

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"

1. Індивідуальне завдання

"РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ"

Виконав студент групи КІ-307:

Патрило Ю.А.

Перевірив:

Козак Н.Б.

Львів-2024

## ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ

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

### Деталізація завдання на проектування:

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

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

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

## Деталізований опис власної мови програмування:

- Тип даних: INTEGER
- Блок тіла програми: MAIMPROGRAM DATA...; START END
- Оператор вводу: GET ()
- Оператор виводу: PUT ()
- Оператори: IF ELSE (C)  
GOTO (C)  
FOR-TO-DO (Паскаль)  
FOR-DOWNTO-DO (Паскаль)  
WHILE (Бейсік)  
REPEAT-UNTIL (Паскаль)
- Регістр ключових слів: Up
- Регістр ідентифікаторів: Up6 перший символ \_
- Операції арифметичні: +, -, \*, DIV, MOD
- Операції порівняння: ==, !=, GT, LT
- Операції логічні: !!, AND, OR
- Коментар: %%... %%
- Ідентифікатори змінних, числові константи
- Оператор присвоєння: <-

## Анотація

Цей курсовий проект приводить до розробки транслятора, який здатен конвертувати вхідну мову, визначену відповідно до варіанту, у мову асемблера. Процес трансляції включає в себе лексичний аналіз, синтаксичний аналіз та генерацію коду.

Лексичний аналіз розбиває вхідну послідовність символів на лексеми, які записуються у відповідну таблицю лексем. Кожній лексемі присвоюється числове значення для полегшення порівнянь, а також зберігається додаткова інформація, така як номер рядка, значення (якщо тип лексеми є числом) та інші деталі.

Синтаксичний аналіз: використовується висхідний метод аналізу без повернення. Призначений для побудови дерева розбору, послідовно рухаючись від листків вгору до кореня дерева розбору.

Генерація коду включає повторне прочитання таблиці лексем та створення відповідного асемблерного коду для кожного блоку лексем. Отриманий код записується у результуючий файл, готовий для виконання.

Отриманий після трансляції код можна скомпілювати за допомогою відповідних програм (наприклад, LINK, ML і т. д.).

## Зміст

<b>ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ</b> .....	2
Анотація .....	5
Вступ .....	7
1. Огляд методів та способів проектування трансляторів .....	8
2. Формальний опис вхідної мови програмування .....	11
2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура .....	11
Опис вхідної мови програмування у термінах розширеної форми Бекуса- Наура: .....	11
2.2. Опис термінальних символів та ключових слів .....	18
3. Розробка транслятора вхідної мови програмування.....	19
3.1. Вибір технології програмування .....	19
3.2. Проектування таблиць транслятора та вибір структур даних. ....	20
3.3. Розробка лексичного аналізатора.....	22
3.3.1. Розробка блок-схеми алгоритму роботи лексичного аналізатора .....	23
3.3.2. Опис програми реалізації лексичного аналізатора .....	24
3.4. Розробка синтаксичного та семантичного аналізатора.....	25
3.4.1. Розробка дерева граматичного розбору.....	26
3.4.2. Розробка алгоритму роботи синтаксичного і семантичного аналізатора .....	29
3.4.3. Опис програми реалізації синтаксичного та семантичного аналізатора .....	32
3.5. Розробка генератора коду .....	33
3.5.1. Розробка алгоритму роботи генератора коду.....	34
3.5.2. Опис програми реалізації генератора коду.....	35
4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА .....	37
4.1. Опис інтерфейсу та інструкція користувачеві .....	38
4.2. Виявлення лексичних та синтаксичних помилок.....	39
4.3. Перевірка роботи транслятора за допомогою тестових задач. ....	40
Висновки .....	43
Список використаної літератури.....	44
Додатки .....	45

## Вступ

Термін "транслятор" визначає програму, яка виконує переклад (трансляцію) початкової програми, написаної на вхідній мові, у еквівалентну їй об'єктну програму. У випадку, коли мова високого рівня є вхідною, а мова асемблера або машинна – вихідною, такий транслятор отримує назву компілятора.

Транслятори можуть бути розділені на два основних типи: компілятори та інтерпретатори. Процес компіляції включає дві основні фази: аналіз та синтез. Під час аналізу вхідну програму розбивають на окремі елементи (лексеми), перевіряють її відповідність граматичним правилам і створюють проміжне представлення програми. На етапі синтезу з проміжного представлення формується програма в машинних кодах, яку називають об'єктною програмою. Останню можна виконати на комп'ютері без додаткової трансляції.

У відмінну від компіляторів, інтерпретатор не створює нову програму; він лише виконує – інтерпретує – кожну інструкцію вхідної мови програмування. Подібно компілятору, інтерпретатор аналізує вхідну програму, створює проміжне представлення, але не формує об'єктну програму, а негайно виконує команди, передбачені вхідною програмою.

Компілятор виконує переклад програми з однієї мови програмування в іншу. На вхід компілятора надходить ланцюг символів, який представляє вхідну програму на певній мові програмування. На виході компілятора (об'єктна програма) також представляє собою ланцюг символів, що вже відповідає іншій мові програмування, наприклад, машинній мові конкретного комп'ютера. При цьому сам компілятор може бути написаний на третій мові.

# 1. Огляд методів та способів проектування трансляторів

Термін "транслятор" визначає обслуговуючу програму, що проводить трансляцію вихідної програми, представленої на вхідній мові програмування, у робочу програму, яка відображена на об'єктній мові. Наведене визначення застосовне до різноманітних трансляторів програм. Однак кожна з таких програм може виявляти свої особливості в організації процесу трансляції. В сучасному контексті транслятори поділяються на три основні групи: асемблери, компілятори та інтерпретатори.

Асемблер - це системна обслуговуюча програма, яка перетворює символічні конструкції в команди машинної мови. Типовою особливістю асемблерів є дослівна трансляція однієї символічної команди в одну машинну.

Компілятор - обслуговуюча програма, яка виконує трансляцію програми, написаної мовою оригіналу програмування, в машинну мову. Схоже до асемблера, компілятор виконує перетворення програми з однієї мови в іншу, найчастіше - у мову конкретного комп'ютера.

Інтерпретатор - це програма чи пристрій, що виконує пооператорну трансляцію та виконання вихідної програми. Відмінно від компілятора, інтерпретатор не створює на виході програму на машинній мові. Розпізнавши команду вихідної мови, він негайно її виконує, забезпечуючи більшу гнучкість у процесі розробки та налагодження програм.

Процес трансляції включає фази лексичного аналізу, синтаксичного та семантичного аналізу, оптимізації коду та генерації коду. Лексичний аналіз розбиває вхідну програму на лексеми, що представляють слова відповідно до визначень мови. Синтаксичний аналіз визначає структуру програми, створюючи синтаксичне дерево. Семантичний аналіз виявляє залежності між частинами програми, недосяжні контекстно-вільним синтаксисом. Оптимізація коду та генерація коду спрямовані на оптимізацію та створення машинно-залежного коду відповідно.

Зазначені фази можуть об'єднуватися або відсутні у трансляторах в залежності від їхньої реалізації. Наприклад, у простих однопрохідних трансляторах може відсутня фаза генерації проміжного представлення та оптимізації, а інші фази можуть об'єднуватися.

Під час процесу виділення лексем лексичний аналізатор може виконувати дві основні функції: автоматично побудову таблиць об'єктів (таких як ідентифікатори, рядки, числа і т. д.) і видачу значень для кожної лексеми при кожному новому зверненні до нього. У цьому контексті таблиці об'єктів формуються в подальших етапах, наприклад, під час синтаксичного аналізу.



На етапі лексичного аналізу виявляються деякі прості помилки, такі як неприпустимі символи або невірний формат чисел та ідентифікаторів.

Основним завданням синтаксичного аналізу є розбір структури програми. Зазвичай під структурою розуміється дерево, яке відповідає розбору в контекстно-вільній граматиці мови програмування. У сучасній практиці найчастіше використовуються методи аналізу, такі як LL (1) або LR (1) та їхні варіанти (рекурсивний спуск для LL (1) або LR (1), LR (0), SLR (1), LALR (1) та інші для LR (1)). Рекурсивний спуск застосовується частіше при ручному програмуванні синтаксичного аналізатора, тоді як LR (1) використовується при автоматичній генерації синтаксичних аналізаторів.

Результатом синтаксичного аналізу є синтаксичне дерево з посиланнями на таблиці об'єктів. Під час синтаксичного аналізу також виявляються помилки, пов'язані зі структурою програми.

На етапі контекстного аналізу виявляються взаємозалежності між різними частинами програми, які не можуть бути адекватно описані за допомогою контекстно-вільної граматики. Ці взаємозалежності, зокрема, включають аналіз типів об'єктів, областей видимості, відповідності параметрів, міток та інших аспектів "опис-використання". У ході контекстного аналізу таблиці об'єктів доповнюються інформацією, пов'язаною з описами (властивостями) об'єктів.

В основі контекстного аналізу лежить апарат атрибутних граматики. Результатом цього аналізу є створення атрибутованого дерева програми, де інформація про об'єкти може бути розсіяна в самому дереві чи сконцентрована в окремих таблицях об'єктів. Під час контекстного аналізу також можуть бути виявлені помилки, пов'язані з неправильним використанням об'єктів.

Після завершення контекстного аналізу програма може бути перетворена во внутрішнє представлення. Це здійснюється з метою оптимізації та/або для полегшення генерації коду. Крім того, перетворення програми у внутрішнє представлення може бути використано для створення переносимого компілятора. У цьому випадку, тільки остання фаза (генерація коду) є залежною від конкретної архітектури. В якості внутрішнього представлення може використовуватися префіксний або постфіксний запис, орієнтований граф, трійки, четвірки та інші формати.

Фаза оптимізації транслятора може включати декілька етапів, які спрямовані на покращення якості та ефективності згенерованого коду. Ці оптимізації часто розподіляються за двома головними критеріями: машинно-залежні та машинно-незалежні, а також локальні та глобальні.

Машинно-залежні оптимізації, як правило, проводяться на етапі генерації коду, і вони орієнтовані на конкретну архітектуру машини. Ці оптимізації можуть включати розподіл регістрів, вибір довгих або коротких переходів та оптимізацію вартості команд для конкретних послідовностей команд.

Глобальна оптимізація спрямована на поліпшення ефективності всієї програми і базується на глобальному потоковому аналізі, який виконується на графі програми. Цей аналіз враховує властивості програми, такі як межпроцедурний аналіз, міжмодульний аналіз та аналіз галузей життя змінних.

Фінальна фаза трансляції - генерація коду, результатом якої є або асемблерний модуль, або об'єктний (або завантажувальний) модуль. На цьому етапі можуть застосовуватися деякі локальні оптимізації для полегшення генерації вартісного та ефективного коду.

Важливо відзначити, що фази транслятора можуть бути відсутніми або об'єднаними в залежності від конкретної реалізації. В простіших випадках, таких як у випадку однопроходових трансляторів, може відсутній окремий етап генерації проміжного представлення та оптимізації, а інші фази можуть бути об'єднані в одну, при цьому не створюється явно побудованого синтаксичного дерева.

## 2. Формальний опис вхідної мови програмування

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

Однією з перших задач, що виникають при побудові компілятора, є визначення вхідної мови програмування. Для цього використовують різні способи формального опису, серед яких я застосував розширену нотацію Бекуса-Наура (Backus/Naur Form - BNF). БНФ визначає обмежену кількість символів (нетерміналів) і встановлює правила їх заміни на певні послідовності букв (терміналів) і символів. Процес побудови ланцюжка з букв можна описати поетапно: спочатку є один символ (зазвичай його позначають у кутових дужках, і його назва не має значення). Потім цей символ замінюється на послідовність букв і символів згідно з одним із заданих правил. Далі процес повторюється: на кожному етапі один із символів замінюється на нову послідовність за відповідним правилом. У підсумку утворюється ланцюжок, що складається лише з букв і не містить символів. Це означає, що такий ланцюжок може бути отриманий з початкового символу.

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

```
labeled_point = label , ":"  
goto_label = tokenGOTO, label, ";"  
program_name = ident, ";"  
value_type = tokenINTEGER16  
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 |  
tokenLESSOREQUAL | tokenGREATEROREQUAL | tokenPLUS | tokenMINUS |  
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 , ";"

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 = tokenGET , tokenGROUPEXPRESSIONBEGIN , ident ,
tokenGROUPEXPRESSIONEND

output = tokenPUT , tokenGROUPEXPRESSIONBEGIN , expression ,
tokenGROUPEXPRESSIONEND

statement = bind_right_to_left | bind_left_to_right | cond_block | for_to_cycle |
while_cycle | repeat_until_cycle | labeled_point | goto_label | input | output

program = tokenNAME , program_name , tokenSEMICOLON , tokenBODY ,
tokenDATA , [declaration] , tokenSEMICOLON , {statement} , tokenEND

//

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

// -- pashalka

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 = tokenUNDERSCORE , letter_in_upper_case , letter_in_upper_case ,
letter_in_upper_case , letter_in_upper_case , letter_in_upper_case

label = letter_in_upper_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 = "INTEGER"
tokenCOMMA = ","
tokenNOT = "!!"
tokenAND = "AND"
tokenOR = "OR"
tokenEQUAL = "=="
tokenNOTEQUAL = "!="
tokenLESSOREQUAL = "LT"
tokenGREATEROREQUAL = "GT"
tokenPLUS = "+"
tokenMINUS = "-"
tokenMUL = "*"
tokenDIV = "DIV"
tokenMOD = "%"
tokenGROUPEXPRESSIONBEGIN = "("

tokenGROUPEXPRESSIONEND = ")"
tokenRLBIND = "<-"
```

```
tokenLRBIND = ","
tokenELSE = "ELSE"
tokenIF = "IF"
tokenDO = "DO"
tokenFOR = "FOR"
tokenTO = "TO"
tokenWHILE = "WHILE"
tokenCONTINUE = "CONTINUE"
tokenEXIT = "EXIT"
tokenREPEAT = "REPEAT"
tokenUNTIL = "UNTIL"
tokenGET = "GET"
tokenPUT = "PUT"
tokenNAME = "MAIMPROGRAM"
tokenBODY = "START"
tokenDATA = "DATA"
tokenEND = "END"
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"

## 2.2. Опис термінальних символів та ключових слів

Визначимо окремі термінальні символи та нерозривні набори термінальних символів (ключові слова):

Термінальний символ або ключове слово      Значення

MAIMPROGRAM -Початок програми

START -Початок тексту програми

DATA -Початок блоку опису змінних

END -Кінець розділу операторів

GET -Оператор вводу змінних

PUT -Оператор виводу (змінних або рядкових констант)

<- -Оператор присвоєння

IF -Оператор умови

ELSE -Оператор умови

GOTO -Оператор переходу

FOR -Оператор циклу

TO -Інкремент циклу

DOWNTO -Декремент циклу

DO -Початок тіла циклу

WHILE -Оператор циклу

REPEAT -Початок тіла циклу

UNTIL -Оператор циклу

+ -Оператор додавання

- -Оператор віднімання

\* -Оператор множення

DIV -Оператор ділення

MOD -Оператор знаходження залишку від ділення

== -Оператор перевірки на рівність

!= -Оператор перевірки на нерівність

LT -Оператор перевірки чи менше

GT -Оператор перевірки чи більше

!! -Оператор логічного заперечення

AND -Оператор кон'юнкції

OR -Оператор диз'юнкції

INTEGER тип даних

%%...%% -Коментар

, -Розділювач

; -Ознака кінця оператора

( -Відкриваюча дужка

) -Закриваюча дужка

a...z - маленькі латинські букви

0...9 – цифри символи табуляції, переходу на новий рядок, пробіл

## **3. Розробка транслятора вхідної мови програмування**

### **3.1. Вибір технології програмування**

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

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

Для ефективної роботи створюваної програми важливу роль відіграє попереднє складення алгоритму роботи програми, алгоритму написання програми і вибір технології програмування.

Тому при складанні транслятора треба брати до уваги швидкість компіляції, якість об'єктної програми. Проект повинен давати можливість просто вносити зміни.

Також був обраний прямий метод лексичного аналізу. Характерною ознакою цього методу є те, що його реалізація відбувається без повернення назад. Його можна сприймати, як один спільний скінченний автомат. Такий автомат на кожному кроці читає один вхідний символ і переходить у наступний стан, що наближає його до розпізнавання поточної лексеми чи формування інформації про помилки. Для лексем, що мають однакові підланцюжки, автомат має спільні фрагменти, що реалізують єдину множину станів.

### 3.2. Проектування таблиць транслятора та вибір структур даних.

Таблиця 1 Опис термінальних символів та ключових слів

Токен	Значення
Program	MAIMPROGRAM
Start	START
Vars	DATA
End	END
VarType	INTEGER
Read	GET
Write	PUT
Assignment	<-
If	x
Else	ELSE
Goto	GOTO
Colon	:
For	FOR
To	TO
DownTo	DOWNTO
Do	DO
While	WHILE
Repeat	REPEAT
Until	UNTIL
Addition	+
Subtraction	-
Multiplication	*
Division	DIV
Mod	MOD
Equal	==

NotEqual	!=
Less	LT
Greate	GT
Not	!!
And	AND
Or	OR
Plus	+
Minus	-
Identifier	
Number	
String	
Undefined	
Unknown	
Comma	,
Semicolon	;
LBracket	(
RBracket	)
LComment	%%
RComment	%%
Comment	

### 3.3. Розробка лексичного аналізатора

На етапі лексичного аналізу вхідна програма, представлена потоком символів, розбивається на лексеми — слова, що відповідають правилам мови.

На цьому етапі лексеми записуються не у вигляді тексту, а у форматі, який містить їх символ, тип, значення та рядок, що полегшує роботу синтаксичного аналізатора.

Крім того, під час лексичного аналізу виявляються базові помилки, такі як неприпустимі символи чи неправильний запис чисел та ідентифікаторів.

Вхідними даними для лексичного аналізатора є текст вихідної програми, а вихідні дані передаються синтаксичному аналізатору для подальшої обробки.

Лексичний аналіз включається в більшість компіляторів із кількох причин:

- Спрощує обробку вихідного тексту на етапі синтаксичного аналізу.
- Дозволяє використовувати прості, ефективні й теоретично обґрунтовані методи аналізу для виділення та обробки лексем.

### 3.3.1. Розробка блок-схеми алгоритму роботи лексичного аналізатора

Алгоритм роботи лексичного аналізатора виконується поетапно:

1. **Читання символу.** Лексичний аналізатор отримує символ із тексту програми.
2. **Перевірка символу.** Визначається, чи є символ частиною лексеми (буква, цифра, розділовий знак) або помилковим.
3. **Формування лексеми.** Символи об'єднуються у слова (лексеми) відповідно до правил мови.

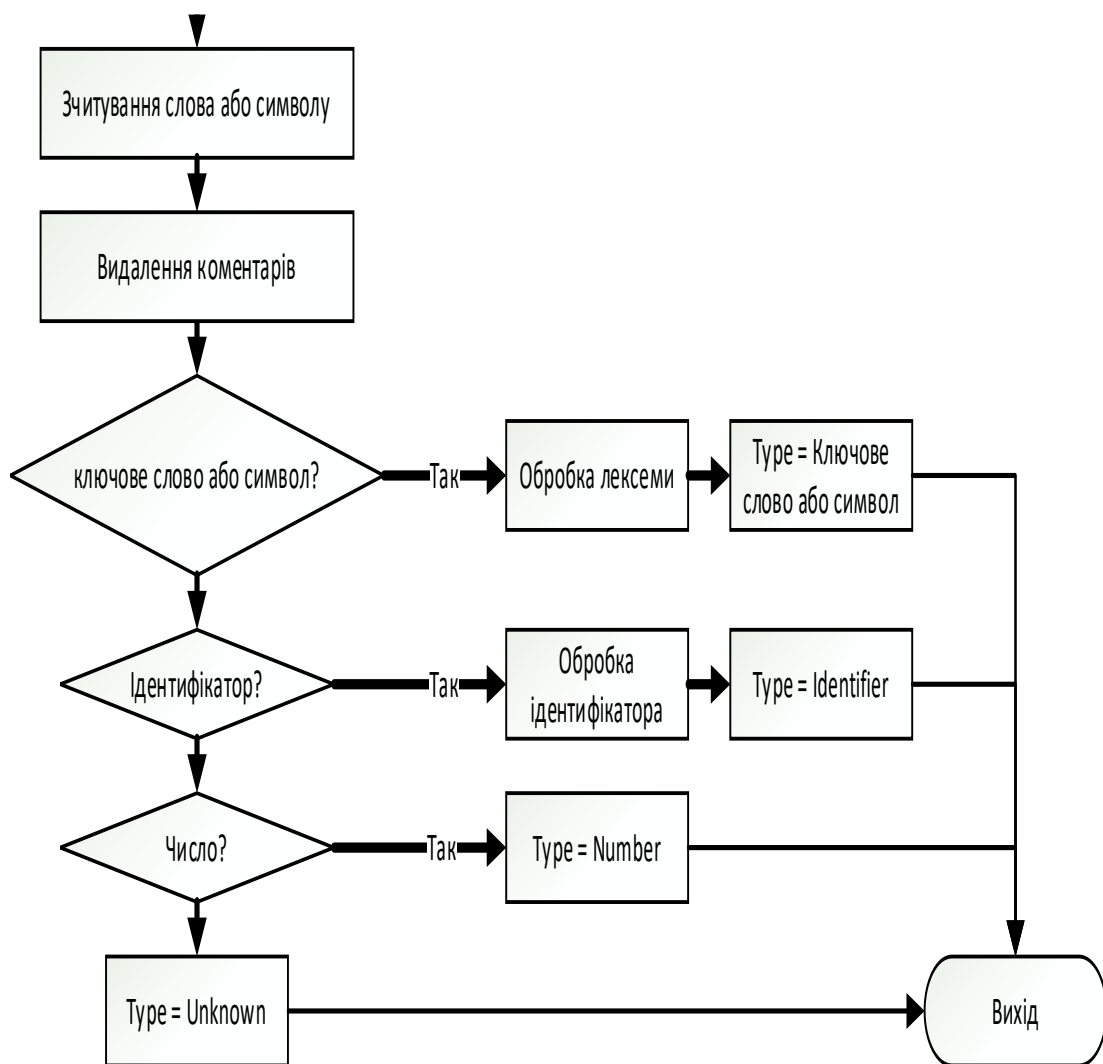


Рис. 3.3.1 Блок-схема роботи лексичного аналізатора

### 3.3.2. Опис програми реалізації лексичного аналізатора

Програма реалізації лексичного аналізатора розроблена для аналізу вхідного тексту з метою розпізнавання лексем, таких як ключові слова, ідентифікатори, числові значення та інші символи. Описаний лексичний аналізатор складається з кількох ключових модулів і функцій, які забезпечують обробку тексту, розпізнавання лексем і генерацію таблиці лексем. Нижче наведено основні аспекти його реалізації.

#### Основні компоненти програми

##### 1. Структура даних `LexemInfo`

Структура зберігає інформацію про лексеми:

- `lexemStr` — текст лексеми;
- `lexemId` — унікальний ідентифікатор лексеми;
- `tokenType` — тип токена (ідентифікатор, ключове слово, значення тощо);
- `ifvalue` — значення (для числових лексем);
- `row, col` — позиція лексеми в тексті (рядок, стовпець).

##### 2. Таблиці даних

- `lexemesInfoTable` — масив для збереження розпізнаних лексем.
- `identifierIdsTable` — таблиця унікальних ідентифікаторів.

#### Основні функції програми

##### 1. Розпізнавання лексем

- `tryToGetKeyWord()` — перевіряє, чи є лексема ключовим словом.
- `tryToGetIdentifier()` — перевіряє, чи є лексема ідентифікатором.
- `tryToGetUnsignedValue()` — перевіряє, чи є лексема числовим значенням.

##### 2. Видалення коментарів

Функція `commentRemover()` очищує текст від коментарів, використовуючи задані роздільники (початок і кінець коментаря).

##### 3. Вивід результатів

- `printLexemes()` — виводить таблицю лексем у консоль.

#### Алгоритм роботи лексичного аналізатора



1. Вхідний текст очищується від коментарів за допомогою функції `commentRemover()`.
2. Для кожного токена визначається його тип (ключове слово, ідентифікатор, числове значення тощо).
3. Лексеми зберігаються у таблиці `lexemesInfoTable`.
4. У разі помилки розпізнавання створюється запис про некоректну лексему.

### Особливості реалізації

- Регулярні вирази використовуються для розпізнавання різних типів лексем, що дозволяє легко налаштовувати правила аналізу.
- Програма підтримує обробку ключових слів, числових значень ідентифікаторів, а також виявлення помилкових лексем.

Реалізація лексичного аналізатора забезпечує ефективний і гнучкий спосіб обробки тексту, що може бути використано у різних компіляторах чи інтерпретаторах.

## 3.4. Розробка синтаксичного та семантичного аналізатора

Синтаксичний аналізатор — це компонент компілятора, що відповідає за виявлення основних синтаксичних конструкцій у вихідній мові програмування. Його завдання включає: пошук і виділення ключових синтаксичних елементів у тексті програми, визначення їх типів і перевірку правильності, підготовку інформації у форматі, придатному для подальшої генерації результуючого коду.

Основою роботи синтаксичного аналізатора є використання граматики мови програмування для розпізнавання тексту програми. Більшість мов програмування описуються за допомогою контекстно-вільних граматик (КС-грамматик), тоді як регулярні граматики рідше застосовуються і найчастіше використовуються для опису мов низького рівня, таких як мови асемблера. Натомість мови високого рівня зазвичай базуються на КС-грамматиках.

Синтаксичний розбір є ключовим етапом аналізу під час компіляції. Його виконання є необхідним для роботи компілятора, на відміну від лексичного аналізу, який є допоміжним і не обов'язковим. Завдання перевірки лексики можуть бути виконані в процесі синтаксичного розбору. Використання лексичного аналізатора (сканера) спрощує синтаксичний аналіз, делегуючи йому прості завдання, як-от розпізнавання та збереження лексем програми.

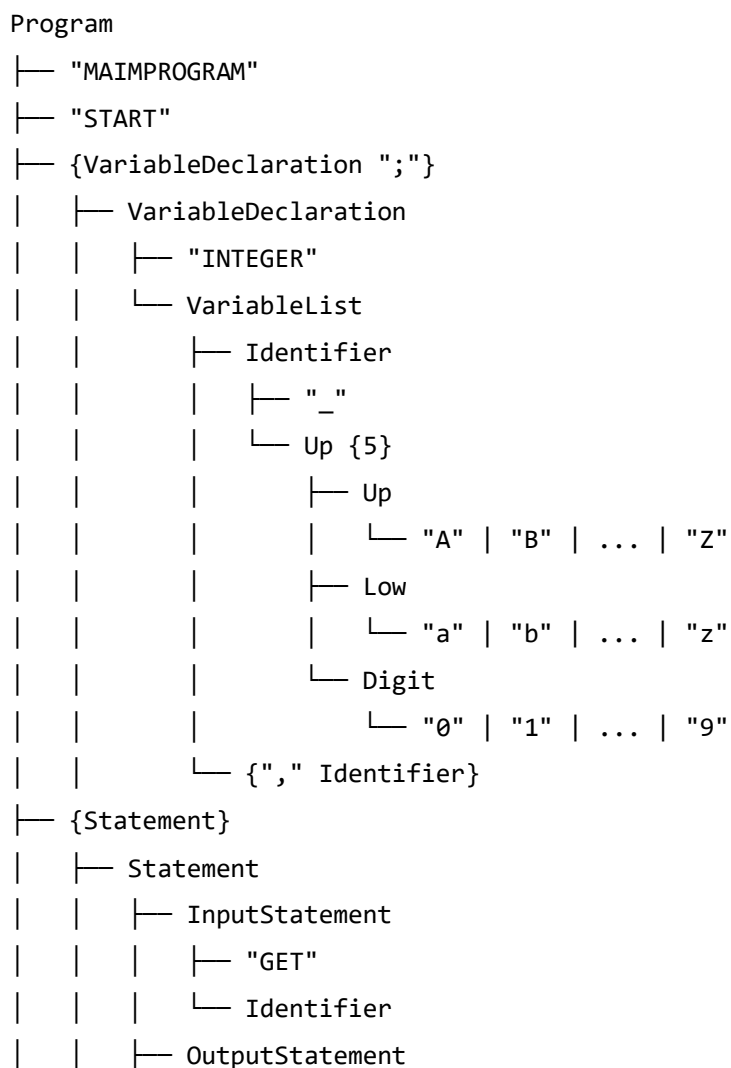
### 3.4.1. Розробка дерева граматичного розбору

Процес розробки дерева граматичного розбору полягає у створенні структури, яка відображає синтаксичні взаємозв'язки елементів мови. Для цього використовуються методи лексичного та синтаксичного аналізу, які дозволяють подати вхідний текст у вигляді дерева, де кожен вузол відповідає певній граматичній одиниці.

На початковому етапі відбувається лексичний розбір, який дозволяє поділити текст на окремі токени. Далі на основі граматики, що описує синтаксис мови, будується синтаксичне дерево. Це дерево відображає ієрархічну структуру мовних елементів та їх взаємозв'язки, що важливо для подальшої обробки або компіляції коду.

Побудоване дерево граматичного розбору є основою для наступних етапів обробки, таких як оптимізація, генерація коду чи виконання запитів, що базуються на синтаксичній структурі.

Схема дерева розбору виглядає наступним чином:



```

| | | | | └─ "PUT"
| | | | | └─ ArithmeticExpression
| | | | |   └─ LowPriorityExpression
| | | | |     └─ MiddlePriorityExpression
| | | | |       └─ Identifier
| | | | |       └─ Number
| | | | |       └─ ["-"]
| | | | |       └─ Digit {5}
| | | | |       └─ "(" ArithmeticExpression ")"
| | | | |       └─ {MiddlePriorityOperator MiddlePriorityExpression}
| | | | |       └─ {LowPriorityOperator LowPriorityExpression}
| | | └─ AssignStatement
| | |   └─ ArithmeticExpression
| | |   └─ "==" Identifier
| | └─ IfElseStatement
| |   └─ "IF"
| |   └─ "(" LogicalExpression ")"
| |   └─ AndExpression
| |   └─ Comparison
| |   └─ ComparisonExpression
| |   └─ ArithmeticExpression
| |   └─ ComparisonOperator
| |   └─ ArithmeticExpression
| |   └─ [NotOperator] "(" LogicalExpression ")"
| |   └─ {AndOperator AndExpression}
| |   └─ {OrOperator AndExpression}
| |   └─ Statement
| |   └─ ["ELSE" Statement]
| └─ GotoStatement
|   └─ "GOTO"
|   └─ Identifier
| └─ LabelPoint
|   └─ Identifier
|   └─ ":"
| └─ ForToStatement
|   └─ "FOR"
|   └─ AssignStatement
|   └─ "TO" | "DOWNT0"
|   └─ ArithmeticExpression
|   └─ "DO"
|   └─ Statement
| └─ WhileStatement

```

