

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



Пояснювальна записка

до курсового проєкту "СИСТЕМНЕ ПРОГРАМУВАННЯ"

на тему: "РОЗРОБКА СИСТЕМНИХ ПРОГРАМНИХ МОДУЛІВ ТА КОМПОНЕНТ
СИСТЕМ ПРОГРАМУВАННЯ"

Індивідуальне завдання

"РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ"

Виконав студент групи КІ-307:

Затварницький
Віталій

Перевірив:
Козак Назар

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 проєктування:

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

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

3. За допомогою оператора вводу можна зчитати з клавіатури значення змінної; за допомогою оператора виводу можна вивести на екран значення змінної, виразу чи цілої константи.

4. В кожному завданні обов'язковим є оператор присвоєння, за допомогою якого можна реалізувати обчислення виразів з використанням заданих операцій і операцій круглі дужки (). У якості операндів можуть бути цілі константи, змінні, значення виразу.
5. В кожному завданні обов'язковим є оператор типу "блок" (вкладеність операторів), в якому мають бути вирази з тілом типу програми.
6. Необхідно реалізувати синтаксис вихідної мови, забезпечити реалізацію обчислення значень змінних, написати алгоритм з розгалуженням і циклічних обчислень.
7. Оператори різного виду допускаються і в будь-якій послідовності.
8. Для перевірки роботи розробленого транслятора необхідно написати три тестові програми на вихідній мові програмування.

АНОТАЦІЯ

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

Для реалізації транслятора визначено граматику вхідної мови програмування у термінах розширеної нотації Бекуса-Наура.

Реалізовано лексичний, синтаксичний, семантичний аналізатор. На етапі синтаксичного і семантичного аналізу відбувається перевірка програми на вхідній мові програмування на наявність помилок.

Перед генеруванням вихідного коду програма на вхідній мові програмування перетворюється у двійкове абстрактне синтаксичне дерево, обходячи яке генератор коду буде вихідний код на мові програмування C.

Розроблене програмне забезпечення налаштоване і протестоване на тестових прикладах.

ЗМІСТ

ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ.....	2
АНОТАЦІЯ	4
ЗМІСТ	5
ВСТУП	6
ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ	9
1. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ.....	12
1.1. ДЕТАЛІЗОВАНИЙ ОПИС ВХІДНОЇ МОВИ В ТЕРМІНАХ РОЗШИРЕНОЇ НОТАЦІЇ БЕКУСА-НАУРА.....	12
ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ У ТЕРМІНАХ РОЗШИРЕНОЇ ФОРМИ БЕКУСА- НАУРА:	13
1.2. ОПИС ТЕРМІНАЛЬНИХ СИМВОЛІВ ТА КЛЮЧОВИХ СЛІВ.	17
2. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ	19
2.1. ВИБІР ТЕХНОЛОГІЇ ПРОГРАМУВАННЯ.	19
2.2. ПРОЄКТУВАННЯ ТАБЛИЦЬ ТРАНСЛЯТОРА ТА ВИБІР СТРУКТУР ДАНИХ.	19
2.3. РОЗРОБКА ЛЕКСИЧНОГО АНАЛІЗАТОРА.	21
2.4. РОЗРОБКА СИНТАКСИЧНОГО ТА СЕМАНТИЧНОГО АНАЛІЗАТОРА.....	31
2.5. РОЗРОБКА ГЕНЕРАТОРА КОДУ.	39
3. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА	48
3.1. ОПИС ІНТЕРФЕЙСУ ТА ІНСТРУКЦІЇ КОРИСТУВАЧУ.	48
3.2. ВИЯВЛЕННЯ ЛЕКСИЧНИХ І СИНТАКСИЧНИХ ПОМИЛОК.....	50
3.3. ПЕРЕВІРКА РОБОТИ ТРАНСЛЯТОРА ЗА ДОПОМОГОЮ ТЕСТОВИХ ЗАДАЧ.....	52
ВИСНОВКИ	56
СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ	57
ДОДАТКИ	58

ВСТУП

Транслятор

Транслятор – це програма чи комплекс програм, що здійснюють переклад тексту, написаного однією мовою програмування (вхідна мова), в текст, поданий іншою мовою (вихідна мова).

Види трансляторів

Розрізняють транслятори двох видів:

1. **Компілюючого типу**
2. **Інтерпретуючого типу**

Компілятор

Компілятор – це транслятор, для якого:

- **Вхідна мова:** мова високого рівня (наприклад, C, Pascal, Algol).
- **Вихідна мова:** мова асемблера чи мова машинних команд.

Особливості:

- Переклад вхідної програми на вихідну мову виконується одразу цілком.
- Вхідна та вихідна програми завжди подаються у вигляді тексту.

Асемблер

Асемблер – це компілятор, у якому:

- **Вхідна мова:** мова асемблера.
- **Вихідна мова:** мова машинних команд.

Інтерпретатор

Інтерпретатор – це транслятор, що:

- Здійснює пооператорний переклад тексту програми на вихідну мову.
- Одночасно виконує ці оператори.

Результат: на виході інтерпретатора отримуємо результат роботи вхідної програми.

Структура транслятора (компілятора)

Загальна структура транслятора (компілятора) показана на рис.1.

Лексичний аналізатор (scanner, сканер)

- Здійснює перетворення вхідного тексту програми (рядок символів) у рядок лексем, поданий у цифровій формі.
- Виявляє лексичні помилки.

Лексема – це найменша одиниця інформації, яка обробляється синтаксичним аналізатором.

Приклади лексем:

1. **Односимвольні роздільники:** ,, ;, .
2. **Знаки операцій:** +, -, *, /
3. **Багатосимвольні роздільники:** <=, <>
4. **Ідентифікатори**
5. **Константи**
6. **Ключові слова:** for, while тощо.

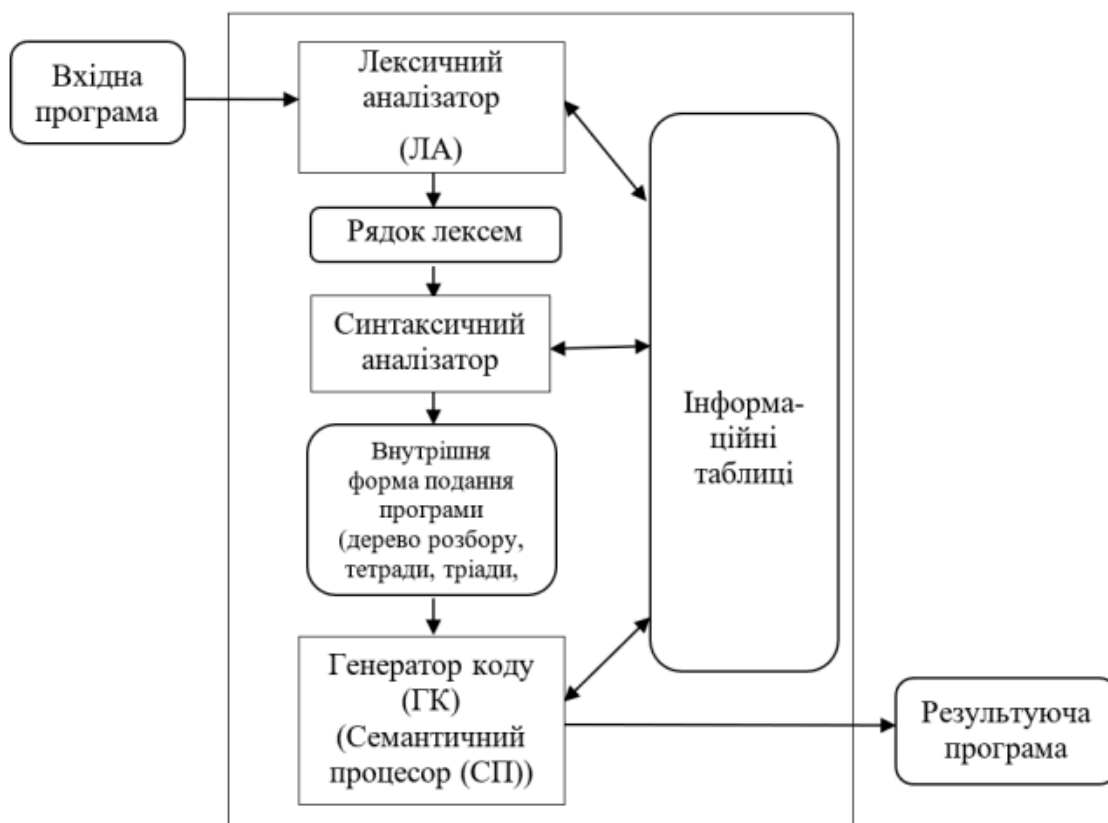


Рис.1. Структура транслятора (компілятора)

Синтаксичний аналізатор (parser, парсер)

Синтаксичний аналізатор:

- Здійснює декомпозицію вхідної програми (рядок лексем) у структурні одиниці мови:
 - Оператори
 - Описання
 - Декларації
- Перевіряє відповідність граматиці вхідної мови.
- Виявляє синтаксичні помилки.

Дерево розбору

Це внутрішня форма подання вхідної програми, яка:

- Містить структурні одиниці мови.
- Відображає зв'язки між ними.

Генератор коду (семантичний процесор)

Генератор коду перетворює:

- Вхідну програму, подану у внутрішній формі.
- Вихідний текст у вигляді операторів або команд вихідною мовою.

Це відбувається на основі семантики вхідної мови.

Семантичний процесор – це інша назва генератора коду.

Метамови

Для опису мов програмування використовуються спеціальні мови – метамови:

- Метасинтаксична мова для опису синтаксису.
- Метасемантична мова для опису семантики.

Теоретична основа трансляторів

1. Лексичний і синтаксичний аналізатори
 - Базуються на теорії формальних граматик.
2. Генератор коду
 - Використовує мови та методи опису семантики мов програмування.

ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ

1. Огляд методів та способів проєктування трансляторів

Проектування трансляторів є багатоступеневим процесом, що базується на використанні формальних методів опису мов програмування та побудови алгоритмів. Основні методи та способи проєктування трансляторів поділяються на такі етапи:

Формальний опис мови

Для опису мов програмування використовуються спеціальні засоби:

- Граматики (контекстно-вільні, регулярні) для формалізації синтаксису
- Семантичні правила для визначення поведінки програми
- Метамови:
 - Метасинтаксична мова для опису структури програми
 - Метасемантична мова для визначення значення та дій

Розробка лексичного аналізатора

- Виділення лексем із тексту програми
- Перетворення вхідного тексту на послідовність символів, що обробляються синтаксичним аналізатором

Синтаксичний аналіз

- Перетворення послідовності лексем у дерево розбору
- Виявлення синтаксичних помилок

Семантичний аналіз

- Перевірка семантичної коректності програми (типи даних, області видимості)
- Генерація проміжного представлення програми

Генерація коду

- Перетворення програми у вихідний код цільової мови (мови асемблера чи машинних команд)
- Оптимізація отриманого коду

Тестування та верифікація

- Перевірка коректності роботи транслятора на тестових програмах
- Виявлення та виправлення помилок у реалізації

Види підходів до проєктування

Ручне проєктування: використання алгоритмів та структур даних для реалізації транслятора

Автоматизоване проєктування: застосування генераторів аналізаторів, таких як Lex і Yacc

Комбіноване проєктування: поєднання ручного та автоматизованого підходів

Проектування трансляторів є складною інженерною задачею, яка потребує ґрунтовних знань теорії формальних мов і алгоритмів, а також практичних навичок у розробці програмного забезпечення.

1. Огляд методів та способів проєктування трансляторів

Проектування трансляторів є багатоступеневим процесом, що включає формальний опис мов, створення алгоритмів та інструментів для їх реалізації. Основні методи та способи проєктування поділяються на три категорії: ручне, автоматизоване та комбіноване проєктування.

Ручне проєктування

Цей метод передбачає розробку транслятора без використання спеціалізованих інструментів автоматизації. Основні етапи:

1. Аналіз вимог

- Визначення вхідної та вихідної мов.
- Опис граматики вхідної мови.

2. Розробка алгоритмів

- Лексичний аналізатор для виділення лексем.
- Синтаксичний аналізатор для побудови дерева розбору.
- Семантичний аналізатор для перевірки логічної коректності програми.
- Генератор коду для створення вихідного тексту цільовою мовою.

3. Реалізація

- Написання програмного коду аналізаторів та генератора вручну.
- Оптимізація реалізації для покращення продуктивності.

4. Переваги

- Гнучкість у реалізації.
- Можливість адаптації під специфічні вимоги.

5. Недоліки

- Значний час розробки.
- Велика ймовірність помилок через людський фактор.

Автоматизоване проєктування

Цей метод використовує спеціалізовані інструменти для автоматичної генерації частин транслятора.

1. Генерація лексичного аналізатора

- Використовуються інструменти, такі як Lex або Flex, для автоматичного створення сканера.

2. Генерація синтаксичного аналізатора

- Інструменти, наприклад, Yacc або Bison, забезпечують автоматичну генерацію парсера на основі грамматики.

3. Автоматизоване створення генератора коду

- Використовуються генератори шаблонів або середовища з підтримкою метасемантичних мов.

4. Переваги

- Швидкість розробки.
- Зменшення ймовірності помилок завдяки перевіреним інструментам.

5. Недоліки

- Обмежена гнучкість.
- Залежність від специфіки інструментів.

Комбіноване проєктування

Цей підхід поєднує ручне та автоматизоване проєктування, щоб використати переваги обох методів.

1. Основні етапи

- Лексичний і синтаксичний аналізатор генеруються автоматично.
- Семантичний аналізатор та генератор коду створюються вручну для забезпечення максимальної гнучкості.

2. Переваги

- Баланс між швидкістю розробки та гнучкістю.
- Можливість точного налаштування критичних компонентів.

3. Недоліки

- Необхідність знань як ручного, так і автоматизованого підходу.
- Може зайняти більше часу, ніж повністю автоматизований метод.

1. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ

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

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

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

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

символ = <вираз, що містить символи>

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

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

Опис вхідної мови програмування у термінах розширеної форми Бекуса-

Наура:

```

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 | forto_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
// -- hello wolrd
letter_in_lower_case = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w
| x | y | z
    letter_in_upper_case = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
U | V | W | X | Y | Z
    ident = tokenUNDERSCORE , letter_in_upper_case , letter_in_upper_case ,
letter_in_upper_case , letter_in_upper_case , letter_in_upper_case , letter_in_upper_case ,
letter_in_upper_case
    label = letter_in_lower_case , {letter_in_lower_case}
//
sign = sign_plus | sign_minus
sign_plus = '-'
sign_minus = '+'
//
digit_0 = '0'
digit_1 = '1'
digit_2 = '2'
digit_3 = '3'
digit_4 = '4'
digit_5 = '5'
digit_6 = '6'
digit_7 = '7'
digit_8 = '8'
digit_9 = '9'
//
tokenCOLON = ":"
tokenGOTO = "goto"
tokenINTEGER16 = "integer16"
tokenCOMMA = ","
tokenNOT = "not"
tokenAND = "and"
tokenOR = "or"
tokenEQUAL = "="
tokenNOTEQUAL = "<>"
tokenLESSOREQUAL = "<"
tokenGREATEROREQUAL = ">"
tokenPLUS = "add"
tokenMINUS = "sub"
tokenMUL = "*"

```

```

tokenDIV = "/"
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 = "scan"
tokenPUT = "print"
tokenNAME = "program"
tokenBODY = "start"
tokenDATA = "var"
tokenEND = "finish"
tokenSEMICOLON = ""
//
tokenUNDERSCORE = "_"
//
A = "A"
B = "B"
C = "C"
D = "D"
E = "E"
F = "F"
G = "G"
H = "H"
I = "I"
J = "J"
K = "K"
L = "L"
M = "M"
N = "N"
O = "O"
P = "P"
Q = "Q"
R = "R"
S = "S"
T = "T"
U = "U"

```

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


1.2. Опис термінальних символів та ключових слів.

Визначаємо термінальні символи і ключові слова:

- **start** – початок програми
- **var** – оголошення змінних
- **finish** – кінець програми
- **integer16** – тип даних
- **scan** – оператор вводу
- **print** – оператор виводу
- **if, else** – умовний оператор
- **<-** – оператор присвоєння
- **goto** – оператор переходу
- **for (to – downto)** – оператор циклу
- **repeat-until** – оператор циклу з постумовою
- **add** – додавання
- **sub** – віднімання
- ***** – множення
- **/** – ділення
- **%** – додавання за модулем 2
- **>** – більше
- **<** – менше
- **=** – рівність
- **<>** – нерівність
- **not** – заперечення
- **and** – логічне І
- **or** – логічне АБО
- **;** – кінець оператора
- **,** – розділювач змінних
- **(** – відкрита дужка
- **)** – закрита дужка
- **!!** – початок коментаря
- **[a...z][A...Z]** – маленькі і великі латинські букви
- **0...9** – цифри
- символи табуляції, переходу на новий рядок, пробіл

Програма на вихідній мові програмування має починатись з ключового слова `start`, далі має йти розділ опису змінних `var`. Між розділом `var` і ключовим словом `finish` розміщуються оператори програми. Оператори є 4: оператор вводу даних `scan`, оператор виводу даних `print`, оператор присвоєння `<-` і умовний оператор `if` – `[- else]`. Кожен оператор має завершуватись символом крапка з комою `;`.

Оператор присвоєння дозволяє присвоїти деякій змінній значення арифметичного виразу. Допустимі арифметичні операції: `add`, `-`, `sub`, `/`. Операндами можуть бути змінні, цілі додатні константи і інші вирази, взяті в дужки.

В умовному операторі використовуються логічні вирази, допустимі такі операції порівняння `>`, `<`, `=`, `<>` і такі логічні операції `not` and `or`.

Ідентифікаторами (імена змінних) можуть бути довжиною 6-х символів і складатись з 6 латинських літер. Перший символ ідентифікатора завжди велика літера наступні 5 завжди мала літера. Тип даних лише один – `integer16`, при оголошенні декількох змінних вони записуються через кому, вкінці опису змінних ставиться символ крапка з комою `;`. Коментарі починаються з `!!`.

Приклади оголошення змінних:

```
integer16 aINTEG;
```

```
integer16 aINTEG, hINTEG, cINTEG;
```

Приклади арифметичних виразів:

```
1000 sub 7
```

```
aINTEG add 69
```

```
666 add 666 * aINTEG add 4308
```

```
cNIGEG * (14 add 88) sub 50 / b
```

Приклади логічних виразів:

```
aINTEG > bINTEG
```

```
aINTEG > bINTEG and aINTEG > 1
```

```
not(aINTEG < cINTEG)
```

```
aINTEG <> bINTEG
```

```
aINTEG <> 0 sub 100
```

2. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ

2.1. Вибір технології програмування.

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

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

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

2.2. Проектування таблиць транслятора та вибір структур даних.

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

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

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

Опис структури LexemInfo

LexemInfo — це структура, яка використовується для зберігання інформації про окрему лексему, отриману під час лексичного аналізу. Вона має публічний доступ до своїх членів і призначена для забезпечення зручного доступу до атрибутів лексеми. Нижче детально описані її елементи та функції:

Члени структури:

1. **char lexemStr[MAX_LEXEM_SIZE]**
Масив символів, що містить саму лексему у вигляді рядка.
MAX_LEXEM_SIZE — це максимальний розмір лексеми, зазвичай визначений як константа.
2. **unsigned long long int lexemId**
Унікальний ідентифікатор лексеми. Він дозволяє відрізнити лексеми між собою.
3. **unsigned long long int tokenType**
Тип токена, який відповідає лексемі. Наприклад, це може бути константа, оператор, ключове слово тощо.
4. **unsigned long long int ifvalue**
Додаткове значення, яке використовується для обробки умовних виразів або контексту лексеми. Наприклад, це може бути значення для порівняння чи виконання умов.
5. **unsigned long long int row**
Номер рядка, де знаходиться лексема в коді. Це корисно для відлагодження або повідомлень про помилки.
6. **unsigned long long int col**
Номер колонки в рядку, де розташована лексема.
7. **// TODO: ...**
Коментар, який вказує, що до структури можуть бути додані нові члени або властивості для розширення її функціональності.

Конструктори:

1. **Конструктор за замовчуванням: LexemInfo()**
Ініціалізує структуру з початковими значеннями. Зазвичай це нульові або порожні значення для членів структури.
2. **Параметризований конструктор: LexemInfo(const char* lexemStr, unsigned long long int lexemId, unsigned long long int tokenType, unsigned long long int ifvalue, unsigned long long int row, unsigned long long int col)**
Ініціалізує структуру з заданими значеннями.
 - **lexemStr**: рядок лексеми.
 - **lexemId**: унікальний ідентифікатор.
 - **tokenType**: тип токена.
 - **ifvalue**: додаткове значення.

- **row**: номер рядка.
- **col**: номер колонки.

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

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

Призначення:

Ця структура є корисною для:

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

2.3. Розробка лексичного аналізатора.

Основна задача лексичного аналізу – розбити вихідний текст, що складається з послідовності символів, на послідовність слів, або лексем, тобто виділити ці слова з безперервної послідовності символів. Всі символи вхідної послідовності з цієї точки зору розділяються на символи, що належать яким-небудь лексемам, і символи, що розділяють лексеми. В цьому випадку використовуються звичайні засоби обробки рядків. Вхідна програма проглядається послідовно з початку до кінця. Базові елементи, або лексичні одиниці, розділяються пробілами, знаками операцій і спеціальними символами (новий рядок, знак табуляції), і таким чином виділяються та розпізнаються ідентифікатори, літерали і термінальні символи (операції, ключові слова).

При виділенні лексеми вона розпізнається та записується у таблицю лексем за допомогою відповідного номера лексеми, що є унікальним для кожної лексеми із усього можливого їх набору. Це дає можливість наступним фазам компіляції звертатись лексеми не як до послідовності символів, а як до унікального номера лексеми, що значно спрощує роботу синтаксичного аналізатора: легко перевіряти належність лексеми до відповідної синтаксичної конструкції та є можливість легкого перегляду програми, як вгору, так і вниз, від текучої позиції аналізу. Також в таблиці лексем ведуться записи, щодо рядка відповідної лексеми – для місця помилки – та додаткова інформація.

Лексична фаза відкидає коментарі, оскільки вони не мають ніякого впливу на виконання програми, отже ж й на синтаксичний розбір та генерацію коду.

Розділимо лексеми на типи або лексичні класи:

- Ключові слова (**start, var, finish, scan, print, integer16, if, else, for, goto, downto, repeat, until, while**)
- Ідентифікатори
- Числові константи (ціле число без знаку)
- Оператор присвоєння (<-)
- Знаки операції (**add, sub, *, /, >, <, =, <>, not, and, or**)
- Розділювачі (; ,)
- Дужки ((,))

2.3.1. Розробка алгоритму роботи лексичного аналізатора.

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

Ось як працює цей аналізатор:

1. Основні структури даних

LexemInfo

Містить інформацію про кожну лексему:

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

NonContainedLexemInfo

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

2. Основні масиви

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

3. Алгоритм лексичного аналізу

3.1. Токенізація (tokenize)

Ця функція розбиває текст на токени відповідно до регулярного виразу:

- Регулярний вираз (TOKENS_RE) визначає, які символи формують токен (ідентифікатори, ключові слова, числа тощо).

- За допомогою ітератора (`std::sregex_token_iterator`) текст обробляється токен за токеном.

3.2. Ідентифікація токена (`lexicalAnalyze`)

Для кожного токена викликаються функції:

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

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

4. Обробка ключових слів, ідентифікаторів та значень

Ключові слова

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

Ідентифікатори

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

Значення

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

5. Обробка коментарів (`commentRemover`)

Функція видаляє коментарі з тексту. Вона підтримує:

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

6. Збереження позицій (`setPositions`)

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

7. Друк результатів (printLexemes)

Результати аналізу виводяться у вигляді таблиці, де показано:

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

Структура та поля результатів лексичного аналізатора

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

Поля таблиці:

1. Індекс лексеми (index)

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

2. Текст лексеми (lexemStr)

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

3. Ідентифікатор лексеми (lexemId)

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

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

4. Тип лексеми (tokenType)

Визначає тип лексеми, наприклад:

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

5. Значення (ifvalue)

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

6. Рядок (row)

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

7. Стовпець (col)

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

Стани під час аналізу

Лексичний аналізатор проходить кілька основних станів:

1. Ініціалізація

Підготовка таблиць і структур, зокрема:

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

2. Обробка тексту

- Видалення коментарів.
- Розбиття тексту на токени.

3. Класифікація лексем

Для кожної лексеми визначають:

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

4. Формування таблиці результатів

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

5. Виведення результатів

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

Табл 1.1

Індекс	Текст лексеми	Ідентифікатор	Тип	Значення	Рядок	Стовпець
0	program	101	Ключове слово	-	1	1
1	m3rgty	1	Ідентифікатор	-	1	9
2	start	102	Ключове слово	-	2	1
3	123	1001	Значення	123	3	5
4	finish	103	Ключове слово	-	4	1

Преваги такої структури:

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

8. Особливості

1. Буферизація:

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

2. Гнучкість:

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

3. Обробка помилок:

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

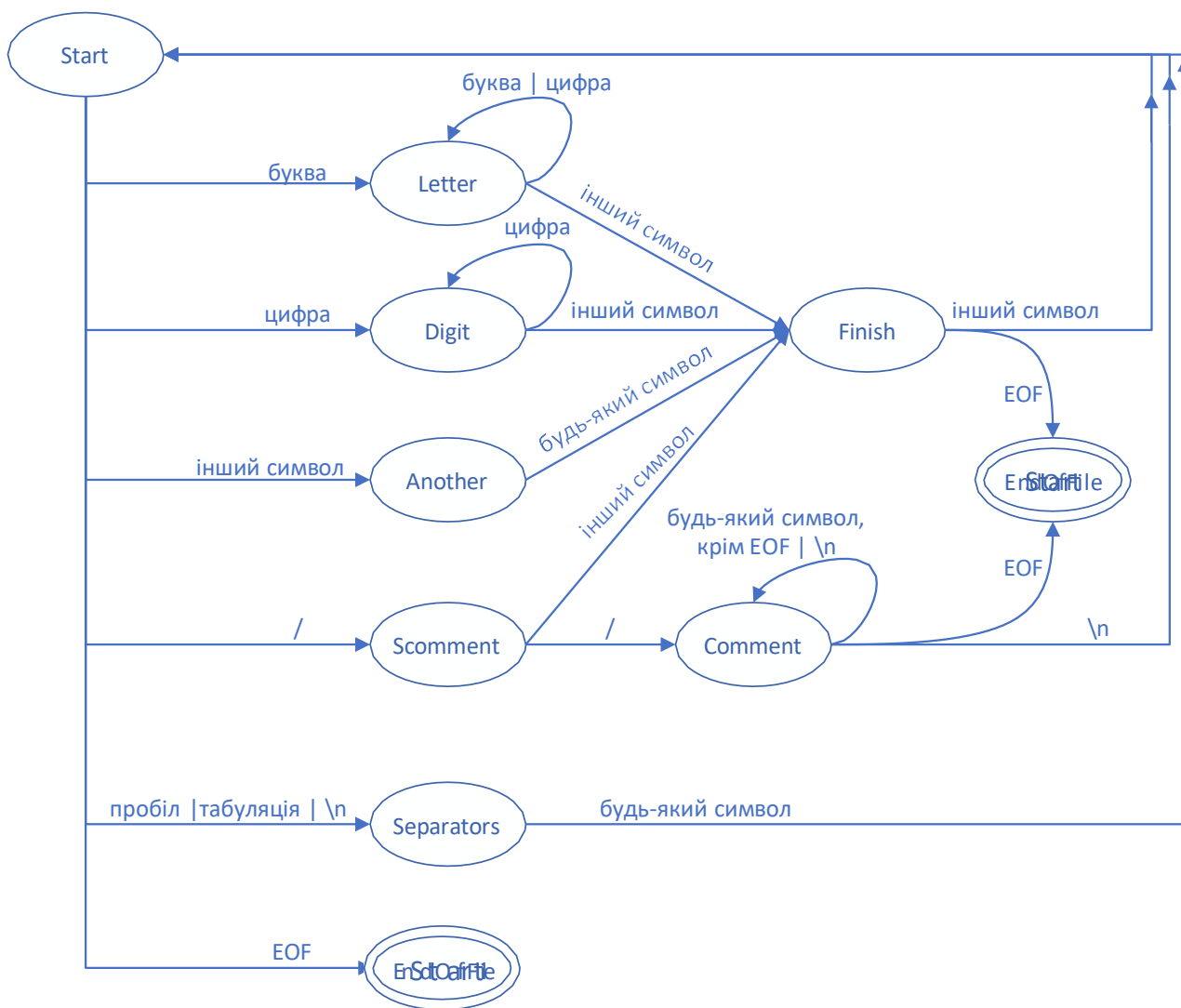


Рис. 3.1. Граф-схема алгоритму роботи лексичного аналізатора.

2.3.2. Опис програми реалізації лексичного аналізатора.

