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.

Львів-2024

**ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ**

1. Цільова мова транслятора – мова програмування С 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: С/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.

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

**Деталізований опис власної мови програмування:**

* 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-DOWNTO (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 5](#_Toc188297120)

[ВСТУП 7](#_Toc188297121)

[1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ 8](#_Toc188297122)

[2. Формальний опис вхідної мови програмування 10](#_Toc188297123)

[2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура. 10](#_Toc188297124)

[3. Розробка транслятора з вхідної мови програмування 15](#_Toc188297125)

[3.1. Вибір технології програмування. 15](#_Toc188297126)

[3.2. Проектування таблиць транслятора та вибір структур даних. 15](#_Toc188297127)

[3.3. Розробка лексичного аналізатора. 18](#_Toc188297128)

[**3.3.1.** **Розробка алгоритму роботи лексичного аналізатора.** 19](#_Toc188297129)

[**3.3.2.** **Опис програми реалізації лексичного аналізатора.** 25](#_Toc188297130)

[3.4. Розробка синтаксичного та семантичного аналізатора. 27](#_Toc188297131)

[**3.4.1.** **Розробка дерева граматичного розбору.** 28](#_Toc188297132)

[**3.4.2.** **Розробка алгоритму роботи синтаксичного і семантичного аналізатора.** 29](#_Toc188297133)

[**3.4.3.** **Опис програми реалізації синтаксичного та семантичного аналізатора.** 30](#_Toc188297134)

[3.5. Розробка генератора коду. 34](#_Toc188297135)

[**3.5.1.** **Розробка алгоритму роботи генератора коду.** 35](#_Toc188297136)

[**3.5.2.** **Опис програми реалізації генератора коду.** 41](#_Toc188297137)

[4. Налагодження та тестування розробленого транслятора 47](#_Toc188297138)

[4.1. Опис інтерфейсу та інструкції користувачу. 47](#_Toc188297139)

[4.2. Виявлення лексичних і синтаксичних помилок. 48](#_Toc188297140)

[4.3. Перевірка роботи транслятора за допомогою тестових задач. 48](#_Toc188297141)

[Висновки 51](#_Toc188297142)

[Список літературних джерел 52](#_Toc188297143)

[Додатки 53](#_Toc188297144)

**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 транслюючих 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.

1. Формальний опис вхідної мови програмування
   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"

tokenDOWNTO = "DOWNTO"

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"

//

1. Розробка транслятора з вхідної мови програмування
   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.

* 1. Проектування таблиць транслятора та вибір структур даних.

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

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 відрізняти 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.
  1. 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, DOWNTO, 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**: ( , ).
  + 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 і 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
     1. **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 метаінформація, 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.

* 1. Розробка синтаксичного та семантичного аналізатора.

Синтаксичний аналіз – це процес, що визначає, чи належить деяка послідовність лексем граматиці мови програмування. В принципі, для будь-якої граматики можна побудувати синтаксичний аналізатор, але граматики, які використовуються на практиці, мають спеціальну форму. Наприклад, відомо, що для будь-якої контекстно-вільної граматики може бути побудований аналізатор, складність якого не перевищує O(n3) для вхідного рядка довжиною n.

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**

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 CYK 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 CYK 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 CYK**

Таблиця результатів CYK 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 CYK.
* Якщо CYK не успішний, виконується рекурсивний спуск.
* У разі помилки виводиться інформація про невідповідність та позицію помилки у вхідному коді.
  + 1. **Розробка дерева граматичного розбору.**

<програма>

│

├── '#PROGRAM'

├── <назва програми>

│ └── <ідентифікатор>

├── 'START'

├── <оголошення змінних>

│ ├── ‘VARIABLE’

│ │ ├ <тип даних>

│ │ └── 'INTEGER16\_t'

│ └── <список змінних>

│ ├── <ідентифікатор>

│ └── { ',' <ідентифікатор> }

├── <тіло програми>

│ ├── <оператор>

│ └── { <оператор> }

│ │

│ ├── <присвоєння>

│ │ ├── <ідентифікатор>

│ │ ├── '<-'

