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:

М’якішев Є.М.

Перевірив:

старший викладач каф. ЕОМ

Козак Н.Б.

Львів-2024

**ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ**

1. Цільова мова транслятора – мова програмування С або асемблер для 32/64 розрядного процесора.

2. Для отримання виконавчого файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio бо будь-яким іншим.

3. Мова розробки транслятора: С/C++.

4. Реалізувати оболонку або інтерфейс з командного рядка.

5. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.

6. На виході розробленого транслятора мають створюватись такі файли:

* *файл з лексемами;*
* *файл з повідомленнями про помилки (або про їх відсутність);*
* *файл на мові асемблера;*
* *об’єктний файл;*
* *виконавчий файл.*

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

**Деталізація завдання на проєктування:**

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

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

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

**Деталізований опис власної мови програмування:**

* Тип даних: int\_2
* Блок тіла програми: startprogram; startblok variable…; endblok
* Оператор вводу: get ()
* Оператор виводу: put ()
* Оператори: if-else (C)

goto (C)

for-To (Паскаль)

for-downto (Паскаль)

while (Бейсік)

repeat-until (Паскаль)

* Регістр ключових слів: Low
* Регістр ідентифікаторів: Up4 перший символ \_
* Операції арифметичні: ++, -, \*, div, mod
* Операції порівняння: eq, noteq, less, gr
* Операції логічні: not, and, or
* Коментар: \*/….
* Ідентифікатори змінних, числові константи
* Оператор присвоєння: <-

АНОТАЦІЯ

У цьому курсовому проєкті розроблено транслятор, який здійснює перетворення вхідної мови, визначеної варіантом завдання, у мову асемблера. Трансляція включає три основні етапи: лексичний аналіз, синтаксичний аналіз і генерацію коду.

На етапі лексичного аналізу вхідний потік символів розбивається на лексеми, які заносяться до спеціальної таблиці. Для кожної лексеми створюється унікальний числовий ідентифікатор, що спрощує подальшу обробку. У таблицю також додаються номер рядка, значення числових лексем та інші важливі параметри.

Синтаксичний аналіз виконується за допомогою висхідного методу без повернення, що забезпечує побудову дерева розбору, починаючи від листків і завершуючи коренем. Цей процес гарантує правильну структуру даних відповідно до синтаксичних правил вхідної мови.

Етап генерації коду використовує таблицю лексем для створення асемблерного коду, який відповідає кожному блоку програми. Згенерований код зберігається у вихідному файлі та готовий до компіляції.

Отриманий асемблерний код може бути скомпільований і виконаний за допомогою інструментів, таких як LINK або ML.

**ЗМІСТ**

Зміст

[АНОТАЦІЯ 5](#_Toc189252104)

[2. Формальний опис вхідної мови програмування 10](#_Toc189252105)

[2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура. 10](#_Toc189252106)

[2.2Опис термінальних символів та ключових слів 12](#_Toc189252107)

[3. Розробка транслятора з вхідної мови програмування 15](#_Toc189252108)

[3.1. Вибір технології програмування. 15](#_Toc189252109)

[3.2. Проектування таблиць транслятора та вибір структур даних. 15](#_Toc189252110)

[3.3. Розробка лексичного аналізатора. 17](#_Toc189252111)

[Обробка лексем 17](#_Toc189252112)

[Структура таблиці лексем 18](#_Toc189252113)

[Особливості лексичного аналізу 18](#_Toc189252114)

[**3.3.1.** **Розробка алгоритму роботи лексичного аналізатора.** 19](#_Toc189252115)

[Основні елементи алгоритму 19](#_Toc189252116)

[Етапи роботи алгоритму 20](#_Toc189252117)

[Висновок 21](#_Toc189252118)

[Особливості 21](#_Toc189252119)

[Переваги 22](#_Toc189252120)

[**3.3.2.** **Опис програми реалізації лексичного аналізатора.** 23](#_Toc189252121)

[Мета і принципи роботи лексичного аналізатора 23](#_Toc189252122)

[Алгоритм роботи 23](#_Toc189252123)

[Переваги реалізації 24](#_Toc189252124)

[Висновок 24](#_Toc189252125)

[3.4. Розробка синтаксичного та семантичного аналізатора. 24](#_Toc189252126)

[**Процес синтаксичного аналізу** 24](#_Toc189252127)

[**Семантичний аналіз** 25](#_Toc189252128)

[**3.4.1.** **Розробка дерева граматичного розбору.** 25](#_Toc189252129)

[**3.4.2.** **Розробка алгоритму роботи синтаксичного і семантичного аналізатора.** 26](#_Toc189252130)

[Основні функції 28](#_Toc189252131)

[Ключові аспекти 28](#_Toc189252132)

[Типовий процес 28](#_Toc189252133)

[3.5. Розробка генератора коду. 30](#_Toc189252134)

[**3.5.1.** **Розробка алгоритму роботи генератора коду.** 32](#_Toc189252135)

[**3.5.2.** **Опис програми реалізації генератора коду.** 33](#_Toc189252136)

[Основні компоненти генератора коду: 33](#_Toc189252137)

[Етапи роботи генератора коду: 33](#_Toc189252138)

[Режими роботи генератора: 34](#_Toc189252139)

[4. Налагодження та тестування розробленого транслятора 35](#_Toc189252140)

[4.1. Опис інтерфейсу та інструкції користувачу. 35](#_Toc189252141)

[Інтерактивний режим 36](#_Toc189252142)

[Інструкція користувача 36](#_Toc189252143)

[Помилки та способи їх усунення 36](#_Toc189252144)

[4.2. Виявлення лексичних і синтаксичних помилок. 36](#_Toc189252145)

[4.3. Перевірка роботи транслятора за допомогою тестових задач. 37](#_Toc189252146)

[Тестова програма №1 *Лінійний алгоритм* 37](#_Toc189252147)

[Х = (А - В) \* 10 + (А + В) / 10 37](#_Toc189252148)

[Тестова програма №2 38](#_Toc189252149)

[Список літературних джерел 41](#_Toc189252150)

[Додатки 42](#_Toc189252151)

**ВСТУП**

**Транслятор** — це програмне забезпечення, яке забезпечує перетворення вихідного коду, написаного однією мовою програмування, у його еквівалент на іншій мові. Якщо вхідною є мова високого рівня, а вихідною — асемблер або машинний код, такий транслятор називається **компілятором**.

Транслятори поділяються на два основних типи: **компілятори** та **інтерпретатори**.

**Компіляція** складається з двох ключових етапів:

1. **Аналіз**, під час якого вихідний код розбивається на лексеми, перевіряється його синтаксична коректність, а також створюється проміжне представлення програми.
2. **Синтез**, що передбачає перетворення цього проміжного представлення на об'єктний код, який складається з інструкцій для конкретної апаратної платформи.

На відміну від компіляторів, **інтерпретатори** не створюють виконуваного файлу. Вони аналізують вихідний код і формують його проміжне представлення, після чого одразу виконують команди без генерування об'єктного коду.

Компілятор виконує перетворення вихідного коду з однієї мови на іншу. Вхідними даними є текст програми, а результатом — об'єктний код, який можна виконувати безпосередньо на конкретному комп’ютері. Примітно, що компілятор може бути створений за допомогою іншої мови програмування, що свідчить про його універсальність і важливість у розробці програмного забезпечення.

1. **ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ**

**Транслятор** — це програмне забезпечення, яке здійснює перетворення вихідного коду, написаного на одній мові програмування, у виконуваний код, представлений об’єктною мовою. До трансляторів належать програми, які відрізняються за способом і типом обробки вихідного коду. Основні види сучасних трансляторів — асемблери, компілятори та інтерпретатори.

**Асемблер** — програма, яка перетворює символічний код, написаний на асемблерній мові, безпосередньо в машинні інструкції. Ключова характеристика асемблерів полягає у відповідності кожної команди вихідного коду одній машинній інструкції.

**Компілятор** — це інструмент, який здійснює трансляцію вихідного коду, написаного мовою високого рівня, у машинний код, що виконується безпосередньо комп’ютером. Компілятори реалізують складніший процес перетворення порівняно з асемблерами, адаптуючи код для конкретної платформи.

**Інтерпретатор** — програма, яка виконує вихідний код без створення окремого виконуваного файлу. Інтерпретатор обробляє інструкції послідовно під час виконання, що робить його зручним для тестування й налагодження програм.

Процес трансляції складається з кількох основних етапів:

* **Лексичний аналіз**: вихідний код розбивається на лексеми — мінімальні одиниці мови програмування. На цьому етапі відбувається виявлення помилок, пов’язаних із некоректними символами чи невірними форматами ідентифікаторів.
* **Синтаксичний аналіз**: будується синтаксичне дерево, що відображає структуру програми згідно з граматикою мови. Застосовуються методи аналізу, наприклад LL(1) або LR(1).
* **Семантичний аналіз**: виконується перевірка логічної коректності програми, яка включає контроль типів даних, областей видимості та відповідності параметрів функцій.
* **Оптимізація коду**: здійснюється вдосконалення проміжного представлення програми для підвищення ефективності її виконання. Це може бути як локальна, так і глобальна оптимізація, залежна чи незалежна від архітектури.
* **Генерація коду**: завершується створенням об’єктного чи асемблерного коду, готового до виконання або подальшої компіляції.

Залежно від типу транслятора деякі етапи можуть бути поєднані або скорочені. Наприклад, однопрохідні транслятори зазвичай пропускають стадії проміжного представлення та оптимізації.

Під час лексичного аналізу створюється таблиця, яка містить інформацію про ідентифікатори, рядки та числові значення. Синтаксичний аналіз формує дерево розбору, яке використовується на наступних етапах для оптимізації та генерації коду. Семантичний аналіз забезпечує перевірку типів, контроль за областями видимості та правильність виклику функцій.

Результатом усіх етапів трансляції є об’єктний код, оптимізований для конкретної апаратної архітектури. Цей код може бути скомпільований і виконаний на цільовій платформі. Таким чином, трансляція забезпечує поступове перетворення вихідного коду у готовий до запуску виконуваний файл.

1. Формальний опис вхідної мови програмування
   1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура.

labeled\_point = label , ":"

goto\_label = tokenGOTO, label, ";"

program\_name = ident,";"

value\_type = tokenINTEGER16

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 = '+'

//

//

tokenCOLON = ":"

tokenGOTO = "goto"

tokenINTEGER16 = "int\_2"

tokenCOMMA = ","

tokenNOT = "!"

tokenAND = "and"

tokenOR = "or"

tokenEQUAL = "eq"

tokenNOTEQUAL = "noteq"

tokenPLUS = "+"

tokenMINUS = "-"

tokenMUL = "\*"

tokenDIV = "div"

tokenMOD = "mod"

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 = "startprogram"

tokenBODY = "startblok"

tokenDATA = "variable"

tokenEND = "endblok"

tokenSEMICOLON = ";"

//

tokenUNDERSCORE = "\_"

//

//

2.2Опис термінальних символів та ключових слів

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

|  |  |
| --- | --- |
| Термінальний символ або ключове слово | Значення |
| startprogram | Початок програми |
| startblok | Початок тексту програми |
| variable | Початок блоку опису змінних |
| endblok | Кінець розділу операторів |
| get | Оператор вводу змінних |
| put | Оператор виводу (змінних або рядкових констант) |
| <- | Оператор присвоєння |
| if | Оператор умови |
| else | Оператор умови |
| goto | Оператор переходу |
| label | Мітка переходу |
| for | Оператор циклу |
| to | Інкремент циклу |
| downto | Декремент циклу |
| do | Початок тіла циклу |
| while | Оператор циклу |
| repeat | Початок тіла циклу |
| until | Оператор циклу |
| + | Оператор додавання |
| - | Оператор віднімання |
| \* | Оператор множення |
| / | Оператор ділення |
| % | Оператор знаходження залишку від ділення |
| eq | Оператор перевірки на рівність |
| noteq | Оператор перевірки на нерівність |
| less | Оператор перевірки чи менше |
| gr | Оператор перевірки чи більше |
| ! | Оператор логічного заперечення |
| and | Оператор кон’юнкції |
| or | Оператор диз’юнкції |
| int\_2 | 16-ти розрядні знакові цілі |
| /\*… | Коментар |
| , | Розділювач |
| ( | Відкриваюча дужка |
| ) | Закриваюча дужка |

До термінальних символів віднесемо також усі цифри (0-9), латинські букви (a-z, A-Z), символи табуляції, символ переходу на нову стрічку, пробілу.

1. Розробка транслятора з вхідної мови програмування
   1. Вибір технології програмування.

Перед початком розробки програми важливо визначитися з технологією програмування, середовищем розробки та розробити алгоритм її роботи. Це дозволить забезпечити ефективність, зручність і швидкість процесу створення програмного забезпечення.

Для реалізації поставленого завдання обрано середовище розробки **Microsoft Visual Studio 2022** і мову програмування **C/C++**. Такий вибір обґрунтований можливістю створення високопродуктивного коду та широкими можливостями налагодження й оптимізації, які надає це середовище.

Для забезпечення простоти використання розробленої програми було вирішено створити **консольний інтерфейс**, що є зручним для взаємодії користувача із програмою та достатнім для виконання поставлених задач.

* 1. Проектування таблиць транслятора та вибір структур даних.

**LexemInfo** — це структура, призначена для зберігання та обробки даних про лексеми, отримані під час лексичного аналізу. Вона забезпечує зручний доступ до характеристик кожної лексеми, що є важливим для подальших етапів аналізу та обробки.

#### Члени структури:

1. **char lexemStr[MAX\_LEXEM\_SIZE]**
   * Масив символів, у якому зберігається текстове представлення лексеми.
   * MAX\_LEXEM\_SIZE визначає максимальну довжину лексеми (зазвичай задається як константа).
2. **unsigned long long int lexemId**
   * Унікальний числовий ідентифікатор, що дозволяє відрізняти лексему від інших.
3. **unsigned long long int tokenType**
   * Тип токена, що визначає категорію, до якої належить лексема (наприклад, оператор, ключове слово чи константа).
4. **unsigned long long int ifvalue**
   * Додаткове значення для умовних виразів або семантичного аналізу, наприклад, числове значення для констант чи інформація для перевірки умов.
5. **unsigned long long int row**
   * Номер рядка у вихідному коді, де знаходиться лексема. Використовується для відстеження розташування лексеми, що полегшує діагностику помилок.
6. **unsigned long long int col**
   * Номер колонки в рядку, яка відповідає позиції лексеми у вихідному коді.
7. **Місце для розширення**
   * Коментар // TODO: ... передбачає можливість додавання нових полів для майбутніх функціональних покращень.

#### Конструктори:

1. **Конструктор за замовчуванням**
   * LexemInfo()
   * Ініціалізує всі поля нульовими або стандартними значеннями (наприклад, порожнім рядком для lexemStr).
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)
   * Дозволяє створити об'єкт із заданими значеннями:
     + lexemStr — текстове значення лексеми.
     + lexemId — унікальний ідентифікатор.
     + tokenType — тип токена.
     + ifvalue — додаткові дані.
     + row та col — позиція лексеми у вихідному коді.
3. **Конструктор копіювання**
   * LexemInfo(const NonContainedLexemInfo& nonContainedLexemInfo)
   * Ініціалізує структуру на основі іншого об’єкта типу NonContainedLexemInfo, забезпечуючи сумісність між форматами даних.

#### Призначення:

Структура **LexemInfo** виконує такі функції:

* Зберігання атрибутів лексеми, отриманих під час лексичного аналізу.
* Визначення позиції лексеми у вихідному коді (рядок і колонка), що важливо для повідомлень про помилки.
* Забезпечення структури даних для побудови синтаксичного дерева та подальшого аналізу.
* Можливість розширення для додавання нових функцій, наприклад, для семантичного аналізу чи оптимізації.

**LexemInfo** є гнучким та розширюваним інструментом для обробки лексем, який відіграє ключову роль у процесі трансляції.

* 1. Розробка лексичного аналізатора.

Лексичний аналіз спрямований на розбиття вхідного тексту, що складається з послідовності символів, на окремі компоненти — лексеми. Основне завдання цього етапу полягає у виявленні окремих слів чи елементів програми з суцільного потоку символів. Усі символи тексту класифікуються як:

* Ті, що формують лексеми.
* Роздільники, які відокремлюють лексеми.

Процес виконується шляхом послідовної обробки вхідного тексту з використанням стандартних методів роботи з рядками. Аналізатор проходить текст від початку до кінця, розділяючи лексеми на основі пробілів, знаків операцій, спеціальних символів (таких як переведення рядка чи табуляція).

На цьому етапі виділяються та класифікуються такі типи лексичних одиниць:

* **Ідентифікатори**.
* **Літерали**.
* **Термінальні символи** (наприклад, оператори, дужки, ключові слова).

### Обробка лексем

Кожна виділена лексема аналізується, розпізнається та додається до таблиці лексем. У таблиці кожній лексемі призначається унікальний ідентифікатор. Це дозволяє замість обробки послідовностей символів працювати з числовими представленнями лексем, що значно спрощує подальші етапи компіляції.

Зокрема, робота з таблицею лексем забезпечує:

* Легку перевірку належності лексеми до певної синтаксичної конструкції.
* Зручний доступ до попередніх або наступних лексем під час аналізу.

### Структура таблиці лексем

Окрім унікального ідентифікатора, таблиця містить:

* Номер рядка, де розташована лексема, що допомагає виявляти та вказувати місце помилки у вихідному коді.
* Додаткову інформацію, яка може використовуватися на інших етапах аналізу.

### Особливості лексичного аналізу

Коментарі у вихідному тексті ігноруються, оскільки вони не впливають на виконання програми, синтаксичний розбір чи генерацію коду. Видалення коментарів з тексту забезпечує більш ефективну роботу подальших фаз компіляції.

Таким чином, лексичний аналіз виконує ключову функцію, створюючи основу для синтаксичного аналізу шляхом перетворення тексту програми на набір структурованих лексем.

**Типи лексем (лексичні класи):**

1. **Ключові слова**:  
   startprogram, variable, endblok, get, put, int\_2, if, else, for, goto, downto, repeat, until, while.
2. **Ідентифікатори**.
3. **Числові константи**:  
   Цілі числа без знаку.
4. **Оператор присвоєння**:  
   <-.
5. **Знаки операцій**:  
   +, -, \*, div, gr, less, eq, noteq , !, and, or.
6. **Роздільники**:  
   ; ,
7. **Дужки**:  
   (, )
   * 1. **Розробка алгоритму роботи лексичного аналізатора.**

Лексичний аналізатор є програмним компонентом, який перетворює вихідний текст програми на послідовність лексем — основних синтаксичних елементів. Його ключова задача полягає у розпізнаванні та класифікації лексем для забезпечення наступних етапів компіляції, таких як синтаксичний або семантичний аналіз. Основний функціонал аналізатора охоплює обробку ідентифікаторів, значень, ключових слів та ігнорування коментарів.

### Основні елементи алгоритму

#### 1. Структури даних

**LexemInfo** — головна структура для зберігання характеристик кожної лексеми:

* lexemStr — текст лексеми.
* lexemId — унікальний номер для ідентифікації лексеми.
* tokenType — тип лексеми (ключове слово, ідентифікатор, значення тощо).
* ifvalue — додаткова інформація (наприклад, числове значення).
* row та col — позиція лексеми у вихідному тексті (номер рядка і стовпця).

**NonContainedLexemInfo** — додаткова структура для тимчасового збереження лексем у буфері (наприклад, tempStrFor\_123).

#### 2. Основні масиви

* **lexemesInfoTable** — таблиця для зберігання всіх виявлених лексем.
* **identifierIdsTable** — структура, яка забезпечує унікальність ідентифікаторів, запобігаючи їх дублюванню.

### Етапи роботи алгоритму

#### 1. Розділення тексту на токени (токенізація)

За допомогою регулярних виразів (наприклад, TOKENS\_RE) текст розбивається на окремі елементи: ключові слова, ідентифікатори, значення тощо. Ітератор (std::sregex\_token\_iterator) послідовно знаходить токени, які обробляються відповідними функціями.

#### 2. Розпізнавання токенів

Кожен знайдений токен перевіряється наступними методами:

* **tryToGetKeyWord** — визначає, чи є токен ключовим словом.
* **tryToGetIdentifier** — перевіряє, чи є токен ідентифікатором.
* **tryToGetUnsignedValue** — визначає, чи це числове значення.

Токени, які не відповідають жодному з цих типів, маркуються як "невизначені".

#### 3. Класифікація лексем

* Ключові слова розпізнаються за шаблоном (наприклад, KEYWORDS\_RE) і додаються до таблиці з унікальним lexemId.
* Ідентифікатори перевіряються за допомогою виразу (IDENTIFIERS\_RE) та зберігаються в identifierIdsTable.
* Числові значення обробляються окремо, отримуючи числове представлення в полі ifvalue.

#### 4. Видалення коментарів

Спеціальна функція (commentRemover) видаляє однорядкові (//) та багаторядкові коментарі (/\* ... \*/). Видалені символи замінюються пробілами для збереження коректної структури тексту.

#### 5. Визначення позиції лексем

Функція (setPositions) розраховує, у якому рядку та стовпці знаходиться кожна лексема. Це дозволяє ефективно вказувати на помилки у вихідному тексті.

#### 6. Формування результатів

Результат роботи лексичного аналізатора представляється у вигляді таблиці, яка включає:

* Індекс лексеми у списку.
* Текст лексеми.
* Унікальний ідентифікатор (lexemId).
* Тип лексеми (ключове слово, ідентифікатор, значення тощо).
* Додаткові дані (наприклад, числове значення).
* Позицію у тексті (рядок і стовпець).

### Висновок

Розроблений алгоритм лексичного аналізатора забезпечує ефективну обробку вихідного тексту, виділяючи ключові синтаксичні елементи. Завдяки використанню структурованих даних та регулярних виразів процес класифікації лексем є швидким і гнучким, що полегшує роботу наступних фаз компіляції.

**Табл.1.**

| **Індекс** | **Текст лексеми** | **Ідентифікатор** | **Тип** | **Значення** | **Рядок** | **Стовпець** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | startprogram | 101 | Ключове слово | - | 1 | 1 |
| 1 | Pro | 1 | Ідентифікатор | - | 1 | 9 |
| 2 | startblok | 102 | Ключове слово | - | 2 | 1 |
| 3 | 123 | 1001 | Значення | 123 | 3 | 5 |
| 4 | endblok | 103 | Ключове слово | - | 4 | 1 |

### Особливості

1. **Ефективність за рахунок буферизації:** Буфер (tempStrFor\_123) використовується для збереження тимчасових даних, що мінімізує зайве використання пам’яті.
2. **Адаптивність алгоритму:** Регулярні вирази (TOKENS\_RE, IDENTIFIERS\_RE тощо) легко модифікуються для підтримки нових правил синтаксису або додаткових типів лексем.
3. **Вбудована обробка помилок:** Токени, які не відповідають визначеним правилам, автоматично класифікуються як "помилкові" без зупинки процесу аналізу.

