	Identifier
17	le	0	36	Less
17	_Aaaaaaaaaaaa	0	23	Identifier
18	start	0	3	StartProgram
19	_Ccccccccccca	0	23	Identifier
19	<==	0	25	Assign

	19		0		0		24		Number	
	20		while		0		17		While	
	20		_Ccccccccccca		0		23		Identifier	
	20		le		0		36		Less	
	20		_Bbbbbbbbbbbb		0		23		Identifier	
	21		_XXXXXXXXXXXX		0		23		Identifier	
	21		<==		0		25		Assign	
	21		_XXXXXXXXXXXX		0		23		Identifier	
	21		+		0		26		Add	
	21		1		1		24		Number	
	21		;		0		42		Semicolon	
	22		_Ccccccccccca		0		23		Identifier	
	22		<==		0		25		Assign	
	22		_Ccccccccccca		0		23		Identifier	
	22		+		0		26		Add	
	22		1		1		24		Number	
	22		;		0		42		Semicolon	
	23		end		0		20		End	
	23		while		0		17		While	
	25		_Cccccccccccc		0		23		Identifier	
	25		<==		0		25		Assign	
	25		_Cccccccccccc		0		23		Identifier	

	25		+		0		26		Add	

	25		1		1		24		Number	

	26		end		0		20		End	

	26		while		0		17		While	

	27		output		0		7		Output	

	27		_XXXXXXXXXXXXX		0		23		Identifier	

	28		_XXXXXXXXXXXXX		0		23		Identifier	

	28		<==		0		25		Assign	

	28		0		0		24		Number	

	29		_Cccccccccccc		0		23		Identifier	

	29		<==		0		25		Assign	

	29		1		1		24		Number	

	30		repeat		0		21		Repeat	

	31		_Ccccccccccca		0		23		Identifier	

	31		<==		0		25		Assign	

	31		1		1		24		Number	

	32		repeat		0		21		Repeat	

	33		_XXXXXXXXXXXXX		0		23		Identifier	

	33		<==		0		25		Assign	

	33		_XXXXXXXXXXXXX		0		23		Identifier	

	33		+		0		26		Add	

	33		1		1		24		Number	

	34		_Ccccccccccca		0		23		Identifier	

	34		<==	
			0	
			25	Assign

	34		_Ccccccccccca	
			0	
			23	Identifier

	34		+	
			0	
			26	Add

	34		1	
			1	
			24	Number

	35		until	
			0	
			22	Until

	35		(
			0	
			40	LBracket

	35		!!	
			0	
			37	Not

	35		(
			0	
			40	LBracket

	35		_Ccccccccccca	
			0	
			23	Identifier

	35		ge	
			0	
			35	Greate

	35		_Bbbbbbbbbbbbbb	
			0	
			23	Identifier

	35)	
			0	
			41	RBracket

	35)	
			0	
			41	RBracket

	36		_Cccccccccccc	
			0	
			23	Identifier

	36		<==	
			0	
			25	Assign

	36		_Cccccccccccc	
			0	
			23	Identifier

	36		+	
			0	
			26	Add

	36		1	
			1	
			24	Number

	37		until	
			0	
			22	Until

	37		(
			0	
			40	LBracket

	37		!!	
			0	
			37	Not

	37		(
			0	
			40	LBracket

	37		_Cccccccccccc		0		23		Identifier	

	37		ge		0		35		Greate	

	37		_Aaaaaaaaaaaaa		0		23		Identifier	

	37)		0		41		RBracket	

	37)		0		41		RBracket	

	38		output		0		7		Output	

	38		_XXXXXXXXXXXXX		0		23		Identifier	

	39		end		0		5		EndProgram	

```

|-- program
| |-- var
| | |-- _Yyyyyyyyyyyyy
| | |-- var
| | | |-- _XXXXXXXXXXXXX
| | | |-- var
| | | | |-- _Bbbbbbbbbbbbb
| | | | |-- var
| | | | | |-- _Aaaaaaaaaaaaa
| |-- statement
| | |-- statement
| | | |-- statement
| | | | |-- statement
| | | | | |-- statement
| | | | | | |-- statement
| | | | | | | |-- statement
| | | | | | | | |-- statement
| | | | | | | | | |-- statement
| | | | | | | | | | |-- statement
| | | | | | | | | | | |-- input
| | | | | | | | | | | |-- _Aaaaaaaaaaaaa
| | | | | | | | | | | |-- input
| | | | | | | | | | | |-- _Bbbbbbbbbbbbb
| | | | | | | | | | | |-- output
| | | | | | | | | | | |-- +
| | | | | | | | | | | |-- _Aaaaaaaaaaaaa
| | | | | | | | | | | |-- _Bbbbbbbbbbbbb
| | | | | | | | | | | |-- output

```



```

| | | |-- var
| | | | |-- _Aaaaaaaaaaaaaa
| |-- statement
| | |-- statement
| | | |-- statement
| | | | |-- statement
| | | | | |-- statement
| | | | | | |-- statement
| | | | | | | |-- statement
| | | | | | | | |-- input
| | | | | | | | | |-- _Aaaaaaaaaaaaaa
| | | | | | | | | |-- input
| | | | | | | | | |-- _Bbbbbbbbbbbbbb
| | | | | | | | | |-- input
| | | | | | | | | |-- _Cccccccccccc
| | | | | | | | |-- if
| | | | | | | | | |-- ge
| | | | | | | | | | |-- _Aaaaaaaaaaaaaa
| | | | | | | | | | |-- _Bbbbbbbbbbbbbb
| | | | | | | | | |-- branches
| | | | | | | | | |-- compound
| | | | | | | | | |-- if
| | | | | | | | | | |-- ge
| | | | | | | | | | | |-- _Aaaaaaaaaaaaaa
| | | | | | | | | | | |-- _Cccccccccccc
| | | | | | | | | | | |-- branches
| | | | | | | | | | | |-- compound
| | | | | | | | | | | |-- goto
| | | | | | | | | | | | |-- _Temporalllll
| | | | | | | | | | | | |-- compound
| | | | | | | | | | | | | |-- statement
| | | | | | | | | | | | | |-- statement
| | | | | | | | | | | | | | |-- statement
| | | | | | | | | | | | | | | |-- output
| | | | | | | | | | | | | | | |-- _Cccccccccccc
| | | | | | | | | | | | | | | |-- goto
| | | | | | | | | | | | | | | |-- _Outgotooooooo
| | | | | | | | | | | | | | | |-- _Temporalllll
| | | | | | | | | | | | | | | |-- output
| | | | | | | | | | | | | | | |-- _Aaaaaaaaaaaaaa
| | | | | | | | | | | | | | | |-- goto
| | | | | | | | | | | | | | | |-- _Outgotooooooo
| | | | | | | | |-- if

```

```
| | | | | | | |-- le
| | | | | | | |-- _Bbbbbbbbbbbbbb
| | | | | | | |-- _Cccccccccccc
| | | | | | | |-- branches
| | | | | | | |-- compound
| | | | | | | |-- output
| | | | | | | | | |-- _Cccccccccccc
| | | | | | | |-- compound
| | | | | | | |-- output
| | | | | | | | | |-- _Bbbbbbbbbbbbbb
| | | | | |-- _Outgotooooooooo
| | | | |-- if
| | | | |-- |
| | | | | |-- eq
| | | | | | |-- _Aaaaaaaaaaaaaa
| | | | | | |-- _Bbbbbbbbbbbbbb
| | | | | |-- &
| | | | | | |-- eq
| | | | | | | |-- _Aaaaaaaaaaaaaa
| | | | | | | |-- _Cccccccccccc
| | | | | | |-- eq
| | | | | | | |-- _Bbbbbbbbbbbbbb
| | | | | | | |-- _Cccccccccccc
| | | | | |-- branches
| | | | | | |-- compound
| | | | | | |-- output
| | | | | | | |-- 1
| | | | | | |-- compound
| | | | | | | |-- output
| | | | | | | |-- 0
| | | |-- if
| | | | |-- |
| | | | | |-- |
| | | | | | |-- le
| | | | | | | |-- _Aaaaaaaaaaaaaa
| | | | | | | |-- 0
| | | | | | |-- le
| | | | | | | |-- _Bbbbbbbbbbbbbb
| | | | | | | |-- 0
| | | | | |-- le
| | | | | | |-- _Cccccccccccc
| | | | | | |-- 0
| | | | |-- branches
| | | | | |-- compound
| | | | | | |-- output
```



```

|-- input
|-- _Aaaaaaaaaaaaaa
|-- input
|-- _Bbbbbbbbbbbbbb
|-- for-to
|-- <==
|-- _Aaaaaaaaaaaaa2
|-- _Aaaaaaaaaaaaaa
|-- body
|-- _Bbbbbbbbbbbbbb
|-- compound
|-- output
|-- *
|-- _Aaaaaaaaaaaaa2
|-- _Aaaaaaaaaaaaa2
|-- for-to
|-- <==
|-- _Aaaaaaaaaaaaa2
|-- _Bbbbbbbbbbbbbb
|-- body
|-- _Aaaaaaaaaaaaaa
|-- compound
|-- output
|-- *
|-- _Aaaaaaaaaaaaa2
|-- _Aaaaaaaaaaaaa2
|-- <==
|-- _XXXXXXXXXXXXXX
|-- 0
|-- <==
|-- _Ccccccccccccc
|-- 0
|-- while
|-- le
|-- _Ccccccccccccc
|-- _Aaaaaaaaaaaaaa
|-- statement
|-- compound
|-- statement
|-- statement
|-- <==
|-- _Cccccccccccca
|-- 0
|-- while
|-- le

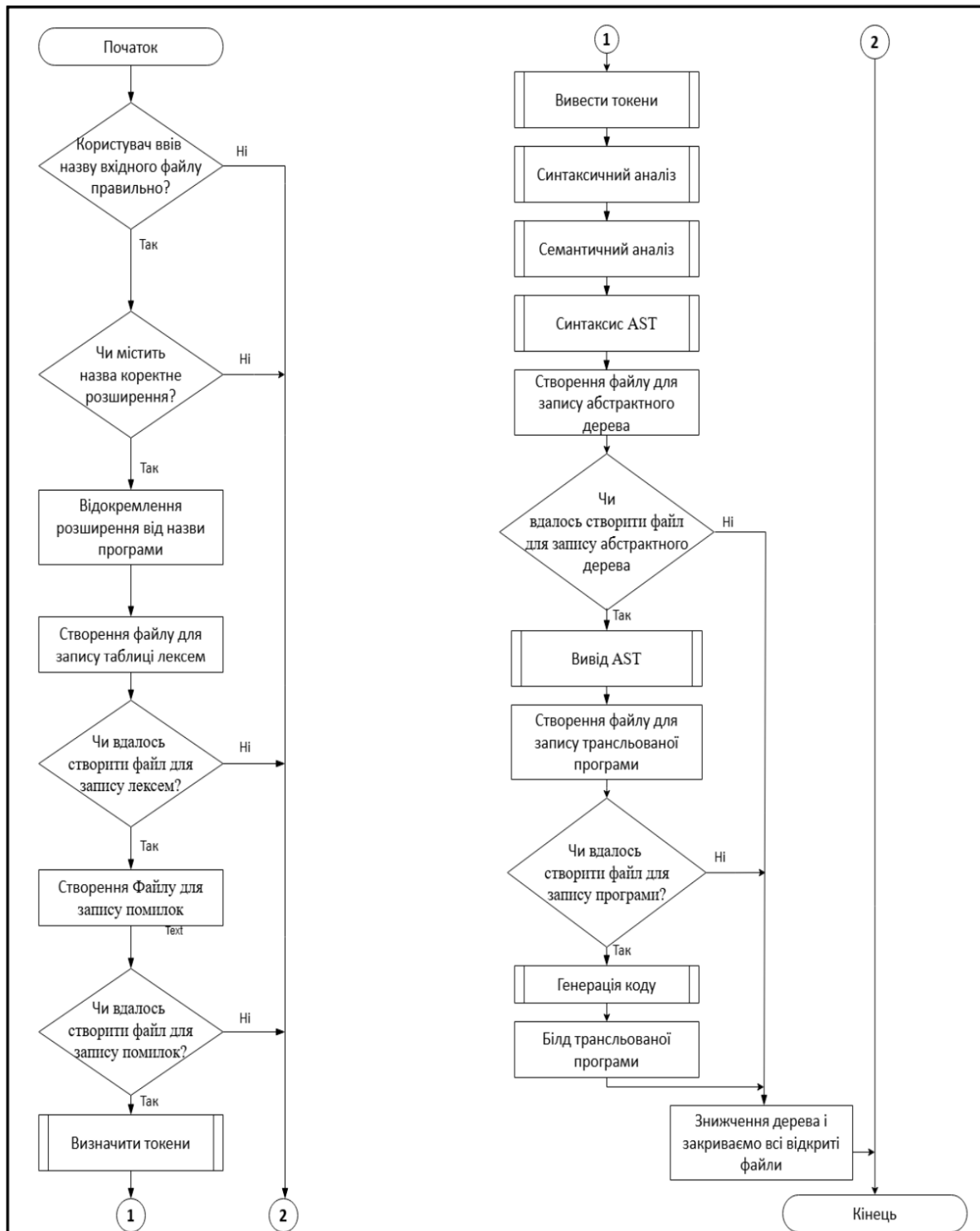
```

```

|-- _Cccccccccccca
|-- _Bbbbbbbbbbbbbb
|-- statement
|-- compound
|-- statement
|-- <==
|-- _XXXXXXXXXXXXXX
|-- +
|-- _XXXXXXXXXXXXXX
|-- 1
|-- <==
|-- _Cccccccccccca
|-- +
|-- _Cccccccccccca
|-- 1
|-- <==
|-- _Cccccccccccc
|-- +
|-- _Cccccccccccc
|-- 1
|-- output
|-- _XXXXXXXXXXXXXX
|-- <==
|-- _XXXXXXXXXXXXXX
|-- 0
|-- <==
|-- _Cccccccccccc
|-- 1
|-- repeat-until
|-- body
|-- compound
|-- statement
|-- statement
|-- <==
|-- _Cccccccccccca
|-- 1
|-- repeat-until
|-- body
|-- compound
|-- statement
|-- <==
|-- _XXXXXXXXXXXXXX
|-- +
|-- _XXXXXXXXXXXXXX
|-- 1

