Міністерство освіти і науки України  
Національний університет "Львівська Політехніка"  
Кафедра ЕОМ



Пояснювальна записка  
до курсового проєкту "СИСТЕМНЕ ПРОГРАМУВАННЯ"

на тему: “РОЗРОБКА СИСТЕМНИХ ПРОГРАМНИХ МОДУЛІВ ТА КОМПОНЕНТ

СИСТЕМ ПРОГРАМУВАННЯ”

Індивідуальне завдання 13

“РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ”

Виконав студент групи КІ-309:

Маковський В. А.

Перевірив:

Козак Н.Б.

Львів-2025

ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ

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

*файл з лексемами;*

*файл з повідомленнями про помилки (або про їх відсутність);*

*файл на мові С або асемблера;*

*об’єктний файл; виконуваний файл.*

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

Деталізація завдання на проектування:

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

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

1. За допомогою оператора вводу можна зчитати з клавіатури значення змінної; за допомогою оператора виводу можна вивести на екран значення змінної, виразу чи цілої константи.
2. В кожному завданні обов’язковим є оператор присвоєння за допомогою якого можна реалізувати обчислення виразів з використанням заданих операцій і операції круглі дужки (); у якості операндів можуть бути цілі константи, змінні, а також інші вирази.
3. В кожному завданні обов’язковим є оператор типу “блок” (складений оператор), його вигляд має бути таким, як і блок тіла програми.
4. Необхідно реалізувати задані варіантом оператори, синтаксис операторів наведено у таблиці 1.1. Синтаксис вхідної мови має забезпечити реалізацію обчислень лінійних алгоритмів, алгоритмів з розгалуженням і циклічних алгоритмів. Опис формальної мови студент погоджує з викладачем.
5. Оператори можуть бути довільної вкладеності і в будь-якій послідовності.
6. Для перевірки роботи розробленого транслятора, необхідно написати три тестові програми на вхідній мові програмування.

Деталізований опис власної мови програмування:

Блок тіла програми: **STARTPROGRAM**

**VARIABLE…**

STARTBLOK

ENDBLOCK

Оператори вводу-виводу: **READ, WRITE**

Оператор присвоєння: <**==**

Оператори:

**IF [-ELSE ]**

**GOTO**

**FOR-TO**

**FOR-DOWNTO**

**WHILE**

**REPEAT-UNTIL**

Регістр ідентифікаторів: **Low8, перший символ \_**

Операції:

* арифметичні:**ADD, SUB, MUL, DIV, MOD**
* порівняння: **EQ, NE, LT, GT**
* логічні: **!, &, |**

Тип даних: **INT\_4**

Коментар: **#\* … \*#**

Програма на вхідній мові програмування має починатись з ключового слова **STARTPROGRAM**, після чого йде розділ опису змінних **VARIABLE**.  
Далі ідкривається блок коду програми з ключового слова **STARTBLOK**, а оператори програми розміщуються між ключовими словами **STARTBLOK, ENDBLOK .**Операторів є 9: оператор вводу даних **READ**, оператор виводу даних **WRITE**, 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 **ADD ,SUB , MUL, DIV, MOD.** Операндами можуть бути змінні, цілі додатні константи і інші вирази, взяті в дужки. В умовному операторі використовуються логічні вирази, допустимі такі операції порівняння **EQ , NE, LT, GT** і такі логічні операції **! , & , |**. Ідентифікатори (імена змінних) можуть бути довжиною до 6-х символів, складаються лише з малих та великих латинських літер або цифр та починаються з символу **\_**. Тип даних лише один – **INT\_4**, при оголошення декількох змінних вони записуються через кому, вкінці опису змінних ставиться символ крапка з комою **;**. Коментарі починаються з знаків **#\* … \*#.** .

Приклади оголошення змінних:

VARIABLE INT\_4 \_aaaaaaa, \_bbbbbbb, \_xxxxxxx, \_yyyyyyy;  
VARIABLE INT\_4 \_aaaaaaa, \_bbbbbbb, \_ccccccc;

VARIABLE INT\_4 \_aaaaaaa, \_bbbbbbb, \_xxxxxxx, \_iiiiiii, \_jjjjjjj;

Приклади ідентифікаторів:

\_aaaaaaa, \_bbbbbbb, \_xxxxxxx, \_yyyyyyy

Приклади арифметичних виразів:

0 SUB 75

\_aaaaaaa SUB \_68

\_bbbbbbb MUL \_ aaaaaaa MUL \_ xxxxxxx ADD 76

\_ccccccc MUL (\_aaaaaaa ADD \_bbbbbbb) SUB 50 DIV \_bbbbbbb

Приклади логічних виразів:

\_aaaaaaa GT \_ bbbbbbb

\_aaaaaaa LT \_ bbbbbbb & \_ aaaaaaa GT \_ ccccccc

! (\_aaaaaaa LT \_ ccccccc)

\_ aaaaaaa NE \_ bbbbbbb

\_ xxxxxxx LT \_ aaaaaaa | \_ xxxxxxx GT \_ bbbbbbb

АНОТАЦІЯ

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

У рамках курсового проєкту створено транслятор для вхідної мови програмування. Граматика мови описана за допомогою розширеної нотації Бекуса-Наура. Реалізовано лексичний, синтаксичний та семантичний аналізатори для перевірки програми на наявність помилок. Вхідна програма перетворюється в абстрактне синтаксичне дерево, яке використовується генератором для побудови вихідного коду мовою програмування C. Розроблене програмне забезпечення успішно протестовано на тестових прикладах, підтверджуючи його коректність і надійність.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**ЗМІСТ**

ВСТУП 8

1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ 9
2. [ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ 10](#bookmark19)
   1. [Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура. 10](#bookmark21)
   2. [Опис термінальних символів та ключових слів 12](#bookmark26)
3. [РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ 13](#bookmark29)
   1. [Вибір технології програмування 13](#bookmark32)
   2. [Проектування таблиць транслятора та вибір структур даних 13](#bookmark35)
   3. [Розробка лексичного аналізатора 14](#bookmark38)
      1. Розробка алгоритму роботи лексичного аналізатора 16
      2. Опис програми реалізації лексичного аналізатора 18
   4. [Розробка синтаксичного та семантичного аналізатора 19](#bookmark43)
      1. Розробка дерева граматичного розбору 21
      2. [Розробка алгоритму роботи синтаксичного і семантичного аналізатора. 22](#bookmark46)
      3. [Опис програми реалізації синтаксичного та семантичного аналізатора. . 26](#bookmark47)
   5. [Розробка генератора коду 27](#bookmark49)
      1. Розробка алгоритму роботи генератора коду 28
      2. Опис програми реалізації генератора коду 31
4. [НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА ... 36](#bookmark53)
   1. [Опис інтерфейсу та інструкції користувачу 36](#bookmark58)
   2. [Виявлення лексичних і синтаксичних помилок 37](#bookmark61)
   3. [Перевірка роботи транслятора за допомогою тестових задач 39](#bookmark64)

ВИСНОВКИ 43

[СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ 44](#bookmark69)

[ДОДАТКИ 45](#bookmark72)

**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 С. 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.

Багатопрохідну 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, багатопрохідні.
* 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.

Багатопрохідний дизайн: забезпечує глибоку оптимізацію та підтримує складні мови.

1. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ
   1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура.

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

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

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

<символ> = <вираз, що містить символи>

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

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

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

program = "**STARTPROGRAM**", {"**VARIABLE**", variable\_declaration, "**;**"}, "**STARTBLOCK**", {statement}, "**ENDBLOK**";

variable\_declaration = "**INT\_4**", variable\_list;

variable\_list = identifier, {"**,**", **identifier**};

identifier = "**\_**", 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 = "**READ**", identifier;

output\_statement = "**WRITE**", arithmetic\_expression;

arithmetic\_expression = low\_priority\_expression {low\_priority\_operator, low\_priority\_expression};

low\_priority\_operator = "**ADD**" | "**SUB**";

low\_priority\_expression = middle\_priority\_expression {middle\_priority\_operator, middle\_priority\_expression};

middle\_priority\_operator = "**MUL**" | "**DIV**" | "**MOD**";

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 = identifier, "<**==**", arithmetic\_expression;

if\_else\_statement = **"IF"**, "(", logical\_expression, ")", {statement\_in\_while},”;”, [ "**ELSE**", {statement\_in\_while},”;”];

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**" | "**LT**" | "**GT**";

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;

* 1. Опис термінальних символів та ключових слів.

Визначаємо термінальні символи і ключові слова:

* **STARTPROGRAM** - початок програми
* **VARIABLE** - оголошення змінних
* **STARTBLOCK** - початок блоку
* **ENDBLOK** - кінець програми
* **INT\_4** - тип даних
* **READ** - оператор вводу
* **WRITE** - оператор виводу
* **IF, ELSE** - умовний оператор
* **GOTO** – 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
* **ADD** - додавання
* **SUB** - віднімання
* **MUL** - множення
* **DIV** - ділення
* **MOD** - остача від ділення
* **LT** - більше
* **GT** - менше
* **EQ** - рівність
* **NE** - нерівність
* **!** - заперечення
* **&** - логічне І
* **|** - логічне АБО
* **,** - 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

1. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ  
   ПРОГРАМУВАННЯ
   1. Вибір технології програмування.

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

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

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

* 1. Проектування таблиць транслятора та вибір структур даних.

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

enum TypeOfToken {

StartProgram,

StartBlock,

Variable,

Type,

EndBlock,

Input,

Output,

If,

Else,

Goto,

For,

To,

Downto,

Do,

While,

Continue,

Exit,

End,

Repeat,

Until,

Identifier,

Number,

Float,

Assign,

Add,

Sub,

Mul,

Div,

Mod,

Equality,

NotEquality,

Greate,

Less,

Not,

And,

Or,

LBracket,

RBracket,

Semicolon,

Colon,

Comma,

Unknown\_

};

struct Token {

char name[16];

int value;

int line;

enum TypeOfToken type;

};

struct id {

char name[16];

};  
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

* 1. Розробка лексичного аналізатора.

Основна задача лексичного аналізу - розбити вихідний текст, що складається з послідовності символів, на послідовність слів, або лексем, тобто виділити ці слова з безперервної послідовності символів. Всі символи вхідної послідовності з цієї точки зору розділяються на символи, що належать яким-небудь лексемам, і символи, що розділяють лексеми. В цьому випадку використовуються звичайні засоби обробки рядків. Вхідна програма проглядається послідовно з початку до кінця. Базові елементи, або лексичнї одиниці, розділяються пробілами, знаками операцїй i 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 (STARTPROGRAM, VARIABLE, STARTBLOK, ENDBLOK, INT\_4, READ, WRITE , IF, ELSE, FOR, TO, DOWNTO, DO, WHILE, END, REPEAT, UNTIL, ADD, SUB,MUL,DIV,MOD, GT, LT)

* Ідентифікатори (починається з символу \_, далі маленька або цифра максимум 8 символів)
* Числові константи (ціле число)
* Оператор присвоєння (<==)
* Розділювачі (,, ;)
* Дужки (**(,** **)**)
* Невідома лексема (символи і ланцюжки символів, які не підпадають під вищеописані правила).
  + 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 8-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**.

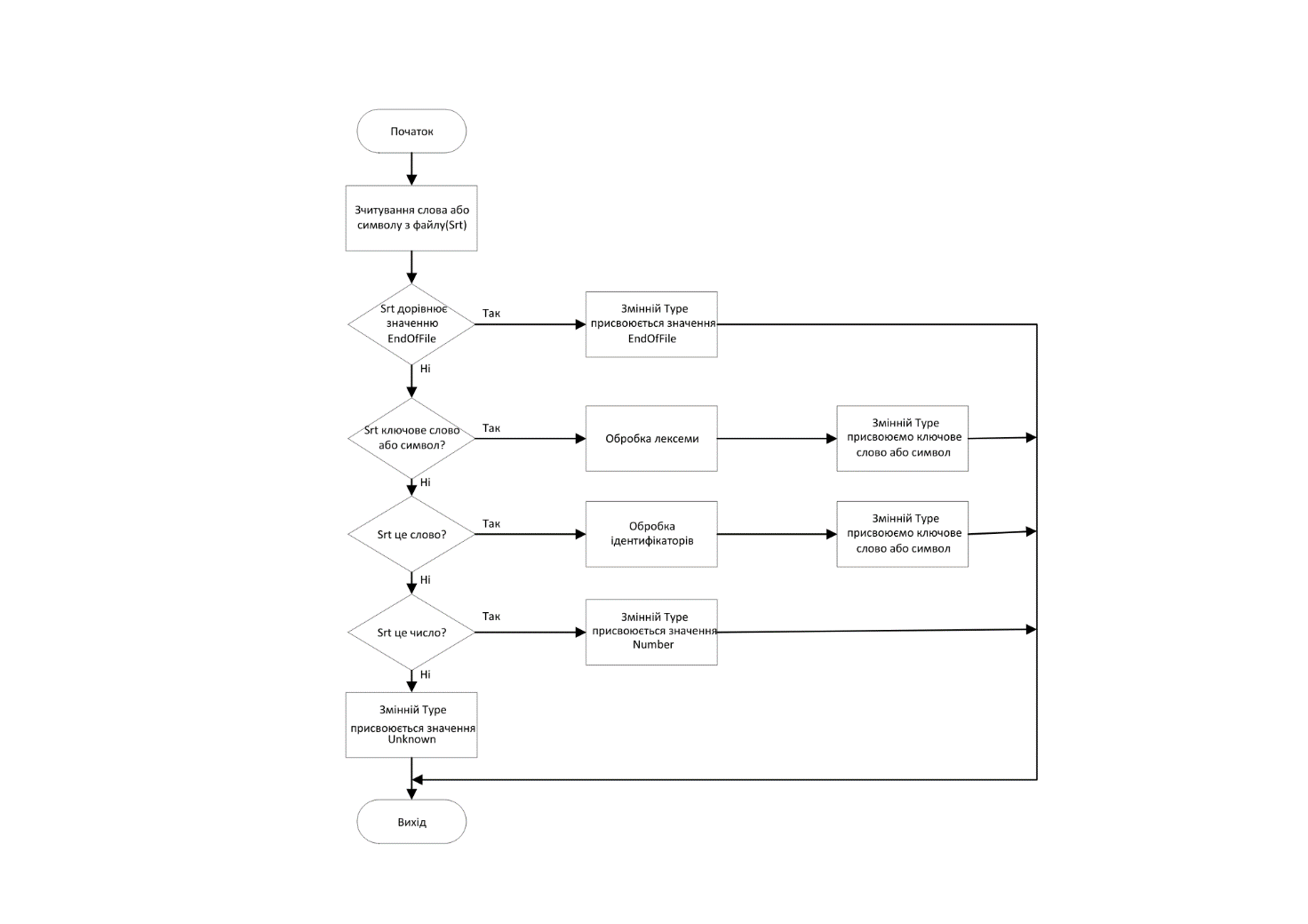
У стані **Separators** читаємо наступний символ і переходимо до виділення нової лексеми - до стану **Start**. Тобто пропускаємо усі пробіли, символи табуляції і переходу на новий рядок.