### Переваги

* **Зрозумілість аналізу:** Завдяки визначенню позицій у вихідному тексті (рядок і стовпець) помилки та невідомі лексеми легко ідентифікуються та локалізуються.
* **Модульність:** Структуровані таблиці лексем забезпечують зручність додавання нових типів даних або розширення функціоналу аналізатора.
* **Цілісність:** Уся інформація про лексеми впорядкована у єдиній базі, що забезпечує швидкий доступ до потрібних даних для наступних етапів обробки.

Зображення, що містить текст, схема, Паралель, квитанція

Вміст, створений ШІ, може бути неправильним.

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

* + 1. **Опис програми реалізації лексичного аналізатора.**

### Мета і принципи роботи лексичного аналізатора

Лексичний аналізатор призначений для розбиття вхідного тексту програми на окремі лексеми — смислові одиниці, які є основою для подальших етапів аналізу. Символи тексту класифікуються як ті, що входять до складу лексем, або як роздільники. Аналіз проводиться поступово, зчитуючи текст з початку до кінця, використовуючи стандартні алгоритми обробки рядків.

Основні лексичні одиниці (ідентифікатори, літерали, оператори, ключові слова тощо) виділяються за допомогою спеціальних символів (пробілів, розділювачів, знаків операцій тощо). Кожна знайдена лексема обробляється та класифікується відповідно до визначених правил.

### Алгоритм роботи

1. **Розділення тексту на лексеми:**  
   Лексеми виявляються за допомогою функції tokenize(). Ця функція обробляє текст, виділяючи послідовності символів, що відповідають певним шаблонам (наприклад, зарезервовані слова або числові константи).
2. **Присвоєння типів:**  
   Виділені лексеми звіряються із зарезервованими ключовими словами. Якщо знайдено збіг, лексемі присвоюється відповідний тип. Наприклад, для числових значень додається їх числове представлення.
3. **Збереження результатів:**  
   Усі знайдені лексеми додаються до таблиці m\_tokens. Це забезпечує можливість подальшої роботи з лексемами не як із текстовими рядками, а як із структурованими одиницями даних із чітко визначеними типами.
4. **Навігація текстом:**  
   Завдяки збереженню позицій кожної лексеми (номер рядка та стовпця), аналізатор дозволяє швидко знаходити місця, де виникають помилки, або перевіряти правильність синтаксису.
5. **Обробка помилок:**  
   У разі виявлення некоректних символів або неправильних ідентифікаторів, вони позначаються як помилкові, але аналіз триває, щоб забезпечити обробку коректної частини тексту.

### Переваги реалізації

* **Систематизація даних:** Таблиця лексем містить як текстовий вигляд лексеми, так і її тип, координати у тексті, що спрощує подальший синтаксичний аналіз.
* **Легка інтеграція:** Лексеми зберігаються в уніфікованій формі, що дозволяє зручно передавати їх у наступні модулі аналізу.
* **Виявлення помилок:** Збереження координат помилкових лексем дає змогу швидко ідентифікувати та виправляти проблемні ділянки коду.

### Висновок

Реалізований лексичний аналізатор виконує первинну обробку тексту програми, виділяючи лексеми та формуючи таблицю з їх атрибутами. Такий підхід полегшує подальший синтаксичний і семантичний аналіз, забезпечуючи структуровану основу для роботи компілятора.

* 1. Розробка синтаксичного та семантичного аналізатора.

Синтаксичний аналіз — це етап компіляції, що визначає, чи відповідає послідовність лексем граматиці певної мови програмування. Теоретично для будь-якої граматики можна створити синтаксичний аналізатор, але граматики, що застосовуються на практиці, зазвичай мають спеціальні структури. Наприклад, для будь-якої контекстно-вільної граматики можна побудувати аналізатор з часом роботи, що не перевищує O(n³) для вхідного рядка довжиною n, однак для мов програмування існують більш оптимізовані граматики, що дозволяють створювати аналізатори з лінійною складністю.

У реальних мовах програмування аналізатори зазвичай мають лінійну складність. Це досягається переглядом коду зліва направо з підгляданням лише одного термінального символу (лексем), що дозволяє обробляти вхід ефективніше.

**Процес синтаксичного аналізу**

Вхід синтаксичного аналізатора складається з послідовності лексем та таблиць представлень, які створюються на етапі лексичного аналізу. Синтаксичний аналізатор перевіряє, чи відповідає послідовність лексем граматичним правилам мови. На виході аналізатора утворюється **дерево граматичного розбору**, яке представляє структуру програми згідно з правилами граматики. Також створюються таблиці ідентифікаторів і типів, що служать для подальшої обробки.

**Семантичний аналіз**

Семантичний аналіз здійснює перевірку змістовної коректності програми, аналізуючи відповідність операцій типам даних і забезпечуючи виконання обмежень, визначених мовою програмування. Семантичний аналіз дозволяє виявити логічні помилки, які можуть виникнути через некоректне використання змінних, типів чи операцій. В результаті цього етапу дерева розбору анотовані додатковою інформацією, яка буде використана при генерації машинного коду.

* + 1. **Розробка дерева граматичного розбору.**

Схема дерева розбору виглядає наступним чином:

<програма>

│

├── 'startprogram'

├── <назва програми>

│ └── <ідентифікатор>

├── 'startblok'

├── <оголошення змінних>

│ ├── ‘variable’

│ │ ├ <тип даних>

│ │ └── 'int\_2'

│ └── <список змінних>

│ ├── <ідентифікатор>

│ └── { ',' <ідентифікатор> }

├── <тіло програми>

│ ├── <оператор>

│ └── { <оператор> }

│ │

│ ├── <присвоєння>

│ │ ├── <ідентифікатор>

│ │ ├── '<-'

│ │ └── <арифметичний вираз>

│ │ ├── <доданок>

│ │ │ ├── <множник>

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

│ │ │ └── { (\*' | '/' | '%') <доданок> }

│ │ └── { ('+ | '-') <доданок> }

│ │

│ ├── <ввід>

│ │ ├── 'get'

│ │ └── <ідентифікатор>

│ │

│ ├── <вивід>

│ │ ├── 'put'

│ │ └── <ідентифікатор>

│ │

│ ├── <умовний оператор>

│ │ ├── 'if'

│ │ ├── <логічний вираз>

│ │ │ ├── <вираз І>

│ │ │ │ ├── <порівняння>

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

│ │ │ │ │ ├── <арифметичний вираз>

│ │ │ │ │ ├── <оператор порівняння> ('eq | 'noteq' | 'less' | 'gr')

│ │ │ │ │ └── <арифметичний вираз>

│ │ │ │ └── { 'or' <порівняння> }

│ │ │ └── { 'and' <вираз І> }

│ │ ├── <тіло> ‘;’

│ │ └── ['else' <тіло> ‘;’]

│ │

│ ├── <оператор циклу>

│ │ ├── 'while'

│ │ ├── <логічний вираз>

│ │ ├── <тіло циклу>

│ │ │ ├── <оператор>

│ │ │ └── { <оператор> }

│ │ ├── ['exit while'] (опціонально)

│ │ ├── ['continue while'] (опціонально)

│ │ └── 'endblok while'

└── 'endblok'

* + 1. **Розробка алгоритму роботи синтаксичного і семантичного аналізатора.**

Одним з найбільш простих і найбільш популярних методів низхідного синтаксичного аналізу є метод рекурсивного спуску (recursive descent method).

Метод заснований на тому, що в склад синтаксичного аналізатора входить множина рекурсивних процедур граматичного розбору, по одній для кожного правила граматики.

Визначимо назви процедур, що відповідають нетерміналам граматики таким чином:

Блок-схема алгоритму роботи синтаксичного аналізатора виглядатиме наступним чином:



Рис. 3.3. Блок-сема алгоритму роботи синтаксичного аналізатора

На етапі семантичного аналізу вирішуються такі задачі:

1. **Оголошення ідентифікаторів**: додавання інформації про ідентифікатори до таблиці.
2. **Використання ідентифікаторів**: перевірка їх коректності за таблицею.

Лексичний аналізатор передає атрибути лексеми (ім’я, тип, клас) семантичному аналізатору, який визначає, чи це оголошення або використання ідентифікатора.

### Основні функції

1. **getLastDataSectionLexemIndex**: знаходить останню лексему в секції даних. Повертає індекс або код помилки.
2. **checkingInternalCollisionInDeclarations**: виявляє дублювання ідентифікаторів, міток, чи їх конфлікти. Генерує помилки у разі порушень.
3. **checkingVariableInitialization**: перевіряє, чи всі змінні ініціалізовані перед використанням.
4. **checkingCollisionInDeclarationsByKeyWords**: виявляє збіги між ідентифікаторами та зарезервованими словами.
5. **semantixAnalyze**: координує всі перевірки, генерує коди стану та повідомлення про помилки.

### Ключові аспекти

* Робота з таблицею лексем і граматикою для аналізу.
* Використання регулярних виразів для перевірки ключових слів.
* Виведення детальних помилок у консоль.
* Повернення кодів стану для успішності чи помилок.

### Типовий процес

1. Перевірка декларацій, ініціалізації змінних, колізій та ключових слів.
2. У разі помилки — генерування коду стану та виведення повідомлення.

Алгоритм забезпечує виявлення помилок у логіці та структурах програми до її виконання.



*Рис. 3.4. Графічне представлення роботи семантичного аналізатора*

**3.4.3 Опис програми реалізації синтаксичного та семантичного аналізатора.**Синтаксичний аналізатор отримує на вхід таблицю лексем, сформовану під час лексичного аналізу. Він проходить через цю таблицю, перевіряючи, чи відповідає послідовність лексем визначеним правилам, заданим у формі нотації Бекуса-Наура. У разі виявлення невідповідностей інформація про помилку та її місце (рядок, де вона трапилася) записується у файл з помилками.

При опрацюванні операторів присвоєння або математичних виразів виконується перевірка балансу дужок: кількість відкритих дужок повинна дорівнювати кількості закритих. Крім того, аналізатор перевіряє, щоб не з'являлися підряд кілька лексем одного типу.

Результатом роботи аналізатора є синтаксичне дерево, яке містить посилання на таблиці об'єктів. У процесі побудови дерева також виявляються помилки, пов’язані зі структурою програми. Основу роботи аналізатора становить розпізнавання вхідного тексту програми відповідно до граматики мови.

* 1. Розробка генератора коду.

Синтаксичне дерево відображає лише структуру програми, проте для генерації коду необхідно також зберігати інформацію про змінні (їхні адреси), процедури (рівні, адреси) та мітки. Існують два основні методи організації цих даних:

* збереження у спеціальних таблицях генератора коду;
* включення відповідної інформації безпосередньо у вузли синтаксичного дерева.

Наприклад, якщо використовується Лідер-представлення, яке не містить адрес змінних, ці адреси потрібно створювати під час аналізу оголошень та зберігати у таблицях. Аналогічним чином обробляються описи складних структур даних, таких як масиви чи записи. Крім того, таблиці повинні містити інформацію про процедури: їхнє розташування в пам’яті, рівні вкладеності та модулі, в яких вони оголошені. При вході в процедуру у таблицю рівнів додається новий запис, що містить посилання на відповідні описи, а при виході цей вказівник повертається до попереднього значення. Якщо використовується дерево як проміжне представлення, то всі необхідні дані можна зберігати безпосередньо у його вузлах.

Генерація коду — це етап компіляції, який залежить від архітектури та створює машинний еквівалент вихідної програми. На вхід генератора зазвичай надходить проміжна форма представлення програми, а на виході формується об’єктний код або завантажувальний модуль.

Генератор асемблерного коду працює з масивом лексем, що не містить помилок. Якщо на попередніх етапах компіляції було виявлено помилки, цей процес не виконується. У межах даного курсового проєкту етап генерації коду реалізовано окремо, і його виконання можливе лише після успішного завершення синтаксичного аналізу.

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

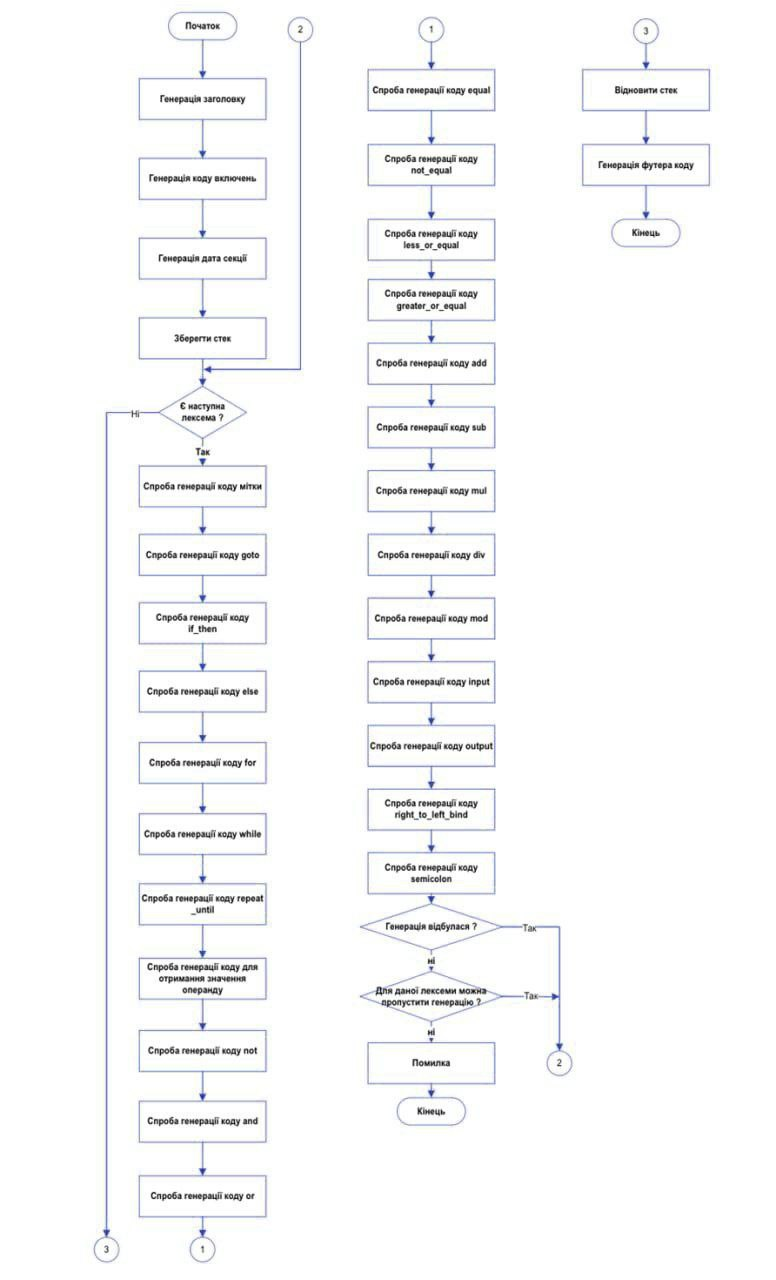


Рис. 3.5.1 Блок схема генератора коду

* + 1. **Опис програми реалізації генератора коду.**

Генератор коду — це програмний модуль, який перетворює внутрішнє представлення програми у виконуваний машинний код. Його основна функція полягає в обробці токенів, створенні інструкцій та управлінні пам’яттю.

### **Основні компоненти генератора коду:**

1. **Структури даних:**
   * Збереження інформації про мітки та їх розташування
   * Дані про переходи та їхні адреси
   * Структури для роботи з токенами та їх характеристиками
2. **Функціональні модулі:**
   * Генерація заголовка програми
   * Формування секції оголошень та залежностей
   * Створення секції даних
   * Ініціалізація стеку та управління пам’яттю
   * Генерація початкового та завершального коду
3. **Обробка токенів:**
   * Аналіз і класифікація токенів
   * Обробка конструкцій мови програмування
   * Генерація відповідних машинних команд
4. **Генерація коду для різних типів операцій:**
   * Арифметичні операції
   * Логічні операції
   * Керуючі конструкції (умовні переходи, цикли)
   * Операції введення/виведення
   * Робота з мітками та адресація
5. **Системні параметри:**
   * Визначення розмірів буферів та області пам’яті
   * Налаштування зміщень даних
   * Конфігурація режимів генерації коду
6. **Обробка помилок:**
   * Перевірка правильності токенів
   * Валідація синтаксичних конструкцій
   * Генерація повідомлень про помилки

### **Етапи роботи генератора коду:**

1. **Ініціалізація:**
   * Налаштування початкових параметрів
   * Підготовка необхідних структур даних
   * Розмітка секцій коду та даних
2. **Основний процес генерації:**
   * Поетапний аналіз токенів
   * Визначення типу кожної конструкції
   * Формування відповідних машинних команд
3. **Завершення роботи:**
   * Генерація фінальних інструкцій
   * Відновлення стану системи
   * Фіналізація та підготовка вихідного коду

### **Режими роботи генератора:**

* Генерація виконуваного машинного коду
* Формування асемблерного коду
* Створення об’єктного коду
* Режим налагодження

Результатом роботи генератора є оптимізований машинний код, готовий до виконання на цільовій платформі. Він забезпечує ефективне перетворення високорівневих конструкцій мови програмування у низькорівневі інструкції процесора.

1. Налагодження та тестування розробленого транслятора

Тестування програмного забезпечення є ключовим етапом у процесі розробки, оскільки дозволяє виявити помилки, допущені на попередніх стадіях. Воно сприяє покращенню якості продукту, зокрема вдосконаленню інтерфейсу та усуненню можливих слабких місць.

Відлагодження цієї програми здійснюється за допомогою набору тестових програм, що відповідають визначеній граматиці. Основна увага приділяється перевірці правильності генерації коду, коректності виявлення помилок та правильності поділу на лексеми.

* 1. Опис інтерфейсу та інструкції користувачу.

Програма підтримує два режими роботи: **інтерактивний**, що дозволяє виконувати завдання покроково, та **автоматичний**, у якому команди виконуються відповідно до параметрів командного рядка. Її основне призначення — аналіз і обробка вихідного коду через такі етапи:

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

### **Інтерактивний режим**

Якщо параметри командного рядка відсутні, програма запускається в інтерактивному режимі, дозволяючи користувачеві виконувати всі етапи покроково:

1. **Лексичний аналіз** – введіть y для виконання або n для пропуску.
2. **Синтаксичний аналіз** – підтвердіть виконання натисканням y.
3. **Семантичний аналіз** – аналогічно підтверджується користувачем.
4. **Генерація машинного коду** – завершує процес обробки.

### **Інструкція користувача**

1. **Завантаження вихідного файлу**
   * Переконайтеся, що вихідний код підготовлений та вказаний за допомогою параметра --input.
2. **Діагностика помилок**
   * Якщо виявлено помилки, програма виведе відповідні повідомлення у консоль або збереже їх у файл, якщо задано параметр --output.
3. **Завершення роботи**
   * Після успішного проходження всіх етапів буде виведено повідомлення про завершення роботи, а результати збережено у відповідних файлах.

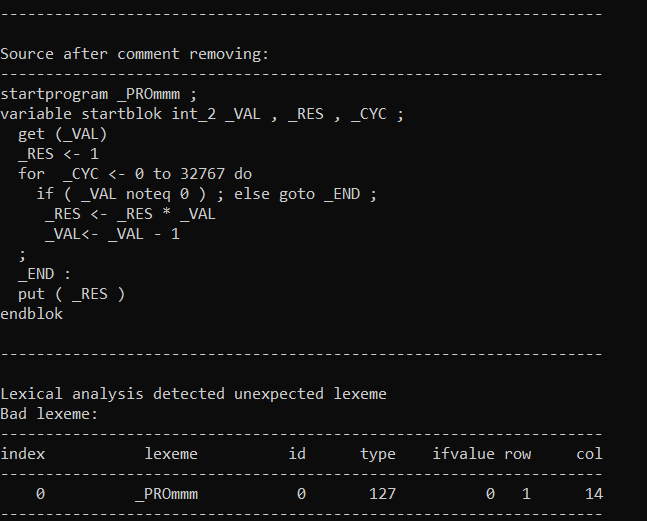
### **Помилки та способи їх усунення**

* **Порожній вихідний файл**
  + Програма повідомить про відсутність вихідного коду та припинить роботу.
* **Синтаксичні та семантичні помилки**
  + Усі виявлені помилки будуть виведені у консоль або записані у файл для подальшого аналізу.
  1. Виявлення лексичних і синтаксичних помилок.

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

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

Якщо зробити у програмі синтаксичну помилку то програма вкаже її:



*Рис. 4.2. Вивід інформації про синтаксичну помилку.*

* 1. Перевірка роботи транслятора за допомогою тестових задач.

Тестова програма №1 *Лінійний алгоритм*

Х = (А - В) \* 10 + (А + В) / 10

***Текст програми***

startprogram \_PRO;

variable startblok int\_2 \_VAL , \_RES , \_CYC;

get (\_VAL)

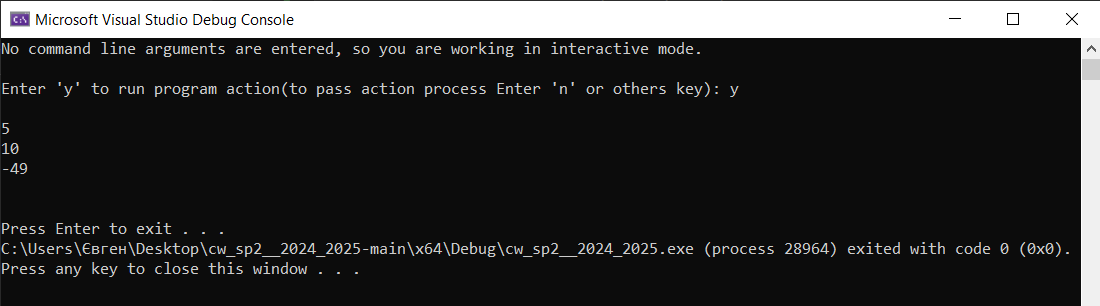
get (\_RES)

\_CYC <- 10 \* (\_VAL - \_RES) + (\_VAL + \_RES) / 10

put ( \_CYC )

endblok

Результат:



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

Тестова програма №2

Тестова програма «Алгоритм з розгалуженням»

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

Використання вкладеного умовного оператора:

Знайти найбільше з них і вивести його на екран.