```

```
| | | | | | | | | | | | |--<==  
| | | | | | | | | | | | |--_Cccccccccccca  
| | | | | | | | | | | | |--+  
| | | | | | | | | | | | |--_Cccccccccccca  
| | | | | | | | | | | | |--1  
  
| | | | | | | | |--!  
  
| | | | | | | | |--ge  
| | | | | | | | |--_Cccccccccccca  
| | | | | | | | |--_Bbbbbbbbbbbbbb  
  
| | | | | | | | |--<==  
| | | | | | | | |--_Cccccccccccc  
| | | | | | | | |--+  
| | | | | | | | |--_Cccccccccccc  
| | | | | | | | |--1  
  
| | | | | |--!  
  
| | | | | |--ge  
| | | | | |--_Cccccccccccc  
| | | | | |--_Aaaaaaaaaaaaa  
  
| | |--output  
| | | |--_XXXXXXXXXXXXX
```



Міністерство освіти і науки України					КУРСОВИЙ ПРОЄКТ				
					Розробка системних програмних модулів та компонент систем програмування				
					Алгоритм транслятора S18				
Зм.	Арк.	№ докум.	Підпис	Дата	Літера		Маса		Масштаб
Виконав		Смільч М.Ю.			у				
Керівник		Козак Н.Б.							
Консул'т.									
Консул'т.									
Зав. каф.		Донець Р. Б.							
Реценз.					Аркуш		Аркушів 1		
					НУ «ЛП», ІКТА, каф. ЕОМ, гр КІ-309				

Додаток В (Код на мові C)

Prog1.c

```
#include <stdio.h>

#include <stdlib.h>

int main()
{
;
    _Aaaaaaaaaaaaaa = _Bbbbbbbbbbbbbbb;
    _XXXXXXXXXXXXXX = _YYYYYYYYYYYYYY;
    printf("Enter _Aaaaaaaaaaaaaa:");
    scanf("%d", &_Aaaaaaaaaaaaaa);
    printf("Enter _Bbbbbbbbbbbbbbb:");
    scanf("%d", &_Bbbbbbbbbbbbbbb);
    printf("%d\n", _Aaaaaaaaaaaaaa + _Bbbbbbbbbbbbbbb);
    printf("%d\n", _Aaaaaaaaaaaaaa - _Bbbbbbbbbbbbbbb);
    printf("%d\n", _Aaaaaaaaaaaaaa * _Bbbbbbbbbbbbbbb);
    printf("%d\n", _Aaaaaaaaaaaaaa / _Bbbbbbbbbbbbbbb);
    printf("%d\n", _Aaaaaaaaaaaaaa % _Bbbbbbbbbbbbbbb);
    _XXXXXXXXXXXXXX = (_Aaaaaaaaaaaaaa - _Bbbbbbbbbbbbbbb) * 10 + (_Aaaaaaaaaaaaaa +
_Bbbbbbbbbbbbbbb) / 10;
    _YYYYYYYYYYYYYY = _XXXXXXXXXXXXXX + (_XXXXXXXXXXXXXX % 10);
    printf("%d\n", _XXXXXXXXXXXXXX);
    printf("%d\n", _YYYYYYYYYYYYYY);
    system("pause");
    return 0;
}
```

Prog2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main()
{
    int32_t _Aaaaaaaaaaaaaa, _Bbbbbbbbbbbbbbb, _Ccccccccccccc;
    printf("Enter _Aaaaaaaaaaaaaa:");
    scanf("%hd", &_Aaaaaaaaaaaaaa);
    printf("Enter _Bbbbbbbbbbbbbbb:");
    scanf("%hd", &_Bbbbbbbbbbbbbbb);
```

```

printf("Enter _Cccccccccccc:");
scanf("%hd", &_Cccccccccccc);
if ((_Aaaaaaaaaaaa > _Bbbbbbbbbbbbbb))
{
if ((_Aaaaaaaaaaaa > _Cccccccccccc))
{
goto _Temporalllll;
}
else
{
printf("%d\n", _Cccccccccccc);
goto _Outgotoooooooo;
}
}
_Temporalllll:
printf("%d\n", _Aaaaaaaaaaaa);
goto _Outgotoooooooo;
}
}
if ((_Bbbbbbbbbbbbbb < _Cccccccccccc))
{
printf("%d\n", _Cccccccccccc);
}
else
{
printf("%d\n", _Bbbbbbbbbbbbbb);
}
}
_Outgotoooooooo:
if (((_Aaaaaaaaaaaa == _Bbbbbbbbbbbbbb) || (_Aaaaaaaaaaaa == _Cccccccccccc) &&
(_Bbbbbbbbbbbbbb == _Cccccccccccc)))
{
printf("%d\n", 1);
}
else
{
printf("%d\n", 0);
}
}
if (((_Aaaaaaaaaaaa < 0) || (_Bbbbbbbbbbbbbb < 0) || (_Cccccccccccc < 0)))
{
printf("%d\n", - 1);
}
else
{
printf("%d\n", 0);
}
}
if (!( (_Aaaaaaaaaaaa < (_Bbbbbbbbbbbbbb + _Cccccccccccc))))

```

```

{
printf("%d\n", 1);
}
else
{
printf("%d\n", 0);
}
system("pause");
return 0;
}

```

Prog3.c

```

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

```

```

int main()
{
    int32_t _Aaaaaaaaaa, _Aaaaaaaaaa2, _Bbbbbbbbbbbb, _XXXXXXXXXXXX, _Cccccccccc,
    _Cccccccccc;
    printf("Enter _Aaaaaaaaaa:");
    scanf("%hd", &_Aaaaaaaaaa);
    printf("Enter _Bbbbbbbbbbbb:");
    scanf("%hd", &_Bbbbbbbbbbbb);
    for (int16_t _Aaaaaaaaaa2 = _Aaaaaaaaaa; _Aaaaaaaaaa2 <= _Bbbbbbbbbbbb; _Aaaaaaaaaa2++)
    {
        printf("%d\n", (_Aaaaaaaaaa2 * _Aaaaaaaaaa2));
    }
    for (int16_t _Aaaaaaaaaa2 = _Bbbbbbbbbbbb; _Aaaaaaaaaa2 <= _Aaaaaaaaaa; _Aaaaaaaaaa2++)
    {
        printf("%d\n", (_Aaaaaaaaaa2 * _Aaaaaaaaaa2));
    }
    _XXXXXXXXXXXX = 0;
    _Cccccccccc = 0;
    while (_Cccccccccc < _Aaaaaaaaaa)
    {
        {
            _Cccccccccc = 0;
            while (_Cccccccccc < _Bbbbbbbbbbbb)
            {
                {
                    _XXXXXXXXXXXX = _XXXXXXXXXXXX + 1;
                    _Cccccccccc = _Cccccccccc + 1;
                }
            }
            _Cccccccccc = _Cccccccccc + 1;
        }
    }
    printf("%d\n", _XXXXXXXXXXXX);
    _XXXXXXXXXXXX = 0;
}

```

```

_Ccccccccccccc = 1;
do
{
_Cccccccccccca = 1;
do
{
_Xxxxxxxxxxxxxx = _Xxxxxxxxxxxxxx + 1;
_Cccccccccccca = _Cccccccccccca + 1;
}
while ((!( _Cccccccccccca > _Bbbbbbbbbbbbbb)));
_Ccccccccccccc = _Ccccccccccccc + 1;
}
while ((!( _Ccccccccccccc > _Aaaaaaaaaaaaaa)));
printf("%d\n", _Xxxxxxxxxxxxxx);
system("pause");
return 0;
}

```

Додаток Б (Лістинг проекту на мові C)

```

Asx.cpp
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "translator.h"
#include <iostream>

// таблиця лексем
extern Token* TokenTable;
// кількість лексем
extern unsigned int TokensNum;

static int pos = 0;

// функція створення вузла AST
ASTNode* createNode(TypeOfNodes type, const char* name, ASTNode* left, ASTNode* right)
{
    ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));
    node->nodetype = type;
    strcpy_s(node->name, name);
    node->left = left;
    node->right = right;
    return node;
}

// функція знищення дерева
void destroyTree(ASTNode* root)
{

```



```

    if (root == NULL)
        return;

    // Рекурсивно знищуємо ліве і праве піддерево
    destroyTree(root->left);
    destroyTree(root->right);

    // Звільняємо пам'ять для поточного вузла
    free(root);
}

// набір функцій для рекурсивного спуску
// на кожне правило - окрема функція
ASTNode* program();
ASTNode* variable_declaration();
ASTNode* variable_list();
ASTNode* program_body();
ASTNode* statement();
ASTNode* assignment();
ASTNode* arithmetic_expression();
ASTNode* term();
ASTNode* factor();
ASTNode* input();
ASTNode* output();
ASTNode* conditional();

ASTNode* goto_statement();
ASTNode* label_statement();
ASTNode* for_to_do();
ASTNode* for_downto_do();
ASTNode* while_statement();
ASTNode* repeat_until();

ASTNode* logical_expression();
ASTNode* and_expression();
ASTNode* comparison();
ASTNode* compound_statement();

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

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

    return tree;
}

static void match(TypeOfTokens expectedType)
{
    if (TokenTable[pos].type == expectedType)
        pos++;
    else

```

```

    {
        printf("\nSyntax error in line %d: Expected another type of lexeme.\n", TokenTable[pos].line);
        std::cout << "AST Type: " << TokenTable[pos].type << std::endl;
        std::cout << "AST Expected type:" << expectedType << std::endl;
        exit(10);
    }
}

// <програма> = 'start' 'var' <оголошення змінних> ';' <тіло програми> 'stop'
ASTNode* program()
{
    match(Mainprogram);
    match(StartProgram);
    match(Variable);
    ASTNode* declarations = variable_declaration();
    match(Semicolon);
    ASTNode* body = program_body();
    match(EndProgram);
    return createNode(program_node, "program", declarations, body);
}