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 “GT”, “LT”. 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.*

* + 1. 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

EndOFile, // 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 fprintTokens(FILE\* F);

* 1. Розробка синтаксичного та семантичного аналізатора.

Синтаксичний аналіз - це процес, що визначає, чи належить деяка послідовність лексем граматиці мови програмування. В принципі, для будь-якої граматики можна побудувати синтаксичний аналізатор, але граматики, які використовуються на практиці, мають спеціальну форму. Наприклад, відомо, що для будь-якої контекстно-вільної граматики може бути побудований аналізатор, складність якого не перевищує O(n3) для вхідного рядка довжиною n, але в більшості випадків для заданої мови програмування ми можемо побудувати таку граматику, що дозволить сконструювати і більш швидкий аналізатор.

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

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

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

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.

* + 1. Розробка дерева граматичного розбору.

Схема дерева розбору виглядає наступним чином:

Program

├── "mainprogram"

├── {VariableDeclaration ";"}

│ ├── VariableDeclaration

│ │ ├── "INT\_4"

│ │ └── VariableList

│ │ ├── Identifier

│ │ │ ├── "\_"

│ │ │ └── Low {7}

│ │ │ ├── Low

│ │ │ │ └── "a" | "b" | ... | "z"

│ │ │ └── Digit

│ │ │ └── "0" | "1" | ... | "9"

│ │ └── {"," Identifier}  
├── "STARTBLOCK"

├── {Statement}

│ ├── Statement

│ │ ├── InputStatement

│ │ │ ├── "READ"

│ │ │ └── Identifier

│ │ ├── OutputStatement

│ │ │ ├── "WRITE"

│ │ │ └── ArithmeticExpression

│ │ │ ├── LowPriorityExpression

│ │ │ │ ├── MiddlePriorityExpression

│ │ │ │ │ ├── Identifier

│ │ │ │ │ ├── Number

│ │ │ │ │ │ ├── ["sub"]

│ │ │ │ │ │ └── Digit {7}

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

│ │ │ │ └── {MiddlePriorityOperator MiddlePriorityExpression}

│ │ │ └── {LowPriorityOperator LowPriorityExpression}

│ │ ├── AssignStatement

│ │ │ ├── Identifier

│ │ │ └── "<==" ArithmeticExpression

│ │ ├── IfElseStatement

│ │ │ ├── "IF"

│ │ │ ├── "(" LogicalExpression ")"

│ │ │ │ ├── AndExpression

│ │ │ │ │ ├── Comparison

│ │ │ │ │ │ ├── ComparisonExpression

│ │ │ │ │ │ │ ├── ArithmeticExpression

│ │ │ │ │ │ │ ├── ComparisonOperator

│ │ │ │ │ │ │ └── ArithmeticExpression

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

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

│ │ │ │ └── {OrOperator AndExpression}

│ │ │ ├── Statement

│ │ │ └── ["ELSE" Statement]

│ │ ├── GotoStatement

│ │ │ ├── "GOTO"

│ │ │ └── Identifier

│ │ ├── LabelPoint

│ │ │ ├── Identifier

│ │ │ └── ":"

│ │ ├── ForToStatement

│ │ │ ├── "FOR"

│ │ │ ├── AssignStatement

│ │ │ ├── "TO" | "DOWNTO"

│ │ │ ├── ArithmeticExpression

│ │ │ ├── "DO"

│ │ │ └── Statement

│ │ ├── WhileStatement

│ │ │ ├── "WHILE"

│ │ │ ├── LogicalExpression

│ │ │ ├── {Statement}

│ │ │ └── "END" "WHILE"

│ │ ├── RepeatUntilStatement

│ │ │ ├── "REPEAT"

│ │ │ ├── {Statement}

│ │ │ └── "UNTIL" "(" LogicalExpression ")"

│ │ └── CompoundStatement

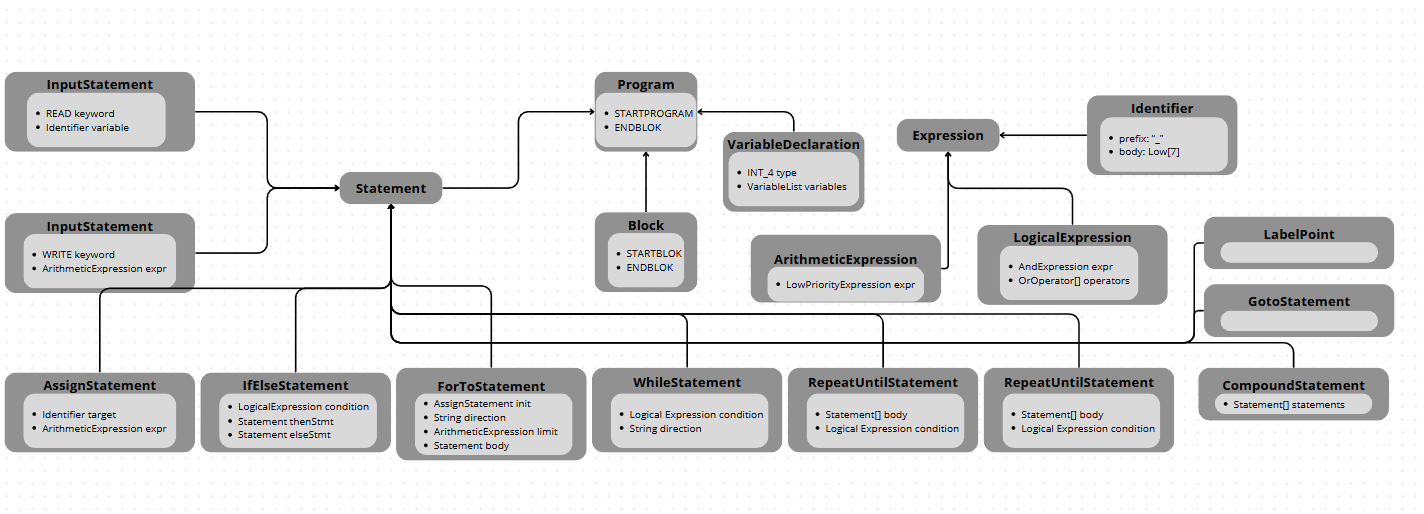
│ │ ├── "STARTBLOK"

│ │ ├── {Statement}

│ │ └── "ENDBLOK"

└── "ENDBLOK"

*Рис. 3.2. Дерево граматичного розбору.*



*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:

void program();

void programBody();

void variableDeclaration();

void variableList();

void statement();

void inputStatement();

void outputStatement();

void arithmeticExpression();

void lowPriorityExpression();

void middlePriorityExpression();

void assignStatement();

void ifStatement();

void logicalExpression();

void andExpression();

void comparison();

void comparisonExpression();

void gotoStatement();

void labelPoint();

void forStatement();

void whileStatement();

void statementInWhile();

void repeatStatement();

void compoundStatement();

Блок-схема алгоритму роботи синтаксичного аналізатора виглядатиме наступним чином:

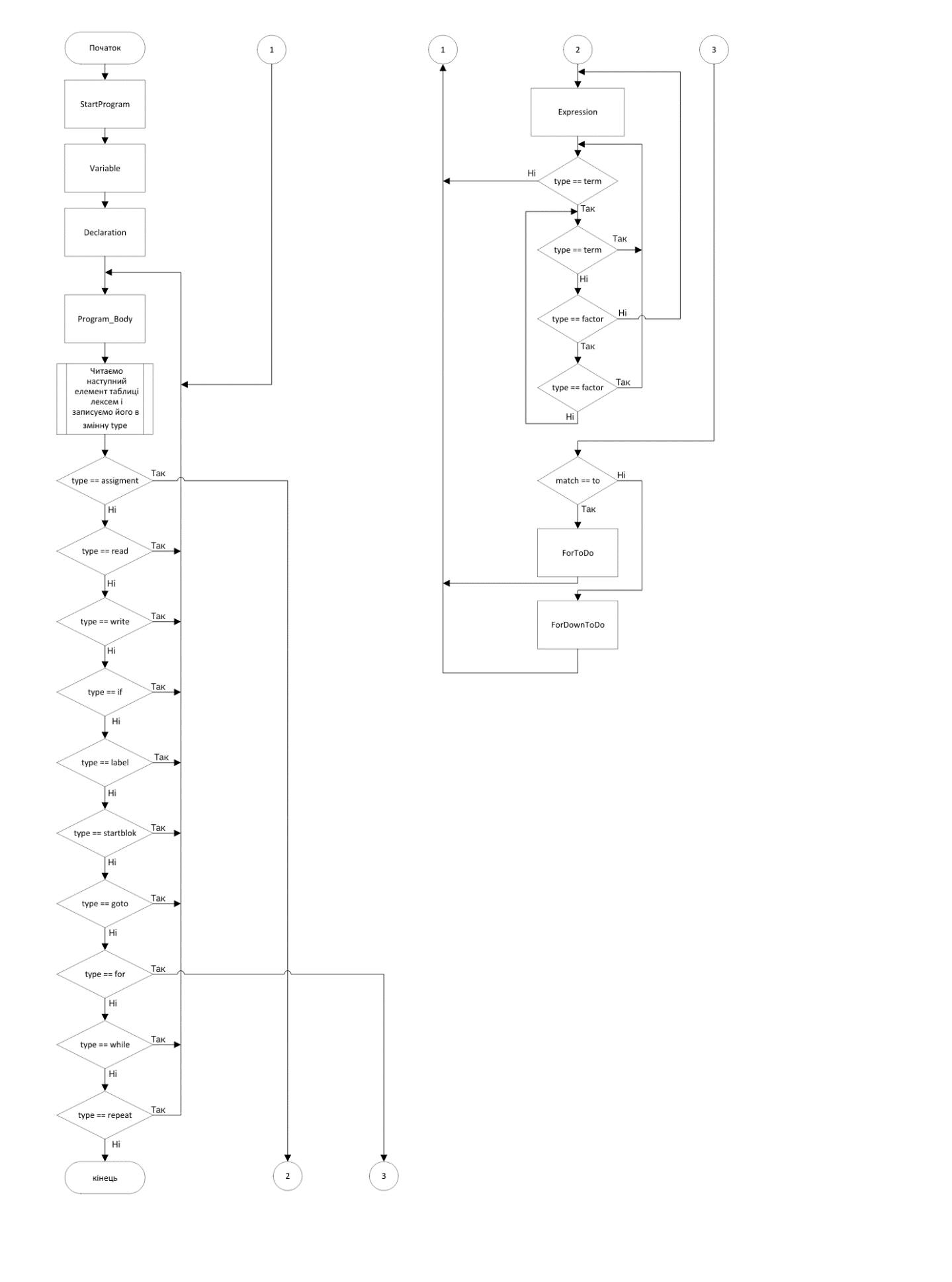


Рис. 3.3. Блок-сема алгоритму роботи синтаксичного аналізатора.

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

void match(enum TypeOfToken expectedType);

яка перевіряє, чи поточна лексема збігається з очікуваною.

Блок-схема алгоритму функції variable\_declaration(), яка перевіряє чи правильно описані змінні виглядає наступним чином:

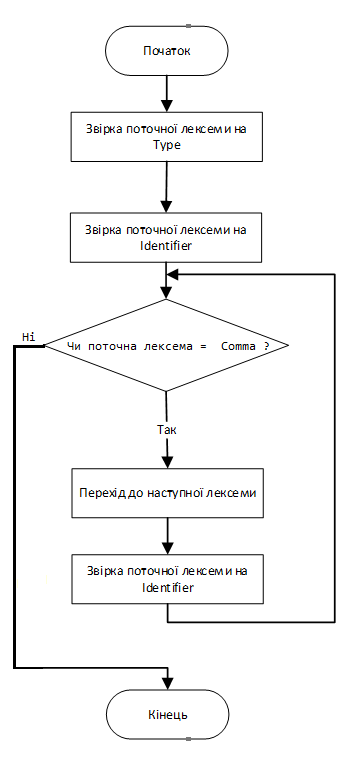


Рис. 3.4. Блок-сема алгоритму роботи функції variableDeclaration().

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.

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).

* + 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:

For

/ \

(to | downto) оператор

/ \

Оп. Прис. ариф. вир.