Використання простого умовного оператора:

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

***Текст програми***

startprogram \_PRO ;

variable startblok int\_2 \_AVV , \_BVV , \_CVV ;

get (\_AVV)

get (\_BVV)

get (\_CVV)

if ( \_AVV eq \_BVV ) ; else goto \_CAL ;

goto \_BAL

\_CAL :

put ( 0 )

goto \_ENA

\_BAL :

if ( \_AVV eq \_CVV ) ; else goto \_CAL;

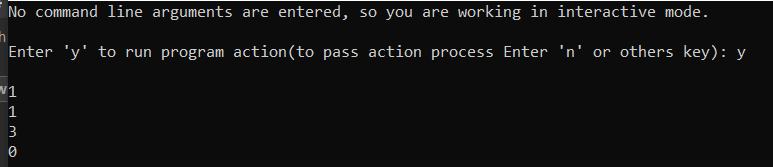
put ( 1 )

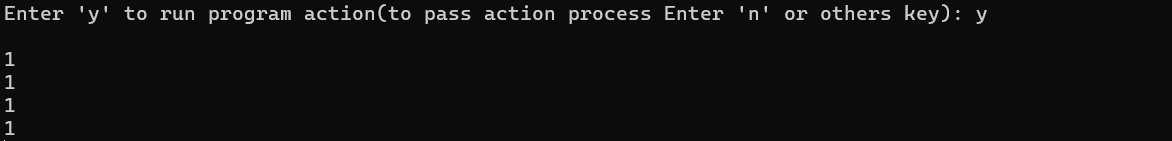
\_ENA:

get (\_AVV)

endblok

Отримую такі результати:

******



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

**Висновки**  
 У рамках курсового проекту було створено транслятор для обробки вихідного коду програмування, який виконує кілька важливих етапів:

1. **Лексичний аналіз:**
   * Лексичний аналізатор відповідає за поділ тексту програми на окремі лексеми, з подальшим формуванням таблиці, яка містить їх типи, значення та номери рядків.
   * Алгоритм лексичного аналізу ґрунтується на скінченному автоматі, здатному ідентифікувати ключові слова, ідентифікатори, константи, оператори та розділові знаки.
2. **Синтаксичний і семантичний аналіз:**
   * Синтаксичний аналізатор перевіряє правильність структури програми за допомогою граматики, створюючи дерево розбору та таблиці, що містять ідентифікатори та їх типи.
   * Семантичний аналізатор перевіряє логічну коректність програми, зокрема відповідність типів даних, правильність області видимості змінних та коректність викликів функцій.
3. **Генерація коду:**
   * Генератор коду перетворює абстрактне синтаксичне дерево на вихідний код, використовуючи мову програмування С, обходячи дерево та генеруючи відповідний код для кожного вузла дерева.
4. **Тестування:**
   * Процес тестування включав перевірку різних типів програм, таких як лінійні, з розгалуженням та циклічні. Це дозволило виявити і виправити лексичні, синтаксичні та семантичні помилки.
   * Транслятор успішно генерує коректний код на основі заданих вхідних програм.

**Переваги проекту:**

* Поетапне виконання трансляції забезпечує високий рівень контролю на кожному етапі.
* Модульна архітектура програми дозволяє легко розширювати та модифікувати систему.
* Ретельно проведене тестування підтвердило стабільність і надійність роботи транслятора в різних умовах.

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. **Теорія формальних мов і автоматів**: Навчальний посібник / Ю. М. Крижановський, І. В. Гапонова. – Харків: ХНУ, 2019. – 142 с.
3. **Вступ до теорії компіляторів**: Підручник / І. Л. Манойленко. – Київ: Наукова думка, 2020. – 234 с.
4. **Системне програмування та компілятори**: Частина 1: Теорія компіляції / І. С. Чернишов. – Львів: ЛНУ, 2017. – 120 с.
5. **Principles of Compiler Design**: Aho, A. V., Sethi, R., Ullman, J. D. – 2nd ed. – Boston: Addison-Wesley, 2006. – 624 p.
6. **Compilers: Principles, Techniques, and Tools (Dragon Book)**: Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D. – 2nd ed. – Boston: Pearson, 2006. – 932 p.

Додатки

**А. Таблиці лексем для тестових прикладів**

Тестова програма «лінійного алгоритму»

Lexemes table:

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

index lexeme id type ifvalue row col

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

0 startprogram 286 1 0 1 1

1 \_PRO 0 2 0 1 14

2 ; 256 1 0 1 18

3 variable 299 1 0 2 1

4 startblok 308 1 0 2 10

5 int\_2 416 1 0 2 21

6 \_VAL 1 2 0 2 27

7 , 267 1 0 2 32

8 \_RES 2 2 0 2 34

9 , 267 1 0 2 32

10 \_CYC 3 2 0 2 41

11 ; 256 1 0 2 45

12 get 340 1 0 3 3

13 ( 281 1 0 3 7

14 \_VAL 1 2 0 3 8

15 ) 284 1 0 3 12

16 get 340 1 0 4 3

17 ( 281 1 0 4 7

18 \_RES 2 2 0 4 8

19 ) 284 1 0 4 12

20 \_CYC 3 2 0 5 3

21 <- 258 1 0 5 8

22 10 320 4 10 5 11

23 \* 265 1 0 5 14

24 ( 281 1 0 5 17

25 \_VAL 1 2 0 5 18

26 - 259 1 0 5 23

27 \_RES 2 2 0 5 25

28 ) 284 1 0 5 29

29 + 261 1 0 5 31

30 ( 281 1 0 5 33

31 \_VAL 1 2 0 5 34

32 + 261 1 0 5 31

33 \_RES 2 2 0 5 41

34 ) 284 1 0 5 45

35 / 397 1 0 5 47

36 10 320 4 10 5 49

37 put 344 1 0 6 1

38 ( 281 1 0 6 5

39 \_CYC 3 2 0 6 7

40 ) 284 1 0 6 12

41 endblok 318 1 0 8 1

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

Тестова програма «Алгоритм з розгалуженням»

Lexemes table:

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

index lexeme id type ifvalue row col

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

0 startprogram 286 1 0 1 1

1 \_PRO 0 2 0 1 14

2 ; 256 1 0 1 19

3 variable 299 1 0 2 1

4 startblok 308 1 0 2 10

5 int\_2 416 1 0 2 20

6 \_AVV 1 2 0 2 26

7 , 267 1 0 2 31

8 \_BVV 2 2 0 2 33

9 , 267 1 0 2 38

10 \_CVV 3 2 0 2 40

11 ; 256 1 0 2 45

12 get 340 1 0 3 3

13 ( 281 1 0 3 7

14 \_AVV 1 2 0 3 8

15 ) 284 1 0 3 12

16 get 340 1 0 4 3

17 ( 281 1 0 4 7

18 \_BVV 2 2 0 4 8

19 ) 284 1 0 4 12

20 get 340 1 0 5 4

21 ( 281 1 0 5 8

22 \_CVV 3 2 0 5 9

23 ) 284 1 0 5 13

24 if 348 1 0 6 5

25 ( 281 1 0 6 8

26 \_AVV 1 2 0 6 10

27 eq 269 1 0 6 15

28 \_BVV 2 2 0 6 18

29 ) 284 1 0 6 23

30 ; 256 1 0 6 25

31 else 351 1 0 6 27

32 goto 392 1 0 6 32

33 \_CAL 4 2 0 6 37

34 ; 256 1 0 6 42

35 goto 392 1 0 6 32

36 \_BAL 5 2 0 7 12

37 \_CAL 4 2 0 8 2

38 : 278 1 0 8 7

39 put 344 1 0 9 3

40 ( 281 1 0 9 7

41 0 320 4 0 9 9

42 ) 284 1 0 9 11

43 goto 392 1 0 10 3

44 \_ENA 6 2 0 10 8

45 \_BAL 5 2 0 11 4

46 : 278 1 0 11 9

47 if 348 1 0 12 6

48 ( 281 1 0 12 9

49 \_AVV 1 2 0 12 11

50 eq 269 1 0 12 16

51 \_CVV 3 2 0 12 19

52 ) 284 1 0 12 24

53 ; 256 1 0 12 26

54 else 351 1 0 12 28

55 goto 392 1 0 12 33

56 \_CAL 4 2 0 12 38

57 ; 256 1 0 12 42

58 put 344 1 0 13 7

59 ( 281 1 0 13 11

60 1 320 4 1 13 13

61 ) 284 1 0 13 15

62 \_ENA 6 2 0 14 1

63 : 278 1 0 14 5

64 get 340 1 0 15 3

65 ( 281 1 0 15 7

66 \_AVV 1 2 0 15 8

67 ) 284 1 0 15 12

68 endblok 318 1 0 16 1

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

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

.686

.model flat, stdcall

option casemap : none

GetStdHandle proto STDCALL, nStdHandle : DWORD

ExitProcess proto STDCALL, uExitCode : DWORD

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

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

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

wsprintfA PROTO C : VARARG

GetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, lpMode : DWORD

SetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, dwMode : DWORD

ENABLE\_LINE\_INPUT EQU 0002h

ENABLE\_ECHO\_INPUT EQU 0004h

.data

data\_start db 8192 dup (0)

;title\_msg db "Output:", 0

valueTemp\_msg db 256 dup(0)

valueTemp\_fmt db "%d", 10, 13, 0

;NumberOfCharsWritten dd 0

hConsoleInput dd 0

hConsoleOutput dd 0

buffer db 128 dup(0)

readOutCount dd ?

.code

start:

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

;NexInstruction:

pop esi

sub esi, 5

mov edi, esi

add edi, 000004000h

mov ecx, edi

add ecx, 512

jmp initConsole

putProc PROC

push eax

push offset valueTemp\_fmt

push offset valueTemp\_msg

call wsprintfA

add esp, 12

;push 40h

;push offset title\_msg

;push offset valueTemp\_msg;

;push 0

;call MessageBoxA

push 0

push 0; offset NumberOfCharsWritten

push eax; NumberOfCharsToWrite

push offset valueTemp\_msg

push hConsoleOutput

call WriteConsoleA

ret

putProc ENDP

getProc PROC

push ebp

mov ebp, esp

push 0

push offset readOutCount

push 15

push offset buffer + 1

push hConsoleInput

call ReadConsoleA

lea esi, offset buffer

add esi, readOutCount

sub esi, 2

call string\_to\_int

mov esp, ebp

pop ebp

ret

getProc ENDP

string\_to\_int PROC

; input: ESI - string

; output: EAX - value

xor eax, eax

mov ebx, 1

xor ecx, ecx

convert\_loop :

movzx ecx, byte ptr[esi]

test ecx, ecx

jz done

sub ecx, '0'

imul ecx, ebx

add eax, ecx

imul ebx, ebx, 10

dec esi

jmp convert\_loop

done:

ret

string\_to\_int ENDP

initConsole:

push -10

call GetStdHandle

mov hConsoleInput, eax

push -11

call GetStdHandle

mov hConsoleOutput, eax

;push ecx

;push ebx

;push esi

;push edi

;push offset mode

;push hConsoleInput

;call GetConsoleMode

;mov ebx, eax

;or ebx, ENABLE\_LINE\_INPUT

;or ebx, ENABLE\_ECHO\_INPUT

;push ebx

;push hConsoleInput

;call SetConsoleMode

;pop edi

;pop esi

;pop ebx

;pop ecx

;hw stack save(save esp)

mov ebp, esp

;";"

;"4"

add ecx, 4

mov eax, 000000004h

mov dword ptr [ecx], eax

;"get"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"8"

add ecx, 4

mov eax, 000000008h

mov dword ptr [ecx], eax

;"get"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"12"

add ecx, 4

mov eax, 00000000Ch

mov dword ptr [ecx], eax

;"10"

add ecx, 4

mov eax, 00000000Ah

mov dword ptr [ecx], eax

;"\_VAL"

mov eax, edi

add eax, 000000004h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"\_RES"

mov eax, edi

add eax, 000000008h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"-"

mov eax, dword ptr[ecx]

sub ecx, 4

sub dword ptr[ecx], eax

mov eax, dword ptr[ecx]

;"\*"

mov eax, dword ptr[ecx - 4]

;cdq

imul dword ptr [ecx]

sub ecx, 4

mov dword ptr [ecx], eax

;"\_VAL"

mov eax, edi

add eax, 000000004h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"\_RES"

mov eax, edi

add eax, 000000008h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"+"

mov eax, dword ptr[ecx]

sub ecx, 4

add dword ptr[ecx], eax

mov eax, dword ptr[ecx]

;"10"

add ecx, 4

mov eax, 00000000Ah

mov dword ptr [ecx], eax

;"/"

mov eax, dword ptr[ecx - 4]

cdq

idiv dword ptr [ecx]

sub ecx, 4

mov dword ptr [ecx], eax

;"+"

mov eax, dword ptr[ecx]

sub ecx, 4

add dword ptr[ecx], eax

mov eax, dword ptr[ecx]

;"<-"

mov eax, dword ptr[ecx]

mov ebx, dword ptr[ecx - 4]

sub ecx, 8

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"\_CYC"

mov eax, edi

add eax, 00000000Ch

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"put"

mov eax, dword ptr[ecx]

mov edx, 00000001Bh

add edx, esi

;push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

;pop ecx

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;hw stack reset(restore esp)

mov esp, ebp

xor eax, eax

ret

end start

File "../test\_programs/file1.asm" saved.

Тестова програма <<Алгоритм з розгалуженням>>  
.686

.model flat, stdcall

option casemap : none

GetStdHandle proto STDCALL, nStdHandle : DWORD

ExitProcess proto STDCALL, uExitCode : DWORD

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

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

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

wsprintfA PROTO C : VARARG

GetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, lpMode : DWORD

SetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, dwMode : DWORD

ENABLE\_LINE\_INPUT EQU 0002h

ENABLE\_ECHO\_INPUT EQU 0004h

.data

data\_start db 8192 dup (0)

;title\_msg db "Output:", 0

valueTemp\_msg db 256 dup(0)

valueTemp\_fmt db "%d", 10, 13, 0

;NumberOfCharsWritten dd 0

hConsoleInput dd 0

hConsoleOutput dd 0

buffer db 128 dup(0)

readOutCount dd ?

.code

start:

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

;NexInstruction:

pop esi

sub esi, 5

mov edi, esi

add edi, 000004000h

mov ecx, edi

add ecx, 512

jmp initConsole

putProc PROC

push eax

push offset valueTemp\_fmt

push offset valueTemp\_msg

call wsprintfA

add esp, 12

;push 40h

;push offset title\_msg

;push offset valueTemp\_msg;

;push 0

;call MessageBoxA

push 0

push 0; offset NumberOfCharsWritten

push eax; NumberOfCharsToWrite

push offset valueTemp\_msg

push hConsoleOutput

call WriteConsoleA

ret

putProc ENDP

getProc PROC

push ebp

mov ebp, esp

push 0

push offset readOutCount

push 15

push offset buffer + 1

push hConsoleInput

call ReadConsoleA

lea esi, offset buffer

add esi, readOutCount

sub esi, 2

call string\_to\_int

mov esp, ebp

pop ebp

ret

getProc ENDP

string\_to\_int PROC

; input: ESI - string

; output: EAX - value

xor eax, eax

mov ebx, 1

xor ecx, ecx

convert\_loop :

movzx ecx, byte ptr[esi]

test ecx, ecx

jz done

sub ecx, '0'

imul ecx, ebx

add eax, ecx

imul ebx, ebx, 10

dec esi

jmp convert\_loop

done:

ret

string\_to\_int ENDP

initConsole:

push -10

call GetStdHandle

mov hConsoleInput, eax

push -11

call GetStdHandle

mov hConsoleOutput, eax

;push ecx

;push ebx

;push esi

;push edi

;push offset mode

;push hConsoleInput

;call GetConsoleMode

;mov ebx, eax

;or ebx, ENABLE\_LINE\_INPUT

;or ebx, ENABLE\_ECHO\_INPUT

;push ebx

;push hConsoleInput

;call SetConsoleMode

;pop edi

;pop esi

;pop ebx

;pop ecx

;hw stack save(save esp)

mov ebp, esp

;";"

;"4"

add ecx, 4

mov eax, 000000004h

mov dword ptr [ecx], eax

;"get"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"8"

add ecx, 4

mov eax, 000000008h

mov dword ptr [ecx], eax

;"get"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"12"

add ecx, 4

mov eax, 00000000Ch

mov dword ptr [ecx], eax

;"get"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"if"

;"\_AVV"

mov eax, edi

add eax, 000000004h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"\_BVV"

mov eax, edi

add eax, 000000008h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"eq"

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\_00007FF7429399B0

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

mov eax, 1

LABEL@AFTER\_THEN\_00007FF7429399B0:

;"else"

cmp eax, 0

jnz LABEL@AFTER\_ELSE\_00007FF74293A628

;"goto" previous ident "\_CAL"(as label)

jmp LABEL@00000198BFB6AD68

;null statement (non-context)

;";" (after "else")

LABEL@AFTER\_ELSE\_00007FF74293A628:

;"goto" previous ident "\_BAL"(as label)

jmp LABEL@00000198BFB6B7F8

;null statement (non-context)

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

LABEL@00000198BFB6AD68:

;"0"

add ecx, 4

mov eax, 000000000h

mov dword ptr [ecx], eax

;"put"

mov eax, dword ptr[ecx]

mov edx, 00000001Bh

add edx, esi

;push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

;pop ecx

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;"goto" previous ident "\_ENA"(as label)

jmp LABEL@00000198BFB67868

;null statement (non-context)

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

LABEL@00000198BFB6B7F8:

;"if"

;"\_AVV"

mov eax, edi

add eax, 000000004h

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"\_CVV"

mov eax, edi

add eax, 00000000Ch

mov eax, dword ptr[eax]

add ecx, 4

mov dword ptr [ecx], eax

;"eq"

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\_00007FF7429422D8

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

mov eax, 1

LABEL@AFTER\_THEN\_00007FF7429422D8:

;"else"

cmp eax, 0

jnz LABEL@AFTER\_ELSE\_00007FF742942F50

;"goto" previous ident "\_CAL"(as label)

jmp LABEL@00000198BFB6AD68

;null statement (non-context)

;";" (after "else")

LABEL@AFTER\_ELSE\_00007FF742942F50:

;"1"

add ecx, 4

mov eax, 000000001h

mov dword ptr [ecx], eax

;"put"

mov eax, dword ptr[ecx]

mov edx, 00000001Bh

add edx, esi

;push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

;pop ecx

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

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

LABEL@00000198BFB67868:

;"4"

add ecx, 4

mov eax, 000000004h

mov dword ptr [ecx], eax

;"get"

mov eax, dword ptr[ecx]

mov edx, 000000044h

add edx, esi

push ecx

;push ebx

push esi

push edi

call edx

pop edi

pop esi

;pop ebx

pop ecx

mov ebx, dword ptr[ecx]

sub ecx, 4

add ebx, edi

mov dword ptr [ebx], eax

mov ecx, edi ; reset second stack

add ecx, 512 ; reset second stack

;null statement (non-context)

;hw stack reset(restore esp)

mov esp, ebp

xor eax, eax

ret

end start

File "../test\_programs/file1.asm" saved.

**Додаток В. Абстрактне синтаксичне дерево для тестових прикладів**Тестова програма «Лінійний алгоритм»  
 |--program

| |--program\_\_\_\_part1

| | |--tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY

| | | |--tokenNAME\_\_program\_name

| | | | |--tokenNAME

| | | | | |--"startprogram"

| | | | |--program\_name

| | | | | |--"\_PRO"

| | | |--tokenSEMICOLON\_\_tokenBODY

| | | | |--tokenSEMICOLON

| | | | | |--";"

| | | | |--tokenBODY

| | | | | |--"startblok"

| | |--tokenDATA\_\_declaration

| | | |--tokenDATA

| | | | |--"variable"

| | | |--declaration

| | | | |--value\_type\_\_ident

| | | | | |--value\_type

| | | | | | |--"int\_2"

| | | | | |--ident

| | | | | | |--"\_VAL"

| | | | |--other\_declaration\_ident\_\_\_\_iteration\_after\_one

| | | | | |--other\_declaration\_ident

| | | | | | |--tokenCOMMA

| | | | | | | |--","

| | | | | | |--ident

| | | | | | | |--"\_RES"

| | | | | |--other\_declaration\_ident\_\_\_\_iteration\_after\_one

| | | | | | |--tokenCOMMA

| | | | | | | |--","

| | | | | | |--ident

| | | | | | | |--"\_CYC"

| |--program\_\_\_\_part2

| | |--tokenSEMICOLON

| | | |--";"

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

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

| | | | |--statement

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

| | | | | | |--tokenGET

| | | | | | | |--"get"

| | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

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

| | | | | | |--ident

| | | | | | | |--"\_VAL"

| | | | | | |--tokenGROUPEXPRESSIONEND

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

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

| | | | | |--statement

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

| | | | | | | |--tokenGET

| | | | | | | | |--"get"

| | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

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

| | | | | | | |--ident

| | | | | | | | |--"\_RES"

| | | | | | | |--tokenGROUPEXPRESSIONEND

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

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

| | | | | | |--statement

| | | | | | | |--ident

| | | | | | | | |--"\_CYC"

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

| | | | | | | | |--tokenRLBIND

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

| | | | | | | | |--expression

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

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

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

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

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

| | | | | | | | | | | | |--"\*"

| | | | | | | | | | | |--expression

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

| | | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

| | | | | | | | | | | | | |--expression

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

| | | | | | | | | | | | | | | |--"\_VAL"

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

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

| | | | | | | | | | | | | | | | |--"-"

| | | | | | | | | | | | | | | |--expression

| | | | | | | | | | | | | | | | |--"\_RES"

| | | | | | | | | | | | |--tokenGROUPEXPRESSIONEND

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

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

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

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

| | | | | | | | | | | | | |--"+"

| | | | | | | | | | | | |--expression

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

| | | | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

| | | | | | | | | | | | | | |--expression

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

| | | | | | | | | | | | | | | | |--"\_VAL"

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

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

| | | | | | | | | | | | | | | | | |--"+"

| | | | | | | | | | | | | | | | |--expression

| | | | | | | | | | | | | | | | | |--"\_RES"

| | | | | | | | | | | | | |--tokenGROUPEXPRESSIONEND

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

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

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

| | | | | | | | | | | | | |--"/"

| | | | | | | | | | | | |--expression

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

| | | | | | |--statement

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

| | | | | | | | |--tokenPUT

| | | | | | | | | |--"put"

| | | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

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

| | | | | | | | |--expression

| | | | | | | | | |--"\_CYC"

| | | | | | | | |--tokenGROUPEXPRESSIONEND

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

| | | |--tokenEND

| | | | |--"endblok"

Тестова програма «Циклічний алгоритм»

|--program

| |--program\_\_\_\_part1

| | |--tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY

| | | |--tokenNAME\_\_program\_name

| | | | |--tokenNAME

| | | | | |--"startprogram"

| | | | |--program\_name

| | | | | |--"\_PRO"

| | | |--tokenSEMICOLON\_\_tokenBODY

| | | | |--tokenSEMICOLON

| | | | | |--";"

| | | | |--tokenBODY

| | | | | |--"startblok"

| | |--tokenDATA\_\_declaration

| | | |--tokenDATA

| | | | |--"variable"

| | | |--declaration

| | | | |--value\_type\_\_ident

| | | | | |--value\_type

| | | | | | |--"int\_2"

| | | | | |--ident

| | | | | | |--"\_AVV"

| | | | |--other\_declaration\_ident\_\_\_\_iteration\_after\_one

| | | | | |--other\_declaration\_ident

| | | | | | |--tokenCOMMA

| | | | | | | |--","

| | | | | | |--ident

| | | | | | | |--"\_BVV"

| | | | | |--other\_declaration\_ident\_\_\_\_iteration\_after\_one

| | | | | | |--tokenCOMMA

| | | | | | | |--","

| | | | | | |--ident

| | | | | | | |--"\_CVV"

| |--program\_\_\_\_part2

| | |--tokenSEMICOLON

| | | |--";"