// <оголошення змінних> = [<тип даних> <список змінних>]
ASTNode* variable_declaration()
{
    {
        if (TokenTable[pos].type == Type)
        {
            pos++;
            return variable_list();
        }
    }
    return NULL;
}

// <список змінних> = <ідентифікатор> { ',' <ідентифікатор> }
ASTNode* variable_list()
{
    {
        match(Identifier);
        ASTNode* id = createNode(id_node, TokenTable[pos - 1].name, NULL, NULL);
        ASTNode* list = list = createNode(var_node, "var", id, NULL);
        while (TokenTable[pos].type == Comma)
        {
            match(Comma);
            match(Identifier);
            id = createNode(id_node, TokenTable[pos - 1].name, NULL, NULL);
            list = createNode(var_node, "var", id, list);
        }
    }
    return list;
}

// <тіло програми> = <оператор> ';' { <оператор> ';' }
ASTNode* program_body()
{
    {
        ASTNode* stmt = statement();
        //match(Semicolon);
    }
}

```

```

ASTNode* body = stmt;
while (TokenTable[pos].type != EndProgram)
{
    ASTNode* nextStmt = statement();
    body = createNode(statement_node, "statement", body, nextStmt);
}
return body;
}

// <оператор> = <присвоєння> | <ввід> | <вивід> | <умовний оператор> | <складений оператор>
ASTNode* statement()
{
    switch (TokenTable[pos].type)
    {
        case Input: return input();
        case Output: return output();
        case If: return conditional();
        case StartProgram: return compound_statement();
        case Goto: return goto_statement();
        case Label: return label_statement();
        case For:
        {
            int temp_pos = pos + 1;
            while (TokenTable[temp_pos].type != To && TokenTable[temp_pos].type != DownTo && temp_pos <
TokensNum)
            {
                temp_pos++;
            }
            if (TokenTable[temp_pos].type == To)
            {
                return for_to_do();
            }
            else if (TokenTable[temp_pos].type == DownTo)
            {
                return for_downto_do();
            }
            else
            {
                printf("Error: Expected 'To' or 'DownTo' after 'For'\n");
                exit(1);
            }
        }
        case While: return while_statement();
        case Exit:
            match(Exit);
            match(While);
            return createNode(exit_while_node, "exit-while", NULL, NULL);
        case Continue:
            match(Continue);
            match(While);
            return createNode(continue_while_node, "continue-while", NULL, NULL);
        case Repeat: return repeat_until();
        default: return assignment();
    }
}

```

```

    }
}

// <присвоєння> = <ідентифікатор> ':' <арифметичний вираз>
ASTNode* assignment()
{
    ASTNode* id = createNode(id_node, TokenTable[pos].name, NULL, NULL);
    match(Identifier);
    match(Assign);
    ASTNode* expr = arithmetic_expression();
    match(Semicolon);
    return createNode(assign_node, "<==>", id, expr);
}

// <арифметичний вираз> = <доданок> { ('+' | '-') <доданок> }
ASTNode* arithmetic_expression()
{
    ASTNode* left = term();
    while (TokenTable[pos].type == Add || TokenTable[pos].type == Sub)
    {
        TypeOfTokens op = TokenTable[pos].type;
        match(op);
        ASTNode* right = term();
        if (op == Add)
            left = createNode(add_node, "+", left, right);
        else
            left = createNode(sub_node, "-", left, right);
    }
    return left;
}

// <доданок> = <множник> { ('*' | '/') <множник> }
ASTNode* term()
{
    ASTNode* left = factor();
    while (TokenTable[pos].type == Mul || TokenTable[pos].type == Div || TokenTable[pos].type == Mod)
    {
        TypeOfTokens op = TokenTable[pos].type;
        match(op);
        ASTNode* right = factor();
        if (op == Mul)
            left = createNode(mul_node, "*", left, right);
        if (op == Div)
            left = createNode(div_node, "/", left, right);
        if (op == Mod)
            left = createNode(mod_node, "%", left, right);
    }
    return left;
}

// <множник> = <ідентифікатор> | <число> | '(' <арифметичний вираз> ')'
ASTNode* factor()
{

```

```

if (TokenTable[pos].type == Identifier)
{
    ASTNode* id = createNode(id_node, TokenTable[pos].name, NULL, NULL);
    match(Identifier);
    return id;
}
else
    if (TokenTable[pos].type == Number)
    {
        ASTNode* num = createNode(num_node, TokenTable[pos].name, NULL, NULL);
        match(Number);
        return num;
    }
    else
        if (TokenTable[pos].type == LBracket)
        {
            match(LBracket);
            ASTNode* expr = arithmetic_expression();
            match(RBracket);
            return expr;
        }
        else
        {
            printf("\nSyntax error in line %d: A multiplier was expected.\n", TokenTable[pos].line);
            exit(11);
        }
}

// <ввід> = 'input' <ідентифікатор>
ASTNode* input()
{
    match(Input);
    ASTNode* id = createNode(id_node, TokenTable[pos].name, NULL, NULL);
    match(Identifier);
    match(Semicolon);
    return createNode(input_node, "input", id, NULL);
}

// <вивід> = 'output' <ідентифікатор>
ASTNode* output()
{
    match(Output); // Match the "Output" token

    ASTNode* expr = NULL;
    // Check for a negative number
    if (TokenTable[pos].type == Sub && TokenTable[pos + 1].type == Number)
    {
        pos++; // Skip the 'Sub' token
        expr = createNode(sub_node, "-", createNode(num_node, "0", NULL, NULL),
            createNode(num_node, TokenTable[pos].name, NULL, NULL));
        match(Number); // Match the number token
    }
    else

```

```

{
    // Parse the arithmetic expression
    expr = arithmetic_expression();
}
match(Semicolon); // Ensure the statement ends with a semicolon

// Create the output node with the parsed expression as its left child
return createNode(output_node, "output", expr, NULL);
}

// <умовний оператор> = 'if' <логічний вираз> <оператор> [ 'else' <оператор> ]
ASTNode* conditional()
{
    match(If);
    ASTNode* condition = logical_expression();
    ASTNode* ifBranch = statement();
    ASTNode* elseBranch = NULL;
    if (TokenTable[pos].type == Else)
    {
        match(Else);
        elseBranch = statement();
    }
    return createNode(if_node, "if", condition, createNode(statement_node, "branches", ifBranch, elseBranch));
}

ASTNode* goto_statement()
{
    match(Goto);
    if (TokenTable[pos].type == Identifier)
    {
        ASTNode* label = createNode(label_node, TokenTable[pos].name, NULL, NULL);
        match(Identifier);
        match(Semicolon);
        return createNode(goto_node, "goto", label, NULL);
    }
    else
    {
        printf("Syntax error: Expected a label after 'goto' at line %d.\n", TokenTable[pos].line);
        exit(1);
    }
}

ASTNode* label_statement()
{
    match(Label);
    ASTNode* label = createNode(label_node, TokenTable[pos - 1].name, NULL, NULL);
    return label;
}

ASTNode* for_to_do()
{

```

```

match(For);

if (TokenTable[pos].type != Identifier)
{
    printf("Syntax error: Expected variable name after 'for' at line %d.\n", TokenTable[pos].line);
    exit(1);
}
ASTNode* var = createNode(id_node, TokenTable[pos].name, NULL, NULL);
match(Identifier);
match(Assign);
ASTNode* start = arithmetic_expression();
match(To);
ASTNode* end = arithmetic_expression();
match(Do);
ASTNode* body = statement();
// Повертаємо вузол циклу for-to
return createNode(for_to_node, "for-to",
    createNode(assign_node, "<==", var, start),
    createNode(statement_node, "body", end, body));
}

```

```

ASTNode* for_downto_do()
{
    // Очікуємо "for"
    match(For);

    // Очікуємо ідентифікатор змінної циклу
    if (TokenTable[pos].type != Identifier)
    {
        printf("Syntax error: Expected variable name after 'for' at line %d.\n", TokenTable[pos].line);
        exit(1);
    }
    ASTNode* var = createNode(id_node, TokenTable[pos].name, NULL, NULL);
    match(Identifier);
    match(Assign);
    ASTNode* start = arithmetic_expression();
    match(DownTo);
    ASTNode* end = arithmetic_expression();
    match(Do);
    ASTNode* body = statement();
    // Повертаємо вузол циклу for-to
    return createNode(for_downto_node, "for-downto",
        createNode(assign_node, "<==", var, start),
        createNode(statement_node, "body", end, body));
}

```

```

ASTNode* while_statement()
{
    match(While);
    ASTNode* condition = logical_expression();

```

```

// Parse the body of the While loop
ASTNode* body = NULL;
while (1) // Process until "End While"
{
    if (TokenTable[pos].type == End)
    {
        match(End);
        match(While);
        break; // End of the While loop
    }
    else
    {
        // Delegate to the `statement` function
        ASTNode* stmt = statement();
        body = createNode(statement_node, "statement", body, stmt);
    }
}

return createNode(while_node, "while", condition, body);
}

// Updated variable validation logic
ASTNode* validate_identifier()
{
    const char* identifierName = TokenTable[pos].name;

    // Check if the identifier was declared
    bool declared = false;
    for (unsigned int i = 0; i < TokensNum; i++)
    {
        if (TokenTable[i].type == Variable && !strcmp(TokenTable[i].name, identifierName))
        {
            declared = true;
            break;
        }
    }

    if (!declared && (pos == 0 || TokenTable[pos - 1].type != Goto))
    {
        printf("Syntax error: Undeclared identifier '%s' at line %d.\n", identifierName, TokenTable[pos].line);
        exit(1);
    }

    match(Identifier);
    return createNode(id_node, identifierName, NULL, NULL);
}

ASTNode* repeat_until()
{
    match(Repeat);
    ASTNode* body = NULL;
    ASTNode* stmt = statement();
    body = createNode(statement_node, "body", body, stmt);
}

```



```

//pos++;
match(Until);
ASTNode* condition = logical_expression();
return createNode(repeat_until_node, "repeat-until", body, condition);
}

// <логічний вираз> = <вираз I> { '|' <вираз I> }
ASTNode* logical_expression()
{
    ASTNode* left = and_expression();
    while (TokenTable[pos].type == Or)
    {
        match(Or);
        ASTNode* right = and_expression();
        left = createNode(or_node, "|", left, right);
    }
    return left;
}

// <вираз I> = <порівняння> { '&' <порівняння> }
ASTNode* and_expression()
{
    ASTNode* left = comparison();
    while (TokenTable[pos].type == And)
    {
        match(And);
        ASTNode* right = comparison();
        left = createNode(and_node, "&", left, right);
    }
    return left;
}

// <порівняння> = <операція порівняння> | '!' '(' <логічний вираз> ')' | '(' <логічний вираз> ')'
// <операція порівняння> = <арифметичний вираз> <менше-більше> <арифметичний вираз>
// <менше-більше> = '>' | '<' | '=' | '<='
ASTNode* comparison()
{
    {
        if (TokenTable[pos].type == Not)
        {
            // Варіант: ! (<логічний вираз>)
            match(Not);
            match(LBraket);
            ASTNode* expr = logical_expression();
            match(RBraket);
            return createNode(not_node, "!", expr, NULL);
        }
        else
        {
            if (TokenTable[pos].type == LBraket)
            {
                // Варіант: ( <логічний вираз> )
                match(LBraket);
                ASTNode* expr = logical_expression();
                match(RBraket);
            }
        }
    }
}

```

```

    return expr; // Повертаємо вираз у дужках як піддерево
}
else
{
    // Варіант: <арифметичний вираз> <менше-більше> <арифметичний вираз>
    ASTNode* left = arithmetic_expression();
    if (TokenTable[pos].type == Greater || TokenTable[pos].type == Less ||
        TokenTable[pos].type == Equality || TokenTable[pos].type == NotEquality)
    {
        TypeOfTokens op = TokenTable[pos].type;
        char operatorName[16];
        strcpy_s(operatorName, TokenTable[pos].name);
        match(op);
        ASTNode* right = arithmetic_expression();
        return createNode(cmp_node, operatorName, left, right);
    }
    else
    {
        printf("\nSyntax error: A comparison operation is expected.\n");
        exit(12);
    }
}
}