Основна задача лексичного аналізатора – розділити вхідний текст програми, що складається з послідовності символів, на окремі лексеми, тобто слова, які мають змістовне значення для подальшого аналізу. Усі символи вхідної послідовності поділяються на ті, що належать лексемам, і ті, що виконують функцію роздільників. У процесі аналізу використовуються стандартні методи обробки рядків. Вхідний текст програми переглядається послідовно від початку до кінця, а базові елементи (лексичні одиниці) виділяються на основі пробілів, знаків операцій, спеціальних символів (таких як новий рядок або табуляція). У результаті розпізнаються ідентифікатори, літерали та термінальні символи (наприклад, операції або ключові слова).

Програма аналізує файл доти, доки не досягне його кінця. Для обробки вхідного файлу викликається функція `tokenize()`. Ця функція читає вміст файлу, виділяє лексеми та порівнює їх із зарезервованими словами. У разі збігу лексемі присвоюється відповідний тип або значення (якщо це числова константа).

Кожна виділена лексема додається до списку `m_tokens` з використанням унікального типу лексеми. Це дозволяє наступним фазам компіляції оперувати лексемами не як послідовностями символів, а як конкретними типами, що значно полегшує синтаксичний аналіз. Наприклад, перевірка належності лексеми до певної синтаксичної конструкції або навігація текстом програми (вперед і назад від поточної позиції) стають більш зручними. У таблиці лексем також зберігається інформація про рядок і стовпець кожної лексеми, що спрощує пошук місця помилки. Додатково зберігається метайнформація, корисна для подальших етапів аналізу.

Під час лексичного аналізу виявляються та відзначаються лексичні помилки, наприклад, некоректні символи або невірні ідентифікатори ігноруються, оскільки вони не впливають ні на синтаксичний розбір, ні на генерацію коду.

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

2.4. Розробка синтаксичного та семантичного аналізатора.

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

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

1. Лексичний аналіз

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

2. Метод СҮК для синтаксичного аналізу

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

3. Рекурсивний спуск

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

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

4. Побудова абстрактного синтаксичного дерева (AST)

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

5. Виведення AST

Для візуалізації AST використовуються функції:

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

6. Збереження таблиці СҮК

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

7. Основна функція синтаксичного аналізу

Функція `syntaxAnalyze` координує процес:

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

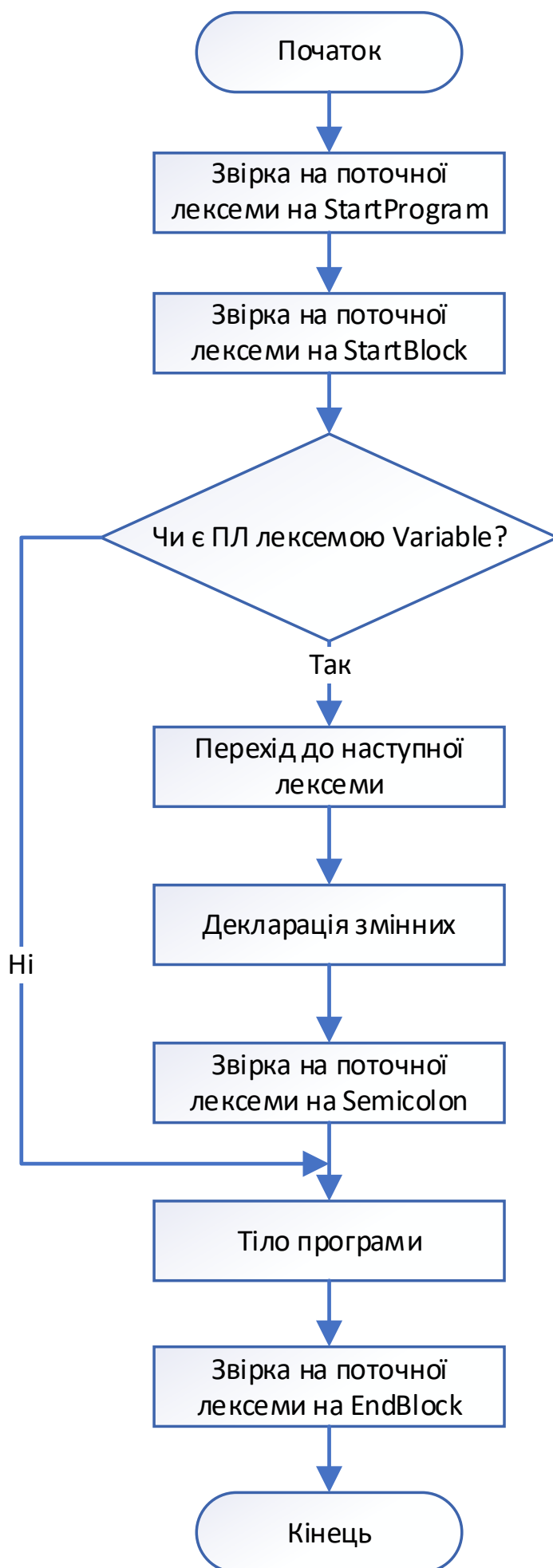


Рис 3. Блок схема роботи синтаксичного аналізатора

2.4.1. Розробка дерева граматичного розбору.

Схема дерева розбору виглядає наступним чином:

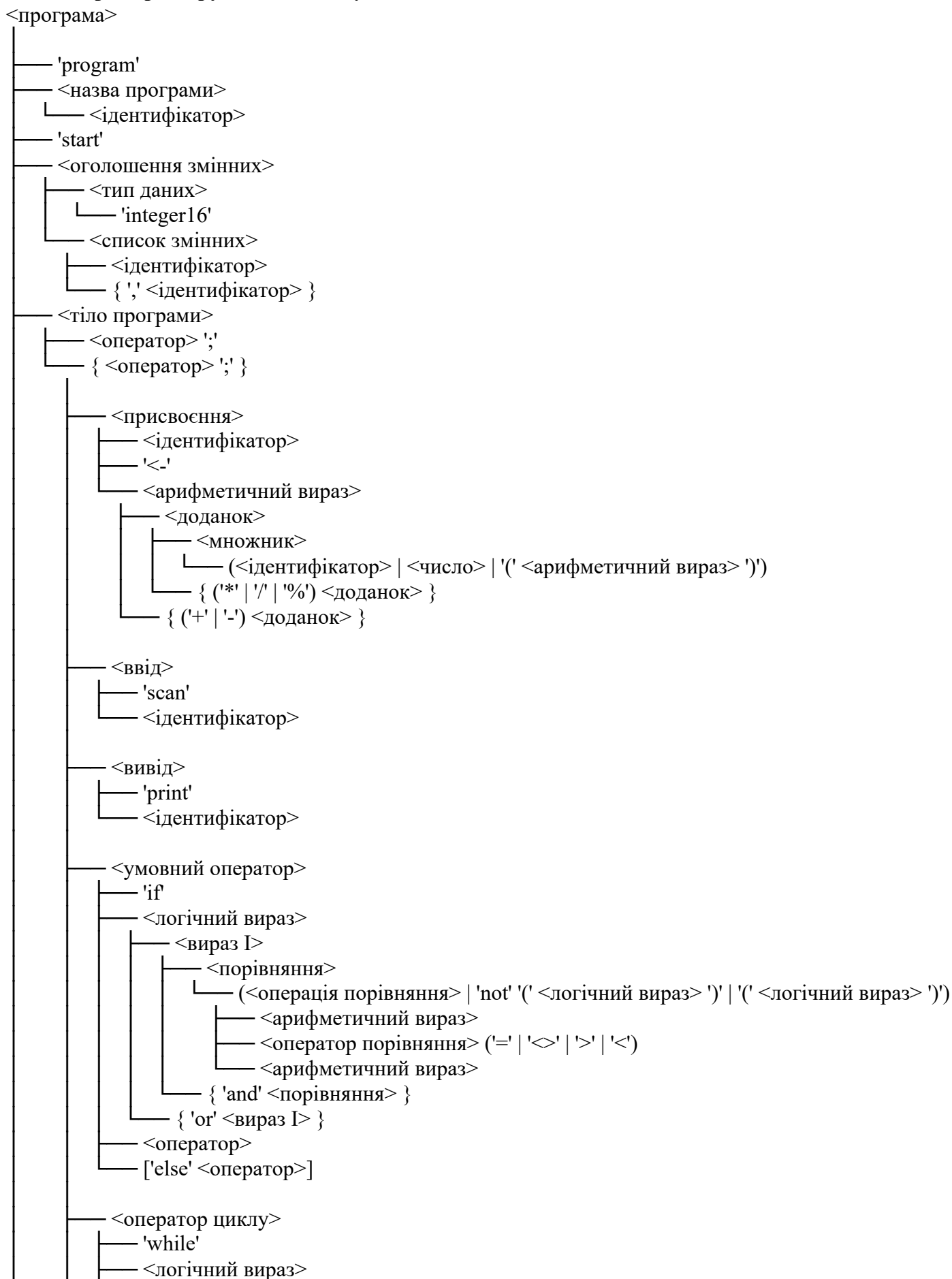




Рис. 3.2. Дерево граматичного розбору.

2.4.2. Розробка алгоритму роботи семантичного аналізатора

На етапі семантичного аналізу нам необхідно вирішити задачу ідентифікації ідентифікаторів. Алгоритм ідентифікації складається з двох частин:

- перша частина алгоритму опрацьовує оголошення ідентифікаторів;
- друга частина алгоритму опрацьовує використання ідентифікаторів.

Нехай лексичний аналізатор видав чергову лексему, що є ідентифікатором. Лексичний аналізатор сформував структуру, що містить атрибути виділеної лексеми, такі як ім'я ідентифікатора, його тип і лексичний клас. Далі вся ця інформація передається семантичному аналізатору. Припустимо, що в даний момент опрацьовується оголошення ідентифікатора. Основна семантична дія в цьому випадку полягає в занесенні інформації про ідентифікатор у таблицю ідентифікаторів.

Опрацювання використання ідентифікатора. Припустимо, що уже побудовано (цілком чи частково) таблицю ідентифікаторів. Далі вся ця інформація передається фазі використання ідентифікаторів. Таким чином, відомо, що опрацьовується використання ідентифікатора. Для того, щоб одержати інформацію про тип ідентифікатора нам достатньо прочитати певне поле таблиці ідентифікаторів.

2.4.3. Опис програми реалізації семантичного аналізатора.

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

Основні функції семантичного аналізатора

1. `getLastDataSectionLexemIndex`

Ця функція знаходить індекс останньої лексеми у секції даних.

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

2. `checkingInternalCollisionInDeclarations`

Перевіряє внутрішні колізії у деклараціях змінних і міток:

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

3. `checkingVariableInitialization`

Перевіряє, чи ініціалізовано всі змінні перед використанням:

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

4. `checkingCollisionInDeclarationsByKeywords`

Перевіряє, чи збігаються імена декларацій з ключовими словами:

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

5. **semantixAnalyze**

Головна функція, що викликає всі попередні модулі аналізу:

- Перевіряє колізії в деклараціях.
- Аналізує ініціалізацію змінних.
- Перевіряє збіг імен з ключовими словами.
- Якщо хоча б одна перевірка не проходить, повертається відповідний код помилки.

Ключові аспекти реалізації

1. **Лексеми та граматика:**

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

2. **Перевірка колізій:**

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

3. **Робота з регулярними виразами:**

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

4. **Повідомлення про помилки:**

Усі помилки виводяться у консоль із деталізацією, наприклад:

5. Collision(identifier/identifier): myVariable

6. Uninitialized: myVariable

7. **Коди стану:**

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

Типовий процес роботи

1. Виклик функції `semantixAnalyze`, яка:
 - Перевіряє декларації та їх колізії.
 - Аналізує ініціалізацію змінних.
 - Виявляє невірне використання ключових слів.
2. У разі помилки повертається відповідний код, і програма виводить інформацію про проблему.

2.5. Розробка генератора коду.

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

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

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

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

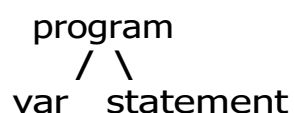
Кожен вузол відповідає певному типу конструкції коду (наприклад, оператору додавання, виклику функції, оголошенню змінної).

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

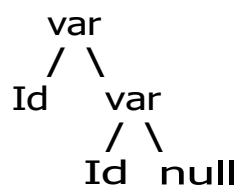
2.5.1. Розробка алгоритму роботи генератора коду.

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

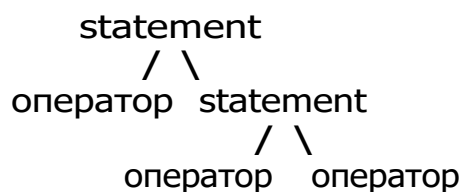
Програма має вигляд:



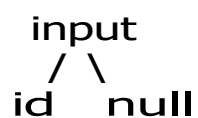
Оголошення змінних:



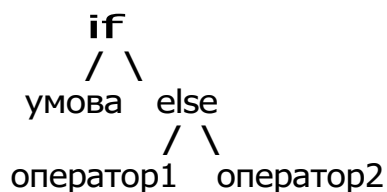
Тіло програми:



Оператор вводу (виводу):



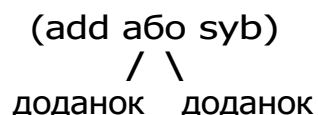
Умовний оператор:



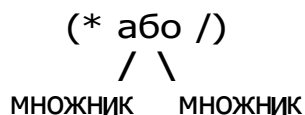
Оператор присвоєння:



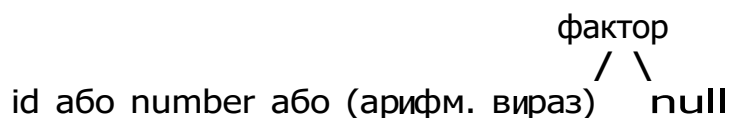
Арифметичний вираз:



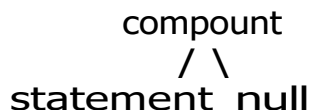
Доданок:



Множник:



Складений оператор:



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

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

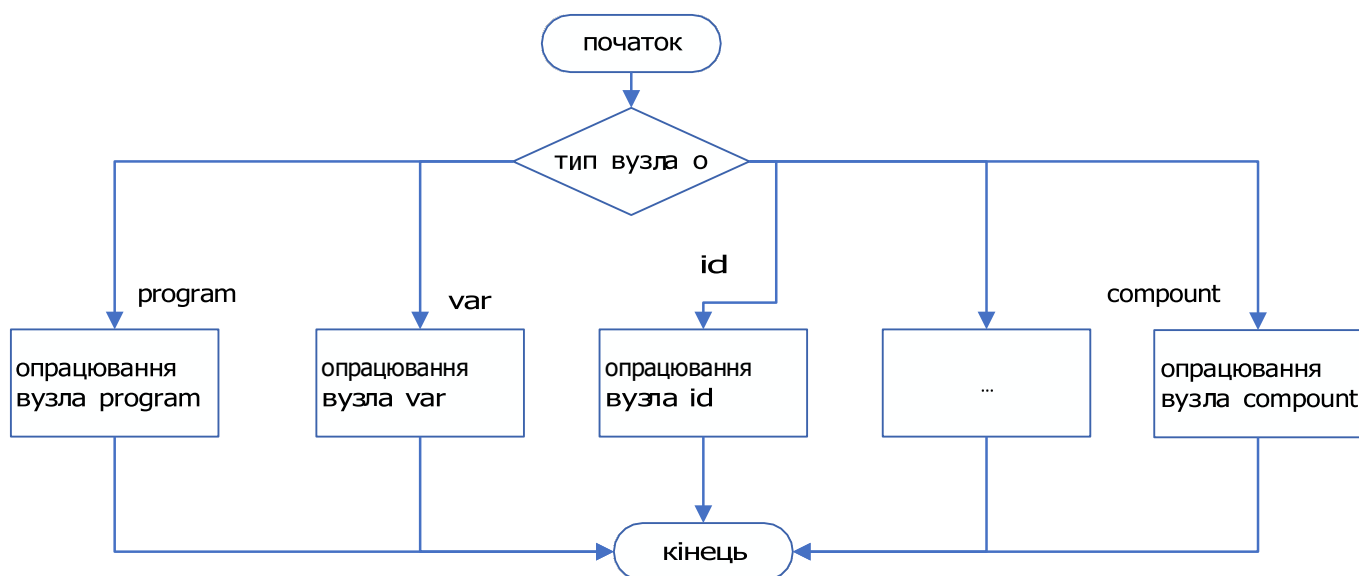


Рис. 3.6. Блок-схема алгоритму роботи генератора коду.

