

Міністерство освіти і науки України
Національний університет “Львівська Політехніка”
Кафедра ЕОМ



Пояснювальна записка

до курсового проєкту “СИСТЕМНЕ ПРОГРАМУВАННЯ”

на тему : “РОЗРОБКА СИСТЕМНИХ ПРОГРАМНИХ МОДУЛІВ ТА КОМПОНЕНТ
СИСТЕМ ПРОГРАМУВАННЯ”

Індивідуальне завдання

“РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ”

Виконав студент групи КІ-307:

Коростенська С.В.

Перевірив:

старший викладач каф. ЕОМ

Козак Н.Б.

ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ

1. Цільова мова транслятора – мова програмування C або асемблер для 32/64 розрядного процесора.
2. Для отримання виконавчого файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio бо будь-яким іншим.
3. Мова розробки транслятора: C/C++.
4. Реалізувати оболонку або інтерфейс з командного рядка.
5. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.
6. На виході розробленого транслятора мають створюватись такі файли:
 - *файл з лексемами;*
 - *файл з повідомленнями про помилки (або про їх відсутність);*
 - *файл на мові асемблера;*
 - *об'єктний файл;*
 - *виконавчий файл.*
7. Назва вхідної мови програмування утворюється від першої букви у прізвищі студента та останніх двох цифр номера його варіанту. Саме таке розширення повинні мати текстові файли, написані на цій мові програмування.

Деталізація завдання на проєктування:

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

Пріоритет операцій наступний - круглі дужки (), логічне заперечення, мультиплікативні (множення, ділення, залишок від ділення), адитивні

(додавання, віднімання), відношення (більше, менше), перевірка на рівність і нерівність, логічне І, логічне АБО.

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

Деталізований опис власної мови програмування:

- Тип даних: INTEGER16_t
- Блок тіла програми: #PROGRAM; VARIABLE...; START-STOP
- Оператор вводу: SCAN ()
- Оператор виводу: PRINT ()
- Оператори: IF ELSE (C)
GOTO (C)
FOR-TO (Паскаль)
FOR-DOWNT0 (Паскаль)
WHILE (Бейсік)
REPEAT-UNTIL (Паскаль)

- Регістр ключових слів: Low-Up8 перший символ Low
- Регістр ідентифікаторів: Low-Up8 перший символ Low
- Операції арифметичні: ADD, SUB, MUL, DIV, MOD
- Операції порівняння: ==, !=, !>, !<
- Операції логічні: NOT, AND, OR
- Коментар: \\\...
- Ідентифікатори змінних, числові константи
- Оператор присвоєння: <=

АНОТАЦІЯ

У цьому курсовому проєкті розроблено транслятор, що забезпечує конвертацію вхідної мови, заданої варіантом, у мову асемблера. Процес трансляції охоплює три основні етапи: лексичний аналіз, синтаксичний аналіз та генерацію коду.

Лексичний аналіз передбачає поділ вхідного потоку символів на лексеми, які заносяться до спеціальної таблиці. Кожній лексемі присвоюється унікальний числовий ідентифікатор для полегшення обробки, а також додається додаткова інформація: номер рядка, значення (якщо лексема є числом) та інші важливі параметри.

Синтаксичний аналіз використовує висхідний метод без повернення для побудови дерева розбору, просуваючись від листків до кореня. Цей етап дозволяє забезпечити коректне структурування вхідних даних відповідно до синтаксичних правил мови.

Генерація коду полягає в обробці таблиці лексем для створення асемблерного коду, що відповідає кожному блоку. Згенерований код зберігається у вихідному файлі та є готовим до подальшої компіляції.

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

ЗМІСТ

АНОТАЦІЯ	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
ВСТУП	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
2. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
2.1. ДЕТАЛІЗОВАНИЙ ОПИС ВХІДНОЇ МОВИ В ТЕРМІНАХ РОЗШИРЕНОЇ НОТАЦІЇ БЕКУСА-НАУРА.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.1. ВИБІР ТЕХНОЛОГІЇ ПРОГРАМУВАННЯ.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.2. ПРОЄКТУВАННЯ ТАБЛИЦЬ ТРАНСЛЯТОРА ТА ВИБІР СТРУКТУР ДАНИХ.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.3. РОЗРОБКА ЛЕКСИЧНОГО АНАЛІЗАТОРА.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.3.1. Розробка алгоритму роботи лексичного аналізатора.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.3.2. Опис програми реалізації лексичного аналізатора.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.4. РОЗРОБКА СИНТАКСИЧНОГО ТА СЕМАНТИЧНОГО АНАЛІЗАТОРА.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.4.1. Розробка дерева граматичного розбору.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.4.2. Розробка алгоритму роботи синтаксичного і семантичного аналізатора.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.4.3. Опис програми реалізації синтаксичного та семантичного аналізатора.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.5. РОЗРОБКА ГЕНЕРАТОРА КОДУ.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.5.1. Розробка алгоритму роботи генератора коду..	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
3.5.2. Опис програми реалізації генератора коду.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
4.1. ОПИС ІНТЕРФЕЙСУ ТА ІНСТРУКЦІЇ КОРИСТУВАЧУ.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
4.2. ВИЯВЛЕННЯ ЛЕКСИЧНИХ І СИНТАКСИЧНИХ ПОМИЛОК.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
4.3. ПЕРЕВІРКА РОБОТИ ТРАНСЛЯТОРА ЗА ДОПОМОГОЮ ТЕСТОВИХ ЗАДАЧ.	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
ВИСНОВКИ	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ.....	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.
ДОДАТКИ	ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.

ВСТУП

Термін «транслятор» позначає програмне забезпечення, що виконує перетворення вихідного коду, написаного на одній мові програмування, у еквівалентний код іншої мови. Якщо вхідною мовою є мова високого рівня, а вихідною — мова асемблера або машинний код, такий транслятор називається компілятором.

Транслятори поділяються на два основні типи: компілятори та інтерпретатори. Процес компіляції складається з двох ключових етапів: аналізу та синтезу. Під час аналізу вихідний код розбивається на окремі елементи (лексеми), перевіряється його синтаксична коректність та створюється проміжне представлення програми. На етапі синтезу це представлення перетворюється на об'єктний код, що складається з машинних інструкцій і може бути виконаний безпосередньо на комп'ютері.

Інтерпретатори, на відміну від компіляторів, не генерують окремий виконуваний файл. Вони аналізують програму, створюють її проміжне представлення, але не синтезують об'єктний код. Натомість інтерпретатор виконує інструкції вхідної програми безпосередньо під час її обробки.

Компілятор забезпечує перетворення вихідного коду з однієї мови програмування в іншу. Вхідними даними для компілятора є послідовність символів, що представляє вихідну програму, а вихідними — об'єктний код, адаптований для конкретного апаратного середовища. Цікаво, що сам компілятор може бути реалізований на третій мові програмування, що підкреслює його універсальність і гнучкість у розробці програмного забезпечення.

1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ

Транслятор — це програма, що виконує перетворення вихідного коду, написаного на одній мові програмування, у робочий код, представлений об'єктною мовою. Це загальне визначення охоплює різні типи транслуючих програм, кожна з яких має свої особливості реалізації процесу трансляції. Сучасні транслятори класифікуються на три основні категорії: асемблери, компілятори та інтерпретатори.

Асемблер — це утиліта, яка перетворює символічний код у машинні інструкції. Головною рисою асемблера є пряме відображення кожної символічної команди у відповідну машинну інструкцію.

Компілятор — це програма, що транслює вихідний код, написаний мовою програмування високого рівня, у машинний код. На відміну від асемблера, компілятор здійснює складніше перетворення з однієї мови на іншу, зазвичай на рівень машинної мови конкретного процесора.

Інтерпретатор — це програма, яка виконує вихідний код послідовно, обробляючи кожну інструкцію окремо. На відміну від компілятора, інтерпретатор не створює готовий виконуваний файл, а негайно виконує інструкції, що забезпечує більш гнучкий підхід до тестування та налагодження програм.

Процес трансляції складається з кількох ключових етапів: лексичного аналізу, синтаксичного аналізу, семантичного аналізу, оптимізації коду та генерації коду.

- **Лексичний аналіз** розбиває вихідний код на окремі лексеми, що відповідають словам і символам мови програмування. Під час цього етапу також можуть бути виявлені базові помилки, такі як некоректні символи чи невірний формат ідентифікаторів.

- **Синтаксичний аналіз** будує синтаксичне дерево, яке відображає структуру програми відповідно до правил контекстно-вільної граматики. Популярними методами аналізу є $LL(1)$ та $LR(1)$, кожен із яких має свої варіанти для конкретних завдань.
- **Семантичний аналіз** визначає логічні залежності між елементами програми, які не можуть бути описані синтаксисом. Цей етап включає перевірку типів даних, областей видимості та відповідності параметрів функцій.
- **Оптимізація коду** спрямована на покращення ефективності виконання програми. Вона поділяється на локальну та глобальну, а також на машинно-залежну та машинно-незалежну.
- **Генерація коду** завершує процес трансляції, створюючи об'єктний або асемблерний код, готовий для подальшої компіляції та виконання.

Фази трансляції можуть об'єднуватися або навіть відсутні в залежності від реалізації конкретного транслятора. У спрощених однопрохідних трансляторах проміжне представлення та оптимізація можуть бути пропущені, а інші фази часто поєднуються.

На етапі лексичного аналізу формується таблиця об'єктів, що включає ідентифікатори, рядки та числові значення. Синтаксичний аналіз створює дерево розбору, яке потім використовується для подальшої оптимізації та генерації коду. У контекстному аналізі забезпечується відповідність типів, контроль областей видимості та коректність параметрів функцій.

Результатом оптимізації та генерації коду є ефективний і продуктивний об'єктний код, який може бути скомпільований для конкретної архітектури. Таким чином, кожен етап трансляції забезпечує послідовне перетворення вихідного коду у робочий виконуваний файл, готовий до запуску на цільовому пристрої.

2. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ

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

```
labeled_point = label , ":"
goto_label = tokenGOTO, label, ";"
program_name = ident, ";"
value_type = tokenINTEGER16_t
other_declaration_ident = tokenCOMMA , ident
declaration = value_type , ident , {other_declaration_ident}
unary_operator = tokenNOT | tokenMINUS | tokenPLUS
unary_operation = unary_operator , expression
binary_operator = tokenAND | tokenOR | tokenEQUAL | tokenNOTEQUAL |
tokenNOTGREATER | tokenNOTLESS | tokenADD | tokenSUB | tokenMUL | tokenDIV |
tokenMOD
binary_action = binary_operator , expression
left_expression = group_expression | unary_operation | ident | value
expression = left_expression , {binary_action}
group_expression = tokenGROUPEXPRESSIONBEGIN , expression ,
tokenGROUPEXPRESSIONEND
//
bind_right_to_left = ident , tokenRLBIND , expression
bind_left_to_right = expression , tokenLRBIND , ident
//
if_expression = expression
body_for_true = {statement} , ";"
body_for_false = tokenELSE , {statement} , ";"
cond_block = tokenIF , tokenGROUPEXPRESSIONBEGIN , if_expression ,
tokenGROUPEXPRESSIONEND , body_for_true , [body_for_false];
//
cycle_begin_expression = expression
cycle_counter = ident
cycle_counter_rl_init = cycle_counter , tokenRLBIND , cycle_begin_expression
cycle_counter_lr_init = cycle_begin_expression , tokenLRBIND , cycle_counter
cycle_counter_init = cycle_counter_rl_init | cycle_counter_lr_init
cycle_counter_last_value = value
cycle_body = tokenDO , statement , {statement}
forto_cycle = tokenFOR , cycle_counter_init , tokenTO , cycle_counter_last_value ,
cycle_body , ";"
fordownto_cycle = tokenFOR , cycle_counter_init , tokenDOWNTO ,
cycle_counter_last_value , cycle_body , ";"
continue_while = tokenCONTINUE , tokenWHILE
exit_while = tokenEXIT , tokenWHILE
statement_in_while_body = statement | continue_while | exit_while
while_cycle_head_expression = expression
```

```

while_cycle = tokenWHILE , while_cycle_head_expression , {statement_in_while_body} ,
tokenEND , tokenWHILE
//
repeat_until_cycle_cond = group_expression
repeat_until_cycle = tokenREPEAT , {statement} , tokenUNTIL , repeat_until_cycle_cond
input = tokenSCAN , tokenGROUPEXPRESSIONBEGIN , ident ,
tokenGROUPEXPRESSIONEND
output = tokenPRINT , tokenGROUPEXPRESSIONBEGIN , expression ,
tokenGROUPEXPRESSIONEND
statement = bind_right_to_left | bind_left_to_right | cond_block | forto_cycle |
fordownto_cycle | while_cycle | repeat_until_cycle | labeled_point | goto_label | input | output
program = tokenPROGRAM , program_name , tokenSEMICOLON , tokenVARIABLE ,
[declaration] , tokenSEMICOLON , {statement} , tokenSTART , tokenSTOP
//
digit = digit_0 | digit_1 | digit_2 | digit_3 | digit_4 | digit_5 | digit_6 | digit_7 | digit_8 |
digit_9
non_zero_digit = digit_1 | digit_2 | digit_3 | digit_4 | digit_5 | digit_6 | digit_7 | digit_8 |
digit_9
unsigned_value = ((non_zero_digit , {digit}) | digit_0)
value = [sign] , unsigned_value
// -- hello wolrd
letter_in_lower_case = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x |
y | z
letter_in_upper_case = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
V | W | X | Y | Z
ident = letter_in_lower_case , letter_in_upper_case , letter_in_upper_case ,
letter_in_upper_case , letter_in_upper_case , letter_in_upper_case , letter_in_upper_case ,
letter_in_upper_case
label = letter_in_lower_case , {letter_in_lower_case}
//
sign = sign_plus | sign_minus
sign_plus = '+'
sign_minus = '-'
//
digit_0 = '0'
digit_1 = '1'
digit_2 = '2'
digit_3 = '3'
digit_4 = '4'
digit_5 = '5'
digit_6 = '6'
digit_7 = '7'
digit_8 = '8'
digit_9 = '9'
//
tokenCOLON = ":"
tokenGOTO = "GOTO"

```

```
tokenINTEGER16_t = "INTEGER16_t"
tokenCOMMA = ","
tokenNOT = "NOT"
tokenAND = "AND"
tokenOR = "OR"
tokenEQUAL = "=="
tokenNOTEQUAL = "!="
tokenNOTGREATER = "!>"
tokenNOTLESS = "!<"
tokenADD = "ADD"
tokenSUB = "SUB"
tokenMUL = "MUL"
tokenDIV = "DIV"
tokenMOD = "MOD"
tokenGROUPEXPRESSIONBEGIN = "("
tokenGROUPEXPRESSIONEND = ")"
tokenRLBIND = "<-"
tokenLRBIND = ","
tokenELSE = "ELSE"
tokenIF = "IF"
tokenDO = "DO"
tokenFOR = "FOR"
tokenTO = "TO"
tokenDOWNTOW = "DOWNTOW"
tokenWHILE = "WHILE"
tokenCONTINUE = "CONTINUE"
tokenEXIT = "EXIT"
tokenREPEAT = "REPEAT"
tokenUNTIL = "UNTIL"
tokenSCAN = "SCAN"
tokenPRINT = "PRINT"
tokenPROGRAM = "#PROGRAM"
tokenVARIABLE = "VARIABLE"
tokenSTART = "START"
tokenSTOP = "STOP"
tokenSEMICOLON = ";"
//
tokenUNDERSCORE = "_"
//
A = "A"
B = "B"
C = "C"
D = "D"
E = "E"
F = "F"
G = "G"
H = "H"
```

```
I = "I"
J = "J"
K = "K"
L = "L"
M = "M"
N = "N"
O = "O"
P = "P"
Q = "Q"
R = "R"
S = "S"
T = "T"
U = "U"
V = "V"
W = "W"
X = "X"
Y = "Y"
Z = "Z"
```

```
//
```

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

```
//
```

3. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ

3.1. Вибір технології програмування.

Перш ніж приступати до розробки програми, для прискорення та підвищення ефективності її створення, необхідно спланувати алгоритм функціонування програми та обрати відповідну технологію й середовище програмування.

Для виконання поставленого завдання найкращим вибором є середовище програмування Microsoft Visual Studio 2022 і мова програмування C/C++.

З метою забезпечення зручності та простоти використання програми користувачем було вирішено створити консольний інтерфейс.

3.2. Проектування таблиць транслятора та вибір структур даних.

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

```
struct LexemInfo {public:
    char lexemStr[MAX_LEXEM_SIZE];
    unsigned long long int lexemId;
    unsigned long long int tokenType;
    unsigned long long int ifvalue;
    unsigned long long int row;
    unsigned long long int col;

    LexemInfo();
    LexemInfo(const char* lexemStr, unsigned long long int lexemId, unsigned long long
int tokenType, unsigned long long int ifvalue, unsigned long long int row, unsigned long
long int col);
    LexemInfo(const NonContainedLexemInfo& nonContainedLexemInfo);
};
```

Опис структури LexemInfo

LexemInfo — це структура, яка слугує для зберігання даних про окрему лексему, визначену в процесі лексичного аналізу. Вона забезпечує публічний доступ до своїх полів і створена для зручного управління атрибутами лексеми. Далі наведено детальний опис її елементів та функцій:

Члени структури:

1. **char lexemStr[MAX_LEXEM_SIZE]**

Массив символів, що містить саму лексему у вигляді рядка.

MAX_LEXEM_SIZE — це максимальний розмір лексеми, зазвичай визначений як константа.

2. **unsigned long long int lexemId**

Унікальний ідентифікатор лексеми. Він дозволяє відрізнити лексеми між собою.

3. **unsigned long long int tokenType**

Тип токена, який відповідає лексемі. Наприклад, це може бути константа, оператор, ключове слово тощо.

4. **unsigned long long int ifvalue**

Додаткове значення, яке використовується для обробки умовних виразів або контексту лексеми. Наприклад, це може бути значення для порівняння чи виконання умов.

5. **unsigned long long int row**

Номер рядка, де знаходиться лексема в коді. Це корисно для відлагодження або повідомлень про помилки.

6. **unsigned long long int col**

Номер колонки в рядку, де розташована лексема.

7. **// TODO: ...**

Коментар, який вказує, що до структури можуть бути додані нові члени або властивості для розширення її функціональності.

Конструктори:

1. Конструктор за замовчуванням: **LexemInfo()**

Ініціалізує структуру з початковими значеннями. Зазвичай це нульові або порожні значення для членів структури.

2. Параметризований конструктор: **LexemInfo(const char* lexemStr, unsigned long long int lexemId, unsigned long long int tokenType, unsigned long long int ifvalue, unsigned long long int row, unsigned long long int col)**

Ініціалізує структуру з заданими значеннями.

- **lexemStr**: рядок лексеми.
- **lexemId**: унікальний ідентифікатор.
- **tokenType**: тип токена.
- **ifvalue**: додаткове значення.
- **row**: номер рядка.
- **col**: номер колонки.

3. Конструктор копіювання: **LexemInfo(const NonContainedLexemInfo& nonContainedLexemInfo)**

Ініціалізує LexemInfo на основі іншої структури NonContainedLexemInfo. Це дозволяє створити об'єкт на основі схожої структури.