Оператор циклу while:

While

/ \

Умова statement

/ \

Statement оператор

/ \

Оператор оператор

Оператор циклу repeat:

Repeat

/ \

Statement умова

/ \

Оператор оператор

Оператор присвоєння:

<==

/ \

Id арифметичний вираз

Арифметичний вираз:

(ADD або SUB)

/ \

Id id

Доданок:

(MUL, DIV або MOD)  
/ \  
множник множник

Множник:

фактор  
/ \

id або number або (арифм. вираз) null

Складений оператор:

compount

/ \

statement null

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

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

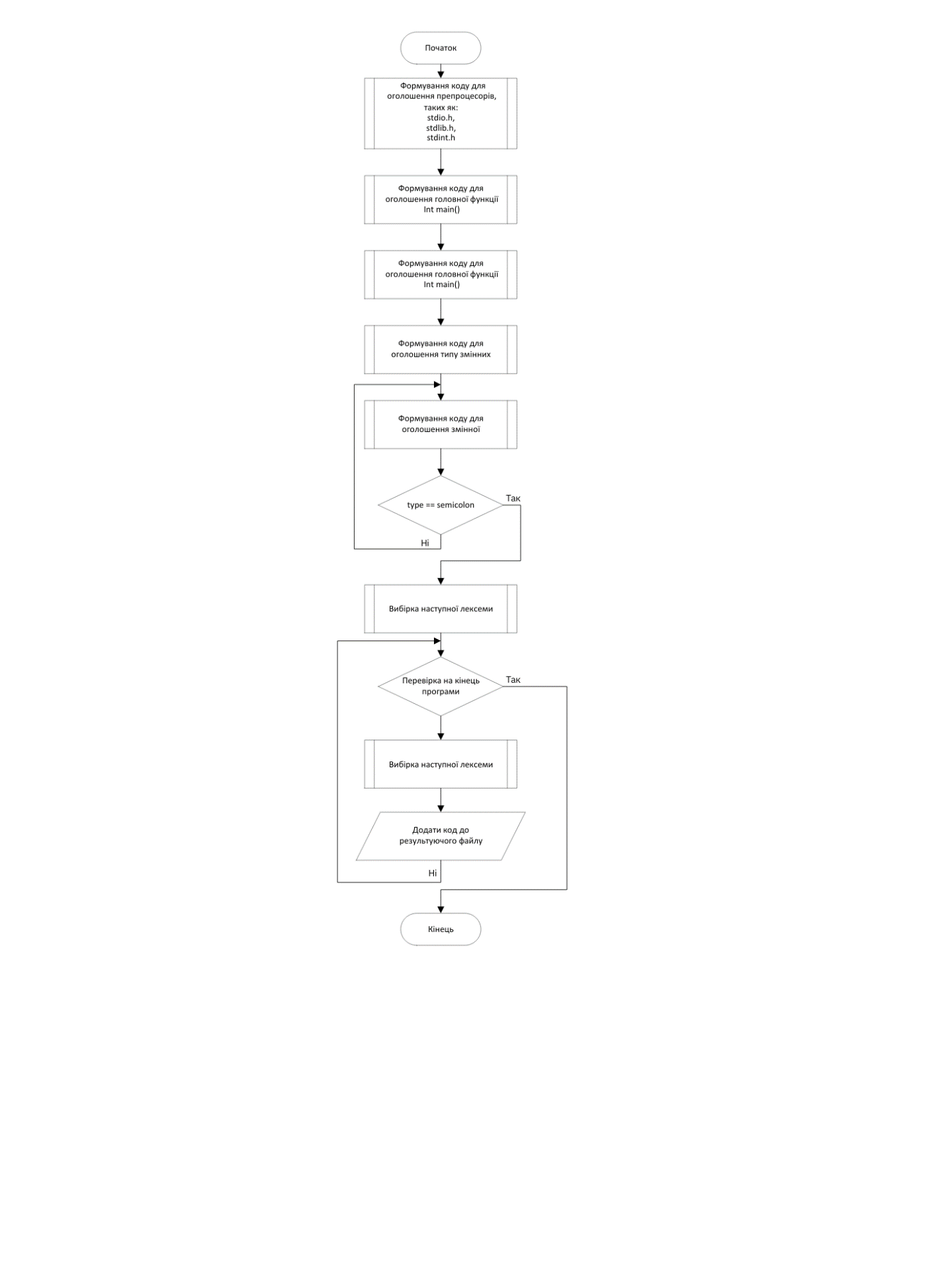


Рис. 3.6. Блок-сема алгоритму роботи генератора коду.

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

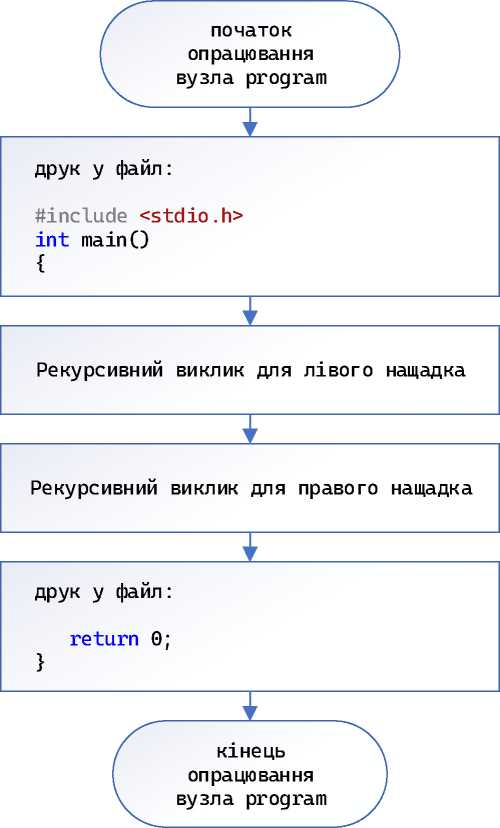


Рис. 3.7. Блок-сема алгоритму опрацювання вузла program.

* + 1. 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 С 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;

}

4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО

ТРАНСЛЯТОРА

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

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

* 1. Опис інтерфейсу та інструкції користувачу.

Розроблений транслятор має простий консольний інтерфейс.

При запуску програми необхідно ввести ім’я файлу з текстом програми на вхідній мові програмування:

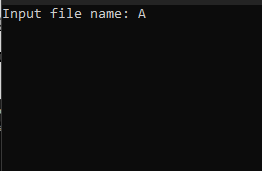


Рис. 4.1. Інтерфейс розробленого транслятора.

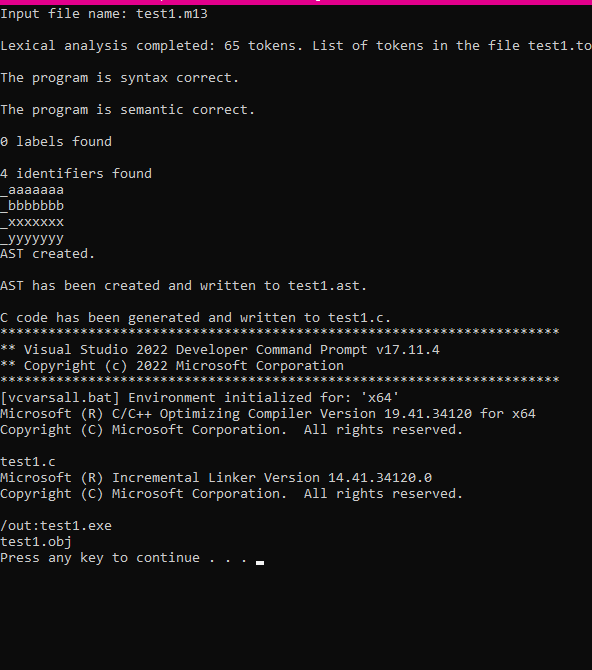
Після вводу імені програми отримуємо результати роботи транслятора:   


Рис. 4.2.Результати роботи розробленого транслятора.

* 1. Виявлення лексичних і синтаксичних помилок.

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

Наприклад, у програмі зробимо синтаксичну помилку - у 24 рядку неправильно вкажемо ідентифікатор:

STARTPROGRAM

VARIABLE INT\_4 \_aaaaaaa, \_bbbbbbb, \_xxxxxxx, \_yyyyyyy;

STARTBLOK

READ \_aaaaaaa

READ \_bbbbbbb

WRITE \_aaaaaaa ADD \_bbbbbbb

WRITE \_aaaaaaa SUB \_bbbbbbb

WRITE \_aaaaaaa MUL \_bbbbbbb

WRITE \_aaaaaaa DIV \_bbbbbbb

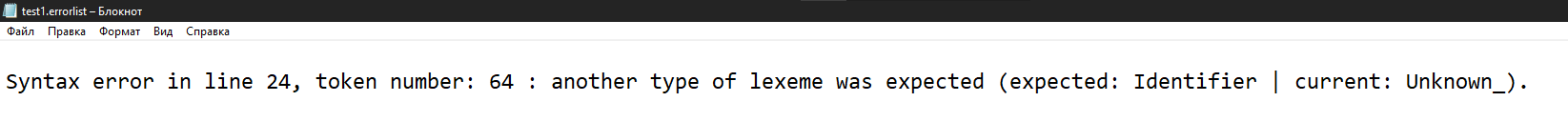
WRITE \_aaaaaaa MOD \_bbbbbbb

\_xxxxxxx <== (\_aaaaaaa SUB \_bbbbbbb) MUL 10 ADD (\_aaaaaaa ADD \_bbbbbbb) DIV 10

\_yyyyyyy <== \_xxxxxxx ADD \_xxxxxxx MOD 10

WRITE \_xxxxxxx

WRITE \_yyyyyyA

ENDBLOK

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

Зробимо семантичну помилку - не оголосимо змінну “\_aaaaaaa”:

STARTPROGRAM

VARIABLE INT\_4 \_aaaaaab, \_bbbbbbb, \_xxxxxxx, \_yyyyyyy;

STARTBLOK

READ \_aaaaaaa

READ \_bbbbbbb

WRITE \_aaaaaaa ADD \_bbbbbbb

WRITE \_aaaaaaa SUB \_bbbbbbb

WRITE \_aaaaaaa MUL \_bbbbbbb

WRITE \_aaaaaaa DIV \_bbbbbbb

WRITE \_aaaaaaa MOD \_bbbbbbb

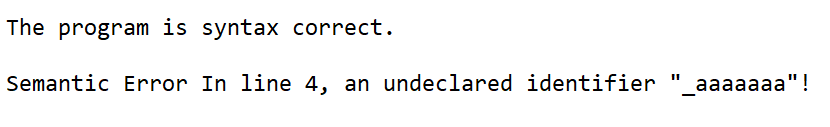
\_xxxxxxx <== (\_aaaaaaa SUB \_bbbbbbb) MUL 10 ADD (\_aaaaaaa ADD \_bbbbbbb) DIV 10

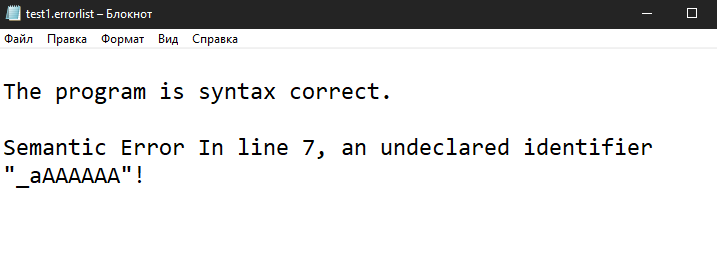
\_yyyyyyy <== \_xxxxxxx ADD \_xxxxxxx MOD 10

WRITE \_xxxxxxx

WRITE \_yyyyyyA

ENDBLOK





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

* 1. Перевірка роботи транслятора за допомогою тестових задач.

Тестова програма «*Лінійний алгоритм*»

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

2. Вивести на екран:

А + В (результат операції додавання);

А - В (результат операції віднімання);

А \* В (результат операції множення);

А / В (результат операції ділення);

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

3. Обрахувати значення виразів

Х = (А - В) \* 10 + (А + В) / 10

Y = Х + Х % 10

1. Вивести значення Х і У на екран.

Напишемо програму на вхідній мові програмування:

STARTPROGRAM

VARIABLE INT\_4 \_aaaaaaa, \_bbbbbbb, \_xxxxxxx, \_yyyyyyy;

STARTBLOK

READ \_aaaaaaa

READ \_bbbbbbb

WRITE \_aaaaaaa ADD \_bbbbbbb

WRITE \_aaaaaaa SUB \_bbbbbbb

WRITE \_aaaaaaa MUL \_bbbbbbb

WRITE \_aaaaaaa DIV \_bbbbbbb

WRITE \_aaaaaaa MOD \_bbbbbbb

\_xxxxxxx <== (\_aaaaaaa SUB \_bbbbbbb) MUL 10 ADD (\_aaaaaaa ADD \_bbbbbbb) DIV 10

\_yyyyyyy <== \_xxxxxxx ADD \_xxxxxxx MOD 10

WRITE \_xxxxxxx

WRITE \_yyyyyyy

ENDBLOK  
В результаті отримаємо файл з програмою на мові програмування С:

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_yyyyyyy;

int \_xxxxxxx;

int \_bbbbbbb;

int \_aaaaaaa;

printf("Enter \_aaaaaaa: ");

scanf("%d", &\_aaaaaaa);

printf("Enter \_bbbbbbb: ");

scanf("%d", &\_bbbbbbb);

printf("%d\n", (\_aaaaaaa + \_bbbbbbb));

printf("%d\n", (\_aaaaaaa - \_bbbbbbb));

printf("%d\n", (\_aaaaaaa \* \_bbbbbbb));

printf("%d\n", (\_aaaaaaa / \_bbbbbbb));

printf("%d\n", (\_aaaaaaa % \_bbbbbbb));

\_xxxxxxx = (((\_aaaaaaa - \_bbbbbbb) \* 10) + ((\_aaaaaaa + \_bbbbbbb) / 10));

\_yyyyyyy = (\_xxxxxxx + (\_xxxxxxx % 10));

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

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

   system("pause");

    return 0;

}

Також на виході маємо виконуваний файл test1.exe, запустивши який маємо:

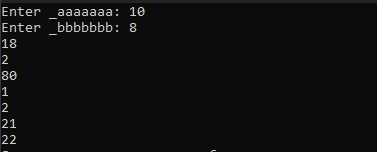


Рис. 4.5. Результати виконання тестової задачі 1.