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

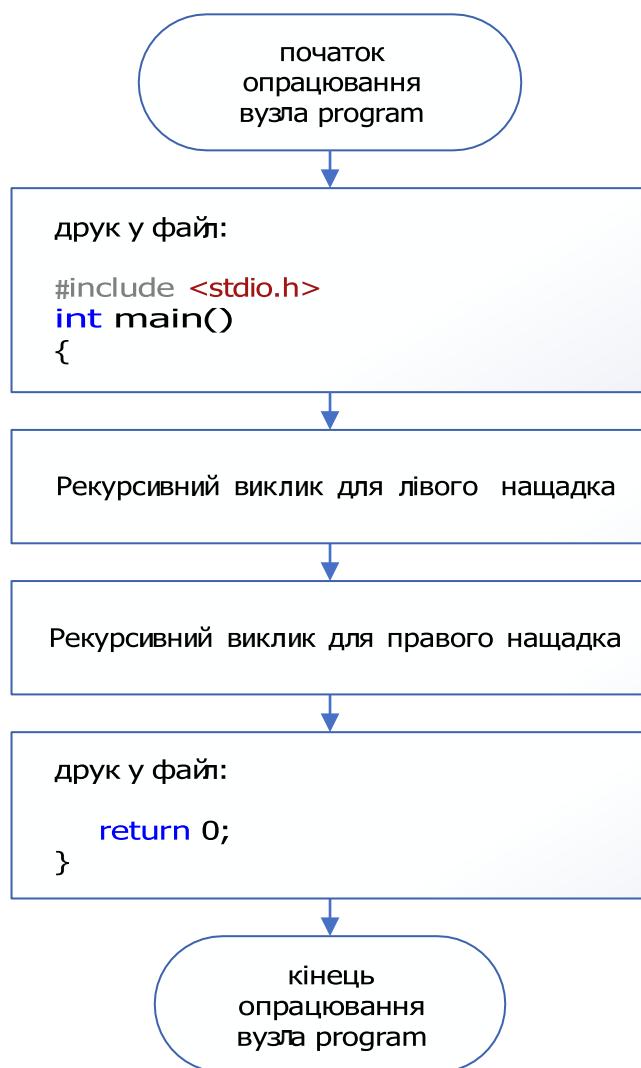


Рис. 3.7. Блок-схема алгоритму опрацювання вузла *program*.

2.5.2. Опис програми реалізації генератора коду.

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

1. Макроси та константи

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

2. Структури даних

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

3. Генерація коду

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

4. Маніпуляція з токенами

- **detectMultiToken:**
 - Перевіряє, чи відповідає поточна лексема багатокomпонентному токenu з таблиці tokenStruct.
- **createMultiToken:**
 - Створює багатокomпонентний токен і зберігає його у структурі LexemInfo.

5. Генерація машинного коду

- **outBytes2Code:**
 - Копіює байти з одного буфера до іншого, формуючи машинний код.
- **Пример генерації команд:**
 - **makeSaveHWStack:**
 - Генерує інструкцію mov ebp, esp для збереження стека.
 - **makeResetHWStack:**
 - Генерує інструкцію mov esp, ebp для відновлення стека.

Як працює генерація коду в функції `makeCode`

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

1. Ініціалізація

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

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

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

2. Ініціалізація стеку

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

```
lexemInfoTransformationTempStackSize = 0;
currBytePtr = makeInitCode(lastLexemInfoInTable, currBytePtr);
currBytePtr = initMake(lastLexemInfoInTable, currBytePtr);
currBytePtr = makeSaveHWStack(lastLexemInfoInTable, currBytePtr);
```

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

3. Обробка лексем у циклі

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

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

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

4. Генерація коду для конструкцій

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

Умовні оператори:

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

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

Цикли:

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

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

Операції та оператори:

```
ADD_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode,
NULL);
SUB_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode,
NULL);
MUL_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode,
NULL);
DIV_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode,
NULL);
MOD_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode,
NULL);
```

- Генерація арифметичних операцій (+, -, *, /, %).

Логічні оператори:

```
AND_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode,
NULL);
OR_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode,
NULL);
NOT_CODER(lastLexemInfoInTable_, lastLexemInfoInTable, currBytePtr, generatorMode,
NULL);
```

- Логічні оператори &&, ||, !.

Інші оператори:

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

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

5. Обробка помилок

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

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

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

6. Завершення програми

Після обробки всіх лексем функція генерує завершальні інструкції:

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

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

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

7. Виведення коду

Функція viewCode виводить згенерований код форматі (шістнадцяткові):

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

3. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА

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

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

3.1. Опис інтерфейсу та інструкції користувачу.

Розроблений транслятор має простий консольний інтерфейс.

При запуску програми обирається базовий файл fille1.z10:


```

Used default mode

Original source:
-----
program pPROGM ;
start var integer16 vALUEI , vALUEV , vALUEY ;
  scan (vALUEI)
  vALUEV <- 1
  if ( vALUEI <> 0 ) ; else goto cALUET ;
  print ( 1 )
  goto bALUET
cALUET :
  print ( 0 )
  !! ya dayn
  bALUET :
  print ( 100 add 2 sub 90)
  print ( 11 % 2)
  scan (vALUEI)
finish
-----

Source after comment removing:
-----
program pPROGM ;
start var integer16 vALUEI , vALUEV , vALUEY ;
  scan (vALUEI)
  vALUEV <- 1
  if ( vALUEI <> 0 ) ; else goto cALUET ;
  print ( 1 )
  goto bALUET
cALUET :
  print ( 0 )

  bALUET :
  print ( 100 add 2 sub 90)
  print ( 11 % 2)
  scan (vALUEI)
finish
-----

Lexemes table:
-----

```

index	lexeme	id	type	ifvalue	row	col
0	program	285	1	0	1	1
1	pPROGM	0	2	0	1	9
2	;	256	1	0	1	16
3	start	297	1	0	2	1
4	var	293	1	0	2	7

Рис. 4.2.Результати роботи розробленого транслятора.

3.2. Виявлення лексичних і синтаксичних помилок.

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

Наприклад, у програмі зробимо синтаксичну помилку:

```
Source after comment removing:
-----
program pPROGM ;
starr var integer16 vALUEI , vALUEV , vALUEY ;
  scan (vALUEI)
  vALUEV <- 1
  if ( vALUEI <> 0 ) ; else goto cALUET ;
  print ( 1 )
  goto bALUET
cALUET :
  print ( 0 )

  bALUET :
  print ( 100 add 2 sub 90)
  print ( 11 % 2)
  scan (vALUEI)
finish
-----

Lexical analysis detected unexpected lexeme
Bad lexeme:
-----
index      lexeme      id      type      ifvalue row    col
-----
89 0         starr       0       127       0    2      1
01
SI
Press Enter to exit . . .
```

Рис. 4.3. Вивід інформації про синтаксичну помилку.

Зробимо семантичну помилку – не оголосимо змінну “vALUEI”:

The screenshot shows the Visual Studio IDE with a C++ file named `one_file_x86_Debug.cpp` open. The code defines a program `pPROGM` with several labels and instructions. A semantic error is introduced by using the identifier `vALUEI` without a prior declaration. The debug window at the bottom shows the execution trace, including the error message: `declared identifier: vALUEI`.

```

1  program pPROGM ;
2  start var integer16  vALUEV , vALUEY ;
3  scan (vALUEI)
4  vALUEV <- 1
5  if ( vALUEI <> 0 ) ; else goto cALUET ;
6  print ( 1 )
7  goto bALUET
8  cALUET :
9  print ( 0 )
10 !! ya dayn
11 bALUET :
12 print ( 100 add 2 sub 90)
13 print ( 11 % 2)
14 scan (vALUEI)
15 finish
16

```

Microsoft Visual Studio Debug Console Output:

39)	283	1	0	9	13
40	bALUET	5	2	0	11	4
41	:	277	1	0	11	11
42	print	324	1	0	12	4
43	(280	1	0	12	10
44	100	320	4	100	12	12
45	add	415	1	0	12	16
46	2	320	4	2	12	20
47	sub	419	1	0	12	22
48	90	320	4	90	12	26
49)	283	1	0	12	28
50	print	324	1	0	13	6
51	(280	1	0	13	12
52	11	320	4	11	13	14
53	%	388	1	0	13	17
54	2	320	4	2	13	19
55)	283	1	0	13	20
56	scan	330	1	0	14	3
57	(280	1	0	14	8
58	vALUEI	3	2	0	14	9
59)	283	1	0	14	15
60	finish	303	1	0	15	1

```

kParse complete.....[   ok   ]: 60          finish
kAlgorithmImplementation return "true".
declared identifier: vALUEI

```

Рис. 4.4. Вивід інформації про семантичну помилку.

3.3. Перевірка роботи транслятора за допомогою тестових задач.

Тестова програма «Лінійний алгоритм»

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

2. Обрахувати значення виразу

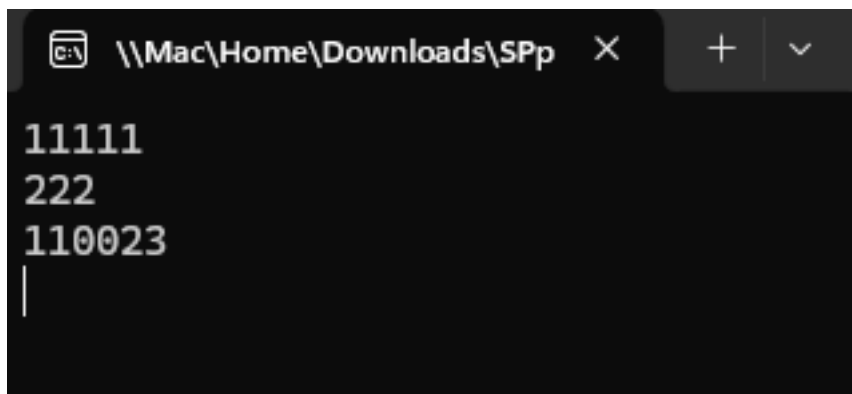
$$X = (A - B) * 10 + (A + B) / 10$$

3. Вивести значення X на екран.

Напишемо програму на вхідній мові програмування:

```
program pPROGM ;
start var integer16 aVVVVV , bVVVVV , xVVVVV ;
  scan (aVVVVV)
  scan (bVVVVV)
  xVVVVV <- 10 * (aVVVVV sub bVVVVV) add (aVVVVV add bVVVVV) / 10
  print ( xVVVVV )
  scan (aVVVVV)
finish
```

На мові Assembler протестуємо у новому проєкті у середовищі Visual Studio 2022 і отримаємо такі результати:

A screenshot of a terminal window with a dark background. The title bar at the top shows a file icon, the path '\\Mac\Home\Downloads\SPp', and window control buttons. The terminal content displays three lines of text: '11111', '222', and '110023'. A vertical cursor is positioned at the start of a fourth line below the last line of text.

```
11111
222
110023
|
```

Рис. 4.5. Результати виконання тестової задачі 1.

Тестова програма «Алгоритм з розгалуженням»

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

Використання вкладеного умовного оператора:

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

Використання простого умовного оператора:

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

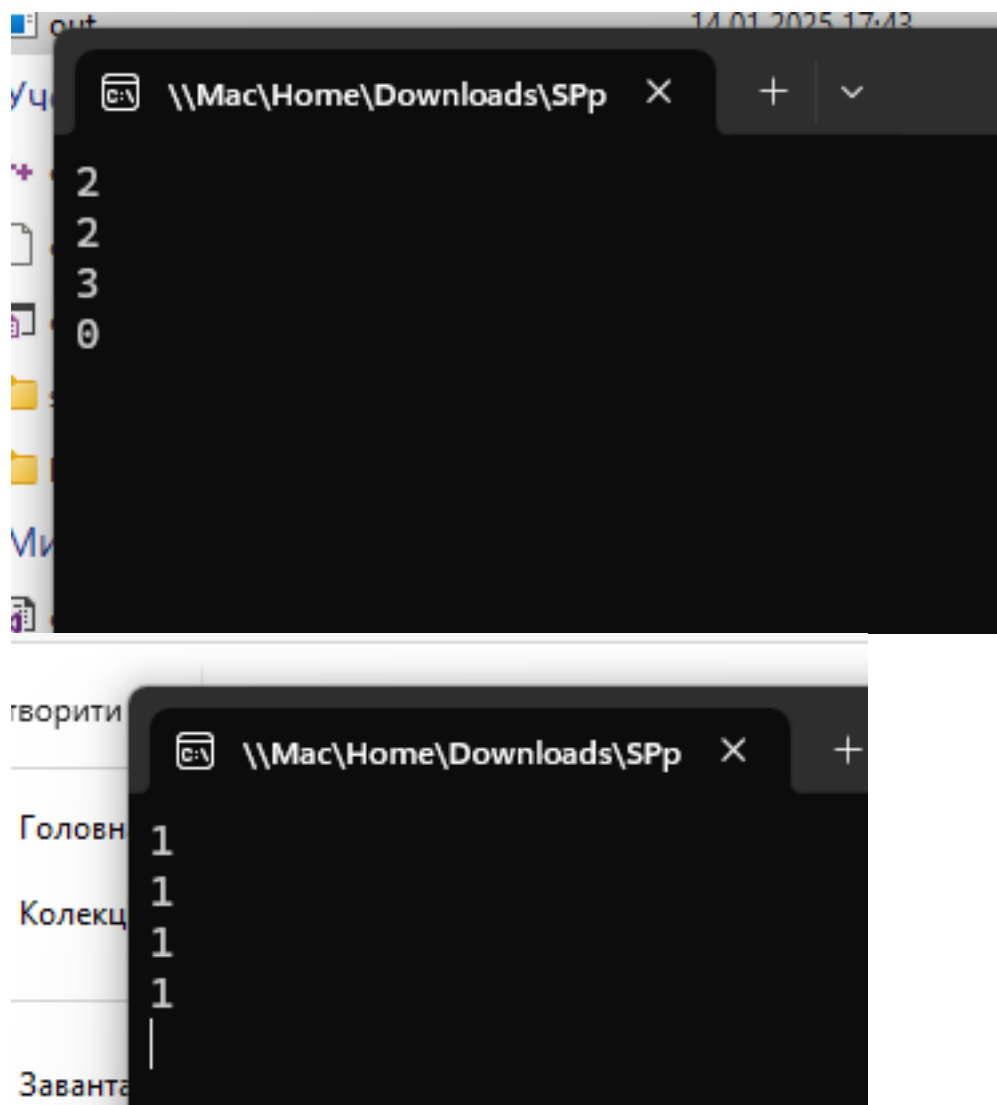
Напишемо програму на вхідній мові програмування:

```

program pPROGM ;
start var integer16 aVVVVV , bVVVVV , cVVVVV ;
scan (aVVVVV)
scan (bVVVVV)
scan (cVVVVV)
if ( aVVVVV = bVVVVV ) ; else goto cALUET ;
goto bALUET
cALUET :
print ( 0 )
goto eNASDF
bALUET :
if ( aVVVVV = cVVVVV ) ; else goto cALUET ;
print ( 1 )
eNASDF:
scan (aVVVVV)
finish