// <складений оператор> = 'start' <тіло програми> 'stop'
ASTNode* compound_statement()
{
    match(StartProgram);
    ASTNode* body = program_body();
    match(EndProgram);
    return createNode(compound_node, "compound", body, NULL);
}

// функція для друку AST у вигляді дерева на екран
void PrintAST(ASTNode* node, int level)
{
    if (node == NULL)
        return;

    // Відступи для позначення рівня вузла
    for (int i = 0; i < level; i++)
        printf("  ");

    // Виводимо інформацію про вузол
    printf("-- %s", node->name);
    printf("\n");

    // Рекурсивний друк лівого та правого піддерева
    if (node->left || node->right)
    {
        PrintAST(node->left, level + 1);
        PrintAST(node->right, level + 1);
    }
}

```

```

}

// функція для друку AST у вигляді дерева у файл
void PrintASTToFile(ASTNode* node, int level, FILE* outFile)
{
    if (node == NULL)
        return;

    // Відступи для позначення рівня вузла
    for (int i = 0; i < level; i++)
        fprintf(outFile, "  ");

    // Виводимо інформацію про вузол
    fprintf(outFile, "|-- %s", node->name);
    fprintf(outFile, "\n");

    // Рекурсивний друк лівого та правого піддерева
    if (node->left || node->right)
    {
        PrintASTToFile(node->left, level + 1, outFile);
        PrintASTToFile(node->right, level + 1, outFile);
    }
}

```

Codegen.cpp

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "translator.h"

// таблиця лексем
extern Token* TokenTable;
// кількість лексем
extern unsigned int TokensNum;

// таблиця ідентифікаторів
extern Id* IdTable;
// кількість ідентифікаторів
extern unsigned int IdNum;

static int pos = 2;

// набір функцій для рекурсивного спуску
// на кожне правило - окрема функція

void gen_variable_declaration(FILE* outFile);
void gen_variable_list(FILE* outFile);
void gen_program_body(FILE* outFile);
void gen_statement(FILE* outFile);
void gen_assignment(FILE* outFile);
void gen_arithmetic_expression(FILE* outFile);

```

```

void gen_term(FILE* outFile);
void gen_factor(FILE* outFile);
void gen_input(FILE* outFile);
void gen_output(FILE* outFile);
void gen_conditional(FILE* outFile);

void gen_goto_statement(FILE* outFile);
void gen_label_statement(FILE* outFile);
void gen_for_to_do(FILE* outFile);
void gen_for_downto_do(FILE* outFile);
void gen_while_statement(FILE* outFile);
void gen_repeat_until(FILE* outFile);

void gen_logical_expression(FILE* outFile);
void gen_and_expression(FILE* outFile);
void gen_comparison(FILE* outFile);
void gen_compound_statement(FILE* outFile);

void generateCCode(FILE* outFile)
{
    fprintf(outFile, "#include <stdio.h>\n\n");
    fprintf(outFile, "#include <stdlib.h>\n\n");
    fprintf(outFile, "int main() \n{\n");
    pos++;
    gen_variable_declaration(outFile);
    fprintf(outFile, ";\n");
    pos++;
    gen_program_body(outFile);
    fprintf(outFile, "    system(\"pause\");\n ");
    fprintf(outFile, "    return 0;\n");
    fprintf(outFile, "}\n");
}

// <оголошення змінних> = [<тип даних> <список змінних>]
void gen_variable_declaration(FILE* outFile)
{
    if (TokenTable[pos + 1].type == Type)
    {
        fprintf(outFile, "    int ");
        pos++;
        pos++;
        gen_variable_list(outFile);
    }
}

// <список змінних> = <ідентифікатор> { ',' <ідентифікатор> }
void gen_variable_list(FILE* outFile)
{
    fprintf(outFile, TokenTable[pos++].name);
    while (TokenTable[pos].type == Comma)
    {
        fprintf(outFile, ", ");
        pos++;
    }
}

```

```

        fprintf(outFile, TokenTable[pos++].name);
    }
}

// <тіло програми> = <оператор> ';' { <оператор> ';' }
void gen_program_body(FILE* outFile)
{
    while (pos < TokensNum && TokenTable[pos].type != EndProgram)
    {
        gen_statement(outFile);
    }

    if (pos >= TokensNum || TokenTable[pos].type != EndProgram)
    {
        printf("Error: 'EndProgram' token not found or unexpected end of tokens.\n");
        exit(1);
    }
}

// <оператор> = <присвоєння> | <ввід> | <вивід> | <умовний оператор> | <складений оператор>
void gen_statement(FILE* outFile)
{
    switch (TokenTable[pos].type)
    {
        case Input: gen_input(outFile); break;
        case Output: gen_output(outFile); break;
        case If: gen_conditional(outFile); break;
        case StartProgram: gen_compound_statement(outFile); break;
        case Goto: gen_goto_statement(outFile); break;
        case Label: gen_label_statement(outFile); break;
        case For:
        {
            int temp_pos = pos + 1;

            while (TokenTable[temp_pos].type != To && TokenTable[temp_pos].type != DownTo && temp_pos <
TokensNum)
            {
                temp_pos++;
            }

            if (TokenTable[temp_pos].type == To)
            {
                gen_for_to_do(outFile);
            }
            else if (TokenTable[temp_pos].type == DownTo)
            {
                gen_for_downto_do(outFile);
            }
            else
            {
                printf("Error: Expected 'To' or 'DownTo' after 'For'\n");
            }
        }
    }
}

```

```

break;
case While: gen_while_statement(outFile); break;
case Exit:
    fprintf(outFile, "    break;\n");
    pos += 2;
    break;

case Continue:
    fprintf(outFile, "    continue;\n");
    pos += 2;
    break;
case Repeat: gen_repeat_until(outFile); break;
default: gen_assignment(outFile);
}
}

// <присвоєння> = <ідентифікатор> ':' '=' <арифметичний вираз>
void gen_assignment(FILE* outFile)
{
    fprintf(outFile, " ");
    fprintf(outFile, TokenTable[pos++].name);
    fprintf(outFile, " = ");
    pos++;
    gen_arithmetic_expression(outFile);
    pos++;
    fprintf(outFile, ";\n");
}

// <арифметичний вираз> = <доданок> { ('+' | '-') <доданок> }
void gen_arithmetic_expression(FILE* outFile)
{
    gen_term(outFile);
    while (TokenTable[pos].type == Add || TokenTable[pos].type == Sub)
    {
        if (TokenTable[pos].type == Add)
            fprintf(outFile, " + ");
        else
            fprintf(outFile, " - ");
        pos++;
        gen_term(outFile);
    }
}

// <доданок> = <множник> { ('*' | '/') <множник> }
void gen_term(FILE* outFile)
{
    gen_factor(outFile);
    while (TokenTable[pos].type == Mul || TokenTable[pos].type == Div || TokenTable[pos].type == Mod)
    {
        if (TokenTable[pos].type == Mul)
            fprintf(outFile, " * ");
        if (TokenTable[pos].type == Div)
            fprintf(outFile, " / ");
    }
}

```

```

        if (TokenTable[pos].type == Mod)
            fprintf(outFile, " %%% ");
        pos++;
        gen_factor(outFile);
    }
}

// <множник> = <дентифікатор> | <число> | '(' <арифметичний вираз> ')'
void gen_factor(FILE* outFile)
{
    if (TokenTable[pos].type == Identifier || TokenTable[pos].type == Number)
        fprintf(outFile, TokenTable[pos++].name);
    else
        if (TokenTable[pos].type == LBraket)
        {
            fprintf(outFile, "(");
            pos++;
            gen_arithmetic_expression(outFile);
            fprintf(outFile, ")");
            pos++;
        }
}

// <ввд> = 'input' <дентифікатор>
void gen_input(FILE* outFile)
{
    fprintf(outFile, " printf(\"Enter \");
    fprintf(outFile, TokenTable[pos + 1].name);
    fprintf(outFile, ":\");\n");
    fprintf(outFile, " scanf(\"%%d\", &");
    pos++;
    fprintf(outFile, TokenTable[pos++].name);
    fprintf(outFile, ");\n");
    pos++;
}

// <вивд> = 'output' <дентифікатор>
void gen_output(FILE* outFile)
{
    pos++;

    if (TokenTable[pos].type == Sub && TokenTable[pos + 1].type == Number)
    {
        fprintf(outFile, " printf(\"%%d\\n\", -%s);\n", TokenTable[pos + 1].name);
        pos += 2;
    }
    else
    {
        fprintf(outFile, " printf(\"%%d\\n\", ");
        gen_arithmetic_expression(outFile);
        fprintf(outFile, ");\n");
    }
}

```

```

    if (TokenTable[pos].type == Semicolon)
    {
        pos++;
    }
    else
    {
        printf("Error: Expected a semicolon at the end of 'Output' statement.\n");
        exit(1);
    }
}

// <умовний оператор> = 'if' <логічний вираз> 'then' <оператор> [ 'else' <оператор> ]
void gen_conditional(FILE* outFile)
{
    fprintf(outFile, "  if (");
    pos++;
    gen_logical_expression(outFile);
    fprintf(outFile, ")\n");
    gen_statement(outFile);
    if (TokenTable[pos].type == Else)
    {
        fprintf(outFile, "  else\n");
        pos++;
        gen_statement(outFile);
    }
}

void gen_goto_statement(FILE* outFile)
{
    fprintf(outFile, "  goto %s;\n", TokenTable[pos + 1].name);
    pos += 3;
}

void gen_label_statement(FILE* outFile)
{
    fprintf(outFile, "%s:\n", TokenTable[pos].name);
    pos++;
}

void gen_for_to_do(FILE* outFile)
{
    int temp_pos = pos + 1;

    const char* loop_var = TokenTable[temp_pos].name;
    temp_pos += 2;

    fprintf(outFile, "  for (int %s = ", loop_var);
    pos = temp_pos;
    gen_arithmetic_expression(outFile);
    fprintf(outFile, "; ");

```



```

while (TokenTable[pos].type != To && pos < TokensNum)
{
    pos++;
}

if (TokenTable[pos].type == To)
{
    pos++;
    fprintf(outFile, "%s <= ", loop_var);
    gen_arithmetic_expression(outFile);
}
else
{
    printf("Error: Expected 'To' in For-To loop\n");
    return;
}

fprintf(outFile, "; %s++)\n", loop_var);

if (TokenTable[pos].type == Do)
{
    pos++;
}
else
{
    printf("Error: Expected 'Do' after 'To' clause\n");
    return;
}

gen_statement(outFile);
}
void gen_for_downto_do(FILE* outFile)
{
    int temp_pos = pos + 1;

    const char* loop_var = TokenTable[temp_pos].name;
    temp_pos += 2;

    fprintf(outFile, " for (int %s = ", loop_var);
    pos = temp_pos;
    gen_arithmetic_expression(outFile);
    fprintf(outFile, "; ");

    while (TokenTable[pos].type != DownTo && pos < TokensNum)
    {
        pos++;
    }

    if (TokenTable[pos].type == DownTo)
    {
        pos++;

        fprintf(outFile, "%s >= ", loop_var);

```

```

    gen_arithmetic_expression(outFile);
}
else
{
    printf("Error: Expected 'Downto' in For-Downto loop\n");
    return;
}

fprintf(outFile, "; %s--)\n", loop_var);

if (TokenTable[pos].type == Do)
{
    pos++;
}
else
{
    printf("Error: Expected 'Do' after 'Downto' clause\n");
    return;
}

gen_statement(outFile);
}

void gen_while_statement(FILE* outFile)
{
    fprintf(outFile, " while (");
    pos++;
    gen_logical_expression(outFile);
    fprintf(outFile, ")\n {\n");

    while (pos < TokensNum)
    {
        if (TokenTable[pos].type == End && TokenTable[pos + 1].type == While)
        {
            pos += 2;
            break;
        }
        else
        {
            gen_statement(outFile);
            if (TokenTable[pos].type == Semicolon)
            {
                pos++;
            }
        }
    }

    fprintf(outFile, " }\n");
}

void gen_repeat_until(FILE* outFile)
{