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

1. Ввести три числа А, В, С (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання). Використання вкладеного умовного оператора:
2. Знайти найбільше з них і вивести його на екран. Використання простого умовного оператора:
3. Вивести на екран число 1, якщо усі числа однакові (логічний вираз в умовному операторі має виглядати так: «(А=В) і (А=С) і (В=С)»), інакше вивести 0.
4. Вивести на екран число -1, якщо хоча б одне з чисел від’ємне (логічний вираз в умовному операторі має виглядати так: «(А<0) або (В<0) або (С<0)»), інакше вивести 0.
5. Вивести на екран число 10, якщо число А більше за суму чисел В і С (логічний вираз в умовному операторі має виглядати так: «!(А<(В+С))»), інакше вивести 0

Напишемо програму на вхідній мові програмування:

STARTPROGRAM

VARIABLE INT\_4 \_aaaaaaa, \_bbbbbbb, \_ccccccc;

STARTBLOK

#\* Ввести три числа А, В, С \*#

READ \_aaaaaaa

READ \_bbbbbbb

READ \_ccccccc

#\* Використання вкладеного умовного оператора: \*#

#\* 2. Знайти найбільше з них і вивести його на екран \*#

IF ( \_aaaaaaa LT \_bbbbbbb)

IF( \_aaaaaaa LT \_ccccccc)

WRITE \_aaaaaaa;

ELSE

WRITE \_ccccccc;

;

ELSE

IF (\_bbbbbbb LT \_ccccccc)

WRITE \_bbbbbbb;

ELSE

WRITE \_ccccccc;

;

#\* Використання простого умовного оператора: \*#

#\* 3. Вивести на екран число 1, якщо усі числа однакові (логічний вираз в умовному операторі \*#

#\* має виглядати так: «(А=В) і (А=С) і (В=С)»), інакше вивести 0. \*#

;

IF ((\_aaaaaaa EQ \_bbbbbbb) & (\_aaaaaaa EQ \_ccccccc) & (\_aaaaaaa EQ \_ccccccc))

WRITE 1;

ELSE

WRITE 0;

#\* 4. Вивести на екран число -1, якщо хоча б одне з чисел від’ємне (логічний вираз в умовному \*#

#\* операторі має виглядати так: «(А<<0) або (В<<0) або (С<<0)»), інакше вивести 0. \*#

IF ((\_aaaaaaa GT 0) | (\_bbbbbbb GT 0) | (\_ccccccc GT 0))

WRITE -1;

ELSE

WRITE 0;

#\* 5. Вивести на екран число 10, якщо число А більше за суму чисел В і С (логічний вираз в \*#

#\* умовному операторі має виглядати так: «!(А<<(В+С))»), інакше вивести 0 \*#

IF (! (\_aaaaaaa GT (\_bbbbbbb ADD \_ccccccc)))

WRITE 10;

ELSE

WRITE 0;

ENDBLOK

В результаті отримаємо файл з програмою на мові програмування С:

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_ccccccc;

int \_bbbbbbb;

int \_aaaaaaa;

printf("Enter \_aaaaaaa: ");

scanf("%d", &\_aaaaaaa);

printf("Enter \_bbbbbbb: ");

scanf("%d", &\_bbbbbbb);

printf("Enter \_ccccccc: ");

scanf("%d", &\_ccccccc);

if ((\_aaaaaaa < \_bbbbbbb)) {

if ((\_aaaaaaa < \_ccccccc)) {

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

}

else {

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

}

}

else {

if ((\_bbbbbbb < \_ccccccc)) {

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

}

else {

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

}

}

if (((\_aaaaaaa == \_bbbbbbb) && ((\_aaaaaaa == \_ccccccc) && (\_aaaaaaa == \_ccccccc)))) {

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

}

else {

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

}

if (((\_aaaaaaa > 0) || ((\_bbbbbbb > 0) || (\_ccccccc > 0)))) {

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

}

else {

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

}

if (!((\_aaaaaaa > (\_bbbbbbb + \_ccccccc)))) {

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

}

else {

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

}

   system("pause");

    return 0;

}

Також на виході маємо виконуваний файл test2.exe, запустивши який маємо:

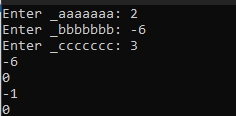


Рис. 4.6. Результати виконання тестової задачі 2.

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

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

Використання простого оператора циклу:

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

Використання вкладеного оператора циклу:

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

Х = 0

Цикл від 1 до А з кроком 1

Цикл від 1 до В з кроком 1

Х = Х + 1

4. Вивести значення Х на екран.

Напишемо програму на вхідній мові програмування:

#\* Тестова програма «Циклічний алгоритм» \*#

STARTPROGRAM

VARIABLE INT\_4 \_aaaaaaa, \_bbbbbbb, \_xxxxxxx, \_iiiiiii, \_jjjjjjj;

STARTBLOK

#\* 1. Ввести два числа А і В, причому А<В \*#

READ \_aaaaaaa

READ \_bbbbbbb

#\* Використання простого оператора циклу: \*#

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

FOR \_xxxxxxx <== \_aaaaaaa TO \_bbbbbbb DO

WRITE \_xxxxxxx MUL \_xxxxxxx

;

#\* Використання вкладеного оператора циклу: \*#

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

\_xxxxxxx <== 0

FOR \_iiiiiii <== 1 TO \_aaaaaaa DO

FOR \_jjjjjjj <== 1 TO \_bbbbbbb DO

\_xxxxxxx <== \_xxxxxxx ADD 1

;

;

#\* 4. Вивести значення Х на екран \*#

WRITE \_xxxxxxx

ENDBLOK

В результаті отримаємо файл з програмою на мові програмування С:

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_jjjjjjj;

int \_iiiiiii;

int \_xxxxxxx;

int \_bbbbbbb;

int \_aaaaaaa;

printf("Enter \_aaaaaaa: ");

scanf("%d", &\_aaaaaaa);

printf("Enter \_bbbbbbb: ");

scanf("%d", &\_bbbbbbb);

for(

\_xxxxxxx = \_aaaaaaa;

\_xxxxxxx <= \_bbbbbbb;

++\_xxxxxxx

) {

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

}\_xxxxxxx = 0;

for(

\_iiiiiii = 1;

\_iiiiiii <= \_aaaaaaa;

++\_iiiiiii

) {

for(

\_jjjjjjj = 1;

\_jjjjjjj <= \_bbbbbbb;

++\_jjjjjjj

) {

\_xxxxxxx = (\_xxxxxxx + 1);

}

}printf("%d\n", \_xxxxxxx);

   system("pause");

    return 0;

}

Також на виході маємо виконуваний файл test3.exe, запустивши який маємо:

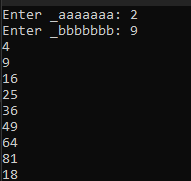


Рис. 4.7. Результати виконання тестової задачі 3.

**ВИСНОВКИ**

У межах курсового проєкту було розроблено транслятор вхідної мови програмування, який виконує такі завдання:

**1. Лексичний аналіз**

* Текст програми розбивається на лексеми з подальшим формуванням таблиці, що містить тип, значення та номер рядка кожної лексеми.
* Лексичний аналізатор працює за принципом скінченного автомату, розпізнаючи ключові слова, ідентифікатори, константи, оператори та розділювачі.

**2. Синтаксичний і семантичний аналіз**

* **Синтаксичний аналізатор** перевіряє відповідність структури програми заданій граматиці, будує дерево розбору, а також формує таблиці ідентифікаторів і типів.
* **Семантичний аналізатор** аналізує логічну коректність програми: перевіряє відповідність типів даних, області видимості змінних і коректність викликів функцій.

**3. Генерація коду**

* На основі абстрактного дерева генератор коду створює вихідний текст на мові C, здійснюючи обхід дерева та генеруючи відповідний код для кожного вузла.

**4. Тестування**

* Було проведено тестування на різноманітних програмах (лінійні алгоритми, конструкції з розгалуженнями та циклами).
* Виявлені лексичні, синтаксичні й семантичні помилки успішно усунуті.
* Транслятор генерує коректний код на основі введених програм.

**Переваги проєкту**

* Повна реалізація всіх основних етапів трансляції.
* Модульна структура, що спрощує розширення та подальші вдосконалення.
* Тестування підтвердило стабільність і надійність роботи.

**Недоліки проєкту**

* Обмежена підтримка лише базових конструкцій мови.
* Відсутність графічного інтерфейсу, що впливає на зручність використання.

**Висновок**

Розроблений транслятор демонструє базову функціональність і може слугувати основою для подальшого розвитку та вдосконалення.

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 - «Комп’ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. - Київ: КПІ ім. Ігоря Сікорського, 2021. - 108 с.
2. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. - Харків: НТУ «ХПІ», 2021. - 133 с.
3. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії формальних мов: Навчальний посібник у двох частинах. - Чернівці: ЧНУ, 2008. - 84 c.
4. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. - Чернівці: ЧНУ, 2008. - 84 c.
5. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. - 1038 c.
6. Системне програмування (курсовий проект) [Електронний ресурс] - Режим доступу до ресурсу: <https://vns.lpnu.ua/course/view.php?id=11685>.
7. MIT OpenCourseWare. Computer Language Engineering [Електронний ресурс] - Режим доступу до ресурсу: [https://ocw.mit.edu/courses/6-035-computer- language-engineering-spring-2010](https://ocw.mit.edu/courses/6-035-computer-language-engineering-spring-2010).

ДОДАТКИ

А. Таблиці лексем для тестових прикладів

Тестова програма «Лінійний алгоритм»

---------------------------------------------------------------------------

| TOKEN TABLE |

---------------------------------------------------------------------------

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

---------------------------------------------------------------------------

| 1 | STARTPROGRAM | 0 | 0 | StartProgram |

---------------------------------------------------------------------------

| 2 | VARIABLE | 0 | 2 | Variable |

---------------------------------------------------------------------------

| 2 | INT\_4 | 0 | 3 | Type |

---------------------------------------------------------------------------

| 2 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 2 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 2 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 2 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 2 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 2 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 2 | \_yyyyyyy | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 2 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 3 | STARTBLOK | 0 | 1 | StartBlock |

---------------------------------------------------------------------------

| 4 | READ | 0 | 5 | Input |

---------------------------------------------------------------------------

| 4 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 5 | READ | 0 | 5 | Input |

---------------------------------------------------------------------------

| 5 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 7 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 7 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 7 | ADD | 0 | 24 | Add |

---------------------------------------------------------------------------

| 7 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 8 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 8 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 8 | SUB | 0 | 25 | Sub |

---------------------------------------------------------------------------

| 8 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 9 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 9 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 9 | MUL | 0 | 26 | Mul |

---------------------------------------------------------------------------

| 9 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 10 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 10 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 10 | DIV | 0 | 27 | Div |

---------------------------------------------------------------------------

| 10 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 11 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 11 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 11 | MOD | 0 | 28 | Mod |

---------------------------------------------------------------------------

| 11 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | <== | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 13 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 13 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | SUB | 0 | 25 | Sub |

---------------------------------------------------------------------------

| 13 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 13 | MUL | 0 | 26 | Mul |

---------------------------------------------------------------------------

| 13 | 10 | 10 | 21 | Number |

---------------------------------------------------------------------------

| 13 | ADD | 0 | 24 | Add |

---------------------------------------------------------------------------

| 13 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 13 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | ADD | 0 | 24 | Add |

---------------------------------------------------------------------------

| 13 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 13 | DIV | 0 | 27 | Div |

---------------------------------------------------------------------------

| 13 | 10 | 10 | 21 | Number |

---------------------------------------------------------------------------

| 14 | \_yyyyyyy | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | <== | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 14 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | ADD | 0 | 24 | Add |

---------------------------------------------------------------------------

| 14 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | MOD | 0 | 28 | Mod |

---------------------------------------------------------------------------

| 14 | 10 | 10 | 21 | Number |

---------------------------------------------------------------------------

| 16 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 16 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 17 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 17 | \_yyyyyyy | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | ENDBLOK | 0 | 4 | EndBlock |

---------------------------------------------------------------------------

Тестова програма «Алгоритм з розгалуженням»

---------------------------------------------------------------------------

| TOKEN TABLE |

---------------------------------------------------------------------------

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

---------------------------------------------------------------------------

| 1 | STARTPROGRAM | 0 | 0 | StartProgram |

---------------------------------------------------------------------------

| 2 | VARIABLE | 0 | 2 | Variable |

---------------------------------------------------------------------------

| 2 | INT\_4 | 0 | 3 | Type |

---------------------------------------------------------------------------

| 2 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 2 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 2 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 2 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 2 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 2 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 3 | STARTBLOK | 0 | 1 | StartBlock |

---------------------------------------------------------------------------

| 6 | READ | 0 | 5 | Input |

---------------------------------------------------------------------------

| 6 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 7 | READ | 0 | 5 | Input |

---------------------------------------------------------------------------

| 7 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 8 | READ | 0 | 5 | Input |

---------------------------------------------------------------------------

| 8 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 12 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 12 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 12 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 12 | LT | 0 | 32 | Less |

---------------------------------------------------------------------------

| 12 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 12 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 13 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 13 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 13 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | LT | 0 | 32 | Less |

---------------------------------------------------------------------------

| 13 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 14 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 14 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 15 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 16 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 16 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 16 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 17 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 18 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 19 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 19 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 19 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | LT | 0 | 32 | Less |

---------------------------------------------------------------------------

| 19 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 20 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 20 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 20 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 21 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 22 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 22 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 22 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 23 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 28 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 29 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 29 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 29 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 29 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 29 | EQ | 0 | 29 | Equality |

---------------------------------------------------------------------------

| 29 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 29 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 29 | & | 0 | 34 | And |

---------------------------------------------------------------------------

| 29 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 29 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 29 | EQ | 0 | 29 | Equality |

---------------------------------------------------------------------------

| 29 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 29 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 29 | & | 0 | 34 | And |

---------------------------------------------------------------------------

| 29 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 29 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 29 | EQ | 0 | 29 | Equality |

---------------------------------------------------------------------------

| 29 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 29 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 29 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 30 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 30 | 1 | 1 | 21 | Number |

---------------------------------------------------------------------------

| 30 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 31 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 32 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 32 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 32 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 36 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 36 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 36 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 36 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 36 | GT | 0 | 31 | Greate |

---------------------------------------------------------------------------

| 36 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 36 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 36 | | | 0 | 35 | Or |

---------------------------------------------------------------------------

| 36 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 36 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 36 | GT | 0 | 31 | Greate |

---------------------------------------------------------------------------

| 36 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 36 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 36 | | | 0 | 35 | Or |

---------------------------------------------------------------------------

| 36 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 36 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 36 | GT | 0 | 31 | Greate |

---------------------------------------------------------------------------

| 36 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 36 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 36 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 37 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 37 | -1 | -1 | 21 | Number |

---------------------------------------------------------------------------

| 37 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 38 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 39 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 39 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 39 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 43 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 43 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 43 | ! | 0 | 33 | Not |

---------------------------------------------------------------------------

| 43 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 43 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 43 | GT | 0 | 31 | Greate |

---------------------------------------------------------------------------

| 43 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 43 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 43 | ADD | 0 | 24 | Add |

---------------------------------------------------------------------------

| 43 | \_ccccccc | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 43 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 43 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 43 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 44 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 44 | 10 | 10 | 21 | Number |

---------------------------------------------------------------------------

| 44 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 45 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 46 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 46 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 46 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 49 | ENDBLOK | 0 | 4 | EndBlock |

Тестова програма «*Циклічний алгоритм*»

---------------------------------------------------------------------------

| TOKEN TABLE |

---------------------------------------------------------------------------

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

---------------------------------------------------------------------------

| 2 | STARTPROGRAM | 0 | 0 | StartProgram |

---------------------------------------------------------------------------

| 3 | VARIABLE | 0 | 2 | Variable |

---------------------------------------------------------------------------

| 3 | INT\_4 | 0 | 3 | Type |

---------------------------------------------------------------------------

| 3 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 3 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 3 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 3 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 3 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 3 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 3 | \_iiiiiii | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 3 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 3 | \_jjjjjjj | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 3 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 4 | STARTBLOK | 0 | 1 | StartBlock |