```

| | | | └─ "WHILE"
| | | | └─ LogicalExpression
| | | | └─ {Statement}
| | | | └─ "END" "WHILE"
| | | └─ RepeatUntilStatement
| | | | └─ "REPEAT"
| | | | └─ {Statement}
| | | | └─ "UNTIL" "(" LogicalExpression ")"
| | └─ CompoundStatement
| | | └─ "START"
| | | └─ {Statement}
| | | └─ "END"
└─ "END"

```

### **3.4.2. Розробка алгоритму роботи синтаксичного і семантичного аналізатора**

Процес розробки алгоритму синтаксичного і семантичного аналізаторів складається з кількох етапів, кожен з яких забезпечує точну перевірку структури та змісту вхідного коду.

Етапи розробки алгоритму:

#### **1. Визначення граматики мови**

На першому етапі створюється граматика, яка описує правила побудови допустимих конструкцій у мові програмування. Для цього використовуються:

- Нотація Бекуса-Наура (BNF) для формалізації синтаксичних правил.
- Типові шаблони програмних конструкцій (оператори, вирази, блоки).

#### **2. Розробка алгоритму обробки лексем**

Синтаксичний аналізатор працює з таблицею лексем, яка отримана в результаті лексичного аналізу. Для кожної лексеми визначається її роль у конструкції, перевіряється відповідність очікуваному правилу граматики.

#### **3. Алгоритм перевірки синтаксичних помилок**

Для виявлення синтаксичних помилок реалізується:

- Виявлення невідповідностей граматиці.
- Вказівка на помилки із зазначенням їхнього розташування у вихідному коді.
- Відновлення після помилок для аналізу наступних конструкцій.

#### **4. Розробка семантичного аналізатора**

Семантичний аналізатор виконує:

- Перевірку правильності типів даних.
- Перевірку наявності визначень змінних і функцій до їх використання.
- Генерацію проміжного представлення для подальшої обробки.

#### **5. Ітеративне тестування алгоритму**

Для перевірки роботи алгоритму використовуються тестові приклади, які охоплюють:

- Коректні конструкції мови.
- Типові та нестандартні синтаксичні помилки.

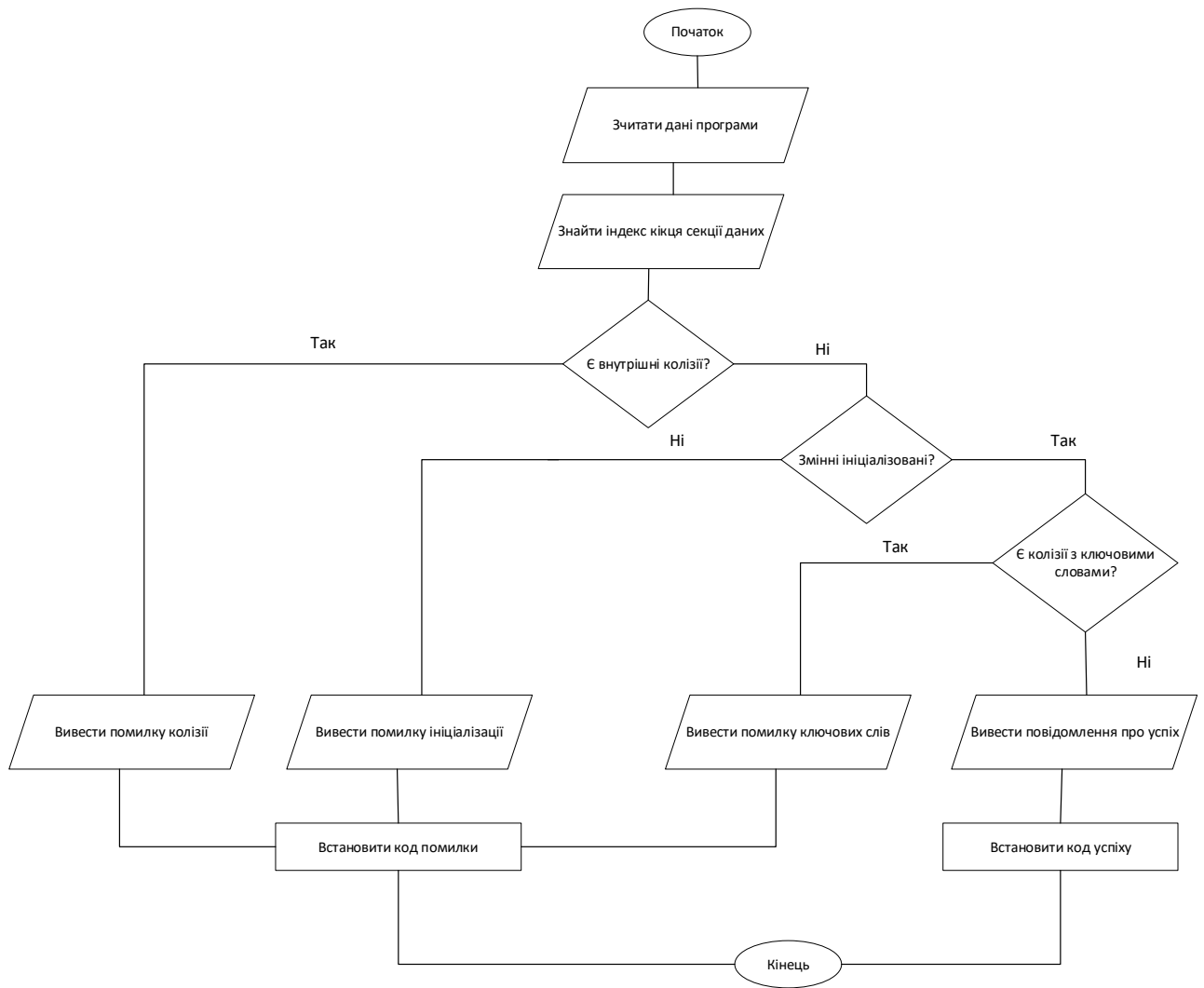


Рис. 3.4.2 Граф-схема роботи семантичного аналізатора

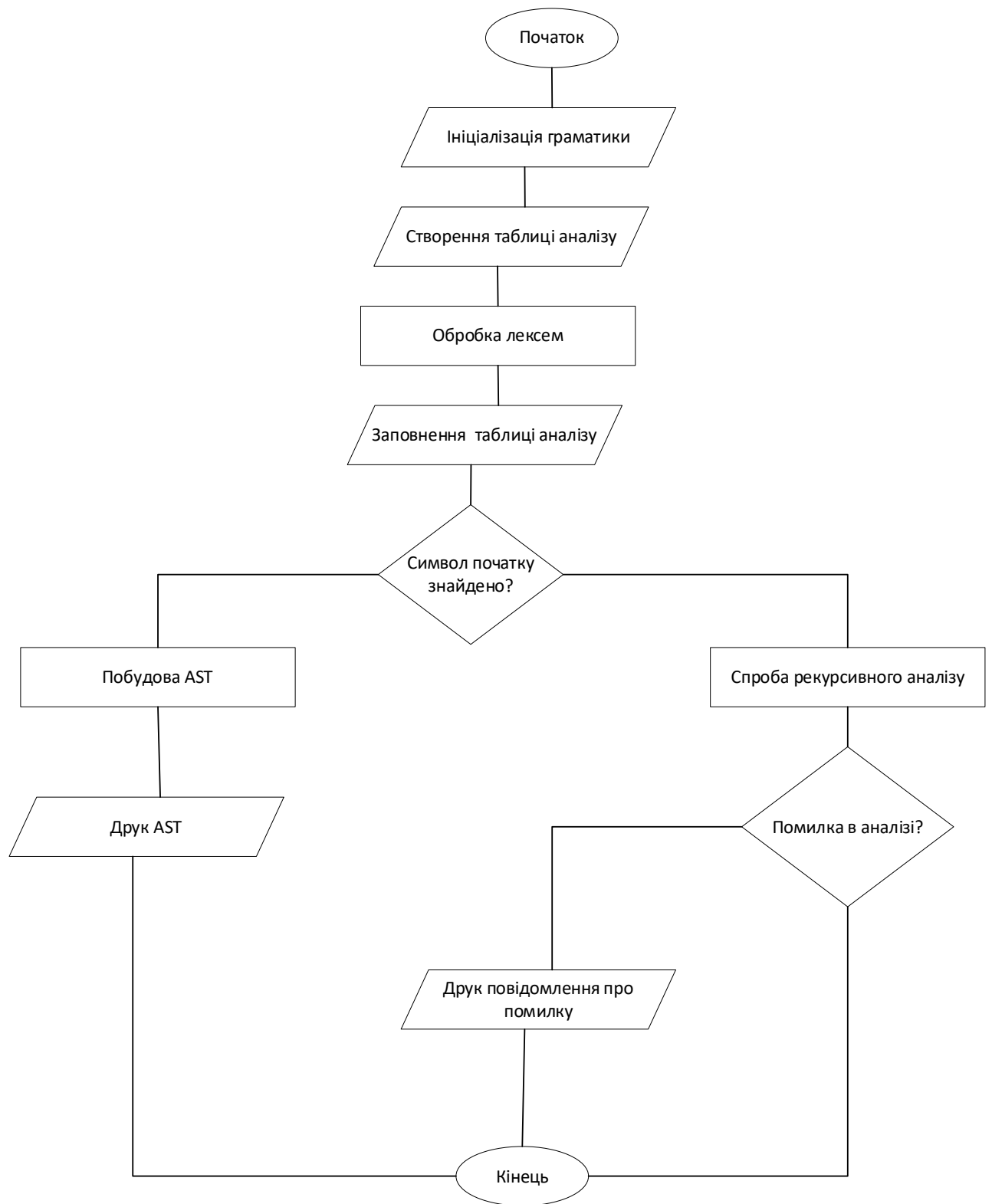


Рис. 3.4.2.2 Граф-схема роботи синтаксичного аналізатора

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

На вхід синтаксичного аналізатора подіється таблиця лексем створена на етапі лексичного аналізу. Аналізатор проходить по ній і перевіряє чи набір лексем відповідає раніше описаним формам нотації Бекуса-Наура. І разі не відповідності у файл з помилками виводиться інформація про помилку і про рядок на якій вона знаходиться.

При знаходженні оператора присвоєння або математичних виразів здійснюється перевірка балансу дужок(кількість відкриваючих дужок має дорівнювати кількості закриваючих). Також здійснюється перевірка чи не йдуть підряд декілька лексем одного типу

Результатом синтаксичного аналізу є синтаксичне дерево з посиланнями на таблиці об'єктів. У процесі синтаксичного аналізу також виявляються помилки, пов'язані зі структурою програми.

В основі синтаксичного аналізатора лежить розпізнавач тексту вхідної програми на основі граматики вхідної мови.



### 3.5. Розробка генератора коду

Синтаксичне дерево містить інформацію лише про структуру програми, однак для генерації коду потрібні також дані про змінні (наприклад, їх адреси), процедури (адреси, рівні), мітки тощо. Існують два основні підходи для представлення цієї інформації:

- зберігання даних у таблицях генератора коду;
- зберігання інформації у відповідних вузлах дерева.

Наприклад, у поєднанні з Лідер-представленням, яке не містить адрес змінних, ці дані потрібно створювати під час обробки оголошень і зберігати у таблицях. Аналогічно обробляються описи масивів, записів тощо. Крім цього, таблиці повинні містити інформацію про процедури: їхні адреси, рівні, модулі, де вони описані тощо. Під час входу в процедуру в таблицю рівнів процедур додається новий запис — посилання на таблицю описів. При виході цей вказівник повертається до попереднього значення. У разі використання дерева як проміжного представлення інформацію можна зберігати безпосередньо у його вузлах.

Генерація коду — це етап компіляції, залежний від архітектури, під час якого створюється машинний еквівалент вхідної програми. На вхід генератору зазвичай надходить проміжна форма представлення програми, а на виході формується об'єктний код або модуль завантаження.

Генератор асемблерного коду отримує масив лексем без помилок. Якщо на попередніх етапах були виявлені помилки, ця фаза не виконується.

У межах цього курсового проєкту етап генерації коду реалізовано окремо. Його виконання можливе лише за умови успішного завершення синтаксичного аналізу.

### 3.5.1. Розробка алгоритму роботи генератора коду

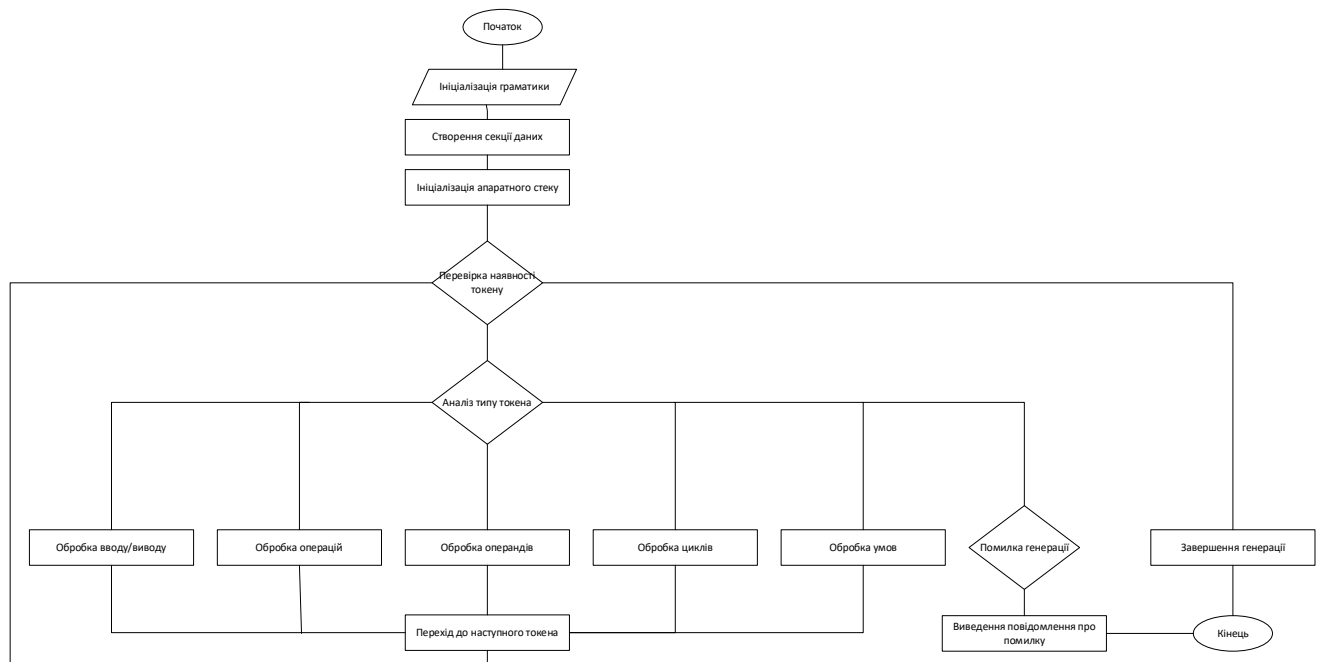


Рис. 3.5.1 Блок схема генератора коду

### **3.5.2. Опис програми реалізації генератора коду**

Програма реалізації генератора коду представляє собою модуль, який виконує перетворення внутрішнього представлення програми у виконуваний машинний код. Основна функціональність включає роботу з токенами, генерацію інструкцій та управління пам'яттю.

Ключові компоненти генератора включають:

#### **1. Структури даних:**

- Інформація про мітки та їх розташування в коді
- Дані про інструкції переходу та їх позиції
- Структури для зберігання токенів та їх властивостей

#### **2. Основні функціональні блоки:**

- Генерація заголовка програми
- Створення оголошень залежностей
- Формування секції даних
- Створення початкового та завершального коду
- Управління апаратним стеком
- Ініціалізація програмних компонентів

#### **3. Механізми обробки токенів:**

- Аналіз та класифікація токенів
- Обробка різних типів конструкцій
- Генерація відповідного машинного коду

#### **4. Модулі генерації коду для різних операцій:**

- Арифметичні операції
- Логічні операції
- Керуючі конструкції
- Операції введення/виведення
- Робота з мітками та переходами

#### **5. Системні параметри:**

- Визначення розмірів буферів
- Налаштування зміщень даних
- Конфігурація режимів роботи

#### **6. Обробка помилок:**

- Перевірка коректності токенів
- Валідація синтаксичних конструкцій
- Формування повідомлень про помилки

Програма працює в декілька етапів:

#### **1. Ініціалізація:**

- Встановлення початкових параметрів
- Підготовка структур даних
- Налаштування секцій коду та даних

#### **2. Основний цикл генерації:**

- Послідовний аналіз токенів
- Визначення типу конструкції
- Створення машинних інструкцій

#### **3. Завершення:**

- Завершальні операції генерації
- Відновлення стану системи
- Фіналізація програми

Генератор підтримує такі режими роботи:

- Генерація машинного коду
- Створення асемблерних інструкцій
- Формування об'єктного коду
- Режим налагодження

Результатом роботи програми є оптимізований машинний код, який готовий до виконання на цільовій платформі. Програма забезпечує ефективне перетворення високорівневих конструкцій мови програмування в низькорівневі інструкції процесора.

## **4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА**

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

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

## 4.1. Опис інтерфейсу та інструкція користувачеві

Програма підтримує інтерактивний режим роботи для покрокового виконання завдань або режим автоматичного виконання команд на основі параметрів командного рядка. Основна мета — аналіз та обробка вихідного коду за допомогою таких етапів:

1. Лексичний аналіз.
2. Синтаксичний аналіз.
3. Семантичний аналіз.
4. Генерація машинного коду.

### Інтерактивний режим

У разі відсутності параметрів командного рядка програма переходить до інтерактивного режиму, де користувачу пропонується покроково виконувати всі етапи обробки. Ключові моменти:

1. Лексичний аналіз: введіть у для виконання, n для пропуску.
2. Синтаксичний аналіз: підтвердьте виконання, натиснувши у.
3. Семантичний аналіз: підтвердьте дію аналогічно.
4. Генерація коду

### Інструкція користувача

#### 1. Завантаження вихідного файлу

Переконайтесь, що файл початкового коду підготовлений і правильно вказаний параметром --input.

#### 2. Діагностика помилок

У разі виявлення помилок програма виведе відповідні повідомлення у консоль або запише їх у файл, якщо це вказано параметром --output.

#### 3. Завершення роботи

Після успішного виконання всіх етапів програма виведе повідомлення про завершення, а результати буде збережено в указаних файлах.

### Помилки та їх усунення

- Якщо вихідний файл порожній, програма повідомить:

Empty source . . .

Press Enter to exit . . .

- У разі помилок у синтаксисі або семантиці відповідні повідомлення буде виведено в консоль та/або записано у файл.

Приклад:

Lexical analysis detected unexpected lexeme.

## 4.2. Виявлення лексичних та синтаксичних помилок

Лексичні помилки виявляються на етапі лексичного аналізу, коли вхідний код розбивається на окремі лексеми. Кожна лексема перевіряється на відповідність до визначених правил мови програмування. Якщо лексема не відповідає жодному з правил, вона позначається як некоректна, і виникає повідомлення про помилку.

Синтаксичні помилки виявляються після лексичного аналізу, коли програма перевіряється на відповідність до граматичних правил мови. Тут аналізу підлягають окремі конструкції, такі як вирази, оператори, цикли, а також загальна структура програми. Якщо виявлені помилки, програма не проходить синтаксичний аналіз і виводиться відповідне повідомлення.

Якщо зробити у програмі синтаксичну помилку то програма вкаже її:

```
Source after comment removing:
-----
MAIMPROGRAMM _PROGR ;
START DATA INTEGER _VALUE , _RESUL , _CYCLE ;
  GET ( _VALUE )
  _RESUL <- 1
  FOR _CYCLE <- 0 TO 32767 DO
    IF ( _VALUE != 0 ) ; ELSE GOTO _ENDCY ;
    _RESUL <- _RESUL * _VALUE
    _VALUE <- _VALUE - 1
  ;
  _ENDCY :
  PUT ( _RESUL )
END
-----

Lexical analysis detected unexpected lexeme
Bad lexeme:
-----
index          lexeme          id      type      ifvalue row    col
-----
0      MAIMPROGRAMM          0      127          0    1      1
-----
```

Рис. 4.2. Вивід інформації про синтаксичну помилку.

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

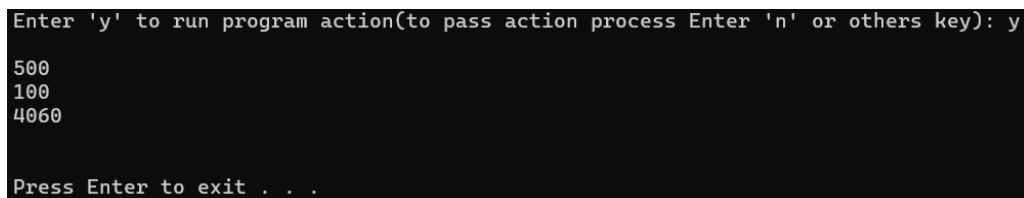
#### Тестова програма №1 *Лінійний алгоритм*

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

#### *Текст програми*

```
MAIMPROGRAM _PROGR ;  
START DATA INTEGER _VALUE , _RESUL , _CYCLE ;  
  GET (_VALUE)  
  GET (_RESUL)  
  _CYCLE <- 10 * (_VALUE - _RESUL) + (_VALUE + _RESUL) DIV 10  
  PUT (_CYCLE )  
  
END
```

Результат:



```
Enter 'y' to run program action(to pass action process Enter 'n' or others key): y  
500  
100  
4060  
  
Press Enter to exit . . .
```

*Рис. 4.3. Результати виконання тестової задачі 2.*



## Тестова програма №2

Тестова програма «Алгоритм з розгалуженням»

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

Використання вкладеного умовного оператора:

Знайти найбільше з них і вивести його на екран.

Використання простого умовного оператора:

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

### *Текст програми*

```
MAIMPROGRAM _PROGR ;
START DATA INTEGER _AVVVV , _BVVVV , _CVVVV ;
  GET (_AVVVV)
  GET (_BVVVV)
  GET (_CVVVV)
  IF ( _AVVVV == _BVVVV ) ; ELSE GOTO _CALUE ;
    GOTO _BALUE
_CALUE :
  PUT ( 0 )
  GOTO _ENASD
_BALUE :
  IF ( _AVVVV == _CVVVV ) ; ELSE GOTO _CALUE ;
    PUT ( 1 )
_ENASD:
  GET (_AVVVV)
END
```

Отримую такі результати:

```
Enter 'y' to run program action(to pass action process Enter 'n' or others key): y
2
2
3
0
.
```

```
Enter 'y' to run program action(to pass action process Enter 'n' or others key): y
1
1
1
1
1
```

*Рис. 4.3.1. Результати виконання тестової задачі 2.*

# Висновки

У рамках курсового проекту було розроблено транслятор для вхідної мови програмування, що виконує наступні основні завдання:

## 1. Лексичний аналіз:

- Алгоритм лексичного аналізу розділяє текст програми на лексеми та формує таблицю з їхнім типом, значенням та номером рядка.
- Лексичний аналізатор працює за принципом скінченного автомата, розпізнаючи ключові слова, ідентифікатори, константи, оператори та розділювачі.

## 2. Синтаксичний і семантичний аналіз:

- Синтаксичний аналізатор перевіряє правильність структури програми за граматикою, створюючи дерево розбору та таблиці ідентифікаторів і типів.
- Семантичний аналізатор перевіряє логічну коректність програми, зокрема відповідність типів даних, області видимості змінних і правильність викликів функцій.

## 3. Генерація коду:

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

## 4. Тестування:

- Процес тестування проводився на різних типах програм (лінійні, з розгалуженням, циклічні), що дозволило виявити й виправити лексичні, синтаксичні та семантичні помилки.
- Транслятор успішно генерує коректний код на основі вхідних програм.

## Переваги проекту:

- Поетапна реалізація всіх етапів трансляції.
- Модульна архітектура, яка спрощує розширення та подальшу модифікацію.
- Ретельне тестування підтвердило стабільність і надійність роботи програми.

## Список використаної літератури

1. Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс]: навч. посіб. для студ. спеціальності 123 – «Комп'ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. – Київ: КПІ ім. Ігоря Сікорського, 2021. – 108 с.
2. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. – Харків: НТУ «ХПІ», 2021. – 133 с.
3. Сопронюк Т.М. Системне програмування. Частина I. Елементи теорії формальних мов: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 с.
4. Сопронюк Т.М. Системне програмування. Частина II. Елементи теорії компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 с.
5. Language Processors: Assembler, Compiler and Interpreter  
URL: [Language Processors: Assembler, Compiler and Interpreter - GeeksforGeeks](#)
6. Error Handling in Compiler Design  
URL: [Error Handling in Compiler Design - GeeksforGeeks](#)

## Додатки

### ДОДАТОК А. Таблиці лексем для тестових прикладів

Тестова програма «лінійного алгоритму»

Lexemes table:						
index	lexeme	id	type	ifvalue	row	col
0	MAIMPROGRAM	289	1	0	1	1
1	_PROGR	0	2	0	1	13
2	;	256	1	0	1	20
3	START	306	1	0	2	1
4	DATA	301	1	0	2	7
5	INTEGER	406	1	0	2	12
6	_VALUE	1	2	0	2	20
7	,	267	1	0	2	27
8	_RESUL	2	2	0	2	29
9	,	267	1	0	2	36
10	_CYCLE	3	2	0	2	38
11	;	256	1	0	2	45
12	GET	334	1	0	3	3
13	(	278	1	0	3	7
14	_VALUE	1	2	0	3	8
15	)	281	1	0	3	14
16	GET	334	1	0	4	3
17	(	278	1	0	4	7
18	_RESUL	2	2	0	4	8
19	)	281	1	0	4	14
20	_CYCLE	3	2	0	5	3
21	<=	258	1	0	5	10
22	10	320	4	10	5	13
23	*	265	1	0	5	16
24	(	278	1	0	5	19
25	_VALUE	1	2	0	5	20
26	-	259	1	0	5	27
27	_RESUL	2	2	0	5	29
28	)	281	1	0	5	35
29	+	261	1	0	5	37
30	(	278	1	0	5	39
31	_VALUE	1	2	0	5	40
32	+	261	1	0	5	37
33	_RESUL	2	2	0	5	49
34	)	281	1	0	5	55
35	DIV	391	1	0	5	57
36	10	320	4	10	5	61
37	PUT	338	1	0	6	1
38	(	278	1	0	6	5
39	_CYCLE	3	2	0	6	7
40	)	281	1	0	6	14
41	END	312	1	0	8	1

# Тестова програма «Алгоритм з розгалуженням»

Lexemes table:

index	lexeme	id	type	ifvalue	row	col
0	MAIMPROGRAM	289	1	0	1	1
1	_PROGR	0	2	0	1	13
2	;	256	1	0	1	20
3	START	306	1	0	2	1
4	DATA	301	1	0	2	7
5	INTEGER	406	1	0	2	12
6	_AVVVV	1	2	0	2	20
7	,	267	1	0	2	27
8	_BVVVV	2	2	0	2	29
9	,	267	1	0	2	36
10	_CVVVV	3	2	0	2	38
11	;	256	1	0	2	45
12	GET	334	1	0	3	3
13	(	278	1	0	3	7
14	_AVVVV	1	2	0	3	8
15	)	281	1	0	3	14
16	GET	334	1	0	4	3
17	(	278	1	0	4	7
18	_BVVVV	2	2	0	4	8
19	)	281	1	0	4	14
20	GET	334	1	0	5	4
21	(	278	1	0	5	8
22	_CVVVV	3	2	0	5	9
23	)	281	1	0	5	15
24	IF	342	1	0	6	5
25	(	278	1	0	6	8
26	_AVVVV	1	2	0	6	10
27	==	269	1	0	6	17
28	_BVVVV	2	2	0	6	20
29	)	281	1	0	6	27
30	;	256	1	0	6	29
31	ELSE	345	1	0	6	31
32	GOTO	386	1	0	6	36
33	_CALUE	4	2	0	6	41
34	;	256	1	0	6	48
35	GOTO	386	1	0	6	36
36	_BALUE	5	2	0	7	12
37	_CALUE	4	2	0	8	2
38	:	275	1	0	8	9
39	PUT	338	1	0	9	3
40	(	278	1	0	9	7
41	0	320	4	0	9	9
42	)	281	1	0	9	11
43	GOTO	386	1	0	10	3
44	_ENASD	6	2	0	10	8
45	_BALUE	5	2	0	11	4
46	:	275	1	0	11	11
47	IF	342	1	0	12	6
48	(	278	1	0	12	9
49	_AVVVV	1	2	0	12	11
50	==	269	1	0	12	18
51	_CVVVV	3	2	0	12	21
52	)	281	1	0	12	28
53	;	256	1	0	12	30
54	ELSE	345	1	0	12	32
55	GOTO	386	1	0	12	37
56	_CALUE	4	2	0	12	42
57	;	256	1	0	12	49
58	PUT	338	1	0	13	7
59	(	278	1	0	13	11
60	1	320	4	1	13	13
61	)	281	1	0	13	15
62	_ENASD	6	2	0	14	1
63	:	275	1	0	14	7
64	GET	334	1	0	15	3
65	(	278	1	0	15	7
66	_AVVVV	1	2	0	15	8
67	)	281	1	0	15	14
68	END	312	1	0	16	1

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

Тестова програма «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


;"GET"

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
```

```
;"GET"
```

```
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
```

```
;"_VALUE"
```

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

```
;"_RESUL"
```

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

```
;"_"
```

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

```
;"*"
```

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

```
;"_VALUE"
```

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

```
;"_RESUL"
```

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

```
;"+"
```

```
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
```

```
;"+"
```

```
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)


;"_CYCLE"

mov eax, edi

add eax, 00000000Ch

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax


;"PUT"

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)
```

```
;hw stack reset(restore esp)
```

```
mov esp, ebp
```

```
xor eax, eax
```

```
ret
```

```
end start
```

Тестова програма «Алгоритм з розгалуженням»

.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

;"GET"
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
```

```
;"GET"
```

```
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
```

```
;"GET"
```

```
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)
```

```
;"IF"
```

```
;"_AVVVV"
```

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

```
;"_BVVVV"
```

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

```
;"=="
```

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

sete al

and eax, 1

mov dword ptr[ecx], eax

;after cond expresion (after "IF")

cmp eax, 0

jz LABEL@AFTER\_THEN\_00007FF7389169B0

;" (after "then"-part of IF-operator)

mov eax, 1

LABEL@AFTER\_THEN\_00007FF7389169B0:

;"ELSE"

cmp eax, 0

jnz LABEL@AFTER\_ELSE\_00007FF738917628

;"GOTO" previous ident "\_CALUE"(as label)

jmp LABEL@0000027F63863778

;null statement (non-context)

;" (after "ELSE")

LABEL@AFTER\_ELSE\_00007FF738917628:

;"GOTO" previous ident "\_BALUE"(as label)

jmp LABEL@0000027F63849848

;null statement (non-context)

;ident "\_CALUE"(as label) previous ":"

LABEL@0000027F63863778:

;"0"

add ecx, 4

mov eax, 000000000h

mov dword ptr [ecx], eax

;"PUT"

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)

; "GOTO" previous ident "\_ENASD"(as label)

jmp LABEL@0000027F6384C0C8

; null statement (non-context)

; ident "\_BALUE"(as label) previous ":"

LABEL@0000027F63849848:

; "IF"

; "\_AVVVV"

mov eax, edi

add eax, 000000004h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

; "\_CVVVV"

mov eax, edi

add eax, 00000000Ch

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

; "=="

mov eax, dword ptr[ecx]

```

sub ecx, 4

cmp dword ptr[ecx], eax

sete al

and eax, 1

mov dword ptr[ecx], eax


;after cond expresion (after "IF")

cmp eax, 0

jz LABEL@AFTER_THEN_00007FF73891F2D8


;";" (after "then"-part of IF-operator)

mov eax, 1

LABEL@AFTER_THEN_00007FF73891F2D8:


;"ELSE"

cmp eax, 0

jnz LABEL@AFTER_ELSE_00007FF73891FF50


;"GOTO" previous ident "_CALUE"(as label)

jmp LABEL@0000027F63863778


;null statement (non-context)


;";" (after "ELSE")

LABEL@AFTER_ELSE_00007FF73891FF50:


;"1"

```

```

add ecx, 4

mov eax, 000000001h

mov dword ptr [ecx], eax


;"PUT"

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)


;ident "_ENASD"(as label) previous ":"

LABEL@0000027F6384C0C8:


;"4"

add ecx, 4

```



```

mov eax, 000000004h
mov dword ptr [ecx], eax

;"GET"

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

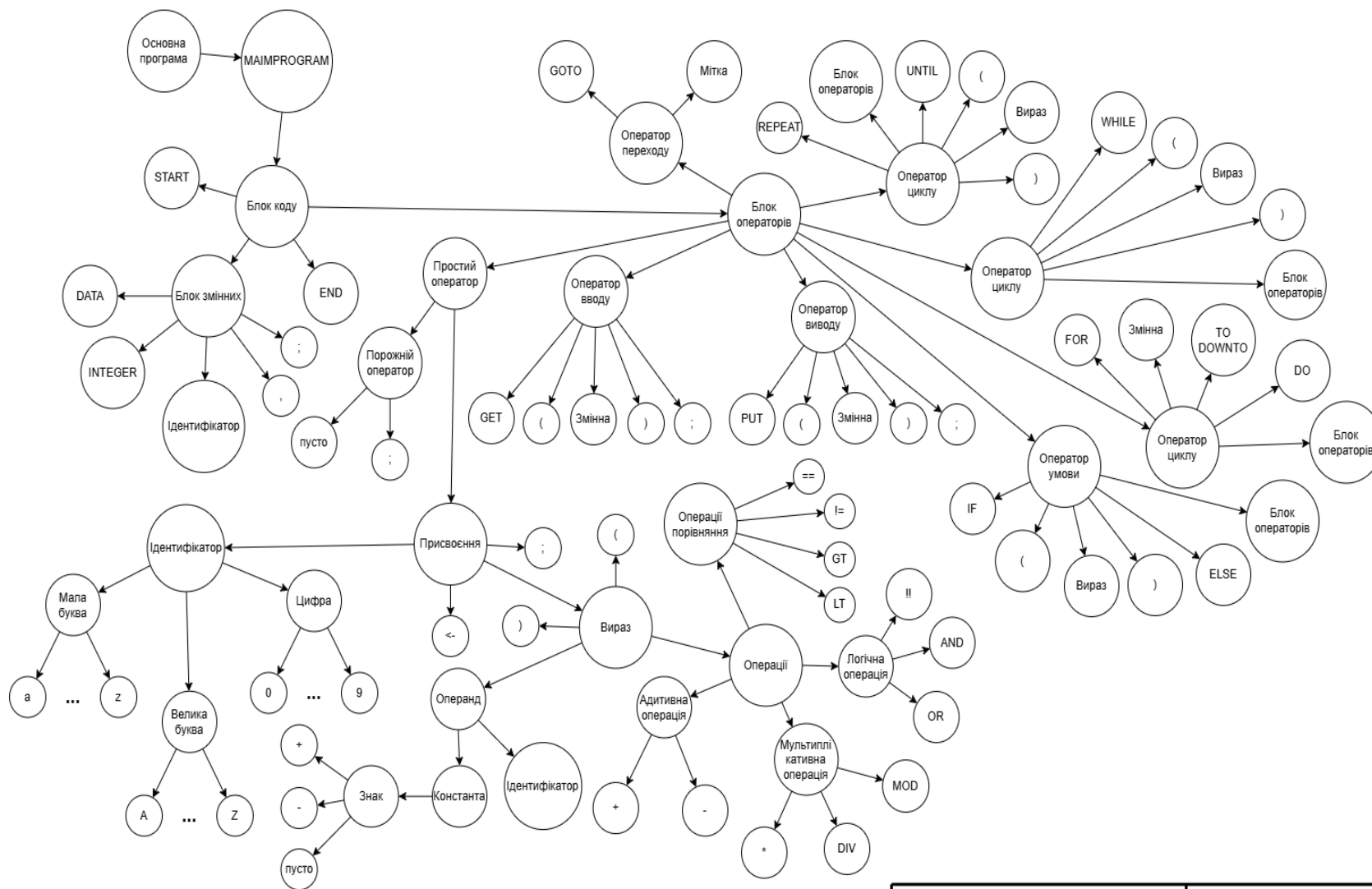
```

```
xor eax, eax
```

```
ret
```

```
end start
```

Додаток В. Дерево граматичного розбору



Міністерство освіти і науки України					КУРСОВИЙ ПРОЄКТ				
					Розробка системних програмних модулів та компонент систем програмування				
					Дерево граматичного розбору				
Зм.	Арк.	№ докум.	Підпис	Дата	Літера		Маса		Масштаб
Виконав		Патрило Ю. А.			у				
Керівник		Козак Н.Б.							
Консульт.					Аркуш		Аркушів 1		
Консульт.					НУ «ЛП», ІКТА, каф. ЕОМ, гр КІ-307				
Зав. каф.		Дунець Р. Б.							
Реценз.									

## D. Лістинг програми

Add.cpp

```
#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: add.cpp           *

*                               (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_ADD);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\\"%s\\"r\n", tokenStruct[MULTI_TOKEN_ADD][0]);

#endif

        const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

        const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };

        const unsigned char code__add_stackTopByECX_eax[] = { 0x01, 0x01 };

        //const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_stackTopByECX_eax, 2);
```

```
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);
```

```
#ifndef DEBUG_MODE_BY_ASSEMBLY
```

```
    printf("    mov eax, dword ptr[ecx]\r\n");
```

```
    printf("    sub ecx, 4\r\n");
```

```
    printf("    add dword ptr[ecx], eax\r\n");
```

```
    printf("    mov eax, dword ptr[ecx]\r\n");
```

```
#endif
```

```
    return *lastLexemInfoInTable += multitokenSize, currBytePtr;
```

```
}
```

```
return currBytePtr;
```

```
}
```

And.cpp

```
#define _CRT_SECURE_NO_WARNINGS
```

```
/* ****
```

```
* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025 *
```

```
* file: and.cpp *
```

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

```
****/
```

```
#include "../include/def.h"
```

```
#include "../include/generator/generator.h"
```

```
#include "../include/lexica/lexica.h"
```

```
#include "stdio.h"
```

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

```
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_AND);
```

```
    if (multitokenSize) {
```

```
#ifdef DEBUG_MODE_BY_ASSEMBLY
```

```
    printf("\r\n");
```

```
    printf("    ;\"%s\"\r\n", tokenStruct[MULTI_TOKEN_AND][0]);
```

```
#endif
```

```
    const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };
```

```
    const unsigned char code__cmp_eax_0[] = { 0x83, 0xF8, 0x00 };
```

```
    const unsigned char code__setne_al[] = { 0x0F, 0x95, 0xC0 };
```

```
    const unsigned char code__and_eax_1[] = { 0x83, 0xE0, 0x01 };
```

```
    const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };
```

```
    //
```

```
    const unsigned char code__cmp_stackTopByECX_0[] = { 0x83, 0x39, 0x00 };
```

```
    const unsigned char code__setne_dl[] = { 0x0F, 0x95, 0xC2 };
```

```
    const unsigned char code__and_edx_1[] = { 0x83, 0xE2, 0x01 };
```

```
    //
```

```
    const unsigned char code__and_eax_edx[] = { 0x23, 0xC2 };
```

```
    //
```

```
    const unsigned char code__mov_stackTopByECX_eax[] = { 0x89, 0x01 };
```

```
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);
```

```
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_eax_0, 3);
```

```
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__setne_al, 3);
```

```
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_eax_1, 3);
```

```
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);
```

```
    //
```

```
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_stackTopByECX_0, 3);
```

```

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__setne_dl, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_edx_1, 3);

//

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_eax_edx, 2);

//

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_stackTopByECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("  mov eax, dword ptr[ecx]\r\n");

    printf("  cmp eax, 0\r\n");

    printf("  setne al\r\n");

    printf("  and eax, 1\r\n");

    printf("  sub ecx, 4\r\n");

    //

    printf("  cmp dword ptr[ecx], 0\r\n");

    printf("  setne dl\r\n");

    printf("  and edx, 1\r\n");

    //

    printf("  and eax, edx\r\n");

    //

    printf("  mov dword ptr[ecx], eax\r\n");

#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

cli.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: cw_lex.cpp           *

*           (draft!) *

*****/

#include "../././src/include/cli/cli.h"

#include "../././src/include/def.h"

#include "../././src/include/config.h"

#include "../././src/include/generator/generator.h"

#include "../././src/include/lexica/lexica.h"

#include "stdio.h"

```

```

#include "stdlib.h"

#include "string.h"


unsigned long long int mode = 0;

char parameters[PARAMETERS_COUNT][MAX_PARAMETERS_SIZE] = { "" };


void comandLineParser(int argc, char* argv[], unsigned long long int* mode, char(*parameters)[MAX_PARAMETERS_SIZE]) {

    char tempTemp[PATH_NAME_LENGTH] = { '\0' }, * tempPtrPrev, * tempPtrNext, nameTemp[PATH_NAME_LENGTH] = { '\0' };

    char modesNotDefined = 1;

    *mode = 0;

    for (int index = 1; index < argc; ++index) {

        if (!strcmp(argv[index], "-lex")) {

            *mode |= LEXICAL_ANALYZE_MODE;

            modesNotDefined = 0;

            continue;

        }

        else if (!strcmp(argv[index], "-stx")) {

            *mode |= SYNTAX_ANALYZE_MODE;

            modesNotDefined = 0;

            continue;

        }

        else if (!strcmp(argv[index], "-smt")) {

            *mode |= SEMANTIX_ANALYZE_MODE;

            modesNotDefined = 0;

            continue;

        }

        else if (!strcmp(argv[index], "-gen")) {

            *mode |= MAKE_ASSEMBLY | MAKE_BINARY;

            modesNotDefined = 0;

            continue;

        }

        else if (!strcmp(argv[index], "-run")) {

            *mode |= RUN_BINARY;

            modesNotDefined = 0;

            continue;

        }

        else if (!strcmp(argv[index], "-all")) {

            *mode |= LEXICAL_ANALYZE_MODE | SYNTAX_ANALYZE_MODE | SEMANTIX_ANALYZE_MODE |
MAKE_ASSEMBLY | MAKE_BINARY | RUN_BINARY;

            modesNotDefined = 0;

            continue;

        }

        else if (!strcmp(argv[index], "-d")) {

```



```

        *mode |= DEBUG_MODE;

        modesNotDefined = 0;

        continue;
    }

    // other keys
    // TODO:...