```

```

    fprintf(outFile, " do\n");
    pos++;
    do
    {
        gen_statement(outFile);
    } while (TokenTable[pos].type != Until);
    fprintf(outFile, " while (");
    pos++;
    gen_logical_expression(outFile);
    fprintf(outFile, ");\n");
}

// <логический выражение> = <выражение> { '|' <выражение> }
void gen_logical_expression(FILE* outFile)
{
    gen_and_expression(outFile);
    while (TokenTable[pos].type == Or)
    {
        fprintf(outFile, " || ");
        pos++;
        gen_and_expression(outFile);
    }
}

// <выражение> = <порядкованное> { '&' <порядкованное> }
void gen_and_expression(FILE* outFile)
{
    gen_comparison(outFile);
    while (TokenTable[pos].type == And)
    {
        fprintf(outFile, " && ");
        pos++;
        gen_comparison(outFile);
    }
}

// <порядкованное> = <оператор порядка> | C!C C(C <логический выражение> C)C | C(C <логический выражение> C)C
// <оператор порядка> = <арифметический выражение> <меньше-больше> <арифметический выражение>
// <меньше-больше> = C>C | C<C | C=C | C<>C
void gen_comparison(FILE* outFile)
{
    if (TokenTable[pos].type == Not)
    {
        // инверсия: !(<логический выражение>)
        fprintf(outFile, "!(");
        pos++;
        pos++;
        gen_logical_expression(outFile);
        fprintf(outFile, ")");
        pos++;
    }
    else
        if (TokenTable[pos].type == LBracket)

```

```

{
    // ¬ар≥ант: ( <лог≥чний вираз> )
    fprintf(outFile, "(");
    pos++;
    gen_logical_expression(outFile);
    fprintf(outFile, ")");
    pos++;
}
else
{
    // ¬ар≥ант: <арифметичний вираз> <менше-б≥льше> <арифметичний вираз>
    gen_arithmetic_expression(outFile);
    if (TokenTable[pos].type == Greate || TokenTable[pos].type == Less ||
        TokenTable[pos].type == Equality || TokenTable[pos].type == NotEquality)
    {
        switch (TokenTable[pos].type)
        {
            case Greate: fprintf(outFile, " > "); break;
            case Less: fprintf(outFile, " < "); break;
            case Equality: fprintf(outFile, " == "); break;
            case NotEquality: fprintf(outFile, " != "); break;
        }
        pos++;
        gen_arithmetic_expression(outFile);
    }
}
}

// <складений оператор> = 'start' <т≥лю програми> 'stop'
void gen_compound_statement(FILE* outFile)
{
    fprintf(outFile, " {\n");
    pos++;
    gen_program_body(outFile);
    fprintf(outFile, " }\n");
    pos++;
}

```

codegenfromast.cpp

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "translator.h"

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

```

```

switch (node->nodetype)
{
case program_node:
    fprintf(outFile, "#include <stdio.h>\n#include <stdlib.h>\n\nint main() \n{\n");
    generateCodefromAST(node->left, outFile); // Оголошення змінних
    generateCodefromAST(node->right, outFile); // Тіло програми
    fprintf(outFile, "    system(\"pause\");\n");
    fprintf(outFile, "    return 0;\n}\n");
    break;

case var_node:
    // Якщо є права частина (інші змінні), додаємо коми і генеруємо для них код
    if (node->right != NULL)
    {
        //fprintf(outFile, ", ");
        generateCodefromAST(node->right, outFile); // Рекурсивно генеруємо код для інших змінних
    }
    fprintf(outFile, "    int "); // Виводимо тип змінних (в даному випадку int)
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ";\n"); // Завершуємо оголошення змінних
    break;

case id_node:
    fprintf(outFile, "%s", node->name);
    break;

case num_node:
    fprintf(outFile, "%s", node->name);
    break;

case assign_node:
    fprintf(outFile, " ");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " = ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ";\n");
    break;

case add_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " + ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case sub_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " - ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

```

```

case mul_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " * ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case mod_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " %% ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case div_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " / ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case input_node:
    fprintf(outFile, "    printf(\"Enter \");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ":\");\n");
    fprintf(outFile, "    scanf(\"%d\", &");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ");\n");
    break;

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

    fprintf(outFile, ");\n");
    break;

case if_node:
    fprintf(outFile, "    if (");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ") \n");
    generateCodefromAST(node->right->left, outFile);
    if (node->right->right != NULL)
    {
        fprintf(outFile, "    else\n");
        generateCodefromAST(node->right->right, outFile);
    }

```

```

    }
    break;

case goto_node:
    fprintf(outFile, " goto %s;\n", node->left->name);
    break;

case label_node:
    fprintf(outFile, "%s:\n", node->name);
    break;

case for_to_node:
    fprintf(outFile, " for (int ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, " = ");
    generateCodefromAST(node->left->right, outFile);
    fprintf(outFile, "; ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, " <= ");
    generateCodefromAST(node->right->left, outFile);
    fprintf(outFile, "; ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, "++)\n");
    generateCodefromAST(node->right->right, outFile);
    break;

case for_downto_node:
    fprintf(outFile, " for (int ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, " = ");
    generateCodefromAST(node->left->right, outFile);
    fprintf(outFile, "; ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, " >= ");
    generateCodefromAST(node->right->left, outFile);
    fprintf(outFile, "; ");
    generateCodefromAST(node->left->left, outFile);
    fprintf(outFile, "--)\n");
    generateCodefromAST(node->right->right, outFile);
    break;

case while_node:
    fprintf(outFile, " while (");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ")\n");
    fprintf(outFile, " {\n");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, " }\n");
    break;

case exit_while_node:
    fprintf(outFile, " break;\n");
    break;

```

```

case continue_while_node:
    fprintf(outFile, " continue;\n");
    break;

case repeat_until_node:
    fprintf(outFile, " do\n");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " while (");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ");\n");
    break;

case or_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " || ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case and_node:
    fprintf(outFile, "(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, " && ");
    generateCodefromAST(node->right, outFile);
    fprintf(outFile, ")");
    break;

case not_node:
    fprintf(outFile, "!(");
    generateCodefromAST(node->left, outFile);
    fprintf(outFile, ")");
    break;

case cmp_node:
    generateCodefromAST(node->left, outFile);
    if (!strcmp(node->name, "Le"))
        fprintf(outFile, " < ");
    else if (!strcmp(node->name, "Ge"))
        fprintf(outFile, " > ");
    else
        if (!strcmp(node->name, "="))
            fprintf(outFile, " == ");
        else
            if (!strcmp(node->name, "<>"))
                fprintf(outFile, " != ");
            else
                fprintf(outFile, " %s ", node->name);
    generateCodefromAST(node->right, outFile);
    break;

case statement_node:

```



```

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

    case compount_node:
        fprintf(outFile, " {\n");
        generateCodefromAST(node->left, outFile);
        fprintf(outFile, " }\n");
        break;

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

```

compile.cpp

```

#include <Windows.h>
#include <stdio.h>
#include <string>
#include <fstream>

#define SCOPE_EXIT_CAT2(x, y) x##y
#define SCOPE_EXIT_CAT(x, y) SCOPE_EXIT_CAT2(x, y)
#define SCOPE_EXIT auto SCOPE_EXIT_CAT(scopeExit_, __COUNTER__) = Safe::MakeScopeExit() += [&]

namespace Safe
{
    template <typename F>
    class ScopeExit
    {
    public:
        explicit ScopeExit(A&& action) : _action(std::move(action)) {}
        ~ScopeExit() { _action(); }

        ScopeExit() = delete;
        ScopeExit(const ScopeExit&) = delete;
        ScopeExit& operator=(const ScopeExit&) = delete;
        ScopeExit(ScopeExit&&) = delete;
        ScopeExit& operator=(ScopeExit&&) = delete;
        ScopeExit(const A&) = delete;
        ScopeExit(A&) = delete;

    private:
        A _action;
    };

    struct MakeScopeExit
    {

```

```

    template <typename F>
    ScopeExit<F> operator+=(F&& f)
    {
        return ScopeExit<F>(std::forward<F>(f));
    }
};

}

bool is_file_accessible(const char* file_path)
{
    std::ifstream file(file_path);
    return file.is_open();
}

void compile_to_exe(const char* source_file, const char* output_file)
{
    if (!is_file_accessible(source_file))
    {
        printf("Error: Source file %s is not accessible.\n", source_file);
        return;
    }

    wchar_t current_dir[MAX_PATH];
    if (!GetCurrentDirectoryW(MAX_PATH, current_dir))
    {
        printf("Error retrieving current directory. Error code: %lu\n", GetLastError());
        return;
    }

    //wprintf(L"CurrentDirectory: %s\n", current_dir);

    wchar_t command[512];
    _snwprintf_s(
        command,
        std::size(command),
        L"compiler\\MinGW-master\\MinGW\\bin\\gcc.exe -std=c11 \"%s\\%S\" -o \"%s\\%S\"",
        current_dir, source_file, current_dir, output_file
    );

    //wprintf(L"Command: %s\n", command);

    STARTUPINFO si = { 0 };
    PROCESS_INFORMATION pi = { 0 };
    si.cb = sizeof(si);

    if (CreateProcessW(
        NULL,
        command,
        NULL,
        NULL,
        FALSE,
        0,
        NULL,

```

```

    current_dir,
    &si,
    &pi
))
{
    WaitForSingleObject(pi.hProcess, INFINITE);

    DWORD exit_code;
    GetExitCodeProcess(pi.hProcess, &exit_code);

    if (exit_code == 0)
    {
        wprintf(L"File successfully compiled into %s\\%S\n", current_dir, output_file);
    }
    else
    {
        wprintf(L"Compilation error for %. Exit code: %lu\n", source_file, exit_code);
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
else
{
    DWORD error_code = GetLastError();
    wprintf(L"Failed to start compiler process. Error code: %lu\n", error_code);
}
}

```

Lexer.cpp

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "translator.h"
#include <locale>

// 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[], FILE* errFile)
{
    States state = Start;
    Token TempToken;
    // кількість лексем
    unsigned int NumberOfTokens = 0;
    char ch, buf[32];
    int line = 1;

    // читання першого символу з файлу
    ch = getc(F);
    printf("Size of buf: %d", sizeof(buf));
    // пошук лексем

```

```

while (1)
{
    switch (state)
    {
        // стан Start - початок виділення чергової лексеми
        // якщо поточний символ маленька літера, то переходимо до стану Letter
        // якщо поточний символ цифра, то переходимо до стану Digit
        // якщо поточний символ пробіл, символ табуляції або переходу на новий рядок, то переходимо до стану
Separators
        // якщо поточний символ / то є ймовірність, що це коментар, переходимо до стану SComment
        // якщо поточний символ EOF (ознака кінця файлу), то переходимо до стану EndOfFile
        // якщо поточний символ відмінний від попередніх, то переходимо до стану Another
    case Start:
    {
        if (ch == EOF)
            state = EndOfFile;
        else
            if ((ch <= 'z' && ch >= 'a') || (ch <= 'Z' && ch >= 'A') || ch == '_')
                state = Letter;
            else
                if (ch <= '9' && ch >= '0')
                    state = Digit;
                else
                    if (ch == ' ' || ch == '\t' || ch == '\n')
                        state = Separators;
                    else
                        if (ch == '/')
                            state = SComment;
                        else
                            state = Another;
        break;
    }

    // стан Finish - кінець виділення чергової лексеми і запис лексеми у таблицю лексем
    case Finish:
    {
        if (NumberOfTokens < MAX_TOKENS)
        {
            TokenTable[NumberOfTokens++] = TempToken;
            if (ch != EOF)
                state = Start;
            else
                state = EndOfFile;
        }
        else
        {
            printf("\n\t\t\ttoo many tokens !!!\n");
            return NumberOfTokens - 1;
        }
        break;
    }

    // стан EndOfFile - кінець файлу, можна завершувати пошук лексем