---------------------------------------------------------------------------

| 8 | READ | 0 | 5 | Input |

---------------------------------------------------------------------------

| 8 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 9 | READ | 0 | 5 | Input |

---------------------------------------------------------------------------

| 9 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | FOR | 0 | 10 | For |

---------------------------------------------------------------------------

| 14 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | <== | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 14 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | TO | 0 | 11 | To |

---------------------------------------------------------------------------

| 14 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | DO | 0 | 13 | Do |

---------------------------------------------------------------------------

| 15 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 15 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 15 | MUL | 0 | 26 | Mul |

---------------------------------------------------------------------------

| 15 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 16 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 21 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 21 | <== | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 21 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 22 | FOR | 0 | 10 | For |

---------------------------------------------------------------------------

| 22 | \_iiiiiii | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 22 | <== | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 22 | 1 | 1 | 21 | Number |

---------------------------------------------------------------------------

| 22 | TO | 0 | 11 | To |

---------------------------------------------------------------------------

| 22 | \_aaaaaaa | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 22 | DO | 0 | 13 | Do |

---------------------------------------------------------------------------

| 23 | FOR | 0 | 10 | For |

---------------------------------------------------------------------------

| 23 | \_jjjjjjj | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 23 | <== | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 23 | 1 | 1 | 21 | Number |

---------------------------------------------------------------------------

| 23 | TO | 0 | 11 | To |

---------------------------------------------------------------------------

| 23 | \_bbbbbbb | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 23 | DO | 0 | 13 | Do |

---------------------------------------------------------------------------

| 24 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 24 | <== | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 24 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 24 | ADD | 0 | 24 | Add |

---------------------------------------------------------------------------

| 24 | 1 | 1 | 21 | Number |

---------------------------------------------------------------------------

| 25 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 26 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 29 | WRITE | 0 | 6 | Output |

---------------------------------------------------------------------------

| 29 | \_xxxxxxx | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 31 | ENDBLOK | 0 | 4 | EndBlock |

---------------------------------------------------------------------------

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

Тестова програма «Лінійний алгоритм»

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_yyyyyyy;

int \_xxxxxxx;

int \_bbbbbbb;

int \_aaaaaaa;

printf("Enter \_aaaaaaa: ");

scanf("%d", &\_aaaaaaa);

printf("Enter \_bbbbbbb: ");

scanf("%d", &\_bbbbbbb);

printf("%d\n", (\_aaaaaaa + \_bbbbbbb));

printf("%d\n", (\_aaaaaaa - \_bbbbbbb));

printf("%d\n", (\_aaaaaaa \* \_bbbbbbb));

printf("%d\n", (\_aaaaaaa / \_bbbbbbb));

printf("%d\n", (\_aaaaaaa % \_bbbbbbb));

\_xxxxxxx = (((\_aaaaaaa - \_bbbbbbb) \* 10) + ((\_aaaaaaa + \_bbbbbbb) / 10));

\_yyyyyyy = (\_xxxxxxx + (\_xxxxxxx % 10));

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

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

   system("pause");

    return 0;

}

Тестова програма «Алгоритм з розгалуженням»

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_ccccccc;

int \_bbbbbbb;

int \_aaaaaaa;

printf("Enter \_aaaaaaa: ");

scanf("%d", &\_aaaaaaa);

printf("Enter \_bbbbbbb: ");

scanf("%d", &\_bbbbbbb);

printf("Enter \_ccccccc: ");

scanf("%d", &\_ccccccc);

if ((\_aaaaaaa < \_bbbbbbb)) {

if ((\_aaaaaaa < \_ccccccc)) {

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

}

else {

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

}

}

else {

if ((\_bbbbbbb < \_ccccccc)) {

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

}

else {

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

}

}

if (((\_aaaaaaa == \_bbbbbbb) && ((\_aaaaaaa == \_ccccccc) && (\_aaaaaaa == \_ccccccc)))) {

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

}

else {

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

}

if (((\_aaaaaaa > 0) || ((\_bbbbbbb > 0) || (\_ccccccc > 0)))) {

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

}

else {

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

}

if (!((\_aaaaaaa > (\_bbbbbbb + \_ccccccc)))) {

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

}

else {

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

}

   system("pause");

    return 0;

}

Тестова програма «*Циклічний алгоритм*»

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_jjjjjjj;

int \_iiiiiii;

int \_xxxxxxx;

int \_bbbbbbb;

int \_aaaaaaa;

printf("Enter \_aaaaaaa: ");

scanf("%d", &\_aaaaaaa);

printf("Enter \_bbbbbbb: ");

scanf("%d", &\_bbbbbbb);

for(

\_xxxxxxx = \_aaaaaaa;

\_xxxxxxx <= \_bbbbbbb;

++\_xxxxxxx

) {

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

}\_xxxxxxx = 0;

for(

\_iiiiiii = 1;

\_iiiiiii <= \_aaaaaaa;

++\_iiiiiii

) {

for(

\_jjjjjjj = 1;

\_jjjjjjj <= \_bbbbbbb;

++\_jjjjjjj

) {

\_xxxxxxx = (\_xxxxxxx + 1);

}

}printf("%d\n", \_xxxxxxx);

   system("pause");

    return 0;

}

В. Абстрактне синтаксичне дерево для тестових прикладів

Тестова програма «Лінійний алгоритм»

|-- Program(0)

| |-- var(2)

| | |-- \_yyyyyyy(1)

| | |-- var(2)

| | | |-- \_xxxxxxx(1)

| | | |-- var(2)

| | | | |-- \_bbbbbbb(1)

| | | | |-- var(2)

| | | | | |-- \_aaaaaaa(1)

| |-- statement(3)

| | |-- statement(3)

| | | |-- statement(3)

| | | | |-- statement(3)

| | | | | |-- statement(3)

| | | | | | |-- statement(3)

| | | | | | | |-- statement(3)

| | | | | | | | |-- statement(3)

| | | | | | | | | |-- statement(3)

| | | | | | | | | | |-- statement(3)

| | | | | | | | | | | |-- input(4)

| | | | | | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | | | | |-- input(4)

| | | | | | | | | | | | |-- \_bbbbbbb(1)

| | | | | | | | | | |-- output(5)

| | | | | | | | | | | |-- Add(6)

| | | | | | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | | | | | |-- \_bbbbbbb(1)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- Sub(7)

| | | | | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | | | | |-- \_bbbbbbb(1)

| | | | | | | | |-- output(5)

| | | | | | | | | |-- Mul(8)

| | | | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | | | |-- \_bbbbbbb(1)

| | | | | | | |-- output(5)

| | | | | | | | |-- Div(9)

| | | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | | |-- \_bbbbbbb(1)

| | | | | | |-- output(5)

| | | | | | | |-- Mod(10)

| | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | |-- \_bbbbbbb(1)

| | | | | |-- assign(12)

| | | | | | |-- \_xxxxxxx(1)

| | | | | | |-- Add(6)

| | | | | | | |-- Mul(8)

| | | | | | | | |-- Sub(7)

| | | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | | |-- \_bbbbbbb(1)

| | | | | | | | |-- 10(11)

| | | | | | | |-- Div(9)

| | | | | | | | |-- Add(6)

| | | | | | | | | |-- \_ aaaaaaa(1)

| | | | | | | | | |-- \_bbbbbbb(1)

| | | | | | | | |-- 10(11)

| | | | |-- assign(12)

| | | | | |-- \_yyyyyyy(1)

| | | | | |-- Add(6)

| | | | | | |-- \_xxxxxxx(1)

| | | | | | |-- Mod(10)

| | | | | | | |-- \_xxxxxxx(1)

| | | | | | | |-- 10(11)

| | | |-- output(5)

| | | | |-- \_xxxxxxx(1)

| | |-- output(5)

| | | |-- \_yyyyyyy(1)

Тестова програма «Алгоритм з розгалуженням»

|-- Program(0)

| |-- var(2)

| | |-- \_ccccccc(1)

| | |-- var(2)

| | | |-- \_bbbbbbb(1)

| | | |-- var(2)

| | | | |-- \_aaaaaaa(1)

| |-- statement(3)

| | |-- statement(3)

| | | |-- statement(3)

| | | | |-- statement(3)

| | | | | |-- statement(3)

| | | | | | |-- statement(3)

| | | | | | | |-- input(4)

| | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | |-- input(4)

| | | | | | | | |-- \_bbbbbbb(1)

| | | | | | |-- input(4)

| | | | | | | |-- \_ccccccc(1)

| | | | | |-- if(13)

| | | | | | |-- Greate(20)

| | | | | | | |-- \_aaaaaaa(1)

| | | | | | | |-- \_bbbbbbb(1)

| | | | | | |-- else(14)

| | | | | | | |-- if(13)

| | | | | | | | |-- Greate(20)

| | | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | | |-- \_ccccccc(1)

| | | | | | | | |-- else(14)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- \_ccccccc(1)

| | | | | | | |-- if(13)

| | | | | | | | |-- Greate(20)

| | | | | | | | | |-- \_bbbbbbb(1)

| | | | | | | | | |-- \_ccccccc(1)

| | | | | | | | |-- else(14)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- \_bbbbbbb(1)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- \_ccccccc(1)

| | | | |-- if(13)

| | | | | |-- and(16)

| | | | | | |-- Equality(18)

| | | | | | | |-- \_aaaaaaa(1)

| | | | | | | |-- \_bbbbbbb(1)

| | | | | | |-- and(16)

| | | | | | | |-- Equality(18)

| | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | | |-- \_ccccccc(1)

| | | | | | | |-- Equality(18)

| | | | | | | | |-- \_bbbbbbb(1)

| | | | | | | | |-- \_ccccccc(1)

| | | | | |-- else(14)

| | | | | | |-- output(5)

| | | | | | | |-- 1(11)

| | | | | | |-- output(5)

| | | | | | | |-- 0(11)

| | | |-- if(13)

| | | | |-- or(15)

| | | | | |-- Less(21)

| | | | | | |-- \_aaaaaaa(1)

| | | | | | |-- 0(11)

| | | | | |-- or(15)

| | | | | | |-- Less(21)

| | | | | | | |-- \_bbbbbbb(1)

| | | | | | | |-- 0(11)

| | | | | | |-- Less(21)

| | | | | | | |-- \_ccccccc(1)

| | | | | | | |-- 0(11)

| | | | |-- else(14)

| | | | | |-- output(5)

| | | | | | |-- -1(11)

| | | | | |-- output(5)

| | | | | | |-- 0(11)

| | |-- if(13)

| | | |-- not(17)

| | | | |-- Less(21)

| | | | | |-- \_aaaaaaa(1)

| | | | | |-- Add(6)

| | | | | | |-- \_bbbbbbb(1)

| | | | | | |-- \_ccccccc(1)

| | | |-- else(14)

| | | | |-- output(5)

| | | | | |-- 10(11)

| | | | |-- output(5)

| | | | | |-- 0(11)

Тестова програма «*Циклічний алгоритм*»

|-- Program(0)

| |-- var(2)

| | |-- \_jjjjjjj(1)

| | |-- var(2)

| | | |-- \_iiiiiii(1)

| | | |-- var(2)

| | | | |-- \_xxxxxxx(1)

| | | | |-- var(2)

| | | | | |-- \_bbbbbbb(1)

| | | | | |-- var(2)

| | | | | | |-- \_aaaaaaa(1)

| |-- statement(3)

| | |-- statement(3)

| | | |-- statement(3)

| | | | |-- statement(3)

| | | | | |-- statement(3)

| | | | | | |-- input(4)

| | | | | | | |-- \_aaaaaaa(1)

| | | | | | |-- input(4)

| | | | | | | |-- \_bbbbbbb(1)

| | | | | |-- for(24)

| | | | | | |-- to(25)

| | | | | | | |-- assign(12)

| | | | | | | | |-- \_xxxxxxx(1)

| | | | | | | | |-- \_aaaaaaa(1)

| | | | | | | |-- \_bbbbbbb(1)

| | | | | | |-- output(5)

| | | | | | | |-- Mul(8)

| | | | | | | | |-- \_xxxxxxx(1)

| | | | | | | | |-- \_xxxxxxx(1)

| | | | |-- assign(12)

| | | | | |-- \_xxxxxxx(1)

| | | | | |-- 0(11)

| | | |-- for(24)

| | | | |-- to(25)

| | | | | |-- assign(12)

| | | | | | |-- \_iiiiiii(1)

| | | | | | |-- 1(11)

| | | | | |-- \_aaaaaaa(1)

| | | | |-- for(24)

| | | | | |-- to(25)

| | | | | | |-- assign(12)

| | | | | | | |-- \_jjjjjjj(1)

| | | | | | | |-- 1(11)

| | | | | | |-- \_bbbbbbb(1)

| | | | | |-- assign(12)

| | | | | | |-- \_xxxxxxx(1)

| | | | | | |-- Add(6)

| | | | | | | |-- \_xxxxxxx(1)

| | | | | | | |-- 1(11)

| | |-- output(5)

| | | |-- \_xxxxxxx(1)

Тестова програма «Алгоритм з розгалуженням»

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_ccccccc;

int \_bbbbbbb;

int \_aaaaaaa;

printf("Enter \_aaaaaaa: ");

scanf("%d", &\_aaaaaaa);

printf("Enter \_bbbbbbb: ");

scanf("%d", &\_bbbbbbb);

printf("Enter \_ccccccc: ");

scanf("%d", &\_ccccccc);

if ((\_aaaaaaa < \_bbbbbbb)) {

if ((\_aaaaaaa < \_ccccccc)) {

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

}

else {

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

}

}

else {

if ((\_bbbbbbb < \_ccccccc)) {

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

}

else {

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

}

}

if (((\_aaaaaaa == \_bbbbbbb) && ((\_aaaaaaa == \_ccccccc) && (\_aaaaaaa == \_ccccccc)))) {

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

}

else {

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

}

if (((\_aaaaaaa > 0) || ((\_bbbbbbb > 0) || (\_ccccccc > 0)))) {

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

}

else {

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

}

if (!((\_aaaaaaa > (\_bbbbbbb + \_ccccccc)))) {

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

}

else {

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

}

   system("pause");

    return 0;

}

Тестова програма «*Циклічний алгоритм*»