│ │ └── <арифметичний вираз>

│ │ ├── <доданок>

│ │ │ ├── <множник>

│ │ │ │ └── (<ідентифікатор> | <число> | '(' <арифметичний вираз> ')')

│ │ │ └── { ('MUL' | 'DIV' | 'MOD') <доданок> }

│ │ └── { ('ADD' | 'SUB') <доданок> }

│ │

│ ├── <ввід>

│ │ ├── 'SCAN'

│ │ └── <ідентифікатор>

│ │

│ ├── <вивід>

│ │ ├── 'PRINT'

│ │ └── <ідентифікатор>

│ │

│ ├── <умовний оператор>

│ │ ├── 'IF'

│ │ ├── <логічний вираз>

│ │ │ ├── <вираз І>

│ │ │ │ ├── <порівняння>

│ │ │ │ │ └── (<операція порівняння> | '!!' '(' <логічний вираз> ')' | '(' <логічний вираз> ')')

│ │ │ │ │ ├── <арифметичний вираз>

│ │ │ │ │ ├── <оператор порівняння> ('==' | '!=' | '!>' | '!<')

│ │ │ │ │ └── <арифметичний вираз>

│ │ │ │ └── { 'OR' <порівняння> }

│ │ │ └── { 'AND' <вираз І> }

│ │ ├── <тіло> ‘;’

│ │ └── ['ELSE' <тіло> ‘;’]

│ │

│ ├── <оператор циклу>

│ │ ├── 'WHILE'

│ │ ├── <логічний вираз>

│ │ ├── <тіло циклу>

│ │ │ ├── <оператор>

│ │ │ └── { <оператор> }

│ │ ├── ['EXIT WHILE'] (опціонально)

│ │ ├── ['CONTINUE WHILE'] (опціонально)

│ │ └── 'END WHILE'

└── 'STOP'

* + 1. **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.
  + 1. **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 (CYK) 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 CYK 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 CYK 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 CYK**

Таблиця результатів CYK 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 CYK.
* Якщо CYK 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\_II\_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
  1. Розробка генератора коду.

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

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

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

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

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

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

* + 1. **Розробка алгоритму роботи генератора коду.**

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

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

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

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

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

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

* + 1. Розробка алгоритму роботи генератора коду.

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

Програма має вигляд:

Program

/ \

var statement

Оголошення змінних:

var

/ \

Id var

/ \

Id null

Тіло програми:

Statement

/ \

Statement Оператор

/ \

statement Оператор

Оператор вводу:

Input

/ \

Id null

Оператор виводу:

Output

/ \

Id null

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

Output

/ \

Add null

/ \

Id num

Умовний оператор (IF() оператор;):

If

/ \

Умова оператор

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

If

/ \

Умова else

/ \

Оператор1 оператор2

Оператор безумовного переходу:

Goto

/ \

Id null

Оператор циклу for:

For

/ \

(to | downto) оператор

/ \

Оп. Прис. ариф. вир.

Оператор циклу while:

While

/ \

Умова statement

/ \

Statement оператор

/ \

Оператор оператор

Оператор циклу repeat:

Repeat

/ \

Statement умова

/ \

Оператор оператор

Оператор присвоєння:

==>

/ \

Id арифметичний вираз

Арифметичний вираз:

(+ або -)

/ \

Id id

Доданок:

(\*, DIV або MOD)  
/ \  
множник множник

Множник:

фактор  
/ \

id або number або (арифм. вираз) null

Складений оператор:

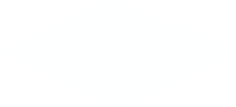
compount

/ \

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.



початок

т п вуз а o

id

program

var

while

опрацювання

вуз а program

опрацювання

вуз а var

опрацювання

вуз а id

...

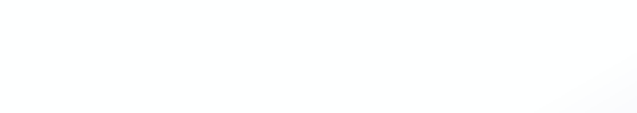
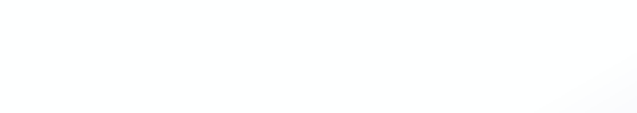
опрацювання

вуз а while

кінець

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

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



початок опрацювання вуз а program

друк у фай :

include <stdio.h>

int main()

{

кінець опрацювання вуз а program

друк у фай :

return 0;

}

Рекурс вн й в к к д я правого нащадка

Рекурс вн й в к к д я івого нащадка

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

* + 1. **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.