```

```

case EndOfFile:
{
    return NumberOfTokens;
}

// стан Letter - поточний символ - маленька літера, поточна лексема - ключове слово або ідентифікатор
case Letter:
{
    buf[0] = ch;
    int j = 1;

    ch = getc(F);

    while (((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') ||
        (ch >= '0' && ch <= '9') || ch == '_' || ch == ':' || ch == '-') && j < 31)
    {
        buf[j++] = ch;
        ch = getc(F);
    }
    buf[j] = '\0';

    TypeOfTokens temp_type = Unknown;

    if (!strcmp(buf, "end"))
    {
        char next_buf[31];
        int next_j = 0;

        while (ch == ' ' || ch == '\t')
        {
            ch = getc(F);
        }

        while (((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) && next_j < 31)
        {
            next_buf[next_j++] = ch;
            ch = getc(F);
        }
        next_buf[next_j] = '\0';

        if (!strcmp(next_buf, "while"))
        {
            temp_type = End;
            strcpy_s(TempToken.name, buf);
            TempToken.type = temp_type;
            TempToken.value = 0;
            TempToken.line = line;
            TokenTable[NumberOfTokens++] = TempToken;

            temp_type = While;
            strcpy_s(TempToken.name, next_buf);
            TempToken.type = temp_type;
            TempToken.value = 0;
        }
    }
}

```

```

TempToken.line = line;
TokenTable[NumberOfTokens++] = TempToken;

state = Start;
break;
}
else
{
    temp_type = EndProgram;
    strcpy_s(TempToken.name, buf);
    TempToken.type = temp_type;
    TempToken.value = 0;
    TempToken.line = line;
    state = Finish;

    for (int k = next_j - 1; k >= 0; k--)
    {
        ungetc(next_buf[k], F);
    }
    break;
}
}

if (!strcmp(buf, "Name"))
{
    char next_buf[32];
    int next_j = 0;

    while (ch == ' ' || ch == '\t')
    {
        ch = getc(F);
    }

    while (((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9' || ch == ',' )) && next_j
< 31)
    {
        next_buf[next_j++] = ch;
        ch = getc(F);
    }
    next_buf[next_j] = '\0';

    if (next_buf[strlen(next_buf) - 1] == ';')
    {
        temp_type = Mainprogram;
        strcpy_s(TempToken.name, buf);
        TempToken.type = temp_type;
        TempToken.value = 0;
        TempToken.line = line;
        TokenTable[NumberOfTokens++] = TempToken;

        next_buf[strlen(next_buf) - 1] = '\0';
        temp_type = ProgramName;
        strcpy_s(TempToken.name, next_buf);

```

```

    TempToken.type = temp_type;
    TempToken.value = 0;
    TempToken.line = line;
    TokenTable[NumberOfTokens++] = TempToken;

    state = Start;
    break;
}
}

if (!strcmp(buf, "mainprogram")) temp_type = Mainprogram;
else if (!strcmp(buf, "start")) temp_type = StartProgram;
else if (!strcmp(buf, "data")) temp_type = Variable;
else if (!strcmp(buf, "integer32")) temp_type = Type;
else if (!strcmp(buf, "input")) temp_type = Input;
else if (!strcmp(buf, "output")) temp_type = Output;

else if (!strcmp(buf, "eq")) temp_type = Equality;
else if (!strcmp(buf, "ne")) temp_type = NotEquality;
else if (!strcmp(buf, "%")) temp_type = Mod;

else if (!strcmp(buf, "le")) temp_type = Less;
else if (!strcmp(buf, "ge")) temp_type = Greater;

else if (!strcmp(buf, "&&")) temp_type = And;
else if (!strcmp(buf, "||")) temp_type = Or;

else if (!strcmp(buf, "if")) temp_type = If;
else if (!strcmp(buf, "else")) temp_type = Else;
else if (!strcmp(buf, "goto")) temp_type = Goto;
else if (!strcmp(buf, "for")) temp_type = For;
else if (!strcmp(buf, "to")) temp_type = To;
else if (!strcmp(buf, "downto")) temp_type = DownTo;
else if (!strcmp(buf, "do")) temp_type = Do;
else if (!strcmp(buf, "exit")) temp_type = Exit;
else if (!strcmp(buf, "while")) temp_type = While;
else if (!strcmp(buf, "continue")) temp_type = Continue;
else if (!strcmp(buf, "repeat")) temp_type = Repeat;
else if (!strcmp(buf, "until")) temp_type = Until;
if (temp_type == Unknown && TokenTable[NumberOfTokens - 1].type == Goto)
{
    temp_type = Identifier;
}
else if (buf[strlen(buf) - 1] == ':')
{
    buf[strlen(buf) - 1] = '\0';
    temp_type = Label;
}
else if (buf[0] == '_' && (strlen(buf) == 14))
{
    bool valid = true;

    if (!(buf[1] >= 'A' && buf[1] <= 'Z')) valid = false;

```

```

    for (int i = 2; i < 14; i++)
    {
        if (!(buf[i] >= 'a' && buf[i] <= 'z') && !(buf[i] >= '0' && buf[i] <= '9'))
        {
            valid = false;
            break;
        }
    }
    if (valid)
    {
        temp_type = Identifier;
    }
}
strcpy_s(TempToken.name, buf);
TempToken.type = temp_type;
TempToken.value = 0;
TempToken.line = line;
if (temp_type == Unknown)
{
    fprintf(errFile, "Lexical Error: line %d, lexem %s is Unknown\n", line, TempToken.name);
}
state = Finish;
break;
}

case Digit:
{
    buf[0] = ch;
    int j = 1;

    ch = getc(F);

    while ((ch <= '9' && ch >= '0') && j < 31)
    {
        buf[j++] = ch;
        ch = getc(F);
    }
    buf[j] = '\0';

    strcpy_s(TempToken.name, buf);
    TempToken.type = Number;
    TempToken.value = atoi(buf);
    TempToken.line = line;
    state = Finish;
    break;
}

case Separators:
{
    if (ch == '\n')
        line++;

    ch = getc(F);

```



```

    state = Start;
    break;
}

case SComment:
{
    ch = getc(F);
    if (ch == '*') {
        state = Comment;
    }
    else {
        strcpy_s(TempToken.name, "/");
        TempToken.type = Div;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
    }
    break;
}

case Comment:
{
    while (1)
    {
        ch = getc(F);

        if (ch == '*')
        {
            ch = getc(F);
            if (ch == '/')
            {
                state = Start;
                ch = getc(F);
                break;
            }
        }
        if (ch == EOF)
        {
            printf("Error: Comment not closed!\n");
            state = EndOfFile;
            break;
        }
    }
    break;
}

case Another:
{

```

```

switch (ch)
{

case '(':
{
    strcpy_s(TempToken.name, "(");
    TempToken.type = LBraket;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case ')':
{
    strcpy_s(TempToken.name, ")");
    TempToken.type = RBraket;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case ';':
{
    strcpy_s(TempToken.name, ";");
    TempToken.type = Semicolon;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case ',':
{
    strcpy_s(TempToken.name, ",");
    TempToken.type = Comma;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case ':':
{
    char next = getc(F);
    strcpy_s(TempToken.name, ":");
    TempToken.type = Colon;
    ungetc(next, F);
}

```

```

    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}
case '+':
{

    strcpy_s(TempToken.name, "+");
    TempToken.type = Add;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;

    break;
}

case '*':
{
    strcpy_s(TempToken.name, "*");
    TempToken.type = Mul;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case '%':
{
    strcpy_s(TempToken.name, "%");
    TempToken.type = Mod;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case '/':
{
    strcpy_s(TempToken.name, "/");
    TempToken.type = Div;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case '-':

```

```

{
    strcpy_s(TempToken.name, "-");
    TempToken.type = Sub;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}
case 'eq':
{
    strcpy_s(TempToken.name, "eq");
    TempToken.type = Equality;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case 'ne':
{
    strcpy_s(TempToken.name, "ne");
    TempToken.type = NotEquality;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case '!!':
{
    strcpy_s(TempToken.name, "!!");
    TempToken.type = Not;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}

case '!':
{
    ch = getc(F);
    if (ch == '!')
    {
        strcpy_s(TempToken.name, "!!");
        TempToken.type = Not;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
    }
}

```

```

        break;
    }
}

case '&':
{
    ch = getc(F);
    if (ch == '&')
    {
        strcpy_s(TempToken.name, "&&");
        TempToken.type = And;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
    }
}

case '|':
{
    ch = getc(F);
    if (ch == '|')
    {
        strcpy_s(TempToken.name, "||");
        TempToken.type = Or;
        TempToken.value = 0;
        TempToken.line = line;
        ch = getc(F);
        state = Finish;
        break;
    }
}

case '<':
{
    ch = getc(F);
    if (ch == '=')
    {
        ch = getc(F);

        if (ch == '=')
        {
            strcpy_s(TempToken.name, "<==");
            TempToken.type = Assign;
            TempToken.value = 0;
            TempToken.line = line;
            ch = getc(F);
            state = Finish;
        }
    }
    break;
}

```

```

default:
{
    TempToken.name[0] = ch;
    TempToken.name[1] = '\0';
    TempToken.type = Unknown;
    TempToken.value = 0;
    TempToken.line = line;
    ch = getc(F);
    state = Finish;
    break;
}
}
}
}
}
}
}
}
}

```

```

void PrintTokens(Token TokenTable[], unsigned int TokensNum)
{
    char type_tokens[16];
    printf("\n\n-----\n");
    printf("|          TOKEN TABLE                      |\n");
    printf("-----\n");
    printf("| line number |   token   |   value   | token code | type of token |\n");
    printf("-----");
    for (unsigned int i = 0; i < TokensNum; i++)
    {
        switch (TokenTable[i].type)
        {
            case Mainprogram:
                strcpy_s(type_tokens, "MainProgram");
                break;
            case ProgramName:
                strcpy_s(type_tokens, "ProgramName");
                break;
            case StartProgram:
                strcpy_s(type_tokens, "StartProgram");
                break;
            case Variable:
                strcpy_s(type_tokens, "Variable");
                break;
            case Type:
                strcpy_s(type_tokens, "Integer");
                break;
            case Identifier:
                strcpy_s(type_tokens, "Identifier");
                break;
            case EndProgram:
                strcpy_s(type_tokens, "EndProgram");
                break;
            case Input:
                strcpy_s(type_tokens, "Input");

```

```

        break;
case Output:
    strcpy_s(type_tokens, "Output");
    break;
case If:
    strcpy_s(type_tokens, "If");
    break;
case Then:
    strcpy_s(type_tokens, "Then");
    break;
case Else:
    strcpy_s(type_tokens, "Else");
    break;
case Assign:
    strcpy_s(type_tokens, "Assign");
    break;
case Add:
    strcpy_s(type_tokens, "Add");
    break;
case Sub:
    strcpy_s(type_tokens, "Sub");
    break;
case Mul:
    strcpy_s(type_tokens, "Mul");
    break;
case Div:
    strcpy_s(type_tokens, "Div");
    break;
case Mod:
    strcpy_s(type_tokens, "Mod");
    break;
case Equality:
    strcpy_s(type_tokens, "Equality");
    break;
case NotEquality:
    strcpy_s(type_tokens, "NotEquality");
    break;
case Greate:
    strcpy_s(type_tokens, "Greate");
    break;
case Less:
    strcpy_s(type_tokens, "Less");
    break;
case Not:
    strcpy_s(type_tokens, "Not");
    break;
case And:
    strcpy_s(type_tokens, "And");
    break;
case Or:
    strcpy_s(type_tokens, "Or");
    break;
case LBracket:

```

```

        strcpy_s(type_tokens, "LBracket");
        break;
case RBracket:
    strcpy_s(type_tokens, "RBracket");
    break;
case Number:
    strcpy_s(type_tokens, "Number");
    break;
case Semicolon:
    strcpy_s(type_tokens, "Semicolon");
    break;
case Comma:
    strcpy_s(type_tokens, "Comma");
    break;
case Goto:
    strcpy_s(type_tokens, "Goto");
    break;
case For:
    strcpy_s(type_tokens, "For");
    break;
case To:
    strcpy_s(type_tokens, "To");
    break;
case DownTo:
    strcpy_s(type_tokens, "DownTo");
    break;
case Do:
    strcpy_s(type_tokens, "Do");
    break;
case While:
    strcpy_s(type_tokens, "While");
    break;
case Exit:
    strcpy_s(type_tokens, "Exit");
    break;
case Continue:
    strcpy_s(type_tokens, "Continue");
    break;
case End:
    strcpy_s(type_tokens, "End");
    break;
case Repeat:
    strcpy_s(type_tokens, "Repeat");
    break;
case Until:
    strcpy_s(type_tokens, "Until");
    break;
case Label:
    strcpy_s(type_tokens, "Label");
    break;
case Unknown:
default:
    strcpy_s(type_tokens, "Unknown");

```



```

        break;
    }

    printf("\n|%12d|%16s|%11d|%11d|%-13s\n",
        TokenTable[i].line,
        TokenTable[i].name,
        TokenTable[i].value,
        TokenTable[i].type,
        type_tokens);
    printf("-----");
}
printf("\n");
}

void PrintTokensToFile(char* FileName, Token TokenTable[], unsigned int TokensNum)
{
    FILE* F;
    if ((fopen_s(&F, FileName, "wt")) != 0)
    {
        printf("Error: Can not create file: %s\n", FileName);
        return;
    }
    char type_tokens[16];
    fprintf(F, "-----\n");
    fprintf(F, "|          TOKEN TABLE          |\n");
    fprintf(F, "-----\n");
    fprintf(F, "| line number |   token   |  value  | token code | type of token |\n");
    fprintf(F, "-----");
    for (unsigned int i = 0; i < TokensNum; i++)
    {
        switch (TokenTable[i].type)
        {
            {
            case Mainprogram:
                strcpy_s(type_tokens, "MainProgram");
                break;
            case ProgramName:
                strcpy_s(type_tokens, "ProgramName");
                break;
            case StartProgram:
                strcpy_s(type_tokens, "StartProgram");
                break;
            case Variable:
                strcpy_s(type_tokens, "Variable");
                break;
            case Type:
                strcpy_s(type_tokens, "Integer");
                break;
            case Identifier:
                strcpy_s(type_tokens, "Identifier");
                break;
            case EndProgram:
                strcpy_s(type_tokens, "EndProgram");
                break;

```

```

case Input:
    strcpy_s(type_tokens, "Input");
    break;
case Output:
    strcpy_s(type_tokens, "Output");
    break;
case If:
    strcpy_s(type_tokens, "If");
    break;
case Then:
    strcpy_s(type_tokens, "Then");
    break;
case Else:
    strcpy_s(type_tokens, "Else");
    break;
case Assign:
    strcpy_s(type_tokens, "Assign");
    break;
case Add:
    strcpy_s(type_tokens, "Add");
    break;
case Sub:
    strcpy_s(type_tokens, "Sub");
    break;
case Mul:
    strcpy_s(type_tokens, "Mul");
    break;
case Div:
    strcpy_s(type_tokens, "Div");
    break;
case Mod:
    strcpy_s(type_tokens, "Mod");
    break;
case Equality:
    strcpy_s(type_tokens, "Equality");
    break;
case NotEquality:
    strcpy_s(type_tokens, "NotEquality");
    break;
case Greate:
    strcpy_s(type_tokens, "Greate");
    break;
case Less:
    strcpy_s(type_tokens, "Less");
    break;
case Not:
    strcpy_s(type_tokens, "Not");
    break;
case And:
    strcpy_s(type_tokens, "And");
    break;
case Or:
    strcpy_s(type_tokens, "Or");

```

```

        break;
case LBracket:
    strcpy_s(type_tokens, "LBracket");
    break;
case RBracket:
    strcpy_s(type_tokens, "RBracket");
    break;
case Number:
    strcpy_s(type_tokens, "Number");
    break;
case Semicolon:
    strcpy_s(type_tokens, "Semicolon");
    break;
case Comma:
    strcpy_s(type_tokens, "Comma");
    break;
case Goto:
    strcpy_s(type_tokens, "Goto");
    break;
case For:
    strcpy_s(type_tokens, "For");
    break;
case To:
    strcpy_s(type_tokens, "To");
    break;
case DownTo:
    strcpy_s(type_tokens, "DownTo");
    break;
case Do:
    strcpy_s(type_tokens, "Do");
    break;
case While:
    strcpy_s(type_tokens, "While");
    break;
case Exit:
    strcpy_s(type_tokens, "Exit");
    break;
case Continue:
    strcpy_s(type_tokens, "Continue");
    break;
case End:
    strcpy_s(type_tokens, "End");
    break;
case Repeat:
    strcpy_s(type_tokens, "Repeat");
    break;
case Until:
    strcpy_s(type_tokens, "Until");
    break;
case Label:
    strcpy_s(type_tokens, "Label");
    break;
case Unknown:

```

```

        default:
            strcpy_s(type_tokens, "Unknown");
            break;
    }

    fprintf(F, "\n%12d %16s %11d %11d | %-13s \n",
        TokenTable[i].line,
        TokenTable[i].name,
        TokenTable[i].value,
        TokenTable[i].type,
        type_tokens);
    fprintf(F, "-----");
}
fclose(F);
}

```

main.cpp

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "translator.h"

// таблиця лексем
Token* TokenTable;
// кількість лексем
unsigned int TokensNum;

// таблиця ідентифікаторів
Id* IdTable;
// кількість ідентифікаторів
unsigned int IdNum;

// Function to validate file extension
int hasValidExtension(const char* fileName, const char* extension)
{
    const char* dot = strchr(fileName, '.');
    if (!dot || dot == fileName) return 0; // No extension found
    return strcmp(dot, extension) == 0;
}

int main(int argc, char* argv[])
{
    // виділення пам'яті під таблицю лексем
    TokenTable = new Token[MAX_TOKENS];

    // виділення пам'яті під таблицю ідентифікаторів
    IdTable = new Id[MAX_IDENTIFIER];

    char InputFile[32] = "";

    FILE* InFile;

```

```

if (argc != 2)
{
    printf("Input file name: ");
    gets_s(InputFile);
}
else
{
    strcpy_s(InputFile, argv[1]);
}

// Check if the input file has the correct extension
if (!hasValidExtension(InputFile, ".s18"))
{
    printf("Error: Input file has invalid extension.\n");
    return 1;
}

if ((fopen_s(&InFile, InputFile, "rt")) != 0)
{
    printf("Error: Cannot open file: %s\n", InputFile);
    return 1;
}

char NameFile[32] = "";
int i = 0;
while (InputFile[i] != '.' && InputFile[i] != '\0')
{
    NameFile[i] = InputFile[i];
    i++;
}
NameFile[i] = '\0';

char TokenFile[32];
strcpy_s(TokenFile, NameFile);
strcat_s(TokenFile, ".token");

char ErrFile[32];
strcpy_s(ErrFile, NameFile);
strcat_s(ErrFile, "_errors.txt");

FILE* errFile;
if (fopen_s(&errFile, ErrFile, "w") != 0)
{
    printf("Error: Cannot open file for writing: %s\n", ErrFile);
    return 1;
}

TokensNum = GetTokens(InFile, TokenTable, errFile);

PrintTokensToFile(TokenFile, TokenTable, TokensNum);
fclose(InFile);

printf("\nLexical analysis completed: %d tokens. List of tokens in the file %s\n", TokensNum, TokenFile);

```

```

printf("\nList of errors in the file %s\n", ErrFile);

Parser(errFile);
fclose(errFile);
ASTNode* ASTTree = ParserAST();

char AST[32];
strcpy_s(AST, NameFile);
strcat_s(AST, ".ast");
// Open output file
FILE* ASTFile;
fopen_s(&ASTFile, AST, "w");
if (!ASTFile)
{
    printf("Failed to open output file.\n");
    exit(1);
}
PrintASTToFile(ASTTree, 0, ASTFile);
printf("\nAST has been created and written to %s.\n", AST);

char OutputFile[32];
strcpy_s(OutputFile, NameFile);
strcat_s(OutputFile, ".c");

FILE* outFile;
fopen_s(&outFile, OutputFile, "w");
if (!outFile)
{
    printf("Failed to open output file.\n");
    exit(1);
}
// генерація вихідного C коду
generateCCode(outFile);
printf("\nC code has been generated and written to %s.\n", OutputFile);

fclose(outFile);

fopen_s(&outFile, OutputFile, "r");
char ExecutableFile[32];
strcpy_s(ExecutableFile, NameFile);
strcat_s(ExecutableFile, ".exe");
compile_to_exe(OutputFile, ExecutableFile);

char OutputFileFromAST[32];
strcpy_s(OutputFileFromAST, NameFile);
strcat_s(OutputFileFromAST, "_fromAST.c");

FILE* outFileFromAST;
fopen_s(&outFileFromAST, OutputFileFromAST, "w");
if (!outFileFromAST)
{
    printf("Failed to open output file.\n");
    exit(1);
}

```

```

    }
    generateCodeFromAST(ASTree, outFileFromAST);
    printf("\nC code has been generated and written to %s.\n", OutputFileFromAST);

    fclose(outFileFromAST);

    fopen_s(&outFileFromAST, OutputFileFromAST, "r");
    char ExecutableFileFromAST[32];
    strcpy_s(ExecutableFileFromAST, NameFile);
    strcat_s(ExecutableFileFromAST, "_fromAST.exe");
    compile_to_exe(OutputFileFromAST, ExecutableFileFromAST);

    // Close the file
    _fcloseall();

    destroyTree(ASTree);

    delete[] TokenTable;
    delete[] IdTable;

    return 0;
}

parser.cpp
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "translator.h"
#include <iostream>
#include <string>

// таблиця лексем
extern Token* TokenTable;
// кількість лексем
extern unsigned int TokensNum;

// таблиця ідентифікаторів
extern Id* IdTable;
// кількість ідентифікаторів
extern unsigned int IdNum;

static int pos = 0;

// набір функцій для рекурсивного спуску
// на кожне правило - окрема функція
void program(FILE* errFile);
void variable_declaration(FILE* errFile);
void variable_list(FILE* errFile);
void program_body(FILE* errFile);
void statement(FILE* errFile);
void assignment(FILE* errFile);
void arithmetic_expression(FILE* errFile);
void term(FILE* errFile);

```

```

void factor(FILE* errFile);
void input(FILE* errFile);
void output(FILE* errFile);
void conditional(FILE* errFile);

void goto_statement(FILE* errFile);
void label_statement(FILE* errFile);
void for_to_do(FILE* errFile);
void for_downto_do(FILE* errFile);
void while_statement(FILE* errFile);
void repeat_until(FILE* errFile);

void logical_expression(FILE* errFile);
void and_expression(FILE* errFile);
void comparison(FILE* errFile);
void compound_statement(FILE* errFile);
std::string TokenTypeToString(TypeOfTokens type);

unsigned int IdIdentification(Id IdTable[], Token TokenTable[], unsigned int tokenCount, FILE* errFile);

void Parser(FILE* errFile)
{
    program(errFile);
    fprintf(errFile, "\nNo errors found.\n");
}

void match(TypeOfTokens expectedType, FILE* errFile)
{
    if (TokenTable[pos].type == expectedType)
        pos++;
    else
    {
        fprintf(errFile, "\nSyntax error in line %d : another type of lexeme was expected.\n", TokenTable[pos].line);
        fprintf(errFile, "\nSyntax error: type %s\n", TokenTypeToString(TokenTable[pos].type).c_str());
        fprintf(errFile, "Expected Type: %s ", TokenTypeToString(expectedType).c_str());
        exit(10);
    }
}

void program(FILE* errFile)
{
    match(Mainprogram, errFile);
    match(StartProgram, errFile);
    match(Variable, errFile);
    variable_declaration(errFile);
    match(Semicolon, errFile);
    program_body(errFile);
    match(EndProgram, errFile);
}

void variable_declaration(FILE* errFile)
{
    if (TokenTable[pos].type == Type)

```