Призначення:

Ця структура є корисною для:

- Лексичного аналізу (збереження інформації про токени у процесі аналізу вхідного коду).
- Збереження позицій (рядок і колонка) для генерації повідомлень про помилки.
- Структурування даних про лексеми, необхідних для побудови синтаксичного дерева.
- Розширення можливостей за допомогою додавання нових полів, наприклад, для семантичного аналізу.

3.3. Розробка лексичного аналізатора.

Основною метою лексичного аналізу є розбиття вихідного тексту, який складається з послідовності символів, на окремі слова або лексеми. Це означає виділення слів із суцільної послідовності символів. Усі символи вхідного тексту поділяються на ті, що належать до лексем, та символи, які їх розділяють. Для цього використовуються стандартні методи роботи з рядками. Вхідна програма опрацьовується послідовно від початку до кінця. Базові елементи, або лексичні одиниці, відокремлюються пробілами, знаками операцій і спеціальними символами (новий рядок, табуляція). Таким чином, розпізнаються і виділяються ідентифікатори, літерали та термінальні символи (операції, ключові слова).

Під час виділення лексеми вона розпізнається і заноситься до таблиці лексем за допомогою унікального номера, який однозначно ідентифікує її серед усіх можливих лексем. Це дозволяє наступним фазам компіляції працювати з лексемами як із унікальними номерами, а не з послідовностями символів. Завдяки цьому значно спрощується робота синтаксичного аналізатора, адже перевірка відповідності лексеми синтаксичній конструкції стає зручнішою, а також забезпечується легкий перегляд програми вперед і назад від поточної позиції аналізу. У таблиці лексем також записуються дані про рядок, де знаходиться лексема, для полегшення діагностики помилок, а також інша додаткова інформація.

Лексичний аналіз пропускає коментарі, оскільки вони не впливають на виконання програми, синтаксичний аналіз чи генерацію коду.

Лексеми поділяються на типи або лексичні класи:

- **Ключові слова:** #PROGRAM, VARIABLE, START, STOP, SCAN, PRINT, INTEGER16_t, IF, ELSE, FOR, GOTO, DOWNT0, REPEAT, UNTIL, WHILE, END, CONTINUE, EXIT
- **Ідентифікатори**
- **Числові константи:** цілі числа без знаку
- **Оператор присвоєння:** <=
- **Знаки операцій:** ADD, SUB, MUL, DIV, MOD, ==, !=, !>, !<, NOT, AND, OR

- **Роздільники:** ;, ,.
- **Дужки:** (,).

3.3.1. Розробка алгоритму роботи лексичного аналізатора.

Даний лексичний аналізатор — це програмний модуль, який розбиває вхідний текст на лексеми (основні синтаксичні одиниці) і класифікує їх за певними типами. Його основна мета — підготовка тексту до подальшого синтаксичного або семантичного аналізу. У цьому коді реалізовано багато функцій, які забезпечують ідентифікацію ключових слів, значень, ідентифікаторів, а також обробку коментарів.

Ось як працює цей аналізатор:

1. Основні структури даних

LexemInfo

Містить інформацію про кожну лексему:

- **lexemStr** — текстовий рядок лексеми.
- **lexemId** — унікальний ідентифікатор лексеми.
- **tokenType** — тип токена (ключове слово, ідентифікатор, значення тощо).
- **ifvalue** — додаткова інформація для значень.
- **row i col** — позиція лексеми в тексті (номер рядка та стовпця).

NonContainedLexemInfo

Служить для тимчасового зберігання лексем, забезпечуючи використання буфера (tempStrFor_123).

2. Основні масиви

- **lexemesInfoTable** — таблиця, де зберігаються всі знайдені лексеми.
- **identifierIdsTable** — таблиця для збереження ідентифікаторів, яка запобігає дублюванню.

3. Алгоритм лексичного аналізу

3.1. Токенізація (tokenize)

Ця функція розбиває текст на токени відповідно до регулярного виразу:

- Регулярний вираз (TOKENS_RE) визначає, які символи формують токен (ідентифікатори, ключові слова, числа тощо).
- За допомогою ітератора (std::sregex_token_iterator) текст обробляється токен за токеном.

3.2. Ідентифікація токена (lexicalAnalyze)

Для кожного токена викликаються функції:

1. **tryToGetKeyWord** — перевіряє, чи є токен ключовим словом.
2. **tryToGetIdentifier** — перевіряє, чи є токен ідентифікатором.
3. **tryToGetUnsignedValue** — перевіряє, чи є токен числовим значенням.

Якщо жоден із цих тестів не вдається, токен помічається як "непередбачувана лексема" (UNEXPECTED_LEXEME_TYPE).

4. Обробка ключових слів, ідентифікаторів та значень

Ключові слова

Ключові слова перевіряються за допомогою регулярного виразу (KEYWORDS_RE) і отримують унікальний lexemId.

Ідентифікатори

- Перевіряються регулярним виразом (IDENTIFIERS_RE).
- Заноситься до таблиці identifierIdsTable.

Значення

- Перевіряються регулярним виразом (UNSIGNEDVALUES_RE).
- Зберігаються у поле ifvalue.

5. Обробка коментарів (commentRemover)

Функція видаляє коментарі з тексту. Вона підтримує:

- Однорядкові коментарі (наприклад, //).
- Багаторядкові коментарі (наприклад, /* ... */). Після видалення коментарі замінюються пробілами, зберігаючи структуру тексту.

6. Збереження позицій (setPositions)

Функція встановлює номер рядка та стовпця кожної лексеми у вхідному тексті. Це дозволяє вказувати точне місце розташування помилок у тексті.

7. Друк результатів (printLexemes)

Результати аналізу виводяться у вигляді таблиці, де показано:

- Індекс лексеми.
- Її текст.
- Ідентифікатор.
- Тип.
- Значення (для чисел).
- Рядок і стовпець у тексті.

Структура та поля результатів лексичного аналізатора

Результати роботи лексичного аналізатора подаються у вигляді таблиці. Кожен рядок цієї таблиці представляє одну лексему та містить наступну інформацію:

Поля таблиці:

1. Індекс лексеми (index)

Це порядковий номер лексеми у загальному списку. Використовується для нумерації та швидкого доступу до конкретної лексеми.

2. Текст лексеми (lexemStr)

Текстовий вигляд лексеми, зчитаний з вихідного тексту програми. Наприклад, це може бути слово, число, символ або оператор.

3. Ідентифікатор лексеми (lexemId)

Унікальний ідентифікатор, який присвоюється кожній лексемі залежно від її типу.

Наприклад:

- Ідентифікатори для ключових слів.
- Ідентифікатори для змінних.
- Унікальні номери для інших лексем.

4. Тип лексеми (tokenType)

Визначає тип лексеми, наприклад:

- **Ключове слово (keyword).**
- **Ідентифікатор (identifier).**
- **Числове значення (value).**
- **Неочікувана лексема (unexpected lexeme).**

5. Значення (ifvalue)

Актуальне значення для числових лексем. Наприклад, якщо лексема — це число 123, то його значення буде 123. Для інших типів лексем це поле може бути неактивним.

6. Рядок (row)

Номер рядка у вихідному тексті, де знаходиться лексема. Це полегшує ідентифікацію її місця у програмному коді.

7. Стовпець (col)

Номер символу у рядку, з якого починається лексема. Це додатково уточнює її позицію у вихідному коді.

Стани під час аналізу

Лексичний аналізатор проходить кілька основних станів:

1. Ініціалізація

Підготовка таблиць і структур, зокрема:

- Таблиці лексем (lexemesInfoTable).
- Таблиці ідентифікаторів (identifierIdsTable).

2. Обробка тексту

- Видалення коментарів.
- Розбиття тексту на токени.

3. Класифікація лексем

Для кожної лексеми визначають:

- Чи є вона ключовим словом.
- Чи є вона ідентифікатором.
- Чи є вона числовим значенням.
- Чи є вона несподіваною або помилковою.

4. Формування таблиці результатів

Для кожної лексеми записується відповідна інформація: індекс, текст, ідентифікатор, тип, значення, позиція в тексті.

5. Виведення результатів

Таблиця лексем друкується у форматі зручному для перегляду, де відображаються всі згадані поля.

Табл 1.1

Індекс	Текст лексеми	Ідентифікатор	Тип	Значення	Рядок	Стовпець
0	#PROGRAM	283	Ключове слово	0	1	1
1	pROGRAMA	0	Ідентифікатор	0	1	10
2	;	256	Ключове слово	0	1	18
3	START	292	Ключове слово	0	2	1
4	VARIABLE	256	Ключове слово	0	2	9
5	;	301	Ключове слово	0	2	11
6	STOP	307	Ключове слово	0	2	17

Преваги такої структури:

- **Простота аналізу:** Користувач легко знаходить помилки або несподівані лексеми завдяки вказаним рядкам і стовпцям.
- **Гнучкість:** Додавання нових типів лексем або розширення можливостей аналізатора не потребує значних змін.
- **Уніфікованість:** Усі дані про лексеми представлені в одній структурованій формі.

8. Особливості

1. Буферизація:

- Для тимчасового збереження рядків використовується буфер tempStrFor_123, що дозволяє ефективно управляти пам'яттю.

2. Гнучкість:

- Регулярні вирази (TOKENS_RE, IDENTIFIERS_RE, KEYWORDS_RE, UNSIGNEDVALUES_RE) можна налаштовувати під конкретні вимоги.

3. Обробка помилок:

- Якщо лексема не відповідає жодному з шаблонів, вона позначається як помилкова

3.3.2. Опис програми реалізації лексичного аналізатора.

Головною метою лексичного аналізатора є поділ вхідного тексту програми, який складається з послідовності символів, на окремі лексеми — слова, що мають значення для подальшого аналізу. Усі символи вхідного тексту класифікуються як такі, що належать до лексем, або як роздільники. У процесі аналізу використовуються стандартні алгоритми роботи з рядками. Вхідний текст програми обробляється послідовно від початку до кінця, а базові елементи (лексичні одиниці) виділяються за допомогою пробілів, знаків операцій та спеціальних символів, таких як новий рядок або табуляція. У результаті розпізнаються ідентифікатори, літерали та термінальні символи (зокрема, операції та ключові слова).

Аналізатор обробляє файл до досягнення його кінця. Для цього викликається функція `tokenize()`, яка читає вміст файлу, виділяє лексеми та порівнює їх із зарезервованими словами. У випадку збігу лексемі присвоюється відповідний тип або значення (якщо це числова константа).

Кожна виділена лексема додається до списку `m_tokens`, використовуючи унікальний тип лексеми. Це дозволяє наступним етапам компіляції працювати з лексемами як із конкретними типами, а не як із послідовностями символів, що значно спрощує синтаксичний аналіз. Наприклад, перевірка належності лексеми до певної синтаксичної конструкції або навігація текстом програми (вперед і назад) стають простішими. У таблиці лексем також зберігається інформація про рядок і колонку кожної лексеми, що полегшує діагностику помилок. Додатково зберігається метайнформація, яка може бути корисною на подальших етапах аналізу.

Під час лексичного аналізу виявляються й відзначаються помилки, пов'язані з некоректними символами чи невірними ідентифікаторами. Такі помилки

ігноруються, оскільки вони не впливають на синтаксичний аналіз або генерацію коду.

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

3.4. Розробка синтаксичного та семантичного аналізатора.

Синтаксичний аналіз – це процес, що визначає, чи належить деяка послідовність лексем граматиці мови програмування. В принципі, для будь-якої граматики можна побудувати синтаксичний аналізатор, але граматики, які використовуються на практиці, мають спеціальну форму. Наприклад, відомо, що для будь-якої контекстно-вільної граматики може бути побудований аналізатор, складність якого не перевищує $O(n^3)$ для вхідного рядка довжиною n .

Код реалізує лексичний і синтаксичний аналізатор із побудовою абстрактного синтаксичного дерева (AST) на основі методу Кока-Янгера-Касамі (СҮК) та рекурсивного спуску. Розглянемо основні етапи роботи:

1. Лексичний аналіз

Лексичний аналізатор розбиває вхідний текст на лексеми (мінімальні значущі одиниці мови, такі як ідентифікатори, ключові слова, константи тощо) та зберігає їх у таблиці LexemInfo.

2. Метод СҮК для синтаксичного аналізу

- **Ініціалізація:** створюється таблиця parseInfoTable, де кожна комірка містить множину символів граматики.
- **Заповнення таблиці:** використовується двовимірний підхід, де кожна комірка заповнюється на основі правил граматики:
- Якщо правило має один елемент справа, перевіряється відповідність лексеми цьому правилу.

- Якщо правило має два елементи, шукається розбиття, яке дозволяє побудувати комбінацію двох піддерев.
- Після завершення побудови таблиці перевіряється наявність стартового символу граматики у верхньому правому куті таблиці. Якщо символ є, аналіз вважається успішним.

3. Рекурсивний спуск

Якщо метод СΥК не успішний або обрано режим рекурсивного спуску, запускається рекурсивний аналізатор:

- Кожне правило граматики перевіряється на відповідність лексемам у поточній позиції.
- Якщо знайдено відповідність, індекс лексем збільшується, і аналіз продовжується для наступних правил.
- У разі помилки повертається інформація про невідповідність лексеми.

4. Побудова абстрактного синтаксичного дерева (AST)

- Дерево будується функцією buildAST. Кожен вузол представляє або термінальний, або нетермінальний символ.
- Для кожного правила створюються дочірні вузли, які відповідають його елементам.
- Якщо правило має два елементи справа, дерево будується рекурсивно для обох піддерев.

5. Виведення AST

Для візуалізації AST використовуються функції:

- printAST: виводить дерево в консоль у вигляді ієрархічної структури.
- printASTToFile: записує дерево у файл.

6. Збереження таблиці СΥК

Таблиця результатів СΥК може бути виведена або збережена у файл за допомогою функцій displayParseInfoTable та saveParseInfoTableToFile.

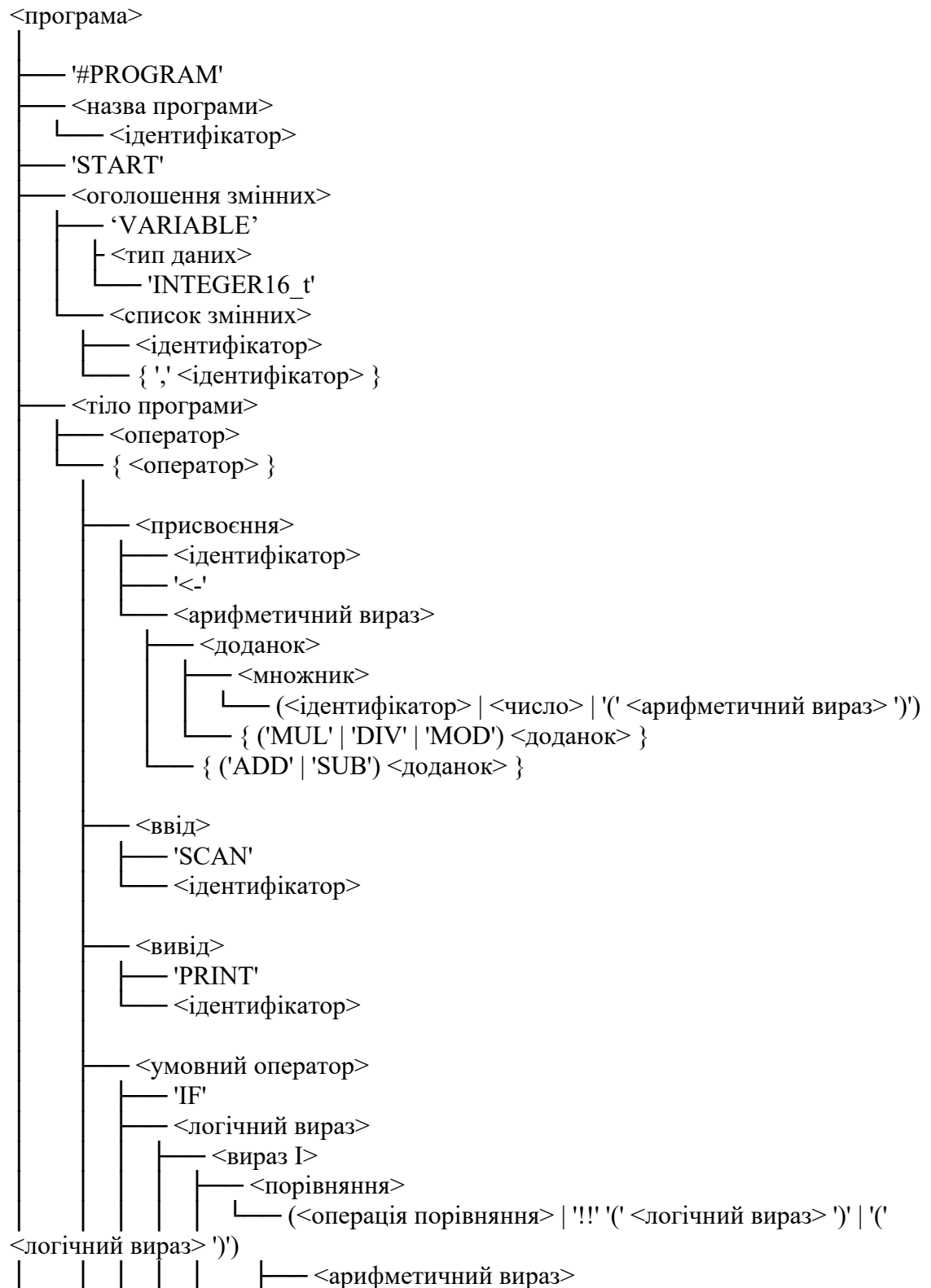
7. Основна функція синтаксичного аналізу

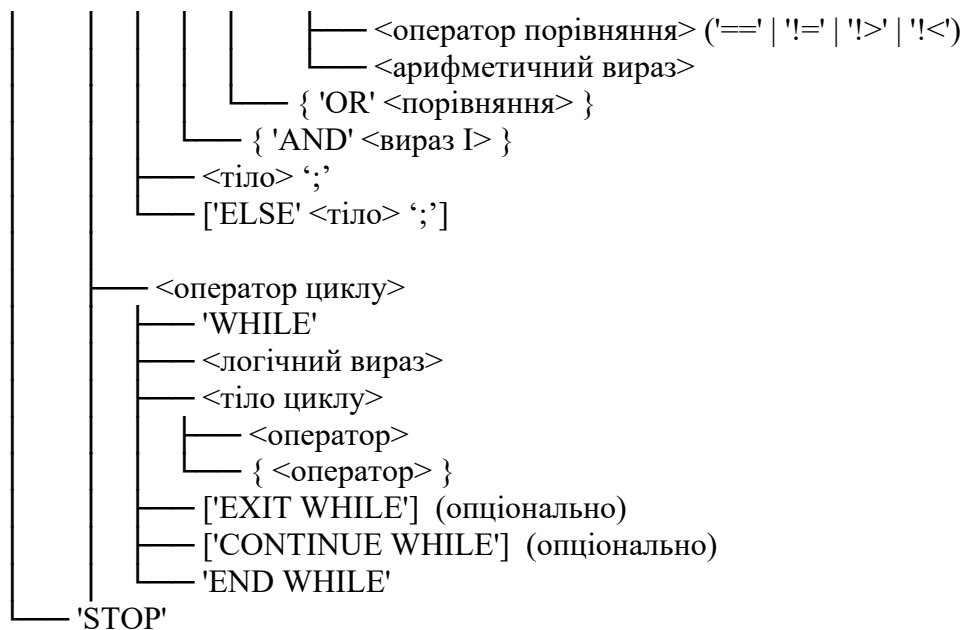
Функція syntaxAnalyze координує процес:

- Спочатку викликається метод СΥК.