1. **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 багатокомпонентні токени, такі як IF ... THEN, FOR ... TO ..., WHILE, тощо.

1. **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).

1. **Маніпуляція з токенами**

* **detectMultiToken**:
  + Перевіряє, чи відповідає поточна лексема багатокомпонентному токену з таблиці tokenStruct.
* **createMultiToken**:
  + Створює багатокомпонентний токен і зберігає його у структурі LexemInfo.

1. **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.

1. **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.

1. **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 зустріне лексему з порожнім рядком (lexemStr[0] == '\0').

1. **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

1. Налагодження та тестування розробленого транслятора

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

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

* 1. Опис інтерфейсу та інструкції користувачу.

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

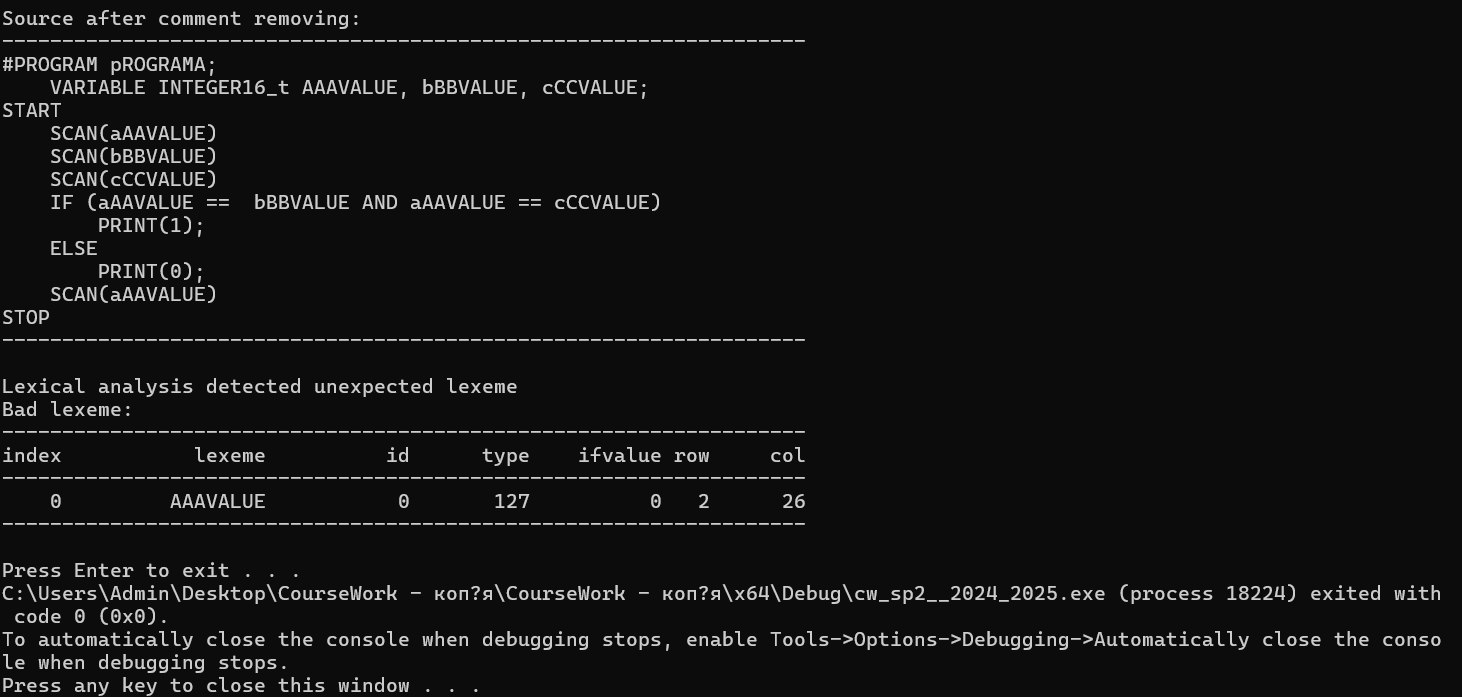
Зображення, що містить текст, знімок екрана, Шрифт, чорний