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

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

| | | | |--statement

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

| | | | | | |--tokenGET

| | | | | | | |--"get"

| | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

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

| | | | | | |--ident

| | | | | | | |--"\_AVV"

| | | | | | |--tokenGROUPEXPRESSIONEND

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

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

| | | | | |--statement

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

| | | | | | | |--tokenGET

| | | | | | | | |--"get"

| | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

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

| | | | | | | |--ident

| | | | | | | | |--"\_BVV"

| | | | | | | |--tokenGROUPEXPRESSIONEND

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

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

| | | | | | |--statement

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

| | | | | | | | |--tokenGET

| | | | | | | | | |--"get"

| | | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

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

| | | | | | | | |--ident

| | | | | | | | | |--"\_CVV"

| | | | | | | | |--tokenGROUPEXPRESSIONEND

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

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

| | | | | | | |--statement

| | | | | | | | |--tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND

| | | | | | | | | |--tokenIF\_\_tokenGROUPEXPRESSIONBEGIN

| | | | | | | | | | |--tokenIF

| | | | | | | | | | | |--"if"

| | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

| | | | | | | | | |--expression\_\_tokenGROUPEXPRESSIONEND

| | | | | | | | | | |--expression

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

| | | | | | | | | | | | |--"\_AVV"

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

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

| | | | | | | | | | | | | |--"eq"

| | | | | | | | | | | | |--expression

| | | | | | | | | | | | | |--"\_BVV"

| | | | | | | | | | |--tokenGROUPEXPRESSIONEND

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

| | | | | | | | |--body\_for\_true\_\_body\_for\_false

| | | | | | | | | |--body\_for\_true

| | | | | | | | | | |--";"

| | | | | | | | | |--body\_for\_false

| | | | | | | | | | |--tokenELSE\_\_statement\_in\_while\_body

| | | | | | | | | | | |--tokenELSE

| | | | | | | | | | | | |--"else"

| | | | | | | | | | | |--statement\_in\_while\_body

| | | | | | | | | | | | |--tokenGOTO

| | | | | | | | | | | | | |--"goto"

| | | | | | | | | | | | |--ident

| | | | | | | | | | | | | |--"\_CAL"

| | | | | | | | | | |--tokenSEMICOLON

| | | | | | | | | | | |--";"

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

| | | | | | | | |--statement

| | | | | | | | | |--tokenGOTO

| | | | | | | | | | |--"goto"

| | | | | | | | | |--ident

| | | | | | | | | | |--"\_BAL"

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

| | | | | | | | | |--statement

| | | | | | | | | | |--ident

| | | | | | | | | | | |--"\_CAL"

| | | | | | | | | | |--tokenCOLON

| | | | | | | | | | | |--":"

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

| | | | | | | | | | |--statement

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

| | | | | | | | | | | | |--tokenPUT

| | | | | | | | | | | | | |--"put"

| | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

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

| | | | | | | | | | | | |--expression

| | | | | | | | | | | | | |--"0"

| | | | | | | | | | | | |--tokenGROUPEXPRESSIONEND

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

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

| | | | | | | | | | | |--statement

| | | | | | | | | | | | |--tokenGOTO

| | | | | | | | | | | | | |--"goto"

| | | | | | | | | | | | |--ident

| | | | | | | | | | | | | |--"\_ENA"

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

| | | | | | | | | | | | |--statement

| | | | | | | | | | | | | |--ident

| | | | | | | | | | | | | | |--"\_BAL"

| | | | | | | | | | | | | |--tokenCOLON

| | | | | | | | | | | | | | |--":"

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

| | | | | | | | | | | | | |--statement

| | | | | | | | | | | | | | |--tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND

| | | | | | | | | | | | | | | |--tokenIF\_\_tokenGROUPEXPRESSIONBEGIN

| | | | | | | | | | | | | | | | |--tokenIF

| | | | | | | | | | | | | | | | | |--"if"

| | | | | | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

| | | | | | | | | | | | | | | |--expression\_\_tokenGROUPEXPRESSIONEND

| | | | | | | | | | | | | | | | |--expression

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

| | | | | | | | | | | | | | | | | | |--"\_AVV"

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

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

| | | | | | | | | | | | | | | | | | | |--"eq"

| | | | | | | | | | | | | | | | | | |--expression

| | | | | | | | | | | | | | | | | | | |--"\_CVV"

| | | | | | | | | | | | | | | | |--tokenGROUPEXPRESSIONEND

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

| | | | | | | | | | | | | | |--body\_for\_true\_\_body\_for\_false

| | | | | | | | | | | | | | | |--body\_for\_true

| | | | | | | | | | | | | | | | |--";"

| | | | | | | | | | | | | | | |--body\_for\_false

| | | | | | | | | | | | | | | | |--tokenELSE\_\_statement\_in\_while\_body

| | | | | | | | | | | | | | | | | |--tokenELSE

| | | | | | | | | | | | | | | | | | |--"else"

| | | | | | | | | | | | | | | | | |--statement\_in\_while\_body

| | | | | | | | | | | | | | | | | | |--tokenGOTO

| | | | | | | | | | | | | | | | | | | |--"goto"

| | | | | | | | | | | | | | | | | | |--ident

| | | | | | | | | | | | | | | | | | | |--"\_CAL"

| | | | | | | | | | | | | | | | |--tokenSEMICOLON

| | | | | | | | | | | | | | | | | |--";"

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

| | | | | | | | | | | | | | |--statement

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

| | | | | | | | | | | | | | | | |--tokenPUT

| | | | | | | | | | | | | | | | | |--"put"

| | | | | | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

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

| | | | | | | | | | | | | | | | |--expression

| | | | | | | | | | | | | | | | | |--"1"

| | | | | | | | | | | | | | | | |--tokenGROUPEXPRESSIONEND

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

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

| | | | | | | | | | | | | | | |--statement

| | | | | | | | | | | | | | | | |--ident

| | | | | | | | | | | | | | | | | |--"\_ENA"

| | | | | | | | | | | | | | | | |--tokenCOLON

| | | | | | | | | | | | | | | | | |--":"

| | | | | | | | | | | | | | | |--statement

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

| | | | | | | | | | | | | | | | | |--tokenGET

| | | | | | | | | | | | | | | | | | |--"get"

| | | | | | | | | | | | | | | | | |--tokenGROUPEXPRESSIONBEGIN

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

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

| | | | | | | | | | | | | | | | | |--ident

| | | | | | | | | | | | | | | | | | |--"\_AVV"

| | | | | | | | | | | | | | | | | |--tokenGROUPEXPRESSIONEND

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

| | | |--tokenEND

| | | | |--"endblok"

**Додаток Г.** **Документований текст програмних модулів (лістинги)**

Config.h  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: config.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

//#define LEXICAL\_ANALISIS\_MODE 1

//#define SEMANTIC\_ANALISIS\_MODE 2

//#define FULL\_COMPILER\_MODE 4

//#define DEBUG\_MODE 512

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

//#define DEFAULT\_MODE (DEBUG\_MODE | LEXICAL\_ANALISIS\_MODE | SYNTAX\_ANALISIS\_MODE | SEMANTIC\_ANALISIS\_MODE | MAKE\_ASSEMBLY | MAKE\_BINARY)

#define TOKENS\_RE ";|<-|\\+|-|\\\*|,|eq|noteq|:|\\(|\\)|less|gr|[\_0-9A-Za-z]+|[^ \t\r\f\v\n]"

#define KEYWORDS\_RE ";|<-|\\+|-|\\\*|,|eq|noteq|:|\\(|\\)|startprogram|variable|startblok|endblok|EXIT|CONTINUE|get|put|if|else|for|to|downto|do|while|repeat|until|goto|/|%|<=|>=|!|and|or|int\_2"

#define IDENTIFIERS\_RE "\_[A-Z][A-Z][A-Z]"

#define UNSIGNEDVALUES\_RE "0|[1-9][0-9]\*"

// first column of the cw term paper option

#define PROGRAM\_FORMAT \

{"tokenNAME\_\_program\_name", 2, {"tokenNAME","program\_name"}},\

{"tokenSEMICOLON\_\_tokenBODY", 2, {"tokenSEMICOLON","tokenBODY"}},\

{"tokenDATA\_\_declaration", 2, {"tokenDATA","declaration"}},\

{"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY", 2, {"tokenNAME\_\_program\_name","tokenSEMICOLON\_\_tokenBODY"}},\

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA\_\_declaration"}},\

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA"}},\

{"statement\_\_tokenEND", 2, {"statement","tokenEND"}},\

{"statement\_\_\_\_iteration\_after\_two\_\_tokenEND", 2, {"statement\_\_\_\_iteration\_after\_two","tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_\_\_iteration\_after\_two\_\_tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","tokenEND"}},\

{"program", 2, {"program\_\_\_\_part1","program\_\_\_\_part2"}},

#define T\_NAME\_0 "startprogram"

#define T\_NAME\_1 ""

#define T\_NAME\_2 ""

#define T\_NAME\_3 ""

#define T\_BODY\_0 "variable"

#define T\_BODY\_1 ""

#define T\_BODY\_2 ""

#define T\_BODY\_3 ""

#define T\_DATA\_0 "startblok"

#define T\_DATA\_1 ""

#define T\_DATA\_2 ""

#define T\_DATA\_3 ""

#define T\_DATA\_TYPE\_0 "int\_2"

#define T\_DATA\_TYPE\_1 ""

#define T\_DATA\_TYPE\_2 ""

#define T\_DATA\_TYPE\_3 ""

//

#define T\_BITWISE\_NOT\_0 ""

#define T\_BITWISE\_NOT\_1 ""

#define T\_BITWISE\_NOT\_2 ""

#define T\_BITWISE\_NOT\_3 ""

#define T\_BITWISE\_AND\_0 ""

#define T\_BITWISE\_AND\_1 ""

#define T\_BITWISE\_AND\_2 ""

#define T\_BITWISE\_AND\_3 ""

#define T\_BITWISE\_OR\_0 ""

#define T\_BITWISE\_OR\_1 ""

#define T\_BITWISE\_OR\_2 ""

#define T\_BITWISE\_OR\_3 ""

#define T\_NOT\_0 "not"

#define T\_NOT\_1 ""

#define T\_NOT\_2 ""

#define T\_NOT\_3 ""

#define T\_AND\_0 "and"

#define T\_AND\_1 ""

#define T\_AND\_2 ""

#define T\_AND\_3 ""

#define T\_OR\_0 "or"

#define T\_OR\_1 ""

#define T\_OR\_2 ""

#define T\_OR\_3 ""

//

#define T\_EQUAL\_0 "eq"

#define T\_EQUAL\_1 ""

#define T\_EQUAL\_2 ""

#define T\_EQUAL\_3 ""

#define T\_NOT\_EQUAL\_0 "noteq"

#define T\_NOT\_EQUAL\_1 ""

#define T\_NOT\_EQUAL\_2 ""

#define T\_NOT\_EQUAL\_3 ""

#define T\_LESS\_0 "less"

#define T\_LESS\_1 ""

#define T\_LESS\_2 ""

#define T\_LESS\_3 ""

#define T\_GREATER\_0 "gr"

#define T\_GREATER\_1 ""

#define T\_GREATER\_2 ""

#define T\_GREATER\_3 ""

#define T\_LESS\_OR\_EQUAL\_0 ""

#define T\_LESS\_OR\_EQUAL\_1 ""

#define T\_LESS\_OR\_EQUAL\_2 ""

#define T\_LESS\_OR\_EQUAL\_3 ""

#define T\_GREATER\_OR\_EQUAL\_0 ""

#define T\_GREATER\_OR\_EQUAL\_1 ""

#define T\_GREATER\_OR\_EQUAL\_2 ""

#define T\_GREATER\_OR\_EQUAL\_3 ""

//

#define T\_ADD\_0 "+"

#define T\_ADD\_1 ""

#define T\_ADD\_2 ""

#define T\_ADD\_3 ""

#define T\_SUB\_0 "-"

#define T\_SUB\_1 ""

#define T\_SUB\_2 ""

#define T\_SUB\_3 ""

#define T\_MUL\_0 "\*"

#define T\_MUL\_1 ""

#define T\_MUL\_2 ""

#define T\_MUL\_3 ""

#define T\_DIV\_0 "/"

#define T\_DIV\_1 ""

#define T\_DIV\_2 ""

#define T\_DIV\_3 ""

#define T\_MOD\_0 "%"

#define T\_MOD\_1 ""

#define T\_MOD\_2 ""

#define T\_MOD\_3 ""

//

#define T\_BIND\_RIGHT\_TO\_LEFT\_0 "<-"

#define T\_BIND\_RIGHT\_TO\_LEFT\_1 ""

#define T\_BIND\_RIGHT\_TO\_LEFT\_2 ""

#define T\_BIND\_RIGHT\_TO\_LEFT\_3 ""

#define T\_BIND\_LEFT\_TO\_RIGHT\_0 ""

#define T\_BIND\_LEFT\_TO\_RIGHT\_1 ""

#define T\_BIND\_LEFT\_TO\_RIGHT\_2 ""

#define T\_BIND\_LEFT\_TO\_RIGHT\_3 ""

//

#define T\_COMA\_0 ","

#define T\_COMA\_1 ""

#define T\_COMA\_2 ""

#define T\_COMA\_3 ""

#define T\_COLON\_0 ":"

#define T\_COLON\_1 ""

#define T\_COLON\_2 ""

#define T\_COLON\_3 ""

#define T\_GOTO\_0 "goto"

#define T\_GOTO\_1 ""

#define T\_GOTO\_2 ""

#define T\_GOTO\_3 ""

//

#define T\_IF\_0 "if"

#define T\_IF\_1 "("

#define T\_IF\_2 ""

#define T\_IF\_3 ""

#define T\_THEN\_0 ")"

#define T\_THEN\_1 ""

#define T\_THEN\_2 ""

#define T\_THEN\_3 ""

#define T\_ELSE\_0 "else"

#define T\_ELSE\_1 ""

#define T\_ELSE\_2 ""

#define T\_ELSE\_3 ""

//

#define T\_FOR\_0 "for"

#define T\_FOR\_1 ""

#define T\_FOR\_2 ""

#define T\_FOR\_3 ""

#define T\_TO\_0 "to"

#define T\_TO\_1 ""

#define T\_TO\_2 ""

#define T\_TO\_3 ""

#define T\_DOWNTO\_0 "downto"

#define T\_DOWNTO\_1 ""

#define T\_DOWNTO\_2 ""

#define T\_DOWNTO\_3 ""

#define T\_DO\_0 "do"

#define T\_DO\_1 ""

#define T\_DO\_2 ""

#define T\_DO\_3 ""

//

#define T\_WHILE\_0 "while"

#define T\_WHILE\_1 ""

#define T\_WHILE\_2 ""

#define T\_WHILE\_3 ""

#define T\_CONTINUE\_WHILE\_0 "continue"

#define T\_CONTINUE\_WHILE\_1 "while"

#define T\_CONTINUE\_WHILE\_2 ""

#define T\_CONTINUE\_WHILE\_3 ""

#define T\_EXIT\_WHILE\_0 "exit"

#define T\_EXIT\_WHILE\_1 "while"

#define T\_EXIT\_WHILE\_2 ""

#define T\_EXIT\_WHILE\_3 ""

#define T\_END\_WHILE\_0 "end"

#define T\_END\_WHILE\_1 "while"

#define T\_END\_WHILE\_2 ""

#define T\_END\_WHILE\_3 ""

//

#define T\_REPEAT\_0 "repeat"

#define T\_REPEAT\_1 ""

#define T\_REPEAT\_2 ""

#define T\_REPEAT\_3 ""

#define T\_UNTIL\_0 "until"

#define T\_UNTIL\_1 ""

#define T\_UNTIL\_2 ""

#define T\_UNTIL\_3 ""

//

#define T\_INPUT\_0 "get"

#define T\_INPUT\_1 ""

#define T\_INPUT\_2 ""

#define T\_INPUT\_3 ""

#define T\_OUTPUT\_0 "put"

#define T\_OUTPUT\_1 ""

#define T\_OUTPUT\_2 ""

#define T\_OUTPUT\_3 ""

//

#define T\_RLBIND\_0 "<-"

#define T\_RLBIND\_1 ""

#define T\_RLBIND\_2 ""

#define T\_RLBIND\_3 ""

#define T\_LRBIND\_0 ""

#define T\_LRBIND\_1 ""

#define T\_LRBIND\_2 ""

#define T\_LRBIND\_3 ""

//

#define T\_SEMICOLON\_0 ";"

#define T\_SEMICOLON\_1 ""

#define T\_SEMICOLON\_2 ""

#define T\_SEMICOLON\_3 ""

//

#define T\_BEGIN\_0 "startblok"

#define T\_BEGIN\_1 ""

#define T\_BEGIN\_2 ""

#define T\_BEGIN\_3 ""

#define T\_END\_0 "endblok"

#define T\_END\_1 ""

#define T\_END\_2 ""

#define T\_END\_3 ""

//

#define T\_NULL\_STATEMENT\_0 "NULL"

#define T\_NULL\_STATEMENT\_1 "STATEMENT"

#define T\_NULL\_STATEMENT\_2 ""

#define T\_NULL\_STATEMENT\_3 ""

#ifndef TOKEN\_STRUCT\_NAME\_

#define TOKEN\_STRUCT\_NAME\_

DECLENUM(TokenStructName,

MULTI\_TOKEN\_BITWISE\_NOT,

MULTI\_TOKEN\_BITWISE\_AND,

MULTI\_TOKEN\_BITWISE\_OR,

MULTI\_TOKEN\_NOT,

MULTI\_TOKEN\_AND,

MULTI\_TOKEN\_OR,

MULTI\_TOKEN\_EQUAL,

MULTI\_TOKEN\_NOT\_EQUAL,

MULTI\_TOKEN\_LESS,

MULTI\_TOKEN\_GREATER,

MULTI\_TOKEN\_LESS\_OR\_EQUAL,

MULTI\_TOKEN\_GREATER\_OR\_EQUAL,

MULTI\_TOKEN\_ADD,

MULTI\_TOKEN\_SUB,

MULTI\_TOKEN\_MUL,

MULTI\_TOKEN\_DIV,

MULTI\_TOKEN\_MOD,

MULTI\_TOKEN\_BIND\_RIGHT\_TO\_LEFT,

MULTI\_TOKEN\_BIND\_LEFT\_TO\_RIGHT,

MULTI\_TOKEN\_COLON,

MULTI\_TOKEN\_GOTO,

MULTI\_TOKEN\_IF,

// MULTI\_TOKEN\_IF\_, // don't change this!

MULTI\_TOKEN\_THEN,

// MULTI\_TOKEN\_THEN\_, // don't change this!

MULTI\_TOKEN\_ELSE,

MULTI\_TOKEN\_FOR,

MULTI\_TOKEN\_TO,

MULTI\_TOKEN\_DOWNTO,

MULTI\_TOKEN\_DO,

//

MULTI\_TOKEN\_WHILE,

/\*while special statement\*/MULTI\_TOKEN\_CONTINUE\_WHILE,

/\*while special statement\*/MULTI\_TOKEN\_EXIT\_WHILE,

MULTI\_TOKEN\_END\_WHILE,

//

//

MULTI\_TOKEN\_REPEAT,

MULTI\_TOKEN\_UNTIL,

//

//

MULTI\_TOKEN\_INPUT,

MULTI\_TOKEN\_OUTPUT,

//

//

MULTI\_TOKEN\_RLBIND,

MULTI\_TOKEN\_LRBIND,

//

MULTI\_TOKEN\_SEMICOLON,

MULTI\_TOKEN\_BEGIN,

MULTI\_TOKEN\_END,

//

MULTI\_TOKEN\_NULL\_STATEMENT

);

//#define PROCESS\_TOKENS(...) HANDLE\_TOKENS(\_\_VA\_ARGS\_\_)

//#define TOKENS\_FOR\_MULTI\_TOKEN(A, B, C, D) A, B, C, D

//#define TOKENS\_FOR\_MULTI\_TOKEN\_BITWISE\_NOT TOKENS\_FOR\_MULTI\_TOKEN("~", "", "", "")

#define INIT\_TOKEN\_STRUCT\_NAME() static void intitTokenStruct(){\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BITWISE\_NOT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BITWISE\_AND)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BITWISE\_OR)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, NOT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, AND)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, OR)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, EQUAL)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, NOT\_EQUAL)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, LESS)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, GREATER)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, LESS\_OR\_EQUAL)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, GREATER\_OR\_EQUAL)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, ADD)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, SUB)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, MUL)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, DIV)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, MOD)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BIND\_RIGHT\_TO\_LEFT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BIND\_LEFT\_TO\_RIGHT)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, COLON)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, GOTO)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, IF)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, THEN)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, ELSE)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, FOR)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, TO)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, DOWNTO)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, DO)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, WHILE)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, CONTINUE\_WHILE)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, EXIT\_WHILE)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, END\_WHILE)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, REPEAT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, UNTIL)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, INPUT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, OUTPUT)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, RLBIND)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, LRBIND)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, SEMICOLON)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BEGIN)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, END)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, NULL\_STATEMENT)\

} char intitTokenStruct\_ = (intitTokenStruct(), 0);

#define MAX\_TOKEN\_STRUCT\_ELEMENT\_COUNT GET\_ENUM\_SIZE(TokenStructName)

#define MAX\_TOKEN\_STRUCT\_ELEMENT\_PART\_COUNT 4

#endif

extern char\* tokenStruct[MAX\_TOKEN\_STRUCT\_ELEMENT\_COUNT][MAX\_TOKEN\_STRUCT\_ELEMENT\_PART\_COUNT];

#define CONFIGURABLE\_GRAMMAR {\

{"labeled\_point", 2, {"ident", "tokenCOLON"}},\

{"goto\_label", 2, {"tokenGOTO","ident"}},\

{"program\_name", 1, {"ident\_terminal"}},\

{"value\_type", 1, {T\_DATA\_TYPE\_0}},\

{"other\_declaration\_ident", 2, {"tokenCOMMA", "ident"}},\

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"other\_declaration\_ident","other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},\

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"tokenCOMMA", "ident"}},\

{"value\_type\_\_ident", 2, {"value\_type", "ident"}},\

{"declaration", 2, {"value\_type\_\_ident", "other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},\