```

    {
        pos++;
        variable_list(errFile);
    }
}

void variable_list(FILE* errFile)
{
    match(Identifier, errFile);
    while (TokenTable[pos].type == Comma)
    {
        pos++;
        match(Identifier, errFile);
    }
}

void program_body(FILE* errFile)
{
    do
    {
        statement(errFile);
    } while (TokenTable[pos].type != EndProgram);
}

void statement(FILE* errFile)
{
    switch (TokenTable[pos].type)
    {
        case Input: input(errFile); break;
        case Output: output(errFile); break;
        case If: conditional(errFile); break;
        case Label: label_statement(errFile); break;
        case StartProgram: compound_statement(errFile); break;
        case Goto: goto_statement(errFile); break;
        case For:
        {
            int temp_pos = pos + 1;
            while (TokenTable[temp_pos].type != To && TokenTable[temp_pos].type != DownTo && temp_pos <
TokensNum)
            {
                temp_pos++;
            }
            if (TokenTable[temp_pos].type == To)
            {
                for_to_do(errFile);
            }
            else if (TokenTable[temp_pos].type == DownTo)
            {
                for_downto_do(errFile);
            }
            else
            {
                printf("Error: Expected 'To' or 'DownTo' after 'For'\n");
            }
        }
    }
}

```

```

    }
    break;
}
case While: while_statement(errFile); break;
case Exit: pos += 2; break;
case Continue: pos += 2; break;
case Repeat: repeat_until(errFile); break;
default: assignment(errFile); break;
}
}

void assignment(FILE* errFile)
{
    match(Identifier, errFile);
    match(Assign, errFile);
    arithmetic_expression(errFile);
    match(Semicolon, errFile);
}

void arithmetic_expression(FILE* errFile)
{
    term(errFile);
    while (TokenTable[pos].type == Add || TokenTable[pos].type == Sub)
    {
        pos++;
        term(errFile);
    }
}

void term(FILE* errFile)
{
    factor(errFile);
    while (TokenTable[pos].type == Mul || TokenTable[pos].type == Div || TokenTable[pos].type == Mod)
    {
        pos++;
        factor(errFile);
    }
}

void factor(FILE* errFile)
{
    if (TokenTable[pos].type == Identifier)
    {
        match(Identifier, errFile);
    }
    else
        if (TokenTable[pos].type == Number)
        {
            match(Number, errFile);
        }
    else
        if (TokenTable[pos].type == LBracket)
        {

```

```

        match(LBraket, errFile);
        arithmetic_expression(errFile);
        match(RBraket, errFile);
    }
    else
    {
        printf("\nSyntax error in line %d : A multiplier was expected.\n", TokenTable[pos].line);
        exit(11);
    }
}

void input(FILE* errFile)
{
    match(Input, errFile);
    match(Identifier, errFile);
    match(Semicolon, errFile);
}

void output(FILE* errFile)
{
    match(Output, errFile);
    if (TokenTable[pos].type == Sub)
    {
        pos++;
        if (TokenTable[pos].type == Number)
        {
            match(Number, errFile);
        }
    }
    else
    {
        arithmetic_expression(errFile);
    }
    match(Semicolon, errFile);
}

void conditional(FILE* errFile)
{
    match(If, errFile);
    logical_expression(errFile);
    statement(errFile);
    if (TokenTable[pos].type == Else)
    {
        pos++;
        statement(errFile);
    }
}

void goto_statement(FILE* errFile)
{
    match(Goto, errFile);
    if (TokenTable[pos].type == Identifier)
    {

```

```

        pos++;
        match(Semicolon, errFile);
    }
    else
    {
        printf("Error: Expected a label after 'goto' at line %d.\n", TokenTable[pos].line);
        exit(1);
    }
}

```

```

void label_statement(FILE* errFile)
{
    match(Label, errFile);
}

```

```

void for_to_do(FILE* errFile)
{
    match(For, errFile);
    match(Identifier, errFile);
    match(Assign, errFile);
    arithmetic_expression(errFile);
    match(To, errFile);
    arithmetic_expression(errFile);
    match(Do, errFile);
    statement(errFile);
}

```

```

void for_downto_do(FILE* errFile)
{
    match(For, errFile);
    match(Identifier, errFile);
    match(Assign, errFile);
    arithmetic_expression(errFile);
    match(DownTo, errFile);
    arithmetic_expression(errFile);
    match(Do, errFile);
    statement(errFile);
}

```

```

void while_statement(FILE* errFile)
{
    match(While, errFile);
    logical_expression(errFile);

    while (1)
    {
        if (TokenTable[pos].type == End)
        {
            pos++;
            match(While, errFile);
            break;
        }
    }
}

```

```

        else
        {
            statement(errFile);
            if (TokenTable[pos].type == Semicolon)
            {
                pos++;
            }
        }
    }
}

```

```

void repeat_until(FILE* errFile)
{
    match(Repeat, errFile);
    statement(errFile);
    match(Until, errFile);
    logical_expression(errFile);
}

```

```

void logical_expression(FILE* errFile)
{
    and_expression(errFile);
    while (TokenTable[pos].type == Or)
    {
        pos++;
        and_expression(errFile);
    }
}

```

```

void and_expression(FILE* errFile)
{
    comparison(errFile);
    while (TokenTable[pos].type == And)
    {
        pos++;
        comparison(errFile);
    }
}

```

```

void comparison(FILE* errFile)
{
    if (TokenTable[pos].type == Not)
    {
        pos++;
        match(LBracket, errFile);
        logical_expression(errFile);
        match(RBracket, errFile);
    }
    else
    if (TokenTable[pos].type == LBracket)
    {
        pos++;
        logical_expression(errFile);
    }
}

```

```

        match(RBraket, errFile);
    }
    else
    {
        arithmetic_expression(errFile);
        if (TokenTable[pos].type == Greate || TokenTable[pos].type == Less ||
            TokenTable[pos].type == Equality || TokenTable[pos].type == NotEquality)
        {
            pos++;
            arithmetic_expression(errFile);
        }
        else
        {
            printf("\nSyntax error: A comparison operation is expected.\n");
            exit(12);
        }
    }
}

void compound_statement(FILE* errFile)
{
    match(StartProgram, errFile);
    program_body(errFile);
    match(EndProgram, errFile);
}

unsigned int IdIdentification(Id IdTable[], Token TokenTable[], unsigned int tokenCount, FILE* errFile)
{
    unsigned int idCount = 0;
    unsigned int i = 0;

    while (TokenTable[i++].type != Variable);

    if (TokenTable[i++].type == Type)
    {
        while (TokenTable[i].type != Semicolon)
        {
            if (TokenTable[i].type == Identifier)
            {
                {
                    int yes = 0;
                    for (unsigned int j = 0; j < idCount; j++)
                    {
                        if (!strcmp(TokenTable[i].name, IdTable[j].name))
                        {
                            yes = 1;
                            break;
                        }
                    }
                }
                if (yes == 1)
                {
                    printf("\nidentifier \"%s\" is already declared !\n", TokenTable[i].name);
                    return idCount;
                }
            }
        }
    }
}

```

```

        if (idCount < MAX_IDENTIFIER)
        {
            strcpy_s(IdTable[idCount++].name, TokenTable[i++].name);
        }
        else
        {
            printf("\nToo many identifiers !\n");
            return idCount;
        }
    }
    else
        i++;
}

for (; i < tokenCount; i++)
{
    if (TokenTable[i].type == Identifier && TokenTable[i + 1].type != Colon)
    {
        int yes = 0;
        for (unsigned int j = 0; j < idCount; j++)
        {
            if (!strcmp(TokenTable[i].name, IdTable[j].name))
            {
                yes = 1;
                break;
            }
        }
        if (yes == 0)
        {
            if (idCount < MAX_IDENTIFIER)
            {
                strcpy_s(IdTable[idCount++].name, TokenTable[i].name);
            }
            else
            {
                printf("\nToo many identifiers!\n");
                return idCount;
            }
        }
    }
}

return idCount;
}

std::string TokenTypeToString(TypeOfTokens type)
{
    switch (type)
    {
        case Mainprogram: return "Mainprogram";
    }
}

```

```

case StartProgram: return "StartProgram";
case Variable: return "Variable";
case Type: return "Type";
case EndProgram: return "EndProgram";
case Input: return "Input";
case Output: return "Output";
case If: return "If";
case Else: return "Else";
case Goto: return "Goto";
case Label: return "Label";
case For: return "For";
case To: return "To";
case DownTo: return "DownTo";
case Do: return "Do";
case While: return "While";
case Exit: return "Exit";
case Continue: return "Continue";
case End: return "End";
case Repeat: return "Repeat";
case Until: return "Until";
case Identifier: return "Identifier";
case Number: return "Number";
case Assign: return "Assign";
case Add: return "Add";
case Sub: return "Sub";
case Mul: return "Mul";
case Div: return "Div";
case Mod: return "Mod";
case Equality: return "Equality";
case NotEquality: return "NotEquality";
case Greate: return "Greate";
case Less: return "Less";
case Not: return "Not";
case And: return "And";
case Or: return "Or";
case LBraket: return "LBraket";
case RBraket: return "RBraket";
case Semicolon: return "Semicolon";
case Colon: return "Colon";
case Comma: return "Comma";
case Unknown: return "Unknown";
default: return "InvalidType";
}
}

```

translator.h

```
#pragma once
```

```
#define MAX_TOKENS 1000
#define MAX_IDENTIFIER 10
```



```
// перерахування, яке описує всі можливі типи лексем
enum TypeOfTokens
{
    Mainprogram,
    ProgramName,
    Variable,
    StartProgram,
    Type,
    EndProgram,
    Input,
    Output,

    If,
    Then,
    Else,

    Goto,
    Label,

    For,
    To,
    DownTo,
    Do,

    While,
    Exit,
    Continue,
    End,

    Repeat,
    Until,

    Identifier,
    Number,
    Assign,
    Add,
    Sub,
    Mul,
    Div,
    Mod,
    Equality,
    NotEquality,
    Greate,
    Less,
    Not,
    And,
    Or,
    LBraket,
    RBraket,
    Semicolon,
    Colon,
    Comma,
    Unknown
}
```

```

};

// структура для зберігання інформації про лексему
struct Token
{
    char name[32];    // 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[32];
};

// 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
};

// перерахування, яке описує всі можливі вузли абстрактного синтаксичного дерева
enum TypeOfNodes
{
    program_node,
    var_node,
    input_node,
    output_node,

    if_node,
    then_node,

    goto_node,
    label_node,

    for_to_node,
    for_downto_node,

    while_node,
    exit_while_node,
    continue_while_node,

    repeat_until_node,

```

```

    id_node,
    num_node,
    assign_node,
    add_node,
    sub_node,
    mul_node,
    div_node,
    mod_node,
    or_node,
    and_node,
    not_node,
    cmp_node,
    statement_node,
    compount_node
};

// структура, яка описує вузол абстрактного синтаксичного дерева (AST)
struct ASTNode
{
    TypeOfNodes nodetype; // Тип вузла
    char name[32];         // Ім'я вузла
    struct ASTNode* left;  // Лівий нащадок
    struct ASTNode* right; // Правий нащадок
};

// 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[], FILE* errFile);

// 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);

// синтаксичний аналіз методом рекурсивного спуску
// вхідні дані - глобальна таблиця лексем TokenTable
void Parser(FILE* errFile);

// функція синтаксичного аналізу і створення абстрактного синтаксичного дерева
ASTNode* ParserAST();

// функція знищення дерева
void destroyTree(ASTNode* root);

// 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);

// Рекурсивна функція для генерації коду з AST

```

```
void generateCodefromAST(ASTNode* node, FILE* output);
```

```
// функція для генерації коду  
void generateCCode(FILE* outFile);
```

```
void compile_to_exe(const char* source_file, const char* output_file);
```