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \_jjjjjjj;

int \_iiiiiii;

int \_xxxxxxx;

int \_bbbbbbb;

int \_aaaaaaa;

printf("Enter \_aaaaaaa: ");

scanf("%d", &\_aaaaaaa);

printf("Enter \_bbbbbbb: ");

scanf("%d", &\_bbbbbbb);

for(

\_xxxxxxx = \_aaaaaaa;

\_xxxxxxx <= \_bbbbbbb;

++\_xxxxxxx

) {

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

}\_xxxxxxx = 0;

for(

\_iiiiiii = 1;

\_iiiiiii <= \_aaaaaaa;

++\_iiiiiii

) {

for(

\_jjjjjjj = 1;

\_jjjjjjj <= \_bbbbbbb;

++\_jjjjjjj

) {

\_xxxxxxx = (\_xxxxxxx + 1);

}

}printf("%d\n", \_xxxxxxx);

   system("pause");

    return 0;

}

Додаток Г. Лістинги коду

Main.cpp

#include <iostream>

#include <windows.h>

#include "LexicAnalyzer.hpp"

#include "Parser.hpp"

#include "Ast.hpp"

#include "Codegen.hpp"

#define LANGUAGE ".m13"

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

FILE\* errorFile;

int main(int argc, char\* argv[])

{

// виділення пам'яті під таблицю лесем

TokenTable = (struct Token\*)malloc(MAX\_TOKENS \* sizeof(struct Token));

// виділення пам'яті під таблицю ідентифікаторів

idTable = (struct id\*)malloc(MAX\_IDENTIFIER \* sizeof(struct id));

labelTable = (struct id\*)malloc(MAX\_IDENTIFIER \* sizeof(struct id));

char InputFile[32] = "";

FILE\* InFile, \*TokenFileP;

if (argc != 2)

{

printf("Input file name: ");

gets\_s(InputFile);

}

else

{

strcpy\_s(InputFile, argv[1]);

}

if (!strstr(InputFile, LANGUAGE)) {

free(TokenTable);

free(idTable);

free(labelTable);

\_fcloseall();

printf("Program file name should contain \".p17\"");

system("pause");

return 1;

}

if ((fopen\_s(&InFile, InputFile, "rt")) != 0)

{

printf("Error: Can not open file: %s\n", InputFile);

system("pause");

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], errorFileName[32];

strcpy\_s(TokenFile, NameFile);

strcat\_s(TokenFile, ".token");

strcpy\_s(errorFileName, NameFile);

strcat\_s(errorFileName, ".errorlist");

// лексичний аналіз

if ((fopen\_s(&TokenFileP, TokenFile, "wt")) != 0)

{

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

free(TokenTable);

free(idTable);

free(labelTable);

\_fcloseall();

system("pause");

return 1;

}

if ((fopen\_s(&errorFile, errorFileName, "wt")) != 0)

{

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

free(TokenTable);

free(idTable);

free(labelTable);

\_fcloseall();

system("pause");

return 1;

}

TokensNum = LexicAnalyzer::getTokens(InFile);

LexicAnalyzer::fprintTokens(TokenFileP);

fclose(InFile);

fclose(TokenFileP);

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

//PrintTokens(TokenTable, TokensNum);

// синтаксичний аналіз

Parser::Parser();

Parser::Semantic();

// створення абстрактного синтаксичного дерева

struct astNode\* ast = AST::astParser();

//printf("\nAbstract Syntax Tree:\n");

//PrintAST(ASTree, 0);

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");

free(TokenTable);

free(idTable);

free(labelTable);

AST::deleteNode(ast);

exit(1);

system("pause");

}

AST::fPrintAST(ASTFile, ast, 0);

printf("\nAST has been created and written to %s.\n", AST);

char OutputFile[32];

strcpy\_s(OutputFile, NameFile);

strcat\_s(OutputFile, ".c");

// Open output file

FILE\* outFile;

fopen\_s(&outFile, OutputFile, "w");

if (!outFile)

{

printf("Failed to open output file.\n");

free(TokenTable);

free(idTable);

free(labelTable);

AST::deleteNode(ast);

exit(1);

system("pause");

}

// Generate C code from AST

Codegen::codegen(outFile, ast);

// генерація вихідного С коду

printf("\nC code has been generated and written to %s.\n", OutputFile);

// Close the file

\_fcloseall();

free(TokenTable);

free(idTable);

free(labelTable);

char setVar[256] = "\"C:\\Program Files\\Microsoft Visual Studio\\2022\\Community\\VC\\Auxiliary\\Build\\vcvars64.bat\"";

char createExe[128];

sprintf\_s(createExe, "cl %s", OutputFile);

strcat\_s(setVar, " && ");

strcat\_s(setVar, createExe);

system(setVar);

//\*/

system("pause");

return 0;

}

**LexicAnalyzer.hpp**

#pragma once

#include "header.hpp"

#include <stdio.h>

namespace LexicAnalyzer {

unsigned int getTokens(FILE\* F);

void printTokens(void);

void fprintTokens(FILE\* F);

//const char\* lexemeTypeName(enum TypeOfToken type);

}

**LexicAnalyzer.cpp**

#include "LexicAnalyzer.hpp"

#include <stdbool.h>

#include <string.h>

#include <stdlib.h>

extern struct Token\* TokenTable; // Таблиця лексем

extern unsigned int TokensNum; // Кількість лексем

unsigned int LexicAnalyzer::getTokens(FILE\* F) {

enum States state = Start;

struct Token tempToken;

char ch, buf[16];

unsigned int tokenCount = 0;

int line = 1;

int tokenLength = 0;

ch = getc(F);

while (true) {

switch (state) {

case Start: {

if (ch == EOF) {

state = EndOFile;

}

else if (('0' <= ch && ch <= '9') || ch == '-' || ch == '.') {

state = Digit;

}

else if (ch == '#') {

state = SComment;

}

else if (('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z') || ch == '\_') {

state = Letter;

}

else if (ch == ' ' || ch == '\t' || ch == '\n') {

state = Separator;

}

else {

state = Another;

}

break;

}

case Digit: {

buf[0] = ch;

int j = 1;

ch = getc(F);

while (((ch <= '9' && ch >= '0') || ch == '.') && j < 10) {

buf[j++] = ch;

ch = getc(F);

}

buf[j] = '\0';

if (!strcmp(buf, "-")) {

strcpy\_s(tempToken.name, "-");

tempToken.type = Sub;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

if (!strcmp(buf, ".")) {

strcpy\_s(tempToken.name, ".");

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

short dotCounter = 0, currCharIndex = 0;

while (buf[currCharIndex] != '\0') {

if (buf[currCharIndex] == '.') {

++dotCounter;

}

++currCharIndex;

}

if (dotCounter > 1) {

strcpy\_s(tempToken.name, buf);

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

if (dotCounter == 1) {

strcpy\_s(tempToken.name, buf);

tempToken.type = Float;

tempToken.value = atof(buf);

tempToken.line = line;

state = Finish;

break;

}

if ((buf[0] == '0' && buf[1] != '\0') || (buf[0] == '-' && buf[1] == '0')) {

strcpy\_s(tempToken.name, buf);

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

strcpy\_s(tempToken.name, buf);

tempToken.type = Number;

tempToken.value = atoi(buf);

tempToken.line = line;

state = Finish;

break;

}

case Separator: {

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 = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

}

break;

}

case Comment: {

while (ch != '\n' && ch != EOF) {

ch = getc(F);

}

if (ch == EOF)

state = EndOFile;

else {

++line;

ch = getc(F);

state = Start;

}

break;

}

case Finish: {

if (tokenCount < MAX\_TOKENS) {

TokenTable[tokenCount++] = tempToken;

if (ch != EOF) {

state = Start;

}

else {

state = EndOFile;

}

}

else {

printf("\n\t\t\ttoo many tokens !!!\n");

return TokensNum = tokenCount - 1;

}

break;

}

case EndOFile: {

return TokensNum = tokenCount;

}

case Letter: {

buf[0] = ch;

int j = 1;

ch = getc(F);

while ((('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z') || ('0' <= ch && ch <= '9')) || ch == '\_' && j < 15) {

buf[j++] = ch;

ch = getc(F);

}

buf[j] = '\0';

enum TypeOfToken tempType = Unknown\_;

if (!strcmp(buf, "STARTPROGRAM")) {

tempType = StartProgram;

}

else if (!strcmp(buf, "STARTBLOK")) {

tempType = StartBlock;

}

else if (!strcmp(buf, "VARIABLE")) {

tempType = Variable;

}

else if (!strcmp(buf, "INT\_4")) {

tempType = Type;

}

else if (!strcmp(buf, "ENDBLOK")) {

tempType = EndBlock;

}

else if (!strcmp(buf, "READ")) {

tempType = Input;

}

else if (!strcmp(buf, "WRITE")) {

tempType = Output;

}

else if (!strcmp(buf, "IF")) {

tempType = If;

}

else if (!strcmp(buf, "ELSE")) {

tempType = Else;

}

else if (!strcmp(buf, "GOTO")) {

tempType = Goto;

}

else if (!strcmp(buf, "FOR")) {

tempType = For;

}

else if (!strcmp(buf, "TO")) {

tempType = To;

}

else if (!strcmp(buf, "DOWNTO")) {

tempType = Downto;

}

else if (!strcmp(buf, "DO")) {

tempType = Do;

}

else if (!strcmp(buf, "WHILE")) {

tempType = While;

}

else if (!strcmp(buf, "CONTINUE")) {

tempType = Continue;

}

else if (!strcmp(buf, "EXIT")) {

tempType = Exit;

}

else if (!strcmp(buf, "REPEAT")) {

tempType = Repeat;

}

else if (!strcmp(buf, "UNTIL")) {

tempType = Until;

}

else if (!strcmp(buf, "ADD")) {

tempType = Add;

}

else if (!strcmp(buf, "SUB")) {

tempType = Sub;

}

else if (!strcmp(buf, "DIV")) {

tempType = Div;

}

else if (!strcmp(buf, "MUL")) {

tempType = Mul;

}

else if (!strcmp(buf, "MOD")) {

tempType = Mod;

}

else if (!strcmp(buf, "EQ")) {

tempType = Equality;

}

else if (!strcmp(buf, "NE")) {

tempType = NotEquality;

}

else if (!strcmp(buf, "LT")) {

tempType = Less;

}

else if (!strcmp(buf, "GT")) {

tempType = Greate;

}

else if (!strcmp(buf, "END")) {

tempType = End;

}

else if (strlen(buf) <= 8 && buf[0] == '\_'

&& ('a' <= buf[1] && buf[1] <= 'z')

&& ('a' <= buf[1] && buf[1] <= 'z')

&& ('a' <= buf[1] && buf[1] <= 'z')

&& ('a' <= buf[1] && buf[1] <= 'z')

&& ('a' <= buf[1] && buf[1] <= 'z')

&& ('a' <= buf[1] && buf[1] <= 'z')

&& ('a' <= buf[1] && buf[1] <= 'z')

) {

tempType = Identifier;

}

strcpy\_s(tempToken.name, buf);

tempToken.type = tempType;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

case Another: {

switch (ch) {

case '(': {

strcpy\_s(tempToken.name, "(");

tempToken.type = LBracket;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case ')': {

strcpy\_s(tempToken.name, ")");

tempToken.type = RBracket;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case ',': {

strcpy\_s(tempToken.name, ",");

tempToken.type = Comma;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case ';': {

strcpy\_s(tempToken.name, ";");

tempToken.type = Semicolon;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case ':': {

strcpy\_s(tempToken.name, ":");

tempToken.type = Colon;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case '<': {

ch = getc(F);

if (ch == '<') {

}

else if (ch == '=') {

ch = getc(F);

if (ch == '=') {

strcpy\_s(tempToken.name, "<==");

tempToken.type = Assign;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

}

else {

strcpy\_s(tempToken.name, "<=");

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

}

}

else {

strcpy\_s(tempToken.name, "<");

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

}

break;

}

case '|': {

strcpy\_s(tempToken.name, "|");

tempToken.type = Or;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case '&': {

strcpy\_s(tempToken.name, "&");

tempToken.type = And;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case '!': {

ch = getc(F);

strcpy\_s(tempToken.name, "!");

tempToken.type = Not;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

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;

}

}

}

}

}

return TokensNum = tokenCount;

}

// Функція друкує таблицю лексем на екран

void LexicAnalyzer::printTokens(void) {

char type\_tokens[16];

printf("---------------------------------------------------------------------------\n");

printf("| TOKEN TABLE |\n");

printf("---------------------------------------------------------------------------\n");

printf("| line number | token | value | token code | type of token |\n");

printf("---------------------------------------------------------------------------\n");

for (unsigned int i = 0; i < TokensNum; i++) {

strcpy\_s(type\_tokens, lexemeTypeName(TokenTable[i].type));

printf("|%12d |%16s |%11d |%11d | %-13s |\n",

TokenTable[i].line,

TokenTable[i].name,

TokenTable[i].value,

TokenTable[i].type,

type\_tokens);

printf("---------------------------------------------------------------------------\n");

}

}

void LexicAnalyzer::fprintTokens(FILE\* F) {

char type\_tokens[16];

fprintf(F, "\n\n---------------------------------------------------------------------------\n");

fprintf(F, "| TOKEN TABLE |\n");

fprintf(F, "---------------------------------------------------------------------------\n");

fprintf(F, "| line number | token | value | token code | type of token |\n");

fprintf(F, "---------------------------------------------------------------------------\n");

for (unsigned int i = 0; i < TokensNum; i++) {

strcpy\_s(type\_tokens, lexemeTypeName(TokenTable[i].type));

fprintf(F, "|%12d |%16s |%11d |%11d | %-13s |\n",

TokenTable[i].line,

TokenTable[i].name,

TokenTable[i].value,

TokenTable[i].type,

type\_tokens);

fprintf(F, "---------------------------------------------------------------------------\n");

}

}  
 **Codegen.hpp**

#pragma once

#include "header.hpp"

#include <stdio.h>

#include <stdlib.h>

namespace Codegen {

void codegen(FILE\* outFile, struct astNode\* node);

}

**Parser.cpp**

#include "Parser.hpp"

#include <stdbool.h>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

extern struct Token\* TokenTable;

extern unsigned int TokensNum;

extern struct id\* idTable;

extern unsigned int idNum;

extern struct id\* labelTable;

extern unsigned int labelNum;

extern FILE\* errorFile;

int pos = 0;

namespace Parser {

void Parser() {

program();

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

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

}

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);