Автоматично згенерований опис

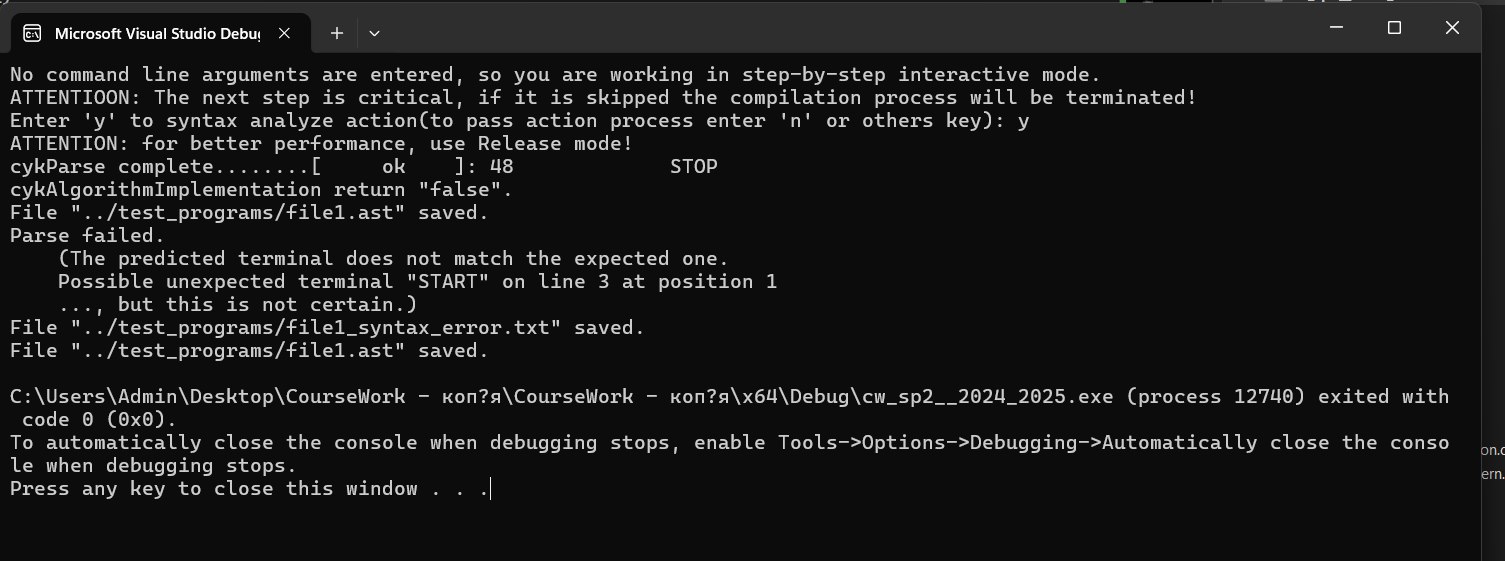
*Рис. 4.1.Результати роботи розробленого транслятора.*

* 1. Виявлення лексичних і синтаксичних помилок.