- Якщо СУК не успішний, виконується рекурсивний спуск.
- У разі помилки виводиться інформація про невідповідність та позицію помилки у вхідному коді.

3.4.1. Розробка дерева граматичного розбору.





3.4.2. Розробка алгоритму роботи синтаксичного і семантичного аналізатора.

Код реалізує лексичний і синтаксичний аналізатор із побудовою абстрактного синтаксичного дерева (AST) на основі методу Кока-Янгера-Касамі (CYK) та рекурсивного спуску. Розглянемо основні етапи роботи:

1. Лексичний аналіз

2. Метод CYK для синтаксичного аналізу

- Ініціалізація
- Заповнення таблиці
- Перевірка успішності

3. Рекурсивний спуск

4. Побудова абстрактного синтаксичного дерева (AST)

5. Виведення AST

6. Збереження таблиці CYK

Семантичний аналізатор виконує перевірку правильності структур та логіки програми на основі аналізу лексем та граматики. У цьому коді реалізовано кілька функцій, які відповідають за різні аспекти семантичного аналізу.

- Перевіряє декларації та їх колізії.
- Аналізує ініціалізацію змінних.
- Виявляє невірне використання ключових слів.

3.4.3. Опис програми реалізації синтаксичного та семантичного аналізатора.

Код реалізує лексичний і синтаксичний аналізатор із побудовою абстрактного синтаксичного дерева (AST) на основі методу Кока-Янгера-Касамі (СҮК) та рекурсивного спуску. Розглянемо основні етапи роботи:

1. Лексичний аналіз

Лексичний аналізатор розбиває вхідний текст на лексеми (мінімальні значущі одиниці мови, такі як ідентифікатори, ключові слова, константи тощо) та зберігає їх у таблиці LexemInfo.

2. Метод СҮК для синтаксичного аналізу

- **Ініціалізація:** створюється таблиця parseInfoTable, де кожна комірка містить множину символів граматики.
- **Заповнення таблиці:** використовується двовимірний підхід, де кожна комірка заповнюється на основі правил граматики:
- Якщо правило має один елемент справа, перевіряється відповідність лексеми цьому правилу.
- Якщо правило має два елементи, шукається розбиття, яке дозволяє побудувати комбінацію двох піддерев.
- Після завершення побудови таблиці перевіряється наявність стартового символу граматики у верхньому правому куті таблиці. Якщо символ є, аналіз вважається успішним.

3. Рекурсивний спуск

Якщо метод СҮК не успішний або обрано режим рекурсивного спуску, запускається рекурсивний аналізатор:

- Кожне правило граматики перевіряється на відповідність лексемам у поточній позиції.

- Якщо знайдено відповідність, індекс лексем збільшується, і аналіз продовжується для наступних правил.
- У разі помилки повертається інформація про невідповідність лексеми.

4. Побудова абстрактного синтаксичного дерева (AST)

- Дерево будується функцією `buildAST`. Кожен вузол представляє або термінальний, або нетермінальний символ.
- Для кожного правила створюються дочірні вузли, які відповідають його елементам.
- Якщо правило має два елементи справа, дерево будується рекурсивно для обох піддерев.

5. Виведення AST

Для візуалізації AST використовуються функції:

- `printAST`: виводить дерево в консоль у вигляді ієрархічної структури.
- `printASTToFile`: записує дерево у файл.

6. Збереження таблиці СҮК

Таблиця результатів СҮК може бути виведена або збережена у файл за допомогою функцій `displayParseInfoTable` та `saveParseInfoTableToFile`.

7. Основна функція синтаксичного аналізу

Функція `syntaxAnalyze` координує процес:

- Спочатку викликається метод СҮК.
- Якщо СҮК не успішний, виконується рекурсивний спуск.
- У разі помилки виводиться інформація про невідповідність та позицію помилки у вхідному коді.

Семантичний аналізатор виконує перевірку правильності структур та логіки програми на основі аналізу лексем та граматики. У цьому коді реалізовано кілька функцій, які відповідають за різні аспекти семантичного аналізу.

Основні функції семантичного аналізатора

1. `getLastDataSectionLexemIndex`

Ця функція знаходить індекс останньої лексеми у секції даних.

- Використовує функцію парсера `recursiveDescentParserRuleWithDebug`, щоб пройти по граматиці секції даних ("program____part1").
- Якщо лексема знайдена, повертається її індекс; якщо ні – повертається помилка (~0).

2. **checkingInternalCollisionInDeclarations**

Перевіряє внутрішні колізії у деклараціях змінних і міток:

- **Колізії identifier/identifier:** Виявляється, якщо ідентифікатор задекларовано кілька разів у тій самій області.
- **Колізії label/label:** Виявляється при дублюванні міток.
- **Колізії identifier/label:** Виявляється, якщо ідентифікатор використовується і як змінна, і як мітка.
- Якщо ідентифікатор або мітка не були задекларовані, виводиться помилка.

3. **checkingVariableInitialization**

Перевіряє, чи ініціалізовано всі змінні перед використанням:

- Аналізує ділянку коду після секції даних.
- Визначає, чи були змінні ініціалізовані (перевіряє наявність операцій присвоєння, введення чи виклику функцій, що ініціалізують значення).

4. **checkingCollisionInDeclarationsByKeyWords**

Перевіряє, чи збігаються імена декларацій з ключовими словами:

- Використовує регулярний вираз для виявлення збігів.
- Якщо ідентифікатор відповідає ключовому слову, генерується помилка (COLLISION_IK_STATE).

5. **semantixAnalyze**

Головна функція, що викликає всі попередні модулі аналізу:

- Перевіряє колізії в деклараціях.
- Аналізує ініціалізацію змінних.
- Перевіряє збіг імен з ключовими словами.

- Якщо хоча б одна перевірка не проходить, повертається відповідний код помилки.

Ключові аспекти реалізації

1. Лексеми та граматика:

- Семантичний аналізатор працює з таблицею лексем (lexemInfoTable) та граматикою (Grammar), які є результатами попередніх етапів аналізу (лексичного та синтаксичного).
- Типи лексем визначаються полем tokenType.

2. Перевірка колізій:

Семантичний аналізатор знаходить конфлікти в ідентифікаторах, щоб уникнути неоднозначності або помилок у виконанні програми.

3. Робота з регулярними виразами:

Для перевірки ідентифікаторів на збіг із зарезервованими словами використовуються регулярні вирази (std::regex).

4. Повідомлення про помилки:

Усі помилки виводяться у консоль із деталізацією, наприклад:

5. Collision(identifier/identifier): myVariable

6. Uninitialized: myVariable

7. Коди стану:

Кожна функція повертає код стану (наприклад, SUCCESS_STATE, COLLISION__STATE), що дозволяє головній функції визначити, чи є помилки.

Типовий процес роботи

1. Виклик функції semantixAnalyze, яка:

- Перевіряє декларації та їх колізії.
- Аналізує ініціалізацію змінних.
- Виявляє невірне використання ключових слів.
- У разі помилки повертається відповідний код, і програма виводить інформацію про проблему

3.5. Розробка генератора коду.

Генерація вихідного коду передбачає спочатку перетворення програми у якесь проміжне представлення, а тоді вже генерацію з проміжного представлення у вихідний код. У якості проміжного представлення виберемо абстрактне синтаксичне дерево.

Абстрактне синтаксичне дерево (AST) — це структура даних, яка представляє синтаксичну структуру вихідного коду програми у вигляді дерева. AST використовується в компіляторах, інтерпретаторах та інструментах статичного аналізу для обробки коду.

AST представляє тільки важливу для аналізу і виконання інформацію, ігноруючи зайві деталі (наприклад, круглі дужки чи крапки з комою). Це спрощений, але точний опис логіки програми.

Вузли дерева представляють конструкції мови програмування (оператори, вирази, змінні, функції тощо). Гілки відповідають підконструкціям або елементам цих конструкцій.

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

AST є спрощеною версією синтаксичного дерева. Воно не включає зайві вузли, що відповідають елементам, які не впливають на логіку програми (наприклад, дужки чи крапки з комою).

3.5.1. Розробка алгоритму роботи генератора коду.

Генерація вихідного коду передбачає спочатку перетворення програми у якесь проміжне представлення, а тоді вже генерацію з проміжного представлення у вихідний код. У якості проміжного представлення виберемо абстрактне синтаксичне дерево.

Абстрактне синтаксичне дерево (AST) — це структура даних, яка представляє синтаксичну структуру вихідного коду програми у вигляді дерева. AST використовується в компіляторах, інтерпретаторах та інструментах статичного аналізу для обробки коду.

AST представляє тільки важливу для аналізу і виконання інформацію, ігноруючи зайві деталі (наприклад, круглі дужки чи крапки з комою). Це спрощений, але точний опис логіки програми.

Вузли дерева представляють конструкції мови програмування (оператори, вирази, змінні, функції тощо). Гілки відповідають підконструкціям або елементам цих конструкцій.

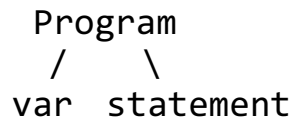
Кожен вузол відповідає певному типу конструкції коду (наприклад, оператору додавання, виклику функції, оголошенню змінної).

AST є спрощеною версією синтаксичного дерева. Воно не включає зайві вузли, що відповідають елементам, які не впливають на логіку програми (наприклад, дужки чи крапки з комою).

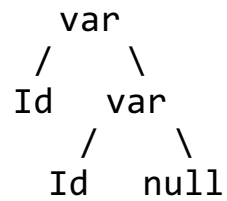
3.5.1. Розробка алгоритму роботи генератора коду.

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

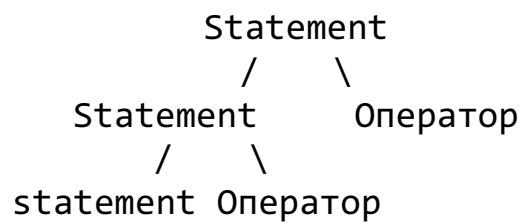
Програма має вигляд:



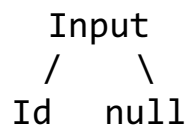
Оголошення змінних:



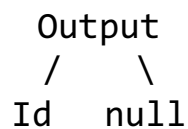
Тіло програми:



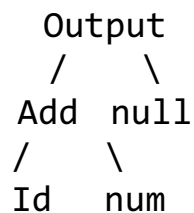
Оператор вводу:



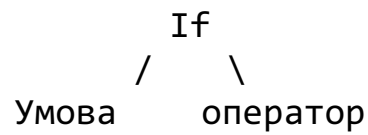
Оператор виводу:



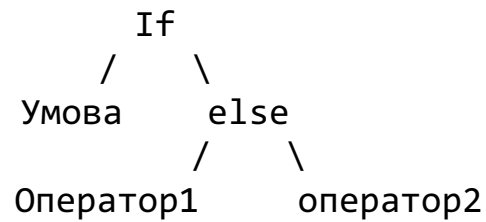
Також оператор виводу може мати за лівого нащадка різні арифметичні вирази, наприклад:



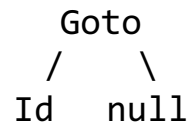
Умовний оператор (IF() оператор;):



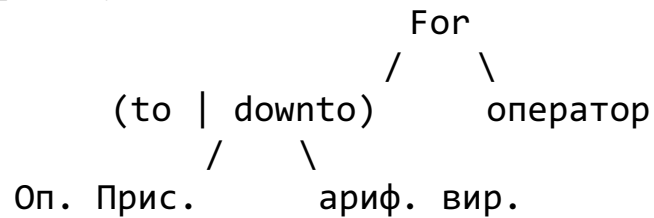
Умовний оператор (IF() оператор1; else оператор2;):



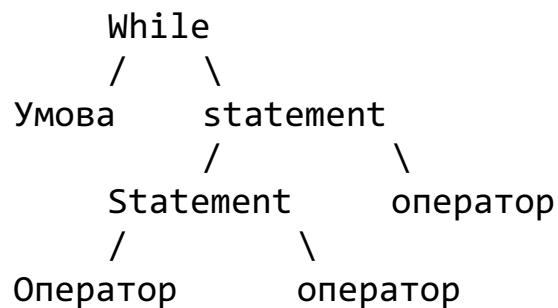
Оператор безумовного переходу:



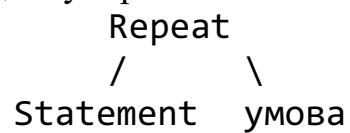
Оператор циклу for:



Оператор циклу while:



Оператор циклу repeat:



/ \
Оператор оператор

Оператор присвоєння:

==>
/ \
Id арифметичний вираз

Арифметичний вираз:

(+ або -)
/ \
Id id

Доданок:

(*, DIV або MOD)
/ \
МНОЖНИК МНОЖНИК

Множник:

фактор
/ \
id або number або (арифм. вираз) null

Складений оператор:

compount
/ \
statement null

Генератор коду буде обходити створене дерево і, маючи усію необхідну інформацію, генерувати вихідний файл типу Portable Exetubale. Опрацювання кожного з вузлів дерева передбачає рекурсивний виклик функції генерування коду для лівого і правого нащадків.

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

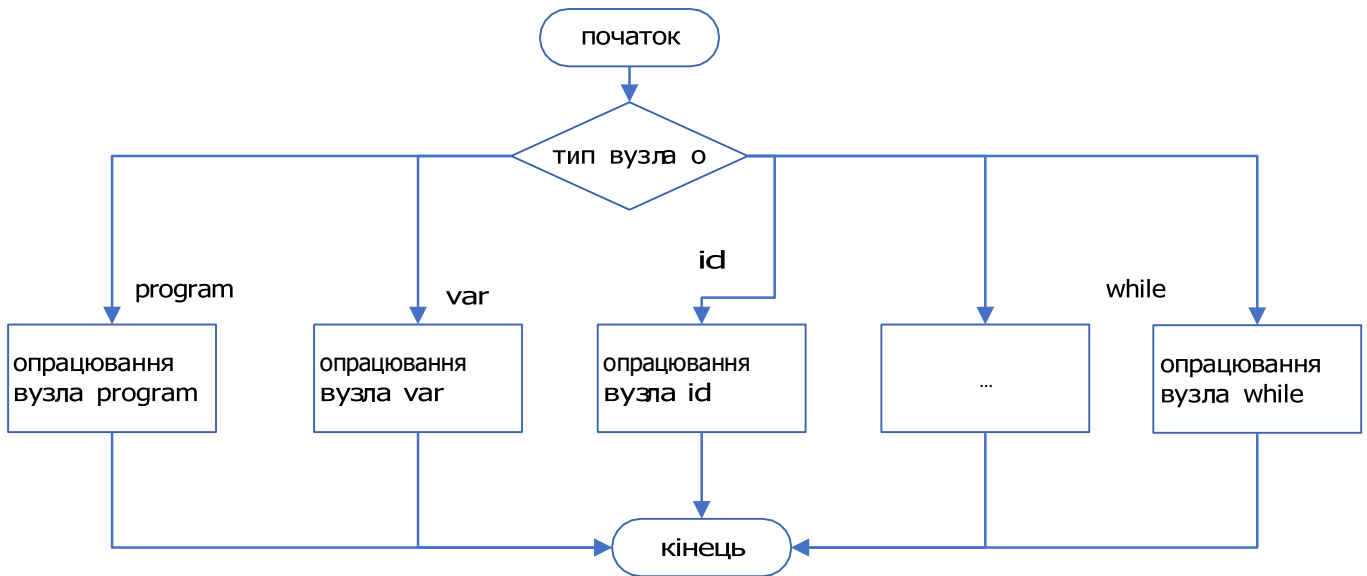


Рис. 3.6. Блок-схема алгоритму роботи генератора коду.

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

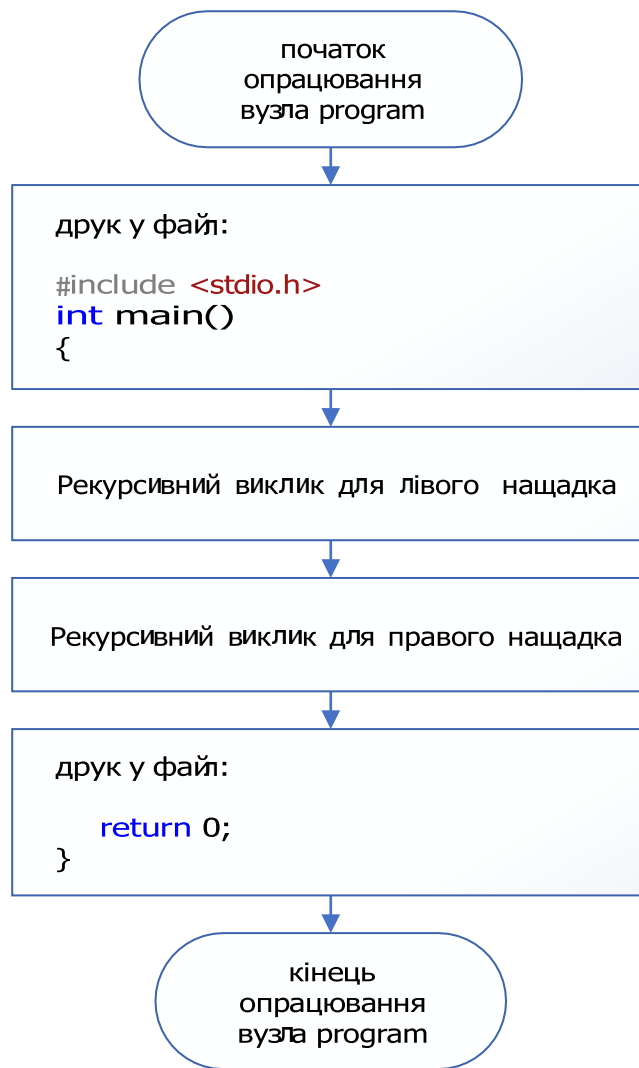


Рис. 3.7. Блок-схема алгоритму опрацювання вузла program.

3.5.2. Опис програми реалізації генератора коду.

Основні функції і макроси забезпечують різні етапи генерації коду: створення секцій даних, секцій коду, ініціалізації змінних і структурування команд. Давайте розглянемо основні компоненти і їх призначення:

1. Макроси та константи

- **MAX_TEXT_SIZE, MAX_GENERATED_TEXT_SIZE:** Визначають максимальний розмір тексту та згенерованого коду.
- **SUCCESS_STATE:** Статус для успішного виконання.
- **MAX_OUTTEXT_SIZE:** Буфер для вихідного тексту.
- **MAX_LEXEM_SIZE:** Максимальний розмір однієї лексеми.
- **MAX_WORD_COUNT:** Максимальна кількість слів/лексем, які обробляються.