}

void match(enum TypeOfToken expectedType) {

if (TokenTable[pos].type == expectedType)

pos++;

else {

printf("\nSyntax error in line %d, token number: %d : another type of lexeme was expected (expected: %s | current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(expectedType), lexemeTypeName(TokenTable[pos].type));

fprintf(errorFile, "\nSyntax error in line %d, token number: %d : another type of lexeme was expected (expected: %s | current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(expectedType), lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

unsigned int IdIdentification(struct id idTable[], struct Token TokenTable[], unsigned int tokenCount) {

unsigned int idCount = 0;

unsigned int i = 0;

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

while (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);

fprintf(errorFile, "\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");

fprintf(errorFile, "\nToo many identifiers !\n");

return idCount;

}

}

else

++i;

}

++i;

}

for (; i < tokenCount; ++i) {

if (

TokenTable[i].type == Identifier

&& TokenTable[i - 1].type != Goto

&& TokenTable[i + 1].type != Colon

) {

bool yes = 0;

for (unsigned int j = 0; j < idCount; ++j) {

if (!strcmp(TokenTable[i].name, idTable[j].name)) {

yes = 1;

break;

}

}

if (!yes) {

printf("\nSemantic Error In line %d, an undeclared identifier \"%s\"!", TokenTable[i].line, TokenTable[i].name);

fprintf(errorFile, "\nSemantic Error In line %d, an undeclared identifier \"%s\"!", TokenTable[i].line, TokenTable[i].name);

exit(1);

}

}

}

return idCount;

}

void program() {

match(StartProgram);

if (TokenTable[pos].type == Variable) {

++pos; // for VARIABLE

variableDeclaration();

match(Semicolon);

}

match(StartBlock);

//match(Variable);

programBody();

match(EndBlock);

}

void variableDeclaration() {

match(Type);

variableList();

}

void variableList() {

match(Identifier);

while (TokenTable[pos].type == Comma) {

++pos; // for Comma

match(Identifier);

}

}

void statement() {

switch (TokenTable[pos].type) {

case Input: inputStatement(); break;

case Output: outputStatement(); break;

case If: ifStatement(); break;

case Goto: gotoStatement(); break;

case For: forStatement(); break;

case While: whileStatement(); break;

case Repeat: repeatStatement(); break;

case StartBlock:compoundStatement(); break;

case Continue:

++pos; // for CONTINUE

match(While); // Ensure it is followed by WHILE

break;

case Exit:

++pos; // for EXIT

match(While); // Ensure it is followed by WHILE

break;

default: {

if (TokenTable[pos + 1].type == Colon)

labelPoint();

else

assignStatement();

}

}

}

void inputStatement() {

++pos; // for Input

match(Identifier);

}

void outputStatement() {

++pos; // for Output

arithmeticExpression();

}

void arithmeticExpression() {

lowPriorityExpression();

//lowPriorityOperator

while (TokenTable[pos].type == Add || TokenTable[pos].type == Sub) {

++pos; // for add or sub

lowPriorityExpression();

}

}

void lowPriorityExpression() {

middlePriorityExpression();

// middlePriorityOperator

if (TokenTable[pos].type == Mul || TokenTable[pos].type == Mod || TokenTable[pos].type == Div) {

++pos; // for \* or / or %

middlePriorityExpression();

}

}

void middlePriorityExpression() {

switch (TokenTable[pos].type) {

case Identifier: ++pos; break;

case Number: ++pos; break;

case LBracket: {

++pos; // for (

arithmeticExpression();

match(RBracket);

break;

}

default: {

printf("\nSyntax error in line %d, token number: %d : middle priority operation was expected (current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));

fprintf(errorFile, "\nSyntax error in line %d, token number: %d : middle priority operation was expected (current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

}

void assignStatement() {

match(Identifier);

match(Assign);

arithmeticExpression();

}

void ifStatement() {

match(If);

match(LBracket);

logicalExpression();

match(RBracket);

// Check if the if body is empty or parse a statement

if (TokenTable[pos].type != Else && TokenTable[pos].type != Semicolon) {

statement(); // Parse the if body

}

if (TokenTable[pos].type == Semicolon) {

++pos; // Consume optional semicolon after the if body

}

// Handle the else part

if (TokenTable[pos].type == Else) {

++pos; // Consume ELSE token

// Check if the else body is empty or parse a statement

if (TokenTable[pos].type != Semicolon) {

statement(); // Parse the else body

match(Semicolon);

}

if (TokenTable[pos].type == Semicolon) {

++pos; // Consume optional semicolon after the else body

}

}

}

void logicalExpression() {

andExpression();

while (TokenTable[pos].type == Or) {

++pos; // for or

andExpression();

}

}

void andExpression() {

comparison();

while (TokenTable[pos].type == And) {

++pos; // for and

andExpression();

}

}

void comparison() {

if (TokenTable[pos].type == Not) {

++pos; // for not

match(LBracket);

logicalExpression();

match(RBracket);

}

else if (TokenTable[pos].type == LBracket) {

++pos; // for (

logicalExpression();

match(RBracket);

} else {

comparisonExpression();

}

}

void comparisonExpression() {

arithmeticExpression();

if (TokenTable[pos].type == Equality ||

TokenTable[pos].type == NotEquality ||

TokenTable[pos].type == Greate ||

TokenTable[pos].type == Less

) {

++pos;

}

else {

printf("\nSyntax error in line %d : Comparison operator was Expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

fprintf(errorFile, "\nSyntax error in line %d : Comparison operator was Expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

arithmeticExpression();

}

void gotoStatement() {

match(Goto);

match(Identifier);

}

void labelPoint() {

match(Identifier);

match(Colon);

}

//void forStatement() {

// match(For);

// assignStatement();

// if (TokenTable[pos].type == To || TokenTable[pos].type == Downto) {

// ++pos; // for to or downto

// }

// else {

// printf("\nSyntax error in line %d : TO or DOWNTO was expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

// fprintf(errorFile, "\nSyntax error in line %d : TO or DOWNTO was expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

// exit(1);

// }

// arithmeticExpression();

// match(Do);

// statement();

//}

void forStatement() {

match(For); // Match the "FOR" keyword

assignStatement(); // Parse the initialization statement (e.g., i := 1)

// Parse the range: TO or DOWNTO

if (TokenTable[pos].type == To || TokenTable[pos].type == Downto) {

++pos; // Advance past "TO" or "DOWNTO"

}

else {

printf("\nSyntax error in line %d: TO or DOWNTO was expected, but '%s' token gained.\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

fprintf(errorFile, "\nSyntax error in line %d: TO or DOWNTO was expected, but '%s' token gained.\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

arithmeticExpression(); // Parse the range's limit (e.g., 10)

match(Do); // Match the "DO" keyword

// Parse all statements between "DO" and ";"

while (TokenTable[pos].type != Semicolon) {

statement(); // Parse each statement in the loop body

}

match(Semicolon); // Ensure the loop ends with a semicolon

}

void whileStatement() {

match(While);

logicalExpression();

while (TokenTable[pos].type != End) {

statementInWhile();

//match(Semicolon);

}

//match(WEnd);

++pos; // for WEND

}

void statementInWhile() {

switch (TokenTable[pos].type) {

case Continue: {

++pos; // for CONTINUE

match(While);

break;

}

case Exit: {

++pos; // for EXIT

match(While);

break;

}

default: {

statement();

break;

}

}

}

void repeatStatement() {

match(Repeat);

while (TokenTable[pos].type != Until) {

statement();

//match(Semicolon);

}

//match(Until);

++pos; // for UNTIL

match(LBracket);

logicalExpression();

match(RBracket);

}

void compoundStatement() {

match(StartBlock);

programBody();

match(EndBlock);

}

void programBody() {

while (TokenTable[pos].type != EndBlock) {

statement();

//match(Semicolon);

}

}

void printIdentifiers(int num, struct id\* table) {

for (int i = 0; i < num; i++) {

printf("%s\n", table[i].name);

}

}

void fprintIdentifiers(FILE\* F, int num, struct id\* table) {

for (int i = 0; i < num; i++) {

fprintf(F, "%s\n", table[i].name);

}

}

unsigned int LabelIdentification(struct id labelTable[], struct Token TokenTable[], unsigned int tokenCount) {

unsigned int labelNum = 0;

unsigned int i = 0;

unsigned int start = 0;

while (TokenTable[start++].type != StartProgram);

i = start;

while (TokenTable[i].type != EndBlock) {

if (TokenTable[i].type == Identifier && TokenTable[i + 1].type == Colon) {

strcpy\_s(labelTable[labelNum++].name, TokenTable[i].name);

}

++i;

}

i = start;

while (TokenTable[i].type != EndBlock) {

if (TokenTable[i].type == Identifier && TokenTable[i - 1].type == Goto) {

bool found = false;

for (unsigned int j = 0; j < labelNum; j++) {

if (strcmp(labelTable[j].name, TokenTable[i].name) == 0) {

found = true;

break;

}

}

if (!found) {

printf("\n Semantic error: In line %d label %s is not defined\n", TokenTable[i].line, TokenTable[i].name);

fprintf(errorFile, "\n Semantic error: In line %d label %s is not defined\n", TokenTable[i].line, TokenTable[i].name);

exit(1);

}

}

++i;

}

return labelNum;

}

}

**Header.hpp**

#pragma once

#define MAX\_TOKENS 1000

#define MAX\_IDENTIFIER 10

enum TypeOfToken {

StartProgram,

StartBlock,

Variable,

Type,

EndBlock,

Input,

Output,

If,

Else,

Goto,

For,

To,

Downto,

Do,

While,

Continue,

Exit,

End,

Repeat,

Until,

Identifier,

Number,

Float,

Assign,

Add,

Sub,

Mul,

Div,

Mod,

Equality,

NotEquality,

Greate,

Less,

Not,

And,

Or,

LBracket,

RBracket,

Semicolon,

Colon,

Comma,

Unknown\_

};

struct Token {

char name[16];

int value;

int line;

enum TypeOfToken type;

};

struct id {

char name[16];

};

enum States {

Start,

Finish,

Letter,

Digit,

Separator,

Another,

EndOFile,

SComment,

Comment

};

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;

};

const char\* lexemeTypeName(enum TypeOfToken type);

**Parser.hpp**

#pragma once

#include "header.hpp"

#include <stdbool.h>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

extern struct Token\* TokenTable;

extern unsigned int TokensNum;

extern struct id\* idTable;

extern unsigned int idNum;

extern struct id\* labelTable;

extern unsigned int labelNum;

namespace Parser {

void Parser();

void Semantic();

void match(enum TypeOfToken expectedType);

unsigned int IdIdentification(struct id idTable[], struct Token TokenTable[], unsigned int tokenCount);

unsigned int LabelIdentification(struct id labelTable[], struct Token TokenTable[], unsigned int tokenCount);

void printIdentifiers(int num, struct id\* table);

void fprintIdentifiers(FILE\* F, int num, struct id\* table);

void program();

void programBody();

void variableDeclaration();

void variableList();

void statement();

void inputStatement();

void outputStatement();

void arithmeticExpression();

void lowPriorityExpression();

void middlePriorityExpression();

void assignStatement();

void ifStatement();

void logicalExpression();

void andExpression();

void comparison();

void comparisonExpression();

void gotoStatement();

void labelPoint();

void forStatement();

void whileStatement();

void statementInWhile();

void repeatStatement();

void compoundStatement();

}

**Ast.hpp**

#pragma once

#include "header.hpp"

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

extern struct Token\* TokenTable;

extern int pos;

namespace AST {

struct astNode\* program();

struct astNode\* programBody();

struct astNode\* variableDeclaration();

struct astNode\* variableList();

struct astNode\* statement();

struct astNode\* inputStatement();

struct astNode\* outputStatement();

struct astNode\* arithmeticExpression();

struct astNode\* lowPriorityExpression();

struct astNode\* middlePriorityExpression();

struct astNode\* assignStatement();

struct astNode\* ifStatement();

struct astNode\* logicalExpression();

struct astNode\* andExpression();

struct astNode\* comparison();

struct astNode\* comparisonExpression();

struct astNode\* gotoStatement();

struct astNode\* labelPoint();

struct astNode\* forStatement();

struct astNode\* whileStatement();

struct astNode\* statementInWhile();

struct astNode\* whileBody();

struct astNode\* repeatStatement();

struct astNode\* repeatBody();

struct astNode\* compoundStatement();

void deleteNode(struct astNode\* node);

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

void printAST(struct astNode\* node, int level);

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

void match(enum TypeOfToken expectedType);

struct astNode\* astParser();

}

**Codegen.cpp**

#include "Codegen.hpp"

namespace Codegen {

void codegen(FILE\* outFile, struct astNode\* node) {

if (node == 0) {

return;

}

switch (node->type) {

case program\_node: {

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

codegen(outFile, node->left); // Оголошення змінних

codegen(outFile, node->right); // Тіло програми

fprintf(outFile, " system(\"pause\");\n ");

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

break;

}

case var\_node: {

fprintf(outFile, "int ");

codegen(outFile, node->left);

fprintf(outFile, ";\n");

codegen(outFile, node->right);

break;

}

case number\_node:

case id\_node: {

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

break;

}

case statement\_node: {

codegen(outFile, node->left);

codegen(outFile, node->right);

break;

}

case input\_node: {

fprintf(outFile, "printf(\"Enter ");

codegen(outFile, node->left);

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

fprintf(outFile, "scanf(\"%%d\", &");

codegen(outFile, node->left);

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

break;

}

case output\_node: {

fprintf(outFile, "printf(\"%%d\\n\", ");

codegen(outFile, node->left);

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

break;

}

case add\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " + ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case sub\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " - ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case mul\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " \* ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case div\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " / ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case mod\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " %% ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case assign\_node: {

codegen(outFile, node->left);

fprintf(outFile, " = ");

codegen(outFile, node->right);

fprintf(outFile, ";\n");

break;

}

case if\_node: {

fprintf(outFile, "if (");

codegen(outFile, node->left); // Condition

fprintf(outFile, ") ");

if (node->right == NULL || node->right->type != else\_node) {

// If the body is empty or there's no else, write an empty block

fprintf(outFile, "{\n");

}

else {

// If there is an else block, ensure it is properly handled

struct astNode\* ifBody = node->right->left; // Body of the if statement

struct astNode\* elseBody = node->right->right; // Body of the else statement

if (ifBody != NULL) {

fprintf(outFile, "{\n");

codegen(outFile, ifBody);

fprintf(outFile, "}\n");