    // input filename
    strncpy(parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER], argv[index], MAX_PARAMETERS_SIZE);
}

// default mode, if not entered manually
if (modesNotDefined) {
    if (parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER][0] != '\0') {
        *mode = LEXICAL_ANALYZE_MODE | SYNTAX_ANALYZE_MODE | SEMANTIX_ANALYZE_MODE |
MAKE_ASSEMBLY | MAKE_BINARY;
    }
    else {
        *mode = UNDEFINED_MODE; // | INTERACTIVE_MODE | ;

        printf("Used interactive mode\r\n\r\n");
    }
}

if (*mode & UNDEFINED_MODE) {
    *mode |= INTERACTIVE_MODE;

    // *mode |= DEFAULT_MODE;

    *mode |= DEBUG_MODE;
}

// default input filename, if not entered manually
if (parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\0') {
    strcpy(parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER], DEFAULT_INPUT_FILENAME);

    // printf("Input filename not setted. Used defaule input filename \"%s\"\r\n\r\n",
parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER]);

    char choice[2] = { parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER][0],
parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER][1] };

    // std::cout << "Enter file name(Enter \"\" << choice[0] << "\"" to use default \"\" DEFAULT_INPUT_FILE "\":\n";

    printf("Input filename not setted. Enter file name(or enter '%c' to use default \"%s\"): ",
parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER][0], parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER]);

    // std::cin >> fileName;

    (void)scanf("%s", parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER]/*, MAX_PARAMETERS_SIZE*/);

    if (parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER][0] == choice[0] &&
parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER][1] == '\0') {

        parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER][1] = choice[1];
    }
}

```

```

    }

    printf("\r\n");
}

strncpy(nameTemp, parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER], PATH_NAME_LENGTH);
nameTemp[PATH_NAME_LENGTH - 1] = '\0';
tempPtrPrev = nameTemp;
tempPtrNext = NULL;
for (; tempPtrNext = strstr(tempPtrPrev + 1, "."); tempPtrPrev = tempPtrNext);
if (tempPtrPrev != nameTemp) {
    *tempPtrPrev = '\0';
}

#if 0

strncpy(tempTemp, parameters[INPUT_FILENAME_WITH_EXTENSION_PARAMETER], PATH_NAME_LENGTH);
tempPtrPrev = tempTemp;
tempPtrPrev[0] == '"' ? ++tempPtrPrev : 0;
tempPtrNext = tempPtrPrev = strtok(tempPtrPrev, ".\\/:");
while (tempPtrNext != NULL) {
    tempPtrNext = strtok(NULL, ".\\/:");
    if (tempPtrPrev && tempPtrNext) {
        strncpy(nameTemp, tempPtrPrev, PATH_NAME_LENGTH);
    }
    tempPtrPrev = tempPtrNext;
}

#endif

// default temp filename, if not entered manually
if (*mode & (MAKE_LEXEMES_SEQUENSE | INTERACTIVE_MODE) &&
parameters[OUT_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\0') {
    if (*mode & INTERACTIVE_MODE) {
        system("CLS");
        fflush(stdin);
        fflush(stdout);
        fflush(stderr);
    }

    strncpy(parameters[OUT_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp,
PATH_NAME_LENGTH);
    strcat(parameters[OUT_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER], "_lexemes.txt",
PATH_NAME_LENGTH - strlen(parameters[OUT_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER]));
    printf("Out lexemes sequense filename not setted. Used defaule input filename \"%s\"\r\n",
parameters[OUT_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER]);
    if (*mode & INTERACTIVE_MODE) {
        printf("Press Enter to next step");
    }
}

```

```

        (void)getchar();

        (void)getchar();

    }

}

// default temp filename, if not entered manually

    if (*mode & (MAKE_LEXEMES_SEQUENSE | INTERACTIVE_MODE) &&
parameters[OUT_LEXEME_ERROR_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\0') {

        if (*mode & INTERACTIVE_MODE) {

            system("CLS");

            fflush(stdin);

            fflush(stdout);

            fflush(stderr);

        }

        strncpy(parameters[OUT_LEXEME_ERROR_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp, PATH_NAME_LENGTH);

        strcat(parameters[OUT_LEXEME_ERROR_FILENAME_WITH_EXTENSION_PARAMETER], "_lexeme_error.txt",
PATH_NAME_LENGTH - strlen(parameters[OUT_LEXEME_ERROR_FILENAME_WITH_EXTENSION_PARAMETER]));

        printf("Out lexemes sequense filename not setted. Used defaule input filename \"%s\"\r\n",
parameters[OUT_LEXEME_ERROR_FILENAME_WITH_EXTENSION_PARAMETER]);

        if (*mode & INTERACTIVE_MODE) {

            printf("Press Enter to next step");

            (void)getchar();

            //(void)getchar();

        }

    }

// default temp filename, if not entered manually

    if (*mode & (MAKE_AST | INTERACTIVE_MODE) && parameters[OUT_AST_FILENAME_WITH_EXTENSION_PARAMETER][0] ==
'\0') {

        if (*mode & INTERACTIVE_MODE) {

            system("CLS");

            fflush(stdin);

            fflush(stdout);

            fflush(stderr);

        }

        strncpy(parameters[OUT_AST_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp, PATH_NAME_LENGTH);

        strcat(parameters[OUT_AST_FILENAME_WITH_EXTENSION_PARAMETER], ".ast", PATH_NAME_LENGTH -
strlen(parameters[OUT_AST_FILENAME_WITH_EXTENSION_PARAMETER]));

        printf("Out AST filename not setted. Used defaule input filename \"%s\"\r\n",
parameters[OUT_AST_FILENAME_WITH_EXTENSION_PARAMETER]);

        if (*mode & INTERACTIVE_MODE) {

            printf("Press Enter to next step");

            (void)getchar();

            //(void)getchar();

        }

    }

```

```

    }

    // default temp filename, if not entered manually

    if (*mode & (MAKE_AST | INTERACTIVE_MODE) &&
parameters[OUT_SYNTAX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\0') {

        if (*mode & INTERACTIVE_MODE) {

            system("CLS");

            fflush(stdin);

            fflush(stdout);

            fflush(stderr);

        }

        strncpy(parameters[OUT_SYNTAX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp, PATH_NAME_LENGTH);

        strncat(parameters[OUT_SYNTAX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER], "_syntax_error.txt", PATH_NAME_LENGTH
- strlen(parameters[OUT_SYNTAX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER]));

        printf("Out AST filename not setted. Used defaule input filename \"%s\"\r\n",
parameters[OUT_SYNTAX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER]);

        if (*mode & INTERACTIVE_MODE) {

            printf("Press Enter to next step");

            (void)getchar();

            //(void)getchar();

        }

    }

    // default temp filename, if not entered manually

    if (*mode & ((MAKE_C | MAKE_ASSEMBLY | MAKE_OBJECT | MAKE_BINARY) | INTERACTIVE_MODE) &&
parameters[OUT_SEMANTIX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\0') {

        if (*mode & INTERACTIVE_MODE) {

            system("CLS");

            fflush(stdin);

            fflush(stdout);

            fflush(stderr);

        }

        strncpy(parameters[OUT_SEMANTIX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp, PATH_NAME_LENGTH);

        strncat(parameters[OUT_SEMANTIX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER], "_semantix_error.txt",
PATH_NAME_LENGTH - strlen(parameters[OUT_SEMANTIX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER]));

        printf("Out AST filename not setted. Used defaule input filename \"%s\"\r\n",
parameters[OUT_SEMANTIX_ERROR_FILENAME_WITH_EXTENSION_PARAMETER]);

        if (*mode & INTERACTIVE_MODE) {

            printf("Press Enter to next step");

            (void)getchar();

            //(void)getchar();

        }

    }

    // default temp filename, if not entered manually

```

```

        if (*mode & (MAKE_LEXEMES_SEQUENSE | INTERACTIVE_MODE) &&
parameters[OUT_PREPARED_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\0') {

            if (*mode & INTERACTIVE_MODE) {

                system("CLS");

                fflush(stdin);

                fflush(stdout);

                fflush(stderr);

            }

            strncpy(parameters[OUT_PREPARED_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp,
PATH_NAME_LENGTH);

            strncpy(parameters[OUT_PREPARED_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER], "_prepared_lexemes.txt",
PATH_NAME_LENGTH - strlen(parameters[OUT_PREPARED_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER]));

            printf("Out lexemes sequense filename not setted. Used defaule input filename \"%s\\r\\n",
parameters[OUT_PREPARED_LEXEMES_SEQUENSE_FILENAME_WITH_EXTENSION_PARAMETER]);

            if (*mode & INTERACTIVE_MODE) {

                printf("Press Enter to next step");

                (void)getchar();

                //(void)getchar();

            }

        }

// default temp filename, if not entered manually

if (*mode & (MAKE_C | INTERACTIVE_MODE) && parameters[OUT_C_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\0') {

    if (*mode & INTERACTIVE_MODE) {

        system("CLS");

        fflush(stdin);

        fflush(stdout);

        fflush(stderr);

    }

    strncpy(parameters[OUT_C_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp, PATH_NAME_LENGTH);

    strncpy(parameters[OUT_C_FILENAME_WITH_EXTENSION_PARAMETER], ".c", PATH_NAME_LENGTH -
strlen(parameters[OUT_C_FILENAME_WITH_EXTENSION_PARAMETER]));

    printf("Out C filename not setted. Used defaule input filename \"%s\\r\\n",
parameters[OUT_C_FILENAME_WITH_EXTENSION_PARAMETER]);

    if (*mode & INTERACTIVE_MODE) {

        printf("Press Enter to next step");

        (void)getchar();

        //(void)getchar();

    }

}

// default temp filename, if not entered manually

if ((*mode & (MAKE_ASSEMBLY | INTERACTIVE_MODE)) &&
parameters[OUT_ASSEMBLY_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\0') {

    if (*mode & INTERACTIVE_MODE) {

        system("CLS");

```

```

        fflush(stdin);

        fflush(stdout);

        fflush(stderr);

    }

    strncpy(parameters[OUT_ASSEMBLY_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp, PATH_NAME_LENGTH);

    strcat(parameters[OUT_ASSEMBLY_FILENAME_WITH_EXTENSION_PARAMETER], ".asm", PATH_NAME_LENGTH -
strlen(parameters[OUT_ASSEMBLY_FILENAME_WITH_EXTENSION_PARAMETER]));

    printf("Out assembly filename not setted. Used defaule input filename \"%s\"\\r\\n",
parameters[OUT_ASSEMBLY_FILENAME_WITH_EXTENSION_PARAMETER]);

    if (*mode & INTERACTIVE_MODE) {

        printf("Press Enter to next step");

        (void)getchar();

    }

}

// default input filename, if not entered manually

if (*mode & (MAKE_OBJECT | INTERACTIVE_MODE) &&
parameters[OUT_OBJECT_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\\0') {

    if (*mode & INTERACTIVE_MODE) {

        system("CLS");

        fflush(stdin);

        fflush(stdout);

        fflush(stderr);

    }

    strncpy(parameters[OUT_OBJECT_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp, PATH_NAME_LENGTH);

    strcat(parameters[OUT_OBJECT_FILENAME_WITH_EXTENSION_PARAMETER], ".obj", PATH_NAME_LENGTH -
strlen(parameters[OUT_OBJECT_FILENAME_WITH_EXTENSION_PARAMETER]));

    printf("Out object filename not setted. Used defaule input filename \"%s\"\\r\\n",
parameters[OUT_OBJECT_FILENAME_WITH_EXTENSION_PARAMETER]);

    if (*mode & INTERACTIVE_MODE) {

        printf("Press Enter to next step");

        (void)getchar();

    }

}

// default input filename, if not entered manually

if (*mode & (MAKE_BINARY | INTERACTIVE_MODE) &&
parameters[OUT_BINARY_FILENAME_WITH_EXTENSION_PARAMETER][0] == '\\0') {

    if (*mode & INTERACTIVE_MODE) {

        system("CLS");

        fflush(stdin);

        fflush(stdout);

        fflush(stderr);

    }

    strncpy(parameters[OUT_BINARY_FILENAME_WITH_EXTENSION_PARAMETER], nameTemp, PATH_NAME_LENGTH);

```

```

        strcat(parameters[OUT_BINARY_FILENAME_WITH_EXTENSION_PARAMETER], ".exe", PATH_NAME_LENGTH -
strlen(parameters[OUT_BINARY_FILENAME_WITH_EXTENSION_PARAMETER]));

        printf("Out binary filename not setted. Used defaule input filename \"%s\"\\r\\n",
parameters[OUT_BINARY_FILENAME_WITH_EXTENSION_PARAMETER]);

        if (*mode & INTERACTIVE_MODE) {

            printf("Press Enter to next step");

            (void) getchar();

        }

    }

    return;

}

```

// after using this function use free(void \*) function to release text buffer

```

size_t loadSource(char** text, char* fileName) {

    if (!fileName) {

        printf("No input file name\\r\\n");

        return 0;

    }

    FILE* file = fopen(fileName, "rb");

    if (file == NULL) {

        printf("File not loaded\\r\\n");

        return 0;

    }

    fseek(file, 0, SEEK_END);

    long fileSize_ = ftell(file);

    if (fileSize_ >= MAX_TEXT_SIZE) {

        printf("the file(%ld bytes) is larger than %d bytes\\r\\n", fileSize_, MAX_TEXT_SIZE);

        fclose(file);

        exit(2); // TODO: ...

        //return 0;

    }

    size_t fileSize = fileSize_;

    rewind(file);

    if (!text) {

        printf("Load source error\\r\\n");

        return 0;

    }

    *text = (char*) malloc(sizeof(char) * (fileSize + 1));

```

```

        if (*text == NULL) {

            fputs("Memory error", stderr);

            fclose(file);

            exit(2); // TODO: ...

            //return 0;

        }

        size_t result = fread(*text, sizeof(char), fileSize, file);

        if (result != fileSize) {

            fputs("Reading error", stderr);

            fclose(file);

            exit(3); // TODO: ...

            //return 0;

        }

        (*text)[fileSize] = '\0';

        fclose(file);

        return fileSize;

    }

}

div.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: div.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_DIV);

    if (multitokenSize) {

        #ifdef DEBUG_MODE_BY_ASSEMBLY

            printf("\r\n");

            printf("    ;\"%s\"\\r\n", tokenStruct[MULTI_TOKEN_DIV][0]);

        #endif

        const unsigned char code__mov_eax_stackTopByECXMinus4[] = { 0x8B, 0x41, 0xFC };

        const unsigned char code__cdq[] = { 0x99 };

```



```

const unsigned char code__idiv_stackTopByECX[] = { 0xF7, 0x39 };

const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };

const unsigned char code__mov_toAddrFromECX_eax[] = { 0x89, 0x01 };


currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECXMinus4, 3);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cdq, 1);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__idiv_stackTopByECX, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_toAddrFromECX_eax, 2);


#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("  mov eax, dword ptr[ecx - 4]\r\n");

    printf("  cdq\r\n");

    printf("  idiv dword ptr [ecx]\r\n");

    printf("  sub ecx, 4\r\n");

    printf("  mov dword ptr [ecx], eax\r\n");

#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

Else.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*          file: else.cpp          *

*          (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

#include "string.h"

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_ELSE);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("  ;\"%s\"\r\n", tokenStruct[MULTI_TOKEN_ELSE][0]);


```

```

#endif

const unsigned char code__cmp_eax_0[] = { 0x83, 0xF8, 0x00 };

const unsigned char code__jnz_offset[] = { 0x0F, 0x85, 0x00, 0x00, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_eax_0, 3);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jnz_offset, 6);

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = **lastLexemInfoInTable;
lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue = (unsigned long long int)(currBytePtr - 4);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("    cmp eax, 0\r\n");

    printf("    jnz LABEL@AFTER_ELSE_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);
#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;
}

return currBytePtr;
}

unsigned char* makePostElseCode_(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    *(unsigned int*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue = (unsigned int)(currBytePtr -
(unsigned char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue - 4);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("    LABEL@AFTER_ELSE_%016lX:\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);
#endif

    return currBytePtr;
}

unsigned char* makeSemicolonAfterElseCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) { // Or Ender!
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_SEMICOLON);

    if (multitokenSize
    &&
    lexemInfoTransformationTempStackSize
    &&
    !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI_TOKEN_ELSE][0],
MAX_LEXEM_SIZE)
    ) {

```

```

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("\r\n");

    printf("    ;\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI_TOKEN_SEMICOLON][0], tokenStruct[MULTI_TOKEN_ELSE][0]);

#endif

    currBytePtr = makePostElseCode_(lastLexemInfoInTable, currBytePtr, generatorMode);

    --lexemInfoTransformationTempStackSize;

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

Equal.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*          file: equal.cpp          *

*          (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

unsigned char* makeIsEqualCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_EQUAL);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\"%s\" \r\n", tokenStruct[MULTI_TOKEN_EQUAL][0]);

#endif

        const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

        const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };

        const unsigned char code__cmp_stackTopByECX_eax[] = { 0x39, 0x01 };

        const unsigned char code__sete_al[] = { 0x0F, 0x94, 0xC0 };

        const unsigned char code__and_eax_1[] = { 0x83, 0xE0, 0x01 };

        const unsigned char code__mov_stackTopByECX_eax[] = { 0x89, 0x01 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);

```

```

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_stackTopByECX_eax, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sete_al, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_eax_1, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_stackTopByECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("  mov eax, dword ptr[ecx]\r\n");

    printf("  sub ecx, 4\r\n");

    printf("  cmp dword ptr[ecx], eax\r\n");

    printf("  sete al\r\n");

    printf("  and eax, 1\r\n");

    printf("  mov dword ptr[ecx], eax\r\n");

#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

For.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*          file: for.cpp          *

*          (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

#include "string.h"

unsigned char* makeForCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_FOR);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("  ;\r\n", tokenStruct[MULTI_TOKEN_FOR][0]);

#endif

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = *lastLexemInfoInTable;

```

```

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;
    }

    return currBytePtr;
}

unsigned char* makeToOrDowntoCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) { // TODO: add
assemblyBytePtr

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_DOWNTO);

    bool toMode = false;

    if (!multitokenSize) {

        toMode = !(multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_TO));

    }

    if (multitokenSize

        &&

        lexemInfoTransformationTempStackSize

        &&

        !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI_TOKEN_FOR][0],
MAX_LEXEM_SIZE)

    ) {

        if (toMode) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

            printf("\r\n");

            printf("    ;\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI_TOKEN_TO][0], tokenStruct[MULTI_TOKEN_FOR][0]);

#endif

        }

        else {

#ifdef DEBUG_MODE_BY_ASSEMBLY

            printf("\r\n");

            printf("    ;\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI_TOKEN_DOWNTO][0], tokenStruct[MULTI_TOKEN_FOR][0]);

#endif

        }

        const unsigned char code__dec_addrFromEBX[] = { 0xFF, 0x0B }; // dec dword ptr [ebx] // init
        const unsigned char code__inc_addrFromEBX[] = { 0xFF, 0x03 }; // inc dword ptr [ebx] // init
        const unsigned char code__push_ebx[] = { 0x53 }; // push ebx

        if (toMode) {

            currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__dec_addrFromEBX, 2); // init

        }

        else {

            currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__inc_addrFromEBX, 2); // init

        }

```

```

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_ebx, 1);

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = **lastLexemInfoInTable;
lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue = (unsigned long long int)currBytePtr;

#ifdef DEBUG_MODE_BY_ASSEMBLY
    if (toMode) {
        printf("  dec dword ptr [ebx]\r\n"); // start from (index - 1)
    }
    else {
        printf("  inc dword ptr [ebx]\r\n"); // start from (index + 1)
    }
    printf("  push ebx\r\n");
    if (toMode) {
        printf("  LABEL@AFTER_TO_%016lX:\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);
    }
    else {
        printf("  LABEL@AFTER_DOWNT0_%016lX:\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);
    }
#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;
}

return currBytePtr;
}

unsigned char* makeDoCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_DO);
    if (multitokenSize) {
        bool toMode = false;
        if (lexemInfoTransformationTempStackSize && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
1].lexemStr, tokenStruct[MULTI_TOKEN_TO][0], MAX_LEXEM_SIZE)) {
            toMode = true;
        }
        else if (lexemInfoTransformationTempStackSize < 2
            || strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_DOWNT0][0], MAX_LEXEM_SIZE)
            || strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_FOR][0], MAX_LEXEM_SIZE)
        ) {
            return currBytePtr;
        }
    }
}

```

```

        if (toMode) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

                printf("\r\n");

                printf("    ;\"%s\" (after \"%s\" after \"%s\")\r\n", tokenStruct[MULTI_TOKEN_DO][0], tokenStruct[MULTI_TOKEN_TO][0],
tokenStruct[MULTI_TOKEN_FOR][0]);

#endif

        }

        else {

#ifdef DEBUG_MODE_BY_ASSEMBLY

                printf("\r\n");

                printf("    ;\"%s\" (after \"%s\" after \"%s\")\r\n", tokenStruct[MULTI_TOKEN_DO][0],
tokenStruct[MULTI_TOKEN_DOWNT0][0], tokenStruct[MULTI_TOKEN_FOR][0]);

#endif

        }


const unsigned char code__mov_ebx_addrFromESP[] = { 0x8B, 0x1C, 0x24 };           // mov ebx, dword ptr [esp]
const unsigned char code__cmp_addrFromEBX_eax[] = { 0x39, 0x03 };           // cmp dword ptr [ebx], eax
const unsigned char code__jge_offset[]      = { 0x0F, 0x8D, 0x00, 0x00, 0x00, 0x00 }; // jge ?? ?? ?? ??
const unsigned char code__jle_offset[]      = { 0x0F, 0x8E, 0x00, 0x00, 0x00, 0x00 }; // jle ?? ?? ?? ??
const unsigned char code__inc_addrFromEBX[]  = { 0xFF, 0x03 };           // inc dword ptr [ebx]
const unsigned char code__dec_addrFromEBX[]  = { 0xFF, 0x0B };           // dec dword ptr [ebx]


currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_ebx_addrFromESP, 3);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_addrFromEBX_eax, 2);
if (toMode) {

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jge_offset, 6);

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue = (unsigned long long int)(currBytePtr -
4);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__inc_addrFromEBX, 2);

}

else {

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jle_offset, 6);

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue = (unsigned long long int)(currBytePtr -
4);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__dec_addrFromEBX, 2);

}


#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("    mov ebx, dword ptr [esp]\r\n");
        printf("    cmp dword ptr [ebx], eax\r\n");

        if (toMode) {

                printf("    jge LABEL@EXIT_FOR_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

                printf("    inc dword ptr [ebx]\r\n");

        }


```

```

        else {

            printf("    jle LABEL@EXIT_FOR_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

            printf("    dec dword ptr [ebx]\r\n");

        }

#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    return currBytePtr;

}

unsigned char* makePostForCode_(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode, bool toMode) {

    const unsigned char code__jmp_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 };

    const unsigned char code__add_esp_4[] = { 0x83, 0xC4, 0x04 };

    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jmp_offset, 5);

    *((unsigned int*)(currBytePtr - 4)) = (unsigned int)((unsigned char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize
- 1].ifvalue - currBytePtr);

    *((unsigned int*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue) = (unsigned int)(currBytePtr -
(unsigned char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue - 4);

    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_esp_4, 3);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    if (toMode) {

        printf("    jmp LABEL@AFTER_TO_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

    }

    else {

        printf("    jmp LABEL@AFTER_DOWNT0_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

    }

    printf("    LABEL@EXIT_FOR_%016lX:\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

    printf("    add esp, 4; add esp, 8\r\n");

#endif

    return currBytePtr;

}

unsigned char* makeSemicolonAfterForCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) { // Or
Ender!

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_SEMICOLON);

    bool toMode = false;

    if (multitokenSize

```



```

        &&

        lexemInfoTransformationTempStackSize > 1

        &&

        !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI_TOKEN_FOR][0],
MAX_LEXEM_SIZE)

        && (

                !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_DOWNT0][0], MAX_LEXEM_SIZE)

                ||

                (toMode = !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_TO][0], MAX_LEXEM_SIZE))

                )

        ) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI_TOKEN_SEMICOLON][0], tokenStruct[MULTI_TOKEN_FOR][0]);

#endif

```

```

currBytePtr = makePostForCode_(lastLexemInfoInTable, currBytePtr, generatorMode, toMode);

```

```

lexemInfoTransformationTempStackSize -= 2;

return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

```

```

return currBytePtr;

```

```

}

```

Generator.cpp

```

#define _CRT_SECURE_NO_WARNINGS

```

```

// TODO: CHANGE BY fRESET() TO END

```

```

/*****

```

```

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

```

```

*           file: generator.cpp           *

```

```

*           (draft!) *

```

```

*****/

```

```

// #define IDENTIFIER_LEXEME_TYPE 2

```

```

// #define VALUE_LEXEME_TYPE 4

```

```

// #define VALUE_SIZE 4

```

```

#ifndef __cplusplus

```

```

#define bool int

```

```

#define false 0

```

```

#define true 1

```

```

#endif

```

```

#include "../././src/include/def.h"

#include "../././src/include/config.h"

#include "../././src/include/generator/generator.h"

#include "../././src/include/lexica/lexica.h"

#include "../././src/include/syntax/syntax.h"

#include "../././src/include/semantix/semantix.h"


#include <stdio.h>

#include <stdlib.h>

#include <string.h>


//#define DEBUG_MODE_BY_ASSEMBLY

//#define C_CODER_MODE          0x01

//#define ASSEMBLY_X86_WIN32_CODER_MODE 0x02

//#define OBJECT_X86_WIN32_CODER_MODE 0x04

//#define MACHINE_CODER_MODE    0x08

//

//unsigned char generatorMode = MACHINE_CODER_MODE;


#define MAX_TEXT_SIZE 8192

#define MAX_GENERATED_TEXT_SIZE (MAX_TEXT_SIZE * 6)

#define GENERATED_TEXT_SIZE_ 32768

//#define GENERATED_TEXT_SIZE (MAX_TEXT_SIZE % MAX_GENERATED_TEXT_SIZE) // ?


#define SUCCESS_STATE 0


#define MAX_OUTTEXT_SIZE (8*8192*1024)

unsigned char outText[MAX_OUTTEXT_SIZE] = ""; // !!!

#define MAX_TEXT_SIZE 8192

#define MAX_WORD_COUNT (MAX_TEXT_SIZE / 5)

#define MAX_LEXEM_SIZE 1024


#if 0


#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;

unsigned long long int dataOffsetMinusCodeOffset = 0x00004000;


//unsigned long long int codeOffset = 0x000004AF;

//unsigned long long int baseOperationOffset = codeOffset + 49;// 0x00000031;

unsigned long long int baseOperationOffset = 0x000004AF;

unsigned long long int putProcOffset = 0x0000001B;

unsigned long long int getProcOffset = 0x00000044;


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


#endif


struct LabelOffsetInfo {

    char labelStr[MAX_LEXEM_SIZE];

    unsigned char* labelBytePtr;

    // TODO: ...

};

struct LabelOffsetInfo labelsOffsetInfoTable[MAX_WORD_COUNT] = { { "", NULL/*, 0, 0*/ } };

struct LabelOffsetInfo* lastLabelOffsetInfoInTable = labelsOffsetInfoTable; // first for begin


struct GotoPositionInfo { // TODO: by Index

    char labelStr[MAX_LEXEM_SIZE];

    unsigned char* gotoInstructionPositionPtr;

    // TODO: ...

};

struct GotoPositionInfo gotoPositionsInfoTable[MAX_WORD_COUNT] = { { "", NULL/*, 0, 0*/ } }; // TODO: by Index

struct GotoPositionInfo* lastGotoPositionInfoInTable = gotoPositionsInfoTable; // first for begin


////////////////////////////////////

//#include "src/include/generator/generator.h"


//unsigned char generatorMode = MACHINE_CODER_MODE;


char* tokenStruct[MAX_TOKEN_STRUCT_ELEMENT_COUNT][MAX_TOKEN_STRUCT_ELEMENT_PART_COUNT] = { NULL };


#if 0

static void intitTokenStruct__OLD() {

    //SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, MULTI_TOKEN_BITWISE_NOT, ("~"), (""), (""), (""))

```

```

//
//      a12345_ptr = a12345;
//

tokenStruct[MULTI_TOKEN_BITWISE_NOT][0] = (char*)"~";
tokenStruct[MULTI_TOKEN_BITWISE_AND][0] = (char*)"&";
tokenStruct[MULTI_TOKEN_BITWISE_OR][0] = (char*)"|";
tokenStruct[MULTI_TOKEN_NOT][0] = (char*)"NOT";
tokenStruct[MULTI_TOKEN_AND][0] = (char*)"AND";
tokenStruct[MULTI_TOKEN_OR][0] = (char*)"OR";

tokenStruct[MULTI_TOKEN_EQUAL][0] = (char*)"==";
tokenStruct[MULTI_TOKEN_NOT_EQUAL][0] = (char*)"!=";
tokenStruct[MULTI_TOKEN_LESS][0] = (char*)"<";
tokenStruct[MULTI_TOKEN_GREATER][0] = (char*)">";
tokenStruct[MULTI_TOKEN_LESS_OR_EQUAL][0] = (char*)"<=";
tokenStruct[MULTI_TOKEN_GREATER_OR_EQUAL][0] = (char*)">=";

tokenStruct[MULTI_TOKEN_ADD][0] = (char*)"+";
tokenStruct[MULTI_TOKEN_SUB][0] = (char*)" - ";
tokenStruct[MULTI_TOKEN_MUL][0] = (char*)" * ";
tokenStruct[MULTI_TOKEN_DIV][0] = (char*)"DIV";
tokenStruct[MULTI_TOKEN_MOD][0] = (char*)"MOD";

tokenStruct[MULTI_TOKEN_BIND_RIGHT_TO_LEFT][0] = (char*)"<<";
tokenStruct[MULTI_TOKEN_BIND_LEFT_TO_RIGHT][0] = (char*)">>";

tokenStruct[MULTI_TOKEN_COLON][0] = (char*)":";
tokenStruct[MULTI_TOKEN_GOTO][0] = (char*)"GOTO";

tokenStruct[MULTI_TOKEN_IF][0] = (char*)"IF"; tokenStruct[MULTI_TOKEN_IF][1] = (char*)"(";
//      tokenStruct[MULTI_TOKEN_IF_][0] = (char*)"IF"; // don't change this!
tokenStruct[MULTI_TOKEN_THEN][0] = (char*)" ";
//      tokenStruct[MULTI_TOKEN_THEN_][0] = (char*)"NULL"; tokenStruct[MULTI_TOKEN_IF][1] = (char*)"STATEMENT"; // don't
change this!

tokenStruct[MULTI_TOKEN_ELSE][0] = (char*)"ELSE";

tokenStruct[MULTI_TOKEN_FOR][0] = (char*)"FOR";
tokenStruct[MULTI_TOKEN_TO][0] = (char*)"TO";
tokenStruct[MULTI_TOKEN_DOWNTONTO][0] = (char*)"DOWNTONTO";
tokenStruct[MULTI_TOKEN_DO][0] = (char*)"DO"; // tokenStruct[MULTI_TOKEN_DO][1] = (char*)".";

//

```

```

tokenStruct[MULTI_TOKEN_WHILE][0] = (char*)"WHILE";

tokenStruct[MULTI_TOKEN_CONTINUE_WHILE][0] = (char*)"CONTINUE"; tokenStruct[MULTI_TOKEN_CONTINUE_WHILE][1] =
(char*)"WHILE";

tokenStruct[MULTI_TOKEN_EXIT_WHILE][0] = (char*)"EXIT"; tokenStruct[MULTI_TOKEN_EXIT_WHILE][1] = (char*)"WHILE";

tokenStruct[MULTI_TOKEN_END_WHILE][0] = (char*)"END"; tokenStruct[MULTI_TOKEN_END_WHILE][1] = (char*)"WHILE";

//

//

tokenStruct[MULTI_TOKEN_REPEAT][0] = (char*)"REPEAT";

tokenStruct[MULTI_TOKEN_UNTIL][0] = (char*)"UNTIL";

//

//

tokenStruct[MULTI_TOKEN_INPUT][0] = (char*)"GET";

tokenStruct[MULTI_TOKEN_OUTPUT][0] = (char*)"PUT";

//

//

tokenStruct[MULTI_TOKEN_RLBIND][0] = (char*)"<<";

tokenStruct[MULTI_TOKEN_LRBIND][0] = (char*)">>";

//

tokenStruct[MULTI_TOKEN_SEMICOLON][0] = (char*)" ";

tokenStruct[MULTI_TOKEN_BEGIN][0] = (char*)"BEGIN";

tokenStruct[MULTI_TOKEN_END][0] = (char*)"END";

tokenStruct[MULTI_TOKEN_NULL_STATEMENT][0] = (char*)"NULL"; tokenStruct[MULTI_TOKEN_NULL_STATEMENT][1] =
(char*)"STATEMENT";

//          NULL_STATEMENT null_statement

//          null statement

//return 0;

}

//char intitTokenStruct_ = (intitTokenStruct__OLD(), 0);

#endif

INIT_TOKEN_STRUCT_NAME(0);

unsigned char detectMultiToken(struct LexemInfo* lexemInfoTable, enum TokenStructName tokenStructName) {

    if (lexemInfoTable == NULL) {

        return false;

    }

    if (!strcmp(lexemInfoTable[0].lexemStr, tokenStruct[tokenStructName][0], MAX_LEXEM_SIZE)

```

```

        && (tokenStruct[tokenStructName][1] == NULL || tokenStruct[tokenStructName][1][0] == '\0' || !strcmp(lexemInfoTable[1].lexemStr,
tokenStruct[tokenStructName][1], MAX_LEXEM_SIZE))

        && (tokenStruct[tokenStructName][2] == NULL || tokenStruct[tokenStructName][2][0] == '\0' || !strcmp(lexemInfoTable[2].lexemStr,
tokenStruct[tokenStructName][2], MAX_LEXEM_SIZE))

        && (tokenStruct[tokenStructName][3] == NULL || tokenStruct[tokenStructName][3][0] == '\0' || !strcmp(lexemInfoTable[3].lexemStr,
tokenStruct[tokenStructName][3], MAX_LEXEM_SIZE))) {

    return !(tokenStruct[tokenStructName][0] != NULL && tokenStruct[tokenStructName][0][0] != '\0')

        + !(tokenStruct[tokenStructName][1] != NULL && tokenStruct[tokenStructName][1][0] != '\0')
        + !(tokenStruct[tokenStructName][2] != NULL && tokenStruct[tokenStructName][2][0] != '\0')
        + !(tokenStruct[tokenStructName][3] != NULL && tokenStruct[tokenStructName][3][0] != '\0')

        ;

}

else {

    return 0;

}

}

```

```

unsigned char createMultiToken(struct LexemInfo** lexemInfoTable, enum TokenStructName tokenStructName) {

    if (lexemInfoTable == NULL || *lexemInfoTable == NULL) {

        return false;

    }

    if (tokenStruct[tokenStructName][0] != NULL && tokenStruct[tokenStructName][0][0] != '\0') {

        strcpy(lexemInfoTable[0][0].lexemStr, tokenStruct[tokenStructName][0], MAX_LEXEM_SIZE);

        lexemInfoTable[0][0].lexemId = 0;

        lexemInfoTable[0][0].tokenType = 0;

        lexemInfoTable[0][0].ifvalue = 0;

        lexemInfoTable[0][0].row = ~0;

        lexemInfoTable[0][0].col = ~0;

        ++* lexemInfoTable;

    }

    else {

        return 0;

    }

    if (tokenStruct[tokenStructName][1] != NULL && tokenStruct[tokenStructName][1][0] != '\0') {

        strcpy((*lexemInfoTable)->lexemStr, tokenStruct[tokenStructName][1], MAX_LEXEM_SIZE);

        lexemInfoTable[0][0].lexemId = 0;

        lexemInfoTable[0][0].tokenType = 0;

        lexemInfoTable[0][0].ifvalue = 0;

        lexemInfoTable[0][0].row = ~0;

        lexemInfoTable[0][0].col = ~0;

        ++* lexemInfoTable;

    }
}

```

```

    }

    else {

        return 1;

    }

    if (tokenStruct[tokenStructName][2] != NULL && tokenStruct[tokenStructName][2][0] != '\0') {

        strncpy((*lexemInfoTable)->lexemStr, tokenStruct[tokenStructName][2], MAX_LEXEM_SIZE);

        lexemInfoTable[0][0].lexemId = 0;

        lexemInfoTable[0][0].tokenType = 0;

        lexemInfoTable[0][0].ifvalue = 0;

        lexemInfoTable[0][0].row = ~0;

        lexemInfoTable[0][0].col = ~0;

        ++* lexemInfoTable;

    }

    else {

        return 2;

    }

    if (tokenStruct[tokenStructName][3] != NULL && tokenStruct[tokenStructName][3][0] != '\0') {

        strncpy((*lexemInfoTable)->lexemStr, tokenStruct[tokenStructName][3], MAX_LEXEM_SIZE);

        lexemInfoTable[0][0].lexemId = 0;

        lexemInfoTable[0][0].tokenType = 0;

        lexemInfoTable[0][0].ifvalue = 0;

        lexemInfoTable[0][0].row = ~0;

        lexemInfoTable[0][0].col = ~0;

        ++* lexemInfoTable;

    }

    else {

        return 3;

    }

    return 4;

}

```

```

// #define MAX_ACCESSORY_STACK_SIZE 128

```

```

struct NonContainedLexemInfo lexemInfoTransformationTempStack[MAX_ACCESSORY_STACK_SIZE];

```

```

unsigned long long int lexemInfoTransformationTempStackSize = 0;

```

```

//

```

```

unsigned long long int getVariableOffset(char* identifierStr) {

```

```

    for (unsigned long long int index = 0; identifierIdsTable[index][0] != '\0'; ++index) {

```

```

        if (!strcmp(identifierIdsTable[index], identifierStr, MAX_LEXEM_SIZE)) {

```

```

        return START_DATA_OFFSET + sizeof(CODEGEN_DATA_TYPE) * index;

    }

}

return OUT_DATA_OFFSET;

}

unsigned char* outBytes2Code(unsigned char* currBytePtr, unsigned char* fragmentFirstBytePtr, unsigned long long int bytesCout) {

    for (; bytesCout--; *currBytePtr++ = *fragmentFirstBytePtr++);

    return currBytePtr;

}

unsigned char* makeEndProgramCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

    const unsigned char code__xor_eax_eax[] = { 0x33, 0xC0 };
    const unsigned char code__ret[] = { 0xC3 };

    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__xor_eax_eax, 2);
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__ret, 1);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("\r\n");
    //printf("imul ebp, 4\r\n");
    //printf("add esp, ebp\r\n");
    //printf("xor ebp, ebp;\r\n");

    printf("    xor eax, eax\r\n");
    printf("    ret\r\n");

    printf("\r\n\r\n");

    printf("end start\r\n");

    printf("\r\n\r\n");

#endif

    return currBytePtr;

}

unsigned char* makeTitle(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf(".686\r\n");


```



```

        printf(".model flat, stdcall\r\n");

        printf("option casemap : none\r\n");

#ifdefif

        return currBytePtr;

}

unsigned char* makeDependenciesDeclaration(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

#ifdefif DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("GetStdHandle proto STDCALL, nStdHandle : DWORD\r\n");

        printf("ExitProcess proto STDCALL, uExitCode : DWORD\r\n");

        printf(";MessageBoxA PROTO hwnd : DWORD, lpText : DWORD, lpCaption : DWORD, uType : DWORD\r\n");

        printf("ReadConsoleA proto STDCALL, hConsoleInput : DWORD, lpBuffer : DWORD, nNumberOfCharsToRead : DWORD,
lpNumberOfCharsRead : DWORD, lpReserved : DWORD\r\n");

        printf("WriteConsoleA proto STDCALL, hConsoleOutput : DWORD, lpBuffert : DWORD, nNumberOfCharsToWrite : DWORD,
lpNumberOfCharsWritten : DWORD, lpReserved : DWORD\r\n");

        printf("wsprintfA PROTO C : VARARG\r\n");

        printf("\r\n");

        printf("GetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, lpMode : DWORD\r\n");

        printf("\r\n");

        printf("SetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, dwMode : DWORD\r\n");

        printf("\r\n");

        printf("ENABLE_LINE_INPUT EQU 0002h\r\n");

        printf("ENABLE_ECHO_INPUT EQU 0004h\r\n");

#ifdefif

        return currBytePtr;

}

unsigned char* makeDataSection(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

#ifdefif DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf(".data\r\n");

        printf("    data_start db 8192 dup (0)\r\n");

        printf("    ;title_msg db \"Output:\", 0\r\n");

        printf("    valueTemp_msg db 256 dup(0)\r\n");

        printf("    valueTemp_fmt db \"%%d\", 10, 13, 0\r\n");

        printf("    ;NumberOfCharsWritten dd 0\r\n");

        printf("    hConsoleInput dd 0\r\n");

        printf("    hConsoleOutput dd 0\r\n");

        printf("    buffer db 128 dup(0)\r\n");

```

```

        printf("  readOutCount dd ?\r\n");

#endif

        return currBytePtr;

}

unsigned char* makeBeginProgramCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf(".code\r\n");

        printf("start:\r\n");

#endif

        return currBytePtr;

}

unsigned char* makeInitCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

        //      unsigned char code__call_NexInstructionLabel[]      = { 0xE8, 0x00, 0x00, 0x00, 0x00 };

        //

        //      unsigned char code__pop_esi[]                        = { 0x5E };