2. Структури даних

- **LabelOffsetInfo:**
 - Зберігає інформацію про мітки (label) та їх позиції в коді.
 - Використовується для управління стрибками (goto) в асемблерному коді.
- **GotoPositionInfo:**
 - Інформація про позиції інструкцій стрибків, які мають бути пов'язані з відповідними мітками.
- **tokenStruct:**
 - Таблиця, що описує багатокomпонентні токени, такі як IF ... THEN, FOR ... TO ..., WHILE, тощо.

3. Генерація коду

- **makeCode:**

- Основна функція для генерації коду. Вона викликає кілька інших функцій для побудови різних секцій:
 - **makeTitle:** Генерує заголовок (наприклад, визначення моделі процесора та архітектури).
 - **makeDependenciesDeclaration:** Додає оголошення необхідних функцій і констант.
 - **makeDataSection:** Створює секцію даних.
 - **makeBeginProgramCode:** Починає секцію коду.
 - **makeInitCode, initMake:** Виконує ініціалізацію змінних.
 - **makeSaveHWStack, makeResetHWStack:** Зберігає та відновлює стек на апаратному рівні.
 - **makeEndProgramCode:** Додає фінальні інструкції (наприклад, ret для завершення програми).

4. Маніпуляція з токенами

- **detectMultiToken:**

- Перевіряє, чи відповідає поточна лексема багатоконпонентному токenu з таблиці tokenStruct.

- **createMultiToken:**

- Створює багатоконпонентний токен і зберігає його у структурі LexemInfo.

5. Генерація машинного коду

- **outBytes2Code:**

- Копіює байти з одного буфера до іншого, формуючи машинний код.

- **Пример генерації команд:**

- **makeSaveHWStack:**
 - Генерує інструкцію `mov ebp, esp` для збереження стека.
- **makeResetHWStack:**
 - Генерує інструкцію `mov esp, ebp` для відновлення стека.

Як працює генерація коду в функції **makeCode**

Функція `makeCode` поступово трансформує лексеми з таблиці `LexemInfo` у машинний код або інший низькорівневий формат. У цьому поясненні з кодовими вставками розглянемо, як саме це реалізовано.

1. Ініціалізація

На початку функція викликає кілька підфункцій для створення основних секцій коду:

```
currBytePtr = makeTitle(lastLexemInfoInTable, currBytePtr);
currBytePtr = makeDependenciesDeclaration(lastLexemInfoInTable, currBytePtr);
currBytePtr = makeDataSection(lastLexemInfoInTable, currBytePtr);
currBytePtr = makeBeginProgramCode(lastLexemInfoInTable, currBytePtr);
```

- **makeTitle:** Генерує заголовок програми
- **makeDependenciesDeclaration:** Додає секцію залежностей (наприклад, бібліотеки або модулі).
- **makeDataSection:** Додає секцію даних (глобальні змінні, константи тощо).
- **makeBeginProgramCode:** Додає інструкції для ініціалізації, наприклад, налаштування стеку чи регістрів.

2. Ініціалізація стеку

Перед початком основної генерації коду функція скидає тимчасовий стек і генерує інструкції для ініціалізації:

```
lexemInfoTransformationTempStackSize = 0;
```

```
currBytePtr = makeInitCode(lastLexemInfoInTable, currBytePtr);  
currBytePtr = initMake(lastLexemInfoInTable, currBytePtr);  
currBytePtr = makeSaveHWStack(lastLexemInfoInTable, currBytePtr);
```

- **makeInitCode:** Генерує код для ініціалізації змінних.

3. Обробка лексем у циклі

Основна логіка генерації знаходиться в циклі `for`, де кожна лексема обробляється залежно від її типу:

```
for (struct LexemInfo* lastLexemInfoInTable_ ;  
     lastLexemInfoInTable_ = *lastLexemInfoInTable,  
     (*lastLexemInfoInTable)->lexemStr[0] != '\0'; ) {
```

Цей цикл ітерує через таблицю лексем, поки не зустрінє лексему з порожнім рядком (`lexemStr[0] == '\0'`).

4. Генерація коду для конструкцій

В залежності від лексеми, викликаються функції-генератори. Наприклад:

Умовні оператори:

```
IF_THEN_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr,  
generatorMode, NULL);  
ELSE_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr,  
generatorMode, NULL);
```

- **IF_THEN_CODER:** Додає інструкції для умовного оператора `if`.
- **ELSE_CODER:** Генерує код для гілки `else`.

Цикли:

```
FOR_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr,  
generatorMode, NULL);  
WHILE_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr,  
generatorMode, NULL);
```

REPEAT_UNTIL_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

- **FOR_CODER:** Генерує код для циклу for.
- **WHILE_CODER:** Генерує інструкції для циклу while.
- **REPEAT_UNTIL_CODER:** Обробляє конструкцію циклу repeat until.

Операції та оператори:

ADD_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

SUB_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

MUL_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

DIV_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

MOD_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

- Генерація арифметичних операцій (ADD, SUB, MUL, DIV, MOD).

Логічні оператори:

AND_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

OR_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

NOT_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

- Логічні оператори NOT, AND, OR.

Інші оператори:

INPUT_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

OUTPUT_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

- **INPUT_CODER:** Обробляє введення.
- **OUTPUT_CODER:** Обробляє виведення.

5. Обробка помилок

Якщо лексема не була оброблена жодною з функцій-генераторів, генерується помилка:

```
if (lastLexemInfoInTable_ == *lastLexemInfoInTable) {
    printf("\r\nError in the code generator! \"%s\" - unexpected token!\r\n",
(*lastLexemInfoInTable)->lexemStr);
    exit(0);
}
```

Це простий механізм обробки помилок, який завершує програму з повідомленням про неочікувану лексему.

6. Завершення програми

Після обробки всіх лексем функція генерує завершальні інструкції:

```
currBytePtr = makeResetHWStack(lastLexemInfoInTable, currBytePtr);
currBytePtr = makeEndProgramCode(lastLexemInfoInTable, currBytePtr);
```

- **makeResetHWStack:** Відновлює стан стеку.
- **makeEndProgramCode:** Додає фінальні інструкції, наприклад, завершення виконання.

7. Виведення коду

Функція viewCode виводить згенерований код форматі (шістнадцяткові):

```
void viewCode(unsigned char* outCodePtr, size_t outCodePrintSize, unsigned
char align) {
    printf("\r\n;      +0x0 +0x1 +0x2 +0x3 +0x4 +0x5 +0x6 +0x7 +0x8 +0x9
+0xA +0xB +0xC +0xD +0xE +0xF ");
    printf("\r\n;0x00000000: ");
    // Вивід кожного байта
```

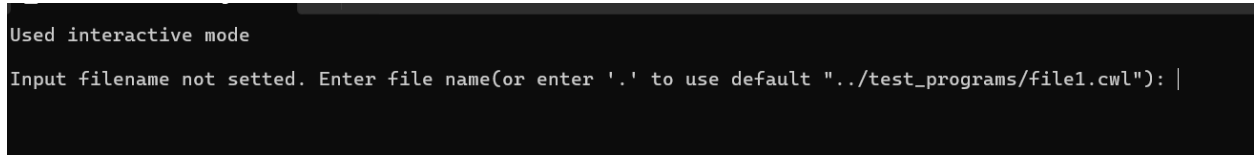
4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА

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

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

4.1. Опис інтерфейсу та інструкції користувачу.

Розроблений транслятор має простий консольний інтерактивний інтерфейс. Одразу після запуску пропонується ввести шлях до файлу, який потрібно обробити.



```
Used interactive mode
Input filename not setted. Enter file name(or enter '.' to use default "../test_programs/file1.cwl"): |
```

Рис. 4.1.Результати роботи розробленого транслятора.

4.2. Виявлення лексичних і синтаксичних помилок.

Помилки у вхідній програмі виявляються на стадіях лексичного, синтаксичного та семантичного аналізів.

```
Source after comment removing:
-----
#PROGRAM pPROGRAMA;
VARIABLE INTEGER16_t AAAVALUE, BBBVALUE, cCCVALUE;
START
  SCAN(aAAVALUE)
  SCAN(bBBVALUE)
  SCAN(cCCVALUE)
  IF (aAAVALUE == BBBVALUE AND aAAVALUE == cCCVALUE)
    PRINT(1);
  ELSE
    PRINT(0);
  SCAN(aAAVALUE)
STOP
-----

Lexical analysis detected unexpected lexeme
Bad lexeme:
-----
index      lexeme      id      type      ifvalue row    col
-----
0          AAAVALUE      0       127       0      2      26
-----

Press Enter to exit . . .
C:\Users\Admin\Desktop\CourseWork - кон?я\CourseWork - кон?я\x64\Debug\cw_sp2__2024_2025.exe (process 18224) exited with
code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Рис. 4.2.Помилка при лексичному аналізі.

```
Microsoft Visual Studio Debu  X  +  v
No command line arguments are entered, so you are working in step-by-step interactive mode.
ATTENTION: The next step is critical, if it is skipped the compilation process will be terminated!
Enter 'y' to syntax analyze action(to pass action process enter 'n' or others key): y
ATTENTION: for better performance, use Release mode!
cykParse complete.....[   ok   ]: 48      STOP
cykAlgorithmImplementation return "false".
File "../test_programs/file1.ast" saved.
Parse failed.
(The predicted terminal does not match the expected one.
Possible unexpected terminal "START" on line 3 at position 1
..., but this is not certain.)
File "../test_programs/file1_syntax_error.txt" saved.
File "../test_programs/file1.ast" saved.

C:\Users\Admin\Desktop\CourseWork - кон?я\CourseWork - кон?я\x64\Debug\cw_sp2__2024_2025.exe (process 12740) exited with
code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .|
```

Рис. 4.3.Помилка при синтаксичному аналізі.

```
Microsoft Visual Studio Debu  X  +  v
No command line arguments are entered, so you are working in step-by-step interactive mode.
ATTENTION: The next step is critical, if it is skipped the compilation process will be terminated!
Enter 'y' to semantix analyze action(to pass action process enter 'n' or others key): y
Undeclared identifier: aAAVALUE
File "../test_programs/file1_semantix_error.txt" saved.

C:\Users\Admin\Desktop\CourseWork - кон?я\CourseWork - кон?я\x64\Debug\cw_sp2__2024_2025.exe (process 11992) exited with
code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .|
```

Рис. 4.4.Помилка при семантичному аналізі.

4.3. Перевірка роботи транслятора за допомогою тестових задач.

Тестова програма «Лінійний алгоритм»

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

2. Обрахувати значення виразу

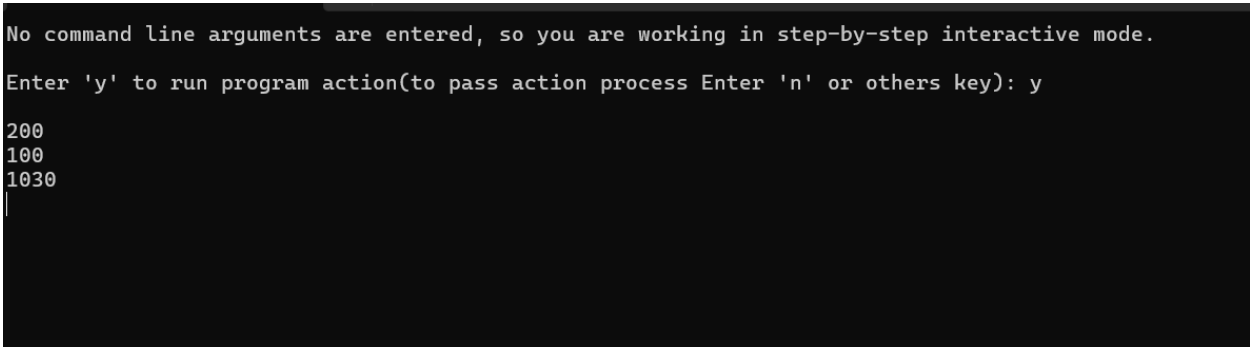
$$X = (A - B) * 10 + (A + B) / 10$$

3. Вивести значення X на екран.

Напишемо програму на вхідній мові програмування:

```
#PROGRAM pPROGRAMA;
VARIABLE INTEGER16_t aAAVALUE, bBBVALUE, xXXVALUE;
START
    SCAN(aAAVALUE)
    SCAN(bBBVALUE)
    xXXVALUE <- 10 * (aAAVALUE - bBBVALUE) + (aAAVALUE + bBBVALUE) / 10
    PRINT(xXXVALUE)
    SCAN(aAAVALUE)
STOP
```

Після запуску програми отримуємо такі результати:



```
No command line arguments are entered, so you are working in step-by-step interactive mode.
Enter 'y' to run program action(to pass action process Enter 'n' or others key): y
200
100
1030
|
```

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 інакше вивести 0.

Напишемо програму на вхідній мові програмування:

```
#PROGRAM pPROGRAMA;  
  VARIABLE INTEGER16_t aAAVALUE, bBBVALUE, cCCVALUE;  
START  
  SCAN(aAAVALUE)  
  SCAN(bBBVALUE)  
  SCAN(cCCVALUE)  
  IF (aAAVALUE == bBBVALUE AND aAAVALUE == cCCVALUE)  
    PRINT(1);  
  ELSE  
    PRINT(0);  
  SCAN(aAAVALUE)  
STOP
```

No command line arguments are entered, so you are working in step-by-step interactive mode.

Enter 'y' to run program action(to pass action process Enter 'n' or others key): y

1
1
1
1
1
|

Рис. 4.6. Результати виконання тестової задачі 1.

ВИСНОВКИ

В процесі виконання курсового проекту було виконано наступне:

1.Складено формальний опис мови програмування z10, в термінах розширеної нотації Бекуса-Наура, виділено усі термінальні символи та ключові слова.

2.Створено, а саме:

2.1.Розроблено прямий лексичний аналізатор, орієнтований на розпізнавання лексем, що є заявлені в формальному описі мови програмування.

2.2.Розроблено синтаксичний аналізатор на основі низхідного методу.

Складено деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура

2.3.Розроблено генератор коду, відповідні процедури якого викликаються після перевірки синтаксичним аналізатором коректності запису чергового оператора, мови програмування p24. Вихідним кодом генератора є програма на мові Assembler(x86).

3.Проведене тестування компілятора на тестових програмах за наступними пунктами:

3.1.На виявлення лексичних помилок.

3.2.На виявлення синтаксичних помилок.

3.3.Загальна перевірка роботи компілятора.

В результаті виконання даної курсового проекту було засвоєно методи розробки та реалізації компонент систем програмування.

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

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

ДОДАТКИ

Додаток А. Таблиці лексем для тестових прикладів.

Таблиця лексем тестової програми “Лінійний алгоритм”

Lexemes table:

index	lexeme	id	type	ifvalue	row	col
0	#PROGRAM	283	1	0	1	1
1	pROGRAMA	0	2	0	1	10
2	;	256	1	0	1	18
3	VARIABLE	292	1	0	2	1
4	INTEGER16_t	421	1	0	2	10
5	aAAVALUE	1	2	0	2	22
6	,	261	1	0	2	30
7	bBBVALUE	2	2	0	2	32
8	,	261	1	0	2	40
9	xXXVALUE	3	2	0	2	42
10	;	256	1	0	2	50
11	START	301	1	0	3	1
12	SCAN	330	1	0	4	5
13	(272	1	0	4	9
14	aAAVALUE	1	2	0	4	10
15)	275	1	0	4	18
16	SCAN	330	1	0	5	5
17	(272	1	0	5	9
18	bBBVALUE	2	2	0	5	10
19)	275	1	0	5	18
20	xXXVALUE	3	2	0	6	5
21	<=	258	1	0	6	14
22	10	320	4	10	6	17
23	MUL	394	1	0	6	20
24	(272	1	0	6	24
25	aAAVALUE	1	2	0	6	25
26	SUB	398	1	0	6	34
27	bBBVALUE	2	2	0	6	38
28)	275	1	0	6	46
29	ADD	390	1	0	6	48
30	(272	1	0	6	52
31	aAAVALUE	1	2	0	6	53
32	ADD	390	1	0	6	62
33	bBBVALUE	2	2	0	6	66
34)	275	1	0	6	74
35	DIV	402	1	0	6	76
36	10	320	4	10	6	80
37	PRINT	335	1	0	7	5
38	(272	1	0	7	10
39	xXXVALUE	3	2	0	7	11
40)	275	1	0	7	19
41	SCAN	330	1	0	8	5
42	(272	1	0	8	9
43	aAAVALUE	1	2	0	8	10
44)	275	1	0	8	18
45	STOP	307	1	0	9	1

File "../test_programs/file1_lexeme_error.txt" saved.

Press Enter to next step

Таблиця лексем тестової програми “Алгоритм з розгалуженням”

Lexemes table:

index	lexeme	id	type	ifvalue	row	col
0	#PROGRAM	283	1	0	1	1
1	pROGRAMA	0	2	0	1	10
2	;	256	1	0	1	18
3	VARIABLE	292	1	0	2	5
4	INTEGER16_t	421	1	0	2	14
5	aAAVALUE	1	2	0	2	26
6	,	261	1	0	2	34
7	bbbVALUE	2	2	0	2	36
8	,	261	1	0	2	44
9	cCCVALUE	3	2	0	2	46
10	;	256	1	0	2	54
11	START	301	1	0	3	1
12	SCAN	330	1	0	4	5
13	(272	1	0	4	9
14	aAAVALUE	1	2	0	4	10
15)	275	1	0	4	18
16	SCAN	330	1	0	5	5
17	(272	1	0	5	9
18	bbbVALUE	2	2	0	5	10
19)	275	1	0	5	18
20	SCAN	330	1	0	6	5
21	(272	1	0	6	9
22	cCCVALUE	3	2	0	6	10
23)	275	1	0	6	18
24	IF	341	1	0	7	5
25	(272	1	0	7	8
26	aAAVALUE	1	2	0	7	9
27	==	263	1	0	7	18
28	bbbVALUE	2	2	0	7	22
29	AND	410	1	0	7	31
30	aAAVALUE	1	2	0	7	35
31	==	263	1	0	7	44
32	cCCVALUE	3	2	0	7	47
33)	275	1	0	7	55
34	PRINT	335	1	0	8	9
35	(272	1	0	8	14
36	1	320	4	1	8	15
37)	275	1	0	8	16
38	;	256	1	0	8	17
39	ELSE	344	1	0	9	5
40	PRINT	335	1	0	10	9
41	(272	1	0	10	14
42	0	320	4	0	10	15
43)	275	1	0	10	16
44	;	256	1	0	10	17
45	SCAN	330	1	0	11	5
46	(272	1	0	11	9
47	aAAVALUE	1	2	0	11	10
48)	275	1	0	11	18
49	STOP	307	1	0	12	1

File "../test_programs/file1_lexeme_error.txt" saved.

Press Enter to next step

Додаток Б. Код на асемблері, отриманий на виході транслятора для тестових прикладів.

Код отриманий для тестової програми “Лінійний алгоритм”

.686

.model flat, stdcall

option casemap : none

GetStdHandle proto STDCALL, nStdHandle : DWORD

ExitProcess proto STDCALL, uExitCode : DWORD

;MessageBoxA PROTO hwnd : DWORD, lpText : DWORD, lpCaption :
DWORD, uType : DWORD

ReadConsoleA proto STDCALL, hConsoleInput : DWORD, lpBuffer : DWORD,
nNumberOfCharsToRead : DWORD, lpNumberOfCharsRead : DWORD,
lpReserved : DWORD

WriteConsoleA proto STDCALL, hConsoleOutput : DWORD, lpBuffert :
DWORD, nNumberOfCharsToWrite : DWORD, lpNumberOfCharsWritten :
DWORD, lpReserved : DWORD

wsprintfA PROTO C : VARARG

GetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, lpMode :
DWORD

SetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, dwMode :
DWORD

ENABLE_LINE_INPUT EQU 0002h

ENABLE_ECHO_INPUT EQU 0004h

.data

data_start db 8192 dup (0)

;title_msg db "Output:", 0

valueTemp_msg db 256 dup(0)

valueTemp_fmt db "%d", 10, 13, 0

;NumberOfCharsWritten dd 0

hConsoleInput dd 0

hConsoleOutput dd 0

buffer db 128 dup(0)

readOutCount dd ?

.code

start:

db 0E8h, 00h, 00h, 00h, 00h; call NexInstruction