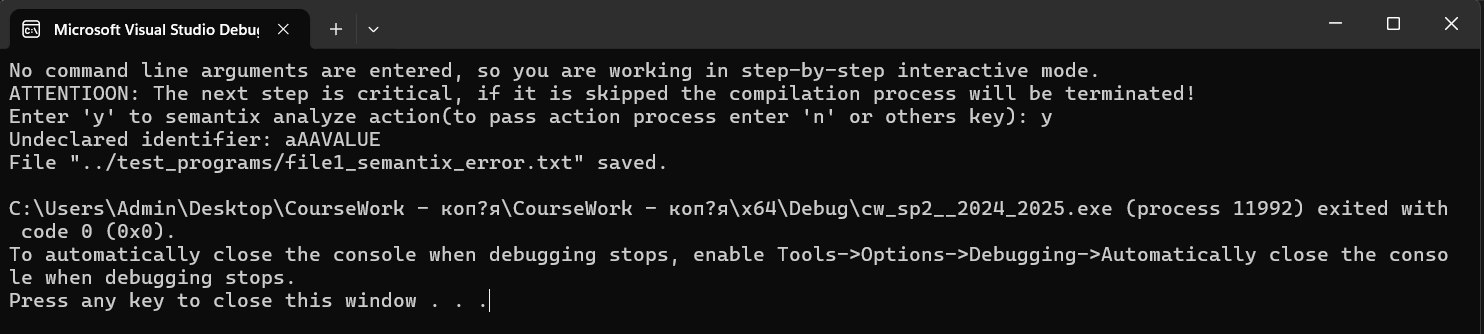
Помилки у вхідній програмі виявляються на стадіяй лексичного, синтаксичного та семантичного аналізів.



*Рис. 4.2.Помилка при лексичному аналізі.*



*Рис. 4.3.Помилка при синтаксичному аналізі.*



*Рис. 4.4.Помилка при семантичному аналізі.*

* 1. Перевірка роботи транслятора за допомогою тестових задач.

**Тестова програма «*Лінійний алгоритм*»**

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

Х = (А - В) \* 10 + (А + В) / 10

1. Вивести значення Х на екран.

Напишемо програму на вхідній мові програмування:

|  |
| --- |
| #PROGRAM pROGRAMA;  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:

Зображення, що містить текст, знімок екрана, Шрифт

Автоматично згенерований опис

*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:

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

Використання простого умовного оператора:

1. Вивести на екран число 1, якщо усі числа однакові інакше вивести 0.

Напишемо програму на вхідній мові програмування:

|  |
| --- |
| #PROGRAM pROGRAMA;  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 |
|  |

**Зображення, що містить текст, знімок екрана, Шрифт

Автоматично згенерований опис**

*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]

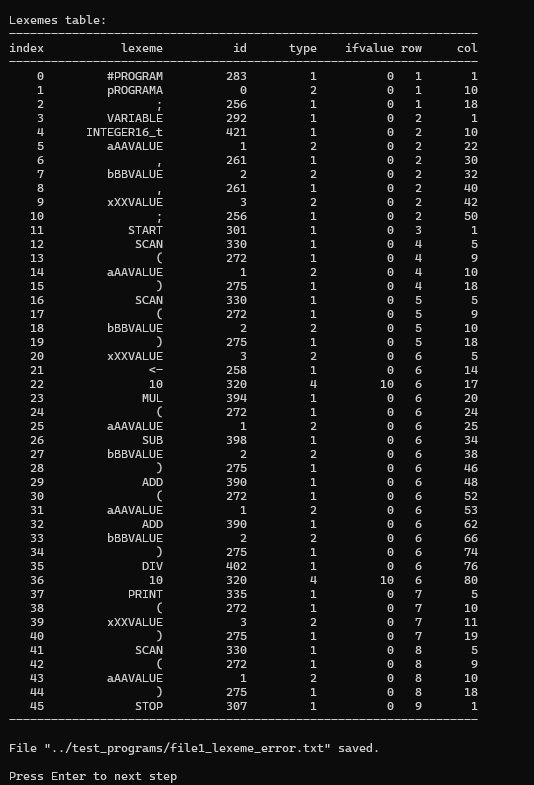
: навч. посіб. для студ. спеціальності 123 – «Комп’ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. – Київ: КПІ ім. Ігоря Сікорського, 2021. – 108 с.

1. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. – Харків: НТУ «ХПІ», 2021. – 133 с.
2. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії формальних мов: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 c.
3. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 c.
4. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. – 1038 c.
5. Системне програмування (курсовий проект) [Електронний ресурс] – Режим доступу до ресурсу: https://vns.lpnu.ua/course/view.php?id=11685.
6. MIT OpenCourseWare. Computer Language Engineering [Електронний ресурс] – Режим доступу до ресурсу: https://ocw.mit.edu/courses/6-035-computer- language-engineering-spring-2010.