}

else {

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

}

fprintf(outFile, "else ");

if (elseBody != NULL) {

fprintf(outFile, "{\n");

codegen(outFile, elseBody);

fprintf(outFile, "}\n");

}

else {

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

}

break;

}

if (node->right != NULL) {

codegen(outFile, node->right); // Process any non-else content

}

fprintf(outFile, "}\n");

break;

}

case else\_node: {

codegen(outFile, node->left);

fprintf(outFile, "else ");

fprintf(outFile, "{ \n");

codegen(outFile, node->right);

fprintf(outFile, "} \n");

break;

}

case or\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " || ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case and\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " && ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case not\_node: {

fprintf(outFile, "!(");

codegen(outFile, node->left);

fprintf(outFile, ")");

break;

}

case eq\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " == ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case neq\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " != ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case gr\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " > ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case ls\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " < ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case goto\_node: {

fprintf(outFile, "goto ");

codegen(outFile, node->left);

fprintf(outFile, ";\n");

break;

}

case label\_node: {

codegen(outFile, node->left);

fprintf(outFile, ":\n");

break;

}

case for\_node: {

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

codegen(outFile, node->left);

fprintf(outFile, "\n) ");

fprintf(outFile, "{\n");

codegen(outFile, node->right);

fprintf(outFile, "\n}");

break;

}

case to\_node: {

codegen(outFile, node->left);

codegen(outFile, node->left->left);

fprintf(outFile, " <= ");

codegen(outFile, node->right);

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

codegen(outFile, node->left->left);

break;

}

case downto\_node: {

codegen(outFile, node->left);

codegen(outFile, node->left->left);

fprintf(outFile, " >= ");

codegen(outFile, node->right);

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

codegen(outFile, node->left->left);

break;

}

case while\_node: {

fprintf(outFile, "while(");

codegen(outFile, node->left);

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

codegen(outFile, node->right);

fprintf(outFile, "}\n");

break;

}

case continue\_node: {

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

break;

}

case exit\_node: {

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

break;

}

case repeat\_node: {

fprintf(outFile, "do {\n");

codegen(outFile, node->left);

fprintf(outFile, "} while(");

codegen(outFile, node->right);

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

break;

}

case compound\_node: {

fprintf(outFile, "{\n");

codegen(outFile, node->left);

codegen(outFile, node->right);

fprintf(outFile, "}\n");

break;

}

default: {

exit(1);

printf("Undescribed node type: %d\n", node->type);

break;

}

}

}

}  
**Ast.cpp**

#include "Ast.hpp"

extern struct Token\* TokenTable;

extern int pos;

namespace AST {

void deleteNode(struct astNode\* node) {

if (node == nullptr) return;

deleteNode(node->left);

deleteNode(node->right);

free(node);

}

struct astNode\* createNode(enum TypeOfNode type, const char\* name, struct astNode\* left, struct astNode\* right) {

struct astNode\* node = (struct astNode\*)malloc(sizeof(struct astNode));

node->type = type;

strcpy\_s(node->name, name);

node->left = left;

node->right = right;

return node;

}

void printAST(struct astNode\* node, int level) {

if (node == nullptr)

return;

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

printf("| ");

printf("|-- %s(%d)", node->name, node->type);

printf("\n");

if (node->left || node->right)

{

printAST(node->left, level + 1);

printAST(node->right, level + 1);

}

}

void fPrintAST(FILE\* outFile, struct astNode\* node, int level) {

if (node == nullptr)

return;

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

fprintf(outFile, "| ");

fprintf(outFile, "|-- %s(%d)", node->name, node->type);

fprintf(outFile, "\n");

if (node->left || node->right)

{

fPrintAST(outFile, node->left, level + 1);

fPrintAST(outFile, node->right, level + 1);

}

}

void match(enum TypeOfToken expectedType) {

if (TokenTable[pos].type == expectedType)

pos++;

else {

printf("\nSyntax error in line %d : another type of lexeme was expected (expected: %s | current: %s).\n", TokenTable[pos].line, lexemeTypeName(expectedType), lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

struct astNode\* astParser() {

pos = 0;

struct astNode\* tree = program();

printf("AST created.\n");

return tree;

}

struct astNode\* program() {

match(StartProgram);

struct astNode\* declaration = nullptr;

if (TokenTable[pos].type == Variable) {

++pos; // for VARIABLE

declaration = variableDeclaration();

match(Semicolon);

}

match(StartBlock);

struct astNode\* body = programBody();

match(EndBlock);

return createNode(program\_node, "Program", declaration, body);

}

struct astNode\* variableDeclaration() {

match(Type);

return variableList();

}

struct astNode\* variableList() {

match(Identifier);

struct astNode\* id = createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr);

struct astNode\* list = list = createNode(var\_node, "var", id, nullptr);

while (TokenTable[pos].type == Comma)

{

match(Comma);

match(Identifier);

id = createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr);

list = createNode(var\_node, "var", id, list);

}

return list;

}

struct astNode\* programBody() {

if (TokenTable[pos].type != EndBlock) {

struct astNode\* stmt = statement();

//match(Semicolon);

struct astNode\* body = stmt;

while (TokenTable[pos].type != EndBlock)

{

struct astNode\* nextStmt = statement();

//match(Semicolon);

body = createNode(statement\_node, "statement", body, nextStmt);

}

return body;

}

return nullptr;

}

struct astNode\* statement() {

switch (TokenTable[pos].type) {

case Input: return inputStatement();

case Output: return outputStatement();

case If: return ifStatement();

case Goto: return gotoStatement();

case For: return forStatement();

case While: return whileStatement();

case Repeat: return repeatStatement();

case StartBlock:return compoundStatement();

case Continue:

++pos; // for CONTINUE

match(While); // Ensure it is followed by WHILE

return createNode(continue\_node, "continue", nullptr, nullptr);

case Exit:

++pos; // for EXIT

match(While); // Ensure it is followed by WHILE

return createNode(exit\_node, "exit", nullptr, nullptr);

default: {

if (TokenTable[pos + 1].type == Colon)

return labelPoint();

else

return assignStatement();

}

}

}

struct astNode\* inputStatement() {

match(Input);

match(Identifier);

return createNode(input\_node, "input", createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr), nullptr);

}

struct astNode\* outputStatement() {

match(Output);

return createNode(output\_node, "output", arithmeticExpression(), nullptr);

}

struct astNode\* arithmeticExpression() {

struct astNode\* left = lowPriorityExpression();

if (TokenTable[pos].type == Add || TokenTable[pos].type == Sub) {

enum TypeOfToken op = TokenTable[pos].type;

++pos; // for add or sub

struct astNode\* right = arithmeticExpression();

return createNode(op == Add ? add\_node : sub\_node, lexemeTypeName(op), left, right);

}

return left;

}

struct astNode\* lowPriorityExpression() {

struct astNode\* left = middlePriorityExpression();

if (TokenTable[pos].type == Mul || TokenTable[pos].type == Mod || TokenTable[pos].type == Div) {

enum TypeOfToken op = TokenTable[pos].type;

++pos; // for mul or mod or div

struct astNode\* right = lowPriorityExpression();

return createNode(op == Mul ? mul\_node : op == Div ? div\_node : mod\_node, lexemeTypeName(op), left, right);

}

return left;

}

struct astNode\* middlePriorityExpression() {

switch (TokenTable[pos].type) {

case Identifier: match(Identifier); return createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr);

case Number: match(Number); return createNode(number\_node, TokenTable[pos - 1].name, nullptr, nullptr);

case LBracket: {

++pos; // for (

struct astNode\* expr = arithmeticExpression();

match(RBracket);

return expr;

}

default: {

printf("\nSyntax error in line %d, token number: %d : middle priority operation was expected (current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

}

struct astNode\* assignStatement() {

const char\* varName = TokenTable[pos].name;

match(Identifier);

match(Assign);

struct astNode\* expr = arithmeticExpression();

return createNode(assign\_node, "assign", createNode(id\_node, varName, nullptr, nullptr), expr);

}

struct astNode\* ifStatement() {

match(If);

match(LBracket);

struct astNode\* expr = logicalExpression();

match(RBracket);

// Allow the if body to be empty

struct astNode\* stmt = NULL;

if (TokenTable[pos].type != Else && TokenTable[pos].type != Semicolon) {

stmt = statement();

}

if (TokenTable[pos].type == Semicolon) {

++pos; // Consume optional semicolon after if body

}

// Handle the else part, if present

if (TokenTable[pos].type == Else) {

++pos; // Consume ELSE token

struct astNode\* elseStmt = NULL;

if (TokenTable[pos].type != Semicolon) {

elseStmt = statement();

match(Semicolon);

}

if (TokenTable[pos].type == Semicolon) {

++pos; // Consume optional semicolon after else body

}

stmt = createNode(else\_node, "else", stmt, elseStmt);

}

return createNode(if\_node, "if", expr, stmt);

}

struct astNode\* logicalExpression() {

struct astNode\* left = andExpression();

if (TokenTable[pos].type == Or) {

++pos; // for Or

struct astNode\* right = logicalExpression();

return createNode(or\_node, "or", left, right);

}

return left;

}

struct astNode\* andExpression() {

struct astNode\* left = comparison();

if (TokenTable[pos].type == And) {

++pos; // for And

struct astNode\* right = andExpression();

return createNode(and\_node, "and", left, right);

}

return left;

}

struct astNode\* comparison() {

struct astNode\* comp;

switch (TokenTable[pos].type) {

case Not: {

++pos; // for Not

match(LBracket);

comp = createNode(not\_node, "not", logicalExpression(), nullptr);

match(RBracket);

break;

}

case LBracket: {

++pos; // for (

comp = logicalExpression();

match(RBracket);

break;

}

default: {

comp = comparisonExpression();

}

}

return comp;

}

struct astNode\* comparisonExpression() {

struct astNode\* left = arithmeticExpression();

if (TokenTable[pos].type == Equality ||

TokenTable[pos].type == NotEquality ||

TokenTable[pos].type == Greate ||

TokenTable[pos].type == Less

) {

enum TypeOfToken op = TokenTable[pos].type;

++pos; // for Equality or NotEquality or Greate or Less

struct astNode\* right = arithmeticExpression();

return createNode(op == Equality ? eq\_node : op == NotEquality ? neq\_node : op == Greate ? gr\_node : ls\_node, lexemeTypeName(op), left, right);

}

else {

printf("\nSyntax error in line %d : Comparison operator was Expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

struct astNode\* gotoStatement() {

match(Goto);

match(Identifier);

return createNode(goto\_node, "goto", createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr), nullptr);

}

struct astNode\* labelPoint() {

match(Identifier);

match(Colon);

return createNode(label\_node, "label", createNode(id\_node, TokenTable[pos - 2].name, nullptr, nullptr), nullptr);

}

// struct astNode\* forStatement() {

// match(For);

// struct astNode\* assign = assignStatement();

// switch (TokenTable[pos].type) {

// case To: {

// ++pos; // for To

// struct astNode\* expr = arithmeticExpression();

// match(Do);

// return createNode(for\_node, "for", createNode(to\_node, "to", assign, expr), statement());

// }

// case Downto: {

// ++pos; // for Downto

// struct astNode\* expr = arithmeticExpression();

//match(Do);

// return createNode(for\_node, "for", createNode(to\_node, "downto", assign, expr), statement());

// }

// default: {

// printf("\nSyntax error in line %d : TO | DOWNTO operator was xpected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

// exit(1);

// }

// }

// }

struct astNode\* forStatement() {

match(For);

// Parse the initialization (e.g., i := 1)

struct astNode\* assign = assignStatement();

// Parse the range (e.g., To or Downto and the expression)

struct astNode\* range;

if (TokenTable[pos].type == To) {

++pos; // for "To"

struct astNode\* expr = arithmeticExpression();

range = createNode(to\_node, "to", assign, expr);

}

else if (TokenTable[pos].type == Downto) {

++pos; // for "Downto"

struct astNode\* expr = arithmeticExpression();

range = createNode(downto\_node, "downto", assign, expr);

}

else {

printf("\nSyntax error in line %d: 'to' or 'downto' operator was expected, but '%s' token found.\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

match(Do); // Match the "DO" keyword

// Parse all statements between "DO" and the semicolon (";")

struct astNode\* body = nullptr;

struct astNode\* last = nullptr;

while (TokenTable[pos].type != Semicolon) {

struct astNode\* stmt = statement();

// Append this statement to the body tree

if (body == nullptr) {

body = stmt; // First statement

}

else {

last = createNode(statement\_node, "statement", body, stmt);

body = last;

}

}

match(Semicolon); // Ensure the loop ends with a semicolon

// Return the FOR node, combining the range and the body

return createNode(for\_node, "for", range, body);

}

struct astNode\* whileStatement() {

match(While);

struct astNode\* expr = logicalExpression();

struct astNode\* body = whileBody();

match(End);

return createNode(while\_node, "while", expr, body);

}

struct astNode\* whileBody() {

if (TokenTable[pos].type !=End) {

struct astNode\* stmt = statementInWhile();

//match(Semicolon);

struct astNode\* body = stmt;

while (TokenTable[pos].type != End)

{

struct astNode\* nextStmt = statementInWhile();

//match(Semicolon);

body = createNode(statement\_node, "statement", body, nextStmt);

}

return body;

}

return nullptr;

}

struct astNode\* statementInWhile() {

switch (TokenTable[pos].type) {

case Continue: {

++pos; // for continue

match(While);

return createNode(continue\_node, "continue", nullptr, nullptr);

}

case Exit: {

++pos; // for exit

match(While);

return createNode(exit\_node, "exit", nullptr, nullptr);

}

default: return statement();

}

}

struct astNode\* repeatStatement() {

match(Repeat);

struct astNode\* body = repeatBody();

match(Until);

match(LBracket);

struct astNode\* expr = logicalExpression();

match(RBracket);

return createNode(repeat\_node, "repeat", body, expr);

}

struct astNode\* repeatBody() {

if (TokenTable[pos].type != Until) {

struct astNode\* stmt = statement();

//match(Semicolon);

struct astNode\* body = stmt;

while (TokenTable[pos].type != Until)

{

struct astNode\* nextStmt = statement();

//match(Semicolon);

body = createNode(statement\_node, "statement", body, nextStmt);

}

return body;

}

return nullptr;

}

struct astNode\* compoundStatement() {

match(StartBlock);

struct astNode\* body = programBody();

match(EndBlock);

return createNode(compound\_node, "compound", body, nullptr);

}

}

**Додаток Д.**