```

;NexInstruction:
    pop esi
    sub esi, 5
    mov edi, esi
    add edi, 000004000h
    mov ecx, edi
    add ecx, 512
    jmp initConsole
putProc PROC
    push eax
    push offset valueTemp_fmt
    push offset valueTemp_msg
    call wsprintfA
    add esp, 12

    ;push 40h
    ;push offset title_msg
    ;push offset valueTemp_msg;
    ;push 0
    ;call MessageBoxA

    push 0
    push 0; offset NumberOfCharsWritten
    push eax; NumberOfCharsToWrite
    push offset valueTemp_msg
    push hConsoleOutput
    call WriteConsoleA

    ret
putProc ENDP

getProc PROC
    push ebp
    mov ebp, esp

    push 0
    push offset readOutCount
    push 15
    push offset buffer + 1
    push hConsoleInput
    call ReadConsoleA

    lea esi, offset buffer
    add esi, readOutCount

```

```
sub esi, 2
call string_to_int
```

```
mov esp, ebp
pop ebp
ret
getProc ENDP
```

```
string_to_int PROC
; input: ESI - string
; output: EAX - value
xor eax, eax
mov ebx, 1
xor ecx, ecx
```

```
convert_loop :
movzx ecx, byte ptr[esi]
test ecx, ecx
jz done
sub ecx, '0'
imul ecx, ebx
add eax, ecx
imul ebx, ebx, 10
dec esi
jmp convert_loop
```

```
done:
ret
string_to_int ENDP
```

```
initConsole:
push -10
call GetStdHandle
mov hConsoleInput, eax
push -11
call GetStdHandle
mov hConsoleOutput, eax
```

```
;push ecx
;push ebx
;push esi
;push edi
;push offset mode
;push hConsoleInput
;call GetConsoleMode
```

```
;mov ebx, eax
;or ebx, ENABLE_LINE_INPUT
;or ebx, ENABLE_ECHO_INPUT
;push ebx
;push hConsoleInput
;call SetConsoleMode
;pop edi
;pop esi
;pop ebx
;pop ecx
```

```
;hw stack save(save esp)
mov ebp, esp
```

```
;";"
```

```
; "4"
add ecx, 4
mov eax, 000000004h
mov dword ptr [ecx], eax
```

```
; "SCAN"
mov eax, dword ptr [ecx]
mov edx, 000000044h
add edx, esi
push ecx
;push ebx
push esi
push edi
call edx
pop edi
pop esi
;pop ebx
pop ecx
mov ebx, dword ptr [ecx]
sub ecx, 4
add ebx, edi
mov dword ptr [ebx], eax
mov ecx, edi ; reset second stack
add ecx, 512 ; reset second stack
```

```
;null statement (non-context)
```

```
; "8"
add ecx, 4
```



```
mov eax, 000000008h
mov dword ptr [ecx], eax

;"SCAN"
mov eax, dword ptr[ecx]
mov edx, 000000044h
add edx, esi
push ecx
;push ebx
push esi
push edi
call edx
pop edi
pop esi
;pop ebx
pop ecx
mov ebx, dword ptr[ecx]
sub ecx, 4
add ebx, edi
mov dword ptr [ebx], eax
mov ecx, edi ; reset second stack
add ecx, 512 ; reset second stack
```

```
;null statement (non-context)
```

```
;"12"
add ecx, 4
mov eax, 00000000Ch
mov dword ptr [ecx], eax
```

```
;"10"
add ecx, 4
mov eax, 00000000Ah
mov dword ptr [ecx], eax
```

```
;"aAAVALUE"
mov eax, edi
add eax, 000000004h
mov eax, dword ptr[eax]
add ecx, 4
mov dword ptr [ecx], eax
```

```
;"bBBVALUE"
mov eax, edi
add eax, 000000008h
```

```
mov eax, dword ptr[eax]
add ecx, 4
mov dword ptr [ecx], eax
```

```
;"SUB"
mov eax, dword ptr[ecx]
sub ecx, 4
sub dword ptr[ecx], eax
mov eax, dword ptr[ecx]
```

```
;"MUL"
mov eax, dword ptr[ecx - 4]
;cdq
imul dword ptr [ecx]
sub ecx, 4
mov dword ptr [ecx], eax
```

```
;"aAAVALUE"
mov eax, edi
add eax, 000000004h
mov eax, dword ptr[eax]
add ecx, 4
mov dword ptr [ecx], eax
```

```
;"bBBVALUE"
mov eax, edi
add eax, 000000008h
mov eax, dword ptr[eax]
add ecx, 4
mov dword ptr [ecx], eax
```

```
;"ADD"
mov eax, dword ptr[ecx]
sub ecx, 4
add dword ptr[ecx], eax
mov eax, dword ptr[ecx]
```

```
;"10"
add ecx, 4
mov eax, 00000000Ah
mov dword ptr [ecx], eax
```

```
;"DIV"
mov eax, dword ptr[ecx - 4]
cdq
```

```
idiv dword ptr [ecx]
sub ecx, 4
mov dword ptr [ecx], eax
```

```
;"ADD"
mov eax, dword ptr[ecx]
sub ecx, 4
add dword ptr[ecx], eax
mov eax, dword ptr[ecx]
```

```
;"<-"
mov eax, dword ptr[ecx]
mov ebx, dword ptr[ecx - 4]
sub ecx, 8
add ebx, edi
mov dword ptr [ebx], eax
mov ecx, edi ; reset second stack
add ecx, 512 ; reset second stack
```

```
;null statement (non-context)
```

```
;"XXXVALUE"
mov eax, edi
add eax, 00000000Ch
mov eax, dword ptr[eax]
add ecx, 4
mov dword ptr [ecx], eax
```

```
;"PRINT"
mov eax, dword ptr[ecx]
mov edx, 00000001Bh
add edx, esi
;push ecx
;push ebx
push esi
push edi
call edx
pop edi
pop esi
;pop ebx
;pop ecx
mov ecx, edi ; reset second stack
add ecx, 512 ; reset second stack
```

```
;null statement (non-context)
```

```
; "4"  
add ecx, 4  
mov eax, 000000004h  
mov dword ptr [ecx], eax
```

```
; "SCAN"  
mov eax, dword ptr [ecx]  
mov edx, 000000044h  
add edx, esi  
push ecx  
; push ebx  
push esi  
push edi  
call edx  
pop edi  
pop esi  
; pop ebx  
pop ecx  
mov ebx, dword ptr [ecx]  
sub ecx, 4  
add ebx, edi  
mov dword ptr [ebx], eax  
mov ecx, edi ; reset second stack  
add ecx, 512 ; reset second stack
```

```
; null statement (non-context)
```

```
; hw stack reset (restore esp)  
mov esp, ebp
```

```
xor eax, eax  
ret
```

```
end start
```

Додаток В. Абстрактне синтаксичне дерево для тестових прикладів

АСД для тестової програми “Лінійний алгоритм”

Abstract Syntax Tree:

```
--program
|--program__part1
|  |--tokenNAME__program_name
|  |  |--tokenNAME
|  |  |  |--"#PROGRAM"
|  |  |--program_name
|  |  |  |--"pROGRAMA"
|  |--tokenSEMICOLON__tokenDATA
|  |  |--tokenSEMICOLON
|  |  |  |--";"
|  |  |--tokenDATA__declaration
|  |  |  |--tokenDATA
|  |  |  |  |--"VARIABLE"
|  |  |  |--declaration
|  |  |  |  |--value_type__ident
|  |  |  |  |  |--value_type
|  |  |  |  |  |  |--"INTEGER16_t"
|  |  |  |  |  |--ident
|  |  |  |  |  |  |--"aAAVALUE"
|  |  |  |  |--other_declaration_ident____iteration_after_one
|  |  |  |  |  |--other_declaration_ident
|  |  |  |  |  |  |--tokenCOMMA
|  |  |  |  |  |  |  |--","
|  |  |  |  |  |  |--ident
|  |  |  |  |  |  |  |--"bBBVALUE"
|  |  |  |  |--other_declaration_ident____iteration_after_one
|  |  |  |  |  |--tokenCOMMA
|  |  |  |  |  |  |--","
|  |  |  |  |  |--ident
|  |  |  |  |  |  |--"xXXVALUE"
|--program__part2
|  |--tokenSEMICOLON__tokenBODY
|  |  |--tokenSEMICOLON
|  |  |  |--";"
|  |  |--tokenBODY
|  |  |  |--"START"
|--statement__iteration_after_two__tokenEND
|  |--statement__iteration_after_two
|  |  |--statement
|  |  |  |--input__first_part
```

[illegible]

[illegible]

Додаток Г. Документований текст програмних модулів.

Файл add.h

```
#define _CRT_SECURE_NO_WARNINGS
#define ADD_CODER(A, B, C, M, R)\
if (A ==* B) C = makeAddCode(B, C, M);
```

```
unsigned char* makeAddCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

Файл and.h

```
#define _CRT_SECURE_NO_WARNINGS
#define AND_CODER(A, B, C, M, R)\
if (A ==* B) C = makeAndCode(B, C, M);
```

```
unsigned char* makeAndCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

Файл cli.h

```
#define _CRT_SECURE_NO_WARNINGS
#define PATH_NAME_LENGTH 2048
```

```
#define MAX_PARAMETERS_SIZE 4096
#define PARAMETERS_COUNT 32
// #define INPUT_FILENAME_PARAMETER 0
#define INPUT_FILENAME_WITH_EXTENSION_PARAMETER 1
#define OUT_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER 2
#define OUT_LEXEME_ERROR_FILENAME_WITH_EXTENSION_PARAMETER 3
#define OUT_AST_FILENAME_WITH_EXTENSION_PARAMETER 4
#define OUT_SYNTAX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER 5
#define OUT_SEMANTIX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER 6
#define OUT_PREPARED_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER 7
#define OUT_C_FILENAME_WITH_EXTENSION_PARAMETER 8
#define OUT_ASSEMBLY_FILENAME_WITH_EXTENSION_PARAMETER 9
#define OUT_OBJECT_FILENAME_WITH_EXTENSION_PARAMETER 10
#define OUT_BINARY_FILENAME_WITH_EXTENSION_PARAMETER 11
```

```
#include "../src/include/def.h"
#include "../src/include/config.h"
#include "../src/include/generator/generator.h"
#include "../src/include/lexica/lexica.h"
#define DEFAULT_INPUT_FILENAME "../test_programs/file1.cwl"
```

```
extern unsigned long long int mode;
extern char parameters[PARAMETERS_COUNT][MAX_PARAMETERS_SIZE];
```

```
void comandLineParser(int argc, char* argv[], unsigned long long int* mode,
char>(*parameters)[MAX_PARAMETERS_SIZE]);
// after using this function use free(void *) function to release text buffer
size_t loadSource(char** text, char* fileName);
```

Файл config.h

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025 *
* file: config.h *
```

```
*
                                (draft!) *
*****/
```

```
#include "../include/def.h"
#define LEXICAL_ANALYSIS_MODE 1
#define SEMANTIC_ANALYSIS_MODE 2
#define FULL_COMPILER_MODE 4

#define DEBUG_MODE 512

#define DEFAULT_MODE (DEBUG_MODE | LEXICAL_ANALYSIS_MODE)
#define DEFAULT_MODE (DEBUG_MODE | LEXICAL_ANALYSIS_MODE |
SYNTAX_ANALYSIS_MODE | SEMANTIC_ANALYSIS_MODE |
MAKE_ASSEMBLY | MAKE_BINARY)
```

```
#define TOKENS_RE      ";|<-|,|=|!=|:|\\(|\\)|!>|!<|[_#0-9A-Za-z]+|[^\t\r\f\v\n]"
#define KEYWORDS_RE    ";|<-
|,|=|!=|:|\\(|\\)|!>|!<|#PROGRAM|VARIABLE|START|STOP|EXIT|CONTINUE|E
ND|SCAN|PRINT|IF|ELSE|FOR|TO|DOWNTO|DO|WHILE|REPEAT|UNTIL|GO
TO|ADD|MUL|SUB|DIV|MOD|AND|OR|NOT|INTEGER16_t"
#define IDENTIFIERS_RE "[a-z][A-Z]{7}"
#define UNSIGNEDVALUES_RE "0|[1-9][0-9]*"
```

```
#define PROGRAM_FORMAT1 \
{"tokenNAME__program_name", 2, {"tokenNAME","program_name"}},\
{"tokenDATA__declaration", 2, {"tokenDATA","declaration"}},\
{"tokenSEMICOLON__tokenDATA", 2,\
{"tokenSEMICOLON","tokenDATA"}},\
{"tokenSEMICOLON__tokenDATA", 2,\
{"tokenSEMICOLON","tokenDATA__declaration"}},\
{"tokenNAME__program_name__tokenSEMICOLON__tokenDATA", 2,\
{"tokenNAME__program_name","tokenSEMICOLON__tokenDATA"}},\
{"program____part1", 2,\
{"tokenNAME__program_name__tokenSEMICOLON__tokenDATA","tokenSE
MICOLON"}},\
{"statement__tokenEND", 2, {"statement","tokenEND"}},\
{"statement____iteration_after_two__tokenEND", 2,\
{"statement____iteration_after_two","tokenEND"}},\
{"program____part2", 2,\
{"tokenBODY","statement____iteration_after_two__tokenEND"}},\
{"program____part2", 2, {"tokenBODY","statement__tokenEND"}},\
{"program____part2", 2, {"tokenBODY","tokenEND"}},\
{"program", 2, {"program____part1","program____part2"}},
```

```
#define PROGRAM_FORMAT \
{"tokenNAME__program_name", 2, {"tokenNAME","program_name"}},\
{"tokenDATA__declaration", 2, {"tokenDATA","declaration"}},\
{"tokenSEMICOLON__tokenDATA", 2,\
{"tokenSEMICOLON","tokenDATA__declaration"}},\
{"tokenSEMICOLON__tokenDATA", 2,\
{"tokenSEMICOLON","tokenDATA"}},\
{"program____part1", 2,\
{"tokenNAME__program_name","tokenSEMICOLON__tokenDATA"}},\
{"statement__tokenEND", 2, {"statement","tokenEND"}},\
{"statement____iteration_after_two__tokenEND", 2,\
{"statement____iteration_after_two","tokenEND"}},\
{"tokenSEMICOLON__tokenBODY", 2,\
{"tokenSEMICOLON","tokenBODY"}},\
{"program____part2", 2,\
{"tokenSEMICOLON__tokenBODY","statement____iteration_after_two__tokenEND"}},\
{"program____part2", 2, {"tokenSEMICOLON__tokenBODY","tokenEND"}},\
{"program", 2, {"program____part1","program____part2"}},
```

// first column of the cw term paper option

```
#define PROGRAM_FORMAT_OLD \
{"tokenNAME__program_name", 2, {"tokenNAME","program_name"}},\
{"tokenSEMICOLON__tokenBODY", 2,\
{"tokenSEMICOLON","tokenBODY"}},\
{"tokenDATA__declaration", 2, {"tokenDATA","declaration"}},\
{"tokenNAME__program_name__tokenSEMICOLON__tokenBODY", 2,\
{"tokenNAME__program_name","tokenSEMICOLON__tokenBODY"}},\
{"program____part1", 2,\
{"tokenNAME__program_name__tokenSEMICOLON__tokenBODY","tokenDATA__declaration"}},\
{"program____part1", 2,\
{"tokenNAME__program_name__tokenSEMICOLON__tokenBODY","tokenDATA"}},\
{"statement__tokenEND", 2, {"statement","tokenEND"}},\
{"statement____iteration_after_two__tokenEND", 2,\
{"statement____iteration_after_two","tokenEND"}},\
{"program____part2", 2,\
{"tokenSEMICOLON","statement____iteration_after_two__tokenEND"}},\
{"program____part2", 2, {"tokenSEMICOLON","statement__tokenEND"}},\
{"program____part2", 2, {"tokenSEMICOLON","tokenEND"}},\
{"program", 2, {"program____part1","program____part2"}},
```

```

#define T_NAME_0 "#PROGRAM"
#define T_BODY_0 "START"
#define T_DATA_0 "VARIABLE"
#define T_DATA_TYPE_0 "INTEGER16_t"
#define T_DATA_TYPE_1 ""
#define T_DATA_TYPE_2 ""
#define T_DATA_TYPE_3 ""
//
#define T_NOT_0 "NOT"
#define T_NOT_1 ""
#define T_NOT_2 ""
#define T_NOT_3 ""
#define T_AND_0 "AND"
#define T_AND_1 ""
#define T_AND_2 ""
#define T_AND_3 ""
#define T_OR_0 "OR"
#define T_OR_1 ""
#define T_OR_2 ""
#define T_OR_3 ""
//
#define T_EQUAL_0 "=="
#define T_EQUAL_1 ""
#define T_EQUAL_2 ""
#define T_EQUAL_3 ""
#define T_NOT_EQUAL_0 "!="
#define T_NOT_EQUAL_1 ""
#define T_NOT_EQUAL_2 ""
#define T_NOT_EQUAL_3 ""
#define T_LESS_OR_EQUAL_0 ">="
#define T_LESS_OR_EQUAL_1 ""
#define T_LESS_OR_EQUAL_2 ""
#define T_LESS_OR_EQUAL_3 ""
#define T_GREATER_OR_EQUAL_0 "<="
#define T_GREATER_OR_EQUAL_1 ""
#define T_GREATER_OR_EQUAL_2 ""
#define T_GREATER_OR_EQUAL_3 ""
//
#define T_ADD_0 "ADD"
#define T_ADD_1 ""
#define T_ADD_2 ""
#define T_ADD_3 ""
#define T_SUB_0 "SUB"
#define T_SUB_1 ""

```

```

#define T_SUB_2 ""
#define T_SUB_3 ""
#define T_MUL_0 "MUL"
#define T_MUL_1 ""
#define T_MUL_2 ""
#define T_MUL_3 ""
#define T_DIV_0 "DIV"
#define T_DIV_1 ""
#define T_DIV_2 ""
#define T_DIV_3 ""
#define T_MOD_0 "MOD"
#define T_MOD_1 ""
#define T_MOD_2 ""
#define T_MOD_3 ""
//
#define T_BIND_RIGHT_TO_LEFT_0 "<-"
#define T_BIND_RIGHT_TO_LEFT_1 ""
#define T_BIND_RIGHT_TO_LEFT_2 ""
#define T_BIND_RIGHT_TO_LEFT_3 ""
//
#define T_COMA_0 ","
#define T_COMA_1 ""
#define T_COMA_2 ""
#define T_COMA_3 ""
#define T_COLON_0 ":"
#define T_COLON_1 ""
#define T_COLON_2 ""
#define T_COLON_3 ""
#define T_GOTO_0 "GOTO"
#define T_GOTO_1 ""
#define T_GOTO_2 ""
#define T_GOTO_3 ""
//
#define T_IF_0 "IF"
#define T_IF_1 "("
#define T_IF_2 ""
#define T_IF_3 ""
#define T_THEN_0 ")"
#define T_THEN_1 ""
#define T_THEN_2 ""
#define T_THEN_3 ""
#define T_ELSE_0 "ELSE"
#define T_ELSE_1 ""
#define T_ELSE_2 ""
#define T_ELSE_3 ""