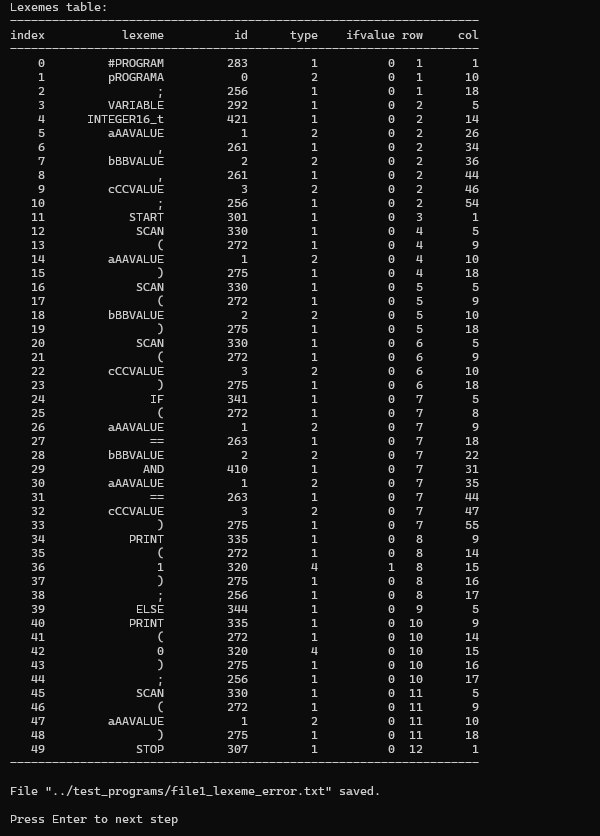
Додатки

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

Таблиця лексем тестової програми “Лінійний алгоритм”



Таблиця лексем тестової програми “Алгоритм з розгалуженням”