        //      unsigned char code__sub_esi_5[]                      = { 0x83, 0xEE, 0x05 };

        //      unsigned char code__mov_edi_esi[]                    = { 0x8B, 0xFE };

        //      unsigned char code__add_edi_dataOffsetMinusCodeOffset[] = { 0xE8, 0xC7, 0x00, 0x00, 0x00, 0x00 };

        //      //unsigned char code__xor_ebp_ebp[]                  = { 0x33, 0xED };

        //      unsigned char code__mov_ecx_edi[]                     = { 0x8B, 0xCF };

        //      unsigned char code__add_ecx_512[]                     = { 0x81, 0xC1, 0x00, 0x02, 0x00, 0x00 };

        //      unsigned char code__jmp_initConsole[] = { 0xEB, 0x7C };

        //

        //      currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__call_NexInstructionLabel, 5);

        //      currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__pop_esi, 1);

        //      currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_esi_5, 3);

        //      currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_edi_esi, 2);

        //      currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_edi_dataOffsetMinusCodeOffset, 6);

        //      *(unsigned int*)(currBytePtr - 4) = dataOffsetMinusCodeOffset;

        //      //currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__xor_ebp_ebp, 2);

        //      currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_ecx_edi, 2);

        //      currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_ecx_512, 6);

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("  db 0E8h, 00h, 00h, 00h, 00h; call NexInstruction\r\n");

        printf(";NexInstruction:\r\n");

        printf("  pop esi\r\n");

#endif

```

```

printf("  sub esi, 5\r\n");

printf("  mov edi, esi\r\n");//printf("  mov edi, offset data_start\r\n");

printf("  add edi, 0%08Xh\r\n", (int)dataOffsetMinusCodeOffset);

//printf("  xor ebp, ebp\r\n");

printf("  mov ecx, edi\r\n");

printf("  add ecx, 512\r\n");

printf("  jmp initConsole\r\n");

```

```

printf("  putProc PROC\r\n");

printf("    push eax\r\n");

printf("    push offset valueTemp_fmt\r\n");

printf("    push offset valueTemp_msg\r\n");

printf("    call wsprintfA\r\n");

printf("    add esp, 12\r\n");

printf("\r\n");

printf("    ;push 40h\r\n");

printf("    ;push offset title_msg\r\n");

printf("    ;push offset valueTemp_msg\r\n");

printf("    ;push 0\r\n");

printf("    ;call MessageBoxA\r\n");

printf("\r\n");

printf("    push 0\r\n");

printf("    push 0; offset NumberOfCharsWritten\r\n");

printf("    push eax; NumberOfCharsToWrite\r\n");

printf("    push offset valueTemp_msg\r\n");

printf("    push hConsoleOutput\r\n");

printf("    call WriteConsoleA\r\n");

printf("\r\n");

printf("    ret\r\n");

printf("  putProc ENDP\r\n");

```

```

printf("\r\n\r\n");

```

```

//printf("  getProc PROC\r\n");

//printf("    push eax\r\n");

//printf("    push offset valueTemp_fmt\r\n");

//printf("    push offset valueTemp_msg\r\n");

//printf("    call wsprintfA\r\n");

//printf("    add esp, 12\r\n");

//printf("\r\n");

//printf("    push 40h\r\n");

//printf("    push offset title_msg\r\n");

//printf("    push offset valueTemp_msg\r\n");

```

```

//printf("    push 0\r\n");

//printf("    call MessageBoxA\r\n");

//printf("\r\n");

//printf("    ret\r\n");

//printf("    getProc ENDP\r\n");


printf("    getProc PROC\r\n");

printf("    push ebp\r\n");

printf("    mov ebp, esp\r\n");

printf("\r\n");

printf("    push 0\r\n");

printf("    push offset readOutCount\r\n");

printf("    push 15\r\n");

printf("    push offset buffer + 1\r\n");

printf("    push hConsoleInput\r\n");

printf("    call ReadConsoleA\r\n");

printf("\r\n");

printf("    lea esi, offset buffer\r\n");

printf("    add esi, readOutCount\r\n");

printf("    sub esi, 2\r\n");

printf("    call string_to_int\r\n");

printf("\r\n");

printf("    mov esp, ebp\r\n");

printf("    pop ebp\r\n");

printf("    ret\r\n");

printf("    getProc ENDP\r\n");


printf("\r\n");


printf("    string_to_int PROC\r\n");

printf("    ; input: ESI - string\r\n");

printf("    ; output: EAX - value\r\n");

printf("    xor eax, eax\r\n");

printf("    mov ebx, 1\r\n");

printf("    xor ecx, ecx\r\n");

printf("\r\n");

printf("convert_loop :\r\n");

printf("    movzx ecx, byte ptr[esi]\r\n");

printf("    test ecx, ecx\r\n");

printf("    jz done\r\n");

printf("    sub ecx, '0'\r\n");

printf("    imul ecx, ebx\r\n");

printf("    add eax, ecx\r\n");

```

```

printf("    imul ebx, ebx, 10\r\n");

printf("    dec esi\r\n");

printf("    jmp convert_loop\r\n");

printf("\r\n");

printf("done:\r\n");

printf("    ret\r\n");

printf("    string_to_int ENDP\r\n");


printf("\r\n");


printf("    initConsole:\r\n");

printf("    push -10\r\n");

printf("    call GetStdHandle\r\n");

printf("    mov hConsoleInput, eax\r\n");

printf("    push -11\r\n");

printf("    call GetStdHandle\r\n");

printf("    mov hConsoleOutput, eax\r\n");

printf("    \r\n");

printf("    ;push ecx\r\n");

printf("    ;push ebx\r\n");

printf("    ;push esi\r\n");

printf("    ;push edi\r\n");

printf("    ;push offset mode\r\n");

printf("    ;push hConsoleInput\r\n");

printf("    ;call GetConsoleMode\r\n");

printf("    ;mov ebx, eax\r\n");

printf("    ;or ebx, ENABLE_LINE_INPUT \r\n");

printf("    ;or ebx, ENABLE_ECHO_INPUT\r\n");

printf("    ;push ebx\r\n");

printf("    ;push hConsoleInput\r\n");

printf("    ;call SetConsoleMode\r\n");

printf("    ;pop edi\r\n");

printf("    ;pop esi\r\n");

printf("    ;pop ebx\r\n");

printf("    ;pop ecx\r\n");

```

```

#endif

```

```

    return currBytePtr;

```

```

}

```

```

//

```

```

#include ".././src/include/preparer/preparer.h"

```

```

//

//

#include "../src/include/generator/not.h"

#include "../src/include/generator/and.h"

#include "../src/include/generator/or.h"

//

#include "../src/include/generator/add.h"

#include "../src/include/generator/sub.h"

#include "../src/include/generator/mul.h"

#include "../src/include/generator/div.h"

#include "../src/include/generator/mod.h"

//

#include "../src/include/generator/null_statement.h"

#include "../src/include/generator/operand.h"

#include "../src/include/generator/input.h"

#include "../src/include/generator/output.h"

#include "../src/include/generator/equal.h"

#include "../src/include/generator/not_equal.h"

#include "../src/include/generator/less_or_equal.h"

#include "../src/include/generator/greater_or_equal.h"

#include "../src/include/generator/rbind.h"

#include "../src/include/generator/goto_label.h"

#include "../src/include/generator/if_then.h"

#include "../src/include/generator/else.h"

#include "../src/include/generator/for.h"

#include "../src/include/generator/while.h"

#include "../src/include/generator/repeat_until.h"

//

#include "../src/include/generator/semicolon.h"

//

unsigned char* initMake(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

    //return currBytePtr;

    unsigned long long int lastDataSectionLexemIndex = getDataSectionLastLexemIndex(*lastLexemInfoInTable, &grammar);

    if(lastDataSectionLexemIndex == ~0) {

        printf("Error: bad section!\r\n");

        exit(0);

    }

    *lastLexemInfoInTable += lastDataSectionLexemIndex;

    return currBytePtr;

}

```

```

unsigned char* makeSaveHWStack(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

    const unsigned char code__mov_ebp_esp[] = { 0x8B, 0xEC };

    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_ebp_esp, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("\r\n");

    printf("    ;hw stack save(save esp)\r\n");

    printf("    mov ebp, esp\r\n");

#endif

    return currBytePtr;

}

unsigned char* makeResetHWStack(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

    const unsigned char code__mov_esp_ebp[] = { 0x8B, 0xE5 };

    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_esp_ebp, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("\r\n");

    printf("    ;hw stack reset(restore esp)\r\n");

    printf("    mov esp, ebp\r\n");

#endif

    return currBytePtr;

}

unsigned char* noMake(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr) {

    if (!strcmp((*lastLexemInfoInTable)->lexemStr, T_NAME_0, MAX_LEXEM_SIZE)
        || !strcmp((*lastLexemInfoInTable)->lexemStr, T_DATA_0, MAX_LEXEM_SIZE)
        || !strcmp((*lastLexemInfoInTable)->lexemStr, T_BODY_0, MAX_LEXEM_SIZE)
        || !strcmp((*lastLexemInfoInTable)->lexemStr, T_DATA_TYPE_0, MAX_LEXEM_SIZE)
        || !strcmp((*lastLexemInfoInTable)->lexemStr, T_COMA_0, MAX_LEXEM_SIZE)
        || !strcmp((*lastLexemInfoInTable)->lexemStr, T_END_0, MAX_LEXEM_SIZE)
    ) {

        return ++ * lastLexemInfoInTable, currBytePtr;

    }

    return currBytePtr;

```

```

}

unsigned char* createPattern() {

    return NULL;

}

unsigned char* getObjectCodeBytePtr(unsigned char* baseBytePtr) {

    return baseBytePtr + baseOperationObjectOffset;

}

unsigned char* getImageCodeBytePtr(unsigned char* baseBytePtr) {

    return baseBytePtr + baseOperationOffset;

}

unsigned char* makeCode(struct LexemInfo** lastLexemInfoInTable/*TODO:...*/, unsigned char* currBytePtr, unsigned char generatorMode) { // TODO:...

    currBytePtr = makeTitle(lastLexemInfoInTable, currBytePtr);
    currBytePtr = makeDependenciesDeclaration(lastLexemInfoInTable, currBytePtr);

    currBytePtr = makeDataSection(lastLexemInfoInTable, currBytePtr);

    currBytePtr = makeBeginProgramCode(lastLexemInfoInTable, currBytePtr);
    lexemInfoTransformationTempStackSize = 0;
    currBytePtr = makeInitCode(lastLexemInfoInTable, currBytePtr);
    currBytePtr = initMake(lastLexemInfoInTable, currBytePtr);
    currBytePtr = makeSaveHWStack(lastLexemInfoInTable, currBytePtr);
    for (struct LexemInfo* lastLexemInfoInTable_; lastLexemInfoInTable_ = *lastLexemInfoInTable, (*lastLexemInfoInTable)->lexemStr[0] != '\0';
    {

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

        //

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

        //currBytePtr = makeForCycleCode(lastLexemInfoInTable, currBytePtr);
        //currBytePtr = makeToOrDowntoCycleCode(lastLexemInfoInTable, currBytePtr);
        //currBytePtr = makeDoCycleCode(lastLexemInfoInTable, currBytePtr);
        //currBytePtr = makeSemicolonAfterForCycleCode(lastLexemInfoInTable, currBytePtr);
        FOR_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
    }
}

```



```

//
WHILE_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
//

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

//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeValueCode(lastLexemInfoInTable, currBytePtr);
//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeIdentifierCode(lastLexemInfoInTable, currBytePtr);
OPERAND_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeNotCode(lastLexemInfoInTable, currBytePtr);
NOT_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
AND_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
OR_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

EQUAL_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
NOT_EQUAL_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
LESS_OR_EQUAL_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
GREATER_OR_EQUAL_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeAddCode(lastLexemInfoInTable, currBytePtr);
//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeSubCode(lastLexemInfoInTable, currBytePtr);
//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeMulCode(lastLexemInfoInTable, currBytePtr);
//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeDivCode(lastLexemInfoInTable, currBytePtr);
//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeModCode(lastLexemInfoInTable, currBytePtr);
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);

//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeRightToLeftBindCode(lastLexemInfoInTable, currBytePtr);
//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeLeftToRightBindCode(lastLexemInfoInTable, currBytePtr);
INPUT_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
OUTPUT_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makeGetCode(lastLexemInfoInTable, currBytePtr);
//if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr = makePutCode(lastLexemInfoInTable, currBytePtr);
RLBIND_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

```

```

        /** (1) Ignore phase*/if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr =
makeSemicolonAfterNonContextCode(lastLexemInfoInTable, currBytePtr);

        /** (2) Ignore phase*/if (lastLexemInfoInTable_ == *lastLexemInfoInTable) currBytePtr =
makeSemicolonIgnoreContextCode(lastLexemInfoInTable, currBytePtr);

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

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

        if (lastLexemInfoInTable_ == *lastLexemInfoInTable) {

            currBytePtr = noMake(lastLexemInfoInTable, currBytePtr);

        }

        if (lastLexemInfoInTable_ == *lastLexemInfoInTable) {

            printf("\r\nError in the code generator! \"%s\" - unexpected token!\r\n", (*lastLexemInfoInTable)->lexemStr);

            exit(0);

        }

    }

    currBytePtr = makeResetHWStack(lastLexemInfoInTable, currBytePtr);

    currBytePtr = makeEndProgramCode(lastLexemInfoInTable, currBytePtr);

    return currBytePtr;

}

//unsigned char outCode[GENERATED_TEXT_SIZE] = { '0' };

void viewCode(unsigned char* outCodePtr, unsigned long long int 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: ");

    unsigned long long int outCodePrintIndex = outCodePrintSize - 1;

    for (unsigned long long int index = 0; index <= outCodePrintIndex;) {

        printf("0x%02X ", outCodePtr[index]);

        if (!(++index % align)) {

            unsigned long long int indexMinus16 = index - align;

            do {

                //printf("0x%02X ", outCodePtr[index]);

                if (outCodePtr[indexMinus16] >= 32 && outCodePtr[indexMinus16] <= 126) {

                    printf("%c", outCodePtr[indexMinus16]);

                }

                else {

                    printf(" ");

                    //printf("%2c", 32);

                }

            }

```

```

        } while (++indexMinus16 % align);

        printf("\r\n;0x%08X: ", (unsigned int)index);

    }

}

Goto.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: goto_lable.cpp           *

*           (draft!) *

*****/

#include <string>

#include <map>

// #include <utility>

#include <stack>

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

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

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

    unsigned char multitokenSize, multitokenSize_ = detectMultiToken(*lastLexemInfoInTable + 1, MULTI_TOKEN_NULL_STATEMENT);

    multitokenSize = detectMultiToken(*lastLexemInfoInTable + multitokenSize_ + 1, MULTI_TOKEN_COLON);

    if (multitokenSize) {

        multitokenSize += multitokenSize_;

    }

    if ((*lastLexemInfoInTable)->tokenType != IDENTIFIER_LEXEME_TYPE){

        return currBytePtr;

    }

    if (multitokenSize++) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;ident \"\%s\"(as label) previous \"\%s\"\\r\\n", (*lastLexemInfoInTable)->lexemStr, tokenStruct[MULTI_TOKEN_COLON][0]);

#endif

        labelInfoTable[(lastLexemInfoInTable)->lexemStr].first = (unsigned long long int)currBytePtr;

```

```

        while(!labelInfoTable[(*lastLexemInfoInTable)->lexemStr].second.empty()){
            *((unsigned int*)labelInfoTable[(*lastLexemInfoInTable)->lexemStr].second.top() = (unsigned int)(currBytePtr - (unsigned
char*)labelInfoTable[(*lastLexemInfoInTable)->lexemStr].second.top() - 4);

            labelInfoTable[(*lastLexemInfoInTable)->lexemStr].second.pop();

        }

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf(" LABEL@%016lX:\r\n", (unsigned long long int)&labelInfoTable[(*lastLexemInfoInTable)->lexemStr].first);

#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    return currBytePtr;

}

unsigned char* makeGotoLabelCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_GOTO);
    if (multitokenSize++) {
        if ((*lastLexemInfoInTable + 1)->tokenType != IDENTIFIER_LEXEME_TYPE) {
            return currBytePtr;
        }
    }

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("\r\n");
    printf(" ;\"%s\" previous ident \"%s\"(as label)\r\n", tokenStruct[MULTI_TOKEN_GOTO][0], (*lastLexemInfoInTable)[1].lexemStr);

#endif

    const unsigned char code__jmp_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 };

    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jmp_offset, 5);

    if (labelInfoTable.find((*lastLexemInfoInTable)[1].lexemStr) == labelInfoTable.end()) {
        labelInfoTable[(*lastLexemInfoInTable)[1].lexemStr].first = ~0;
    }

    if (labelInfoTable[(*lastLexemInfoInTable)[1].lexemStr].first == ~0) {
        labelInfoTable[(*lastLexemInfoInTable)[1].lexemStr].second.push((unsigned long long int)(currBytePtr - 4));
    }
    else {
        *((unsigned int*)(currBytePtr - 4) = (unsigned int)((unsigned char*)labelInfoTable[(*lastLexemInfoInTable)[1].lexemStr].first -
currBytePtr);
    }
}

```

```

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("    jmp LABEL@%016llx\r\n", (unsigned long long int)&labelInfoTable[(*lastLexemInfoInTable)[1].lexemStr].first);

#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

Greater_or_equal.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *
*
*      file: greater_or_equal.cpp      *
*
*      (draft!) *
*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

unsigned char* makeIsGreaterOrEqualCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_GREATER_OR_EQUAL);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\\\"%s\\\"\\r\n", tokenStruct[MULTI_TOKEN_GREATER_OR_EQUAL][0]);

#endif

        const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

        const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };

        const unsigned char code__cmp_stackTopByECX_eax[] = { 0x39, 0x01 };

        const unsigned char code__setge_al[] = { 0x0F, 0x9D, 0xC0 };

        const unsigned char code__and_eax_1[] = { 0x83, 0xE0, 0x01 };

        const unsigned char code__mov_stackTopByECX_eax[] = { 0x89, 0x01 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_stackTopByECX_eax, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__setge_al, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_eax_1, 3);

```

```

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_stackTopByECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("  mov eax, dword ptr[ecx]\r\n");

    printf("  sub ecx, 4\r\n");

    printf("  cmp dword ptr[ecx], eax\r\n");

    printf("  setge al\r\n");

    printf("  and eax, 1\r\n");

    printf("  mov dword ptr[ecx], eax\r\n");

#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

If_then.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: if_then.cpp                       *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

#include "string.h"

unsigned char* makeIfCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_IF);

    if (!multitokenSize

        && tokenStruct[MULTI_TOKEN_IF][1][0] == '('

        && !strcmp((*lastLexemInfoInTable)->lexemStr, tokenStruct[MULTI_TOKEN_IF][0], MAX_LEXEM_SIZE)) {

        multitokenSize = 1;

    }

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("  ;\"%s\"\r\n", tokenStruct[MULTI_TOKEN_IF][0]);

#endif

    }

}

```

```

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = **lastLexemInfoInTable;

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;
    }

    return currBytePtr;
}

unsigned char* makeThenCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_THEN);
    if (!multitokenSize && tokenStruct[MULTI_TOKEN_IF][1][0] == '(') {
        multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_NULL_STATEMENT);
    }
    if (multitokenSize
        && lexemInfoTransformationTempStackSize
        && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_IF][0], MAX_LEXEM_SIZE)
    ) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("\r\n");
        printf("    ;after cond expresion (after \"%s\")\r\n", tokenStruct[MULTI_TOKEN_IF][0]);
#endif

        const unsigned char code__cmp_eax_0[] = { 0x83, 0xF8, 0x00 };
        const unsigned char code__jz_offset[] = { 0x0F, 0x84, 0x00, 0x00, 0x00, 0x00 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_eax_0, 3);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jz_offset, 6);

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = **lastLexemInfoInTable;
        strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI_TOKEN_THEN][0],
MAX_LEXEM_SIZE);

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue = (unsigned long long int)(currBytePtr - 4);

#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("    cmp eax, 0\r\n");
        printf("    jz LABEL@AFTER_THEN_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);
#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;
    }

    return currBytePtr;
}

```

```

}

unsigned char* makePostThenCode_(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    const unsigned char code__mov_eax_1[] = { 0xB8, 0x01, 0x00, 0x00, 0x00 };

    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_1, 5);

    *((unsigned int*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue = (unsigned int)(currBytePtr -
(unsigned char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue - 4);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("    mov eax, 1\r\n");

    printf("    LABEL@AFTER_THEN_%016llx:\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

#endif

    return currBytePtr;

}

unsigned char* makeSemicolonAfterThenCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) { // Or Ender!

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_SEMICOLON);

    if (multitokenSize

        &&

        lexemInfoTransformationTempStackSize >= 2

        && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_IF][0], MAX_LEXEM_SIZE)

        && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_THEN][0], MAX_LEXEM_SIZE)

    ) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\"%s\" (after \"then\"-part of %s-operator)\r\n", tokenStruct[MULTI_TOKEN_SEMICOLON][0],
tokenStruct[MULTI_TOKEN_IF][0]);

#endif

        currBytePtr = makePostThenCode_(lastLexemInfoInTable, currBytePtr, generatorMode);

        lexemInfoTransformationTempStackSize -= 2;

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    return currBytePtr;

}

Input.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

```



```

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: input.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_INPUT);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\"%s\\\"\\r\\n", tokenStruct[MULTI_TOKEN_INPUT][0]);

#endif

        const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

        const unsigned char code__mov_edx_address[] = { 0xBA, 0x00, 0x00, 0x00, 0x00 };

        const unsigned char code__add_edx_esi[] = { 0x03, 0xD6 };

        const unsigned char code__push_ecx[] = { 0x51 };

        //const unsigned char code__push_ebx[] = { 0x53 };

        const unsigned char code__push_esi[] = { 0x56 };

        const unsigned char code__push_edi[] = { 0x57 };

        const unsigned char code__call_edx[] = { 0xFF, 0xD2 };

        const unsigned char code__pop_edi[] = { 0x5F };

        const unsigned char code__pop_esi[] = { 0x5E };

        //const unsigned char code__pop_ebx[] = { 0x5B };

        const unsigned char code__pop_ecx[] = { 0x59 };

        const unsigned char code__mov_ebx_valueByAdrrssInECX[] = { 0x8B, 0x19 };

        const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };

        const unsigned char code__add_ebx_edi[] = { 0x33, 0xDF };

        const unsigned char code__mov_stackTopByEBX_eax[] = { 0x89, 0x03 };

        const unsigned char code__mov_ecx_edi[] = { 0x8B, 0xCF };

        const unsigned char code__add_ecx_512[] = { 0x81, 0xC1, 0x00, 0x02, 0x00, 0x00 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_edx_address, 5);

        *(unsigned int*)&(currBytePtr[-4]) = (unsigned int)getProcOffset;

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_edx_esi, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_ecx, 1);

        //currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_ebx, 1);

```

```

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_esi, 1);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_edi, 1);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__call_edx, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__pop_edi, 1);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__pop_esi, 1);
//currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__pop_ebx, 1);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__pop_ecx, 1);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_ebx_valueByAddressInECX, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_ebx_edi, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_stackTopByEBX_eax, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_ecx_edi, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_ecx_512, 6);

```

```

#ifdef DEBUG_MODE_BY_ASSEMBLY

```

```

    printf("  mov eax, dword ptr[ecx]\r\n");
    printf("  mov edx, 0x08Xh\r\n", (unsigned int)getProcOffset);
    printf("  add edx, esi\r\n");
    printf("  push ecx\r\n");
    printf("  ;push ebx\r\n");
    printf("  push esi\r\n");
    printf("  push edi\r\n");
    printf("  call edx\r\n");
    printf("  pop edi\r\n");
    printf("  pop esi\r\n");
    printf("  ;pop ebx\r\n");
    printf("  pop ecx\r\n");
    printf("  mov ebx, dword ptr[ecx]\r\n");
    printf("  sub ecx, 4\r\n");
    printf("  add ebx, edi\r\n");
    printf("  mov dword ptr [ebx], eax\r\n");
    printf("  mov ecx, edi ; reset second stack\r\n");
    printf("  add ecx, 512 ; reset second stack\r\n");

```

```

#endif

```

```

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;
}

```

```

return currBytePtr;

```

```

}

```

```

Less_or_equal

```

```

#define _CRT_SECURE_NO_WARNINGS

```

```

/*****

```

```

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: less_or_equal.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

unsigned char* makeIsLessOrEqualCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_LESS_OR_EQUAL);
    if (multitokenSize) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("\r\n");
        printf("    ;\"%s\\r\\n\", tokenStruct[MULTI_TOKEN_LESS_OR_EQUAL][0]);
#endif

        const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };
        const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };
        const unsigned char code__cmp_stackTopByECX_eax[] = { 0x39, 0x01 };
        const unsigned char code__setle_al[] = { 0x0F, 0x9E, 0xC0 };
        const unsigned char code__and_eax_1[] = { 0x83, 0xE0, 0x01 };
        const unsigned char code__mov_stackTopByECX_eax[] = { 0x89, 0x01 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_stackTopByECX_eax, 2);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__setle_al, 3);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_eax_1, 3);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_stackTopByECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("    mov eax, dword ptr[ecx]\\r\\n");
        printf("    sub ecx, 4\\r\\n");
        printf("    cmp dword ptr[ecx], eax\\r\\n");
        printf("    setle al\\r\\n");
        printf("    and eax, 1\\r\\n");
        printf("    mov dword ptr[ecx], eax\\r\\n");
#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;
    }
}

```

```

        return currBytePtr;

    }

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: lexica.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/config.h"

#include "../src/include/lexica/lexica.h"


#include "stdio.h"

#include "stdlib.h"

#include "string.h"


#include <fstream>

#include <iostream>

//include <algorithm>

#include <iterator>

#include <regex>


//struct LexemInfo {

//           char lexemStr[MAX_LEXEM_SIZE];

//           unsigned int lexemId;

//           unsigned int tokenType;

//           unsigned int ifvalue;

//           unsigned int row;

//           unsigned int col;

//           // TODO: ...

//};


#define MAX_ACCESSORY_STACK_SIZE_123 128

char tempStrFor_123[MAX_TEXT_SIZE/*?TODO:... MAX_ACCESSORY_STACK_SIZE_123 * 64*/] = {"0"};

unsigned long long int tempStrForCurrIndex = 0;


struct LexemInfo lexemesInfoTable[MAX_WORD_COUNT];// = { { "", 0, 0, 0 } };

struct LexemInfo* lastLexemInfoInTable = lexemesInfoTable; // first for begin


char identifierIdsTable[MAX_WORD_COUNT][MAX_LEXEM_SIZE] = { "" };

```

```

LexemInfo::LexemInfo() {
    lexemStr[0] = '\0';
    lexemId = 0;
    tokenType = 0;
    ifvalue = 0;
    row = ~0;
    col = ~0;
}

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) {
    strncpy(this->lexemStr, lexemStr, MAX_LEXEM_SIZE);
    this->lexemId = lexemId;
    this->tokenType = tokenType;
    this->ifvalue = ifvalue;
    this->row = row;
    this->col = col;
}

LexemInfo::LexemInfo(const NonContainedLexemInfo& nonContainedLexemInfo){
    strncpy(lexemStr, nonContainedLexemInfo.lexemStr, MAX_LEXEM_SIZE);
    lexemId = nonContainedLexemInfo.lexemId;
    tokenType = nonContainedLexemInfo.tokenType;
    ifvalue = nonContainedLexemInfo.ifvalue;
    row = nonContainedLexemInfo.row;
    col = nonContainedLexemInfo.col;
}

NonContainedLexemInfo::NonContainedLexemInfo() {
    (lexemStr = tempStrFor_123 + tempStrForCurrIndex)[0] = '\0';
    tempStrForCurrIndex += 32;// MAX_LEXEM_SIZE;
    lexemId = 0;
    tokenType = 0;
    ifvalue = 0;
    row = ~0;
    col = ~0;
}

NonContainedLexemInfo::NonContainedLexemInfo(const LexemInfo& lexemInfo) {
    //strncpy(lexemStr, lexemInfo.lexemStr, MAX_LEXEM_SIZE); //
    lexemStr = (char*)lexemInfo.lexemStr;
    lexemId = lexemInfo.lexemId;
    tokenType = lexemInfo.tokenType;
    ifvalue = lexemInfo.ifvalue;
    row = lexemInfo.row;
    col = lexemInfo.col;
}

```

```
}
```

```
void printLexemes(struct LexemInfo* lexemInfoTable, char printBadLexeme) {

    if (printBadLexeme) {
        printf("Bad lexeme:\r\n");
    }
    else {
        printf("Lexemes table:\r\n");
    }
    printf("-----\r\n");
    //printf("index\t\tlexeme\t\tid\t\ttype\t\tifvalue\t\trow\t\tcol\r\n");
    printf("index      lexeme      id      type      ifvalue row      col\r\n");
    printf("-----\r\n");

    for (unsigned long long int index = 0; (!index || !printBadLexeme) && lexemInfoTable[index].lexemStr[0] != '\0'; ++index) {
        printf("%5llu%17s%12llu%10llu%11llu%4lld%8lld\r\n", index, lexemInfoTable[index].lexemStr, lexemInfoTable[index].lexemId,
lexemInfoTable[index].tokenType, lexemInfoTable[index].ifvalue, lexemInfoTable[index].row, lexemInfoTable[index].col);
    }
    printf("-----\r\n\r\n");

    return;
}
```

```
void printLexemesToFile(struct LexemInfo* lexemInfoTable, char printBadLexeme, const char* filename) {

    FILE* file = fopen(filename, "wb");

    if (!file) {
        perror("Failed to open file");
        return;
    }

    if (printBadLexeme) {
        fprintf(file, "Bad lexeme:\r\n");
    }
    else {
        fprintf(file, "Lexemes table:\r\n");
    }
    fprintf(file, "-----\r\n");
    //fprintf(file, "index\t\tlexeme\t\tid\t\ttype\t\tifvalue\t\trow\t\tcol\r\n");
    fprintf(file, "index      lexeme      id      type      ifvalue row      col\r\n");
    fprintf(file, "-----\r\n");

    for (unsigned long long int index = 0; (!index || !printBadLexeme) && lexemInfoTable[index].lexemStr[0] != '\0'; ++index) {
        fprintf(file, "%5llu%17s%12llu%10llu%11llu%4lld%8lld\r\n",
            index,
```

```

        lexemInfoTable[index].lexemStr,
        lexemInfoTable[index].lexemId,
        lexemInfoTable[index].tokenType,
        lexemInfoTable[index].ifvalue,
        lexemInfoTable[index].row,
        lexemInfoTable[index].col);
    }
    fprintf(file, "-----\r\n\r\n");

    fclose(file);
}

// get identifier id
unsigned int getIdentifierId(char(*identifierIdsTable)[MAX_LEXEM_SIZE], char* str) {
    unsigned int index = 0;
    for (; identifierIdsTable[index][0] != '\0'; ++index) {
        if (!strcmp(identifierIdsTable[index], str, MAX_LEXEM_SIZE)) {
            return index;
        }
    }
    strcpy(identifierIdsTable[index], str, MAX_LEXEM_SIZE);
    identifierIdsTable[index + 1][0] = '\0'; // not necessarily for zero-init identifierIdsTable
    return index;
}

// try to get identifier
unsigned int tryToGetIdentifier(struct LexemInfo* lexemInfoInTable, char(*identifierIdsTable)[MAX_LEXEM_SIZE]) {
    char identifiers_re[] = IDENTIFIERS_RE;
    //char identifiers_re[] = "[A-Z][A-Z][A-Z][A-Z][A-Z][A-Z]";

    if (std::regex_match(std::string(lexemInfoInTable->lexemStr), std::regex(identifiers_re))) {
        lexemInfoInTable->lexemId = getIdentifierId(identifierIdsTable, lexemInfoInTable->lexemStr);
        lexemInfoInTable->tokenType = IDENTIFIER_LEXEME_TYPE;
        return SUCCESS_STATE;
    }

    return ~SUCCESS_STATE;
}

// try to get value
unsigned int tryToGetUnsignedValue(struct LexemInfo* lexemInfoInTable) {
    char unsignedvalues_re[] = UNSIGNEDVALUES_RE;
    //char unsignedvalues_re[] = "0[1-9][0-9]*";

```

```

    if (std::regex_match(std::string(lexemInfoInTable->lexemStr), std::regex(unsignedvalues_re))) {
        lexemInfoInTable->ifvalue = atoi(lastLexemInfoInTable->lexemStr);
        lexemInfoInTable->lexemId = MAX_VARIABLES_COUNT + MAX_KEYWORD_COUNT;
        lexemInfoInTable->tokenType = VALUE_LEXEME_TYPE;
        return SUCCESS_STATE;
    }

    return ~SUCCESS_STATE;
}

int commentRemover(char* text, const char* openStrSpc, const char* closeStrSpc) {
    bool eofAlternativeCloseStrSpcType = false;
    bool explicitCloseStrSpc = true;
    if (!strcmp(closeStrSpc, "\n")) {
        eofAlternativeCloseStrSpcType = true;
        explicitCloseStrSpc = false;
    }

    unsigned int commentSpace = 0;

    unsigned int textLength = strlen(text); // strlen(text, MAX_TEXT_SIZE);
    unsigned int openStrSpcLength = strlen(openStrSpc); // strlen(openStrSpc, MAX_TEXT_SIZE);
    unsigned int closeStrSpcLength = strlen(closeStrSpc); // strlen(closeStrSpc, MAX_TEXT_SIZE);
    if (!closeStrSpcLength) {
        return -1; // no set closeStrSpc
    }
    unsigned char oneLevelComment = 0;
    if (!strcmp(openStrSpc, closeStrSpc, MAX_LEXEM_SIZE)) {
        oneLevelComment = 1;
    }

    for (unsigned int index = 0; index < textLength; ++index) {
        if (!strcmp(text + index, closeStrSpc, closeStrSpcLength) && (explicitCloseStrSpc || commentSpace)) {
            if (commentSpace == 1 && explicitCloseStrSpc) {
                for (unsigned int index2 = 0; index2 < closeStrSpcLength; ++index2) {
                    text[index + index2] = ' ';
                }
            }
            else if (commentSpace == 1 && !explicitCloseStrSpc) {
                index += closeStrSpcLength - 1;
            }
            oneLevelComment ? commentSpace = !commentSpace : commentSpace = 0;
        }
    }
}

```



```

    }

    else if (!strncmp(text + index, openStrSpc, openStrSpcLength)) {
        oneLevelComment ? commentSpace = !commentSpace : commentSpace = 1;
    }

    if (commentSpace && text[index] != ' ' && text[index] != '\t' && text[index] != '\r' && text[index] != '\n') {
        text[index] = ' ';
    }

}

if (commentSpace && !eofAlternativeCloseStrSpcType) {
    return -1;
}

return 0;
}

void prepareKeyWordIdGetter(char* keywords_, char* keywords_re) {
    if (keywords_ == NULL || keywords_re == NULL) {
        return;
    }

    for (char* keywords_re_ = keywords_re, *keywords__ = keywords_; (*keywords_re_ != '\0') ? 1 : (*keywords__ = '\0', 0); (*keywords_re_ != '\\' ||
(keywords_re_[1] != '+' && keywords_re_[1] != '*' && keywords_re_[1] != '|')) ? *keywords__++ = *keywords_re_ : 0, ++keywords_re_);
}

unsigned int getKeyWordId(char* keywords_, char* lexemStr, unsigned int baseId) {
    if (keywords_ == NULL || lexemStr == NULL) {
        return ~0;
    }

    char* lexemInKeywords_ = keywords_;
    size_t lexemStrLen = strlen(lexemStr);
    if (!lexemStrLen) {
        return ~0;
    }

    for (; lexemInKeywords_ = strstr(lexemInKeywords_, lexemStr), lexemInKeywords_ != NULL && lexemInKeywords_[lexemStrLen] != '|' &&
lexemInKeywords_[lexemStrLen] != '\0'; ++lexemInKeywords_);

    return lexemInKeywords_ - keywords_ + baseId;
}

```

```

// try to get KeyWord

char tryToGetKeyWord(struct LexemInfo* lexemInfoInTable) {

    char keywords_re[] = KEYWORDS_RE;

    //char keywords_re[] = ";<<>>\\|+|-
    \\*|,|=|!=|:|\\(|\\)|NAME|DATA|BODY|END|EXIT|CONTINUE|GET|PUT|IF|ELSE|FOR|TO|DOWNT|DO|WHILE|REPEAT|UNTIL|GOTO|DIV|MOD|<|=|>|=|NOT|
    AND|OR|INTEGER16";

    //char keywords_re[] = ";<<\\|+\\|+|-
    \\*\\*|,|=|\\(|\\)|!=|:|name|data|body|end|get|put|for|to|downto|do|while|continue|exit|repeat|until|if|goto|div|mod|le|ge|not|and|or|long|int";

    char keywords_[sizeof(keywords_re)] = { '0' };

    prepareKeyWordIdGetter(keywords_, keywords_re);

    if (std::regex_match(std::string(lexemInfoInTable->lexemStr), std::regex(keywords_re))) {

        lexemInfoInTable->lexemId = getKeyWordId(keywords_, lexemInfoInTable->lexemStr, MAX_VARIABLES_COUNT);

        lexemInfoInTable->tokenType = KEYWORD_LEXEME_TYPE;

        return SUCCESS_STATE;

    }

    return ~SUCCESS_STATE;

}

void setPositions(const char* text, struct LexemInfo* lexemInfoTable) {

    unsigned long long int line_number = 1;

    const char* pos = text, * line_start = text;

    if (lexemInfoTable) while (*pos != '\0' && lexemInfoTable->lexemStr[0] != '\0') {

        const char* line_end = strchr(pos, '\n');

        if (!line_end) {

            line_end = text + strlen(text);

        }

        char line_[4096], * line = line_; ///! TODO: ...

        strncpy(line, pos, line_end - pos);

        line[line_end - pos] = '\0';

        for (char* found_pos; lexemInfoTable->lexemStr[0] != '\0' && (found_pos = strstr(line, lexemInfoTable->lexemStr)); line +=
        strlen(lexemInfoTable->lexemStr), ++lexemInfoTable) {

            lexemInfoTable->row = line_number;

            lexemInfoTable->col = found_pos - line_ + 1ull;

        }

        line_number++;

        pos = line_end;

        if (*pos == '\n') {

            pos++;