```

```
//
#define T_FOR_0 "FOR"
#define T_FOR_1 ""
#define T_FOR_2 ""
#define T_FOR_3 ""
#define T_TO_0 "TO"
#define T_TO_1 ""
#define T_TO_2 ""
#define T_TO_3 ""
#define T_DOWNT0_0 "DOWNT0"
#define T_DOWNT0_1 ""
#define T_DOWNT0_2 ""
#define T_DOWNT0_3 ""
#define T_DO_0 "DO"
#define T_DO_1 ""
#define T_DO_2 ""
#define T_DO_3 ""
//
#define T_WHILE_0 "WHILE"
#define T_WHILE_1 ""
#define T_WHILE_2 ""
#define T_WHILE_3 ""
#define T_CONTINUE_WHILE_0 "CONTINUE"
#define T_CONTINUE_WHILE_1 "WHILE"
#define T_CONTINUE_WHILE_2 ""
#define T_CONTINUE_WHILE_3 ""
#define T_EXIT_WHILE_0 "EXIT"
#define T_EXIT_WHILE_1 "WHILE"
#define T_EXIT_WHILE_2 ""
#define T_EXIT_WHILE_3 ""
#define T_END_WHILE_0 "END"
#define T_END_WHILE_1 "WHILE"
#define T_END_WHILE_2 ""
#define T_END_WHILE_3 ""
//
#define T_REPEAT_0 "REPEAT"
#define T_REPEAT_1 ""
#define T_REPEAT_2 ""
#define T_REPEAT_3 ""
#define T_UNTIL_0 "UNTIL"
#define T_UNTIL_1 ""
#define T_UNTIL_2 ""
#define T_UNTIL_3 ""
//
#define T_INPUT_0 "SCAN"
```

```

#define T_INPUT_1 ""
#define T_INPUT_2 ""
#define T_INPUT_3 ""
#define T_OUTPUT_0 "PRINT"
#define T_OUTPUT_1 ""
#define T_OUTPUT_2 ""
#define T_OUTPUT_3 ""
//
#define T_RLBIND_0 "<-"
#define T_RLBIND_1 ""
#define T_RLBIND_2 ""
#define T_RLBIND_3 ""
//
#define T_SEMICOLON_0 ";"
#define T_SEMICOLON_1 ""
#define T_SEMICOLON_2 ""
#define T_SEMICOLON_3 ""
//
#define T_BEGIN_0 "START"
#define T_BEGIN_1 ""
#define T_BEGIN_2 ""
#define T_BEGIN_3 ""
#define T_END_0 "STOP"
#define T_END_1 ""
#define T_END_2 ""
#define T_END_3 ""
//
#define T_NULL_STATEMENT_0 "NULL"
#define T_NULL_STATEMENT_1 "STATEMENT"
#define T_NULL_STATEMENT_2 ""
#define T_NULL_STATEMENT_3 ""

```

```

#ifndef TOKEN_STRUCT_NAME_
#define TOKEN_STRUCT_NAME_
DECLENUM(TokenStructName,
    MULTI_TOKEN_NOT,
    MULTI_TOKEN_AND,
    MULTI_TOKEN_OR,

    MULTI_TOKEN_EQUAL,
    MULTI_TOKEN_NOT_EQUAL,
    MULTI_TOKEN_LESS_OR_EQUAL,
    MULTI_TOKEN_GREATER_OR_EQUAL,

```

```
MULTI_TOKEN_ADD,
MULTI_TOKEN_SUB,
MULTI_TOKEN_MUL,
MULTI_TOKEN_DIV,
MULTI_TOKEN_MOD,

MULTI_TOKEN_BIND_RIGHT_TO_LEFT,

MULTI_TOKEN_COLON,
MULTI_TOKEN_GOTO,

MULTI_TOKEN_IF,
//          MULTI_TOKEN_IF_, // don't change this!
MULTI_TOKEN_THEN,
//          MULTI_TOKEN_THEN_, // don't change this!
MULTI_TOKEN_ELSE,

MULTI_TOKEN_FOR,
MULTI_TOKEN_TO,
MULTI_TOKEN_DOWNTO,
MULTI_TOKEN_DO,

//
MULTI_TOKEN_WHILE,
/*while special statement*/MULTI_TOKEN_CONTINUE_WHILE,
/*while special statement*/MULTI_TOKEN_EXIT_WHILE,
MULTI_TOKEN_END_WHILE,
//

//
MULTI_TOKEN_REPEAT,
MULTI_TOKEN_UNTIL,
//

//
MULTI_TOKEN_INPUT,
MULTI_TOKEN_OUTPUT,
//

//
MULTI_TOKEN_RLBIND,
//

MULTI_TOKEN_SEMICOLON,
```



```

MULTI_TOKEN_BEGIN,
MULTI_TOKEN_END,

//

MULTI_TOKEN_NULL_STATEMENT
);

#define PROCESS_TOKENS(...) HANDLE_TOKENS(__VA_ARGS__)
#define TOKENS_FOR_MULTI_TOKEN(A, B, C, D) A, B, C, D
#define TOKENS_FOR_MULTI_TOKEN_BITWISE_NOT
TOKENS_FOR_MULTI_TOKEN("~", "", "", "")
#define INIT_TOKEN_STRUCT_NAME() static void initTokenStruct(){
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, NOT)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, AND)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, OR)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, EQUAL)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, NOT_EQUAL)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct,
LESS_OR_EQUAL)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct,
GREATER_OR_EQUAL)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, ADD)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, SUB)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, MUL)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, DIV)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, MOD)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct,
BIND_RIGHT_TO_LEFT)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, COLON)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, GOTO)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, IF)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, THEN)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, ELSE)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, FOR)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, TO)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, DOWNTO)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, DO)\
\

```

```

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, WHILE)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct,
CONTINUE_WHILE)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, EXIT_WHILE)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, END_WHILE)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, REPEAT)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, UNTIL)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, INPUT)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, OUTPUT)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, RLBIND)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, SEMICOLON)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, BEGIN)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, END)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct,
NULL_STATEMENT)\
} char intitTokenStruct_ = (intitTokenStruct(), 0);
#define MAX_TOKEN_STRUCT_ELEMENT_COUNT
GET_ENUM_SIZE(TokenStructName)
#define MAX_TOKEN_STRUCT_ELEMENT_PART_COUNT 4
#endif

extern char*
tokenStruct[MAX_TOKEN_STRUCT_ELEMENT_COUNT][MAX_TOKEN_ST
RUCT_ELEMENT_PART_COUNT];

#define CONFIGURABLE_GRAMMAR {\
    {"labeled_point", 2, {"ident", "tokenCOLON"}},\
    {"goto_label", 2, {"tokenGOTO", "ident"}},\
    {"program_name", 1, {"ident_terminal"}},\
    {"value_type", 1, {T_DATA_TYPE_0}},\
    {"other_declaration_ident", 2, {"tokenCOMMA", "ident"}},\
    {"other_declaration_ident____iteration_after_one", 2,
{"other_declaration_ident", "other_declaration_ident____iteration_after_one"}},\
    {"other_declaration_ident____iteration_after_one", 2, {"tokenCOMMA",
"ident"}},\
    {"value_type__ident", 2, {"value_type", "ident"}},\
    {"declaration", 2, {"value_type__ident",
"other_declaration_ident____iteration_after_one"}},\
    {"declaration", 2, {"value_type", "ident"}},\

```

```

\
    {"unary_operator", 1, {T_NOT_0}},\
    {"unary_operator", 1, {T_SUB_0}},\
    {"unary_operator", 1, {T_ADD_0}},\
    {"binary_operator", 1, {T_AND_0}},\
    {"binary_operator", 1, {T_OR_0}},\
    {"binary_operator", 1, {T_EQUAL_0}},\
    {"binary_operator", 1, {T_NOT_EQUAL_0}},\
    {"binary_operator", 1, {T_LESS_OR_EQUAL_0}},\
    {"binary_operator", 1, {T_GREATER_OR_EQUAL_0}},\
    {"binary_operator", 1, {T_ADD_0}},\
    {"binary_operator", 1, {T_SUB_0}},\
    {"binary_operator", 1, {T_MUL_0}},\
    {"binary_operator", 1, {T_DIV_0}},\
    {"binary_operator", 1, {T_MOD_0}},\
    {"binary_action", 2, {"binary_operator", "expression"}},\
\
    {"left_expression", 2,
{"tokenGROUPEXPRESSIONBEGIN__expression", "tokenGROUPEXPRESSIONIO
NEND"}},\
    {"left_expression", 2, {"unary_operator", "expression"}},\
    {"left_expression", 1, {"ident_terminal"}},\
    {"left_expression", 1, {"value_terminal"}},\
    {"binary_action____iteration_after_two", 2,
{"binary_action", "binary_action____iteration_after_two"}},\
    {"binary_action____iteration_after_two", 2,
{"binary_action", "binary_action"}},\
    {"expression", 2,
{"left_expression", "binary_action____iteration_after_two"}},\
    {"expression", 2, {"left_expression", "binary_action"}},\
    {"expression", 2,
{"tokenGROUPEXPRESSIONBEGIN__expression", "tokenGROUPEXPRESSIONIO
NEND"}},\
    {"expression", 2, {"unary_operator", "expression"}},\
    {"expression", 1, {"ident_terminal"}},\
    {"expression", 1, {"value_terminal"}},\
\
    {"tokenGROUPEXPRESSIONBEGIN__expression", 2,
{"tokenGROUPEXPRESSIONBEGIN", "expression"}},\
    {"group_expression", 2,
{"tokenGROUPEXPRESSIONBEGIN__expression", "tokenGROUPEXPRESSIONIO
NEND"}},\
\
    {"bind_right_to_left", 2, {"ident", "rl_expression"}},\
\

```

```

        {"body_for_true", 2,
{"statement_in_while_body___iteration_after_two","tokenSEMICOLON"}}},\
        {"body_for_true", 2, {"statement_in_while_body","tokenSEMICOLON"}}},\
        {"body_for_true", 1, {T_SEMICOLON_0}}},\
        {"tokenELSE__statement_in_while_body", 2,
{"tokenELSE","statement_in_while_body"}}},\
        {"tokenELSE__statement_in_while_body___iteration_after_two", 2,
{"tokenELSE","statement_in_while_body___iteration_after_two"}}},\
        {"body_for_false", 2,
{"tokenELSE__statement_in_while_body___iteration_after_two","tokenSEMICOLON"}}},\
        {"body_for_false", 2,
{"tokenELSE__statement_in_while_body","tokenSEMICOLON"}}},\
        {"body_for_false", 2, {"tokenELSE","tokenSEMICOLON"}}},\
        {"tokenIF__tokenGROUPEXPRESSIONBEGIN", 2,
{"tokenIF","tokenGROUPEXPRESSIONBEGIN"}}},\
        {"expression__tokenGROUPEXPRESSIONEND", 2,
{"expression","tokenGROUPEXPRESSIONEND"}}},\

{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEXPRESSIONEND", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN","expression__tokenGROUPEXPRESSIONEND"}}},\
        {"body_for_true__body_for_false", 2, {"body_for_true","body_for_false"}}},\
        {"cond_block", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEXPRESSIONEND","body_for_true__body_for_false"}}},\
        {"cond_block", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEXPRESSIONEND","body_for_true"}}},\
\
        {"cycle_counter", 1, {"ident_terminal"}}},\
        {"rl_expression", 2, {"tokenRLBIND","expression"}}},\
        {"cycle_counter_init", 2, {"cycle_counter","rl_expression"}}},\
        {"cycle_counter_last_value", 1, {"value_terminal"}}},\
        {"cycle_body", 2, {"tokenDO","statement___iteration_after_two"}}},\
        {"cycle_body", 2, {"tokenDO","statement"}}},\
        {"tokenFOR__cycle_counter_init", 2, {"tokenFOR","cycle_counter_init"}}},\
        {"tokenTO__cycle_counter_last_value", 2,
{"tokenTO","cycle_counter_last_value"}}},\
        {"tokenFOR__cycle_counter_init__tokenTO__cycle_counter_last_value", 2,
{"tokenFOR__cycle_counter_init","tokenTO__cycle_counter_last_value"}}},\
        {"cycle_body__tokenSEMICOLON", 2,
{"cycle_body","tokenSEMICOLON"}}},\

```

```

        {"forto_cycle", 2,
{"tokenFOR__cycle_counter_init__tokenTO__cycle_counter_last_value","cycle_b
ody__tokenSEMICOLON"}}},\
\
        {"continue_while", 2, {"tokenCONTINUE","tokenWHILE"}}},\
        {"exit_while", 2, {"tokenEXIT","tokenWHILE"}}},\
        {"tokenWHILE__expression", 2, {"tokenWHILE","expression"}}},\
        {"tokenEND__tokenWHILE", 2, {"tokenEND","tokenWHILE"}}},\
        {"tokenWHILE__expression__statement_in_while_body", 2,
{"tokenWHILE__expression","statement_in_while_body"}}},\

{"tokenWHILE__expression__statement_in_while_body____iteration_after_two",
2,
{"tokenWHILE__expression","statement_in_while_body____iteration_after_two"
}},\
        {"while_cycle", 2,
{"tokenWHILE__expression__statement_in_while_body____iteration_after_two",
"tokenEND__tokenWHILE"}}},\
        {"while_cycle", 2,
{"tokenWHILE__expression__statement_in_while_body","tokenEND__tokenWH
ILE"}}},\
        {"while_cycle", 2,
{"tokenWHILE__expression","tokenEND__tokenWHILE"}}},\
\
        {"tokenUNTIL__expression", 2, {"tokenUNTIL","expression"}}},\
        {"tokenREPEAT__statement____iteration_after_two", 2,
{"tokenREPEAT","statement____iteration_after_two"}}},\
        {"tokenREPEAT__statement", 2, {"tokenREPEAT","statement"}}},\
        {"repeat_until_cycle", 2,
{"tokenREPEAT__statement____iteration_after_two","tokenUNTIL__expression"
}},\
        {"repeat_until_cycle", 2,
{"tokenREPEAT__statement","tokenUNTIL__expression"}}},\
        {"repeat_until_cycle", 2, {"tokenREPEAT","tokenUNTIL__expression"}}},\
\
        {"input__first_part", 2,
{"tokenGET","tokenGROUPEXPRESSIONBEGIN"}}},\
        {"input__second_part", 2, {"ident","tokenGROUPEXPRESSIONEND"}}},\
        {"input", 2, {"input__first_part","input__second_part"}}},\
\
        {"output__first_part", 2,
{"tokenPUT","tokenGROUPEXPRESSIONBEGIN"}}},\
        {"output__second_part", 2,
{"expression","tokenGROUPEXPRESSIONEND"}}},\
        {"output", 2, {"output__first_part","output__second_part"}}},\

```

```

\
    {"statement", 2, {"ident","rl_expression"}},\
    {"statement", 2, {"lr_expression","ident"}},\
    {"statement", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEX
PRESSIONEND","body_for_true__body_for_false"}},\
    {"statement", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEX
PRESSIONEND","body_for_true"}},\
    {"statement", 2,
{"tokenFOR__cycle_counter_init__tokenTO__cycle_counter_last_value","cycle_b
ody__tokenSEMICOLON"}},\
    {"statement", 2,
{"tokenWHILE__expression__statement_in_while_body____iteration_after_two",
"tokenEND__tokenWHILE"}},\
    {"statement", 2,
{"tokenWHILE__expression__statement_in_while_body","tokenEND__tokenWH
ILE"}},\
    {"statement", 2,
{"tokenWHILE__expression","tokenEND__tokenWHILE"}},\
    {"statement", 2,
{"tokenREPEAT__statement____iteration_after_two","tokenUNTIL__expression"
}},\
    {"statement", 2,
{"tokenREPEAT__statement","tokenUNTIL__expression"}},\
    {"statement", 2, {"tokenREPEAT","tokenUNTIL__expression"}},\
    {"statement", 2, {"ident","tokenCOLON"}},\
    {"statement", 2, {"tokenGOTO","ident"}},\
    {"statement", 2, {"input__first_part","input__second_part"}},\
    {"statement", 2, {"output__first_part","output__second_part"}},\
    {"statement____iteration_after_two", 2,
{"statement","statement____iteration_after_two"}},\
    {"statement____iteration_after_two", 2, {"statement","statement"}},\
\
    {"statement_in_while_body", 2, {"ident","rl_expression"}},\
    {"statement_in_while_body", 2, {"lr_expression","ident"}},\
    {"statement_in_while_body", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEX
PRESSIONEND","body_for_true__body_for_false"}},\
    {"statement_in_while_body", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEX
PRESSIONEND","body_for_true"}},\
    {"statement_in_while_body", 2,
{"tokenFOR__cycle_counter_init__tokenTO__cycle_counter_last_value","cycle_b
ody__tokenSEMICOLON"}},\

```

```

        { "statement_in_while_body", 2,
{"tokenWHILE__expression__statement_in_while_body____iteration_after_two",
"tokenEND__tokenWHILE"}}},\
        { "statement_in_while_body", 2,
{"tokenWHILE__expression__statement_in_while_body","tokenEND__tokenWHILE"}},\
        { "statement_in_while_body", 2,
{"tokenWHILE__expression","tokenEND__tokenWHILE"}}},\
        { "statement_in_while_body", 2,
{"tokenREPEAT__statement____iteration_after_two","tokenUNTIL__expression"}},\
        { "statement_in_while_body", 2,
{"tokenREPEAT__statement","tokenUNTIL__expression"}}},\
        { "statement_in_while_body", 2,
{"tokenREPEAT","tokenUNTIL__expression"}}},\
        { "statement_in_while_body", 2, {"ident","tokenCOLON"}}},\
        { "statement_in_while_body", 2, {"tokenGOTO","ident"}}},\
        { "statement_in_while_body", 2,
{"input__first_part","input__second_part"}}},\
        { "statement_in_while_body", 2,
{"output__first_part","output__second_part"}}},\
        { "statement_in_while_body", 2, {"tokenCONTINUE","tokenWHILE"}}},\
        { "statement_in_while_body", 2, {"tokenEXIT","tokenWHILE"}}},\
        { "statement_in_while_body____iteration_after_two", 2,
{"statement_in_while_body","statement_in_while_body____iteration_after_two"}},\
        { "statement_in_while_body____iteration_after_two", 2,
{"statement_in_while_body","statement_in_while_body"}}},\
\