**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**

**| | | | | | |--tokenGET**

**| | | | | | | |--"SCAN"**

**| | | | | | |--tokenGROUPEXPRESSIONBEGIN**

**| | | | | | | |--"("**

**| | | | | |--input\_\_second\_part**

**| | | | | | |--ident**

**| | | | | | | |--"aAAVALUE"**

**| | | | | | |--tokenGROUPEXPRESSIONEND**

**| | | | | | | |--")"**

**| | | | |--statement\_\_\_\_iteration\_after\_two**

**| | | | | |--statement**

**| | | | | | |--input\_\_first\_part**

**| | | | | | | |--tokenGET**

**| | | | | | | | |--"SCAN"**

**| | | | | | | |--tokenGROUPEXPRESSIONBEGIN**

**| | | | | | | | |--"("**

**| | | | | | |--input\_\_second\_part**

**| | | | | | | |--ident**

**| | | | | | | | |--"bBBVALUE"**

**| | | | | | | |--tokenGROUPEXPRESSIONEND**

**| | | | | | | | |--")"**

**| | | | | |--statement\_\_\_\_iteration\_after\_two**

**| | | | | | |--statement**

**| | | | | | | |--ident**

**| | | | | | | | |--"xXXVALUE"**

**| | | | | | | |--rl\_expression**

**| | | | | | | | |--tokenRLBIND**

**| | | | | | | | | |--"<-"**

**| | | | | | | | |--expression**

**| | | | | | | | | |--left\_expression**

**| | | | | | | | | | |--"10"**

**| | | | | | | | | |--binary\_action\_\_\_\_iteration\_after\_two**

**| | | | | | | | | | |--binary\_action**

**| | | | | | | | | | | |--binary\_operator**

**| | | | | | | | | | | | |--"MUL"**

**| | | | | | | | | | | |--expression**

**| | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN\_\_expression**

**| | | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN**

**| | | | | | | | | | | | | | |--"("**

**| | | | | | | | | | | | | |--expression**

**| | | | | | | | | | | | | | |--left\_expression**

**| | | | | | | | | | | | | | | |--"aAAVALUE"**

**| | | | | | | | | | | | | | |--binary\_action**

**| | | | | | | | | | | | | | | |--binary\_operator**

**| | | | | | | | | | | | | | | | |--"SUB"**

**| | | | | | | | | | | | | | | |--expression**

**| | | | | | | | | | | | | | | | |--"bBBVALUE"**

**| | | | | | | | | | | | |--tokenGROUPEXPRESSIONEND**

**| | | | | | | | | | | | | |--")"**

**| | | | | | | | | | |--binary\_action\_\_\_\_iteration\_after\_two**

**| | | | | | | | | | | |--binary\_action**

**| | | | | | | | | | | | |--binary\_operator**

**| | | | | | | | | | | | | |--"ADD"**

**| | | | | | | | | | | | |--expression**

**| | | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN\_\_expression**

**| | | | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN**

**| | | | | | | | | | | | | | | |--"("**

**| | | | | | | | | | | | | | |--expression**

**| | | | | | | | | | | | | | | |--left\_expression**

**| | | | | | | | | | | | | | | | |--"aAAVALUE"**

**| | | | | | | | | | | | | | | |--binary\_action**

**| | | | | | | | | | | | | | | | |--binary\_operator**

**| | | | | | | | | | | | | | | | | |--"ADD"**

**| | | | | | | | | | | | | | | | |--expression**

**| | | | | | | | | | | | | | | | | |--"bBBVALUE"**

**| | | | | | | | | | | | | |--tokenGROUPEXPRESSIONEND**

**| | | | | | | | | | | | | | |--")"**

**| | | | | | | | | | | |--binary\_action**

**| | | | | | | | | | | | |--binary\_operator**

**| | | | | | | | | | | | | |--"DIV"**

**| | | | | | | | | | | | |--expression**

**| | | | | | | | | | | | | |--"10"**

**| | | | | | |--statement\_\_\_\_iteration\_after\_two**

**| | | | | | | |--statement**

**| | | | | | | | |--output\_\_first\_part**

**| | | | | | | | | |--tokenPUT**

**| | | | | | | | | | |--"PRINT"**

**| | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN**

**| | | | | | | | | | |--"("**

**| | | | | | | | |--output\_\_second\_part**

**| | | | | | | | | |--expression**

**| | | | | | | | | | |--"xXXVALUE"**

**| | | | | | | | | |--tokenGROUPEXPRESSIONEND**

**| | | | | | | | | | |--")"**

**| | | | | | | |--statement**

**| | | | | | | | |--input\_\_first\_part**

**| | | | | | | | | |--tokenGET**

**| | | | | | | | | | |--"SCAN"**

**| | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN**

**| | | | | | | | | | |--"("**

**| | | | | | | | |--input\_\_second\_part**

**| | | | | | | | | |--ident**

**| | | | | | | | | | |--"aAAVALUE"**

**| | | | | | | | | |--tokenGROUPEXPRESSIONEND**

**| | | | | | | | | | |--")"**

**| | | |--tokenEND**

**| | | | |--"STOP"**

**cykAlgorithmImplementation return "true".**

**File "../test\_programs/file1\_syntax\_error.txt" saved.**

**Press Enter to next step**

**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\_LENGH 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\_ANALISIS\_MODE 1

//#define SEMANTIC\_ANALISIS\_MODE 2

//#define FULL\_COMPILER\_MODE 4

//#define DEBUG\_MODE 512

//#define DEFAULT\_MODE (DEBUG\_MODE | LEXICAL\_ANALISIS\_MODE)

//#define DEFAULT\_MODE (DEBUG\_MODE | LEXICAL\_ANALISIS\_MODE | SYNTAX\_ANALISIS\_MODE | SEMANTIC\_ANALISIS\_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|END|SCAN|PRINT|IF|ELSE|FOR|TO|DOWNTO|DO|WHILE|REPEAT|UNTIL|GOTO|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","tokenSEMICOLON"}},\

{"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","statement\_\_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\_DOWNTO\_0 "DOWNTO"

#define T\_DOWNTO\_1 ""

#define T\_DOWNTO\_2 ""

#define T\_DOWNTO\_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 intitTokenStruct(){\

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\_STRUCT\_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","tokenGROUPEXPRESSIONEND"}},\

{"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","tokenGROUPEXPRESSIONEND"}},\

{"expression", 2, {"unary\_operator","expression"}},\

{"expression", 1, {"ident\_terminal"}},\

{"expression", 1, {"value\_terminal"}},\

\

{"tokenGROUPEXPRESSIONBEGIN\_\_expression", 2, {"tokenGROUPEXPRESSIONBEGIN","expression"}},\