        }

    }

```

```

    }

}

struct LexemInfo lexicalAnalyze(struct LexemInfo* lexemInfoInPtr, char(*identifierIdsTable)[MAX_LEXEM_SIZE]) {
    struct LexemInfo ifBadLexemeInfo; // = { 0 };

    if (tryToGetKeyWord(lexemInfoInPtr) == SUCCESS_STATE);
    else if (tryToGetIdentifier(lexemInfoInPtr, identifierIdsTable) == SUCCESS_STATE);
    else if (tryToGetUnsignedValue(lexemInfoInPtr) == SUCCESS_STATE);
    else {
        ifBadLexemeInfo.tokenType = UNEXPEXTED_LEXEME_TYPE;
    }

    return ifBadLexemeInfo;
}

struct LexemInfo tokenize(char* text, struct LexemInfo** lastLexemInfoInTable, char(*identifierIdsTable)[MAX_LEXEM_SIZE], struct
LexemInfo(*lexicalAnalyzeFunctionPtr)(struct LexemInfo*, char(*)[MAX_LEXEM_SIZE])) {
    char tokens_re[] = TOKENS_RE;

    //char tokens_re[] = "<|<<|>>|\\+|-|\\*|,|=|!|=:|\\(|\\)|<|=|[_0-9A-Za-z]+|[\\^ \\t\\r\\f\\v\\n]";
    //char tokens_re[] = "<|<<|>>|\\+|-|\\*|,|=|!|=:|\\(|\\)|<|=|[_0-9A-Za-z]+|[\\^ \\t\\r\\f\\v\\n]";

    std::regex tokens_re_(tokens_re);

    struct LexemInfo ifBadLexemeInfo; // = { 0 };

    std::string stringText(text);

    for (std::sregex_token_iterator end, tokenIterator(stringText.begin(), stringText.end(), tokens_re_); tokenIterator != end; ++tokenIterator, ++ *
lastLexemInfoInTable) {
        std::string str = *tokenIterator;

        strncpy((*lastLexemInfoInTable)->lexemStr, str.c_str(), MAX_LEXEM_SIZE);

        if ((ifBadLexemeInfo = (*lexicalAnalyzeFunctionPtr)(*lastLexemInfoInTable, identifierIdsTable)).tokenType ==
UNEXPEXTED_LEXEME_TYPE) {
            break;
        }
    }

    setPositions(text, lexemesInfoTable);

    if (ifBadLexemeInfo.tokenType == UNEXPEXTED_LEXEME_TYPE) {
        strncpy(ifBadLexemeInfo.lexemStr, (*lastLexemInfoInTable)->lexemStr, MAX_LEXEM_SIZE);

        ifBadLexemeInfo.row = (*lastLexemInfoInTable)->row;

        ifBadLexemeInfo.col = (*lastLexemInfoInTable)->col;
    }

    return ifBadLexemeInfo;
}

```

```

}

Machinecodegen

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: machinecodegen_addon.cpp      *

*           (draft!) *

*****/

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// #define DEBUG_RECONSTRUCT

unsigned char buffer[256 * 1024];

extern unsigned long long int a123_array_part_count;
extern unsigned char* a123[39];
extern unsigned long long int a123_array_part_size[39];
extern unsigned long long int a123_zero_part_count;
extern unsigned long long int a123_zeros[39];

extern unsigned long long int o123_array_part_count;
extern unsigned char* o123[5];
extern unsigned long long int o123_array_part_size[5];
extern unsigned long long int o123_zero_part_count;
extern unsigned long long int o123_zeros[4];

unsigned long long int buildTemplateForCodeObject(unsigned char* byteImage) {
    if (!byteImage) {
        exit(EXIT_FAILURE);
    }
    unsigned long long int byteIndex = 0;

    unsigned long long int totalByteCount = 0;

    unsigned long long int array_index = 0;
    unsigned long long int zero_index = 0;

    for (unsigned long long int i = 0; i < o123_array_part_count + o123_zero_part_count; ++i) {
        if (i % 2 == 0) {
            unsigned char* current_array = NULL;

```

```

current_array = o123[array_index++];

if (current_array) {
    //fwrite(current_array, 1, o123_array_part_size[array_index - 1]/*sizeof(current_array)*/, outfile);

    for (unsigned long long int localByteIndex = 0; localByteIndex < o123_array_part_size[array_index - 1]; ++localByteIndex) {
        byteImage[byteIndex++] = current_array[localByteIndex];
    }
}

#ifdef DEBUG_RECONSTRUCT
    printf(": sizeof(current_array) = %llu\r\n", o123_array_part_size[array_index - 1]/*sizeof(current_array)*/);
#endif

totalByteCount += o123_array_part_size[array_index - 1];
}
}

else {
    unsigned long long int zero_count = 0;
    zero_count = o123_zeros[zero_index++];
    if (zero_count > 0) {
        unsigned char* zeros = (unsigned char*)calloc(zero_count, sizeof(unsigned char));

        if (!zeros) {
            perror("Memory allocation failed for zero block");
            //fclose(outfile);
            exit(EXIT_FAILURE);
        }

        //fwrite(zeros, 1, zero_count, outfile);

        for (unsigned long long int localByteIndex = 0; localByteIndex < zero_count; ++localByteIndex) {
            byteImage[byteIndex++] = 0;
        }
    }

#ifdef DEBUG_RECONSTRUCT
        printf("zero_count = %llu\r\n", zero_count);
#endif

totalByteCount += zero_count;
free(zeros);
}
}

}

//fclose(outfile);

#ifdef DEBUG_RECONSTRUCT
    printf("Image reconstructed.\n"/*, output_file*/);
#endif

return byteIndex;
}

```

```

unsigned long long int buildTemplateForCodeImage(unsigned char* byteImage) {
    if (!byteImage) {
        exit(EXIT_FAILURE);
    }

    unsigned long long int byteIndex = 0;

    unsigned long long int totalByteCount = 0;

    unsigned long long int array_index = 0;
    unsigned long long int zero_index = 0;

    for (unsigned long long int i = 0; i < a123_array_part_count + a123_zero_part_count; ++i) {
        if (i % 2 == 0) {
            unsigned char* current_array = NULL;
            current_array = a123[array_index++];
            if (current_array) {
                //fwrite(current_array, 1, a123_array_part_size[array_index - 1]/sizeof(current_array)*/, outfile);

                for (unsigned long long int localByteIndex = 0; localByteIndex < a123_array_part_size[array_index - 1]; ++localByteIndex) {
                    byteImage[byteIndex++] = current_array[localByteIndex];
                }
            }
#ifdef DEBUG_RECONSTRUCT
            printf(": sizeof(current_array) = %llu\r\n", a123_array_part_size[array_index - 1]/sizeof(current_array)*/);
#endif
            totalByteCount += a123_array_part_size[array_index - 1];
        }
        else {
            unsigned long long int zero_count = 0;
            zero_count = a123_zeros[zero_index++];
            if (zero_count > 0) {
                unsigned char* zeros = (unsigned char*)calloc(zero_count, sizeof(unsigned char));
                if (!zeros) {
                    perror("Memory allocation failed for zero block");
                    //fclose(outfile);
                    exit(EXIT_FAILURE);
                }
                //fwrite(zeros, 1, zero_count, outfile);

                for (unsigned long long int localByteIndex = 0; localByteIndex < zero_count; ++localByteIndex) {
                    byteImage[byteIndex++] = 0;
                }
            }
#ifdef DEBUG_RECONSTRUCT
            printf("zero_count = %llu\r\n", zero_count);
#endif
        }
    }
}

```

```

        totalByteCount += zero_count;

        free(zeros);

    }

}

}

//fclose(outfile);

#ifdef DEBUG_RECONSTRUCT

    printf("Image reconstructed.\n"/*, output_file*/);

#endif

return byteIndex;

}

void writeBytesToFile(const char* output_file, unsigned char* byteImage, unsigned long long int imageSize) {

    if (!output_file || !byteImage) {

        perror("Error in write image to file");

        exit(EXIT_FAILURE);

    }

    FILE* outfile = fopen(output_file, "wb");

    if (!outfile) {

        perror("Error opening output file");

        exit(EXIT_FAILURE);

    }

    if (imageSize) {

        fwrite(byteImage, 1, imageSize, outfile);

    }

    fclose(outfile);

    printf("File \"%s\" saved.\n", output_file);

}

Machinecodegen_pattern

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: machinecodegen_pattern.cpp          *

*           (draft!)                                   *

*****/

unsigned char a123_0[] = {

    0x4D,0x5A,0x90,0x00,0x03,0x00,0x00,0x00,0x04,0x00,0x00,0x00,0xFF,0xFF,0x00,0x00,

```

```

    0xB8,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,

};

#define a123_0_SIZE 25

#define a123_ZEROS_0 35

unsigned char a123_1[] = {
    0xD0,0x00,0x00,0x00,0x0E,0x1F,0xBA,0x0E,0x00,0xB4,0x09,0xCD,0x21,0xB8,0x01,0x4C,
    0xCD,0x21,0x54,0x68,0x69,0x73,0x20,0x70,0x72,0x6F,0x67,0x72,0x61,0x6D,0x20,0x63,
    0x61,0x6E,0x6E,0x6F,0x74,0x20,0x62,0x65,0x20,0x72,0x75,0x6E,0x20,0x69,0x6E,0x20,
    0x44,0x4F,0x53,0x20,0x6D,0x6F,0x64,0x65,0x2E,0x0D,0x0A,0x24,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0xBD,0x32,0xF9,0xFD,0xF9,0x53,0x97,0xAE,0xF9,0x53,0x97,0xAE,
    0xF9,0x53,0x97,0xAE,0xED,0x38,0x96,0xAF,0xFC,0x53,0x97,0xAE,0xF9,0x53,0x96,0xAE,
    0xFD,0x53,0x97,0xAE,0x1D,0x27,0x94,0xAF,0xF8,0x53,0x97,0xAE,0x1D,0x27,0x68,0xAE,
    0xF8,0x53,0x97,0xAE,0x1D,0x27,0x95,0xAF,0xF8,0x53,0x97,0xAE,0x52,0x69,0x63,0x68,
    0xF9,0x53,0x97,0xAE,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x50,0x45,0x00,0x00,0x4C,0x01,0x05,0x00,0xCA,0x22,0x65,0x67,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x01,0x0B,0x01,0x0E,0x1C,
    0x00,0x22,0x00,0x00,0x00,0x2A,0x00,0x00,0x00,0x00,0x00,0x00,0x10,0x00,0x00,
    0x00,0x10,0x00,0x00,0x00,0x40,0x00,0x00,0x00,0x40,0x00,0x00,0x10,0x00,0x00,
    0x00,0x02,0x00,0x00,0x06,0x00,0x00,0x00,0x00,0x00,0x00,0x06,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0xA0,0x00,0x00,0x00,0x04,0x00,0x00,0x00,0x00,0x00,
    0x03,0x00,0x40,0x81,0x00,0x00,0x10,0x00,0x00,0x10,0x00,0x00,0x00,0x10,0x00,
    0x00,0x10,0x00,0x00,0x00,0x00,0x00,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0xCC,0x41,0x00,0x00,0x3C,0x00,0x00,0x00,0x00,0x80,0x00,0x00,
    0xE0,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x90,0x00,0x00,0x30,0x00,0x00,0x18,0x40,0x00,0x00,
    0x70,
};

#define a123_1_SIZE 321

#define a123_ZEROS_1 44

unsigned char a123_2[] = {
    0x40,0x00,0x00,0x18,
};

#define a123_2_SIZE 4

#define a123_ZEROS_2 27

unsigned char a123_3[] = {
    0x2E,0x74,0x65,0x78,0x74,0x00,0x00,0x00,0xCA,0x20,0x00,0x00,0x10,0x00,0x00,
    0x00,0x22,0x00,0x00,0x00,0x04,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

```



```

0x00,0x00,0x00,0x00,0x20,0x00,0x00,0x60,0x2E,0x72,0x64,0x61,0x74,0x61,0x00,0x00,
0x76,0x02,0x00,0x00,0x00,0x40,0x00,0x00,0x04,0x00,0x00,0x26,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x40,
0x2E,0x64,0x61,0x74,0x61,0x00,0x00,0x00,0x91,0x21,0x00,0x00,0x50,0x00,0x00,
0x00,0x22,0x00,0x00,0x00,0x2A,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x40,0x00,0x00,0xC0,0x2E,0x72,0x73,0x72,0x63,0x00,0x00,0x00,
0xE0,0x01,0x00,0x00,0x00,0x80,0x00,0x00,0x02,0x00,0x00,0x00,0x4C,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x40,
0x2E,0x72,0x65,0x6C,0x6F,0x63,0x00,0x00,0x30,0x00,0x00,0x00,0x90,0x00,0x00,
0x00,0x02,0x00,0x00,0x00,0x4E,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x42,
};

#define a123_3_SIZE 200

#define a123_ZEROS_3 368

unsigned char a123_4[] = {
    0xE8,0x00,0x00,0x00,0x00,0x5E,0x83,0xEE,0x05,0x8B,0xFE,0x81,0xC7,0x00,0x40,0x00,
    0x00,0x8B,0xCF,0x81,0xC1,0x00,0x02,0x00,0x00,0xEB,0x7C,0x50,0x68,0x00,0x71,0x40,
    0x00,0x68,0x00,0x70,0x40,0x00,0xE8,0x99,0x20,0x00,0x00,0x83,0xC4,0x0C,0x6A,0x00,
    0x6A,0x00,0x50,0x68,0x00,0x70,0x40,0x00,0xFF,0x35,0x09,0x71,0x40,0x00,0xE8,0x7B,
    0x20,0x00,0x00,0xC3,0x55,0x8B,0xEC,0x6A,0x00,0x68,0x8D,0x71,0x40,0x00,0x6A,0x0F,
    0x68,0x0E,0x71,0x40,0x00,0xFF,0x35,0x05,0x71,0x40,0x00,0xE8,0x58,0x20,0x00,0x00,
    0x8D,0x35,0x0D,0x71,0x40,0x00,0x03,0x35,0x8D,0x71,0x40,0x00,0x83,0xEE,0x02,0xE8,
    0x04,0x00,0x00,0x00,0x8B,0xE5,0x5D,0xC3,0x33,0xC0,0xBB,0x01,0x00,0x00,0x00,0x33,
    0xC9,0x0F,0xB6,0x0E,0x85,0xC9,0x74,0x0E,0x83,0xE9,0x30,0x0F,0xAF,0xCB,0x03,0xC1,
    0x6B,0xDB,0x0A,0x4E,0xEB,0xEB,0xC3,0x6A,0xF6,0xE8,0x14,0x20,0x00,0x00,0xA3,0x05,
    0x71,0x40,0x00,0x6A,0xF5,0xE8,0x08,0x20,0x00,0x00,0xA3,0x09,0x71,0x40,0x00,0x33,
    0xC0,0xC3,
};

#define a123_4_SIZE 178

#define a123_ZEROS_4 8192

unsigned char a123_5[] = {
    0xFF,0x25,0x08,0x40,0x40,0x00,0xFF,0x25,0x00,0x40,0x40,0x00,0xFF,0x25,0x04,0x40,
    0x40,0x00,0xFF,0x25,0x10,0x40,0x40,
};

#define a123_5_SIZE 23

#define a123_ZEROS_5 311

unsigned char a123_6[] = {

```

```

0x30,0x42,0x00,0x00,0x40,0x42,0x00,0x00,0x20,0x42,0x00,0x00,0x00,0x00,0x00,
0x5E,0x42,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xCA,0x22,0x65,0x67,
0x00,0x00,0x00,0x00,0x02,0x00,0x00,0x00,0x54,0x00,0x00,0x00,0x88,0x40,0x00,0x00,
0x88,0x26,0x00,0x00,0x00,0x00,0x00,0x00,0xCA,0x22,0x65,0x67,0x00,0x00,0x00,0x00,
0x0C,0x00,0x00,0x00,0x14,0x00,0x00,0x00,0xDC,0x40,0x00,0x00,0xDC,0x26,0x00,0x00,
0x00,0x00,0x00,0x00,0xCA,0x22,0x65,0x67,0x00,0x00,0x00,0x00,0x0D,0x00,0x00,0x00,
0xDC,0x00,0x00,0x00,0xF0,0x40,0x00,0x00,0xF0,0x26,0x00,0x00,0x00,0x00,0x00,0x00,
0xCA,0x22,0x65,0x67,0x00,0x00,0x00,0x00,0x0E,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x52,0x53,0x44,0x53,0x05,0x39,0xA1,0x32,
0x25,0x9B,0x91,0x44,0x8E,0x6F,0x3B,0x20,0xBC,0x79,0x50,0x25,0x22,0x00,0x00,0x00,
0x43,0x3A,0x5C,0x55,0x73,0x65,0x72,0x73,0x5C,0x4E,0x61,0x7A,0x61,0x72,0x5C,0x73,
0x6F,0x75,0x72,0x63,0x65,0x5C,0x72,0x65,0x70,0x6F,0x73,0x5C,0x50,0x72,0x6F,0x6A,
0x65,0x63,0x74,0x36,0x30,0x5C,0x52,0x65,0x6C,0x65,0x61,0x73,0x65,0x5C,0x50,0x72,
0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x2E,0x70,0x64,0x62,

```

```
};
```

```
#define a123_6_SIZE 219
```

```
#define a123_ZEROS_6 21
```

```
unsigned char a123_7[] = {
```

```

0x47,0x43,0x54,0x4C,0x00,0x10,0x00,0x00,0xCA,0x20,0x00,0x00,0x2E,0x74,0x65,0x78,
0x74,0x24,0x6D,0x6E,0x00,0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x18,0x00,0x00,0x00,
0x2E,0x69,0x64,0x61,0x74,0x61,0x24,0x35,0x00,0x00,0x00,0x00,0x18,0x40,0x00,0x00,
0x70,0x00,0x00,0x00,0x2E,0x72,0x64,0x61,0x74,0x61,0x00,0x00,0x88,0x40,0x00,0x00,
0x44,0x01,0x00,0x00,0x2E,0x72,0x64,0x61,0x74,0x61,0x24,0x7A,0x7A,0x7A,0x64,0x62,
0x67,0x00,0x00,0x00,0xCC,0x41,0x00,0x00,0x28,0x00,0x00,0x00,0x2E,0x69,0x64,0x61,
0x74,0x61,0x24,0x32,0x00,0x00,0x00,0x00,0xF4,0x41,0x00,0x00,0x14,0x00,0x00,0x00,
0x2E,0x69,0x64,0x61,0x74,0x61,0x24,0x33,0x00,0x00,0x00,0x00,0x08,0x42,0x00,0x00,
0x18,0x00,0x00,0x00,0x2E,0x69,0x64,0x61,0x74,0x61,0x24,0x34,0x00,0x00,0x00,0x00,
0x20,0x42,0x00,0x00,0x56,0x00,0x00,0x00,0x2E,0x69,0x64,0x61,0x74,0x61,0x24,0x36,
0x00,0x00,0x00,0x00,0x50,0x00,0x00,0x91,0x21,0x00,0x00,0x2E,0x64,0x61,0x74,
0x61,0x00,0x00,0x00,0x00,0x80,0x00,0x00,0x60,0x00,0x00,0x00,0x2E,0x72,0x73,0x72,
0x63,0x24,0x30,0x31,0x00,0x00,0x00,0x00,0x60,0x80,0x00,0x00,0x80,0x01,0x00,0x00,
0x2E,0x72,0x73,0x72,0x63,0x24,0x30,0x32,0x00,0x00,0x00,0x00,0x08,0x42,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x50,0x42,0x00,0x00,0x40,0x00,0x00,
0x18,0x42,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x6A,0x42,0x00,0x00,
0x10,0x40,

```

```
};
```

```
#define a123_7_SIZE 258
```

```
#define a123_ZEROS_7 22
```

```
unsigned char a123_8[] = {
```

```

0x30,0x42,0x00,0x00,0x40,0x42,0x00,0x00,0x20,0x42,0x00,0x00,0x00,0x00,0x00,0x00,
0x5E,0x42,0x00,0x00,0x00,0x00,0x00,0x00,0xD5,0x02,0x47,0x65,0x74,0x53,0x74,0x64,
0x48,0x61,0x6E,0x64,0x6C,0x65,0x00,0x00,0x68,0x04,0x52,0x65,0x61,0x64,0x43,0x6F,
0x6E,0x73,0x6F,0x6C,0x65,0x41,0x00,0x00,0x0B,0x06,0x57,0x72,0x69,0x74,0x65,0x43,
0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x41,0x00,0x4B,0x45,0x52,0x4E,0x45,0x4C,0x33,0x32,
0x2E,0x64,0x6C,0x00,0x00,0xE1,0x03,0x77,0x73,0x70,0x72,0x69,0x6E,0x74,0x66,
0x41,0x00,0x55,0x53,0x45,0x52,0x33,0x32,0x2E,0x64,0x6C,0x6C,
};

#define a123_8_SIZE 108

#define a123_ZEROS_8 8844

unsigned char a123_9[] = {
    0x25,0x64,0x0A,0x0D,
};

#define a123_9_SIZE 4

#define a123_ZEROS_9 266

unsigned char a123_10[] = {
    0x01,0x00,0x18,0x00,0x00,0x00,0x18,0x00,0x00,0x80,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x00,0x01,0x00,0x00,0x00,0x30,0x00,
    0x00,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x01,0x00,0x09,0x04,0x00,0x00,0x48,0x00,0x00,0x00,0x60,0x80,0x00,0x00,0x7D,0x01,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x3C,0x3F,0x78,0x6D,0x6C,0x20,0x76,0x65,0x72,0x73,0x69,0x6F,0x6E,0x3D,
    0x27,0x31,0x2E,0x30,0x27,0x20,0x65,0x6E,0x63,0x6F,0x64,0x69,0x6E,0x67,0x3D,0x27,
    0x55,0x54,0x46,0x2D,0x38,0x27,0x20,0x73,0x74,0x61,0x6E,0x64,0x61,0x6C,0x6F,0x6E,
    0x65,0x3D,0x27,0x79,0x65,0x73,0x27,0x3F,0x3E,0x0D,0x0A,0x3C,0x61,0x73,0x73,0x65,
    0x6D,0x62,0x6C,0x79,0x20,0x78,0x6D,0x6C,0x6E,0x73,0x3D,0x27,0x75,0x72,0x6E,0x3A,
    0x73,0x63,0x68,0x65,0x6D,0x61,0x73,0x2D,0x6D,0x69,0x63,0x72,0x6F,0x73,0x6F,0x66,
    0x74,0x2D,0x63,0x6F,0x6D,0x3A,0x61,0x73,0x6D,0x2E,0x76,0x31,0x27,0x20,0x6D,0x61,
    0x6E,0x69,0x66,0x65,0x73,0x74,0x56,0x65,0x72,0x73,0x69,0x6F,0x6E,0x3D,0x27,0x31,
    0x2E,0x30,0x27,0x3E,0x0D,0x0A,0x20,0x20,0x3C,0x74,0x72,0x75,0x73,0x74,0x49,0x6E,
    0x66,0x6F,0x20,0x78,0x6D,0x6C,0x6E,0x73,0x3D,0x22,0x75,0x72,0x6E,0x3A,0x73,0x63,
    0x68,0x65,0x6D,0x61,0x73,0x2D,0x6D,0x69,0x63,0x72,0x6F,0x73,0x6F,0x66,0x74,0x2D,
    0x63,0x6F,0x6D,0x3A,0x61,0x73,0x6D,0x2E,0x76,0x33,0x22,0x3E,0x0D,0x0A,0x20,0x20,
    0x20,0x20,0x3C,0x73,0x65,0x63,0x75,0x72,0x69,0x74,0x79,0x3E,0x0D,0x0A,0x20,0x20,
    0x20,0x20,0x20,0x20,0x3C,0x72,0x65,0x71,0x75,0x65,0x73,0x74,0x65,0x64,0x50,0x72,
    0x69,0x76,0x69,0x6C,0x65,0x67,0x65,0x73,0x3E,0x0D,0x0A,0x20,0x20,0x20,0x20,0x20,
    0x20,0x20,0x20,0x20,0x3C,0x72,0x65,0x71,0x75,0x65,0x73,0x74,0x65,0x64,0x45,0x78,0x65,
    0x63,0x75,0x74,0x69,0x6F,0x6E,0x4C,0x65,0x76,0x65,0x6C,0x20,0x6C,0x65,0x76,0x65,
    0x6C,0x3D,0x27,0x61,0x73,0x49,0x6E,0x76,0x6F,0x6B,0x65,0x72,0x27,0x20,0x75,0x69,

```

```

0x41,0x63,0x65,0x73,0x73,0x3D,0x27,0x66,0x61,0x6C,0x73,0x65,0x27,0x20,0x2F,
0x3E,0x0D,0x0A,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x3C,0x2F,0x72,0x65,0x71,0x75,0x65,
0x73,0x74,0x65,0x64,0x50,0x72,0x69,0x76,0x69,0x6C,0x65,0x67,0x65,0x73,0x3E,0x0D,
0x0A,0x20,0x20,0x20,0x20,0x3C,0x2F,0x73,0x65,0x63,0x75,0x72,0x69,0x74,0x79,0x3E,
0x0D,0x0A,0x20,0x20,0x3C,0x2F,0x74,0x72,0x75,0x73,0x74,0x49,0x6E,0x66,0x6F,0x3E,
0x0D,0x0A,0x3C,0x2F,0x61,0x73,0x73,0x65,0x6D,0x62,0x6C,0x79,0x3E,0x0D,0x0A,
};

#define a123_10_SIZE 463

#define a123_ZEROS_10 36

#define a123_ZEROS_11 464

unsigned char a123_11[] = {
    0x10,0x00,0x00,0x20,0x00,0x00,0x00,0x1D,0x30,0x22,0x30,0x34,0x30,0x3A,0x30,0x4A,
    0x30,0x51,0x30,0x57,0x30,0x62,0x30,0x68,0x30,0x9F,0x30,0xAB,0x30,0x00,0x00,0x00,
    0x30,0x00,0x00,0x10,0x00,0x00,0x00,0xB4,0x30,0xBA,0x30,0xC0,0x30,0xC6,0x30,
};

#define a123_11_SIZE 47

unsigned long long int a123_array_part_count = 12;

unsigned char * a123[12] = {
    a123_0
    , a123_1
    , a123_2
    , a123_3
    , a123_4
    , a123_5
    , a123_6
    , a123_7
    , a123_8
    , a123_9
    , a123_10
    , a123_11
};

unsigned long long int a123_array_part_size[12] = {
    a123_0_SIZE
    , a123_1_SIZE
    , a123_2_SIZE
    , a123_3_SIZE
    , a123_4_SIZE
    , a123_5_SIZE
    , a123_6_SIZE

```

```

, a123_7_SIZE
, a123_8_SIZE
, a123_9_SIZE
, a123_10_SIZE
, a123_11_SIZE
};

unsigned long long int a123_zero_part_count = 12;

unsigned long long int a123_zeros[12] = {
    a123_ZEROS_0
, a123_ZEROS_1
, a123_ZEROS_2
, a123_ZEROS_3
, a123_ZEROS_4
, a123_ZEROS_5
, a123_ZEROS_6
, a123_ZEROS_7
, a123_ZEROS_8
, a123_ZEROS_9
, a123_ZEROS_10
, a123_ZEROS_11
};

Mod.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****
* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *
*
*          file: mod.cpp          *
*
*          (draft!) *
*****/

#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"
#include "stdio.h"

unsigned char* makeModCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) { // task
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_MOD);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\"%s\\\"\\r\\n\", tokenStruct[MULTI_TOKEN_MOD][0]);

#endif

    }

    return currBytePtr;
}

```

```

const unsigned char code__mov_eax_stackTopByECXMinus4[] = { 0x8B, 0x41, 0xFC }; // mov eax, dword ptr[ecx - 4]

const unsigned char code__cdq[] = { 0x99 }; // cdq

const unsigned char code__idiv_stackTopByECX[] = { 0xF7, 0x39 }; // idiv dword ptr [ecx]

const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 }; // sub ecx, 4

const unsigned char code__mov_eax_edx[] = { 0x8B, 0xC2 }; // mov eax, edx

const unsigned char code__mov_toAddrFromECX_eax[] = { 0x89, 0x01 }; // mov dword ptr [ecx], eax


currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECXMinus4, 3);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cdq, 1);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__idiv_stackTopByECX, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_edx, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_toAddrFromECX_eax, 2);


#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("  mov eax, dword ptr[ecx - 4]\r\n");
    printf("  cdq\r\n");
    printf("  idiv dword ptr [ecx]\r\n");
    printf("  sub ecx, 4\r\n");
    printf("  mov eax, edx\r\n");
    printf("  mov dword ptr [ecx], eax\r\n");

#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

Mul.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: mul.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_MUL);

```

```

        if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

            printf("\r\n");

            printf("    ;\\"%s\\"r\n", tokenStruct[MULTI_TOKEN_MUL][0]);

#endif

            const unsigned char code__mov_eax_stackTopByECXMinus4[] = { 0x8B, 0x41, 0xFC };

            const unsigned char code__imul_stackTopByECX[] = { 0xF7, 0x29 };

            const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };

            const unsigned char code__mov_toAddrFromECX_eax[] = { 0x89, 0x01 };

            currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECXMinus4, 3);

            //currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cdq, 1);

            currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__imul_stackTopByECX, 2);

            currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);

            //currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_eax, 1);

            //currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__dec_ebp, 1);

            currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_toAddrFromECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

            printf("    mov eax, dword ptr[ecx - 4]\r\n");

            printf("    ;cdq\r\n");

            printf("    imul dword ptr [ecx]\r\n");

            printf("    sub ecx, 4\r\n");

            printf("    mov dword ptr [ecx], eax\r\n");

#endif

            return *lastLexemInfoInTable += multitokenSize, currBytePtr;

        }

        return currBytePtr;

    }

Not.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: not.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

```

```

#include "stdio.h"

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_NOT);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\\%s\\r\n", tokenStruct[MULTI_TOKEN_NOT][0]);

#endif

        const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

        const unsigned char code__cmp_eax_0[] = { 0x83, 0xF8, 0x00 };

        const unsigned char code__sete_al[] = { 0x0F, 0x94, 0xC0 };

        const unsigned char code__and_eax_1[] = { 0x83, 0xE0, 0x01 };

        //

        const unsigned char code__mov_stackTopByECX_eax[] = { 0x89, 0x01 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_eax_0, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sete_al, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_eax_1, 3);

        //

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_stackTopByECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("    mov eax, dword ptr[ecx]\\r\n");

        printf("    cmp eax, 0\\r\n");

        printf("    sete al\\r\n");

        printf("    and eax, 1\\r\n");

        //

        printf("    mov dword ptr[ecx], eax\\r\n");

#endif

    }

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

Not_equal.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*               file: not_equal.cpp               *

*****/

```



```

*                               (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

unsigned char* makeIsNotEqualCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_NOT_EQUAL);
    if (multitokenSize) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("\r\n");
        printf("    ;\\\"%s\\\"\\r\n", tokenStruct[MULTI_TOKEN_NOT_EQUAL][0]);
#endif

        const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };
        const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };
        const unsigned char code__cmp_stackTopByECX_eax[] = { 0x39, 0x01 };
        const unsigned char code__setne_al[] = { 0x0F, 0x95, 0xC0 };
        const unsigned char code__and_eax_1[] = { 0x83, 0xE0, 0x01 };
        const unsigned char code__mov_stackTopByECX_eax[] = { 0x89, 0x01 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_stackTopByECX_eax, 2);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__setne_al, 3);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_eax_1, 3);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_stackTopByECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("    mov eax, dword ptr[ecx]\\r\n");
        printf("    sub ecx, 4\\r\n");
        printf("    cmp dword ptr[ecx], eax\\r\n");
        printf("    setne al\\r\n");
        printf("    and eax, 1\\r\n");
        printf("    mov dword ptr[ecx], eax\\r\n");
#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;
    }

    return currBytePtr;
}

```

```

}

Null.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: null_statement.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_NULL_STATEMENT);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;null statement (non-context)\r\n");

#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    return currBytePtr;

}

Objectgen_pattern.cpp

// Generated sparse arrays

unsigned char o123_0[] = {

    0x4C,0x01,0x05,0x00,0x85,0x62,0x84,0x67,0x86,0x4B,0x00,0x00,0x1F,0x00,0x00,0x00,

    0x00,0x00,0x00,0x00,0x2E,0x74,0x65,0x78,0x74,0x24,0x6D,0x6E,0x00,0x00,0x00,0x00,

    0x00,0x00,0x00,0x00,0xB2,0x20,0x00,0x00,0xDC,0x00,0x00,0x00,0x8E,0x21,0x00,0x00,

    0x00,0x00,0x00,0x00,0x10,0x00,0x00,0x00,0x20,0x00,0x50,0x60,0x2E,0x64,0x61,0x74,

    0x61,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x91,0x21,0x00,0x00,

    0x2E,0x22,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

    0x40,0x00,0x50,0xC0,0x2E,0x64,0x65,0x62,0x75,0x67,0x24,0x53,0x00,0x00,0x00,0x00,

    0x00,0x00,0x00,0x00,0x4C,0x05,0x00,0x00,0xBF,0x43,0x00,0x00,0x0C,0x49,0x00,0x00,

    0x00,0x00,0x00,0x00,0x1E,0x00,0x00,0x00,0x40,0x00,0x10,0x42,0x2E,0x64,0x65,0x62,

    0x75,0x67,0x24,0x54,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0x01,0x00,0x00,

    0x38,0x4A,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

    0x40,0x00,0x10,0x42,0x2E,0x64,0x72,0x65,0x63,0x74,0x76,0x65,0x00,0x00,0x00,0x00,

    0x00,0x00,0x00,0x00,0x0D,0x00,0x00,0x00,0x78,0x4B,0x00,0x00,0x00,0x00,0x00,0x00,

```

```

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0A,0x00,0x00,0xE8,0x00,0x00,0x00,
0x00,0x5E,0x83,0xEE,0x05,0x8B,0xFE,0x81,0xC7,0x00,0x40,0x00,0x00,0x8B,0xCF,0x81,
0xC1,0x00,0x02,0x00,0x00,0xEB,0x7C,0x50,0x68,0x00,0x00,0x00,0x00,0x68,0x00,0x00,
0x00,0x00,0xE8,0x00,0x00,0x00,0x00,0x83,0xC4,0x0C,0x6A,0x00,0x6A,0x00,0x50,0x68,
0x00,0x00,0x00,0x00,0xFF,0x35,0x00,0x00,0x00,0x00,0xE8,0x00,0x00,0x00,0x00,0xC3,
0x55,0x8B,0xEC,0x6A,0x00,0x68,0x00,0x00,0x00,0x00,0x6A,0x0F,0x68,0x01,0x00,0x00,
0x00,0xFF,0x35,0x00,0x00,0x00,0x00,0xE8,0x00,0x00,0x00,0x00,0x8D,0x35,0x00,0x00,
0x00,0x00,0x03,0x35,0x00,0x00,0x00,0x00,0x83,0xEE,0x02,0xE8,0x04,0x00,0x00,0x00,
0x8B,0xE5,0x5D,0xC3,0x33,0xC0,0xBB,0x01,0x00,0x00,0x00,0x33,0xC9,0x0F,0xB6,0x0E,
0x85,0xC9,0x74,0x0E,0x83,0xE9,0x30,0x0F,0xAF,0xCB,0x03,0xC1,0x6B,0xDB,0x0A,0x4E,
0xEB,0xEB,0xC3,0x6A,0xF6,0xE8,0x00,0x00,0x00,0x00,0xA3,0x00,0x00,0x00,0x6A,
0xF5,0xE8,0x00,0x00,0x00,0x00,0xA3,0x00,0x00,0x00,0x00,0x33,0xC0,0xC3,

```

```
};
```

```
#define o123_0_SIZE 398
```

```
#define o123_ZEROS_0 8192
```

```
unsigned char o123_1[] = {
```

```

0x1D,0x00,0x00,0x00,0x14,0x00,0x00,0x00,0x06,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,
0x00,0x00,0x06,0x00,0x27,0x00,0x00,0x00,0x0D,0x00,0x00,0x00,0x14,0x00,0x34,0x00,
0x00,0x00,0x0E,0x00,0x00,0x00,0x06,0x00,0x3A,0x00,0x00,0x00,0x10,0x00,0x00,0x00,
0x06,0x00,0x3F,0x00,0x00,0x00,0x0C,0x00,0x00,0x00,0x14,0x00,0x4A,0x00,0x00,0x00,
0x11,0x00,0x00,0x00,0x06,0x00,0x51,0x00,0x00,0x00,0x12,0x00,0x00,0x00,0x06,0x00,
0x57,0x00,0x00,0x00,0x0F,0x00,0x00,0x00,0x06,0x00,0x5C,0x00,0x00,0x00,0x0B,0x00,
0x00,0x00,0x14,0x00,0x62,0x00,0x00,0x00,0x12,0x00,0x00,0x00,0x06,0x00,0x68,0x00,
0x00,0x00,0x11,0x00,0x00,0x00,0x06,0x00,0x9A,0x00,0x00,0x00,0x0A,0x00,0x00,0x00,
0x14,0x00,0x9F,0x00,0x00,0x00,0x0F,0x00,0x00,0x00,0x06,0x00,0xA6,0x00,0x00,0x00,
0x0A,0x00,0x00,0x00,0x14,0x00,0xAB,0x00,0x00,0x00,0x10,0x00,0x00,0x00,0x06,

```