```

PROGRAM_FORMAT\

```

{"tokenCOLON", 1, {T_COLON_0}}},\
{"tokenGOTO", 1, {T_GOTO_0}}},\
{"tokenINTEGER16", 1, {T_DATA_TYPE_0}}},\
{"tokenCOMMA", 1, {T_COMA_0}}},\
{"tokenNOT", 1, {T_NOT_0}}},\
{"tokenAND", 1, {T_AND_0}}},\
{"tokenOR", 1, {T_OR_0}}},\
{"tokenEQUAL", 1, {T_EQUAL_0}}},\
{"tokenNOTEQUAL", 1, {T_NOT_EQUAL_0}}},\
{"tokenLESSOREQUAL", 1, {T_LESS_OR_EQUAL_0}}},\
{"tokenGREATEROREQUAL", 1, {T_GREATER_OR_EQUAL_0}}},\
{"tokenPLUS", 1, {T_ADD_0}}},\
{"tokenMINUS", 1, {T_SUB_0}}},\
{"tokenMUL", 1, {T_MUL_0}}},\

```

```

{"tokenDIV", 1, {T_DIV_0}},\
{"tokenMOD", 1, {T_MOD_0}},\
{"tokenGROUPEXPRESSIONBEGIN", 1, {"("}},\
{"tokenGROUPEXPRESSIONEND", 1, {"("}},\
{"tokenRLBIND", 1, {T_RLBIND_0}},\
{"tokenELSE", 1, {T_ELSE_0}},\
{"tokenIF", 1, {T_IF_0}},\
{"tokenDO", 1, {T_DO_0}},\
{"tokenFOR", 1, {T_FOR_0}},\
{"tokenTO", 1, {T_TO_0}},\
{"tokenWHILE", 1, {T_WHILE_0}},\
{"tokenCONTINUE", 1, {T_CONTINUE_WHILE_0}},\
{"tokenEXIT", 1, {T_EXIT_WHILE_0}},\
{"tokenREPEAT", 1, {T_REPEAT_0}},\
{"tokenUNTIL", 1, {T_UNTIL_0}},\
{"tokenGET", 1, {T_INPUT_0}},\
{"tokenPUT", 1, {T_OUTPUT_0}},\
{"tokenNAME", 1, {T_NAME_0}},\
{"tokenBODY", 1, {T_BODY_0}},\
{"tokenDATA", 1, {T_DATA_0}},\
{"tokenEND", 1, {T_END_0}},\
{"tokenSEMICOLON", 1, {T_SEMICOLON_0}},\
\
{"value", 1, {"value_terminal"}},\
\
{"ident", 1, {"ident_terminal"}},\
\
{"", 2, {"", ""}}\
},\
176,\
"program"

```

```

#define ORIGINAL_GRAMMAR {\
  {"labeled_point", 2, {"ident", "tokenCOLON"}},\
  {"goto_label", 2, {"tokenGOTO", "ident"}},\
  {"program_name", 1, {"ident_terminal"}},\
  {"value_type", 1, {"INTEGER16"}},\
  {"other_declaration_ident", 2, {"tokenCOMMA", "ident"}},\
  {"other_declaration_ident____iteration_after_one", 2,\
  {"other_declaration_ident", "other_declaration_ident____iteration_after_one"}},\
  {"other_declaration_ident____iteration_after_one", 2, {"tokenCOMMA",\
  "ident"}},\
  {"value_type__ident", 2, {"value_type", "ident"}},\
  {"declaration", 2, {"value_type__ident",\
  "other_declaration_ident____iteration_after_one"}},\

```



```

    {"declaration", 2, {"value_type", "ident"}},\
\
    {"unary_operator", 1, {"NOT"}},\
    {"unary_operator", 1, {"-"}},\
    {"unary_operator", 1, {"+"}},\
    {"binary_operator", 1, {"AND"}},\
    {"binary_operator", 1, {"OR"}},\
    {"binary_operator", 1, {"=="}},\
    {"binary_operator", 1, {"!="}},\
    {"binary_operator", 1, {"<="}},\
    {"binary_operator", 1, {">="}},\
    {"binary_operator", 1, {"+"}},\
    {"binary_operator", 1, {"-"}},\
    {"binary_operator", 1, {"*"}},\
    {"binary_operator", 1, {"DIV"}},\
    {"binary_operator", 1, {"MOD"}},\
    {"binary_action", 2, {"binary_operator", "expression"}},\
\
    {"left_expression", 2,
{"tokenGROUPEXPRESSIONBEGIN__expression", "tokenGROUPEXPRESSIONIO
NEND"}},\
    {"left_expression", 2, {"unary_operator", "expression"}},\
    {"left_expression", 1, {"ident_terminal"}},\
    {"left_expression", 1, {"value_terminal"}},\
    {"binary_action____iteration_after_two", 2,
{"binary_action", "binary_action____iteration_after_two"}},\
    {"binary_action____iteration_after_two", 2,
{"binary_action", "binary_action"}},\
    {"expression", 2, {"left_expression", "binary_action____iteration_after_two"}},\
    {"expression", 2, {"left_expression", "binary_action"}},\
    {"expression", 2,
{"tokenGROUPEXPRESSIONBEGIN__expression", "tokenGROUPEXPRESSIONIO
NEND"}},\
    {"expression", 2, {"unary_operator", "expression"}},\
    {"expression", 1, {"ident_terminal"}},\
    {"expression", 1, {"value_terminal"}},\
\
    {"tokenGROUPEXPRESSIONBEGIN__expression", 2,
{"tokenGROUPEXPRESSIONBEGIN", "expression"}},\
    {"group_expression", 2,
{"tokenGROUPEXPRESSIONBEGIN__expression", "tokenGROUPEXPRESSIONIO
NEND"}},\
\
    {"bind_right_to_left", 2, {"ident", "rl_expression"}},\
    {"bind_left_to_right", 2, {"lr_expression", "ident"}},\

```

```

\
    {"body_for_true", 2,
{"statement_in_while_body____iteration_after_two","tokenSEMICOLON"}}},\
    {"body_for_true", 2, {"statement_in_while_body","tokenSEMICOLON"}}},\
    {"body_for_true", 1, {";"}}},\
    {"tokenELSE__statement_in_while_body", 2,
{"tokenELSE","statement_in_while_body"}}},\
    {"tokenELSE__statement_in_while_body____iteration_after_two", 2,
{"tokenELSE","statement_in_while_body____iteration_after_two"}}},\
    {"body_for_false", 2,
{"tokenELSE__statement_in_while_body____iteration_after_two","tokenSEMICOLON"}}},\
    {"body_for_false", 2,
{"tokenELSE__statement_in_while_body","tokenSEMICOLON"}}},\
    {"body_for_false", 2, {"tokenELSE","tokenSEMICOLON"}}},\
    {"tokenIF__tokenGROUPEXPRESSIONBEGIN", 2,
{"tokenIF","tokenGROUPEXPRESSIONBEGIN"}}},\
    {"expression__tokenGROUPEXPRESSIONEND", 2,
{"expression","tokenGROUPEXPRESSIONEND"}}},\

{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEXPRESSIONEND", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN","expression__tokenGROUPEXPRESSIONEND"}}},\
    {"body_for_true__body_for_false", 2, {"body_for_true","body_for_false"}}},\
    {"cond_block", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEXPRESSIONEND","body_for_true__body_for_false"}}},\
    {"cond_block", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEXPRESSIONEND","body_for_true"}}},\
\
    {"cycle_counter", 1, {"ident_terminal"}}},\
    {"rl_expression", 2, {"tokenRLBIND","expression"}}},\
    {"lr_expression", 2, {"expression","tokenLRBIND"}}},\
    {"cycle_counter_init", 2, {"cycle_counter","rl_expression"}}},\
    {"cycle_counter_init", 2, {"lr_expression","cycle_counter"}}},\
    {"cycle_counter_last_value", 1, {"value_terminal"}}},\
    {"cycle_body", 2, {"tokenDO","statement____iteration_after_two"}}},\
    {"cycle_body", 2, {"tokenDO","statement"}}},\
    {"tokenFOR__cycle_counter_init", 2, {"tokenFOR","cycle_counter_init"}}},\
    {"tokenTO__cycle_counter_last_value", 2,
{"tokenTO","cycle_counter_last_value"}}},\
    {"tokenFOR__cycle_counter_init__tokenTO__cycle_counter_last_value", 2,
{"tokenFOR__cycle_counter_init","tokenTO__cycle_counter_last_value"}}},\

```

```

    {"cycle_body__tokenSEMICOLON", 2,
{"cycle_body","tokenSEMICOLON"}},\
    {"for_to_cycle", 2,
{"tokenFOR__cycle_counter_init__tokenTO__cycle_counter_last_value","cycle_b
ody__tokenSEMICOLON"}},\
\
    {"continue_while", 2, {"tokenCONTINUE","tokenWHILE"}},\
    {"exit_while", 2, {"tokenEXIT","tokenWHILE"}},\
    {"tokenWHILE__expression", 2, {"tokenWHILE","expression"}},\
    {"tokenEND__tokenWHILE", 2, {"tokenEND","tokenWHILE"}},\
    {"tokenWHILE__expression__statement_in_while_body", 2,
{"tokenWHILE__expression","statement_in_while_body"}},\

{"tokenWHILE__expression__statement_in_while_body____iteration_after_two",
2,
{"tokenWHILE__expression","statement_in_while_body____iteration_after_two"
}},\
    {"while_cycle", 2,
{"tokenWHILE__expression__statement_in_while_body____iteration_after_two",
"tokenEND__tokenWHILE"}},\
    {"while_cycle", 2,
{"tokenWHILE__expression__statement_in_while_body","tokenEND__tokenWH
ILE"}},\
    {"while_cycle", 2,
{"tokenWHILE__expression","tokenEND__tokenWHILE"}},\
\
    {"tokenUNTIL__expression", 2, {"tokenUNTIL","expression"}},\
    {"tokenREPEAT__statement____iteration_after_two", 2,
{"tokenREPEAT","statement____iteration_after_two"}},\
    {"tokenREPEAT__statement", 2, {"tokenREPEAT","statement"}},\
    {"repeat_until_cycle", 2,
{"tokenREPEAT__statement____iteration_after_two","tokenUNTIL__expression"
}},\
    {"repeat_until_cycle", 2,
{"tokenREPEAT__statement","tokenUNTIL__expression"}},\
    {"repeat_until_cycle", 2, {"tokenREPEAT","tokenUNTIL__expression"}},\
\
    {"input__first_part", 2, {"tokenGET","tokenGROUPEXPRESSIONBEGIN"}},\
    {"input__second_part", 2, {"tokenIDENT","tokenGROUPEXPRESSIONEND"}},\
    {"input", 2, {"input__first_part","input__second_part"}},\
\
    {"output__first_part", 2,
{"tokenPUT","tokenGROUPEXPRESSIONBEGIN"}},\
    {"output__second_part", 2,
{"expression","tokenGROUPEXPRESSIONEND"}},\

```

```

    {"output", 2, {"output__first_part","output__second_part"}},\
\
    {"statement", 2, {"ident","rl_expression"}},\
    {"statement", 2, {"lr_expression","ident"}},\
    {"statement", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEX
PRESSIONEND","body_for_true__body_for_false"}},\
    {"statement", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEX
PRESSIONEND","body_for_true"}},\
    {"statement", 2,
{"tokenFOR__cycle_counter_init__tokenTO__cycle_counter_last_value","cycle_b
ody__tokenSEMICOLON"}},\
    {"statement", 2,
{"tokenWHILE__expression__statement_in_while_body____iteration_after_two",
"tokenEND__tokenWHILE"}},\
    {"statement", 2,
{"tokenWHILE__expression__statement_in_while_body","tokenEND__tokenWH
ILE"}},\
    {"statement", 2, {"tokenWHILE__expression","tokenEND__tokenWHILE"}},\
    {"statement", 2,
{"tokenREPEAT__statement____iteration_after_two","tokenUNTIL__expression"
}},\
    {"statement", 2, {"tokenREPEAT__statement","tokenUNTIL__expression"}},\
    {"statement", 2, {"tokenREPEAT","tokenUNTIL__expression"}},\
    {"statement", 2, {"ident","tokenCOLON"}},\
    {"statement", 2, {"tokenGOTO","ident"}},\
    {"statement", 2, {"input__first_part","input__second_part"}},\
    {"statement", 2, {"output__first_part","output__second_part"}},\
    {"statement__iteration_after_two", 2,
{"statement","statement__iteration_after_two"}},\
    {"statement__iteration_after_two", 2, {"statement","statement"}},\
\
    {"statement_in_while_body", 2, {"ident","rl_expression"}},\
    {"statement_in_while_body", 2, {"lr_expression","ident"}},\
    {"statement_in_while_body", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEX
PRESSIONEND","body_for_true__body_for_false"}},\
    {"statement_in_while_body", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEX
PRESSIONEND","body_for_true"}},\
    {"statement_in_while_body", 2,
{"tokenFOR__cycle_counter_init__tokenTO__cycle_counter_last_value","cycle_b
ody__tokenSEMICOLON"}},\

```

```

    {"statement_in_while_body", 2,
{"tokenWHILE__expression_statement_in_while_body____iteration_after_two",
"tokenEND__tokenWHILE"}}},\
    {"statement_in_while_body", 2,
{"tokenWHILE__expression_statement_in_while_body","tokenEND__tokenWHILE"}},\
    {"statement_in_while_body", 2,
{"tokenWHILE__expression","tokenEND__tokenWHILE"}}},\
    {"statement_in_while_body", 2,
{"tokenREPEAT__statement____iteration_after_two","tokenUNTIL__expression"}},\
    {"statement_in_while_body", 2,
{"tokenREPEAT__statement","tokenUNTIL__expression"}}},\
    {"statement_in_while_body", 2, {"ident","tokenCOLON"}}},\
    {"statement_in_while_body", 2, {"tokenGOTO","ident"}}},\
    {"statement_in_while_body", 2, {"input__first_part","input__second_part"}}},\
    {"statement_in_while_body", 2,
{"output__first_part","output__second_part"}}},\
    {"statement_in_while_body", 2, {"tokenCONTINUE","tokenWHILE"}}},\
    {"statement_in_while_body", 2, {"tokenEXIT","tokenWHILE"}}},\
    {"statement_in_while_body____iteration_after_two", 2,
{"statement_in_while_body","statement_in_while_body____iteration_after_two"}},\
    {"statement_in_while_body____iteration_after_two", 2,
{"statement_in_while_body","statement_in_while_body"}}},\
\
    {"tokenNAME__program_name", 2, {"tokenNAME","program_name"}}},\
    {"tokenSEMICOLON__tokenBODY", 2,
{"tokenSEMICOLON","tokenBODY"}}},\
    {"tokenDATA__declaration", 2, {"tokenDATA","declaration"}}},\
    {"tokenNAME__program_name__tokenSEMICOLON__tokenBODY", 2,
{"tokenNAME__program_name","tokenSEMICOLON__tokenBODY"}}},\
    {"program____part1", 2,
{"tokenNAME__program_name__tokenSEMICOLON__tokenBODY","tokenDATA__declaration"}}},\
    {"program____part1", 2,
{"tokenNAME__program_name__tokenSEMICOLON__tokenBODY","tokenDATA"}},\
    {"statement__tokenEND", 2, {"statement","tokenEND"}}},\
    {"statement____iteration_after_two__tokenEND", 2,
{"statement____iteration_after_two","tokenEND"}}},\
    {"program____part2", 2,
{"tokenSEMICOLON","statement____iteration_after_two__tokenEND"}}},\

```

```

{"program____part2", 2, {"tokenSEMICOLON","statement__tokenEND"}},\
{"program____part2", 2, {"tokenSEMICOLON","tokenEND"}},\
{"program", 2, {"program____part1","program____part2"}},\
\
{"tokenCOLON", 1, {"":"}},\
{"tokenGOTO", 1, {"GOTO"}},\
{"tokenINTEGER16", 1, {"INTEGER16"}},\
{"tokenCOMMA", 1, {"","}},\
{"tokenNOT", 1, {"NOT"}},\
{"tokenAND", 1, {"AND"}},\
{"tokenOR", 1, {"OR"}},\
{"tokenEQUAL", 1, {"="}},\
{"tokenNOTEQUAL", 1, {"!="}},\
{"tokenLESSOREQUAL", 1, {"<="}},\
{"tokenGREATEROREQUAL", 1, {">="}},\
{"tokenPLUS", 1, {"+"}},\
{"tokenMINUS", 1, {"-"}},\
{"tokenMUL", 1, {"*"}},\
{"tokenDIV", 1, {"DIV"}},\
{"tokenMOD", 1, {"MOD"}},\
{"tokenGROUPEXPRESSIONBEGIN", 1, {"("}},\
{"tokenGROUPEXPRESSIONEND", 1, {")"}},\
{"tokenRLBIND", 1, {"<<"}},\
{"tokenLRBIND", 1, {">>"}},\
{"tokenELSE", 1, {"ELSE"}},\
{"tokenIF", 1, {"IF"}},\
{"tokenDO", 1, {"DO"}},\
{"tokenFOR", 1, {"FOR"}},\
{"tokenTO", 1, {"TO"}},\
{"tokenWHILE", 1, {"WHILE"}},\
{"tokenCONTINUE", 1, {"CONTINUE"}},\
{"tokenEXIT", 1, {"EXIT"}},\
{"tokenREPEAT", 1, {"REPEAT"}},\
{"tokenUNTIL", 1, {"UNTIL"}},\
{"tokenGET", 1, {"GET"}},\
{"tokenPUT", 1, {"PUT"}},\
{"tokenNAME", 1, {"NAME"}},\
{"tokenBODY", 1, {"BODY"}},\
{"tokenDATA", 1, {"DATA"}},\
{"tokenEND", 1, {"END"}},\
{"tokenSEMICOLON", 1, {";"}},\
\
{"value", 1, {"value_terminal"}},\
\
{"ident", 1, {"ident_terminal"}},\

```

```
\
    {"" , 2, {"" , ""}}\
\
},\
176,\
"program"
```

```
////////////////////////////////////
////////////////////////////////////
```

```
//#define DEFAULT_MODE (DEBUG_MODE | LEXICAL_ANALYSIS_MODE)
```

Файл add.h

```
#define _CRT_SECURE_NO_WARNINGS
#define SUCCESS_STATE 0

#define LEXICAL_ANALYZE_MODE 1 // lexicalAnalyze
#define MAKE_LEXEMES_SEQUENSE 2 // ADD MODE
#define SYNTAX_ANALYZE_MODE 4
#define MAKE_AST 8 // ADD MODE
#define SEMANTIX_ANALYZE_MODE 16 // ADD MODE
#define MAKE_PREPARE 32 // ADD MODE
#define MAKE_C 64 // ADD MODE
#define MAKE_ASSEMBLY 128 // ADD MODE
#define MAKE_OBJECT 256 // ADD MODE
#define MAKE_BINARY 512 // ADD MODE
#define RUN_BINARY 1024 // ADD MODE

#define UNDEFINED_MODE 16384

#define INTERACTIVE_MODE 32768

#define FULL_COMPILER_MODE 2048 // ?

#define DEBUG_MODE 4096

//#define DECLENUM(NAME, ...) typedef enum {__VA_ARGS__, size##NAME} NAME;
#define DECLENUM(NAME, ...) enum NAME {__VA_ARGS__, size##NAME};
#define GET_ENUM_SIZE(NAME) size##NAME
#define SET_QUADRUPLE_STR_MACRO_IN_ARRAY(ARRAY, NAME)\
ARRAY[MULTI_TOKEN_ ##NAME][0] = (char*)T_ ##NAME##_0;\
ARRAY[MULTI_TOKEN_ ##NAME][1] = (char*)T_ ##NAME##_1;\
ARRAY[MULTI_TOKEN_ ##NAME][2] = (char*)T_ ##NAME##_2;\
ARRAY[MULTI_TOKEN_ ##NAME][3] = (char*)T_ ##NAME##_3;
```