```

Отриманий код на мові assembler протестуємо у новому проекті у середовищі Visual Studio 2022 і отримаємо такі результати:



The image consists of two screenshots of a Visual Studio 2022 console window. The top screenshot shows the output of an assembly test, with the text '2', '2', '3', and '0' displayed on separate lines. The bottom screenshot shows the output of another assembly test, with the text '1', '1', '1', and '1' displayed on separate lines. Both screenshots show the file path '\\Mac\Home\Downloads\SPp' in the title bar of the console window.

```
2
2
3
0
```

```
1
1
1
1
```

Рис. 4.6. Результати виконання тестової задачі 2.

ВИСНОВКИ

В процесі виконання курсового проекту було виконано наступне:

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

2.Створено, а саме:

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

2.2.Розроблено синтаксичний аналізатор на основі низхідного методу. Складено деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура

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

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

3.1.На виявлення лексичних помилок.

3.2.На виявлення синтаксичних помилок.

3.3.Загальна перевірка роботи компілятора.

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

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

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

ДОДАТКИ

А. Таблиці лексем для тестових прикладів

Тестова програма «лінійного алгоритму»

Lexemes table:

index	lexeme	id	type	ifvalue	row	col
0	program	285	1	0	1	1
1	pPROGM	0	2	0	1	9
2	NULL	0	0	0	-1	-1
3	STATEMENT	0	0	0	-1	-1
4	;	256	1	0	1	16
5	start	297	1	0	2	1
6	var	293	1	0	2	7
7	integer16	405	1	0	2	11
8	aVVVVV	1	2	0	2	21
9	NULL	0	0	0	-1	-1
10	STATEMENT	0	0	0	-1	-1
11	,	270	1	0	2	28
12	bVVVVV	2	2	0	2	30
13	NULL	0	0	0	-1	-1
14	STATEMENT	0	0	0	-1	-1
15	,	270	1	0	2	37
16	xVVVVV	3	2	0	2	39
17	NULL	0	0	0	-1	-1
18	STATEMENT	0	0	0	-1	-1
19	;	256	1	0	2	46
20	4	0	4	4	-1	-1
21	scan	330	1	0	3	3
22	NULL	0	0	0	-1	-1
23	STATEMENT	0	0	0	-1	-1
24	8	0	4	8	-1	-1
25	scan	330	1	0	4	3
26	NULL	0	0	0	-1	-1
27	STATEMENT	0	0	0	-1	-1
28	12	0	4	12	-1	-1
29	10	320	4	10	5	13
30	aVVVVV	1	2	0	5	20
31	bVVVVV	2	2	0	5	31
32	sub	419	1	0	5	27
33	*	268	1	0	5	16
34	aVVVVV	1	2	0	5	44
35	bVVVVV	2	2	0	5	55
36	add	415	1	0	5	39
37	10	320	4	10	5	65
38	/	385	1	0	5	63
39	add	415	1	0	5	39
40	<=	258	1	0	5	10
41	NULL	0	0	0	-1	-1
42	STATEMENT	0	0	0	-1	-1
43	xVVVVV	3	2	0	6	10
44	print	324	1	0	6	2
45	NULL	0	0	0	-1	-1
46	STATEMENT	0	0	0	-1	-1
47	4	0	4	4	-1	-1
48	scan	330	1	0	7	3
49	NULL	0	0	0	-1	-1
50	STATEMENT	0	0	0	-1	-1
51	finish	303	1	0	8	1

Тестова програма «Алгоритм з розгалуженням»

Lexemes table:

index	lexeme	id	type	ifvalue	row	col
0	program	285	1	0	1	2
1	pPROGM	0	2	0	1	10
2	NULL	0	0	0	-1	-1
3	STATEMENT	0	0	0	-1	-1
4	;	256	1	0	1	17
5	start	297	1	0	2	1
6	var	293	1	0	2	7
7	integer16	405	1	0	2	11
8	aVVVVV	1	2	0	2	21
9	NULL	0	0	0	-1	-1
10	STATEMENT	0	0	0	-1	-1
11	,	270	1	0	2	28
12	bVVVVV	2	2	0	2	30
13	NULL	0	0	0	-1	-1
14	STATEMENT	0	0	0	-1	-1
15	,	270	1	0	2	37
16	cVVVVV	3	2	0	2	39
17	NULL	0	0	0	-1	-1
18	STATEMENT	0	0	0	-1	-1
19	;	256	1	0	2	46
20	4	0	4	4	-1	-1
21	scan	330	1	0	3	3
22	NULL	0	0	0	-1	-1
23	STATEMENT	0	0	0	-1	-1
24	8	0	4	8	-1	-1
25	scan	330	1	0	4	3
26	NULL	0	0	0	-1	-1
27	STATEMENT	0	0	0	-1	-1
28	12	0	4	12	-1	-1
29	scan	330	1	0	5	4
30	NULL	0	0	0	-1	-1
31	STATEMENT	0	0	0	-1	-1
32	if	335	1	0	6	5
33	aVVVVV	1	2	0	6	10
34	bVVVVV	2	2	0	6	19
35	=	272	1	0	6	17
36	NULL	0	0	0	-1	-1
37	STATEMENT	0	0	0	-1	-1
38	;	256	1	0	6	28
39	else	338	1	0	6	30
40	goto	379	1	0	6	35
41	cALUET	4	2	0	6	40
42	NULL	0	0	0	-1	-1
43	STATEMENT	0	0	0	-1	-1
44	;	256	1	0	6	47
45	goto	379	1	0	6	35
46	bALUET	5	2	0	7	12
47	NULL	0	0	0	-1	-1
48	STATEMENT	0	0	0	-1	-1
49	cALUET	4	2	0	8	2
50	NULL	0	0	0	-1	-1
51	STATEMENT	0	0	0	-1	-1
52	:	277	1	0	8	9
53	0	320	4	0	9	11
54	print	324	1	0	9	3
55	NULL	0	0	0	-1	-1
56	STATEMENT	0	0	0	-1	-1
57	goto	379	1	0	10	3
58	eNASDF	6	2	0	10	8
59	NULL	0	0	0	-1	-1
60	STATEMENT	0	0	0	-1	-1
61	bALUET	5	2	0	11	4
62	NULL	0	0	0	-1	-1
63	STATEMENT	0	0	0	-1	-1
64	:	277	1	0	11	11
65	if	335	1	0	12	6
66	aVVVVV	1	2	0	12	11
67	cVVVVV	3	2	0	12	20
68	=	272	1	0	12	18
69	NULL	0	0	0	-1	-1
70	STATEMENT	0	0	0	-1	-1
71	;	256	1	0	12	29
72	else	338	1	0	12	31
73	goto	379	1	0	12	36
74	cALUET	4	2	0	12	41
75	NULL	0	0	0	-1	-1
76	STATEMENT	0	0	0	-1	-1
77	;	256	1	0	12	48
78	1	320	4	1	13	15
79	print	324	1	0	13	7
80	NULL	0	0	0	-1	-1
81	STATEMENT	0	0	0	-1	-1
82	eNASDF	6	2	0	14	1
83	NULL	0	0	0	-1	-1
84	STATEMENT	0	0	0	-1	-1

В. С код (або код на асемблері), отриманий на виході транслятора для тестових прикладів
Тестова програма «Лінійний алгоритм»

```
.model flat,stdcall
```

```
option casemap:none
```

```
GetStdHandle proto STDCALL,nStdHandle:DWORD
```

```
ExitProcess proto STDCALL,uExitCode:DWORD
```

```
ReadConsoleA proto
```

```
STDCALL,hConsoleInput:DWORD,lpBuffer:DWORD,nNumberOfCharsToRead:DWORD,  
D,lpNumberOfCharsRead:DWORD,lpReserved:DWORD
```

```
WriteConsoleA proto
```

```
STDCALL,hConsoleOutput:DWORD,lpBuffert:DWORD,nNumberOfCharsToWrite:DWOR  
RD,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)
```

```
valueTemp_msg db 256 dup(0)
```

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

```
hConsoleInput dd 0
```

```
hConsoleOutput dd 0

buffer db 128 dup(0)

readOutCount dd ?

.code

start:

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

push 0

push eax

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
```

```
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

mov ebp,esp

mov eax,edi

add eax,000000000h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,edi

add eax,000000004h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,edi


```
add eax,000000008h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

mov eax,edi

add eax,00000000Ch

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

add ecx,4

mov eax,000000004h

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx]

mov edx,000000044h

add edx,esi

push ecx

push esi

push edi

call edx

pop edi
```

```
pop esi

mov ebx,dword ptr[ecx]

sub ecx,4

add ebx,edi

mov dword ptr[ebx],eax

mov ecx,edi

add ecx,512

add ecx,4

mov eax,000000008h

mov dword ptr[ecx],eax

mov eax,dword ptr[ecx]

mov edx,000000044h

add edx,esi

push ecx

push esi

push edi

call edx

pop edi

pop esi

mov ebx,dword ptr[ecx]
```

```
sub ecx,4

add ebx,edi

mov dword ptr[ebx],eax

mov ecx,edi

add ecx,512

add ecx,4

mov eax,00000000Ch

mov dword ptr[ecx],eax

add ecx,4

mov eax,00000000Ah

mov dword ptr[ecx],eax

mov eax,edi

add eax,000000004h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

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]

imul dword ptr[ecx]

sub ecx,4

mov dword ptr[ecx],eax

mov eax,edi

add eax,000000004h

mov eax,dword ptr[eax]

add ecx,4

mov dword ptr[ecx],eax

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]

add ecx,4

mov eax,00000000Ah

mov dword ptr[ecx],eax

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  
  
add ecx,512  
  
mov eax,edi  
  
add eax,00000000Ch  
  
mov eax,dword ptr[eax]  
  
add ecx,4  
  
mov dword ptr[ecx],eax  
  
mov eax,dword ptr[ecx]  
  
mov edx,00000001Bh  
  
add edx,esi  
  
push esi  
  
push edi  
  
call edx  
  
pop edi  
  
pop esi  
  
mov ecx,edi  
  
add ecx,512  
  
add ecx,4  
  
mov eax,000000004h  
  
mov dword ptr[ecx],eax
```

```
mov eax,dword ptr[ecx]

mov edx,000000044h

add edx,esi

push ecx

push esi

push edi

call edx

pop edi

pop esi

mov ebx,dword ptr[ecx]

sub ecx,4

add ebx,edi

mov dword ptr[ebx],eax

mov ecx,edi

add ecx,512

mov esp,ebp

xor eax,eax

ret

end starta
```

Тестова програма «Алгоритм з розгалуженням»

Код на мові assembler:

.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

;"pPROGM"
mov eax, edi
add eax, 000000000h
mov eax, dword ptr[eax]
add ecx, 4
mov dword ptr [ecx], eax

;null statement (non-context)

```

```
;"
```

```
;"aVVVVV"
```

```
mov eax, edi
```

```
add eax, 000000004h
```

```
mov eax, dword ptr[eax]
```

```
add ecx, 4
```

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

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

```
;"bVVVVV"
```

```
mov eax, edi
```

```
add eax, 000000008h
```

```
mov eax, dword ptr[eax]
```

```
add ecx, 4
```

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

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

```
;"cVVVVV"
```

```
mov eax, edi
```

```
add eax, 00000000Ch
```

```
mov eax, dword ptr[eax]
```

```
add ecx, 4
```

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

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

```
;"
```

```
;"4"
```

```
add ecx, 4
```

```
mov eax, 000000004h
```

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

```
;"scan"
```

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

```

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

```

```

;null statement (non-context)

```

```

;"8"
add ecx, 4
mov eax, 000000008h
mov dword ptr [ecx], eax

```

```

;"scan"
mov eax, dword ptr[ecx]
mov edx, 000000044h
add edx, esi
push ecx
;push ebx
push esi
push edi
call edx
pop edi
pop esi
;pop ebx
pop ecx

```

```

mov ebx, dword ptr[ecx]
sub ecx, 4
add ebx, edi
mov dword ptr [ebx], eax
mov ecx, edi ; reset second stack
add ecx, 512 ; reset second stack

```

```

;null statement (non-context)

```

```

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

```

```

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

```

```

;null statement (non-context)

```

```

;"if"

```

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

```
; "bVVVVV"
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_000000002193DDC8
```

```
; ";" (after "then"-part of if-operator)
mov eax, 1
LABEL@AFTER_THEN_000000002193DDC8:
```

```
; "else"
cmp eax, 0
jnz LABEL@AFTER_ELSE_000000002193EA40
```

```
; "goto" previous ident "cALUET"(as label)
jmp LABEL@00000000223C7B50
```

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

;";" (after "else")

LABEL@AFTER_ELSE_000000002193EA40:

;"goto" previous ident "bALUET"(as label)

jmp LABEL@00000000223C0E08

;null statement (non-context)

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

LABEL@00000000223C7B50:

;"0"

add ecx, 4

mov eax, 00000000h

mov dword ptr [ecx], eax

;"print"

mov eax, dword ptr[ecx]

mov edx, 00000001Bh

add edx, esi

;push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

;pop ecx

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"goto" previous ident "eNASDF"(as label)

jmp LABEL@00000000223C0E58

;null statement (non-context)

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

LABEL@00000000223C0E08:

;"if"

;"aVVVVV"

mov eax, edi

add eax, 000000004h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"cVVVVV"

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_00000000219466F0

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

mov eax, 1

LABEL@AFTER_THEN_00000000219466F0:

;"else"

```
cmp eax, 0
jnz LABEL@AFTER_ELSE_0000000021947368
```

```
;"goto" previous ident "cALUET"(as label)
jmp LABEL@00000000223C7B50
```

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

```
;";" (after "else")
LABEL@AFTER_ELSE_0000000021947368:
```

```
;"1"
add ecx, 4
mov eax, 000000001h
mov dword ptr [ecx], eax
```

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

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

```
;ident "eNASDF"(as label) previous ":"
LABEL@00000000223C0E58:
```

```
;"4"
```

```

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

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

;null statement (non-context)

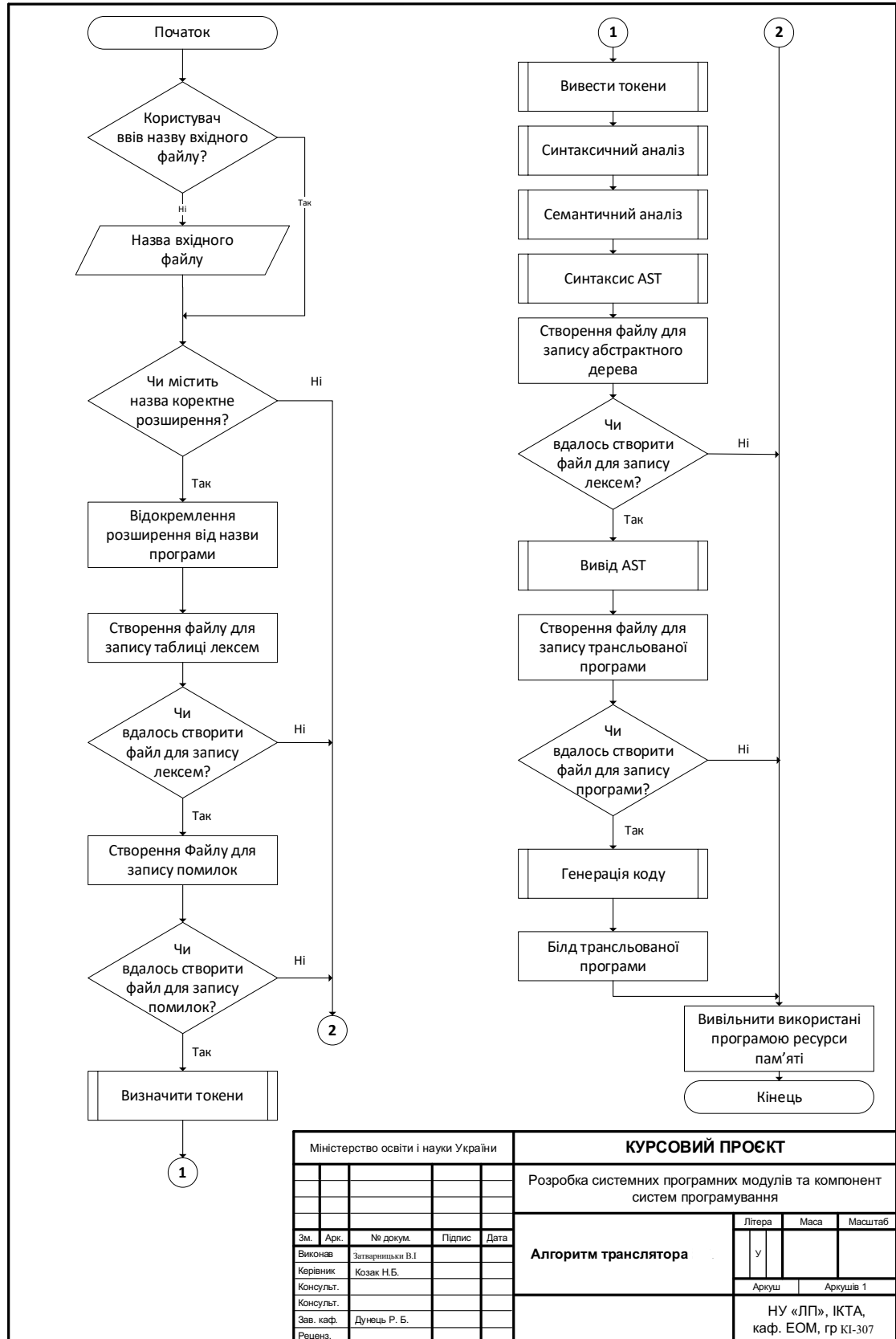
;hw stack reset(restore esp)
mov esp, ebp

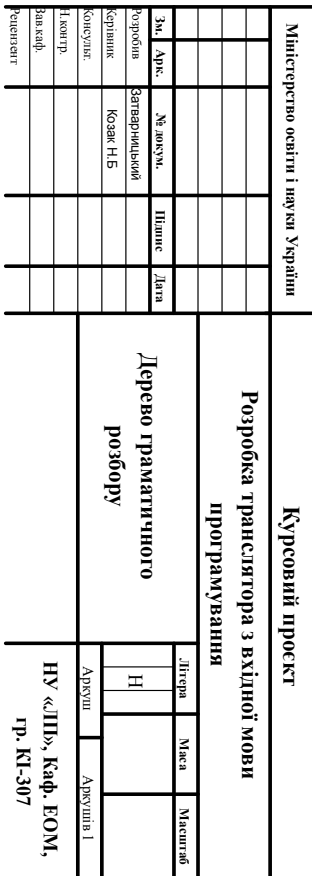
xor eax, eax
ret

```

end start

В. Креслення алгоритму транслятора





D. Лістинг програми

cli.cpp

#define _CRT_SECURE_NO_WARNINGS

/*****

* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *

* 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"

//#define DEFAULT_INPUT_FILENAME "file44.cwl"

//

//#define PREDEFINED_TEXT \

// "name MN\r\n" \

// "data\r\n" \

// " /*argumentValue*#\r\n" \

// " 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" \\

unsigned int mode = 0;
char parameters[PARAMETERS_COUNT][MAX_PARAMETERS_SIZE] = { "" };

void comandLineParser(int argc, char* argv[], unsigned int* mode,
char(*parameters)[MAX_PARAMETERS_SIZE]) {
    char useDefaultModes = 1;
    *mode = 0;
    for (int index = 1; index < argc; ++index) {
        if (!strcmp(argv[index], "-lex")) {
            *mode |= LEXICAL_ANALISIS_MODE;
            useDefaultModes = 0;
            continue;
        }
        else if (!strcmp(argv[index], "-d")) {
            *mode |= DEBUG_MODE;
            useDefaultModes = 0;
            continue;
        }

        // other keys
        // TODO:...

        // input file name

        strncpy(parameters[INPUT_FILENAME_PARAMETER],
argv[index], MAX_PARAMETERS_SIZE);
    }

    // default input file name, if not entered manually
    if (parameters[INPUT_FILENAME_PARAMETER][0] ==
'\0') {

        strcpy(parameters[INPUT_FILENAME_PARAMETER],
DEFAULT_INPUT_FILENAME);

        printf("Input file name not setted. Used default input
file name \\\"file1.z10\\\"\\r\\n\\r\\n");
    }

    // default mode, if not entered manually
    if (useDefaultModes) {
        *mode = DEFAULT_MODE;
        printf("Used default mode\\r\\n\\r\\n");
    }
}

```

```

        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;
}
Add.cpp

#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           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);

#ifdef 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 // lex + rpn + MACHINECODEGEN! *
*          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 };

```

```

0xC0 };
0xE0, 0x01 };
0xE9, 0x04 };

= { 0x83, 0x39, 0x00 };
0xC2 };
0xE2, 0x01 };
0xC2 };

code__mov_stackTopByECX_eax[] = { 0x89, 0x01 };

char*)code__mov_eax_stackTopByECX, 2);
char*)code__cmp_eax_0, 3);
char*)code__setne_al, 3);
char*)code__and_eax_1, 3);
char*)code__sub_ecx_4, 3);

char*)code__cmp_stackTopByECX_0, 3);
char*)code__setne_dl, 3);
char*)code__and_edx_1, 3);

char*)code__and_eax_edx, 2);

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");

```

```

const unsigned char code__setne_al[] = { 0x0F, 0x95,
const unsigned char code__and_eax_1[] = { 0x83,
const unsigned char code__sub_ecx_4[] = { 0x83,
//
const unsigned char code__cmp_stackTopByECX_0[]
const unsigned char code__setne_dl[] = { 0x0F, 0x95,
const unsigned char code__and_edx_1[] = { 0x83,
//
const unsigned char code__and_eax_edx[] = { 0x23,
//
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;
}
}
Bitwise_and.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           file: bitwise_and.cpp           *
*           (draft!) *
*****/

#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"
#include "stdio.h"

unsigned char* makeBitwiseAndCode(struct LexemInfo** lastLexemInfoInTable, unsigned
char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize =
detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_BITWISE_AND);
    if (multitokenSize) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("\r\n");
        printf("    ;\"%s\"\r\n",
tokenStruct[MULTI_TOKEN_BITWISE_AND][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__and_stackTopByECX_eax[] = { 0x21, 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__and_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("  and dword ptr[ecx], eax\r\n");
                                printf("  mov eax, dword ptr[ecx]\r\n");
#endif

                                return *lastLexemInfoInTable += multitokenSize,
currBytePtr;
                                }

                                return currBytePtr;
}
}
Bitwise_not.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           file: bitwise_not.cpp           *
*           (draft!) *
*****/

#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"
#include "stdio.h"

unsigned char* makeBitwiseNotCode(struct LexemInfo** lastLexemInfoInTable, unsigned
char* currBytePtr, unsigned char generatorMode) {
                                unsigned char multitokenSize =
detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_BITWISE_NOT);
                                if (multitokenSize) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
                                printf("\r\n");

```

```

                                printf("    ;\"%s\"\\r\\n",
tokenStruct[MULTI_TOKEN_BITWISE_NOT][0]);
#endif

                                const unsigned char code__not_stackTopByECX[] = {
0xF7, 0x11 };

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

                                currBytePtr = outBytes2Code(currBytePtr, (unsigned
char*)code__not_stackTopByECX, 2);
                                currBytePtr = outBytes2Code(currBytePtr, (unsigned
char*)code__mov_eax_stackTopByECX, 3);

#ifdef DEBUG_MODE_BY_ASSEMBLY
                                printf("    not dword ptr[ecx]\\r\\n");
                                printf("    mov eax, dword ptr[ecx]\\r\\n");
#endif

                                return *lastLexemInfoInTable += multitokenSize,
currBytePtr;

                                }

                                return currBytePtr;
}
}
Bitwise_or.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           file: bitwise_or.cpp           *
*           (draft!) *
*****/

#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"
#include "stdio.h"

unsigned char* makeBitwiseOrCode(struct LexemInfo** lastLexemInfoInTable, unsigned
char* currBytePtr, unsigned char generatorMode) {
                                unsigned char multitokenSize =
detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_BITWISE_OR);
                                if (multitokenSize) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
                                printf("\\r\\n");
                                printf("    ;\"%s\"\\r\\n",
tokenStruct[MULTI_TOKEN_BITWISE_OR][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__or_stackTopByECX_eax[]
= { 0x09, 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__or_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("  aor dword ptr[ecx], eax\r\n");
                                printf("  mov eax, dword ptr[ecx]\r\n");

```

```
#endif
```

```

                                return *lastLexemInfoInTable += multitokenSize,
currBytePtr;
                                }

```

```
                                return currBytePtr;
```

```

}
div.cpp

```

```
#define _CRT_SECURE_NO_WARNINGS
```

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

```
* N.Kozak // Lviv'2024 // lex + rpн + MACHINECODEGEN! *
```

```
*          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("    ;\n\"%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 // lex + rpn + MACHINECODEGEN! *
*           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[lexemInfoTransforma
tionTempStackSize++] = **lastLexemInfoInTable;

    lexemInfoTransformationTempStack[lexemInfoTransforma
tionTempStackSize - 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
&&

                                !strcmp(lexemInfoTransformationTempStack[lexemInfoT
ransformationTempStackSize - 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 // lex + rpn + MACHINECODEGEN! *
*          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 // lex + rpn + MACHINECODEGEN! *
*          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("    ;\"%s\"\\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_DWNTTO);
        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_DOWNT0][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[lexemInfoTransforma
tionTempStackSize++] = **lastLexemInfoInTable;

                                lexemInfoTransformationTempStack[lexemInfoTransforma
tionTempStackSize - 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("    ;\"%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[lexemInfoTransforma
tionTempStackSize - 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[lexemInfoTransforma
tionTempStackSize - 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[lexemInfoT
ransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI_TOKEN_FOR][0],
MAX_LEXEM_SIZE)

        && (

        !strcmp(lexemInfoTransformationTempStack[lexemInfoT
ransformationTempStackSize - 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 // lex + rpn + MACHINECODEGEN! *
*                                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 <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_E
LEMENT_PART_COUNT] = { NULL };

#if 0
static void intitTokenStruct__OLD() {
    // SET_QUADRUPLE_STR_MACRO_IN_ARRAY(token
Struct, 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*)">";

```

```

(char*)"<";
0] = (char*)">";

tokenStruct[MULTI_TOKEN_LESS_OR_EQUAL][0] =
tokenStruct[MULTI_TOKEN_GREATER_OR_EQUAL][

tokenStruct[MULTI_TOKEN_ADD][0] = (char*)"add";
tokenStruct[MULTI_TOKEN_SUB][0] = (char*)"sub";
tokenStruct[MULTI_TOKEN_MUL][0] = (char*)"*";
tokenStruct[MULTI_TOKEN_DIV][0] = (char*)" / ";
tokenStruct[MULTI_TOKEN_MOD][0] = (char*)" % ";

tokenStruct[MULTI_TOKEN_BIND_RIGHT_TO_LEFT][
0] = (char*)"<-";
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_DOWNTO][0] =
(char*)"downto";

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*)"finish"; 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*)"scan";
tokenStruct[MULTI_TOKEN_OUTPUT][0] =
(char*)"print";

//

//
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*)"finish";

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 (!strncmp(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') {
        strncpy(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') {
            strncpy((*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;
}
//