{"declaration", 2, {"value\_type", "ident"}},\

\

{"unary\_operator", 1, {T\_NOT\_0}},\

{"unary\_operator", 1, {T\_SUB\_0}},\

{"unary\_operator", 1, {T\_ADD\_0}},\

{"binary\_operator", 1, {T\_AND\_0}},\

{"binary\_operator", 1, {T\_OR\_0}},\

{"binary\_operator", 1, {T\_EQUAL\_0}},\

{"binary\_operator", 1, {T\_NOT\_EQUAL\_0}},\

{"binary\_operator", 1, {T\_LESS\_OR\_EQUAL\_0}},\

{"binary\_operator", 1, {T\_GREATER\_OR\_EQUAL\_0}},\

{"binary\_operator", 1, {T\_ADD\_0}},\

{"binary\_operator", 1, {T\_SUB\_0}},\

{"binary\_operator", 1, {T\_MUL\_0}},\

{"binary\_operator", 1, {T\_DIV\_0}},\

{"binary\_operator", 1, {T\_MOD\_0}},\

{"binary\_action", 2, {"binary\_operator","expression"}},\

\

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

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

{"left\_expression", 1, {"ident\_terminal"}},\

{"left\_expression", 1, {"value\_terminal"}},\

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action\_\_\_\_iteration\_after\_two"}},\

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action"}},\

{"expression", 2, {"left\_expression","binary\_action\_\_\_\_iteration\_after\_two"}},\

{"expression", 2, {"left\_expression","binary\_action"}},\

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

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

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

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

\

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

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

\

{"bind\_right\_to\_left", 2, {"ident","rl\_expression"}},\

{"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, {"tokenENDWHILE\_END","tokenENDWHILE\_WHILE"}},\

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

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

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

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

{"while\_cycle", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},\

\

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

{"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two", 2, {"tokenREPEAT","statement\_\_\_\_iteration\_after\_two"}},\

{"tokenREPEAT\_\_statement", 2, {"tokenREPEAT","statement"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

\

{"input\_\_first\_part", 2, {"tokenGET","tokenGROUPEXPRESSIONBEGIN"}},\

{"input\_\_second\_part", 2, {"ident","tokenGROUPEXPRESSIONEND"}},\

{"input", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

\

{"output\_\_first\_part", 2, {"tokenPUT","tokenGROUPEXPRESSIONBEGIN"}},\

{"output\_\_second\_part", 2, {"expression","tokenGROUPEXPRESSIONEND"}},\

{"output", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

\

{"statement", 2, {"ident","rl\_expression"}},\

{"statement", 2, {"lr\_expression","ident"}},\

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

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

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

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

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

{"statement", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},\

{"statement", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"ident","tokenCOLON"}},\

{"statement", 2, {"tokenGOTO","ident"}},\

{"statement", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

{"statement", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement\_\_\_\_iteration\_after\_two"}},\

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement"}},\

\

{ "statement\_in\_while\_body", 2, {"ident","rl\_expression"}},\

{ "statement\_in\_while\_body", 2, {"lr\_expression","ident"}},\

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

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

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

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

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

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

{ "statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{ "statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{ "statement\_in\_while\_body", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

{ "statement\_in\_while\_body", 2, {"ident","tokenCOLON"}},\

{ "statement\_in\_while\_body", 2, {"tokenGOTO","ident"}},\

{ "statement\_in\_while\_body", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

{ "statement\_in\_while\_body", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

{ "statement\_in\_while\_body", 2, {"tokenCONTINUE","tokenWHILE"}},\

{ "statement\_in\_while\_body", 2, {"tokenEXIT","tokenWHILE"}},\

{ "statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"}},\

{ "statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body"}},\

\

PROGRAM\_FORMAT\

\

{"tokenCOLON", 1, {T\_COLON\_0}},\

{"tokenGOTO", 1, {T\_GOTO\_0}},\

{"tokenINTEGER16", 1, {T\_DATA\_TYPE\_0}},\

{"tokenCOMMA", 1, {T\_COMA\_0}},\

{"tokenNOT", 1, {T\_NOT\_0}},\

{"tokenAND", 1, {T\_AND\_0}},\

{"tokenOR", 1, {T\_OR\_0}},\

{"tokenEQUAL", 1, {T\_EQUAL\_0}},\

{"tokenNOTEQUAL", 1, {T\_NOT\_EQUAL\_0}},\

{"tokenLESSOREQUAL", 1, {T\_LESS\_OR\_EQUAL\_0}},\

{"tokenGREATEROREQUAL", 1, {T\_GREATER\_OR\_EQUAL\_0}},\

{"tokenPLUS", 1, {T\_ADD\_0}},\

{"tokenMINUS", 1, {T\_SUB\_0}},\

{"tokenMUL", 1, {T\_MUL\_0}},\

{"tokenDIV", 1, {T\_DIV\_0}},\

{"tokenMOD", 1, {T\_MOD\_0}},\

{"tokenGROUPEXPRESSIONBEGIN", 1, {"("}},\

{"tokenGROUPEXPRESSIONEND", 1, {")"}},\

{"tokenRLBIND", 1, {T\_RLBIND\_0}},\

{"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}},\

{"tokenENDWHILE\_END", 1, {T\_END\_WHILE\_0}},\

{"tokenENDWHILE\_WHILE", 1, {T\_END\_WHILE\_1}},\

{"tokenREPEAT", 1, {T\_REPEAT\_0}},\

{"tokenUNTIL", 1, {T\_UNTIL\_0}},\

{"tokenGET", 1, {T\_INPUT\_0}},\

{"tokenPUT", 1, {T\_OUTPUT\_0}},\

{"tokenNAME", 1, {T\_NAME\_0}},\

{"tokenBODY", 1, {T\_BODY\_0}},\

{"tokenDATA", 1, {T\_DATA\_0}},\

{"tokenEND", 1, {T\_END\_0}},\

{"tokenSEMICOLON", 1, {T\_SEMICOLON\_0}},\

\

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

\

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

\

{"", 2, {"",""}}\

},\

178,\

"program"

#define ORIGINAL\_GRAMMAR {\

{"labeled\_point", 2, {"ident", "tokenCOLON"}},\

{"goto\_label", 2, {"tokenGOTO","ident"}},\

{"program\_name", 1, {"ident\_terminal"}},\

{"value\_type", 1, {"INTEGER16"}},\

{"other\_declaration\_ident", 2, {"tokenCOMMA", "ident"}},\

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"other\_declaration\_ident","other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},\

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"tokenCOMMA", "ident"}},\

{"value\_type\_\_ident", 2, {"value\_type", "ident"}},\

{"declaration", 2, {"value\_type\_\_ident", "other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},\

{"declaration", 2, {"value\_type", "ident"}},\

\

{"unary\_operator", 1, {"NOT"}},\

{"unary\_operator", 1, {"-"}},\

{"unary\_operator", 1, {"+"}},\

{"binary\_operator", 1, {"AND"}},\

{"binary\_operator", 1, {"OR"}},\

{"binary\_operator", 1, {"=="}},\

{"binary\_operator", 1, {"!="}},\

{"binary\_operator", 1, {"<="}},\

{"binary\_operator", 1, {">="}},\

{"binary\_operator", 1, {"+"}},\

{"binary\_operator", 1, {"-"}},\

{"binary\_operator", 1, {"\*"}},\

{"binary\_operator", 1, {"DIV"}},\

{"binary\_operator", 1, {"MOD"}},\

{"binary\_action", 2, {"binary\_operator","expression"}},\

\

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

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

{"left\_expression", 1, {"ident\_terminal"}},\

{"left\_expression", 1, {"value\_terminal"}},\

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action\_\_\_\_iteration\_after\_two"}},\

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action"}},\

{"expression", 2, {"left\_expression","binary\_action\_\_\_\_iteration\_after\_two"}},\

{"expression", 2, {"left\_expression","binary\_action"}},\

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

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

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

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

\

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

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

\

{"bind\_right\_to\_left", 2, {"ident","rl\_expression"}},\

{"bind\_left\_to\_right", 2, {"lr\_expression","ident"}},\

\

{"body\_for\_true", 2, {"statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenSEMICOLON"}},\

{"body\_for\_true", 2, {"statement\_in\_while\_body","tokenSEMICOLON"}},\

{"body\_for\_true", 1, {";"}},\

{"tokenELSE\_\_statement\_in\_while\_body", 2, {"tokenELSE","statement\_in\_while\_body"}},\

{"tokenELSE\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"tokenELSE","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"}},\

{"body\_for\_false", 2, {"tokenELSE\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenSEMICOLON"}},\

{"body\_for\_false", 2, {"tokenELSE\_\_statement\_in\_while\_body","tokenSEMICOLON"}},\

{"body\_for\_false", 2, {"tokenELSE","tokenSEMICOLON"}},\

{"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN", 2, {"tokenIF","tokenGROUPEXPRESSIONBEGIN"}},\

{"expression\_\_tokenGROUPEXPRESSIONEND", 2, {"expression","tokenGROUPEXPRESSIONEND"}},\

{"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN","expression\_\_tokenGROUPEXPRESSIONEND"}},\

{"body\_for\_true\_\_body\_for\_false", 2, {"body\_for\_true","body\_for\_false"}},\

{"cond\_block", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND","body\_for\_true\_\_body\_for\_false"}},\

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

\

{"cycle\_counter", 1, {"ident\_terminal"}},\

{"rl\_expression", 2, {"tokenRLBIND","expression"}},\

{"lr\_expression", 2, {"expression","tokenLRBIND"}},\

{"cycle\_counter\_init", 2, {"cycle\_counter","rl\_expression"}},\

{"cycle\_counter\_init", 2, {"lr\_expression","cycle\_counter"}},\

{"cycle\_counter\_last\_value", 1, {"value\_terminal"}},\

{"cycle\_body", 2, {"tokenDO","statement\_\_\_\_iteration\_after\_two"}},\

{"cycle\_body", 2, {"tokenDO","statement"}},\

{"tokenFOR\_\_cycle\_counter\_init", 2, {"tokenFOR","cycle\_counter\_init"}},\

{"tokenTO\_\_cycle\_counter\_last\_value", 2, {"tokenTO","cycle\_counter\_last\_value"}},\

{"tokenFOR\_\_cycle\_counter\_init\_\_tokenTO\_\_cycle\_counter\_last\_value", 2, {"tokenFOR\_\_cycle\_counter\_init","tokenTO\_\_cycle\_counter\_last\_value"}},\

{"cycle\_body\_\_tokenSEMICOLON", 2, {"cycle\_body","tokenSEMICOLON"}},\

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

\

{"continue\_while", 2, {"tokenCONTINUE","tokenWHILE"}},\

{"exit\_while", 2, {"tokenEXIT","tokenWHILE"}},\

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

{"tokenEND\_\_tokenWHILE", 2, {"tokenEND","tokenWHILE"}},\

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

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

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

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

{"while\_cycle", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},\

\

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

{"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two", 2, {"tokenREPEAT","statement\_\_\_\_iteration\_after\_two"}},\

{"tokenREPEAT\_\_statement", 2, {"tokenREPEAT","statement"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

\

{"input\_\_first\_part", 2, {"tokenGET","tokenGROUPEXPRESSIONBEGIN"}},\

{"input\_\_second\_part", 2, {"ident","tokenGROUPEXPRESSIONEND"}},\

{"input", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

\

{"output\_\_first\_part", 2, {"tokenPUT","tokenGROUPEXPRESSIONBEGIN"}},\

{"output\_\_second\_part", 2, {"expression","tokenGROUPEXPRESSIONEND"}},\

{"output", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

\

{"statement", 2, {"ident","rl\_expression"}},\

{"statement", 2, {"lr\_expression","ident"}},\

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

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

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

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

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

{"statement", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},\

{"statement", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"ident","tokenCOLON"}},\

{"statement", 2, {"tokenGOTO","ident"}},\

{"statement", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

{"statement", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement\_\_\_\_iteration\_after\_two"}},\

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement"}},\

\

{"statement\_in\_while\_body", 2, {"ident","rl\_expression"}},\

{"statement\_in\_while\_body", 2, {"lr\_expression","ident"}},\

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

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

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

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

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

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

{"statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"statement\_in\_while\_body", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

{"statement\_in\_while\_body", 2, {"ident","tokenCOLON"}},\

{"statement\_in\_while\_body", 2, {"tokenGOTO","ident"}},\

{"statement\_in\_while\_body", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

{"statement\_in\_while\_body", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

{"statement\_in\_while\_body", 2, {"tokenCONTINUE","tokenWHILE"}},\

{"statement\_in\_while\_body", 2, {"tokenEXIT","tokenWHILE"}},\

{"statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"}},\

{"statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body"}},\

\

{"tokenNAME\_\_program\_name", 2, {"tokenNAME","program\_name"}},\

{"tokenSEMICOLON\_\_tokenBODY", 2, {"tokenSEMICOLON","tokenBODY"}},\

{"tokenDATA\_\_declaration", 2, {"tokenDATA","declaration"}},\

{"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY", 2, {"tokenNAME\_\_program\_name","tokenSEMICOLON\_\_tokenBODY"}},\

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA\_\_declaration"}},\

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA"}},\

{"statement\_\_tokenEND", 2, {"statement","tokenEND"}},\

{"statement\_\_\_\_iteration\_after\_two\_\_tokenEND", 2, {"statement\_\_\_\_iteration\_after\_two","tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_\_\_iteration\_after\_two\_\_tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","tokenEND"}},\

{"program", 2, {"program\_\_\_\_part1","program\_\_\_\_part2"}},\

\

{"tokenCOLON", 1, {":"}},\

{"tokenGOTO", 1, {"GOTO"}},\

{"tokenINTEGER16", 1, {"INTEGER16"}},\

{"tokenCOMMA", 1, {","}},\

{"tokenNOT", 1, {"NOT"}},\

{"tokenAND", 1, {"AND"}},\

{"tokenOR", 1, {"OR"}},\

{"tokenEQUAL", 1, {"=="}},\

{"tokenNOTEQUAL", 1, {"!="}},\

{"tokenLESSOREQUAL", 1, {"<="}},\

{"tokenGREATEROREQUAL", 1, {">="}},\

{"tokenPLUS", 1, {"+"}},\

{"tokenMINUS", 1, {"-"}},\

{"tokenMUL", 1, {"\*"}},\

{"tokenDIV", 1, {"DIV"}},\

{"tokenMOD", 1, {"MOD"}},\

{"tokenGROUPEXPRESSIONBEGIN", 1, {"("}},\

{"tokenGROUPEXPRESSIONEND", 1, {")"}},\

{"tokenRLBIND", 1, {"<<"}},\

{"tokenLRBIND", 1, {">>"}},\

{"tokenELSE", 1, {"ELSE"}},\

{"tokenIF", 1, {"IF"}},\

{"tokenDO", 1, {"DO"}},\

{"tokenFOR", 1, {"FOR"}},\

{"tokenTO", 1, {"TO"}},\

{"tokenWHILE", 1, {"WHILE"}},\

{"tokenCONTINUE", 1, {"CONTINUE"}},\

{"tokenEXIT", 1, {"EXIT"}},\

{"tokenREPEAT", 1, {"REPEAT"}},\

{"tokenUNTIL", 1, {"UNTIL"}},\

{"tokenGET", 1, {"GET"}},\

{"tokenPUT", 1, {"PUT"}},\

{"tokenNAME", 1, {"NAME"}},\

{"tokenBODY", 1, {"BODY"}},\

{"tokenDATA", 1, {"DATA"}},\

{"tokenEND", 1, {"END"}},\

{"tokenSEMICOLON", 1, {";"}},\

\

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

\

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

\

{"", 2, {"",""}}\

\

},\

176,\

"program"

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

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

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

#define DEFAULT\_MODE (DEBUG\_MODE | LEXICAL\_ANALYZE\_MODE | SYNTAX\_ANALYZE\_MODE | SEMANTIX\_ANALYZE\_MODE | MAKE\_ASSEMBLY | MAKE\_BINARY)  
  
ADD.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: add.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_ADD);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if(generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_ADD][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_ADD][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] += opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

AND.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: and.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_AND);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_mov\_eax\_stackTopByECX[] = { 0x8B, 0x01 };

const unsigned char code\_\_cmp\_eax\_0[] = { 0x83, 0xF8, 0x00 };

const unsigned char code\_\_setne\_al[] = { 0x0F, 0x95, 0xC0 };

const unsigned char code\_\_and\_eax\_1[] = { 0x83, 0xE0, 0x01 };

const unsigned char code\_\_sub\_ecx\_4[] = { 0x83, 0xE9, 0x04 };

//

const unsigned char code\_\_cmp\_stackTopByECX\_0[] = { 0x83, 0x39, 0x00 };

const unsigned char code\_\_setne\_dl[] = { 0x0F, 0x95, 0xC2 };

const unsigned char code\_\_and\_edx\_1[] = { 0x83, 0xE2, 0x01 };

//

const unsigned char code\_\_and\_eax\_edx[] = { 0x23, 0xC2 };

//

const unsigned char code\_\_mov\_stackTopByECX\_eax[] = { 0x89, 0x01 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_eax\_stackTopByECX, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_cmp\_eax\_0, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_setne\_al, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_and\_eax\_1, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_sub\_ecx\_4, 3);

//

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_cmp\_stackTopByECX\_0, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_setne\_dl, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_and\_edx\_1, 3);

//

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_and\_eax\_edx, 2);

//

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_stackTopByECX\_eax, 2);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_AND][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp eax, 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " setne al\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and eax, 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

//

currBytePtr += sprintf((char\*)currBytePtr, " cmp dword ptr[ecx], 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " setne dl\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and edx, 1\r\n");

//

currBytePtr += sprintf((char\*)currBytePtr, " and eax, edx\r\n");

//

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr[ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_AND][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] = opStack[opStackIndex - 1] && opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

BITWISE\_AND.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192," ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_BITWISE\_AND][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_BITWISE\_AND][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] &= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

BITWISE\_NOT.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192," ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_BITWISE\_AND][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_BITWISE\_AND][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] &= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

BITWISE\_OR.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_BITWISE\_OR][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " aor dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_BITWISE\_OR][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] |= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
CLI.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_BITWISE\_OR][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " aor dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_BITWISE\_OR][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] |= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
CW.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS // for using sscanf in VS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: cw.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//#pragma comment(linker, "/STACK:516777216")

#include <windows.h>

//#include <winbase.h>

//#include <winuser.h>

//#include <shlobj.h>

//#include <shlwapi.h>

//#include <objbase.h>

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

//#include "conio.h"

//#include "locale.h"

#include <direct.h>

#include <fstream>

#include <iostream>

#include <algorithm>//

#include <iterator>

#include <regex>

#include "src/include/def.h"

#include "src/include/config.h"

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

//bool reSetDefaultInputFileName\_ =

//strcpy(parameters[INPUT\_FILENAME\_WITH\_EXTENSION\_PARAMETER], "../test\_programs/file1.cwl")

//!= NULL;

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

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

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

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

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

struct LexemInfo lexemesInfoTableTemp[MAX\_WORD\_COUNT]; // = { { "", 0, 0, 0 } };

struct LexemInfo\* lastLexemInfoInTableTemp = lexemesInfoTableTemp; // first for begin

unsigned char new\_code[8 \* 1024 \* 1024] = { '\0' }; //

unsigned char tempCodeBuffer[8 \* 1024 \* 1024] = { '\0' };

unsigned char outCodeBuffer[8 \* 1024 \* 1024] = { '\0' };

unsigned char errorMessagesPtrToLastBytePtr[8 \* 1024 \* 1024] = { '\0' };

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

PostMessage(GetForegroundWindow(), WM\_INPUTLANGCHANGEREQUEST, 2, (UINT)LoadKeyboardLayoutA("00000409", KLF\_ACTIVATE));

char path[PATH\_NAME\_LENGH];

char temp[2 \* PATH\_NAME\_LENGH];

char productionOut[MAX\_TEXT\_SIZE] = { 0 };

comandLineParser(argc, argv, &mode, parameters);

char\* text;

size\_t sourceSize = loadSource(&text, parameters[INPUT\_FILENAME\_WITH\_EXTENSION\_PARAMETER]);

if (!sourceSize) {

printf("Empty source . . .");

printf("Press Enter to exit . . .");

getchar();

return 0;

}

if (!\_getcwd(path, PATH\_NAME\_LENGH))

{

printf("getcwd error ...\r\n");

printf("Press Enter to exit . . .");

return -1;

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in step-by-step interactive mode.\r\n");

printf("ATTENTIOON: The next step is critical, if it is skipped the compilation process will be terminated!\r\n");

printf("Enter 'y' to lexical analyze action(to pass action process enter 'n' or others key): ");

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || getchar() == 'y') || mode & LEXICAL\_ANALYZE\_MODE) {

if (mode & (DEBUG\_MODE | INTERACTIVE\_MODE)) {

printf("Original source:\r\n");

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

printf("%s\r\n", text);

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

}

int commentRemoverResult = commentRemover(text, "/\*", "/n");

if (commentRemoverResult) {

printf("Comment remover return %d\r\n", commentRemoverResult);

printf("Press Enter to exit . . .");

(void)getchar();

return 0;

}

if (mode & (DEBUG\_MODE | INTERACTIVE\_MODE)) {

printf("Source after comment removing:\r\n");

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

printf("%s\r\n", text);

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

}

struct LexemInfo ifBadLexemeInfo = tokenize(text, &lastLexemInfoInTable, identifierIdsTable, lexicalAnalyze);

if (ifBadLexemeInfo.tokenType == UNEXPEXTED\_LEXEME\_TYPE) {

UNEXPEXTED\_LEXEME\_TYPE;

ifBadLexemeInfo.tokenType;

printf("Lexical analysis detected unexpected lexeme\r\n");

printLexemes(&ifBadLexemeInfo, 1);

if (parameters[OUT\_LEXEME\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

printLexemesToFile(lexemesInfoTable, 1, parameters[OUT\_LEXEME\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER]);

}

printf("Press Enter to exit . . .");

(void)getchar();

return 0;

}

if (mode & (DEBUG\_MODE | INTERACTIVE\_MODE)) {

printLexemes(lexemesInfoTable, 0);

if (parameters[OUT\_LEXEMES\_SEQUENSE\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

printLexemesToFile(lexemesInfoTable, 0, parameters[OUT\_LEXEMES\_SEQUENSE\_FILENAME\_WITH\_EXTENSION\_PARAMETER]);

}

if (parameters[OUT\_LEXEME\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

writeBytesToFile(parameters[OUT\_LEXEME\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER], (unsigned char\*)"No error.", strlen("No error."));

}

}

else {

printf("Lexical analysis complete success\r\n");

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

printf("\r\nPress Enter to next step");

(void)getchar();

(void)getchar();

}

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in step-by-step interactive mode.\r\n");

printf("ATTENTIOON: The next step is critical, if it is skipped the compilation process will be terminated!\r\n");

printf("Enter 'y' to syntax analyze action(to pass action process enter 'n' or others key): ");