{"group\_expression", 2, {"tokenGROUPEXPRESSIONBEGIN\_\_expression","tokenGROUPEXPRESSIONEND"}},\

\

{"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\_body\_\_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\_\_tokenWHILE"}},\

{"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\_\_tokenGROUPEXPRESSIONEND","body\_for\_true\_\_body\_for\_false"}},\

{"statement", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND","body\_for\_true"}},\

{"statement", 2, {"tokenFOR\_\_cycle\_counter\_init\_\_tokenTO\_\_cycle\_counter\_last\_value","cycle\_body\_\_tokenSEMICOLON"}},\

{"statement", 2, {"tokenWHILE\_\_expression\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenEND\_\_tokenWHILE"}},\

{"statement", 2, {"tokenWHILE\_\_expression\_\_statement\_in\_while\_body","tokenEND\_\_tokenWHILE"}},\

{"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\_\_tokenGROUPEXPRESSIONEND","body\_for\_true\_\_body\_for\_false"}},\

{ "statement\_in\_while\_body", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND","body\_for\_true"}},\

{ "statement\_in\_while\_body", 2, {"tokenFOR\_\_cycle\_counter\_init\_\_tokenTO\_\_cycle\_counter\_last\_value","cycle\_body\_\_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","tokenGROUPEXPRESSIONEND"}},\

{"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","tokenGROUPEXPRESSIONEND"}},\

{"expression", 2, {"unary\_operator","expression"}},\

{"expression", 1, {"ident\_terminal"}},\

{"expression", 1, {"value\_terminal"}},\

\

{"tokenGROUPEXPRESSIONBEGIN\_\_expression", 2, {"tokenGROUPEXPRESSIONBEGIN","expression"}},\

{"group\_expression", 2, {"tokenGROUPEXPRESSIONBEGIN\_\_expression","tokenGROUPEXPRESSIONEND"}},\

\

{"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"}},\

{"forto\_cycle", 2, {"tokenFOR\_\_cycle\_counter\_init\_\_tokenTO\_\_cycle\_counter\_last\_value","cycle\_body\_\_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\_\_tokenWHILE"}},\

{"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\_\_tokenGROUPEXPRESSIONEND","body\_for\_true\_\_body\_for\_false"}},\

{"statement", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND","body\_for\_true"}},\

{"statement", 2, {"tokenFOR\_\_cycle\_counter\_init\_\_tokenTO\_\_cycle\_counter\_last\_value","cycle\_body\_\_tokenSEMICOLON"}},\

{"statement", 2, {"tokenWHILE\_\_expression\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenEND\_\_tokenWHILE"}},\

{"statement", 2, {"tokenWHILE\_\_expression\_\_statement\_in\_while\_body","tokenEND\_\_tokenWHILE"}},\

{"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\_\_tokenGROUPEXPRESSIONEND","body\_for\_true\_\_body\_for\_false"}},\

{"statement\_in\_while\_body", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND","body\_for\_true"}},\

{"statement\_in\_while\_body", 2, {"tokenFOR\_\_cycle\_counter\_init\_\_tokenTO\_\_cycle\_counter\_last\_value","cycle\_body\_\_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"}},\

\

{"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\_ANALISIS\_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 = makeIsEqualCode(B, C, M);  unsigned char\* makeIsEqualCode(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\* byteImage);  unsigned long long int buildTemplateForCodeImage(unsigned char\* byteImage);  void writeBytesToFile(const char\* output\_file, unsigned char\* byteImage, 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 = makeIsGreaterOrEqualCode(B, C, M);  unsigned char\* makeIsGreaterOrEqualCode(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 = makeIfCode(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 UNEXPEXTED\_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 makePrepareEnder(struct LexemInfo\*\* lastLexemInfoInTable, struct LexemInfo\*\* lastTempLexemInfoInTable);  long long int getPrevNonParenthesesIndex(struct LexemInfo\* lexemInfoInTable, unsigned long long currIndex);  long long int getEndOfNewPrevExpressioIndex(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 rlbind.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 syntaxlAnalyzeMode, 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 |