//0x20 , 0x20 , 0x20 , 0x20 , 0x20 , 0x20 , 0x20 , 0x20 ,
0x20 , 0x20 , 0x20 , 0x20 , 0x20 , 0x20 , 0x20 , 0x20 ,
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");

#ifdef

        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");
#endif

        return currBytePtr;

}

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

#ifdef 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");
#endif

        return currBytePtr;
}

unsigned char* makeDataSection(struct LexemInfo** lastLexemInfoInTable, unsigned char*
currBytePtr) {
#ifdef 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");
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/bitwise_not.h"
```

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

```
#include "../src/include/generator/bitwise_or.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.h"
#include "../src/include/generator/greater.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/lrbind.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;
//    for (; (*lastLexemInfoInTable)->lexemStr[0] &&
strncmp((*lastLexemInfoInTable)->lexemStr, "start", MAX_LEXEM_SIZE); ++ *
lastLexemInfoInTable);
//    for (; (*lastLexemInfoInTable)->lexemStr[0] &&
strncmp((*lastLexemInfoInTable)->lexemStr, ";", MAX_LEXEM_SIZE); ++ *
lastLexemInfoInTable);
//    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* getCodeBytePtr(unsigned char* baseBytePtr) {

    return baseBytePtr + baseOperationOffset;

}

void makeCode(struct LexemInfo** lastLexemInfoInTable/*TODO:...*/, unsigned char*
currBytePtr) { // 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_LABELE_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);
                                BITWISE_NOT_CODER(lastLexemInfoInTable_,
lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
                                BITWISE_AND_CODER(lastLexemInfoInTable_,
lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
                                BITWISE_OR_CODER(lastLexemInfoInTable_,
lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
                                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_CODER(lastLexemInfoInTable_,
lastLexemInfoInTable, currBytePtr, generatorMode, NULL);
        GREATER_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);
        LRBIND_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(lastLexemInfo
InTable_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

        NON_CONTEXT_NULL_STATEMENT(lastLexemInfoIn
Table_, 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);
}

unsigned char outCode[GENERATED_TEXT_SIZE] = { '0' };
void viewCode(unsigned char* outCodePtr, size_t outCodePrintSize, unsigned char align) {
    printf("\r\n;      +0x0 +0x1 +0x2 +0x3 +0x4 +0x5 +0x6
+0x7 +0x8 +0x9 +0xA +0xB +0xC +0xD +0xE +0xF ");
    printf("\r\n;0x00000000: ");
    size_t outCodePrintIndex = outCodePrintSize - 1;
    for (size_t index = 0; index <= outCodePrintIndex;) {

```

```

        printf("0x%02X ", outCodePtr[index]);
        if (!(++index % align)) {
            size_t indexMinus16 = index - align;
            do {
                //printf("0x%02X ", outCodePtr[index]);
                if (outCodePtr[indexMinus16] >= 32 &&
outCodePtr[indexMinus16] <= 126) {
                    printf("%c",
outCodePtr[indexMinus16]);
                }
                else {
                    printf(" ");
                    //printf("%02c", 32);
                }
            } while (++indexMinus16 % align);

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

```

Goto_label.cpp

```
#define _CRT_SECURE_NO_WARNINGS
```

```

/*****

```

```

* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *

```

```

*           file: goto_label.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@%016llx:\\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].seco
nd.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.cpp

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024-2025 // lex + rpn + MACHINECODEGEN! *
*           file: greater.cpp           *
*           (draft!) *
*****/

#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"
#include "stdio.h"

unsigned char* makeIsGreaterCode(struct LexemInfo** lastLexemInfoInTable, unsigned
char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize =
detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_GREATER);
    if (multitokenSize) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("\r\n");
        printf("    ;\"%s\"\r\n",
tokenStruct[MULTI_TOKEN_GREATER][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__setg_al[] = { 0x0F, 0x9F,
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__setg_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("    setg al\r\n");
    printf("    and eax, 1\r\n");
    printf("    mov dword ptr[ecx], eax\r\n");
#endif

return *lastLexemInfoInTable += multitokenSize,
currBytePtr;
}

return currBytePtr;
}
}
Greater_or_equal.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*          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("    ;\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 // lex + rpn + MACHINECODEGEN! *
*           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] == '('
        && !strncmp((*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[lexemInfoTransforma
tionTempStackSize++] = **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
        &&
!strncmp(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[lexemInfoTransforma
tionTempStackSize++] = **lastLexemInfoInTable;

    strncpy(lexemInfoTransformationTempStack[lexemInfoTra
nsformationTempStackSize - 1].lexemStr, tokenStruct[MULTI_TOKEN_THEN][0],
MAX_LEXEM_SIZE);

    lexemInfoTransformationTempStack[lexemInfoTransforma
tionTempStackSize - 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_%016lX:\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 // lex + rp + MACHINECODEGEN! *
*           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_valueByAdressInECX[] = { 0x8B, 0x19 };
        const unsigned char code__sub_ecx_4[] = { 0x83,
0xE9, 0x04 };

```



```

0xDF };
const unsigned char code __add_ebx_edi[] = { 0x33,
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_valueByAdrrssInECX, 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, 0%08Xh\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.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rp_n + MACHINECODEGEN! *
*          file: less.cpp          *
*          (draft!) *
*****/

#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"
#include "stdio.h"

unsigned char* makeIsLessCode(struct LexemInfo** lastLexemInfoInTable, unsigned char*
currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize =
detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_LESS);
    if (multitokenSize) {

```

```

#ifdef DEBUG_MODE_BY_ASSEMBLY
    printf("\r\n");
    printf("    ;\n\"%s\n\"\\r\n",
tokenStruct[MULTI_TOKEN_LESS][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__setl_al[] = { 0x0F, 0x9C,
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__setl_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("    setl al\\r\n");
    printf("    and eax, 1\\r\n");
    printf("    mov dword ptr[ecx], eax\\r\n");
#endif

    return *lastLexemInfoInTable += multitokenSize,
currBytePtr;
}

return currBytePtr;
}

```

Less_or_equal.cpp

```
#define _CRT_SECURE_NO_WARNINGS
```

```
/*
*****

```

```
* N.Kozak // Lviv'2024 // lex + rp_n + MACHINECODEGEN! *
```

```
* 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;
}
Lrbind.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lrbind codegen
* file: lrbind.cpp
* (draft!)
*****/

#include "../include/def.h"
#include "../include/generator/generator.h"
#include "../include/lexica/lexica.h"
#include "stdio.h"

unsigned char* makeLeftToRightBindCode(struct LexemInfo** lastLexemInfoInTable,
unsigned char* currBytePtr, unsigned char generatorMode) {
    unsigned char multitokenSize =
detectMultiToken(*lastLexemInfoInTable, MULTI_TOKEN_LRBIND);
    if (multitokenSize) {
#ifdef DEBUG_MODE_BY_ASSEMBLY
        printf("\r\n");
        printf("    ;\r\n",
tokenStruct[MULTI_TOKEN_LRBIND][0]);
#endif

const unsigned char
code__mov_ebx_stackTopByECX[] = { 0x8B, 0x19 };
const unsigned char
code__mov_eax_stackTopByECXMinus4[] = { 0x8B, 0x41, 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_ebx_stackTopByECX, 2);
currBytePtr = outBytes2Code(currBytePtr, (unsigned
char*)code__mov_eax_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 ebx, dword ptr[ecx]\r\n");
printf("  mov eax, 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;
}
Mod.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*                               *
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

        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 // lex + rpn + MACHINECODEGEN! *
*          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 // lex + rpn + MACHINECODEGEN! *
*          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 // lex + rpn + MACHINECODEGEN! *
```

```
* 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
```

```
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_statement.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rp_n + MACHINECODEGEN! *
*           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;
}

```

Operand.cpp

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rp_n + MACHINECODEGEN! *
*           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 (!strncmp((*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_edx[] = { 0x8B, 0xC7
};

    unsigned char
code__add_eax_variableOffsetInDataSection[] = { 0x05, 0x00, 0x00, 0x00, 0x00 };

```

```

                                const unsigned char
code__mov_eax_valueByAdrrssInEAX[] = { 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_valueByAdrrssInEAX, 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 // lex + rpn + MACHINECODEGEN! *
*
*           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);

```



```

char*)code__setne_al, 3);
char*)code__and_eax_1, 3);
char*)code__sub_ecx_4, 3);
char*)code__cmp_stackTopByECX_0, 3);
char*)code__setne_dl, 3);
char*)code__and_edx_1, 3);
char*)code__or_eax_edx, 2);
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 // lex + rpn + MACHINECODEGEN! *
*                               *
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;

```

```

char*)code__add_edx_esi, 2);
char*)code__push_ecx, 1);
char*)code__push_ebx, 1);
char*)code__push_esi, 1);
char*)code__push_edi, 1);
char*)code__call_edx, 2);
char*)code__pop_edi, 1);
char*)code__pop_esi, 1);
char*)code__pop_ebx, 1);
char*)code__pop_ecx, 1);
char*)code__mov_ecx_edi, 2);
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;
    }
Repeat_until.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           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("    ;\"%0s\"\\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_%016lX\r\n",
(unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
2].lexemStr);
#endif

                                lexemInfoTransformationTempStackSize -= 2;
                                return *lastLexemInfoInTable += multitokenSize,

currBytePtr;

                                }

                                return currBytePtr;
}
Rlbind.cpp

#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lrbind codegen
*
* file: rlbind.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;
}

```

Semicolon.cpp

```

#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           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("    ;\"%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("    ;\"%s\"\r\n", ";");
#endif

            * lastLexemInfoInTable += multitokenSize;
        }

        return currBytePtr;
    }
}
Sub.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*          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;
}
While.cpp
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*          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_%016llx:\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
            ||
strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
2].lexemStr, tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)
            ||
strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
1].lexemStr, tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)
            || lexemInfoTransformationTempStackSize >= 4
        &&
!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
4].lexemStr, tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)
            || lexemInfoTransformationTempStackSize >= 3
        &&
!strncmp(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 expresion (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[lexemInfoTransforma
tionTempStackSize - 1].ifvalue = (unsigned long long int)(currBytePtr - 4);

                                //lexemInfoTransformationTempStack[lexemInfoTransfor
mationTempStackSize++] =
lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1];

                                strncpy(lexemInfoTransformationTempStack[lexemInfoTra
nsformationTempStackSize - 1].lexemStr,
lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++ -
1].lexemStr, MAX_LEXEM_SIZE);

                                //lexemInfoTransformationTempStack[lexemInfoTransfor
mationTempStackSize++] =
lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1];

                                strncpy(lexemInfoTransformationTempStack[lexemInfoTra
nsformationTempStackSize - 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);

            strncpy(lexemInfoTransformationTempStack[lexemInfoTra

```

```

nsformationTempStackSize - 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
        &&
!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
1].lexemStr, tokenStruct[MULTI_TOKEN_ELSE][0], MAX_LEXEM_SIZE)
        &&
!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
4].lexemStr, tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)
        &&
!strncmp(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[lexemInfoTra

```

```

nsformationTempStackSize - 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
&&
!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
3].lexemStr, tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)
&&
!strncmp(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[lexemInfoTransfor
mationTempStackSize - 2].ifvalue = (unsigned long long int)(currBytePtr - 4);
                                *((unsigned int*)(currBytePtr - 4)) = (unsigned
int)((unsigned
char*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
4].ifvalue - currBytePtr);

                                strncpy(lexemInfoTransformationTempStack[lexemInfoTra
nsformationTempStackSize - 2].lexemStr,
tokenStruct[MULTI_TOKEN_CONTINUE_WHILE][0], MAX_LEXEM_SIZE);

#ifdef DEBUG_MODE_BY_ASSEMBLY

```

```

                                                                    printf("    jmp LABEL@WHILE_%016llx\r\n",
(unsigned long long
int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
4].lexemStr);
#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);

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr, tokenStruct[MULTI_TOKEN_EXIT_WHILE][0],
MAX_LEXEM_SIZE);

#ifdef DEBUG_MODE_BY_ASSEMBLY
printf("  jmp
LABEL@AFTER_WHILE_%016llx\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);

strncpy(lexemInfoTransformationTempStack[lexemInfoTra

```

```

nsformationTempStackSize - 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
&&
!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
3].lexemStr, tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)
&&
!strncmp(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[lexemInfoTransforma
tionTempStackSize - 1].ifvalue = (unsigned long long int)(currBytePtr - 4);

                                strncpy(lexemInfoTransformationTempStack[lexemInfoTra
nsformationTempStackSize - 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 depthOfNontext) {
    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
            &&
!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize -
3].lexemStr, tokenStruct[MULTI_TOKEN_WHILE][0], MAX_LEXEM_SIZE)
            &&
!strncmp(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;
}

```

Lexica.cpp

```
#define _CRT_SECURE_NO_WARNINGS
```

```

/*****

```

```

* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *

```

```

*           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("-----\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;
    }

// get identifier id
unsigned int getIdentfierId(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;

        }
    }
    strncpy(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][A-Z]";

    if (std::regex_match(std::string(lexemInfoInTable->lexemStr), std::regex(identifiers_re))) {
        lexemInfoInTable->lexemId =
getIdentfierId(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 (!strncmp(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 (!strcmp(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|
//
|BODY|END|EXIT|CONTINUE|GET|PUT|IF|ELSE|FOR|TO|DOWNTOW|DO|WHILE|REPE
AT|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|unti
l|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;
}
Preparer.cpp #define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
* file: preparer.hxx *
* (draft!) *
*****/

```

```

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

int precedenceLevel(char* lexemStr) {
    //printf("TODO: (in precedenceLevel)\r\n");
    if (!strcmp(lexemStr,
tokenStruct[MULTI_TOKEN_BITWISE_NOT][0], MAX_LEXEM_SIZE)) {
        return 6;
    }
    else if (!strcmp(lexemStr,
tokenStruct[MULTI_TOKEN_NOT][0], MAX_LEXEM_SIZE)) {
        return 6;
    }

    else if (!strcmp(lexemStr,
tokenStruct[MULTI_TOKEN_BITWISE_AND][0], MAX_LEXEM_SIZE)) {
        return 5;
    }
    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_BITWISE_OR][0], MAX_LEXEM_SIZE)) {
        return 4;
    }
}