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || getchar() == 'y') || mode & SYNTAX\_ANALYZE\_MODE) {

errorMessagesPtrToLastBytePtr[0] = '\0';

unsigned char\* errorMessagesPtrToLastBytePtr\_ = errorMessagesPtrToLastBytePtr;

if (SUCCESS\_STATE != syntaxAnalyze(lexemesInfoTable, &grammar, DEFAULT\_SYNTAX\_ANAlYZE\_MODE, parameters[OUT\_AST\_FILENAME\_WITH\_EXTENSION\_PARAMETER], (char\*)&errorMessagesPtrToLastBytePtr)) { // TODO: add AST param

if (parameters[OUT\_SYNTAX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

writeBytesToFile(parameters[OUT\_SYNTAX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER], errorMessagesPtrToLastBytePtr, strlen((const char\*)errorMessagesPtrToLastBytePtr));

}

if (parameters[OUT\_AST\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

writeBytesToFile(parameters[OUT\_AST\_FILENAME\_WITH\_EXTENSION\_PARAMETER], (unsigned char\*)"AST build failed.", strlen("AST build failed."));

}

return 0;

}

if (parameters[OUT\_SYNTAX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

writeBytesToFile(parameters[OUT\_SYNTAX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER], (unsigned char\*)"No error.", strlen("No error."));

printf("No error.\r\n");

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

printf("\r\nPress Enter to next step");

(void)getchar();

(void)getchar();

}

}

else {

printf("\r\ncw terminated.");

return 0;

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in step-by-step interactive mode.\r\n");

printf("ATTENTIOON: The next step is critical, if it is skipped the compilation process will be terminated!\r\n");

printf("Enter 'y' to semantix analyze action(to pass action process enter 'n' or others key): ");

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || getchar() == 'y') || mode & SEMANTIX\_ANALYZE\_MODE) {

errorMessagesPtrToLastBytePtr[0] = '\0';

unsigned char\* errorMessagesPtrToLastBytePtr\_ = errorMessagesPtrToLastBytePtr;

if (SUCCESS\_STATE != semantixAnalyze(lexemesInfoTable, &grammar, identifierIdsTable, (char \*)errorMessagesPtrToLastBytePtr)) {

if (parameters[OUT\_SEMANTIX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

writeBytesToFile(parameters[OUT\_SEMANTIX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER], errorMessagesPtrToLastBytePtr, strlen((const char\*)errorMessagesPtrToLastBytePtr));

}

return 0;

}

if (parameters[OUT\_SEMANTIX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

writeBytesToFile(parameters[OUT\_SEMANTIX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER], (unsigned char\*)"No error.", strlen("No error."));

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

printf("\r\nPress Enter to next step");

(void)getchar();

(void)getchar();

}

}

else {

printf("\r\ncw terminated.");

return 0;

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in step-by-step interactive mode.\r\n");

printf("ATTENTIOON: The next step is critical, if it is skipped the compilation process will be terminated!\r\n");

printf("Enter 'y' for the action to prepare for the compilation process(to pass the action process, enter 'n' or another key): ");

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || getchar() == 'y') || mode & MAKE\_PREPARE) {

if (mode & (DEBUG\_MODE | INTERACTIVE\_MODE)) {

printLexemes(lexemesInfoTable, 0);

}

lastLexemInfoInTable = lexemesInfoTable;

makePrepare(lexemesInfoTable, &lastLexemInfoInTable, &lastLexemInfoInTableTemp);

if (mode & (DEBUG\_MODE | INTERACTIVE\_MODE)) {

printLexemes(lexemesInfoTableTemp, 0);

if (parameters[OUT\_PREPARED\_LEXEMES\_SEQUENSE\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

printLexemesToFile(lexemesInfoTableTemp, 0, parameters[OUT\_PREPARED\_LEXEMES\_SEQUENSE\_FILENAME\_WITH\_EXTENSION\_PARAMETER]);

}

}

else {

printf("Make prepare(expressions separation + creating reverse Polish notation) complete success\r\n");

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

if (parameters[OUT\_PREPARED\_LEXEMES\_SEQUENSE\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

printf("File \"%s\" saved.\n", parameters[OUT\_PREPARED\_LEXEMES\_SEQUENSE\_FILENAME\_WITH\_EXTENSION\_PARAMETER]);

}

printf("\r\nPress Enter to next step");

(void)getchar();

(void)getchar();

}

}

else {

printf("\r\ncw terminated.");

return 0;

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in step-by-step interactive mode.\r\n");

printf("\r\n");

printf("Enter 'y' to create C-code(to pass action process enter 'n' or others key): ");

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || getchar() == 'y') || mode & MAKE\_C) { // MAKE\_ASSEMBLY\_MODE

lastLexemInfoInTableTemp = lexemesInfoTableTemp;// printLexemes(lexemesInfoTableTemp, 0);

//outCodeBuffer[0] = '\0';

makeCode(&lastLexemInfoInTableTemp, outCodeBuffer, C\_CODER\_MODE); /\*The lexem info table will be changed and will need to be rebuilt. TODO: change the implementation!\*/

printf("\r\n\r\n%s\r\n\r\n", outCodeBuffer);

if (parameters[OUT\_C\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

writeBytesToFile(parameters[OUT\_C\_FILENAME\_WITH\_EXTENSION\_PARAMETER], outCodeBuffer, strlen((const char\*)outCodeBuffer));

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

printf("\r\nPress Enter to next step");

(void)getchar();

(void)getchar();

}

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in step-by-step interactive mode.\r\n");

printf("\r\n");

printf("Enter 'y' to create assembly(to pass action process enter 'n' or others key): ");

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || getchar() == 'y') || mode & MAKE\_ASSEMBLY) { // MAKE\_ASSEMBLY\_MODE

lastLexemInfoInTableTemp = lexemesInfoTableTemp;// printLexemes(lexemesInfoTableTemp, 0);

lastLexemInfoInTable = lexemesInfoTable;

makePrepare(lexemesInfoTable, &lastLexemInfoInTable, &lastLexemInfoInTableTemp); /\* Rerun preapare process after previous etap. TODO: change the implementation!\*/

lastLexemInfoInTableTemp = lexemesInfoTableTemp;// printLexemes(lexemesInfoTableTemp, 0);

//outCodeBuffer[0] = '\0';

makeCode(&lastLexemInfoInTableTemp, outCodeBuffer, ASSEMBLY\_X86\_WIN32\_CODER\_MODE);

printf("\r\n\r\n%s\r\n\r\n", outCodeBuffer);

if (parameters[OUT\_ASSEMBLY\_FILENAME\_WITH\_EXTENSION\_PARAMETER][0] != '\0') {

writeBytesToFile(parameters[OUT\_ASSEMBLY\_FILENAME\_WITH\_EXTENSION\_PARAMETER], outCodeBuffer, strlen((const char\*)outCodeBuffer));

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

printf("\r\nPress Enter to next step");

(void)getchar();

(void)getchar();

}

}

unsigned long long int byteCountWritedToTempCodeBuffer = 0;

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) { // BUILD NATIVE CODE

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in step-by-step interactive mode.\r\n");

printf("ATTENTIOON: The next step is critical, if it is skipped the compilation process will be terminated!\r\n");

printf("Enter 'y' to create native code(to pass action process enter 'n' or others key): ");

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || getchar() == 'y') || mode & MAKE\_ASSEMBLY) {

lastLexemInfoInTableTemp = lexemesInfoTableTemp;// printLexemes(lexemesInfoTableTemp, 0);

lastLexemInfoInTable = lexemesInfoTable;

makePrepare(lexemesInfoTable, &lastLexemInfoInTable, &lastLexemInfoInTableTemp); /\* Rerun preapare process after previous etap. TODO: change the implementation!\*/

lastLexemInfoInTableTemp = lexemesInfoTableTemp;// printLexemes(lexemesInfoTableTemp, 0);

//outCodeBuffer[0] = '\0';

byteCountWritedToTempCodeBuffer = makeCode(&lastLexemInfoInTableTemp, tempCodeBuffer, MACHINE\_X86\_WIN32\_CODER\_MODE) - tempCodeBuffer;

if (mode & (DEBUG\_MODE | INTERACTIVE\_MODE)) {

viewCode(tempCodeBuffer, byteCountWritedToTempCodeBuffer, 16);

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

}

else {

printf("Native code created complete successfully.\r\n\r\n");

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

printf("\r\nPress Enter to next step");

(void)getchar();

(void)getchar();

}

}

else {

printf("\r\ncw terminated.");

return 0;

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in step-by-step interactive mode.\r\n");

printf("\r\n");

printf("Enter 'y' to create obj-file(to pass action process enter 'n' or others key): ");

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || getchar() == 'y') || mode & MAKE\_OBJECT) {

unsigned long long int objectSize = buildTemplateForCodeObject(outCodeBuffer);

unsigned char\* currBytePtr = getObjectCodeBytePtr(outCodeBuffer, MACHINE\_X86\_WIN32\_CODER\_MODE);

(void)outBytes2Code(currBytePtr, tempCodeBuffer, byteCountWritedToTempCodeBuffer);

if (mode & (DEBUG\_MODE | INTERACTIVE\_MODE)) {

viewCode(outCodeBuffer, objectSize, 16);

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

}

else {

printf("obj-code created complete successfully.\r\n\r\n");

}

writeBytesToFile(parameters[OUT\_OBJECT\_FILENAME\_WITH\_EXTENSION\_PARAMETER], outCodeBuffer, objectSize);

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

printf("\r\nPress Enter to next step");

(void)getchar();

(void)getchar();

}

}

else if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

(void)getchar();

}

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in step-by-step interactive mode.\r\n");

printf("\r\n");

printf("Enter 'y' to create exe-file(to pass action process enter 'n' or others key): ");

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || getchar() == 'y') || mode & MAKE\_BINARY) {

unsigned long long int imageSize = buildTemplateForCodeImage(outCodeBuffer);

unsigned char\* currBytePtr = getImageCodeBytePtr(outCodeBuffer, MACHINE\_X86\_WIN32\_CODER\_MODE);

(void)outBytes2Code(currBytePtr, tempCodeBuffer, byteCountWritedToTempCodeBuffer);

if (mode & (DEBUG\_MODE | INTERACTIVE\_MODE)) {

viewCode(outCodeBuffer, imageSize, 16);

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

}

else {

printf("exe-code created complete successfully.\r\n\r\n");

}

writeBytesToFile(parameters[OUT\_BINARY\_FILENAME\_WITH\_EXTENSION\_PARAMETER], outCodeBuffer, imageSize);

if (mode & INTERACTIVE\_MODE && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)) {

printf("\r\nPress Enter to next step");

(void)getchar();

(void)getchar();

}

}

else {

printf("\r\ncw terminated.");

return 0;

}

if (mode & INTERACTIVE\_MODE/\* && !(mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE)\*/) {

if (mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE) {

(void)getchar();

}

system("CLS");

fflush(stdin);

fflush(stdout);

fflush(stderr);

printf("No command line arguments are entered, so you are working in interactive mode.\r\n");

printf("\r\n");

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

}

fflush(stdin);

if (mode & INTERACTIVE\_MODE && (/\*mode & SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE || \*/getchar() == 'y') || mode & RUN\_BINARY) {

printf("\r\n");

//sprintf(temp, "START /b /wait \"\" /D \"%s\\masm32p\" %s.exe", path, parameters[OUT\_BINARY\_FILENAME\_WITHOUT\_EXTENSION\_PARAMETER]);

snprintf(temp, MAX\_PARAMETERS\_SIZE, "START /b /wait \"\" %s", parameters[OUT\_BINARY\_FILENAME\_WITH\_EXTENSION\_PARAMETER]);

fflush(stdin);

system((char\*)temp);

fflush(stdin);

}

else if (mode ^ RUN\_BINARY) {

printf("\r\n");

}

printf("\r\n\r\nPress Enter to exit . . .");

(void)getchar();

return 0;

}  
  
DIV.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: div.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_DIV);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_DIV][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx - 4]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cdq\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " idiv dword ptr [ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr [ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_DIV][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] /= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

ELSE.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: else.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

#include "string.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_ELSE);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_ELSE][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_ELSE][0]);

}

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = \*\*lastLexemInfoInTable;

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, " cmp eax, 0\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jnz LABEL@AFTER\_ELSE\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " if (opTemp != 0) goto LABEL\_\_AFTER\_ELSE\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

unsigned char\* makePostElseCode\_(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL@AFTER\_ELSE\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL\_\_AFTER\_ELSE\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

return currBytePtr;

}

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_SEMICOLON);

if (multitokenSize

&&

lexemInfoTransformationTempStackSize

&&

!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_ELSE][0], MAX\_LEXEM\_SIZE)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0], tokenStruct[MULTI\_TOKEN\_ELSE][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0], tokenStruct[MULTI\_TOKEN\_ELSE][0]);

}

currBytePtr = makePostElseCode\_(lastLexemInfoInTable, currBytePtr, generatorMode);

--lexemInfoTransformationTempStackSize;

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
EQUAL.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: equal.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_EQUAL);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_EQUAL][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sete al\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and eax, 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr[ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_EQUAL][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] = opStack[opStackIndex - 1] == opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
FOR.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: for.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

#include "string.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_FOR);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = \*\*lastLexemInfoInTable;

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_DOWNTO);

bool toMode = false;

if (!multitokenSize) {

toMode = !!(multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_TO));

}

if (multitokenSize

&&

lexemInfoTransformationTempStackSize

&&

!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_FOR][0], MAX\_LEXEM\_SIZE)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

if (toMode) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_TO][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

else {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_DOWNTO][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

}

else if (generatorMode == C\_CODER\_MODE) {

if (toMode) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_TO][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

else {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_DOWNTO][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

}

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = \*\*lastLexemInfoInTable;

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

if (toMode) {

currBytePtr += sprintf((char\*)currBytePtr, " dec dword ptr [ebx]\r\n"); // start from (index - 1)

}

else {

currBytePtr += sprintf((char\*)currBytePtr, " inc dword ptr [ebx]\r\n"); // start from (index + 1)

}

currBytePtr += sprintf((char\*)currBytePtr, " push ebx\r\n");

if (toMode) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL@AFTER\_TO\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

else {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL@AFTER\_DOWNTO\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

}

else if (generatorMode == C\_CODER\_MODE) {

if (toMode) {

currBytePtr += sprintf((char\*)currBytePtr, " --data[lastBindDataIndex];\r\n"); // start from (index - 1)

}

else {

currBytePtr += sprintf((char\*)currBytePtr, " ++data[lastBindDataIndex];\r\n"); // start from (index + 1)

}

currBytePtr += sprintf((char\*)currBytePtr, " contextStack[++contextStackIndex] = lastBindDataIndex;\r\n");

if (toMode) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL\_\_AFTER\_TO\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

else {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL\_\_AFTER\_DOWNTO\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_DO);

if (multitokenSize) {

bool toMode = false;

if (lexemInfoTransformationTempStackSize && !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_TO][0], MAX\_LEXEM\_SIZE)) {

toMode = true;

}

else if (lexemInfoTransformationTempStackSize < 2

|| strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_DOWNTO][0], MAX\_LEXEM\_SIZE)

|| strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI\_TOKEN\_FOR][0], MAX\_LEXEM\_SIZE)

) {

return currBytePtr;

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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 ?? ?? ?? ??

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_ebx\_addrFromESP, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_cmp\_addrFromEBX\_eax, 2);

if (toMode) {

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jge\_offset, 6);

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

}

else {

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jle\_offset, 6);

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

}

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

if (toMode) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\" (after \"%s\" after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_DO][0], tokenStruct[MULTI\_TOKEN\_TO][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

else {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\" (after \"%s\" after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_DO][0], tokenStruct[MULTI\_TOKEN\_DOWNTO][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

}

else if (generatorMode == C\_CODER\_MODE) {

if (toMode) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\" (after \"%s\" after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_DO][0], tokenStruct[MULTI\_TOKEN\_TO][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

else {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\" (after \"%s\" after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_DO][0], tokenStruct[MULTI\_TOKEN\_DOWNTO][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

}

if (toMode) {

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

}

else {

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

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_inc\_addrFromEBX[] = { 0xFF, 0x03 }; // inc dword ptr [ebx]

const unsigned char code\_\_dec\_addrFromEBX[] = { 0xFF, 0x0B }; // dec dword ptr [ebx]

if (toMode) {

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_inc\_addrFromEBX, 2);

}

else {

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_dec\_addrFromEBX, 2);

}

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, " mov ebx, dword ptr [esp]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp dword ptr [ebx], eax\r\n");

if (toMode) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jge LABEL@EXIT\_FOR\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

currBytePtr += sprintf((char\*)currBytePtr, " inc dword ptr [ebx]\r\n");

}

else {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jle LABEL@EXIT\_FOR\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

currBytePtr += sprintf((char\*)currBytePtr, " dec dword ptr [ebx]\r\n");

}

}

else if (generatorMode == C\_CODER\_MODE) {

if (toMode) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " if (opStack[contextStack[contextStackIndex]] >= opTemp) goto LABEL\_\_EXIT\_FOR\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

currBytePtr += sprintf((char\*)currBytePtr, " ++opStack[contextStack[contextStackIndex]];\r\n");

}

else {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " if (opStack[contextStack[contextStackIndex]] <= opTemp) goto LABEL\_\_EXIT\_FOR\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

currBytePtr += sprintf((char\*)currBytePtr, " --opStack[contextStack[contextStackIndex]];\r\n");

}

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

unsigned char\* makePostForCode\_(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode, bool toMode) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_jmp\_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jmp\_offset, 5);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == C\_CODER\_MODE) {

//

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

\*(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);

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_add\_esp\_4[] = { 0x83, 0xC4, 0x04 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_add\_esp\_4, 3);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

if (toMode) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@AFTER\_TO\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

else {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@AFTER\_DOWNTO\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL@EXIT\_FOR\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

currBytePtr += sprintf((char\*)currBytePtr, " add esp, 4; add esp, 8\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

if (toMode) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_AFTER\_TO\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

else {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_AFTER\_DOWNTO\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL\_\_EXIT\_FOR\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

currBytePtr += sprintf((char\*)currBytePtr, " --contextStackIndex;\r\n");

}

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

&&

!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI\_TOKEN\_FOR][0], MAX\_LEXEM\_SIZE)

&& (

!strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_DOWNTO][0], MAX\_LEXEM\_SIZE)

||

(toMode = !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_TO][0], MAX\_LEXEM\_SIZE))

)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\" (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0], tokenStruct[MULTI\_TOKEN\_FOR][0]);

}

currBytePtr = makePostForCode\_(lastLexemInfoInTable, currBytePtr, generatorMode, toMode);

lexemInfoTransformationTempStackSize -= 2;

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
GENERATOR.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

// TODO: CHANGE BY fRESET() TO END

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: generator.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//#define IDENTIFIER\_LEXEME\_TYPE 2

//#define VALUE\_LEXEME\_TYPE 4

//#define VALUE\_SIZE 4

#ifndef \_\_cplusplus

#define bool int

#define false 0

#define true 1

#endif

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

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

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

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

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

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

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

//#define DEBUG\_MODE\_BY\_ASSEMBLY

//#define C\_CODER\_MODE 0x01

//#define ASSEMBLY\_X86\_WIN32\_CODER\_MODE 0x02

//#define OBJECT\_X86\_WIN32\_CODER\_MODE 0x04

//#define MACHINE\_CODER\_MODE 0x08

//

//unsigned char generatorMode = MACHINE\_CODER\_MODE;

#define MAX\_TEXT\_SIZE 8192

#define MAX\_GENERATED\_TEXT\_SIZE (MAX\_TEXT\_SIZE \* 6)

#define GENERATED\_TEXT\_SIZE\_ 32768

//#define GENERATED\_TEXT\_SIZE (MAX\_TEXT\_SIZE % MAX\_GENERATED\_TEXT\_SIZE) // ?

#define SUCCESS\_STATE 0

#define MAX\_OUTTEXT\_SIZE (8\*8192\*1024)

unsigned char outText[MAX\_OUTTEXT\_SIZE] = ""; // !!!

#define MAX\_TEXT\_SIZE 8192

#define MAX\_WORD\_COUNT (MAX\_TEXT\_SIZE / 5)

#define MAX\_LEXEM\_SIZE 1024

#if 0

#define CODEGEN\_DATA\_TYPE int

#define START\_DATA\_OFFSET 512

#define OUT\_DATA\_OFFSET (START\_DATA\_OFFSET + 512)

#define M1 1024

#define M2 1024

//unsigned long long int dataOffsetMinusCodeOffset = 0x00003000;

unsigned long long int dataOffsetMinusCodeOffset = 0x00004000;

//unsigned long long int codeOffset = 0x000004AF;

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

unsigned long long int baseOperationOffset = 0x000004AF;

unsigned long long int putProcOffset = 0x0000001B;

unsigned long long int getProcOffset = 0x00000044;

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

#endif

struct LabelOffsetInfo {

char labelStr[MAX\_LEXEM\_SIZE];

unsigned char\* labelBytePtr;

// TODO: ...

};

struct LabelOffsetInfo labelsOffsetInfoTable[MAX\_WORD\_COUNT] = { { "", NULL/\*, 0, 0\*/ } };

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

struct GotoPositionInfo { // TODO: by Index

char labelStr[MAX\_LEXEM\_SIZE];

unsigned char\* gotoInstructionPositionPtr;

// TODO: ...