```
};
```

```
#define o123_1_SIZE 159
```

```
#define o123_ZEROS_1 8449
```

```
unsigned char o123_2[] = {
```

```
0x25,0x64,0x0A,0x0D,
```

```
};
```

```
#define o123_2_SIZE 4
```

```
#define o123_ZEROS_2 141
```

```
unsigned char o123_3[] = {
```

```

0x04,0x00,0x00,0x00,0xF3,0x00,0x00,0x00,0x3C,0x00,0x00,0x00,0x43,0x3A,0x5C,
0x55,0x73,0x65,0x72,0x73,0x5C,0x4E,0x61,0x7A,0x61,0x72,0x5C,0x73,0x6F,0x75,0x72,

```

0x63,0x65,0x5C,0x72,0x65,0x70,0x6F,0x73,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,  
0x36,0x30,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x5C,0x73,0x6F,0x75,  
0x72,0x63,0x65,0x2E,0x61,0x73,0x6D,0x00,0xF4,0x00,0x00,0x00,0x18,0x00,0x00,0x00,  
0x01,0x00,0x00,0x00,0x10,0x01,0xD3,0x4E,0x23,0x32,0x9C,0x94,0x3A,0xC3,0x61,0x14,  
0xF6,0x0B,0x9E,0xC4,0xCE,0x14,0x00,0x00,0xF2,0x00,0x00,0x00,0xE8,0x01,0x00,0x00,  
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xB2,0x20,0x00,0x00,0x00,0x00,0x00,0x00,  
0x3A,0x00,0x00,0x00,0xDC,0x01,0x00,0x00,0x05,0x00,0x00,0x00,0x23,0x00,0x00,0x80,  
0x06,0x00,0x00,0x00,0x24,0x00,0x00,0x80,0x09,0x00,0x00,0x00,0x25,0x00,0x00,0x80,  
0x0B,0x00,0x00,0x00,0x26,0x00,0x00,0x80,0x11,0x00,0x00,0x00,0x27,0x00,0x00,0x80,  
0x13,0x00,0x00,0x00,0x28,0x00,0x00,0x80,0x19,0x00,0x00,0x00,0x29,0x00,0x00,0x80,  
0x1B,0x00,0x00,0x00,0x2A,0x00,0x00,0x80,0x1B,0x00,0x00,0x00,0x2B,0x00,0x00,0x80,  
0x1C,0x00,0x00,0x00,0x2C,0x00,0x00,0x80,0x21,0x00,0x00,0x00,0x2D,0x00,0x00,0x80,  
0x26,0x00,0x00,0x00,0x2E,0x00,0x00,0x80,0x2B,0x00,0x00,0x00,0x2F,0x00,0x00,0x80,  
0x2E,0x00,0x00,0x00,0x37,0x00,0x00,0x80,0x30,0x00,0x00,0x00,0x38,0x00,0x00,0x80,  
0x32,0x00,0x00,0x00,0x39,0x00,0x00,0x80,0x33,0x00,0x00,0x00,0x3A,0x00,0x00,0x80,  
0x38,0x00,0x00,0x00,0x3B,0x00,0x00,0x80,0x3E,0x00,0x00,0x00,0x3C,0x00,0x00,0x80,  
0x43,0x00,0x00,0x00,0x3E,0x00,0x00,0x80,0x44,0x00,0x00,0x00,0x42,0x00,0x00,0x80,  
0x44,0x00,0x00,0x00,0x43,0x00,0x00,0x80,0x45,0x00,0x00,0x00,0x44,0x00,0x00,0x80,  
0x47,0x00,0x00,0x00,0x46,0x00,0x00,0x80,0x49,0x00,0x00,0x00,0x47,0x00,0x00,0x80,  
0x4E,0x00,0x00,0x00,0x48,0x00,0x00,0x80,0x50,0x00,0x00,0x00,0x49,0x00,0x00,0x80,  
0x55,0x00,0x00,0x00,0x4A,0x00,0x00,0x80,0x5B,0x00,0x00,0x00,0x4B,0x00,0x00,0x80,  
0x60,0x00,0x00,0x00,0x4D,0x00,0x00,0x80,0x66,0x00,0x00,0x00,0x4E,0x00,0x00,0x80,  
0x6C,0x00,0x00,0x00,0x4F,0x00,0x00,0x80,0x6F,0x00,0x00,0x00,0x50,0x00,0x00,0x80,  
0x74,0x00,0x00,0x00,0x52,0x00,0x00,0x80,0x76,0x00,0x00,0x00,0x53,0x00,0x00,0x80,  
0x77,0x00,0x00,0x00,0x54,0x00,0x00,0x80,0x78,0x00,0x00,0x00,0x57,0x00,0x00,0x80,  
0x78,0x00,0x00,0x00,0x5A,0x00,0x00,0x80,0x7A,0x00,0x00,0x00,0x5B,0x00,0x00,0x80,  
0x7F,0x00,0x00,0x00,0x5C,0x00,0x00,0x80,0x81,0x00,0x00,0x00,0x5F,0x00,0x00,0x80,  
0x84,0x00,0x00,0x00,0x60,0x00,0x00,0x80,0x86,0x00,0x00,0x00,0x61,0x00,0x00,0x80,  
0x88,0x00,0x00,0x00,0x62,0x00,0x00,0x80,0x8B,0x00,0x00,0x00,0x63,0x00,0x00,0x80,  
0x8E,0x00,0x00,0x00,0x64,0x00,0x00,0x80,0x90,0x00,0x00,0x00,0x65,0x00,0x00,0x80,  
0x93,0x00,0x00,0x00,0x66,0x00,0x00,0x80,0x94,0x00,0x00,0x00,0x67,0x00,0x00,0x80,  
0x96,0x00,0x00,0x00,0x6A,0x00,0x00,0x80,0x97,0x00,0x00,0x00,0x6E,0x00,0x00,0x80,  
0x99,0x00,0x00,0x00,0x6F,0x00,0x00,0x80,0x9E,0x00,0x00,0x00,0x70,0x00,0x00,0x80,  
0xA3,0x00,0x00,0x00,0x71,0x00,0x00,0x80,0xA5,0x00,0x00,0x00,0x72,0x00,0x00,0x80,  
0xAA,0x00,0x00,0x00,0x73,0x00,0x00,0x80,0xAF,0x00,0x00,0x00,0x87,0x00,0x00,0x80,  
0xB1,0x00,0x00,0x00,0x88,0x00,0x00,0x80,0xF1,0x00,0x00,0x00,0xEA,0x02,0x00,0x00,  
0x49,0x00,0x01,0x11,0x00,0x00,0x00,0x00,0x43,0x3A,0x5C,0x55,0x73,0x65,0x72,0x73,  
0x5C,0x4E,0x61,0x7A,0x61,0x72,0x5C,0x73,0x6F,0x75,0x72,0x63,0x65,0x5C,0x72,0x65,  
0x70,0x6F,0x73,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x5C,0x50,0x72,  
0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x5C,0x52,0x65,0x6C,0x65,0x61,0x73,0x65,0x5C,  
0x73,0x6F,0x75,0x72,0x63,0x65,0x2E,0x6F,0x62,0x6A,0x00,0x37,0x00,0x3C,0x11,0x03,  
0x00,0x00,0x00,0x06,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0E,0x00,0x1C,  
0x00,0xE4,0x74,0x00,0x00,0x4D,0x69,0x63,0x72,0x6F,0x73,0x6F,0x66,0x74,0x20,0x28,

0x52,0x29,0x20,0x4D,0x61,0x63,0x72,0x6F,0x20,0x41,0x73,0x73,0x65,0x6D,0x62,0x6C,  
0x65,0x72,0x00,0x00,0xBB,0x00,0x3D,0x11,0x00,0x63,0x77,0x64,0x00,0x43,0x3A,0x5C,  
0x55,0x73,0x65,0x72,0x73,0x5C,0x4E,0x61,0x7A,0x61,0x72,0x5C,0x73,0x6F,0x75,0x72,  
0x63,0x65,0x5C,0x72,0x65,0x70,0x6F,0x73,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,  
0x36,0x30,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x00,0x65,0x78,0x65,  
0x00,0x43,0x3A,0x5C,0x50,0x72,0x6F,0x67,0x72,0x61,0x6D,0x20,0x46,0x69,0x6C,0x65,  
0x73,0x20,0x28,0x78,0x38,0x36,0x29,0x5C,0x4D,0x69,0x63,0x72,0x6F,0x73,0x6F,0x66,  
0x74,0x20,0x56,0x69,0x73,0x75,0x61,0x6C,0x20,0x53,0x74,0x75,0x64,0x69,0x6F,0x5C,  
0x32,0x30,0x31,0x39,0x5C,0x45,0x6E,0x74,0x65,0x72,0x70,0x72,0x69,0x73,0x65,0x5C,  
0x56,0x43,0x5C,0x54,0x6F,0x6F,0x6C,0x73,0x5C,0x4D,0x53,0x56,0x43,0x5C,0x31,0x34,  
0x2E,0x32,0x38,0x2E,0x32,0x39,0x39,0x31,0x30,0x5C,0x62,0x69,0x6E,0x5C,0x48,0x6F,  
0x73,0x74,0x58,0x38,0x36,0x5C,0x78,0x38,0x36,0x5C,0x6D,0x6C,0x2E,0x65,0x78,0x65,  
0x00,0x73,0x72,0x63,0x00,0x73,0x6F,0x75,0x72,0x63,0x65,0x2E,0x61,0x73,0x6D,0x00,  
0x00,0x1A,0x00,0x0C,0x11,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x76,  
0x61,0x6C,0x75,0x65,0x54,0x65,0x6D,0x70,0x5F,0x6D,0x73,0x67,0x00,0x1A,0x00,0x0C,  
0x11,0x22,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x68,0x43,0x6F,0x6E,0x73,  
0x6F,0x6C,0x65,0x49,0x6E,0x70,0x75,0x74,0x00,0x1B,0x00,0x0C,0x11,0x22,0x00,0x00,  
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x68,0x43,0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x4F,  
0x75,0x74,0x70,0x75,0x74,0x00,0x2D,0x00,0x10,0x11,0x00,0x00,0x00,0x00,0x00,0x00,  
0x00,0x00,0x00,0x00,0x00,0x00,0x34,0x00,0x00,0x00,0x00,0x00,0x00,0x34,0x00,  
0x00,0x00,0x06,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x67,0x65,0x74,  
0x50,0x72,0x6F,0x63,0x00,0x02,0x00,0x06,0x00,0x2D,0x00,0x10,0x11,0x00,0x00,0x00,  
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x29,0x00,0x00,0x00,0x00,0x00,  
0x00,0x29,0x00,0x00,0x00,0x08,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  
0x70,0x75,0x74,0x50,0x72,0x6F,0x63,0x00,0x02,0x00,0x06,0x00,0x15,0x00,0x05,0x11,  
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x69,0x6E,0x69,0x74,0x43,0x6F,0x6E,0x73,0x6F,  
0x6C,0x65,0x00,0x19,0x00,0x0C,0x11,0x22,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  
0x00,0x72,0x65,0x61,0x64,0x4F,0x75,0x74,0x43,0x6F,0x75,0x6E,0x74,0x00,0x13,0x00,  
0x0C,0x11,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x62,0x75,0x66,0x66,  
0x65,0x72,0x00,0x0F,0x00,0x05,0x11,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x73,0x74,  
0x61,0x72,0x74,0x00,0x33,0x00,0x10,0x11,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  
0x00,0x00,0x00,0x00,0x1F,0x00,0x00,0x00,0x00,0x00,0x00,0x1F,0x00,0x00,0x00,  
0x10,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x73,0x74,0x72,0x69,0x6E,  
0x67,0x5F,0x74,0x6F,0x5F,0x69,0x6E,0x74,0x00,0x16,0x00,0x05,0x11,0x00,0x00,0x00,  
0x00,0x00,0x00,0x00,0x63,0x6F,0x6E,0x76,0x65,0x72,0x74,0x5F,0x6C,0x6F,0x6F,0x70,  
0x00,0x0E,0x00,0x05,0x11,0x00,0x00,0x00,0x00,0x00,0x00,0x64,0x6F,0x6E,0x65,  
0x00,0x02,0x00,0x06,0x00,0x17,0x00,0x0C,0x11,0x20,0x00,0x00,0x00,0x00,0x00,0x00,  
0x00,0x00,0x00,0x64,0x61,0x74,0x61,0x5F,0x73,0x74,0x61,0x72,0x74,0x00,0x1A,0x00,  
0x0C,0x11,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x76,0x61,0x6C,0x75,  
0x65,0x54,0x65,0x6D,0x70,0x5F,0x66,0x6D,0x74,0x00,0x00,0x00,0x70,0x00,0x00,  
0x00,0x18,0x00,0x00,0x00,0x0B,0x00,0x74,0x00,0x00,0x00,0x18,0x00,0x00,0x00,0x0A,  
0x00,0xA9,0x03,0x00,0x00,0x0E,0x00,0x00,0x00,0x0B,0x00,0xAD,0x03,0x00,0x00,0x0E,  
0x00,0x00,0x00,0x0A,0x00,0xC5,0x03,0x00,0x00,0x0F,0x00,0x00,0x00,0x0B,0x00,0xC9,

0x03,0x00,0x00,0x0F,0x00,0x00,0x00,0x0A,0x00,0xE1,0x03,0x00,0x00,0x10,0x00,0x00,  
0x00,0x0B,0x00,0xE5,0x03,0x00,0x00,0x10,0x00,0x00,0x00,0x0A,0x00,0x16,0x04,0x00,  
0x00,0x16,0x00,0x00,0x00,0x0B,0x00,0x1A,0x04,0x00,0x00,0x16,0x00,0x00,0x00,0x0A,  
0x00,0x49,0x04,0x00,0x00,0x15,0x00,0x00,0x0B,0x00,0x4D,0x04,0x00,0x00,0x15,  
0x00,0x00,0x00,0x0A,0x00,0x60,0x04,0x00,0x00,0x19,0x00,0x00,0x00,0x0B,0x00,0x64,  
0x04,0x00,0x00,0x19,0x00,0x00,0x00,0x0A,0x00,0x7B,0x04,0x00,0x00,0x11,0x00,0x00,  
0x00,0x0B,0x00,0x7F,0x04,0x00,0x00,0x11,0x00,0x00,0x00,0x0A,0x00,0x96,0x04,0x00,  
0x00,0x12,0x00,0x00,0x00,0x0B,0x00,0x9A,0x04,0x00,0x00,0x12,0x00,0x00,0x00,0x0A,  
0x00,0xA7,0x04,0x00,0x00,0x13,0x00,0x00,0x00,0x0B,0x00,0xAB,0x04,0x00,0x00,0x13,  
0x00,0x00,0x00,0x0A,0x00,0xD4,0x04,0x00,0x00,0x17,0x00,0x00,0x00,0x0B,0x00,0xD8,  
0x04,0x00,0x00,0x17,0x00,0x00,0x00,0x0A,0x00,0xED,0x04,0x00,0x00,0x1A,0x00,0x00,  
0x00,0x0B,0x00,0xF1,0x04,0x00,0x00,0x1A,0x00,0x00,0x00,0x0A,0x00,0x05,0x05,0x00,  
0x00,0x1B,0x00,0x00,0x00,0x0B,0x00,0x09,0x05,0x00,0x00,0x1B,0x00,0x00,0x00,0x0A,  
0x00,0x1D,0x05,0x00,0x00,0x1C,0x00,0x00,0x00,0x0B,0x00,0x21,0x05,0x00,0x00,0x1C,  
0x00,0x00,0x00,0x0A,0x00,0x36,0x05,0x00,0x00,0x14,0x00,0x00,0x00,0x0B,0x00,0x3A,  
0x05,0x00,0x00,0x14,0x00,0x00,0x00,0x0A,0x00,0x04,0x00,0x00,0x0A,0x00,0x01,  
0x12,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,  
0x00,0x00,0x00,0x01,0x00,0x00,0x10,0x00,0x00,0x06,0x00,0x0E,0x00,0x00,0x00,0xF2,  
0xF1,0x1A,0x00,0x01,0x12,0x05,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,  
0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,  
0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x05,0x00,0x03,0x10,0x00,0x00,0x06,0x00,0x01,  
0x12,0x00,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x00,  
0x00,0x05,0x10,0x00,0x00,0x06,0x00,0x01,0x12,0x00,0x00,0x00,0x0E,0x00,0x08,  
0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x00,0x00,0x07,0x10,0x00,0x00,0x0A,0x00,0x01,  
0x12,0x01,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,  
0x00,0x07,0x00,0x01,0x00,0x09,0x10,0x00,0x00,0x0E,0x00,0x01,0x12,0x02,0x00,0x00,  
0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,  
0x00,0x07,0x00,0x02,0x00,0x0B,0x10,0x00,0x00,0x0E,0x00,0x01,0x12,0x02,0x00,0x00,  
0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,  
0x00,0x07,0x00,0x02,0x00,0x0D,0x10,0x00,0x00,0x06,0x00,0x01,0x12,0x00,0x00,0x00,  
0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x00,0x00,0x0F,0x10,0x00,  
0x00,0x1A,0x00,0x01,0x12,0x05,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,  
0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,  
0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x05,0x00,0x11,0x10,0x00,0x00,0x0A,0x00,0x01,  
0x12,0x01,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,  
0x00,0x07,0x00,0x01,0x00,0x13,0x10,0x00,0x00,0x2F,0x45,0x4E,0x54,0x52,0x59,0x3A,  
0x73,0x74,0x61,0x72,0x74,0x20,0x00,0x40,0x63,0x6F,0x6D,0x70,0x2E,0x69,0x64,0xE4,  
0x74,0x03,0x01,0xFF,0xFF,0x00,0x00,0x03,0x00,0x40,0x66,0x65,0x61,0x74,0x2E,0x30,  
0x30,0x10,0x00,0x00,0x00,0xFF,0xFF,0x00,0x00,0x03,0x00,0x2E,0x74,0x65,0x78,0x74,  
0x24,0x6D,0x6E,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x03,0x01,0xB2,0x20,0x00,  
0x00,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x2E,  
0x64,0x61,0x74,0x61,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x00,0x00,0x00,0x03,  
0x01,0x91,0x21,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

```

0x00,0x00,0x00,0x2E,0x64,0x65,0x62,0x75,0x67,0x24,0x53,0x00,0x00,0x00,0x00,0x03,
0x00,0x00,0x00,0x03,0x01,0x4C,0x05,0x00,0x00,0x1E,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x2E,0x64,0x65,0x62,0x75,0x67,0x24,0x54,0x00,
0x00,0x00,0x00,0x04,0x00,0x00,0x00,0x03,0x01,0x40,0x01,
};

#define o123_3_SIZE 2155

#define o123_ZEROS_3 20

unsigned char o123_4[] = {
0x04,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x00,0x02,0x00,0x00,0x00,
0x00,0x00,0x14,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x00,0x02,0x00,
0x00,0x00,0x00,0x00,0x25,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x00,
0x02,0x00,0x00,0x00,0x00,0x37,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x20,0x00,0x02,0x00,0x00,0x00,0x00,0x42,0x00,0x00,0x00,0x20,0x00,0x00,
0x02,0x00,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0x50,0x00,0x00,0x05,0x21,
0x00,0x00,0x02,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0x5E,0x00,0x00,0x00,
0x09,0x21,0x00,0x00,0x02,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0x6D,0x00,
0x00,0x00,0x8D,0x21,0x00,0x00,0x02,0x00,0x00,0x03,0x00,0x62,0x75,0x66,0x66,
0x65,0x72,0x00,0x00,0x0D,0x21,0x00,0x00,0x02,0x00,0x00,0x03,0x00,0x5F,0x73,
0x74,0x61,0x72,0x74,0x00,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x02,0x00,
0x00,0x00,0x00,0x00,0x7A,0x00,0x00,0x00,0x00,0x21,0x00,0x00,0x02,0x00,0x00,0x00,
0x03,0x00,0x00,0x00,0x00,0x88,0x00,0x00,0x00,0x1B,0x00,0x00,0x00,0x01,0x00,
0x20,0x00,0x02,0x00,0x00,0x00,0x00,0x93,0x00,0x00,0x00,0x44,0x00,0x00,0x00,
0x01,0x00,0x20,0x00,0x02,0x00,0x00,0x00,0x00,0x9E,0x00,0x00,0x00,0x78,0x00,
0x00,0x00,0x01,0x00,0x20,0x00,0x02,0x00,0x24,0x24,0x30,0x30,0x30,0x30,0x30,
0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0xAF,0x00,
0x00,0x00,0x97,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x06,0x00,0x00,0x00,0x00,
0xBB,0x00,0x00,0x00,0x81,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x06,0x00,0x64,0x6F,
0x6E,0x65,0x00,0x00,0x00,0x96,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x06,0x00,
0x00,0x00,0x00,0x00,0xC8,0x00,0x00,0x00,0x00,0x00,0x02,0x00,0x00,0x00,
0x03,0x00,0x2E,0x64,0x72,0x65,0x63,0x74,0x76,0x65,0x00,0x00,0x00,0x05,0x00,
0x00,0x00,0x03,0x01,0x0D,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0xD3,0x00,0x00,0x5F,0x47,0x65,0x74,0x53,0x74,
0x64,0x48,0x61,0x6E,0x64,0x6C,0x65,0x40,0x34,0x00,0x5F,0x52,0x65,0x61,0x64,0x43,
0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x41,0x40,0x32,0x30,0x00,0x5F,0x57,0x72,0x69,0x74,
0x65,0x43,0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x41,0x40,0x32,0x30,0x00,0x5F,0x77,0x73,
0x70,0x72,0x69,0x6E,0x74,0x66,0x41,0x00,0x76,0x61,0x6C,0x75,0x65,0x54,0x65,0x6D,
0x70,0x5F,0x6D,0x73,0x67,0x00,0x68,0x43,0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x49,0x6E,
0x70,0x75,0x74,0x00,0x68,0x43,0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x4F,0x75,0x74,0x70,
0x75,0x74,0x00,0x72,0x65,0x61,0x64,0x4F,0x75,0x74,0x43,0x6F,0x75,0x6E,0x74,0x00,
0x76,0x61,0x6C,0x75,0x65,0x54,0x65,0x6D,0x70,0x5F,0x66,0x6D,0x74,0x00,0x5F,0x70,
0x75,0x74,0x50,0x72,0x6F,0x63,0x40,0x30,0x00,0x5F,0x67,0x65,0x74,0x50,0x72,0x6F,

```

$$\};$$

```
unsigned long long int o123_array_part_count = 5;
```

o123\_0

, o123\_1

, o123\_2

, o123\_3

, o123\_4

$$\};$$

o123\_0\_SIZE

```
, o123_1_SIZE
```

```
, o123_2_SIZE
```

```
, o123_3_SIZE
```

```
, o123_4_SIZE
```

$$\};$$

```
unsigned long long int o123_zero_part_count = 4;
```

o123\_ZEROS\_0

, o123\_ZEROS\_1

, o123\_ZEROS\_2

, o123\_ZEROS\_3

$$\};$$

Operand.cpp

```
#define _CRT_SECURE_NO_WARNINGS
```

/\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_2024\_2025 \*

```
* file: identifier_or_value.cpp *
```

\* (draft!) \*

\*\*\*\*\*/

```
#include "../include/def.h"
```

```
#include "../include/generator/generator.h"
```

```
#include "../include/lexica/lexica.h"
```

```
#include "stdio.h"
```

```
#include "stdlib.h"
```



```
#include "string.h"
```

```
unsigned char* makeValueCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    if ((*lastLexemInfoInTable)->tokenType == VALUE_LEXEME_TYPE) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("\r\n");
        printf("    ;\"%lld\"\r\n", (*lastLexemInfoInTable)->ifvalue);
#endif

        const unsigned char code__add_ecx_4[] = { 0x83, 0xC1, 0x04 };
        const unsigned char code__mov_eax_value[] = { 0xB8, 0x00, 0x00, 0x00, 0x00 };
        unsigned char code__mov_toAddrFromECX_eax[] = { 0x89, 0x01 };
//        const unsigned char* valueParts = (const unsigned char*)&(*lastLexemInfoInTable)->ifvalue;
//        code__mov_toAddrFromECX_value[2] = valueParts[0];
//        code__mov_toAddrFromECX_value[3] = valueParts[1];
//        code__mov_toAddrFromECX_value[4] = valueParts[2];
//        code__mov_toAddrFromECX_value[5] = valueParts[3];

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_ecx_4, 3);
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_value, 5);
        *(unsigned int*)(currBytePtr - 4) = (unsigned int)(*lastLexemInfoInTable)->ifvalue;
        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_toAddrFromECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("    add ecx, 4\r\n");
        printf("    mov eax, 0%08Xh\r\n", (int)(*lastLexemInfoInTable)->ifvalue);
        printf("    mov dword ptr [ecx], eax\r\n");
#endif

        return ++ *lastLexemInfoInTable, currBytePtr;
    }

    return currBytePtr;
}
```

```
unsigned char* makeIdentifierCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    if ((*lastLexemInfoInTable)->tokenType == IDENTIFIER_LEXEME_TYPE) {
        bool findComplete = false;
        unsigned long long int variableIndex = 0;
        for (; identifierIdsTable[variableIndex][0] != '\0'; ++variableIndex) {
            if (!strcmp((*lastLexemInfoInTable)->lexemStr, identifierIdsTable[variableIndex], MAX_LEXEM_SIZE)) {
                findComplete = true;
                break;
            }
        }
    }
}
```

```

    }

    if (!findComplete) {
        printf("\r\nError!\r\n");
        exit(0);
    }

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("\r\n");
    printf("    ;\\"%s\\"r\n", (*lastLexemInfoInTable)->lexemStr);

#endif

    variableIndex *= VALUE_SIZE;

    unsigned char code__mov_eax_edi[] = { 0x8B, 0xC7 };
    unsigned char code__add_eax_variableOffsetInDataSection[] = { 0x05, 0x00, 0x00, 0x00, 0x00 };
    const unsigned char code__mov_eax_valueByAdressInEAX[] = { 0x8B, 0x00 };
    const unsigned char code__add_ecx_4[] = { 0x83, 0xC1, 0x04 };
    const unsigned char code__mov_toAddrFromECX_eax[] = { 0x89, 0x01 };
    const unsigned char* variableIndexValueParts = (const unsigned char*)&variableIndex;
    code__add_eax_variableOffsetInDataSection[1] = variableIndexValueParts[0];
    code__add_eax_variableOffsetInDataSection[2] = variableIndexValueParts[1];
    code__add_eax_variableOffsetInDataSection[3] = variableIndexValueParts[2];
    code__add_eax_variableOffsetInDataSection[4] = variableIndexValueParts[3];

    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_edi, 2);
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_eax_variableOffsetInDataSection, 5);
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_valueByAdressInEAX, 2);
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_ecx_4, 3);
    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_toAddrFromECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("    mov eax, edi\r\n");
    printf("    add eax, 0%08Xh\r\n", (int)variableIndex);
    printf("    mov eax, dword ptr[eax]\r\n");
    printf("    add ecx, 4\r\n");
    printf("    mov dword ptr [ecx], eax\r\n");

#endif

    return ++ * lastLexemInfoInTable, currBytePtr;
}

return currBytePtr;

}

Or.cpp

```

```

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: or.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_OR);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\"%s\\\"r\\n\", tokenStruct[MULTI_TOKEN_OR][0]);

#endif

        const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

        const unsigned char code__cmp_eax_0[] = { 0x83, 0xF8, 0x00 };

        const unsigned char code__setne_al[] = { 0x0F, 0x95, 0xC0 };

        const unsigned char code__and_eax_1[] = { 0x83, 0xE0, 0x01 };

        const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };

        //

        const unsigned char code__cmp_stackTopByECX_0[] = { 0x83, 0x39, 0x00 };

        const unsigned char code__setne_dl[] = { 0x0F, 0x95, 0xC2 };

        const unsigned char code__and_edx_1[] = { 0x83, 0xE2, 0x01 };

        //

        const unsigned char code__or_eax_edx[] = { 0x0B, 0xC2 };

        //

        const unsigned char code__mov_stackTopByECX_eax[] = { 0x89, 0x01 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_eax_0, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__setne_al, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_eax_1, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);

        //

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_stackTopByECX_0, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__setne_dl, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__and_edx_1, 3);

```

```

//
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__or_eax_edx, 2);

//
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_stackTopByECX_eax, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("  mov eax, dword ptr[ecx]\r\n");

    printf("  cmp eax, 0\r\n");

    printf("  setne al\r\n");

    printf("  and eax, 1\r\n");

    printf("  sub ecx, 4\r\n");

    //

    printf("  cmp dword ptr[ecx], 0\r\n");

    printf("  setne dl\r\n");

    printf("  and edx, 1\r\n");

    //

    printf("  or eax, edx\r\n");

    //

    printf("  mov dword ptr[ecx], eax\r\n");

#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

Output.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: output.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_OUTPUT);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

```

```

printf("\r\n");

printf("    ;\\\"%s\\\"\\r\\n", tokenStruct[MULTI_TOKEN_OUTPUT][0]);

#endif

const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

const unsigned char code__mov_edx_address[] = { 0xBA, 0x00, 0x00, 0x00, 0x00 };

const unsigned char code__add_edx_esi[] = { 0x03, 0xD6 };

//const unsigned char code__push_ecx[] = { 0x51 };

//const unsigned char code__push_ebx[] = { 0x53 };

const unsigned char code__push_esi[] = { 0x56 };

const unsigned char code__push_edi[] = { 0x57 };

const unsigned char code__call_edx[] = { 0xFF, 0xD2 };

const unsigned char code__pop_edi[] = { 0x5F };

const unsigned char code__pop_esi[] = { 0x5E };

//const unsigned char code__pop_ebx[] = { 0x5B };

//const unsigned char code__pop_ecx[] = { 0x59 };

const unsigned char code__mov_ecx_edi[] = { 0x8B, 0xCF };

const unsigned char code__add_ecx_512[] = { 0x81, 0xC1, 0x00, 0x02, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_edx_address, 5);

*(unsigned int*)&(currBytePtr[-4]) = (unsigned int)putProcOffset;

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_edx_esi, 2);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_ecx, 1);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_ebx, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_esi, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__push_edi, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__call_edx, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__pop_edi, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__pop_esi, 1);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__pop_ebx, 1);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__pop_ecx, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_ecx_edi, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_ecx_512, 6);

#ifdef DEBUG_MODE_BY_ASSEMBLY

printf("    mov eax, dword ptr[ecx]\\r\\n");

printf("    mov edx, 0%08Xh\\r\\n", (unsigned int)putProcOffset);

printf("    add edx, esi\\r\\n");

printf("    ;push ecx\\r\\n");

printf("    ;push ebx\\r\\n");

printf("    push esi\\r\\n");

printf("    push edi\\r\\n");

```

```

        printf("  call edx\r\n");

        printf("  pop edi\r\n");

        printf("  pop esi\r\n");

        printf("  ;pop ebx\r\n");

        printf("  ;pop ecx\r\n");

        printf("  mov ecx, edi ; reset second stack\r\n");

        printf("  add ecx, 512 ; reset second stack\r\n");

#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    return currBytePtr;

}

```

Preparer.cpp

```

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*          file: preparer.hxx          *

*          (draft!) *

*****/

#include "../././src/include/preparer/preparer.h"

#include "../././src/include/def.h"

#include "../././src/include/config.h"

#include "../././src/include/lexica/lexica.h"

#include "../././src/include/syntax/syntax.h"

#include "../././src/include/semantix/semantix.h"

#include "../././src/include/generator/generator.h"

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

int precedenceLevel(char* lexemStr) {

    if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_NOT][0], MAX_LEXEM_SIZE)) {

        return 6;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_AND][0], MAX_LEXEM_SIZE)) {

        return 5;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_MUL][0], MAX_LEXEM_SIZE)) {

```

```

    return 5;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_DIV][0], MAX_LEXEM_SIZE)) {
    return 5;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_MOD][0], MAX_LEXEM_SIZE)) {
    return 5;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_OR][0], MAX_LEXEM_SIZE)) {
    return 4;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_ADD][0], MAX_LEXEM_SIZE)) {
    return 4;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_SUB][0], MAX_LEXEM_SIZE)) {
    return 4;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_EQUAL][0], MAX_LEXEM_SIZE)) {
    return 3;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_NOT_EQUAL][0], MAX_LEXEM_SIZE)) {
    return 3;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_LESS_OR_EQUAL][0], MAX_LEXEM_SIZE)) {
    return 3;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_GREATER_OR_EQUAL][0], MAX_LEXEM_SIZE)) {
    return 3;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_RLBIND][0], MAX_LEXEM_SIZE)) {
    return 2;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_INPUT][0], MAX_LEXEM_SIZE)) {
    return 1;
}

else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_OUTPUT][0], MAX_LEXEM_SIZE)) {
    return 1;
}

```

```

return 0;

}

bool isLeftAssociative(char* lexemStr) {

    if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_AND][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_MUL][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_DIV][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_MOD][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_OR][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_ADD][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_SUB][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_EQUAL][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_NOT_EQUAL][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_LESS_OR_EQUAL][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_GREATER_OR_EQUAL][0], MAX_LEXEM_SIZE)) {

        return true;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_RLBIND][0], MAX_LEXEM_SIZE)) {

        return false;

    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_NOT][0], MAX_LEXEM_SIZE)) {

```



```

        return false;
    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_INPUT][0], MAX_LEXEM_SIZE)) {
        return false;
    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_OUTPUT][0], MAX_LEXEM_SIZE)) {
        return false;
    }

    return false;
}

bool isSplittingOperator(char* lexemStr) {
    if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_INPUT][0], MAX_LEXEM_SIZE)) {
        return true;
    }

    else if (!strcmp(lexemStr, tokenStruct[MULTI_TOKEN_OUTPUT][0], MAX_LEXEM_SIZE)) {
        return true;
    }

    return false;
}

void makePrepare4IdentifierOrValue(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo** lastTempLexemInfoInTable) { //
    if ((*lastLexemInfoInTable)->tokenType == IDENTIFIER_LEXEME_TYPE || (*lastLexemInfoInTable)->tokenType ==
    VALUE_LEXEME_TYPE) {
        int prevNonOpenParenthesesIndex = -1;

        for (; !strcmp((*lastLexemInfoInTable)[prevNonOpenParenthesesIndex].lexemStr, "(", MAX_LEXEM_SIZE); --
        prevNonOpenParenthesesIndex);

        if (!strcmp((*lastLexemInfoInTable)[1].lexemStr, tokenStruct[MULTI_TOKEN_RLBIND][0], MAX_LEXEM_SIZE)
            ||
            //!strcmp((*lastLexemInfoInTable)[-1].lexemStr, tokenStruct[MULTI_TOKEN_INPUT][0], MAX_LEXEM_SIZE)
            //||
            //!strcmp((*lastLexemInfoInTable)[-2].lexemStr, tokenStruct[MULTI_TOKEN_INPUT][0], MAX_LEXEM_SIZE)
            //||
            !strcmp((*lastLexemInfoInTable)[prevNonOpenParenthesesIndex].lexemStr, tokenStruct[MULTI_TOKEN_INPUT][0],
    MAX_LEXEM_SIZE)
        ) {
            bool findComplete = false;

            for (unsigned long long int index = 0; identifierIdsTable[index][0] != '\0'; ++index) {
                if (!strcmp((*lastLexemInfoInTable)->lexemStr, identifierIdsTable[index], MAX_LEXEM_SIZE)) {
                    findComplete = true;

                    (*lastTempLexemInfoInTable)->ifvalue = /*dataOffset + */VALUE_SIZE * /*(unsigned long long int)*/index;
                    _itoa((*lastTempLexemInfoInTable)->ifvalue, (*lastTempLexemInfoInTable)->lexemStr, 10);
                }
            }
        }
    }
}

```

```

        ((*lastTempLexemInfoInTable)++)->tokenType = VALUE_LEXEME_TYPE; // ADDRESS_LEXEME_TYPE
        ++*lastLexemInfoInTable;

    }

}

if (!findComplete) {

    printf("\r\nError!\r\n");

    exit(0);

}

}

else {

    ((*lastTempLexemInfoInTable)++ = (*lastLexemInfoInTable)++);

}

}

}

```

```

void makePrepare4Operators(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo** lastTempLexemInfoInTable) {

    if (precedenceLevel((*lastLexemInfoInTable)->lexemStr)) {

        while (lexemInfoTransformationTempStackSize > 0) {

            struct LexemInfo/*&*/ currLexemInfo = lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1];

            if (precedenceLevel(currLexemInfo.lexemStr) && (

                (isLeftAssociative((*lastLexemInfoInTable)->lexemStr) && (precedenceLevel((*lastLexemInfoInTable)->lexemStr) <=
precedenceLevel(currLexemInfo.lexemStr)))

                ||

                (!isLeftAssociative((*lastLexemInfoInTable)->lexemStr) && (precedenceLevel((*lastLexemInfoInTable)->lexemStr) <
precedenceLevel(currLexemInfo.lexemStr)))

            )) {

                **lastTempLexemInfoInTable = currLexemInfo; ++*lastTempLexemInfoInTable;

                --lexemInfoTransformationTempStackSize;

            }

            else {

                break;

            }

        }

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = ((*lastLexemInfoInTable)++);

    }

}

```

```

void makePrepare4LeftParenthesis(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo** lastTempLexemInfoInTable) {

    if ((*lastLexemInfoInTable)->lexemStr[0] == '(') {

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = ((*lastLexemInfoInTable)++);

    }

}

```

```
}
```

```
void makePrepare4RightParenthesis(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo** lastTempLexemInfoInTable) {  
    if ((*lastLexemInfoInTable)->lexemStr[0] == ')') {  
        bool findLeftParenthesis = false;  
        while (lexemInfoTransformationTempStackSize > 0) {  
            struct LexemInfo/*&*/ currLexemInfo = lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1];  
            if (currLexemInfo.lexemStr[0] == '(') {  
                findLeftParenthesis = true;  
                break;  
            }  
            else {  
                **lastTempLexemInfoInTable = currLexemInfo; ++* lastTempLexemInfoInTable;  
                lexemInfoTransformationTempStackSize--;  
            }  
        }  
        if (!findLeftParenthesis) {  
            printf("Warning: parentheses mismatched\n");  
  
            **lastTempLexemInfoInTable = **lastLexemInfoInTable; ++* lastTempLexemInfoInTable;  
        }  
        else {  
            --lexemInfoTransformationTempStackSize;  
        }  
  
        ++* lastLexemInfoInTable;  
    }  
}
```

```
unsigned int makePrepareEnde(struct LexemInfo** lastLexemInfoInTable, struct LexemInfo** lastTempLexemInfoInTable) {  
    unsigned int addedLexemCount = (unsigned int)lexemInfoTransformationTempStackSize;  
    while (lexemInfoTransformationTempStackSize > 0) {  
        struct LexemInfo/*&*/ currLexemInfo = lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1];  
        if (currLexemInfo.lexemStr[0] == '(' || currLexemInfo.lexemStr[0] == ')') {  
            printf("Error: parentheses mismatched\n");  
            exit(0);  
        }  
  
        **lastTempLexemInfoInTable = currLexemInfo, ++(*lastTempLexemInfoInTable); // (*lastTempLexemInfoInTable)++ = currLexemInfo;  
        --lexemInfoTransformationTempStackSize;  
    }  
  