```

```

        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][0], MAX_LEXEM_SIZE)) {
            return 3;
        }
        else if (!strcmp(lexemStr,
tokenStruct[MULTI_TOKEN_GREATER][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_LRBIND][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_BITWISE_AND][0], MAX_LEXEM_SIZE)) {
        return true;
    }
    else 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_BITWISE_OR][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][0], MAX_LEXEM_SIZE)) {
            return true;
        }
        else if (!strcmp(lexemStr,
tokenStruct[MULTI_TOKEN_GREATER][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_LRBIND][0], MAX_LEXEM_SIZE)) { // ! TODO: ...
            return false;
        }

        else if (!strcmp(lexemStr,
tokenStruct[MULTI_TOKEN_RLBIND][0], MAX_LEXEM_SIZE)) {
            return false;
        }
        if (!strcmp(lexemStr,
tokenStruct[MULTI_TOKEN_BITWISE_NOT][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) {
        if (!strcmp((*lastLexemInfoInTable)[1].lexemStr,
tokenStruct[MULTI_TOKEN_RLBIND][0], MAX_LEXEM_SIZE)
            ||
            !strcmp((*lastLexemInfoInTable)[-
1].lexemStr, tokenStruct[MULTI_TOKEN_LRBIND][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)
        ) {
            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 makePrepareEnder(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;

    lexemInfoTransformationTempStackSize = 0;
    for (; (*lastLexemInfoInTable)->lexemStr[0] != '\0';
        *(*lastTempLexemInfoInTable)++ = *(*lastLexemInfoInTable)++) {
        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);
}
}
}
}

```

Semantic.cpp

```

#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
* file: semantix.h *
* (draft!) *
*****/

#include "../include/semantix/semantix.h"
#include "stdio.h"
#include "string.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 133
//
// #define NO_IMPLEMENT_CODE_STATE 256

int checkingInternalCollisionInDeclarations(/*TODO: add arg*/) {
    for (unsigned int index = 0; identifierIdsTable[index][0] !=
'\0'; ++index) {
        char isDeclaredIdentifier = 0;
        char isDeclaredIdentifierCollision = 0;
        unsigned int lexemIndex = 0;
        for (;
strncmp(lexemesInfoTable[lexemIndex].lexemStr, ";", MAX_LEXEM_SIZE) &&
lexemesInfoTable[lexemIndex].lexemStr[0] != '\0'; ++lexemIndex) {

```



```

        if (lexemesInfoTable[lexemIndex].tokenType
== IDENTIFIER_LEXEME_TYPE) {
            if (!strncmp(identifierIdsTable[index],
lexemesInfoTable[lexemIndex].lexemStr, MAX_LEXEM_SIZE)) {
                if (isDeclaredIdentifier) {
                    isDeclaredIdentifierCollision
= 1;
                }
                isDeclaredIdentifier = 1;
            }
        }
        ++lexemIndex;
        for (;
strncmp(lexemesInfoTable[lexemIndex].lexemStr, ";", MAX_LEXEM_SIZE) &&
lexemesInfoTable[lexemIndex].lexemStr[0] != '\0'; ++lexemIndex) {
            if (lexemesInfoTable[lexemIndex].tokenType
== IDENTIFIER_LEXEME_TYPE) {
                if (!strncmp(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 || strncmp(identifierIdsTable[index],
lexemesInfoTable[lexemIndex].lexemStr, MAX_LEXEM_SIZE)) {
                continue;
            }
            if (lexemesInfoTable[lexemIndex +
1].lexemStr[0] == ':') {
                if (isDeclaredLabel) {
                    isDeclaredLabelCollision = 1;
                }
                isLabel = 1;
                isDeclaredLabel = 1;
            }

```

```

        if (lexemIndex &&
!strcmp(lexemesInfoTable[lexemIndex - 1].lexemStr, "goto", MAX_LEXEM_SIZE)) {
            isLabel = 1;
        }
    }

    if (isDeclaredIdentifierCollision) {
        printf("Collision(identifier/identifier): %s\r\n",
identifierIdsTable[index]);

        return COLLISION_II_STATE;
    }
    if (isDeclaredLabelCollision) {
        printf("Collision(label/label): %s\r\n",
identifierIdsTable[index]);

        return COLLISION_LL_STATE;
    }
    if (isDeclaredIdentifier && isLabel) {
        printf("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]);

        return COLLISION_I_STATE;
    }
    else if (isLabel && !isDeclaredLabel) {
        printf("Undeclared label: %s\r\n",
identifierIdsTable[index]);

        return COLLISION_L_STATE;
    }
}

printf("Declaration verification was successful!\r\n");
return SUCCESS_STATE;
}

int checkingVariableInitialization(/*TODO: add args*/) {
    //TODO: implement this

    printf("\r\nTODO: implment \"int
checkingVariableInitialization(/*TODO: add args*/)\" \r\n\r\n");
    return NO_IMPLEMENT_CODE_STATE;
}

int checkingCollisionInDeclarationsByKeyWords(/*TODO: add args*/) {

```

```

//TODO: implement this

printf("\r\nTODO: implent \"int
checkingCollisionInDeclarationsByKeyWords(/*TODO: add args*/)\r\n\r\n");
return NO_IMPLEMENT_CODE_STATE;
}

```

Syntax.cpp

```

#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*          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__tokenGROUPEXPRESSIONIO
NEND","body_for_true__body_for_false"} },
    { "statement_in_while_body", 2,
{"tokenIF__tokenGROUPEXPRESSIONBEGIN__expression__tokenGROUPEXPRESSIONIO
NEND","body_for_true"} },
    { "statement_in_while_body", 2,
{"tokenFOR__cycle_counter_init__tokenTO__cycle_counter_last_value","cycle_body__tok
enSEMICOLON"} },
    { "statement_in_while_body", 2,
{"tokenWHILE__expression__statement_in_while_body____iteration_after_two","tokenEN
D__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__decla
ration"}}},
    {"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, {"sub"}},
        {"unary_operator", 1, {"add"}},
        {"binary_operator", 1, {"and"}},
        {"binary_operator", 1, {"or"}},
        {"binary_operator", 1, {"="}},
        {"binary_operator", 1, {"<>"}},
        {"binary_operator", 1, {"<"}},
        {"binary_operator", 1, {">"}},
        {"binary_operator", 1, {"add"}},
        {"binary_operator", 1, {"sub"}},
        {"binary_operator", 1, {"*"}},

```

```

    {"binary_operator", 1, {"/"}}},
    {"binary_operator", 1, {"%"}}},
    {"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"}},

```

```

NEND", 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, {"add"}},
{"tokenMINUS", 1, {"sub"}},
{"tokenMUL", 1, {"*"}},
{"tokenDIV", 1, {"/"}},
{"tokenMOD", 1, {"%"}},
{"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, {"scan"}},
{"tokenPUT", 1, {"print"}},
{"tokenNAME", 1, {"program"}},
{"tokenBODY", 1, {"start"}},
{"tokenDATA", 1, {"var"}},
{"tokenEND", 1, {"finish"}},
{"tokenSEMICOLON", 1, {";"}},
//
{ "value", 1, {"value_terminal"} },
//
{ "ident", 1, {"ident_terminal"} },
//
// { "label", 1, {"ident_terminal"} },
//
{ "", 2, {"",""} }
},

```

```

176,
"program"
#endif
};

#define DEBUG_STATES

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* buildAST(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(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) << "sub";
            }
        }
        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) << "sub";
        }
    }
    file << endl;
}

file.close();
}

#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} } };
bool cykAlgorithmImplementation(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

#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")
            || !strncmp(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";

return containsSymbolSet(&parseInfoTable[0][lexemIndex - 1], grammar->start_symbol);
}

#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")
                || !strncmp(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;
}

//

bool syntaxAnalyze(LexemInfo* lexemInfoTable, Grammar* grammar, char
syntaxlAnalyzeMode) {
    bool cykAlgorithmImplementationReturnValue = false;
    if (syntaxlAnalyzeMode == SYNTAX_ANALYZE_BY_CYK_ALGORITHM) {
        cykAlgorithmImplementationReturnValue =
        cykAlgorithmImplementation(lexemesInfoTable, grammar);
        printf("cykAlgorithmImplementation return \"%s\".\r\n",
        cykAlgorithmImplementationReturnValue ? "true" : "false");
        if (cykAlgorithmImplementationReturnValue) {
            return true;
        }
    }

    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 true;
            }
        }
        else {
            printf("Parse failed: Extra tokens remain.\n");
            return false;
        }
    }
}

```



```

    }
}
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);
    }
    else {
        printf("Parse failed: unexpected terminal.\r\n");
    }
    return false;
}
return false;
}

return false;
}

```

```

bool syntaxlAnalyze_(LexemInfo* lexemInfoTable, Grammar* grammar, char
syntaxlAnalyzeMode) {
    bool cykAlgorithmImplementationReturnValue = false;
    if (syntaxlAnalyzeMode == SYNTAX_ANALYZE_BY_CYK_ALGORITHM) {
        bool cykAlgorithmImplementationReturnValue =
cykAlgorithmImplementation(lexemesInfoTable, grammar);

        printf("cykAlgorithmImplementation return \"%s\".\r\n",
cykAlgorithmImplementationReturnValue ? "true" : "false");
    }

    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 \"\n 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

#ifdef 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][lexemIndex].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 = buildAST(parseInfoTable, grammar, 0, lexemIndex - 1, grammar-
>start_symbol);
if (astRoot) {
    std::cout << "Abstract Syntax Tree:\n";
    printAST(lexemInfoTable, astRoot);
    delete astRoot; // Íâ çàáóâà°î çâ³ëüíÿòè ïàì'ÿòü
}
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;
}

#if 0
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

```

Syntax.h

```

#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           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 cykAlgorithmImplementation(struct LexemInfo* lexemInfoTable, Grammar*
grammar);
bool syntaxAnalyze(LexemInfo* lexemInfoTable, Grammar* grammar, char
syntaxlAnalyzeMode);

semantic.h
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
* 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 133
```

```
#define NO_IMPLEMENT_CODE_STATE 256
```

```
int checkingInternalCollisionInDeclarations(/*TODO: add arg*/);
int checkingVariableInitialization(/*TODO: add args*/);
int checkingCollisionInDeclarationsByKeyWords(/*TODO: add args*/);
```

preparer.h

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*          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);

```

lexica.h

```

/////#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*/);
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]));

```

add.h

```

#define _CRT_SECURE_NO_WARNINGS

```

```

/*****

```

```

* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *

```

```

* file: add.hxx *

```

```

* (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 // lex + rpn + MACHINECODEGEN! *
*           file: and.hxx           *
*                               (draft!) *
*****/
```

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

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

bitwise_and.h

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           file: bitwise_and.hxx           *
*                               (draft!) *
*****/
```

```
#define BITWISE_AND_CODER(A, B, C, M, R)\
if (A ==* B) C = makeBitwiseAndCode(B, C, M);
```

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

bitwise_or.h

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           file: bitwise_not.h           *
*                               (draft!) *
*****/
```

```
#define BITWISE_NOT_CODER(A, B, C, M, R)\
```

```
if (A ==* B) C = makeBitwiseNotCode(B, C, M);
```

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

```
bitwise_or.h
```

```
#define _CRT_SECURE_NO_WARNINGS
```

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

```
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
```

```
* file: bitwise_or.hxx *
```

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

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

```
#define BITWISE_OR_CODER(A, B, C, M, R)\
```

```
if (A ==* B) C = makeBitwiseOrCode(B, C, M);
```

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

```
div.h
```

```
#define _CRT_SECURE_NO_WARNINGS
```

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

```
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
```

```
* file: div.hxx *
```

```
* (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 // lex + rpn + MACHINECODEGEN! *
```

```
* file: else.hxx *
```

```
* (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 // lex + rpn + MACHINECODEGEN! *
*           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 // lex + rpn + MACHINECODEGEN! *
*           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

```
#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 OBJECT_X86_WIN32_CODER_MODE 0x04
#define MACHINE_CODER_MODE 0x08

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 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* getCodeBytePtr(unsigned char* baseBytePtr);
```

```
void makeCode(struct LexemInfo** lastLexemInfoInTable/*TODO:...*/, unsigned char*
currBytePtr);
```

```
#endif
```

```
Goto_label.h
```

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rp + MACHINECODEGEN! *
* file: goto_label.h *
* (draft!) *
*****/
```

```
#include <string>
#include <map>
```

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

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

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

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

```
greater.h
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024-2025 // lex + rp + MACHINECODEGEN! *
* file: greater.h *
* (draft!) *
*****/
```

```
#define GREATER_CODER(A, B, C, M, R)\
if (A ==* B) C = makeIsGreaterCode(B, C, M);
```

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

```
greater_or_equal.h
```

```
#define _CRT_SECURE_NO_WARNINGS
```

```

/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN!  *
*           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 // lex + rpn + MACHINECODEGEN!  *
*           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 // lex + rpn + MACHINECODEGEN!  *
*           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.h

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rp + MACHINECODEGEN!  *
*           file: less.h                               *
*           (draft!) *
*****/

#define LESS_CODER(A, B, C, M, R)\
if (A ==* B) C = makeIsLessCode(B, C, M);

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

less_or_equal.h

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rp + MACHINECODEGEN!  *
*           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);
```

lrbind.h

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lrbind codegen                *
*           file: lrbind.hxx                             *
*           (draft!) *
*****/

#define LRBIND_CODER(A, B, C, M, R)\
if (A ==* B) C = makeLeftToRightBindCode(B, C, M);
```



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

mod.h

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
*           file: mod.hxx                               *
*           (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 // lex + rpn + MACHINECODEGEN! *
*           file: mul.hxx                               *
*           (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 // lex + rpn + MACHINECODEGEN! *
*           file: not.hxx                               *
*           (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 // lex + rpn + MACHINECODEGEN! *
```

```
* file: not_equal.hxx *
```

```
* (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);
```

```
null_statement.h
```

```
#define _CRT_SECURE_NO_WARNINGS
```

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

```
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
```

```
* file: null_statement.hxx *
```

```
* (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 // lex + rpn + MACHINECODEGEN! *
```

```
* 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 // lex + rp + MACHINECODEGEN! *
*           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 // lex + rp + MACHINECODEGEN! *
*           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);
```

repeat-until.h

```
#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rp + MACHINECODEGEN! *
*           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);

```

```

rlbind.h

```

```

#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lrbind codegen *
* 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);

```

```

semicolon.h

```

```

#define _CRT_SECURE_NO_WARNINGS
/*****
* N.Kozak // Lviv'2024 // lex + rpn + MACHINECODEGEN! *
* 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);

```



```

*****/

```

```

#define MAX_PARAMETERS_SIZE 4096
#define PARAMETERS_COUNT 4
#define INPUT_FILENAME_PARAMETER 0

```

```

#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"

```

```

#define DEFAULT_INPUT_FILENAME "file44.z10" // TODO: move!

```

```

#define PREDEFINED_TEXT \
    "name MN\r\n" \
    "data\r\n" \
    "  #*argumentValue*#\r\n" \
    "  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; --\r\n" \
    "argumentValue)*#\r\n" \
    "    RV << RV ** AV; #*resultValue *= argumentValue;\r\n" \
    "    *#\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 int mode;
extern char parameters[PARAMETERS_COUNT][MAX_PARAMETERS_SIZE];

```

```
void comandLineParser(int argc, char* argv[], unsigned 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);
```