Файл div.h

```
#define _CRT_SECURE_NO_WARNINGS

#define DIV_CODER(A, B, C, M, R)\
if (A ==* B) C = makeDivCode(B, C, M);

unsigned char* makeDivCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

Файл else.h

```

#define _CRT_SECURE_NO_WARNINGS
#define ELSE_CODER(A, B, C, M, R)\
if (A ==* B) C = makeElseCode(B, C, M);\
if (A ==* B) C = makeSemicolonAfterElseCode(B, C, M);

unsigned char* makeElseCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char
generatorMode);
unsigned char* makeSemicolonAfterElseCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr,
unsigned char generatorMode);

```

Файл equal.h

```

#define _CRT_SECURE_NO_WARNINGS
#define EQUAL_CODER(A, B, C, M, R)\
if (A ==* B) C = makelsEqualCode(B, C, M);

unsigned char* makelsEqualCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned
char generatorMode);

```

Файл for.h

```

#define _CRT_SECURE_NO_WARNINGS
#define FOR_CODER(A, B, C, M, R)\
if (A ==* B) C = makeForCycleCode(B, C, M);\
if (A ==* B) C = makeToOrDowntoCycleCode(B, C, M);\
if (A ==* B) C = makeDoCycleCode(B, C, M);\
if (A ==* B) C = makeSemicolonAfterForCycleCode(B, C, M);

unsigned char* makeForCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned
char generatorMode);
unsigned char* makeToOrDowntoCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr,
unsigned char generatorMode);
unsigned char* makeDoCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned
char generatorMode);
unsigned char* makeSemicolonAfterForCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char*
currBytePtr, unsigned char generatorMode);

```

Файл generator.h

```

#define _CRT_SECURE_NO_WARNINGS
#include "../include/def.h"
#include "../include/config.h"

// TODO: CHANGE BY fRESET() TO END
#define DEBUG_MODE_BY_ASSEMBLY
#define C_CODER_MODE 0x01
#define ASSEMBLY_X86_WIN32_CODER_MODE 0x02
#define MACHINE_X86_WIN32_CODER_MODE 0x04

extern unsigned char generatorMode;

#define CODEGEN_DATA_TYPE int

#define START_DATA_OFFSET 512
#define OUT_DATA_OFFSET (START_DATA_OFFSET + 512)

#define M1 1024
#define M2 1024

//unsigned long long int dataOffsetMinusCodeOffset = 0x00003000;
#define dataOffsetMinusCodeOffset 0x00004000ull

//unsigned long long int codeOffset = 0x000004AF;

```



```

//unsigned long long int baseOperationOffset = codeOffset + 49;// 0x00000031;
#define baseOperationObjectOffset 0x0000018Bull
#define baseOperationOffset 0x000004AFull
#define putProcOffset 0x0000001Bull
#define getProcOffset 0x00000044ull

//unsigned long long int startCodeSize = 64 - 14; // 50 // -1

unsigned char detectMultiToken(struct LexemInfo* lexemInfoTable, enum TokenStructName tokenStructName);
unsigned char createMultiToken(struct LexemInfo** lexemInfoTable, enum TokenStructName tokenStructName);
#define MAX_ACCESSORY_STACK_SIZE 128
extern struct NonContainedLexemInfo lexemInfoTransformationTempStack[MAX_ACCESSORY_STACK_SIZE];
extern unsigned long long int lexemInfoTransformationTempStackSize;
unsigned char* outBytes2Code(unsigned char* currBytePtr, unsigned char* fragmentFirstBytePtr, unsigned long long int bytesCout);

#if 1
unsigned char* getObjectCodeBytePtr(unsigned char* baseBytePtr);
unsigned char* getImageCodeBytePtr(unsigned char* baseBytePtr);
unsigned char* makeCode(struct LexemInfo** lastLexemInfoInTable/*TODO:...*/, unsigned char* currBytePtr, unsigned char generatorMode);
void viewCode(unsigned char* outCodePtr, unsigned long long int outCodePrintSize, unsigned char align);
#endif

unsigned long long int buildTemplateForCodeObject(unsigned char* bytelImage);
unsigned long long int buildTemplateForCodeImage(unsigned char* bytelImage);
void writeBytesToFile(const char* output_file, unsigned char* bytelImage, unsigned long long int imageSize);

```

Файл goto_label.h

```

#define _CRT_SECURE_NO_WARNINGS
#include <string>
#include <map>

extern std::map<std::string, unsigned long long int> labelInfoTable;

#define LABEL_GOTO_LABELE_CODER(A, B, C, M, R)\
if (A ==* B) C = makeLabelCode(B, C, M);\
if (A ==* B) C = makeGotoLabelCode(B, C, M);

unsigned char* makeLabelCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);

unsigned char* makeGotoLabelCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);

```

Файл greater_or_equal.h

```

#define _CRT_SECURE_NO_WARNINGS

#define GREATER_OR_EQUAL_CODER(A, B, C, M, R)\
if (A ==* B) C = makelsGreaterOrEqualCode(B, C, M);

unsigned char* makelsGreaterOrEqualCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);

```

Файл if_then.h

```

#define _CRT_SECURE_NO_WARNINGS
#define IF_THEN_CODER(A, B, C, M, R)\
if (A ==* B) C = makelfCode(B, C, M);\
if (A ==* B) C = makeThenCode(B, C, M);\
if (A ==* B) C = makeSemicolonAfterThenCode(B, C, M);

```

```

unsigned char* makeIfCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char
generatorMode);
unsigned char* makeThenCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char
generatorMode);
unsigned char* makeSemicolonAfterThenCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr,
unsigned char generatorMode);

```

Файл input.h

```

#define _CRT_SECURE_NO_WARNINGS
#define INPUT_CODER(A, B, C, M, R)\
if (A ==* B) C = makeGetCode(B, C, M);

unsigned char* makeGetCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char
generatorMode);

```

Файл less_or_equal.h

```

#define _CRT_SECURE_NO_WARNINGS
#define LESS_OR_EQUAL_CODER(A, B, C, M, R)\
if (A ==* B) C = makeIsLessOrEqualCode(B, C, M);

unsigned char* makeIsLessOrEqualCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr,
unsigned char generatorMode);

```

Файл lexica.h

```

#define _CRT_SECURE_NO_WARNINGS
#define VALUE_SIZE 4

#define MAX_TEXT_SIZE 8192
#define MAX_WORD_COUNT (MAX_TEXT_SIZE / 5)
#define MAX_LEXEM_SIZE 1024
#define MAX_VARIABLES_COUNT 256
#define MAX_KEYWORD_COUNT 64

#define KEYWORD_LEXEME_TYPE 1
#define IDENTIFIER_LEXEME_TYPE 2 // #define LABEL_LEXEME_TYPE 8
#define VALUE_LEXEME_TYPE 4
#define UNEXPEXED_LEXEME_TYPE 127

#ifndef LEXEM_INFO_
#define LEXEM_INFO_
struct NonContainedLexemInfo;
struct LexemInfo {public:
    char lexemStr[MAX_LEXEM_SIZE];
    unsigned long long int lexemId;
    unsigned long long int tokenType;
    unsigned long long int ifvalue;
    unsigned long long int row;
    unsigned long long int col;
    // TODO: ...

    LexemInfo();
    LexemInfo(const char* lexemStr, unsigned long long int lexemId, unsigned long long int tokenType, unsigned
long long int ifvalue, unsigned long long int row, unsigned long long int col);
    LexemInfo(const NonContainedLexemInfo& nonContainedLexemInfo);
};
#endif

```

```

#ifndef NON_CONTAINED_LEXEM_INFO_
#define NON_CONTAINED_LEXEM_INFO_
struct LexemInfo;
struct NonContainedLexemInfo {
    //char lexemStr[MAX_LEXEM_SIZE];
    char* lexemStr;
    unsigned long long int lexemId;
    unsigned long long int tokenType;
    unsigned long long int ifvalue;
    unsigned long long int row;
    unsigned long long int col;
    // TODO: ...

    NonContainedLexemInfo();
    NonContainedLexemInfo(const LexemInfo& lexemInfo);
};
#endif

extern struct LexemInfo lexemesInfoTable[MAX_WORD_COUNT];
extern struct LexemInfo* lastLexemInfoInTable;

extern char identifierIdsTable[MAX_WORD_COUNT][MAX_LEXEM_SIZE];

void printLexemes(struct LexemInfo* lexemInfoTable, char printBadLexeme/* = 0 */);
void printLexemesToFile(struct LexemInfo* lexemInfoTable, char printBadLexeme, const char* filename);
unsigned int getIdentifierId(char(*identifierIdsTable)[MAX_LEXEM_SIZE], char* str);
unsigned int tryToGetIdentifier(struct LexemInfo* lexemInfoInTable, char(*identifierIdsTable)[MAX_LEXEM_SIZE]);
unsigned int tryToGetUnsignedValue(struct LexemInfo* lexemInfoInTable);
int commentRemover(char* text, const char* openStrSpC/* = "/*", const char* closeStrSpC/* = "\n"*/);
void prepareKeyWordIdGetter(char* keywords_, char* keywords_re);
unsigned int getKeyWordId(char* keywords_, char* lexemStr, unsigned int baseId);
char tryToGetKeyWord(struct LexemInfo* lexemInfoInTable);
void setPositions(const char* text, struct LexemInfo* lexemInfoTable);
struct LexemInfo lexicalAnalyze(struct LexemInfo* lexemInfoInPtr, char(*identifierIdsTable)[MAX_LEXEM_SIZE]);
struct LexemInfo tokenize(char* text, struct LexemInfo** lastLexemInfoInTable,
char(*identifierIdsTable)[MAX_LEXEM_SIZE], struct LexemInfo(*lexicalAnalyzeFunctionPtr)(struct LexemInfo*,
char(*)[MAX_LEXEM_SIZE]));

```

Файл mod.h

```

#define _CRT_SECURE_NO_WARNINGS
#define MOD_CODER(A, B, C, M, R)\
if (A ==* B) C = makeModCode(B, C, M);

unsigned char* makeModCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char
generatorMode);

```

Файл mul.h

```

#define _CRT_SECURE_NO_WARNINGS

#define MUL_CODER(A, B, C, M, R)\
if (A ==* B) C = makeMulCode(B, C, M);

unsigned char* makeMulCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char
generatorMode);

```

Файл not.h

```

#define _CRT_SECURE_NO_WARNINGS
#define NOT_CODER(A, B, C, M, R)\
if (A ==* B) C = makeNotCode(B, C, M);

```

```
unsigned char* makeNotCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

Файл not_equal.h

```
#define _CRT_SECURE_NO_WARNINGS
#define NOT_EQUAL_CODER(A, B, C, M, R)\
if (A ==* B) C = makeIsNotEqualCode(B, C, M);
```

```
unsigned char* makeIsNotEqualCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

Файл null_statement.h

```
#define _CRT_SECURE_NO_WARNINGS
#define NON_CONTEXT_NULL_STATEMENT(A, B, C, M, R)\
if (A ==* B) C = makeNullStatementAfterNonContextCode(B, C, M);
```

```
unsigned char* makeNullStatementAfterNonContextCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

Файл operand.h

```
#define _CRT_SECURE_NO_WARNINGS
#define OPERAND_CODER(A, B, C, M, R)\
if (A ==* B) C = makeValueCode(B, C, M);\
if (A ==* B) C = makeIdentifierCode(B, C, M);
```

```
unsigned char* makeValueCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

```
unsigned char* makeIdentifierCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

Файл or.h

```
#define _CRT_SECURE_NO_WARNINGS
#define OR_CODER(A, B, C, M, R)\
if (A ==* B) C = makeOrCode(B, C, M);
```

```
unsigned char* makeOrCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

Файл output.h

```
#define _CRT_SECURE_NO_WARNINGS
#define OUTPUT_CODER(A, B, C, M, R)\
if (A ==* B) C = makePutCode(B, C, M);
```

```
unsigned char* makePutCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode);
```

Файл preparer.h

```
#define _CRT_SECURE_NO_WARNINGS
int precedenceLevel(char* lexemStr);
bool isLeftAssociative(char* lexemStr);
bool isSplittingOperator(char* lexemStr);
void makePrepare4IdentifierOrValue(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo** lastTempLexemInfoInTable);
void makePrepare4Operators(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo** lastTempLexemInfoInTable);
void makePrepare4LeftParenthesis(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo** lastTempLexemInfoInTable);
```

```

void makePrepare4RightParenthesis(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo**
lastTempLexemInfoInTable);
unsigned int makePrepareEnd(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo**
lastTempLexemInfoInTable);
long long int getPrevNonParenthesesIndex(struct LexemInfo* lexemInfoInTable, unsigned long long currIndex);
long long int getEndOfNewPrevExpressionIndex(struct LexemInfo* lexemInfoInTable, unsigned long long currIndex);
unsigned long long int getNextEndOfExpressionIndex(struct LexemInfo* lexemInfoInTable, unsigned long long
prevEndOfExpressionIndex);
void makePrepare(struct LexemInfo* lexemInfoInTable, struct LexemInfo** lastLexemInfoInTable, struct LexemInfo**
lastTempLexemInfoInTable);

```

Файл repeat_until.h

```

#define _CRT_SECURE_NO_WARNINGS
#define REPEAT_UNTIL_CODER(A, B, C, M, R)\
if (A ==* B) C = makeRepeatCycleCode(B, C, M);\
if (A ==* B) C = makeUntileCode(B, C, M);\
if (A ==* B) C = makeNullStatementAfterUntilCycleCode(B, C, M);

unsigned char* makeRepeatCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr,
unsigned char generatorMode);
unsigned char* makeUntileCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char
generatorMode);
unsigned char* makeNullStatementAfterUntilCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char*
currBytePtr, unsigned char generatorMode);

```

Файл rbind.h

```

#define _CRT_SECURE_NO_WARNINGS
#define RLBIND_CODER(A, B, C, M, R)\
if (A ==* B) C = makeRightToLeftBindCode(B, C, M);

unsigned char* makeRightToLeftBindCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr,
unsigned char generatorMode);

```

Файл semantix.h

```

#define _CRT_SECURE_NO_WARNINGS
#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"

#define COLLISION_II_STATE 128
#define COLLISION_LL_STATE 129
#define COLLISION_IL_STATE 130
#define COLLISION_I_STATE 132
#define COLLISION_L_STATE 136
#define COLLISION_IK_STATE 144
#define UNINITIALIZED_I_STATE 160

#define NO_IMPLEMENT_CODE_STATE 256

unsigned long long int getDataSectionLastLexemIndex(LexemInfo* lexemInfoTable, Grammar* grammar);
int checkingInternalCollisionInDeclarations(LexemInfo* lexemInfoTable, Grammar* grammar,
char(*identifierIdsTable)[MAX_LEXEM_SIZE], char** errorMessagesPtrToLastBytePtr);
int checkingVariableInitialization(LexemInfo* lexemInfoTable, Grammar* grammar,
char(*identifierIdsTable)[MAX_LEXEM_SIZE], char** errorMessagesPtrToLastBytePtr);
int checkingCollisionInDeclarationsByKeyWords(char(*identifierIdsTable)[MAX_LEXEM_SIZE], char**
errorMessagesPtrToLastBytePtr);
int semantixAnalyze(LexemInfo* lexemInfoTable, Grammar* grammar, char(*identifierIdsTable)[MAX_LEXEM_SIZE],
char** errorMessagesPtrToLastBytePtr);

```

Файл semicolon.h

```
#define _CRT_SECURE_NO_WARNINGS
#define NON_CONTEXT_SEMICOLON_CODER(A, B, C, M, R)\
/* (1) Ignore phase*/if (A ==* B) C = makeSemicolonAfterNonContextCode(B, C, M);\
/* (2) Ignore phase*/if (A ==* B) C = makeSemicolonIgnoreContextCode(B, C, M);

unsigned char* makeSemicolonAfterNonContextCode(struct LexemInfo** lastLexemInfoInTable, unsigned char*
currBytePtr, unsigned char generatorMode);
unsigned char* makeSemicolonIgnoreContextCode(struct LexemInfo** lastLexemInfoInTable, unsigned char*
currBytePtr, unsigned char generatorMode);
```

Файл sub.h

```
#define _CRT_SECURE_NO_WARNINGS
#define SUB_CODER(A, B, C, M, R)\
if (A ==* B) C = makeSubCode(B, C, M);

unsigned char* makeSubCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char
generatorMode);
```

Файл syntax.h

```
#define _CRT_SECURE_NO_WARNINGS
#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"

#define SYNTAX_ANALYZE_BY_CYK_ALGORITHM 0
#define SYNTAX_ANALYZE_BY_RECURSIVE_DESCENT 1

#define DEFAULT_SYNTAX_ANALYZE_MODE SYNTAX_ANALYZE_BY_CYK_ALGORITHM

using namespace std;

#define MAX_RULES 356

#define MAX_TOKEN_SIZE 128
#define MAX_RTOKEN_COUNT 2 // 3

typedef struct {
    char lhs[MAX_TOKEN_SIZE];
    int rhs_count;
    char rhs[MAX_RTOKEN_COUNT][MAX_TOKEN_SIZE];
} Rule;

typedef struct {
    Rule rules[MAX_RULES];
    int rule_count;
    char start_symbol[MAX_TOKEN_SIZE] ;
} Grammar;

extern Grammar grammar;

#define DEBUG_STATES

bool recursiveDescentParserRuleWithDebug(const char* ruleName, int& lexemIndex, LexemInfo* lexemInfoTable,
Grammar* grammar, int depth, const struct LexemInfo** unexpectedLexemfailedTerminal);
//bool cykAlgorithmImplementation(struct LexemInfo* lexemInfoTable, Grammar* grammar);
int syntaxAnalyze(LexemInfo* lexemInfoTable, Grammar* grammar, char syntaxAnalyzeMode, char* astFileName,
char* errorMessagesPtrToLastBytePtr);
```

Файл while.h

```
#define _CRT_SECURE_NO_WARNINGS
#define WHILE_CODER(A, B, C, M, R)\
if (A ==* B) C = makeWhileCycleCode(B, C, M);\
if (A ==* B) C = makeNullStatementWhileCycleCode(B, C, M);\
if (A ==* B) C = makeContinueWhileCycleCode(B, C, M);\
if (A ==* B) C = makeExitWhileCycleCode(B, C, M);\
if (A ==* B) C = makeEndWhileAfterWhileCycleCode(B, C, M);

unsigned char* makeWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr,
unsigned char generatorMode);
unsigned char* makeNullStatementWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char*
currBytePtr, unsigned char generatorMode);
unsigned char* makeContinueWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr,
unsigned char generatorMode);
unsigned char* makeExitWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr,
unsigned char generatorMode);
unsigned char* makeEndWhileAfterWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char*
currBytePtr, unsigned char generatorMode);
```

Файл add.h

```
#define _CRT_SECURE_NO_WARNINGS
```