    (*lastTempLexemInfoInTable)->lexemStr[0] = '\0';  
}
```

```

        return addedLexemCount;
    }

long long int getPrevNonParenthesesIndex(struct LexemInfo* lexemInfoInTable, unsigned long long currIndex) {
    if (!currIndex) {
        return currIndex;
    }

    long long int index = currIndex - 1;
    for (; index != ~0 && (
        lexemInfoInTable[index].lexemStr[0] == '('
        || lexemInfoInTable[index].lexemStr[0] == ')'
    );
        --index);

    return index;
}

long long int getEndOfNewPrevExpressioIndex(struct LexemInfo* lexemInfoInTable, unsigned long long currIndex) {
    if (!currIndex) { // || lexemInfoInTable[currIndex - 1].lexemStr[0] != '('
        return currIndex;
    }

    long long int index = currIndex - 1;
    for (; index != ~0 && lexemInfoInTable[index].lexemStr[0] == '(';
        --index);

    return index;
}

unsigned long long int getNextEndOfExpressionIndex(struct LexemInfo* lexemInfoInTable, unsigned long long prevEndOfExpressionIndex) {
    bool isPreviousExpressionComplete = false;

    for (unsigned long long int index = prevEndOfExpressionIndex + 2; lexemInfoInTable[index].lexemStr[0] != '\0'; ++index) {

        if (!strncmp(lexemInfoInTable[index].lexemStr, "(", MAX_LEXEM_SIZE) || !strncmp(lexemInfoInTable[index].lexemStr, ")",
MAX_LEXEM_SIZE)) {
            continue;
        }

        long long int prevNonParenthesesIndex = getPrevNonParenthesesIndex(lexemInfoInTable, index);
    }
}

```

```

        if (lexemInfoInTable[index].tokenType == IDENTIFIER_LEXEME_TYPE || lexemInfoInTable[index].tokenType ==
VALUE_LEXEME_TYPE) {

            if (lexemInfoInTable[prevNonParenthesesIndex].tokenType == IDENTIFIER_LEXEME_TYPE ||
lexemInfoInTable[prevNonParenthesesIndex].tokenType == VALUE_LEXEME_TYPE) {

                return getEndOfNewPrevExpressioIndex(lexemInfoInTable, index);

            }

        }

        else if (precedenceLevel(lexemInfoInTable[index].lexemStr) && isLeftAssociative(lexemInfoInTable[index].lexemStr)) {

            if (precedenceLevel(lexemInfoInTable[prevNonParenthesesIndex].lexemStr)) {

                return getEndOfNewPrevExpressioIndex(lexemInfoInTable, index);

            }

        }

        else if (isSplittingOperator(lexemInfoInTable[index].lexemStr)) {

            if (lexemInfoInTable[prevNonParenthesesIndex].tokenType == IDENTIFIER_LEXEME_TYPE ||
lexemInfoInTable[prevNonParenthesesIndex].tokenType == VALUE_LEXEME_TYPE) {

                return getEndOfNewPrevExpressioIndex(lexemInfoInTable, index);

            }

        }

        else if (lexemInfoInTable[index].tokenType != IDENTIFIER_LEXEME_TYPE && lexemInfoInTable[index].tokenType !=
VALUE_LEXEME_TYPE && !precedenceLevel(lexemInfoInTable[index].lexemStr)) {

            if (lexemInfoInTable[prevNonParenthesesIndex].tokenType == IDENTIFIER_LEXEME_TYPE ||
lexemInfoInTable[prevNonParenthesesIndex].tokenType == VALUE_LEXEME_TYPE || precedenceLevel(lexemInfoInTable[prevNonParenthesesIndex].lexemStr)) {

                return getEndOfNewPrevExpressioIndex(lexemInfoInTable, index);

            }

        }

    }

    return ~0;

}

void makePrepare(struct LexemInfo* lexemInfoInTable, struct LexemInfo** lastLexemInfoInTable, struct LexemInfo** lastTempLexemInfoInTable) {

    unsigned long long int nullStatementIndex = 0;

    unsigned long long int passMakePrepareElementCount = getDataSectionLastLexemIndex(*lastLexemInfoInTable, &grammar);

    if (passMakePrepareElementCount++ == ~0) {

        printf("Error: bad section!\r\n");

        exit(0);

    }

    //

    *lastLexemInfoInTable += lastDataSectionLexemIndex;

    //

    while (lastDataSectionLexemIndex--> 0) {

    //

    //

    }

    //

    for (; false && (*lastLexemInfoInTable)->lexemStr[0] != '\0'; (*lastTempLexemInfoInTable)++ = (*lastLexemInfoInTable)++) {

    //

    if (passMakePrepareElementCount) {

```

```

//          --passMakePrepareElementCount;
//          ++lexemInfoInTable;
//          continue;
//      }
//      else {
//          break;
//      }
//  }

lexemInfoTransformationTempStackSize = 0;
for (; (*lastLexemInfoInTable)->lexemStr[0] != '\0'; *(*lastTempLexemInfoInTable)++ = *(*lastLexemInfoInTable)++) {
    if (passMakePrepareElementCount) {
        --passMakePrepareElementCount;
        ++lexemInfoInTable;
        continue;
    }

    for (struct LexemInfo* lastLexemInfoInTable_ = NULL; lastLexemInfoInTable_ != *lastLexemInfoInTable;) {

        lastLexemInfoInTable_ = *lastLexemInfoInTable;
        makePrepare4IdentifierOrValue(lastLexemInfoInTable, lastTempLexemInfoInTable);
        if (lastLexemInfoInTable_ == *lastLexemInfoInTable)
            makePrepare4Operators(lastLexemInfoInTable, lastTempLexemInfoInTable);

        if (lastLexemInfoInTable_ == *lastLexemInfoInTable)
            makePrepare4LeftParenthesis(lastLexemInfoInTable, lastTempLexemInfoInTable);

        if (lastLexemInfoInTable_ == *lastLexemInfoInTable)
            makePrepare4RightParenthesis(lastLexemInfoInTable, lastTempLexemInfoInTable);

        if (lastLexemInfoInTable_ != *lastLexemInfoInTable
            && (!nullStatementIndex || (lexemInfoInTable + nullStatementIndex == lastLexemInfoInTable_))) {
            if (nullStatementIndex != ~0) {
                if (nullStatementIndex) {
//                    printf("Added null statement after %lld(lexem index)\r\n", nullStatementIndex);
                    makePrepareEnder(lastLexemInfoInTable, lastTempLexemInfoInTable);
                    (void)createMultiToken(lastTempLexemInfoInTable, MULTI_TOKEN_NULL_STATEMENT);
                }

                nullStatementIndex = getNextEndOfExpressionIndex(lexemInfoInTable, nullStatementIndex);
            }
        }
    }
}

```

```

    }

    makePrepareEnder(lastLexemInfoInTable, lastTempLexemInfoInTable);

    if ((!nullStatementIndex || (lexemInfoInTable + nullStatementIndex == *lastLexemInfoInTable))) {
        if (nullStatementIndex != ~0) {
            if (nullStatementIndex) {
                //                printf("Added null statement after %lld(lexem index)\r\n", nullStatementIndex);
                makePrepareEnder(lastLexemInfoInTable, lastTempLexemInfoInTable);
                (void)createMultiToken(lastTempLexemInfoInTable, MULTI_TOKEN_NULL_STATEMENT);
            }

            nullStatementIndex = getNextEndOfExpressionIndex(lexemInfoInTable, nullStatementIndex);
        }
    }

    }

}

Repeat_until.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****
* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *
*
*           file: repeat_until.cpp           *
*
*           (draft!) *
*****/

#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"
#include "stdio.h"
#include "string.h"

unsigned char* makeRepeatCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_REPEAT);

    if (multitokenSize) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("\r\n");
        printf("    ;\|%s|\r\n", tokenStruct[MULTI_TOKEN_REPEAT][0]);
#endif
    }

    lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = *lastLexemInfoInTable;
    lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue = (unsigned long long int)currBytePtr;

```

```

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf(" LABEL@REPEAT_%016lX:\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

unsigned char* makeUntileCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) { // Or Ender!

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_UNTIL);

    if (multitokenSize

        && lexemInfoTransformationTempStackSize

        && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_REPEAT][0], MAX_LEXEM_SIZE)

    ) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;%s\r\n", tokenStruct[MULTI_TOKEN_UNTIL][0]);

#endif

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = **lastLexemInfoInTable;

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    return currBytePtr;

}

unsigned char* makeNullStatementAfterUntilCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_NULL_STATEMENT);

    if (multitokenSize) {

        if (lexemInfoTransformationTempStackSize < 2

            || strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_UNTIL][0], MAX_LEXEM_SIZE)

            || strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_REPEAT][0], MAX_LEXEM_SIZE)

        ) {

            return currBytePtr;

        }

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");


```



```

        printf("    :after cond expresion (after \"%s\" after \"%s\")\r\n", tokenStruct[MULTI_TOKEN_UNTIL][0],
tokenStruct[MULTI_TOKEN_REPEAT][0]);

#endif

const unsigned char code__cmp_eax_0[] = { 0x83, 0xF8, 0x00 };

const unsigned char code__jnz_offset[] = { 0x0F, 0x85, 0x00, 0x00, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_eax_0, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jnz_offset, 6);

*(unsigned int*)(currBytePtr - 4) = (unsigned int)((unsigned
char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue - currBytePtr);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("    cmp eax, 0\r\n");

    printf("    jnz LABEL@REPEAT_%016llx\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

#endif

    lexemInfoTransformationTempStackSize -= 2;

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

```

Rbind.cpp

```
#define _CRT_SECURE_NO_WARNINGS
```

```

/*****

```

```

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

```

```

*           file: rbind.cpp           *

```

```

*           (draft!) *

```

```

*****/

```

```
#include "../include/def.h"
```

```
#include "../include/generator/generator.h"
```

```
#include "../include/lexica/lexica.h"
```

```
#include "stdio.h"
```

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

```
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_RLBIND);
```

```
    if (multitokenSize) {
```

```

#ifdef DEBUG_MODE_BY_ASSEMBLY

```

```
    printf("\r\n");
```

```
    printf("    ;\"%s\"\r\n", tokenStruct[MULTI_TOKEN_RLBIND][0]);
```

```

#endif

```

```

const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

const unsigned char code__mov_ebx_stackTopByECXMinus4[] = { 0x8B, 0x59, 0xFC };

const unsigned char code__sub_ecx_8[] = { 0x83, 0xE9, 0x08 };

const unsigned char code__add_ebx_edi[] = { 0x03, 0xDF };

const unsigned char code__mov_addrFromEBX_eax[] = { 0x89, 0x03 };

const unsigned char code__mov_ecx_edi[] = { 0x8B, 0xCF };

const unsigned char code__add_ecx_512[] = { 0x81, 0xC1, 0x00, 0x02, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_ebx_stackTopByECXMinus4, 3);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_8, 3);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_ebx_edi, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_addrFromEBX_eax, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_ecx_edi, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__add_ecx_512, 6);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("  mov eax, dword ptr[ecx]\r\n");
    printf("  mov ebx, dword ptr[ecx - 4]\r\n");
    printf("  sub ecx, 8\r\n");
    printf("  add ebx, edi\r\n");
    printf("  mov dword ptr [ebx], eax\r\n");
    printf("  mov ecx, edi ; reset second stack\r\n");
    printf("  add ecx, 512 ; reset second stack\r\n");

#endif

    return *lastLexemInfoInTable += multitokenSize, currBytePtr;
}

return currBytePtr;
}

Semantix.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: semantix.cpp           *

*           (draft!) *

*****/

//#include "../include/config.h"

#include "../include/syntax/syntax.h"

#include "../include/semantix/semantix.h"

```

```

#include "stdio.h"

#include "string.h"


//#include <iterator>

#include <regex>


//

//#define COLLISION_II_STATE 128

//#define COLLISION_LL_STATE 129

//#define COLLISION_IL_STATE 130

//#define COLLISION_I_STATE 132

//#define COLLISION_L_STATE 133

//

//#define NO_IMPLEMENT_CODE_STATE 256


unsigned long long int getDataSectionLastLexemIndex(LexemInfo* lexemInfoTable, Grammar* grammar) {

    int lexemIndex = 0;

    const struct LexemInfo* unexpectedLexemfailedTerminal = nullptr;

    if (recursiveDescentParserRuleWithDebug("program____part1", lexemIndex, lexemInfoTable, grammar, 0, &unexpectedLexemfailedTerminal)

        && lexemInfoTable[lexemIndex].lexemStr[0] != '\0') {

        return lexemIndex;

    }

    printf("Error: No find data section end index!\r\n");

    return ~0;

}


int checkingInternalCollisionInDeclarations(LexemInfo* lexemInfoTable, Grammar* grammar, char(*identifierIdsTable)[MAX_LEXEM_SIZE], char **

errorMessageesPtrToLastBytePtr) {

//    int returnState = SUCCESS_STATE;

    unsigned long long int lastDataSectionLexemIndex = 0;

    if (~0 == (lastDataSectionLexemIndex = getDataSectionLastLexemIndex(lexemInfoTable, grammar))) { // TODO: ADD TO START CODE

        *errorMessageesPtrToLastBytePtr += sprintf(*errorMessageesPtrToLastBytePtr, "Error get of data section last lexem index.\r\n");

        return ~SUCCESS_STATE;

    }

    for (unsigned int index = 0; identifierIdsTable[index][0] != '\0'; ++index) {

        char isDeclaredIdentifier = 0;

        char isDeclaredIdentifierCollision = 0;

        unsigned int lexemIndex = 0;

        for (lexemIndex = 0; lexemIndex <= lastDataSectionLexemIndex; ++lexemIndex) {

            if (lexemesInfoTable[lexemIndex].tokenType == IDENTIFIER_LEXEME_TYPE) {

```

```

        if (!strcmp(identifierIdsTable[index], lexemesInfoTable[lexemIndex].lexemStr, MAX_LEXEM_SIZE)) {
            if (isDeclaredIdentifier) {
                isDeclaredIdentifierCollision = 1;
            }
            isDeclaredIdentifier = 1;
        }
    }
}

char isLabel = 0;
char isDeclaredLabel = 0;
char isDeclaredLabelCollision = 0;
for (unsigned int lexemIndex = 0; lexemesInfoTable[lexemIndex].lexemStr[0] != '\0'; ++lexemIndex) {
    if (lexemesInfoTable[lexemIndex].tokenType != IDENTIFIER_LEXEME_TYPE || strcmp(identifierIdsTable[index],
lexemesInfoTable[lexemIndex].lexemStr, MAX_LEXEM_SIZE)) {
        continue;
    }
    if (!strcmp(lexemesInfoTable[lexemIndex + 1].lexemStr, tokenStruct[MULTI_TOKEN_COLON][0], MAX_LEXEM_SIZE)) {
        if (isDeclaredLabel) {
            isDeclaredLabelCollision = 1;
        }
        isLabel = 1;
        isDeclaredLabel = 1;
    }
    if (lexemIndex && !strcmp(lexemesInfoTable[lexemIndex - 1].lexemStr, tokenStruct[MULTI_TOKEN_GOTO][0],
MAX_LEXEM_SIZE)) {
        isLabel = 1;
    }
}

//          //tryToGetKeyWord(struct LexemInfo* lexemInfoInTable);
//          if (SUCCESS_STATE != checkingCollisionInDeclarationsByKeyWords(identifierIdsTable[index])) {
//              return COLLISION_IK_STATE;
//          }

    if (isDeclaredIdentifierCollision) {
        printf("Collision(identifier/identifier): %s\r\n", identifierIdsTable[index]);
        *errorMessagesPtrToLastBytePtr += snprintf(*errorMessagesPtrToLastBytePtr, MAX_LEXEM_SIZE +
strlen("Collision(identifier/identifier): #\r\n"), "Collision(identifier/identifier): %s\r\n", identifierIdsTable[index]);
        return COLLISION_II_STATE;
    }
    if (isDeclaredLabelCollision) {
        printf("Collision(label/label): %s\r\n", identifierIdsTable[index]);
    }
}

```

```

        *errorMessagesPtrToLastBytePtr += snprintf(*errorMessagesPtrToLastBytePtr, MAX_LEXEM_SIZE + strlen("Collision(label/label):
#\r\n"), "Collision(label/label): %s\r\n", identifierIdsTable[index]);

        return COLLISION_LL_STATE;

    }

    if (isDeclaredIdentifier && isLabel) {

        printf("Collision(identifier/label): %s\r\n", identifierIdsTable[index]);

        *errorMessagesPtrToLastBytePtr += snprintf(*errorMessagesPtrToLastBytePtr, MAX_LEXEM_SIZE +
strlen("Collision(identifier/label): #\r\n"), "Collision(identifier/label): %s\r\n", identifierIdsTable[index]);

        return COLLISION_IL_STATE;

    }

    else if (!isDeclaredIdentifier && !isLabel && !isDeclaredLabel) {

        printf("Undeclared identifier: %s\r\n", identifierIdsTable[index]);

        *errorMessagesPtrToLastBytePtr += snprintf(*errorMessagesPtrToLastBytePtr, MAX_LEXEM_SIZE + strlen("Undeclared
identifier: #\r\n"), "Undeclared identifier: %s\r\n", identifierIdsTable[index]);

        return COLLISION_I_STATE;

    }

    else if (isLabel && !isDeclaredLabel) {

        printf("Undeclared label: %s\r\n", identifierIdsTable[index]);

        *errorMessagesPtrToLastBytePtr += snprintf(*errorMessagesPtrToLastBytePtr, MAX_LEXEM_SIZE + strlen("Undeclared label:
#\r\n"), "Undeclared label: %s\r\n", identifierIdsTable[index]);

        return COLLISION_L_STATE;

    }

}

//      if (returnState == SUCCESS_STATE) {

//          printf("Declaration verification was successful!\r\n");

//      }

//

return SUCCESS_STATE;

}

int checkingVariableInitialization(LexemInfo* lexemInfoTable, Grammar* grammar, char(*identifierIdsTable)[MAX_LEXEM_SIZE], char**
errorMessagesPtrToLastBytePtr) {

    int returnState = SUCCESS_STATE;

    unsigned long long int lastDataSectionLexemIndex = 0;

    if (~0 == (lastDataSectionLexemIndex = getDataSectionLastLexemIndex(lexemInfoTable, grammar))) { // TODO: ADD TO START CODE

        *errorMessagesPtrToLastBytePtr += sprintf(*errorMessagesPtrToLastBytePtr, "Error get of data section last lexem index.\r\n");

        return ~SUCCESS_STATE;

    }

    for (unsigned int index = 0; identifierIdsTable[index][0] != '\0'; ++index) {

        for (unsigned int lexemIndex = lastDataSectionLexemIndex; lexemesInfoTable[lexemIndex].lexemStr[0] != '\0'; ++lexemIndex) {

            if (lexemesInfoTable[lexemIndex].tokenType != IDENTIFIER_LEXEME_TYPE || strcmp(identifierIdsTable[index],
lexemesInfoTable[lexemIndex].lexemStr, MAX_LEXEM_SIZE)) {

```

```

        continue;

    }

    if (!strcmp.lexemesInfoTable[lexemIndex + 1].lexemStr, tokenStruct[MULTI_TOKEN_COLON][0], MAX_LEXEM_SIZE)) {

        continue;

    }

    if (lexemIndex && !strcmp.lexemesInfoTable[lexemIndex - 1].lexemStr, tokenStruct[MULTI_TOKEN_GOTO][0],
MAX_LEXEM_SIZE)) {

        continue;

    }

    int prevNonOpenParenthesesIndex = -1;

    for (; !strcmp.lexemesInfoTable[lexemIndex + prevNonOpenParenthesesIndex].lexemStr, "(", MAX_LEXEM_SIZE); --
prevNonOpenParenthesesIndex);

    if (!strcmp.lexemesInfoTable[lexemIndex + 1].lexemStr, tokenStruct[MULTI_TOKEN_RLBIND][0], MAX_LEXEM_SIZE)

        ||

        //!strcmp.lexemesInfoTable[-1].lexemStr, tokenStruct[MULTI_TOKEN_INPUT][0], MAX_LEXEM_SIZE)

        //||

        //!strcmp.lexemesInfoTable[-2].lexemStr, tokenStruct[MULTI_TOKEN_INPUT][0], MAX_LEXEM_SIZE)

        //||

        !strcmp.lexemesInfoTable[lexemIndex + prevNonOpenParenthesesIndex].lexemStr,
tokenStruct[MULTI_TOKEN_INPUT][0], MAX_LEXEM_SIZE)

        ){

            break;

        }

    printf("Uninitialized: %s\r\n", identifierIdsTable[index]);

    *errorMessagesPtrToLastBytePtr += snprintf(*errorMessagesPtrToLastBytePtr, MAX_LEXEM_SIZE + strlen("Uninitialized: #\r\n"),
"Uninitialized: %s\r\n", identifierIdsTable[index]);

    returnState = UNINITIALIZED_I_STATE;

    break;

}

}

if (returnState == SUCCESS_STATE) {

    printf("Variable initialization checking was successful!\r\n");

}

return returnState;

}

int checkingCollisionInDeclarationsByKeyWords(char(*identifierIdsTable)[MAX_LEXEM_SIZE], char** errorMessagesPtrToLastBytePtr) {

    int returnState = SUCCESS_STATE;

```

```

        char keywords_re[] = KEYWORDS_RE;

        char keywords_[sizeof(keywords_re)] = { '\0' };

        prepareKeyWordIdGetter(keywords_, keywords_re);

        for (unsigned int index = 0; identifierIdsTable[index][0] != '\0'; ++index) {

            if (std::regex_match(std::string(identifierIdsTable[index]), std::regex(keywords_re))) {

                printf("Declaration matches keyword: %s\r\n", identifierIdsTable[index]);

                *errorMessagesPtrToLastBytePtr += snprintf(*errorMessagesPtrToLastBytePtr, MAX_LEXEM_SIZE + strlen("Declaration matches
keyword: #\r\n"), "Declaration matches keyword: %s\r\n", identifierIdsTable[index]);

                returnState = COLLISION_IK_STATE;

            }

        }

        printf("Declaration verification for keyword collision was successful!\r\n");

        return SUCCESS_STATE;

    }

int semantixAnalyze(LexemInfo* lexemInfoTable, Grammar* grammar, char(*identifierIdsTable)[MAX_LEXEM_SIZE], char* errorMessagesPtrToLastBytePtr){

    int returnState = SUCCESS_STATE;

    if ( SUCCESS_STATE != (returnState = checkingInternalCollisionInDeclarations(lexemesInfoTable, grammar, identifierIdsTable,
&errorMessagesPtrToLastBytePtr))

        || SUCCESS_STATE != (returnState = checkingVariableInitialization(lexemesInfoTable, grammar, identifierIdsTable,
&errorMessagesPtrToLastBytePtr))

        || SUCCESS_STATE != (returnState = checkingCollisionInDeclarationsByKeyWords(identifierIdsTable, &errorMessagesPtrToLastBytePtr))

    ) {

        return returnState;

    }

    return SUCCESS_STATE;

}

Semikolon.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: semicolon.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

```

```

unsigned char* makeSemicolonAfterNonContextCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_SEMICOLON);

    if (multitokenSize

        &&

        !lexemInfoTransformationTempStackSize // !

    ) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\n%s\r\n", "");

#endif

        *lastLexemInfoInTable += multitokenSize;

    }

    return currBytePtr;

}

```

```

unsigned char* makeSemicolonIgnoreContextCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_SEMICOLON);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\n%s\r\n", "");

#endif

        *lastLexemInfoInTable += multitokenSize;

    }

    return currBytePtr;

}

```

Sub.cpp

```

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: sub.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

```



```

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

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_SUB);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\"%s\"\\r\n", tokenStruct[MULTI_TOKEN_SUB][0]);

#endif

        const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

        const unsigned char code__sub_ecx_4[] = { 0x83, 0xE9, 0x04 };

        const unsigned char code__sub_stackTopByECX_eax[] = { 0x29, 0x01 };

        //const unsigned char code__mov_eax_stackTopByECX[] = { 0x8B, 0x01 };

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_ecx_4, 3);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__sub_stackTopByECX_eax, 2);

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__mov_eax_stackTopByECX, 2);

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("    mov eax, dword ptr[ecx]\\r\n");

        printf("    sub ecx, 4\\r\n");

        printf("    sub dword ptr[ecx], eax\\r\n");

        printf("    mov eax, dword ptr[ecx]\\r\n");

#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    return currBytePtr;

}

```

Syntax.cpp

```

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: syntax.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/config.h"

#include "../include/syntax/syntax.h"

#include <iostream>

```

```

#include <fstream>

#include <iomanip>

#include <vector>

#include <map>

// #include <unordered_map>

#include <string>

#include <set>


using namespace std;


Grammar grammar = {

    CONFIGURABLE_GRAMMAR

    #if 0

    {

        {"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"}},
{"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, {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"}},
{"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, {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}},

{"tokenLRBIND", 1, {T_LRBIND_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"} },

//

//      { "label", 1, {"ident_terminal"} },

//

{ "", 2, {"", ""} }

},

176,

```

```

"program"

#endif

};

Grammar originalGrammar = {

    ORIGINAL_GRAMMAR

#ifdef 0

{

    {"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, {"!r_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"} },

//

//      {"label", 1, {"ident_terminal"} },

//

    {"", 2, {"",""}}

},

176,

"program"

#endif

};

```

```

#define DEBUG_STATES

```

```

#define MAX_LEXEMS 256

// #define MAX_RULES 128

#define MAX_SYMBOLS 64

typedef struct {
    char symbols[MAX_SYMBOLS][MAX_TOKEN_SIZE];
    int count;
} SymbolSet;

typedef SymbolSet ParseInfoTable[MAX_LEXEMS][MAX_LEXEMS];

bool insertIntoSymbolSet(SymbolSet* set, const char* symbol) {
    for (int i = 0; i < set->count; ++i) {
        if (strcmp(set->symbols[i], symbol) == 0) {
            // symbol already exists
            return false;
        }
    }
    strncpy(set->symbols[set->count], symbol, MAX_TOKEN_SIZE);
    set->symbols[set->count][MAX_TOKEN_SIZE - 1] = '\0';
    ++set->count;
    return true;
}

bool containsSymbolSet(const SymbolSet* set, const char* symbol) {
    for (int i = 0; i < set->count; ++i) {
        if (strcmp(set->symbols[i], symbol) == 0) {
            return true;
        }
    }
    return false;
}

// initialize with empty SymbolSets
ParseInfoTable parseInfoTable = { { {0} } };

struct ASTNode {
    std::string value;
    bool isTerminal;
    std::vector<ASTNode*> children;

```

```

ASTNode(const std::string& val, bool isTerminal) : isTerminal(isTerminal), value(val) {}

~ASTNode() {
    for (ASTNode* child : children) {
        delete child;
    }
}

};

ASTNode* buildASTByCPPMap(const std::map<int, std::map<int, std::set<std::string>>>& parseInfoTable,
    Grammar* grammar,
    int start,
    int end,
    const std::string& symbol) {
    if (start > end) return nullptr;

    ASTNode* node = new ASTNode(symbol, false);

    for (const Rule& rule : grammar->rules) {
        if (rule.lhs != symbol) continue;

        if (rule.rhs_count == 1) {
            //if (parseInfoTable.at(start).at(end).count(rule.rhs[0])) {
            node->children.push_back(new ASTNode(rule.rhs[0], true));
            return node;
            //}
        }
        else if (rule.rhs_count == 2) {
            for (int split = start; split < end; ++split) {
                if (parseInfoTable.at(start).at(split).count(rule.rhs[0]) &&
                    parseInfoTable.at(split + 1).at(end).count(rule.rhs[1])) {
                    node->children.push_back(buildASTByCPPMap(parseInfoTable, grammar, start, split, rule.rhs[0]));
                    node->children.push_back(buildASTByCPPMap(parseInfoTable, grammar, split + 1, end, rule.rhs[1]));
                    return node;
                }
            }
        }
    }

    return nullptr;
}

ASTNode* buildAST(const std::map<int, std::map<int, std::set<std::string>>>& parseInfoTable,
    ParseInfoTable& parseInfoTable,

```

```

Grammar* grammar,

int start,

int end,

const std::string& symbol) {

if (start > end) return nullptr;


ASTNode* node = new ASTNode(symbol, false);


for (const Rule& rule : grammar->rules) {

    if (rule.lhs != symbol) continue;


    if (rule.rhs_count == 1) {

        //if (parseInfoTable.at(start).at(end).count(rule.rhs[0])) {

        node->children.push_back(new ASTNode(rule.rhs[0], true));

        return node;

        //}

    }

    else if (rule.rhs_count == 2) {

        for (int split = start; split < end; ++split) {

            if (containsSymbolSet(&parseInfoTable[start][split], rule.rhs[0]) &&

                containsSymbolSet(&parseInfoTable[split + 1][end], rule.rhs[1])) {

                node->children.push_back(buildAST(parseInfoTable, grammar, start, split, rule.rhs[0]));

                node->children.push_back(buildAST(parseInfoTable, grammar, split + 1, end, rule.rhs[1]));

                return node;

            }

        }

    }

}

return nullptr;

}


void printAST(struct LexemInfo* lexemInfoTable, const ASTNode* node, int depth = 0) {

    static int lexemInfoTableIndexForPrintAST = 0; // ATTENTION: multithreading is not supported for this!

    if (!node) {

        return;

    }

    if (!depth) {

        lexemInfoTableIndexForPrintAST = 0;

    }

    for (unsigned int depthIndex = 0; depthIndex <= depth; ++depthIndex) {

        std::cout << "    " << "|";

```

```

}

std::cout << "--";

if (node->isTerminal) {
    std::cout << "\"" << lexemInfoTable[lexemInfoTableIndexForPrintAST++].lexemStr << "\"";
}
else {
    std::cout << node->value;
}

std::cout << "\n";

for (const ASTNode* child : node->children) {
    printAST(lexemInfoTable, child, depth + 1);
}
}

void printASTToFile(struct LexemInfo* lexemInfoTable, const ASTNode* node, std::ofstream& outFile, int depth = 0) {
    static int lexemInfoTableIndexForPrintAST = 0; // ATTENTION: multithreading is not supported for this!

    if (!node) {
        return;
    }

    if (!depth) {
        lexemInfoTableIndexForPrintAST = 0;
    }

    for (unsigned int depthIndex = 0; depthIndex <= depth; ++depthIndex) {
        outFile << "  |";
    }

    outFile << "--";

    if (node->isTerminal) {
        outFile << "\"" << lexemInfoTable[lexemInfoTableIndexForPrintAST++].lexemStr << "\"";
    }
    else {
        outFile << node->value;
    }

    outFile << "\n";

    for (const ASTNode* child : node->children) {
        printASTToFile(lexemInfoTable, child, outFile, depth + 1);
    }
}

```

```

void printAST__OLD_123(struct LexemInfo* lexemInfoTable, const ASTNode* node, int depth = 0) {

    static int lexemInfoTableIndexForPrintAST = 0; // ATTENTION: multithreading is not supported for this!

    if (!node) {

        return;

    }

    if (!depth) {

        lexemInfoTableIndexForPrintAST = 0;

    }

    for (unsigned int depthIndex = 0; depthIndex <= depth; ++depthIndex) {

        std::cout << "  " << "|";

    }

    std::cout << "--";

    if (node->isTerminal) {

        std::cout << "\"" << lexemInfoTable[lexemInfoTableIndexForPrintAST++].lexemStr << "\"";

    }

    else {

        std::cout << node->value;

    }

    std::cout << "\n";

    for (const ASTNode* child : node->children) {

        printAST(lexemInfoTable, child, depth + 1);

    }

}

```

```

void displayParseInfoTable(const map<int, map<int, set<string>>>& parseInfoTable) {

    constexpr int CELL_WIDTH = 128;

    cout << left << setw(CELL_WIDTH) << "[i\\j]";

    for (const auto& outerEntry : parseInfoTable) {

        cout << setw(CELL_WIDTH) << outerEntry.first;

    }

    cout << endl;

    for (const auto& outerEntry : parseInfoTable) {

        int i = outerEntry.first;

        cout << setw(CELL_WIDTH) << i;

        for (const auto& innerEntry : parseInfoTable) {

            int j = innerEntry.first;

            if (parseInfoTable.at(i).find(j) != parseInfoTable.at(i).end()) {

                const set<string>& rules = parseInfoTable.at(i).at(j);

                string cellContent;

```



```

        for (const string& rule : rules) {
            cellContent += rule + ", ";
        }

        if (!cellContent.empty()) {
            cellContent.pop_back();
            cellContent.pop_back();
        }

        cout << setw(CELL_WIDTH) << cellContent;
    }
    else {
        cout << setw(CELL_WIDTH) << "-";
    }
}

cout << endl;
}
}

void saveParseInfoTableToFile(const map<int, map<int, set<string>>>& parseInfoTable, const string& filename) {
    constexpr int CELL_WIDTH = 128;

    ofstream file(filename);
    if (!file.is_open()) {
        cerr << "Error: Unable to open file " << filename << endl;
        return;
    }

    file << left << setw(CELL_WIDTH) << "[i\\j]";

    for (const auto& outerEntry : parseInfoTable) {
        file << setw(CELL_WIDTH) << outerEntry.first;
    }
    file << endl;

    for (const auto& outerEntry : parseInfoTable) {
        int i = outerEntry.first;
        file << setw(CELL_WIDTH) << i;

        for (const auto& innerEntry : parseInfoTable) {
            int j = innerEntry.first;
            if (parseInfoTable.at(i).find(j) != parseInfoTable.at(i).end()) {
                const set<string>& rules = parseInfoTable.at(i).at(j);

```

```

        string cellContent;

        for (const string& rule : rules) {
            cellContent += rule + ", ";
        }
        if (!cellContent.empty()) {
            cellContent.pop_back();
            cellContent.pop_back();
        }

        file << setw(CELL_WIDTH) << cellContent;
    }
    else {
        file << setw(CELL_WIDTH) << "-";
    }
}
file << endl;
}

file.close();
}

bool cykAlgorithmImplementation(struct LexemInfo* lexemInfoTable, Grammar* grammar, char * astFileName) {
    if (lexemInfoTable == NULL || grammar == NULL) {
        return false;
    }

#ifdef _DEBUG
    printf("ATTENTION: for better performance, use Release mode!\r\n");
#endif

#ifdef DEBUG_STATES
    cout << "cykParse in progress.....[please wait]";
#else
    cout << "cykParse in progress.....[please wait]: ";
#endif

// ParseInfoTable parseInfoTable = { {{0}} }; // Initialize with empty SymbolSets

    int lexemIndex = 0;
    for (--lexemIndex; lexemInfoTable[++lexemIndex].lexemStr[0];) {
#ifdef DEBUG_STATES
        printf("\rcykParse in progress.....[please wait]: %02d %16s", lexemIndex, lexemInfoTable[lexemIndex].lexemStr);

```

```
#endif
```

```
// Iterate over the rules

for (int xIndex = 0; xIndex < grammar->rule_count; ++xIndex) {

    Rule& rule = grammar->rules[xIndex];

    // If a terminal is found
    if (rule.rhs_count == 1 && (

        lexemInfoTable[lexemIndex].tokenType == IDENTIFIER_LEXEME_TYPE && !strcmp(rule.rhs[0], "ident_terminal")

        || lexemInfoTable[lexemIndex].tokenType == VALUE_LEXEME_TYPE && !strcmp(rule.rhs[0], "value_terminal")

        || !strcmp(rule.rhs[0], lexemInfoTable[lexemIndex].lexemStr, MAX_LEXEM_SIZE)

    )) {

        insertIntoSymbolSet(&parseInfoTable[lexemIndex][lexemIndex], rule.lhs);

    }

}

for (int iIndex = lexemIndex; iIndex >= 0; --iIndex) {

    for (int kIndex = iIndex; kIndex <= lexemIndex; ++kIndex) {

        for (int xIndex = 0; xIndex < grammar->rule_count; ++xIndex) {

            Rule& rule = grammar->rules[xIndex];

            if (rule.rhs_count == 2

                && containsSymbolSet(&parseInfoTable[iIndex][kIndex], rule.rhs[0])

                && containsSymbolSet(&parseInfoTable[kIndex + 1][lexemIndex], rule.rhs[1])

            ) {

                insertIntoSymbolSet(&parseInfoTable[iIndex][lexemIndex], rule.lhs);

            }

        }

    }

}

cout << "\r" << "cykParse complete.....[   ok   ]\n";

if (!containsSymbolSet(&parseInfoTable[0][lexemIndex - 1], grammar->start_symbol)) {

    return false;

}

ASTNode* astRoot = buildAST(parseInfoTable, grammar, 0, lexemIndex - 1, grammar->start_symbol);

if (astRoot) {

    std::cout << "Abstract Syntax Tree:\n";

    printAST(lexemInfoTable, astRoot);

    if (astFileName && astFileName[0] != '\0') {

        std::ofstream astOFStream(astFileName, std::ofstream::out);

        printASTToFile(lexemInfoTable, astRoot, astOFStream);

        astOFStream.close();

    }

}
```

```

    }

    delete astRoot; // Не забуваємо звільняти пам'ять
}

else {

    std::cout << "Failed to build AST.\n";

}

//return parseInfoTable[0][lexemIndex - 1].find(grammar->start_symbol) != parseInfoTable[0][lexemIndex - 1].end(); // return !!parseInfoTable[0][lexemIndex -
1].size();

return true;

}

#define MAX_STACK_DEPTH 256

bool recursiveDescentParserRuleWithDebug(const char* ruleName, int& lexemIndex, LexemInfo* lexemInfoTable, Grammar* grammar, int depth, const struct
LexemInfo** unexpectedLexemfailedTerminal) {

    if (depth > MAX_STACK_DEPTH) {

        printf("Error: Maximum recursion depth reached.\n");

        return false;

    }

    char isError = false;

    for (int i = 0; i < grammar->rule_count; ++i) {

        Rule& rule = grammar->rules[i];

        if (strcmp(rule.lhs, ruleName) != 0) continue;

        int savedIndex = lexemIndex;

        if (rule.rhs_count == 1) {

            if (

                lexemInfoTable[lexemIndex].tokenType == IDENTIFIER_LEXEME_TYPE && !strcmp(rule.rhs[0], "ident_terminal")

                || lexemInfoTable[lexemIndex].tokenType == VALUE_LEXEME_TYPE && !strcmp(rule.rhs[0], "value_terminal")

                || !strcmp(rule.rhs[0], lexemInfoTable[lexemIndex].lexemStr, MAX_LEXEM_SIZE)