};

struct GotoPositionInfo gotoPositionsInfoTable[MAX\_WORD\_COUNT] = { { "", NULL/\*, 0, 0\*/ } }; // TODO: by Index

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

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

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

//unsigned char generatorMode = MACHINE\_CODER\_MODE;

char\* tokenStruct[MAX\_TOKEN\_STRUCT\_ELEMENT\_COUNT][MAX\_TOKEN\_STRUCT\_ELEMENT\_PART\_COUNT] = { NULL };

#if 0

static void intitTokenStruct\_\_OLD() {

//SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, MULTI\_TOKEN\_BITWISE\_NOT, ("~"), (""), (""), (""))

//

// a12345\_ptr = a12345;

//

tokenStruct[MULTI\_TOKEN\_BITWISE\_NOT][0] = (char\*)"~";

tokenStruct[MULTI\_TOKEN\_BITWISE\_AND][0] = (char\*)"&";

tokenStruct[MULTI\_TOKEN\_BITWISE\_OR][0] = (char\*)"|";

tokenStruct[MULTI\_TOKEN\_NOT][0] = (char\*)"NOT";

tokenStruct[MULTI\_TOKEN\_AND][0] = (char\*)"AND";

tokenStruct[MULTI\_TOKEN\_OR][0] = (char\*)"OR";

tokenStruct[MULTI\_TOKEN\_EQUAL][0] = (char\*)"==";

tokenStruct[MULTI\_TOKEN\_NOT\_EQUAL][0] = (char\*)"!=";

tokenStruct[MULTI\_TOKEN\_LESS][0] = (char\*)"<";

tokenStruct[MULTI\_TOKEN\_GREATER][0] = (char\*)">";

tokenStruct[MULTI\_TOKEN\_LESS\_OR\_EQUAL][0] = (char\*)"<=";

tokenStruct[MULTI\_TOKEN\_GREATER\_OR\_EQUAL][0] = (char\*)">=";

tokenStruct[MULTI\_TOKEN\_ADD][0] = (char\*)"+";

tokenStruct[MULTI\_TOKEN\_SUB][0] = (char\*)"-";

tokenStruct[MULTI\_TOKEN\_MUL][0] = (char\*)"\*";

tokenStruct[MULTI\_TOKEN\_DIV][0] = (char\*)"DIV";

tokenStruct[MULTI\_TOKEN\_MOD][0] = (char\*)"MOD";

tokenStruct[MULTI\_TOKEN\_BIND\_RIGHT\_TO\_LEFT][0] = (char\*)"<<";

tokenStruct[MULTI\_TOKEN\_BIND\_LEFT\_TO\_RIGHT][0] = (char\*)">>";

tokenStruct[MULTI\_TOKEN\_COLON][0] = (char\*)":";

tokenStruct[MULTI\_TOKEN\_GOTO][0] = (char\*)"GOTO";

tokenStruct[MULTI\_TOKEN\_IF][0] = (char\*)"IF"; tokenStruct[MULTI\_TOKEN\_IF][1] = (char\*)"(";

// tokenStruct[MULTI\_TOKEN\_IF\_][0] = (char\*)"IF"; // don't change this!

tokenStruct[MULTI\_TOKEN\_THEN][0] = (char\*)")";

// tokenStruct[MULTI\_TOKEN\_THEN\_][0] = (char\*)"NULL"; tokenStruct[MULTI\_TOKEN\_IF][1] = (char\*)"STATEMENT"; // don't change this!

tokenStruct[MULTI\_TOKEN\_ELSE][0] = (char\*)"ELSE";

tokenStruct[MULTI\_TOKEN\_FOR][0] = (char\*)"FOR";

tokenStruct[MULTI\_TOKEN\_TO][0] = (char\*)"TO";

tokenStruct[MULTI\_TOKEN\_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\*)"END"; tokenStruct[MULTI\_TOKEN\_END\_WHILE][1] = (char\*)"WHILE";

//

//

tokenStruct[MULTI\_TOKEN\_REPEAT][0] = (char\*)"REPEAT";

tokenStruct[MULTI\_TOKEN\_UNTIL][0] = (char\*)"UNTIL";

//

//

tokenStruct[MULTI\_TOKEN\_INPUT][0] = (char\*)"GET";

tokenStruct[MULTI\_TOKEN\_OUTPUT][0] = (char\*)"PUT";

//

//

tokenStruct[MULTI\_TOKEN\_RLBIND][0] = (char\*)"<<";

tokenStruct[MULTI\_TOKEN\_LRBIND][0] = (char\*)">>";

//

tokenStruct[MULTI\_TOKEN\_SEMICOLON][0] = (char\*)";";

tokenStruct[MULTI\_TOKEN\_BEGIN][0] = (char\*)"BEGIN";

tokenStruct[MULTI\_TOKEN\_END][0] = (char\*)"END";

tokenStruct[MULTI\_TOKEN\_NULL\_STATEMENT][0] = (char\*)"NULL"; tokenStruct[MULTI\_TOKEN\_NULL\_STATEMENT][1] = (char\*)"STATEMENT";

// NULL\_STATEMENT null\_statement

// null statement

//return 0;

}

//char intitTokenStruct\_ = (intitTokenStruct\_\_OLD(), 0);

#endif

INIT\_TOKEN\_STRUCT\_NAME(0);

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

if (lexemInfoTable == NULL) {

return false;

}

if (!strncmp(lexemInfoTable[0].lexemStr, tokenStruct[tokenStructName][0], MAX\_LEXEM\_SIZE)

&& (tokenStruct[tokenStructName][1] == NULL || tokenStruct[tokenStructName][1][0] == '\0' || !strncmp(lexemInfoTable[1].lexemStr, tokenStruct[tokenStructName][1], MAX\_LEXEM\_SIZE))

&& (tokenStruct[tokenStructName][2] == NULL || tokenStruct[tokenStructName][2][0] == '\0' || !strncmp(lexemInfoTable[2].lexemStr, tokenStruct[tokenStructName][2], MAX\_LEXEM\_SIZE))

&& (tokenStruct[tokenStructName][3] == NULL || tokenStruct[tokenStructName][3][0] == '\0' || !strncmp(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 (!strncmp(identifierIdsTable[index], identifierStr, MAX\_LEXEM\_SIZE)) {

return START\_DATA\_OFFSET + sizeof(CODEGEN\_DATA\_TYPE) \* index;

}

}

return OUT\_DATA\_OFFSET;

}

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

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

return currBytePtr;

}

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, "imul ebp, 4\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, "add esp, ebp\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, "xor ebp, ebp;\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " xor eax, eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ret\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "end start\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " return 0;\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "}");

}

return currBytePtr;

}

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, ".686\r\n");

currBytePtr += sprintf((char\*)currBytePtr, ".model flat, stdcall\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "option casemap : none\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "#define \_CRT\_SECURE\_NO\_WARNINGS\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, ".model flat, stdcall\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, "option casemap : none\r\n");

}

return currBytePtr;

}

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "GetStdHandle proto STDCALL, nStdHandle : DWORD\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "ExitProcess proto STDCALL, uExitCode : DWORD\r\n");

currBytePtr += sprintf((char\*)currBytePtr, ";MessageBoxA PROTO hwnd : DWORD, lpText : DWORD, lpCaption : DWORD, uType : DWORD\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "ReadConsoleA proto STDCALL, hConsoleInput : DWORD, lpBuffer : DWORD, nNumberOfCharsToRead : DWORD, lpNumberOfCharsRead : DWORD, lpReserved : DWORD\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "WriteConsoleA proto STDCALL, hConsoleOutput : DWORD, lpBuffert : DWORD, nNumberOfCharsToWrite : DWORD, lpNumberOfCharsWritten : DWORD, lpReserved : DWORD\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "wsprintfA PROTO C : VARARG\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "GetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, lpMode : DWORD\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "SetConsoleMode PROTO STDCALL, hConsoleHandle:DWORD, dwMode : DWORD\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "ENABLE\_LINE\_INPUT EQU 0002h\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "ENABLE\_ECHO\_INPUT EQU 0004h\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "#include \"stdio.h\"\r\n");

}

#ifdef DEBUG\_MODE\_BY\_ASSEMBLY

#endif

return currBytePtr;

}

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, ".data\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " data\_start db 8192 dup (0)\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;title\_msg db \"Output:\", 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " valueTemp\_msg db 256 dup(0)\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " valueTemp\_fmt db \"%%d\", 10, 13, 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;NumberOfCharsWritten dd 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " hConsoleInput dd 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " hConsoleOutput dd 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " buffer db 128 dup(0)\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " readOutCount dd ?\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "int data[8192] = {0};\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "int contextStack[8192] = {0}, contextStackIndex = 0;\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "int opStack[8192] = {0}, opStackIndex = 0, opTemp = 0;;\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "int lastBindDataIndex = 0;\r\n");

}

return currBytePtr;

}

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, ".code\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "start:\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "int main() {\r\n");

}

return currBytePtr;

}

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

// 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);

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " db 0E8h, 00h, 00h, 00h, 00h; call NexInstruction\r\n");

currBytePtr += sprintf((char\*)currBytePtr, ";NexInstruction:\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " pop esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub esi, 5\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov edi, esi\r\n");//printf(" mov edi, offset data\_start\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add edi, 0%08Xh\r\n", (int)dataOffsetMinusCodeOffset);

//currBytePtr += sprintf((char\*)currBytePtr, " xor ebp, ebp\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ecx, edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add ecx, 512\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " jmp initConsole\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " putProc PROC\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push offset valueTemp\_fmt\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push offset valueTemp\_msg\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " call wsprintfA\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add esp, 12\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push 40h\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push offset title\_msg\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push offset valueTemp\_msg;\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;call MessageBoxA\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push 0; offset NumberOfCharsWritten\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push eax; NumberOfCharsToWrite\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push offset valueTemp\_msg\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push hConsoleOutput\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " call WriteConsoleA\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ret\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " putProc ENDP\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " getProc PROC\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " push eax\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " push offset valueTemp\_fmt\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " push offset valueTemp\_msg\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " call wsprintfA\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " add esp, 12\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " push 40h\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " push offset title\_msg\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " push offset valueTemp\_msg;\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " push 0\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " call MessageBoxA\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " ret\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " getProc ENDP\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " getProc PROC\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push ebp\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ebp, esp\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push offset readOutCount\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push 15\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push offset buffer + 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push hConsoleInput\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " call ReadConsoleA\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " lea esi, offset buffer\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add esi, readOutCount\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub esi, 2\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " call string\_to\_int\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov esp, ebp\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " pop ebp\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ret\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " getProc ENDP\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " string\_to\_int PROC\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ; input: ESI - string\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ; output: EAX - value\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " xor eax, eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ebx, 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " xor ecx, ecx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "convert\_loop :\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " movzx ecx, byte ptr[esi]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " test ecx, ecx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " jz done\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, '0'\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " imul ecx, ebx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add eax, ecx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " imul ebx, ebx, 10\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " dec esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " jmp convert\_loop\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "done:\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ret\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " string\_to\_int ENDP\r\n");

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " initConsole:\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push -10\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " call GetStdHandle\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov hConsoleInput, eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push -11\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " call GetStdHandle\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov hConsoleOutput, eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " \r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push ecx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push ebx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push offset mode\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push hConsoleInput\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;call GetConsoleMode\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;mov ebx, eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;or ebx, ENABLE\_LINE\_INPUT \r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;or ebx, ENABLE\_ECHO\_INPUT\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push ebx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push hConsoleInput\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;call SetConsoleMode\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;pop edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;pop esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;pop ebx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;pop ecx\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

//currBytePtr += sprintf((char\*)currBytePtr, " int data[8192] = {0};\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " int contextStack[8192] = {0}, contextStackIndex = 0;\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " int opStack[8192] = {0}, opStackIndex = 0, opTemp = 0;;\r\n");

//currBytePtr += sprintf((char\*)currBytePtr, " int lastBindDataIndex = 0;\r\n");

//

currBytePtr += sprintf((char\*)currBytePtr, " contextStackIndex = 0;\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " opStackIndex = 0;\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = 0;\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " lastBindDataIndex = 0;\r\n");

}

#ifdef DEBUG\_MODE\_BY\_ASSEMBLY

#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/rlbind.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, unsigned char generatorMode) {

//return currBytePtr;

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

if(lastDataSectionLexemIndex == ~0) {

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

exit(0);

}

labelInfoTable.clear(); /\* It's not pretty. TODO: change it. \*/

\*lastLexemInfoInTable += lastDataSectionLexemIndex;

return currBytePtr;

}

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_mov\_ebp\_esp[] = { 0x8B, 0xEC };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_ebp\_esp, 2);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;hw stack save(save esp)\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ebp, esp\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

//

}

return currBytePtr;

}

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_mov\_esp\_ebp[] = { 0x8B, 0xE5 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_esp\_ebp, 2);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;hw stack reset(restore esp)\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov esp, ebp\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

//

}

#ifdef DEBUG\_MODE\_BY\_ASSEMBLY

#endif

return currBytePtr;

}

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

if (!strncmp((\*lastLexemInfoInTable)->lexemStr, T\_NAME\_0, MAX\_LEXEM\_SIZE)

|| !strncmp((\*lastLexemInfoInTable)->lexemStr, T\_DATA\_0, MAX\_LEXEM\_SIZE)

|| !strncmp((\*lastLexemInfoInTable)->lexemStr, T\_BODY\_0, MAX\_LEXEM\_SIZE)

|| !strncmp((\*lastLexemInfoInTable)->lexemStr, T\_DATA\_TYPE\_0, MAX\_LEXEM\_SIZE)

|| !strncmp((\*lastLexemInfoInTable)->lexemStr, T\_COMA\_0, MAX\_LEXEM\_SIZE)

|| !strncmp((\*lastLexemInfoInTable)->lexemStr, T\_END\_0, MAX\_LEXEM\_SIZE)

) {

return ++ \* lastLexemInfoInTable, currBytePtr;

}

return currBytePtr;

}

unsigned char\* createPattern() {

return NULL;

}

unsigned char\* getObjectCodeBytePtr(unsigned char\* baseBytePtr, unsigned char generatorMode) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

}

else if (generatorMode == C\_CODER\_MODE) {

}

return baseBytePtr + baseOperationObjectOffset;

}

unsigned char\* getImageCodeBytePtr(unsigned char\* baseBytePtr, unsigned char generatorMode) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

}

else if (generatorMode == C\_CODER\_MODE) {

}

return baseBytePtr + baseOperationOffset;

}

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

currBytePtr = makeTitle(lastLexemInfoInTable, currBytePtr, generatorMode);

currBytePtr = makeDependenciesDeclaration(lastLexemInfoInTable, currBytePtr, generatorMode);

currBytePtr = makeDataSection(lastLexemInfoInTable, currBytePtr, generatorMode);

currBytePtr = makeBeginProgramCode(lastLexemInfoInTable, currBytePtr, generatorMode);

lexemInfoTransformationTempStackSize = 0;

currBytePtr = makeInitCode(lastLexemInfoInTable, currBytePtr, generatorMode);

currBytePtr = initMake(lastLexemInfoInTable, currBytePtr, generatorMode);

currBytePtr = makeSaveHWStack(lastLexemInfoInTable, currBytePtr, generatorMode);

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(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

NON\_CONTEXT\_NULL\_STATEMENT(lastLexemInfoInTable\_, lastLexemInfoInTable, currBytePtr, generatorMode, NULL);

if (lastLexemInfoInTable\_ == \*lastLexemInfoInTable) {

currBytePtr = noMake(lastLexemInfoInTable, currBytePtr);

}

if (lastLexemInfoInTable\_ == \*lastLexemInfoInTable) {

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

exit(0);

}

}

currBytePtr = makeResetHWStack(lastLexemInfoInTable, currBytePtr, generatorMode);

currBytePtr = makeEndProgramCode(lastLexemInfoInTable, currBytePtr, generatorMode);

return currBytePtr;

}

//unsigned char outCode[GENERATED\_TEXT\_SIZE] = { '\0' };

void viewCode(unsigned char\* outCodePtr, unsigned long long int outCodePrintSize, unsigned char align) {

printf("\r\n; +0x0 +0x1 +0x2 +0x3 +0x4 +0x5 +0x6 +0x7 +0x8 +0x9 +0xA +0xB +0xC +0xD +0xE +0xF ");

printf("\r\n;0x00000000: ");

unsigned long long int outCodePrintIndex = outCodePrintSize - 1;

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

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

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

unsigned long long int indexMinus16 = index - align;

do {

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

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

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

}

else {

printf(" ");

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

}

} while (++indexMinus16 % align);

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

}

}

}  
  
GOTO\_LABEL.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: goto\_lable.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <string>

#include <map>

//#include <utility>

#include <stack>

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

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

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

#include "stdio.h"

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

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

unsigned char multitokenSize, multitokenSize\_ = detectMultiToken(\*lastLexemInfoInTable + 1, MULTI\_TOKEN\_NULL\_STATEMENT);

multitokenSize = detectMultiToken(\*lastLexemInfoInTable + multitokenSize\_ + 1, MULTI\_TOKEN\_COLON);

if (multitokenSize) {

multitokenSize += multitokenSize\_;

}

if ((\*lastLexemInfoInTable)->tokenType != IDENTIFIER\_LEXEME\_TYPE){

return currBytePtr;

}

if (multitokenSize++) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;ident \"%s\"(as label) previous \"%s\"\r\n", (\*lastLexemInfoInTable)->lexemStr, tokenStruct[MULTI\_TOKEN\_COLON][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //ident \"%s\"(as label) previous \"%s\"\r\n", (\*lastLexemInfoInTable)->lexemStr, tokenStruct[MULTI\_TOKEN\_COLON][0]);

}

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

while (!labelInfoTable[(\*lastLexemInfoInTable)->lexemStr].second.empty()) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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

}

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

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL@%016llX:\r\n", (unsigned long long int) & labelInfoTable[(\*lastLexemInfoInTable)->lexemStr].first);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL\_\_%016llX:\r\n", (unsigned long long int) & labelInfoTable[(\*lastLexemInfoInTable)->lexemStr].first);

}

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;

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_jmp\_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jmp\_offset, 5);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\" previous ident \"%s\"(as label)\r\n", tokenStruct[MULTI\_TOKEN\_GOTO][0], (\*lastLexemInfoInTable)[1].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\" previous ident \"%s\"(as label)\r\n", tokenStruct[MULTI\_TOKEN\_GOTO][0], (\*lastLexemInfoInTable)[1].lexemStr);

}

if (labelInfoTable.find((\*lastLexemInfoInTable)[1].lexemStr) == labelInfoTable.end()) {

labelInfoTable[(\*lastLexemInfoInTable)[1].lexemStr].first = ~0;

}

if (labelInfoTable[(\*lastLexemInfoInTable)[1].lexemStr].first == ~0) {

labelInfoTable[(\*lastLexemInfoInTable)[1].lexemStr].second.push((unsigned long long int)(currBytePtr - 4));

}

else if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

\*(unsigned int\*)(currBytePtr - 4) = (unsigned int)((unsigned char\*)labelInfoTable[(\*lastLexemInfoInTable)[1].lexemStr].first - currBytePtr);

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@%016llX\r\n", (unsigned long long int) & labelInfoTable[(\*lastLexemInfoInTable)[1].lexemStr].first);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_%016llX;\r\n", (unsigned long long int) & labelInfoTable[(\*lastLexemInfoInTable)[1].lexemStr].first);

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
GREATER.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_GREATER][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " setg al\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and eax, 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr[ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_GREATER][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] = opStack[opStackIndex - 1] > opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
GREATER\_OR\_EQUAL.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: greater\_or\_equal.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_GREATER\_OR\_EQUAL);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_GREATER\_OR\_EQUAL][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " setge al\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and eax, 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr[ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_GREATER\_OR\_EQUAL][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] = opStack[opStackIndex - 1] >= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
IF\_THEN.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: if\_then.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

#include "string.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_IF);

if (!multitokenSize

&& tokenStruct[MULTI\_TOKEN\_IF][1][0] == '('

&& !strncmp((\*lastLexemInfoInTable)->lexemStr, tokenStruct[MULTI\_TOKEN\_IF][0], MAX\_LEXEM\_SIZE)) {

multitokenSize = 1;

}

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_IF][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_IF][0]);

}

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = \*\*lastLexemInfoInTable;

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_THEN);

if (!multitokenSize && tokenStruct[MULTI\_TOKEN\_IF][1][0] == '(') {

multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_NULL\_STATEMENT);

}

if (multitokenSize

&& lexemInfoTransformationTempStackSize

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_IF][0], MAX\_LEXEM\_SIZE)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;after cond expresion (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_IF][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //after cond expresion (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_IF][0]);

}

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = \*\*lastLexemInfoInTable;

/\*The lexem info table will be changed and will need to be rebuilt. TODO: change the implementation!\*/strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_THEN][0], MAX\_LEXEM\_SIZE);

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, " cmp eax, 0\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jz LABEL@AFTER\_THEN\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " if (opTemp == 0) goto LABEL\_\_AFTER\_THEN\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}

unsigned char\* makePostThenCode\_(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, 1\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL@AFTER\_THEN\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = 1;\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL\_\_AFTER\_THEN\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

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

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI\_TOKEN\_IF][0], MAX\_LEXEM\_SIZE)

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_THEN][0], MAX\_LEXEM\_SIZE)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\" (after \"then\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0], tokenStruct[MULTI\_TOKEN\_IF][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\" (after \"then\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0], tokenStruct[MULTI\_TOKEN\_IF][0]);

}

currBytePtr = makePostThenCode\_(lastLexemInfoInTable, currBytePtr, generatorMode);

lexemInfoTransformationTempStackSize -= 2;

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
INPUT.CPP  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: input.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_INPUT);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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\_valueByAdrressInECX[] = { 0x8B, 0x19 };

const unsigned char code\_\_sub\_ecx\_4[] = { 0x83, 0xE9, 0x04 };

const unsigned char code\_\_add\_ebx\_edi[] = { 0x33, 0xDF };

const unsigned char code\_\_mov\_stackTopByEBX\_eax[] = { 0x89, 0x03 };

const unsigned char code\_\_mov\_ecx\_edi[] = { 0x8B, 0xCF };

const unsigned char code\_\_add\_ecx\_512[] = { 0x81, 0xC1, 0x00, 0x02, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_eax\_stackTopByECX, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_edx\_address, 5);

\*(unsigned int\*)&(currBytePtr[-4]) = (unsigned int)getProcOffset;

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_add\_edx\_esi, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_push\_ecx, 1);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_push\_ebx, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_push\_esi, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_push\_edi, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_call\_edx, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_pop\_edi, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_pop\_esi, 1);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_pop\_ebx, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_pop\_ecx, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_ebx\_valueByAdrressInECX, 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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_INPUT][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " mov edx, 0%08Xh\r\n", (unsigned int)getProcOffset);

currBytePtr += sprintf((char\*)currBytePtr, " add edx, esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push ecx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push ebx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " call edx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " pop edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " pop esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;pop ebx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " pop ecx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ebx, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add ebx, edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr [ebx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ecx, edi ; reset second stack\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add ecx, 512 ; reset second stack\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_INPUT][0]);

currBytePtr += sprintf((char\*)currBytePtr, " (void)scanf\_s(\"%%d\", &opTemp);\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " data[opStack[opStackIndex]] = opTemp, opStackIndex = 0;\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_LESS][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " setl al\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and eax, 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr[ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_LESS][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] = opStack[opStackIndex - 1] < opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: less\_or\_equal.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_LESS\_OR\_EQUAL);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_LESS\_OR\_EQUAL][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " setle al\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and eax, 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr[ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_LESS\_OR\_EQUAL][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] = opStack[opStackIndex - 1] <= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: lexica.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#include <fstream>

#include <iostream>

//#include <algorithm>

#include <iterator>

#include <regex>

//struct LexemInfo {

// char lexemStr[MAX\_LEXEM\_SIZE];

// unsigned int lexemId;

// unsigned int tokenType;

// unsigned int ifvalue;

// unsigned int row;

// unsigned int col;

// // TODO: ...