            ) {

                ++lexemIndex;

                return true;

            }

            else {

                *unexpectedLexemfailedTerminal = lexemInfoTable + lexemIndex;

                if (0)printf("<< \"%s\" >>\n", rule.rhs[0]);

            }

        }

        else if (rule.rhs_count == 2) {

            if (recursiveDescentParserRuleWithDebug(rule.rhs[0], lexemIndex, lexemInfoTable, grammar, depth + 1, unexpectedLexemfailedTerminal) &&

                recursiveDescentParserRuleWithDebug(rule.rhs[1], lexemIndex, lexemInfoTable, grammar, depth + 1, unexpectedLexemfailedTerminal)) {

```

```

        return true;
    }
}

lexemIndex = savedIndex;
}

return false;
}

const LexemInfo* recursiveDescentParserWithDebug_(const char* ruleName, int& lexemIndex, LexemInfo* lexemInfoTable, Grammar* grammar, int depth, const
struct LexemInfo* unexpectedUnknownLexemfailedTerminal) {
    if (depth > MAX_STACK_DEPTH) {
        printf("Error: Maximum recursion depth reached.\n");
        return unexpectedUnknownLexemfailedTerminal;
    }
    char isError = false;
    const LexemInfo* currUnexpectedLexemfailedTerminalPtr = nullptr, * returnUnexpectedLexemfailedTerminalPtr = nullptr;
    for (int i = 0; i < grammar->rule_count; ++i) {
        Rule& rule = grammar->rules[i];
        if (strcmp(rule.lhs, ruleName) != 0) continue;

        int savedIndex = lexemIndex;
        if (rule.rhs_count == 1) {
            if (
                lexemInfoTable[lexemIndex].tokenType == IDENTIFIER_LEXEME_TYPE && !strcmp(rule.rhs[0], "ident_terminal")
                || lexemInfoTable[lexemIndex].tokenType == VALUE_LEXEME_TYPE && !strcmp(rule.rhs[0], "value_terminal")
                || !strcmp(rule.rhs[0], lexemInfoTable[lexemIndex].lexemStr, MAX_LEXEM_SIZE)
            ) {
                ++lexemIndex;
                return nullptr;
            }
            else {
                currUnexpectedLexemfailedTerminalPtr = lexemInfoTable + lexemIndex;
            }
        }
        else if (rule.rhs_count == 2) {
            if (nullptr == (returnUnexpectedLexemfailedTerminalPtr = recursiveDescentParserWithDebug_(rule.rhs[0], lexemIndex, lexemInfoTable, grammar, depth + 1,
unexpectedUnknownLexemfailedTerminal)))
                && nullptr == (returnUnexpectedLexemfailedTerminalPtr = recursiveDescentParserWithDebug_(rule.rhs[1], lexemIndex, lexemInfoTable, grammar, depth
+ 1, unexpectedUnknownLexemfailedTerminal))) {
                return nullptr;
            }
        }
        lexemIndex = savedIndex;
    }
}

```

```

}

if (returnUnexpectedLexemfailedTerminalPtr != nullptr && returnUnexpectedLexemfailedTerminalPtr != unexpectedUnknownLexemfailedTerminal
    &&( returnUnexpectedLexemfailedTerminalPtr->tokenType == IDENTIFIER_LEXEME_TYPE
        || returnUnexpectedLexemfailedTerminalPtr->tokenType == VALUE_LEXEME_TYPE
        || returnUnexpectedLexemfailedTerminalPtr->tokenType == KEYWORD_LEXEME_TYPE
    )) {
    return returnUnexpectedLexemfailedTerminalPtr;
}

if (currUnexpectedLexemfailedTerminalPtr != nullptr) {
    return currUnexpectedLexemfailedTerminalPtr;
}

if(returnUnexpectedLexemfailedTerminalPtr != nullptr){
    return returnUnexpectedLexemfailedTerminalPtr;
}

return unexpectedUnknownLexemfailedTerminal;
}

//

int syntaxAnalyze(LexemInfo* lexemInfoTable, Grammar* grammar, char syntaxlAnalyzeMode, char* astFileName, char* errorMessagesPtrToLastBytePtr) {
    bool cykAlgorithmImplementationReturnValue = false;
    if (syntaxlAnalyzeMode == SYNTAX_ANALYZE_BY_CYK_ALGORITHM) {
        cykAlgorithmImplementationReturnValue = cykAlgorithmImplementation(lexemesInfoTable, grammar, astFileName);
        printf("cykAlgorithmImplementation return \"%s\\\".\r\n", cykAlgorithmImplementationReturnValue ? "true" : "false");
        if (cykAlgorithmImplementationReturnValue) {
            return SUCCESS_STATE;
        }
        else {
            writeBytesToFile(astFileName, (unsigned char*)"Error of AST build", strlen("Error of AST build"));
        }
    }
    else if (astFileName && astFileName[0] != '\0') {
        writeBytesToFile(astFileName, (unsigned char*)"AST build no support.", strlen("AST build no support."));
    }

    if (cykAlgorithmImplementationReturnValue == false || syntaxlAnalyzeMode == SYNTAX_ANALYZE_BY_RECURSIVE_DESCENT) {
        int lexemIndex = 0;
        const struct LexemInfo* unexpectedLexemfailedTerminal = nullptr;

```

```

if (recursiveDescentParserRuleWithDebug(grammar->start_symbol, lexemIndex, lexemInfoTable, grammar, 0, &unexpectedLexemfailedTerminal)) {
    if (lexemInfoTable[lexemIndex].lexemStr[0] == '\0') {
        printf("Parse successful.\n");
        printf("%d.\n", lexemIndex);
        return SUCCESS_STATE;
    }
    else {
        printf("Parse failed: Extra tokens remain.\r\n");
        errorMessagesPtrToLastBytePtr += sprintf(errorMessagesPtrToLastBytePtr, "Parse failed: Extra tokens remain.\r\n");
        return ~SUCCESS_STATE;
    }
}
else {
    if (unexpectedLexemfailedTerminal) {
        printf("Parse failed.\r\n");
        printf(" (The predicted terminal does not match the expected one.\r\n Possible unexpected terminal \"%s\" on line %lld at position %lld\r\n ..., but this is not certain.)\r\n", unexpectedLexemfailedTerminal->lexemStr, unexpectedLexemfailedTerminal->row, unexpectedLexemfailedTerminal->col);
        errorMessagesPtrToLastBytePtr += sprintf(errorMessagesPtrToLastBytePtr, "Parse failed.\r\n");
        errorMessagesPtrToLastBytePtr += snprintf(errorMessagesPtrToLastBytePtr, MAX_LEXEM_SIZE + 128 + strlen(" (The predicted terminal does not match the expected one.\r\n Possible unexpected terminal \"%#\" on line # at position #\r\n ..., but this is not certain.)\r\n"), " (The predicted terminal does not match the expected one.\r\n Possible unexpected terminal \"%s\" on line %lld at position %lld\r\n ..., but this is not certain.)\r\n", unexpectedLexemfailedTerminal->lexemStr, unexpectedLexemfailedTerminal->row, unexpectedLexemfailedTerminal->col);
    }
    else {
        printf("Parse failed: unexpected terminal.\r\n");
        errorMessagesPtrToLastBytePtr += sprintf(errorMessagesPtrToLastBytePtr, "Parse failed: unexpected terminal.\r\n");
    }
    return ~SUCCESS_STATE;
}
}

return ~SUCCESS_STATE;
}

bool syntaxlAnalyze_(LexemInfo* lexemInfoTable, Grammar* grammar, char syntaxlAnalyzeMode, char* astFileName, char** errorMessagesPtrToLastBytePtr) {
    bool cykAlgorithmImplementationReturnValue = false;
    if (syntaxlAnalyzeMode == SYNTAX_ANALYZE_BY_CYK_ALGORITHM) {
        bool cykAlgorithmImplementationReturnValue = cykAlgorithmImplementation(lexemesInfoTable, grammar, astFileName);

        printf("cykAlgorithmImplementation return \"%s\".\r\n", cykAlgorithmImplementationReturnValue ? "true" : "false");
        if (!cykAlgorithmImplementationReturnValue) {
            writeBytesToFile(astFileName, (unsigned char*)"Error of AST build", strlen("Error of AST build"));
        }
    }
}

```

```

else if(astFileName && astFileName[0] != '\0') {

    writeBytesToFile(astFileName, (unsigned char*)"AST build no support.", strlen("AST build no support.));

}

if (cykAlgorithmImplementationReturnValue && syntaxlAnalyzeMode == SYNTAX_ANALYZE_BY_RECURSIVE_DESCENT) {

    int lexemIndex = 0;

    const struct LexemInfo unexpectedUnknownLexemfailedTerminal("unknown", 0, 0, 0, ~0, ~0); //

    const struct LexemInfo* returnUnexpectedLexemfailedTerminal = nullptr;

    if (nullptr == (returnUnexpectedLexemfailedTerminal = recursiveDescentParserWithDebug_(grammar->start_symbol, lexemIndex, lexemInfoTable, grammar, 0,
&unexpectedUnknownLexemfailedTerminal))) {

        if (lexemInfoTable[lexemIndex].lexemStr[0] == '\0') {

            printf("Parse successful.\n");

            printf("%d.\n", lexemIndex);

            return true;

        }

        else {

            printf("Parse failed: Extra tokens remain.\n");

            return false;

        }

    }

    else {

        if (returnUnexpectedLexemfailedTerminal->lexemStr[1]) {

            printf("Parse failed.\r\n");

            printf(" (The predicted terminal does not match the expected one.\r\n Possible unexpected terminal \"%s\" on line %lld at position %lld\r\n ..., but this is
not certain.)\r\n", returnUnexpectedLexemfailedTerminal->lexemStr, returnUnexpectedLexemfailedTerminal->row, returnUnexpectedLexemfailedTerminal->col);

        }

        else {

            printf("Parse failed: unexpected terminal.\r\n");

        }

        return false;

    }

    return false;

}

return false;

}

// OLD //

bool cykAlgorithmImplementationByCPPMap(struct LexemInfo* lexemInfoTable, Grammar* grammar) {

    if (lexemInfoTable == NULL || grammar == NULL) {

        return false;

    }

```



```

}

#ifdef _DEBUG

printf("ATTENTION: for better performance, use Release mode!\r\n");

#endif

#ifndef DEBUG_STATES

cout << "cykParse in progress.....[please wait]";

#else

cout << "cykParse in progress.....[please wait]: ";

#endif

map<int, map<int, set<string>>>> parseInfoTable;

int lexemIndex = 0;

for (--lexemIndex; lexemInfoTable[++lexemIndex].lexemStr[0];) {

#ifdef DEBUG_STATES

printf("\rcykParse in progress.....[please wait]: %02d %16s", lexemIndex, lexemInfoTable[lexemIndex].lexemStr);

#endif

// Iterate over the rules

for (int xIndex = 0; xIndex < grammar->rule_count; ++xIndex) {

string&& lhs = grammar->rules[xIndex].lhs;

Rule& rule = grammar->rules[xIndex];

// If a terminal is found

if (rule.rhs_count == 1 && (

lexemInfoTable[lexemIndex].tokenType == IDENTIFIER_LEXEME_TYPE && !strcmp(rule.rhs[0], "ident_terminal")

|| lexemInfoTable[lexemIndex].tokenType == VALUE_LEXEME_TYPE && !strcmp(rule.rhs[0], "value_terminal")

|| !strcmp(rule.rhs[0], lexemInfoTable[lexemIndex].lexemStr, MAX_LEXEM_SIZE)

)) {

parseInfoTable[lexemIndex][xIndex].insert(lhs);

}

}

for (int iIndex = lexemIndex; iIndex >= 0; --iIndex) {

for (int kIndex = iIndex; kIndex <= lexemIndex; ++kIndex) {

for (int xIndex = 0; xIndex < grammar->rule_count; ++xIndex) {

string&& lhs = grammar->rules[xIndex].lhs;

Rule& rule = grammar->rules[xIndex];

if (rule.rhs_count == 2

&& parseInfoTable[iIndex][kIndex].find(rule.rhs[0]) != parseInfoTable[iIndex][kIndex].end()

&& parseInfoTable[kIndex + 1][lexemIndex].find(rule.rhs[1]) != parseInfoTable[kIndex + 1][lexemIndex].end()

) {

parseInfoTable[iIndex][lexemIndex].insert(lhs);

}

}

}

}

}

```

```

    }
    }
    }
    }
}

cout << "\r" << "cykParse complete.....[   ok   ]\n";

if (parseInfoTable[0][lexemIndex - 1].find(grammar->start_symbol) == parseInfoTable[0][lexemIndex - 1].end()) {
    return false;
}

// parseByRecursiveDescent_(lexemInfoTable, grammar);
// displayParseInfoTable(parseInfoTable);
// saveParseInfoTableToFile(parseInfoTable, "parseInfoTable.txt");

ASTNode* astRoot = buildASTByCPPMap(parseInfoTable, grammar, 0, lexemIndex - 1, grammar->start_symbol);
if (astRoot) {
    std::cout << "Abstract Syntax Tree:\n";
    printAST(lexemInfoTable, astRoot);
    delete astRoot; // Не забуваємо звільняти пам'ять
}
else {
    std::cout << "Failed to build AST.\n";
}

//return parseInfoTable[0][lexemIndex - 1].find(grammar->start_symbol) != parseInfoTable[0][lexemIndex - 1].end(); // return !!parseInfoTable[0][lexemIndex - 1].size();
return true;
}

#endif

bool parseByRecursiveDescent(LexemInfo* lexemInfoTable, Grammar* grammar) {
    int lexemIndex = 0;

    const struct LexemInfo* unexpectedLexemfailedTerminal = nullptr;

    if (recursiveDescentParserRuleWithDebug(grammar->start_symbol, lexemIndex, lexemInfoTable, grammar, 0, &unexpectedLexemfailedTerminal)) {
        if (lexemInfoTable[lexemIndex].lexemStr[0] == '\0') {
            printf("Parse successful.\n");
            printf("%d.\n", lexemIndex);
            return true;
        }
    }
    else {

```

```

        printf("Parse failed: Extra tokens remain.\n");

        exit(0);

    }

}

else {

    if (unexpectedLexemfailedTerminal) {

        printf("Parse failed in line.\r\n");

        printf("    (The predicted terminal does not match the expected one.\r\n    Possible unexpected terminal \"%s\" on line %lld at position %lld\r\n    ..., but this is not certain.)\r\n", unexpectedLexemfailedTerminal->lexemStr, unexpectedLexemfailedTerminal->row, unexpectedLexemfailedTerminal->col);

    }

    else {

        printf("Parse failed: unexpected terminal.\r\n");

    }

    exit(0);

}

return false;

}

bool parseByRecursiveDescent_(LexemInfo* lexemInfoTable, Grammar* grammar) {

    int lexemIndex = 0;

    const struct LexemInfo unexpectedUnknownLexemfailedTerminal("unknown", 0, 0, 0, ~0, ~0); //

    const struct LexemInfo* returnUnexpectedLexemfailedTerminal = nullptr;

    if (nullptr == (returnUnexpectedLexemfailedTerminal = recursiveDescentParserWithDebug_(grammar->start_symbol, lexemIndex, lexemInfoTable, grammar, 0, &unexpectedUnknownLexemfailedTerminal))) {

        if (lexemInfoTable[lexemIndex].lexemStr[0] == '\0') {

            printf("Parse successful.\n");

            printf("%d.\n", lexemIndex);

            return true;

        }

        else {

            printf("Parse failed: Extra tokens remain.\n");

            exit(0);

        }

    }

    else {

        if (returnUnexpectedLexemfailedTerminal->lexemStr[1]) {

            printf("Parse failed.\r\n");

            printf("    (The predicted terminal does not match the expected one.\r\n    Possible unexpected terminal \"%s\" on line %lld at position %lld\r\n    ..., but this is not certain.)\r\n", returnUnexpectedLexemfailedTerminal->lexemStr, returnUnexpectedLexemfailedTerminal->row, returnUnexpectedLexemfailedTerminal->col);

        }

        else {

            printf("Parse failed: unexpected terminal.\r\n");

        }

        exit(0);

    }

}

```

```

    }

    return false;

}

#endif

While.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: while.cpp           *

*           (draft!) *

*****/

#include "../include/def.h"

#include "../include/generator/generator.h"

#include "../include/lexica/lexica.h"

#include "stdio.h"

#include "string.h"

unsigned char* makeWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_WHILE);

    if (multitokenSize) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;\"%s\"\\r\\n", tokenStruct[MULTI_TOKEN_WHILE][0]);

#endif

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = **lastLexemInfoInTable;

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = **lastLexemInfoInTable;

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue = (unsigned long long int)currBytePtr;

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("    LABEL@WHILE_%016lX:\\r\\n", (unsigned long long

int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    return currBytePtr;

}

unsigned char* makeNullStatementWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_NULL_STATEMENT);

```

```

        if (multitokenSize) {

            if (lexemInfoTransformationTempStackSize < 2

                || strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

                || strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

                || lexemInfoTransformationTempStackSize >= 4 &&
!strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr, tokenStruct[MULTI_TOKEN_WHILE][0],
MAX_LEXEM_SIZE)

                || lexemInfoTransformationTempStackSize >= 3 &&
!strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr, tokenStruct[MULTI_TOKEN_WHILE][0],
MAX_LEXEM_SIZE)

            ) {

                return currBytePtr;

            }

#ifdef DEBUG_MODE_BY_ASSEMBLY

                printf("\r\n");

                printf("    ;after cond expression (after \"%s\")\r\n", tokenStruct[MULTI_TOKEN_WHILE][0]);

#endif

const unsigned char code__cmp_eax_0[] = { 0x83, 0xF8, 0x00 };

const unsigned char code__jz_offset[] = { 0x0F, 0x84, 0x00, 0x00, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__cmp_eax_0, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jz_offset, 6);

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue = (unsigned long long int)(currBytePtr - 4);

//lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] =
lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1];

    strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++ - 1].lexemStr, MAX_LEXEM_SIZE);

//lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] =
lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1];

    strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++ - 1].lexemStr, MAX_LEXEM_SIZE);

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("    cmp eax, 0\r\n");

    printf("    jz LABEL@AFTER_WHILE_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr);

#endif

return *lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

```

```

unsigned char* makeContinueWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_CONTINUE_WHILE);

    if (multitokenSize) {

        if (

            lexemInfoTransformationTempStackSize >= 6

            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_THEN][0], MAX_LEXEM_SIZE)

            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_IF][0], MAX_LEXEM_SIZE)

            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 6].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

        ) {

            #ifdef DEBUG_MODE_BY_ASSEMBLY

                printf("\r\n");

                printf("    ;continue while (in \"then\"-part of %s-operator)\r\n", tokenStruct[MULTI_TOKEN_WHILE][0]);

            #endif

            const unsigned char code__jmp_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

            currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jmp_offset, 5);

            //lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].ifvalue = (unsigned long long int)(currBytePtr -
4);

            *((unsigned int*)(currBytePtr - 4)) = (unsigned int)((unsigned
char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 6].ifvalue - currBytePtr);

            strcpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr,
tokenStruct[MULTI_TOKEN_CONTINUE_WHILE][0], MAX_LEXEM_SIZE);

            #ifdef DEBUG_MODE_BY_ASSEMBLY

                printf("    jmp LABEL@WHILE_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 6].lexemStr);

            #endif

            return *lastLexemInfoInTable += multitokenSize, currBytePtr;

        }

        else if (

            lexemInfoTransformationTempStackSize >= 5

            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_ELSE][0], MAX_LEXEM_SIZE)

            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

        ) {

            #ifdef DEBUG_MODE_BY_ASSEMBLY

```

```

printf("\r\n");

printf("    ;continue while (in \"else\"-part of %s-operator)\r\n", tokenStruct[MULTI_TOKEN_WHILE][0]);

#endif

const unsigned char code__jmp_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jmp_offset, 5);

//lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].ifvalue = (unsigned long long int)(currBytePtr -
4);

*(unsigned int*)(currBytePtr - 4) = (unsigned int)((unsigned
char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].ifvalue - currBytePtr);

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr,
tokenStruct[MULTI_TOKEN_CONTINUE_WHILE][0], MAX_LEXEM_SIZE);

#ifdef DEBUG_MODE_BY_ASSEMBLY

printf("    jmp LABEL@WHILE_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr);

#endif

return *lastLexemInfoInTable += multitokenSize, currBytePtr;
}

else if (lexemInfoTransformationTempStackSize >= 4

    && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

    && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

    ) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

printf("\r\n");

printf("    ;continue while (in \"%s\")\r\n", tokenStruct[MULTI_TOKEN_WHILE][0]);

#endif

const unsigned char code__jmp_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jmp_offset, 5);

//lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue = (unsigned long long int)(currBytePtr -
4);

*(unsigned int*)(currBytePtr - 4) = (unsigned int)((unsigned
char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].ifvalue - currBytePtr);

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_CONTINUE_WHILE][0], MAX_LEXEM_SIZE);

#ifdef DEBUG_MODE_BY_ASSEMBLY

printf("    jmp LABEL@WHILE_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr);

#endif

```

```

#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

}

return currBytePtr;

}

unsigned char* makeExitWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_EXIT_WHILE);
    if (multitokenSize) {
        if (
            lexemInfoTransformationTempStackSize >= 6
            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_THEN][0], MAX_LEXEM_SIZE)
            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_IF][0], MAX_LEXEM_SIZE)
            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)
            && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 6].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)
        ) {

#ifdef DEBUG_MODE_BY_ASSEMBLY
            printf("\r\n");
            printf("    ;exit while (in \"then\"-part of %s-operator)\r\n", tokenStruct[MULTI_TOKEN_WHILE][0]);
#endif

            const unsigned char code__jmp_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

            currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jmp_offset, 5);

            lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].ifvalue = (unsigned long long int)(currBytePtr -
4);

            strcpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr,
tokenStruct[MULTI_TOKEN_EXIT_WHILE][0], MAX_LEXEM_SIZE);

#ifdef DEBUG_MODE_BY_ASSEMBLY
            printf("    jmp LABEL@AFTER_WHILE_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr);
#endif

            return *lastLexemInfoInTable += multitokenSize, currBytePtr;

        }

        else if (
            lexemInfoTransformationTempStackSize >= 5

```



```

        && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_ELSE][0], MAX_LEXEM_SIZE)

        && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

        && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

    ) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;exit while (in \"else\"-part of %s-operator)\r\n", tokenStruct[MULTI_TOKEN_WHILE][0]);

#endif

        const unsigned char code__jmp_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jmp_offset, 5);

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue = (unsigned long long int)(currBytePtr -
4);

        strcpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_EXIT_WHILE][0], MAX_LEXEM_SIZE);

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("    jmp LABEL@AFTER_WHILE_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr);

#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    else if (lexemInfoTransformationTempStackSize >= 4

        && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

        && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

    ) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("\r\n");

        printf("    ;exit while (in \"%s\")\r\n", tokenStruct[MULTI_TOKEN_WHILE][0]);

#endif

        const unsigned char code__jmp_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

        currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jmp_offset, 5);

        lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue = (unsigned long long int)(currBytePtr -
4);

```

```

        strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_EXIT_WHILE][0], MAX_LEXEM_SIZE);

#ifdef DEBUG_MODE_BY_ASSEMBLY

        printf("    jmp LABEL@ AFTER_WHILE_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr);

#endif

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

}

return currBytePtr;

}

unsigned char* makePostWhileCode_(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode, unsigned char
depthOfContext) {

    const unsigned char code__jmp_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 };

    //        if (!strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_CONTINUE_WHILE][0], MAX_LEXEM_SIZE)) {

    //            *(unsigned int*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue = (unsigned int)((unsigned
char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].ifvalue - currBytePtr - 4);

    //        }

    currBytePtr = outBytes2Code(currBytePtr, (unsigned char*)code__jmp_offset, 5);

    *(unsigned int*)(currBytePtr - 4) = (unsigned int)((unsigned char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize
- 4].ifvalue - currBytePtr);

    *(unsigned int*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].ifvalue = (unsigned int)(currBytePtr -
(unsigned char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].ifvalue - 4);

    if (!strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr,
tokenStruct[MULTI_TOKEN_EXIT_WHILE][0], MAX_LEXEM_SIZE)) {

        *(unsigned int*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue = (unsigned int)(currBytePtr -
(unsigned char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].ifvalue - 4);

    }

#ifdef DEBUG_MODE_BY_ASSEMBLY

    printf("    jmp LABEL@ WHILE_%016lX\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr);

    printf("    LABEL@ AFTER_WHILE_%016lX:\r\n", (unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr);

#endif

    return currBytePtr;

}

unsigned char* makeEndWhileAfterWhileCycleCode(struct LexemInfo** lastLexemInfoInTable, unsigned char* currBytePtr, unsigned char generatorMode) { // Or
Ender!

    unsigned char multitokenSize = detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_END_WHILE);

```

```

        if (multitokenSize

                && lexemInfoTransformationTempStackSize >= 4

                && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

                && !strcmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr,
tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)

        ) {

#ifdef DEBUG_MODE_BY_ASSEMBLY

                printf("\r\n");

                printf("        ;end of while\r\n");

#endif

        currBytePtr = makePostWhileCode_(lastLexemInfoInTable, currBytePtr, generatorMode, 0);

        lexemInfoTransformationTempStackSize -= 4;

        return *lastLexemInfoInTable += multitokenSize, currBytePtr;

    }

    return currBytePtr;

}

```

Add.h

```

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: add.h                               *

*           (draft!) *

*****/

#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);

and.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: and.h                               *

*           (draft!) *

*****/

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

```



```

//      "  long int AV\r\n" \
//      "  #*resultValue*#\r\n" \
//      "  long int RV\r\n" \
//      ";\r\n" \
//      "\r\n" \
//      "body\r\n" \
//      "  RV << 1; #*resultValue = 1; *#\r\n" \
//      "\r\n" \
//      "  #*input*#\r\n" \
//      "  get AV; #*scanf(\"%d\", &argumentValue); *#\r\n" \
//      "\r\n" \
//      "  #*compute*#\r\n" \
//      "  CL: #*label for cycle*#\r\n" \
//      "  if AV == 0 goto EL #*for (; argumentValue; --argumentValue)*#\r\n" \
//      "      RV << RV ** AV; #*resultValue *= argumentValue; *#\r\n" \
//      "      AV << AV -- 1; \r\n" \
//      "  goto CL\r\n" \
//      "  EL: #*label for end cycle*#\r\n" \
//      "\r\n" \
//      "  #*output*#\r\n" \
//      "  put RV; #*printf(\"%d\", resultValue); *#\r\n" \
//      "end" \

```

```
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);
```

```
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_ANALYSIS_MODE)

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

#define TOKENS_RE      ";|<-|\\+|-|\\*|,|=|!=|:|\\(|\\)|\\LT|\\GT|[_#0-9A-Za-z]+|[^\t\r\f\v\n]"

#define KEYWORDS_RE    ";|<-|\\+|-|\\*|,|=|!=|:|\\(|\\)|\\LT|\\GT|MAIMPROGRAM|DATA|START|END|EXIT|CONTINUE|END|GET|PUT|IF|ELSE|FOR|TO|DOWNTWO|DO|WHILE|REPEAT|UNTIL|GOTO|DIV|MOD|AND|OR|INTEGER"

#define IDENTIFIERS_RE  "_[A-Z][A-Z][A-Z][A-Z][A-Z]"

#define UNSIGNEDVALUES_RE "0|[1-9][0-9]*"

#define PROGRAM_FORMAT \

{"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 "MAIMPROGRAM"

#define T_BODY_0 "START"

#define T_DATA_0 "DATA"

#define T_DATA_TYPE_0 "INTEGER"

#define T_DATA_TYPE_1 ""

#define T_DATA_TYPE_2 ""

#define T_DATA_TYPE_3 ""

//

#define T_NOT_0 "!!"

#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 "LT"

#define T_LESS_OR_EQUAL_1 ""

#define T_LESS_OR_EQUAL_2 ""

#define T_LESS_OR_EQUAL_3 ""

#define T_GREATER_OR_EQUAL_0 "GT"

#define T_GREATER_OR_EQUAL_1 ""

#define T_GREATER_OR_EQUAL_2 ""

#define T_GREATER_OR_EQUAL_3 ""

//

#define T_ADD_0 "+"

#define T_ADD_1 ""

#define T_ADD_2 ""

#define T_ADD_3 ""

#define T_SUB_0 "-"

#define T_SUB_1 ""

#define T_SUB_2 ""

#define T_SUB_3 ""

#define T_MUL_0 "*"

#define T_MUL_1 ""

#define T_MUL_2 ""

#define T_MUL_3 ""

#define T_DIV_0 "DIV"

#define T_DIV_1 ""

#define T_DIV_2 ""

#define T_DIV_3 ""

#define T_MOD_0 "MOD"

#define T_MOD_1 ""

#define T_MOD_2 ""

#define T_MOD_3 ""

//

```

```

#define T_BIND_RIGHT_TO_LEFT_0 "<-"
#define T_BIND_RIGHT_TO_LEFT_1 ""
#define T_BIND_RIGHT_TO_LEFT_2 ""
#define T_BIND_RIGHT_TO_LEFT_3 ""

//

#define T_COMA_0 ","
#define T_COMA_1 ""
#define T_COMA_2 ""
#define T_COMA_3 ""
#define T_COLON_0 ":"
#define T_COLON_1 ""
#define T_COLON_2 ""
#define T_COLON_3 ""
#define T_GOTO_0 "GOTO"
#define T_GOTO_1 ""
#define T_GOTO_2 ""
#define T_GOTO_3 ""

//

#define T_IF_0 "IF"
#define T_IF_1 "("
#define T_IF_2 ""
#define T_IF_3 ""
#define T_THEN_0 ")"
#define T_THEN_1 ""
#define T_THEN_2 ""
#define T_THEN_3 ""
#define T_ELSE_0 "ELSE"
#define T_ELSE_1 ""
#define T_ELSE_2 ""
#define T_ELSE_3 ""

//

#define T_FOR_0 "FOR"
#define T_FOR_1 ""
#define T_FOR_2 ""
#define T_FOR_3 ""
#define T_TO_0 "TO"
#define T_TO_1 ""
#define T_TO_2 ""
#define T_TO_3 ""
#define T_DOWNT0_0 "DOWNT0"
#define T_DOWNT0_1 ""
#define T_DOWNT0_2 ""
#define T_DOWNT0_3 ""

```



```

#define T_DO_0 "DO"

#define T_DO_1 ""

#define T_DO_2 ""

#define T_DO_3 ""

//

#define T_WHILE_0 "WHILE"

#define T_WHILE_1 ""

#define T_WHILE_2 ""

#define T_WHILE_3 ""

#define T_CONTINUE_WHILE_0 "CONTINUE"

#define T_CONTINUE_WHILE_1 "WHILE"

#define T_CONTINUE_WHILE_2 ""

#define T_CONTINUE_WHILE_3 ""

#define T_EXIT_WHILE_0 "EXIT"

#define T_EXIT_WHILE_1 "WHILE"

#define T_EXIT_WHILE_2 ""

#define T_EXIT_WHILE_3 ""

#define T_END_WHILE_0 "END"

#define T_END_WHILE_1 "WHILE"

#define T_END_WHILE_2 ""

#define T_END_WHILE_3 ""

//

#define T_REPEAT_0 "REPEAT"

#define T_REPEAT_1 ""

#define T_REPEAT_2 ""

#define T_REPEAT_3 ""

#define T_UNTIL_0 "UNTIL"

#define T_UNTIL_1 ""

#define T_UNTIL_2 ""

#define T_UNTIL_3 ""

//

#define T_INPUT_0 "GET"

#define T_INPUT_1 ""

#define T_INPUT_2 ""

#define T_INPUT_3 ""

#define T_OUTPUT_0 "PUT"

#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 "END"
#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_DOWNT0,
MULTI_TOKEN_DO,

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

//
MULTI_TOKEN_REPEAT,
MULTI_TOKEN_UNTIL,
//

//
MULTI_TOKEN_INPUT,
MULTI_TOKEN_OUTPUT,
//

//
MULTI_TOKEN_RLBIND,
//

MULTI_TOKEN_SEMICOLON,

MULTI_TOKEN_BEGIN,
MULTI_TOKEN_END,

//

MULTI_TOKEN_NULL_STATEMENT
);

```

```

// #define PROCESS_TOKENS(...) HANDLE_TOKENS(__VA_ARGS__)

// #define TOKENS_FOR_MULTI_TOKEN(A, B, C, D) A, B, C, D

// #define TOKENS_FOR_MULTI_TOKEN_BITWISE_NOT TOKENS_FOR_MULTI_TOKEN("~", "", "", "")

#define INIT_TOKEN_STRUCT_NAME() static void initTokenStruct(){\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, NOT)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, AND)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, OR)\

\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, EQUAL)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, NOT_EQUAL)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, LESS_OR_EQUAL)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, GREATER_OR_EQUAL)\

\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, ADD)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, SUB)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, MUL)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, DIV)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, MOD)\

\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, BIND_RIGHT_TO_LEFT)\

\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, COLON)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, GOTO)\

\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, IF)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, THEN)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, ELSE)\

\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, FOR)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, TO)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, DOWNTO)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, DO)\

\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, WHILE)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, CONTINUE_WHILE)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, EXIT_WHILE)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, END_WHILE)\

\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, REPEAT)\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, UNTIL)\

\

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, INPUT)\

```

```

SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, OUTPUT)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, RLBIND)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, SEMICOLON)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, BEGIN)\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, END)\
\
SET_QUADRUPLE_STR_MACRO_IN_ARRAY(tokenStruct, NULL_STATEMENT)\

} char inititTokenStruct_ = (inititTokenStruct(), 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_ANALYSIS_MODE)

#define DEFAULT_MODE (DEBUG_MODE | LEXICAL_ANALYZE_MODE | SYNTAX_ANALYZE_MODE | SEMANTIX_ANALYZE_MODE |
MAKE_ASSEMBLY | MAKE_BINARY)

Def.h

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: def.h           *

*                               (draft!) *

*****/

#define SUCCESS_STATE 0

#define LEXICAL_ANALYZE_MODE 1 // lexicalAnalyze

#define MAKE_LEXEMES_SEQUENCE 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;


// #define EXPAND_MACRO(...) __VA_ARGS__ // Проміжний макрос для розгортання
//
// #define SET_QUADRUPLE_STR_MACRO_IN_ARRAY_(ARRAY, QUADRUPLE_ELEMENT_INDEX, ...) \
// SET_QUADRUPLE_STR_MACRO_IN_ARRAY(ARRAY, QUADRUPLE_ELEMENT_INDEX, EXPAND_MACRO(__VA_ARGS__))
//
// #define TOKENS_FOR_MULTI_TOKEN_BITWISE_NOT "~", "", "", ""
//
// SET_QUADRUPLE_STR_MACRO_IN_ARRAY_(tokenStruct, MULTI_TOKEN_BITWISE_NOT, TOKENS_FOR_MULTI_TOKEN_BITWISE_NOT)


// #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

//
// #define LEXICAL_ANALYSIS_MODE 1

```

```

//#define SEMANTIC_ANALISIS_MODE 2

//#define FULL_COMPILER_MODE 4

//

//#define DEBUG_MODE 512

//

//+!//#define MAX_PARAMETERS_SIZE 4096

//+!//#define PARAMETERS_COUNT 4

//+!//#define INPUT_FILENAME_PARAMETER 0

//

//#define DEFAULT_MODE (LEXICAL_ANALISIS_MODE | DEBUG_MODE)

div.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: div.h           *

*           (draft!) *

*****/

#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);

else.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: else.h           *

*           (draft!) *

*****/

#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);

equal.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: equal.h           *

*           (draft!) *

*****/

```



```

#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);

for.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: for.h           *

*           (draft!) *

*****/

#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);

generator.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: generator.h           *

*           (draft!) *

*****/

#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);

goto_label.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: goto_label.h           *

*           (draft!) *

*****/

```

```

#include <string>

#include <map>

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

#define LABEL_GOTO_LABEL_CODE(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);

greater_or_equal.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*          file: greater_or_equal.h          *

*          (draft!) *

*****/

#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);

if_then.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*          file: if_then.h          *

*          (draft!) *

*****/

#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);

input.h

#define _CRT_SECURE_NO_WARNINGS

```

```

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: input.h           *

*           (draft!) *

*****/

#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);

less_or_equal.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: less_or_equal.h           *

*           (draft!) *

*****/

#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);

lexica.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: lexica.h           *

*           (draft!) *

*****/

/////#define IDENTIFIER_LEXEME_TYPE 2
/////#define VALUE_LEXEME_TYPE 4

#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]));

mod.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: mod.h           *

*           (draft!) *

*****/

#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);

mul.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: mul.h           *

*           (draft!) *

*****/

#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);

not.h

#define _CRT_SECURE_NO_WARNINGS

```

```
/******
```

```
* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *
```

```
*           file: not.h           *
```

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

```
*****/
```

```
#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);
```

```
not_equal.h
```

```
#define _CRT_SECURE_NO_WARNINGS
```

```
/******
```

```
* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *
```

```
*           file: not_equal.h      *
```

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

```
*****/
```

```
#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);
```

```
#define _CRT_SECURE_NO_WARNINGS
```

```
/******
```

```
* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *
```

```
*           file: null_statement.h  *
```

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

```
*****/
```

```
#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);
```

```
operand.h
```

```
#define _CRT_SECURE_NO_WARNINGS
```

```
/******
```

```
* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *
```

```
*           file: operand.h         *
```

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

```
*****/
```

```
#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);

or.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: or.hxx           *

*           (draft!) *

*****/

#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);

output.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: output.hxx           *

*           (draft!) *

*****/

#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);

preparer.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: preparer.h           *

*           (draft!) *

*****/

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);

repeat_until.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: repeat_until.h           *

*           (draft!) *

*****/

#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);

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: rlbind.hxx           *

*           (draft!) *

*****/

#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);

semantix.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: semantix.h           *

*           (draft!) *

*****/

#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);

semikolon.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*          file: semicolon.hxx          *

*          (draft!) *

*****/

#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);

sub.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*          file: sub.hxx          *

*          (draft!) *

*****/

#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);

syntax.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: syntax.h           *

*           (draft!) *

*****/

#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);

while.h

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024-2025 // cw_sp2__2024_2025      *

*           file: while.hxx           *

*           (draft!) *

*****/

#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);

```