//};

#define MAX\_ACCESSORY\_STACK\_SIZE\_123 128

char tempStrFor\_123[MAX\_TEXT\_SIZE/\*?TODO:... MAX\_ACCESSORY\_STACK\_SIZE\_123 \* 64\*/] = {'\0'};

unsigned long long int tempStrForCurrIndex = 0;

struct LexemInfo lexemesInfoTable[MAX\_WORD\_COUNT];// = { { "", 0, 0, 0 } };

struct LexemInfo\* lastLexemInfoInTable = lexemesInfoTable; // first for begin

char identifierIdsTable[MAX\_WORD\_COUNT][MAX\_LEXEM\_SIZE] = { "" };

LexemInfo::LexemInfo() {

lexemStr[0] = '\0';

lexemId = 0;

tokenType = 0;

ifvalue = 0;

row = ~0;

col = ~0;

}

LexemInfo::LexemInfo(const char \* lexemStr, unsigned long long int lexemId, unsigned long long int tokenType, unsigned long long int ifvalue, unsigned long long int row, unsigned long long int col) {

strncpy(this->lexemStr, lexemStr, MAX\_LEXEM\_SIZE);

this->lexemId = lexemId;

this->tokenType = tokenType;

this->ifvalue = ifvalue;

this->row = row;

this->col = col;

}

LexemInfo::LexemInfo(const NonContainedLexemInfo& nonContainedLexemInfo){

strncpy(lexemStr, nonContainedLexemInfo.lexemStr, MAX\_LEXEM\_SIZE);

lexemId = nonContainedLexemInfo.lexemId;

tokenType = nonContainedLexemInfo.tokenType;

ifvalue = nonContainedLexemInfo.ifvalue;

row = nonContainedLexemInfo.row;

col = nonContainedLexemInfo.col;

}

NonContainedLexemInfo::NonContainedLexemInfo() {

(lexemStr = tempStrFor\_123 + tempStrForCurrIndex)[0] = '\0';

tempStrForCurrIndex += 32;// MAX\_LEXEM\_SIZE;

lexemId = 0;

tokenType = 0;

ifvalue = 0;

row = ~0;

col = ~0;

}

NonContainedLexemInfo::NonContainedLexemInfo(const LexemInfo& lexemInfo) {

//strncpy(lexemStr, lexemInfo.lexemStr, MAX\_LEXEM\_SIZE); //

lexemStr = (char\*)lexemInfo.lexemStr;

lexemId = lexemInfo.lexemId;

tokenType = lexemInfo.tokenType;

ifvalue = lexemInfo.ifvalue;

row = lexemInfo.row;

col = lexemInfo.col;

}

void printLexemes(struct LexemInfo\* lexemInfoTable, char printBadLexeme) {

if (printBadLexeme) {

printf("Bad lexeme:\r\n");

}

else {

printf("Lexemes table:\r\n");

}

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

//printf("index\t\tlexeme\t\tid\ttype\tifvalue\trow\tcol\r\n");

printf("index lexeme id type ifvalue row col\r\n");

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

for (unsigned long long int index = 0; (!index || !printBadLexeme) && lexemInfoTable[index].lexemStr[0] != '\0'; ++index) {

printf("%5llu%17s%12llu%10llu%11llu%4lld%8lld\r\n", index, lexemInfoTable[index].lexemStr, lexemInfoTable[index].lexemId, lexemInfoTable[index].tokenType, lexemInfoTable[index].ifvalue, lexemInfoTable[index].row, lexemInfoTable[index].col);

}

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

return;

}

void printLexemesToFile(struct LexemInfo\* lexemInfoTable, char printBadLexeme, const char\* filename) {

FILE\* file = fopen(filename, "wb");

if (!file) {

perror("Failed to open file");

return;

}

if (printBadLexeme) {

fprintf(file, "Bad lexeme:\r\n");

}

else {

fprintf(file, "Lexemes table:\r\n");

}

fprintf(file, "-------------------------------------------------------------------\r\n");

//fprintf(file, "index\t\tlexeme\t\tid\ttype\tifvalue\trow\tcol\r\n");

fprintf(file, "index lexeme id type ifvalue row col\r\n");

fprintf(file, "-------------------------------------------------------------------\r\n");

for (unsigned long long int index = 0; (!index || !printBadLexeme) && lexemInfoTable[index].lexemStr[0] != '\0'; ++index) {

fprintf(file, "%5llu%17s%12llu%10llu%11llu%4lld%8lld\r\n",

index,

lexemInfoTable[index].lexemStr,

lexemInfoTable[index].lexemId,

lexemInfoTable[index].tokenType,

lexemInfoTable[index].ifvalue,

lexemInfoTable[index].row,

lexemInfoTable[index].col);

}

fprintf(file, "-------------------------------------------------------------------\r\n\r\n");

fclose(file);

}

// get identifier id

unsigned int getIdentifierId(char(\*identifierIdsTable)[MAX\_LEXEM\_SIZE], char\* str) {

unsigned int index = 0;

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

if (!strncmp(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 = getIdentifierId(identifierIdsTable, lexemInfoInTable->lexemStr);

lexemInfoInTable->tokenType = IDENTIFIER\_LEXEME\_TYPE;

return SUCCESS\_STATE;

}

return ~SUCCESS\_STATE;

}

// try to get value

unsigned int tryToGetUnsignedValue(struct LexemInfo\* lexemInfoInTable) {

char unsignedvalues\_re[] = UNSIGNEDVALUES\_RE;

//char unsignedvalues\_re[] = "0|[1-9][0-9]\*";

if (std::regex\_match(std::string(lexemInfoInTable->lexemStr), std::regex(unsignedvalues\_re))) {

lexemInfoInTable->ifvalue = atoi(lastLexemInfoInTable->lexemStr);

lexemInfoInTable->lexemId = MAX\_VARIABLES\_COUNT + MAX\_KEYWORD\_COUNT;

lexemInfoInTable->tokenType = VALUE\_LEXEME\_TYPE;

return SUCCESS\_STATE;

}

return ~SUCCESS\_STATE;

}

int commentRemover(char\* text, const char\* openStrSpc, const char\* closeStrSpc) {

bool eofAlternativeCloseStrSpcType = false;

bool explicitCloseStrSpc = true;

if (!strcmp(closeStrSpc, "\n")) {

eofAlternativeCloseStrSpcType = true;

explicitCloseStrSpc = false;

}

unsigned int commentSpace = 0;

unsigned int textLength = strlen(text); // strnlen(text, MAX\_TEXT\_SIZE);

unsigned int openStrSpcLength = strlen(openStrSpc); // strnlen(openStrSpc, MAX\_TEXT\_SIZE);

unsigned int closeStrSpcLength = strlen(closeStrSpc); // strnlen(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 (!strncmp(text + index, closeStrSpc, closeStrSpcLength) && (explicitCloseStrSpc || commentSpace)) {

if (commentSpace == 1 && explicitCloseStrSpc) {

for (unsigned int index2 = 0; index2 < closeStrSpcLength; ++index2) {

text[index + index2] = ' ';

}

}

else if (commentSpace == 1 && !explicitCloseStrSpc) {

index += closeStrSpcLength - 1;

}

oneLevelComment ? commentSpace = !commentSpace : commentSpace = 0;

}

else if (!strncmp(text + index, openStrSpc, openStrSpcLength)) {

oneLevelComment ? commentSpace = !commentSpace : commentSpace = 1;

}

if (commentSpace && text[index] != ' ' && text[index] != '\t' && text[index] != '\r' && text[index] != '\n') {

text[index] = ' ';

}

}

if (commentSpace && !eofAlternativeCloseStrSpcType) {

return -1;

}

return 0;

}

void prepareKeyWordIdGetter(char\* keywords\_, char\* keywords\_re) {

if (keywords\_ == NULL || keywords\_re == NULL) {

return;

}

for (char\* keywords\_re\_ = keywords\_re, \*keywords\_\_ = keywords\_; (\*keywords\_re\_ != '\0') ? 1 : (\*keywords\_\_ = '\0', 0); (\*keywords\_re\_ != '\\' || (keywords\_re\_[1] != '+' && keywords\_re\_[1] != '\*' && keywords\_re\_[1] != '|')) ? \*keywords\_\_++ = \*keywords\_re\_ : 0, ++keywords\_re\_);

}

unsigned int getKeyWordId(char\* keywords\_, char\* lexemStr, unsigned int baseId) {

if (keywords\_ == NULL || lexemStr == NULL) {

return ~0;

}

char\* lexemInKeywords\_ = keywords\_;

size\_t lexemStrLen = strlen(lexemStr);

if (!lexemStrLen) {

return ~0;

}

for (; lexemInKeywords\_ = strstr(lexemInKeywords\_, lexemStr), lexemInKeywords\_ != NULL && lexemInKeywords\_[lexemStrLen] != '|' && lexemInKeywords\_[lexemStrLen] != '\0'; ++lexemInKeywords\_);

return lexemInKeywords\_ - keywords\_ + baseId;

}

// try to get KeyWord

char tryToGetKeyWord(struct LexemInfo\* lexemInfoInTable) {

char keywords\_re[] = KEYWORDS\_RE;

//char keywords\_re[] = ";|<<|>>|\\+|-|\\\*|,|==|!=|:|\\(|\\)|NAME|DATA|BODY|END|EXIT|CONTINUE|GET|PUT|IF|ELSE|FOR|TO|DOWNTO|DO|WHILE|REPEAT|UNTIL|GOTO|DIV|MOD|<=|>=|NOT|AND|OR|INTEGER16";

//char keywords\_re[] = ";|<<|\\+\\+|--|\\\*\\\*|==|\\(|\\)|!=|:|name|data|body|end|get|put|for|to|downto|do|while|continue|exit|repeat|until|if|goto|div|mod|le|ge|not|and|or|long|int";

char keywords\_[sizeof(keywords\_re)] = { '\0' };

prepareKeyWordIdGetter(keywords\_, keywords\_re);

if (std::regex\_match(std::string(lexemInfoInTable->lexemStr), std::regex(keywords\_re))) {

lexemInfoInTable->lexemId = getKeyWordId(keywords\_, lexemInfoInTable->lexemStr, MAX\_VARIABLES\_COUNT);

lexemInfoInTable->tokenType = KEYWORD\_LEXEME\_TYPE;

return SUCCESS\_STATE;

}

return ~SUCCESS\_STATE;

}

void setPositions(const char\* text, struct LexemInfo\* lexemInfoTable) {

unsigned long long int line\_number = 1;

const char\* pos = text, \* line\_start = text;

if (lexemInfoTable) while (\*pos != '\0' && lexemInfoTable->lexemStr[0] != '\0') {

const char\* line\_end = strchr(pos, '\n');

if (!line\_end) {

line\_end = text + strlen(text);

}

char line\_[4096], \* line = line\_; //!! TODO: ...

strncpy(line, pos, line\_end - pos);

line[line\_end - pos] = '\0';

for (char\* found\_pos; lexemInfoTable->lexemStr[0] != '\0' && (found\_pos = strstr(line, lexemInfoTable->lexemStr)); line += strlen(lexemInfoTable->lexemStr), ++lexemInfoTable) {

lexemInfoTable->row = line\_number;

lexemInfoTable->col = found\_pos - line\_ + 1ull;

}

line\_number++;

pos = line\_end;

if (\*pos == '\n') {

pos++;

}

}

}

struct LexemInfo lexicalAnalyze(struct LexemInfo\* lexemInfoInPtr, char(\*identifierIdsTable)[MAX\_LEXEM\_SIZE]) {

struct LexemInfo ifBadLexemeInfo; // = { 0 };

if (tryToGetKeyWord(lexemInfoInPtr) == SUCCESS\_STATE);

else if (tryToGetIdentifier(lexemInfoInPtr, identifierIdsTable) == SUCCESS\_STATE);

else if (tryToGetUnsignedValue(lexemInfoInPtr) == SUCCESS\_STATE);

else {

ifBadLexemeInfo.tokenType = UNEXPEXTED\_LEXEME\_TYPE;

}

return ifBadLexemeInfo;

}

struct LexemInfo tokenize(char\* text, struct LexemInfo\*\* lastLexemInfoInTable, char(\*identifierIdsTable)[MAX\_LEXEM\_SIZE], struct LexemInfo(\*lexicalAnalyzeFunctionPtr)(struct LexemInfo\*, char(\*)[MAX\_LEXEM\_SIZE])) {

char tokens\_re[] = TOKENS\_RE;

//char tokens\_re[] = ";|<<|>>|\\+|-|\\\*|,|==|!=|:|\\(|\\)|<=|>=|[\_0-9A-Za-z]+|[^ \t\r\f\v\n]";

//char tokens\_re[] = "<<|\\+\\+|--|\\\*\\\*|==|\\(|\\)|!=|[\_0-9A-Za-z]+|[^ \t\r\f\v\n]";

std::regex tokens\_re\_(tokens\_re);

struct LexemInfo ifBadLexemeInfo; // = { 0 };

std::string stringText(text);

for (std::sregex\_token\_iterator end, tokenIterator(stringText.begin(), stringText.end(), tokens\_re\_); tokenIterator != end; ++tokenIterator, ++ \* lastLexemInfoInTable) {

std::string str = \*tokenIterator;

strncpy((\*lastLexemInfoInTable)->lexemStr, str.c\_str(), MAX\_LEXEM\_SIZE);

if ((ifBadLexemeInfo = (\*lexicalAnalyzeFunctionPtr)(\*lastLexemInfoInTable, identifierIdsTable)).tokenType == UNEXPEXTED\_LEXEME\_TYPE) {

break;

}

}

setPositions(text, lexemesInfoTable);

if (ifBadLexemeInfo.tokenType == UNEXPEXTED\_LEXEME\_TYPE) {

strncpy(ifBadLexemeInfo.lexemStr, (\*lastLexemInfoInTable)->lexemStr, MAX\_LEXEM\_SIZE);

ifBadLexemeInfo.row = (\*lastLexemInfoInTable)->row;

ifBadLexemeInfo.col = (\*lastLexemInfoInTable)->col;

}

return ifBadLexemeInfo;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_LRBIND][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov ebx, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx - 4]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 8\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add ebx, edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr [ebx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ecx, edi ; reset second stack\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add ecx, 512 ; reset second stack\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_LRBIND][0]);

currBytePtr += sprintf((char\*)currBytePtr, " lastBindDataIndex = opStack[opStackIndex];\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " data[lastBindDataIndex] = opTemp = opStack[opStackIndex - 1], opStackIndex = 0;\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: machinecodegen\_addon.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

//#define DEBUG\_RECONSTRUCT

unsigned char buffer[256 \* 1024];

extern unsigned long long int a123\_array\_part\_count;

extern unsigned char\* a123[39];

extern unsigned long long int a123\_array\_part\_size[39];

extern unsigned long long int a123\_zero\_part\_count;

extern unsigned long long int a123\_zeros[39];

extern unsigned long long int o123\_array\_part\_count;

extern unsigned char\* o123[5];

extern unsigned long long int o123\_array\_part\_size[5];

extern unsigned long long int o123\_zero\_part\_count;

extern unsigned long long int o123\_zeros[4];

unsigned long long int buildTemplateForCodeObject(unsigned char\* byteImage) {

if (!byteImage) {

exit(EXIT\_FAILURE);

}

unsigned long long int byteIndex = 0;

unsigned long long int totalByteCount = 0;

unsigned long long int array\_index = 0;

unsigned long long int zero\_index = 0;

for (unsigned long long int i = 0; i < o123\_array\_part\_count + o123\_zero\_part\_count; ++i) {

if (i % 2 == 0) {

unsigned char\* current\_array = NULL;

current\_array = o123[array\_index++];

if (current\_array) {

//fwrite(current\_array, 1, o123\_array\_part\_size[array\_index - 1]/\*sizeof(current\_array)\*/, outfile);

for (unsigned long long int localByteIndex = 0; localByteIndex < o123\_array\_part\_size[array\_index - 1]; ++localByteIndex) {

byteImage[byteIndex++] = current\_array[localByteIndex];

}

#ifdef DEBUG\_RECONSTRUCT

printf(": sizeof(current\_array) = %llu\r\n", o123\_array\_part\_size[array\_index - 1]/\*sizeof(current\_array)\*/);

#endif

totalByteCount += o123\_array\_part\_size[array\_index - 1];

}

}

else {

unsigned long long int zero\_count = 0;

zero\_count = o123\_zeros[zero\_index++];

if (zero\_count > 0) {

unsigned char\* zeros = (unsigned char\*)calloc(zero\_count, sizeof(unsigned char));

if (!zeros) {

perror("Memory allocation failed for zero block");

//fclose(outfile);

exit(EXIT\_FAILURE);

}

//fwrite(zeros, 1, zero\_count, outfile);

for (unsigned long long int localByteIndex = 0; localByteIndex < zero\_count; ++localByteIndex) {

byteImage[byteIndex++] = 0;

}

#ifdef DEBUG\_RECONSTRUCT

printf("zero\_count = %llu\r\n", zero\_count);

#endif

totalByteCount += zero\_count;

free(zeros);

}

}

}

//fclose(outfile);

#ifdef DEBUG\_RECONSTRUCT

printf("Image reconstructed.\n"/\*, output\_file\*/);

#endif

return byteIndex;

}

unsigned long long int buildTemplateForCodeImage(unsigned char\* byteImage) {

if (!byteImage) {

exit(EXIT\_FAILURE);

}

unsigned long long int byteIndex = 0;

unsigned long long int totalByteCount = 0;

unsigned long long int array\_index = 0;

unsigned long long int zero\_index = 0;

for (unsigned long long int i = 0; i < a123\_array\_part\_count + a123\_zero\_part\_count; ++i) {

if (i % 2 == 0) {

unsigned char\* current\_array = NULL;

current\_array = a123[array\_index++];

if (current\_array) {

//fwrite(current\_array, 1, a123\_array\_part\_size[array\_index - 1]/\*sizeof(current\_array)\*/, outfile);

for (unsigned long long int localByteIndex = 0; localByteIndex < a123\_array\_part\_size[array\_index - 1]; ++localByteIndex) {

byteImage[byteIndex++] = current\_array[localByteIndex];

}

#ifdef DEBUG\_RECONSTRUCT

printf(": sizeof(current\_array) = %llu\r\n", a123\_array\_part\_size[array\_index - 1]/\*sizeof(current\_array)\*/);

#endif

totalByteCount += a123\_array\_part\_size[array\_index - 1];

}

}

else {

unsigned long long int zero\_count = 0;

zero\_count = a123\_zeros[zero\_index++];

if (zero\_count > 0) {

unsigned char\* zeros = (unsigned char\*)calloc(zero\_count, sizeof(unsigned char));

if (!zeros) {

perror("Memory allocation failed for zero block");

//fclose(outfile);

exit(EXIT\_FAILURE);

}

//fwrite(zeros, 1, zero\_count, outfile);

for (unsigned long long int localByteIndex = 0; localByteIndex < zero\_count; ++localByteIndex) {

byteImage[byteIndex++] = 0;

}

#ifdef DEBUG\_RECONSTRUCT

printf("zero\_count = %llu\r\n", zero\_count);

#endif

totalByteCount += zero\_count;

free(zeros);

}

}

}

//fclose(outfile);

#ifdef DEBUG\_RECONSTRUCT

printf("Image reconstructed.\n"/\*, output\_file\*/);

#endif

return byteIndex;

}

void writeBytesToFile(const char\* output\_file, unsigned char\* byteImage, unsigned long long int imageSize) {

if (!output\_file || !byteImage) {

perror("Error in write image to file");

exit(EXIT\_FAILURE);

}

FILE\* outfile = fopen(output\_file, "wb");

if (!outfile) {

perror("Error opening output file");

exit(EXIT\_FAILURE);

}

if (imageSize) {

fwrite(byteImage, 1, imageSize, outfile);

}

fclose(outfile);

printf("File \"%s\" saved.\n", output\_file);

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: machinecodegen\_pattern.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned char a123\_0[] = {

0x4D,0x5A,0x90,0x00,0x03,0x00,0x00,0x00,0x04,0x00,0x00,0x00,0xFF,0xFF,0x00,0x00,

0xB8,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,

};

#define a123\_0\_SIZE 25

#define a123\_ZEROS\_0 35

unsigned char a123\_1[] = {

0xD0,0x00,0x00,0x00,0x0E,0x1F,0xBA,0x0E,0x00,0xB4,0x09,0xCD,0x21,0xB8,0x01,0x4C,

0xCD,0x21,0x54,0x68,0x69,0x73,0x20,0x70,0x72,0x6F,0x67,0x72,0x61,0x6D,0x20,0x63,

0x61,0x6E,0x6E,0x6F,0x74,0x20,0x62,0x65,0x20,0x72,0x75,0x6E,0x20,0x69,0x6E,0x20,

0x44,0x4F,0x53,0x20,0x6D,0x6F,0x64,0x65,0x2E,0x0D,0x0D,0x0A,0x24,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0xBD,0x32,0xF9,0xFD,0xF9,0x53,0x97,0xAE,0xF9,0x53,0x97,0xAE,

0xF9,0x53,0x97,0xAE,0xED,0x38,0x96,0xAF,0xFC,0x53,0x97,0xAE,0xF9,0x53,0x96,0xAE,

0xFD,0x53,0x97,0xAE,0x1D,0x27,0x94,0xAF,0xF8,0x53,0x97,0xAE,0x1D,0x27,0x68,0xAE,

0xF8,0x53,0x97,0xAE,0x1D,0x27,0x95,0xAF,0xF8,0x53,0x97,0xAE,0x52,0x69,0x63,0x68,

0xF9,0x53,0x97,0xAE,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x50,0x45,0x00,0x00,0x4C,0x01,0x05,0x00,0xCA,0x22,0x65,0x67,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xE0,0x00,0x02,0x01,0x0B,0x01,0x0E,0x1C,

0x00,0x22,0x00,0x00,0x00,0x2A,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x10,0x00,0x00,

0x00,0x10,0x00,0x00,0x00,0x40,0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x10,0x00,0x00,

0x00,0x02,0x00,0x00,0x06,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x06,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0xA0,0x00,0x00,0x00,0x04,0x00,0x00,0x00,0x00,0x00,0x00,

0x03,0x00,0x40,0x81,0x00,0x00,0x10,0x00,0x00,0x10,0x00,0x00,0x00,0x00,0x10,0x00,

0x00,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0xCC,0x41,0x00,0x00,0x3C,0x00,0x00,0x00,0x00,0x80,0x00,0x00,

0xE0,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x90,0x00,0x00,0x30,0x00,0x00,0x00,0x18,0x40,0x00,0x00,

0x70,

};

#define a123\_1\_SIZE 321

#define a123\_ZEROS\_1 44

unsigned char a123\_2[] = {

0x40,0x00,0x00,0x18,

};

#define a123\_2\_SIZE 4

#define a123\_ZEROS\_2 27

unsigned char a123\_3[] = {

0x2E,0x74,0x65,0x78,0x74,0x00,0x00,0x00,0xCA,0x20,0x00,0x00,0x00,0x10,0x00,0x00,

0x00,0x22,0x00,0x00,0x00,0x04,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x20,0x00,0x00,0x60,0x2E,0x72,0x64,0x61,0x74,0x61,0x00,0x00,

0x76,0x02,0x00,0x00,0x00,0x40,0x00,0x00,0x00,0x04,0x00,0x00,0x00,0x26,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x40,

0x2E,0x64,0x61,0x74,0x61,0x00,0x00,0x00,0x91,0x21,0x00,0x00,0x00,0x50,0x00,0x00,

0x00,0x22,0x00,0x00,0x00,0x2A,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x40,0x00,0x00,0xC0,0x2E,0x72,0x73,0x72,0x63,0x00,0x00,0x00,

0xE0,0x01,0x00,0x00,0x00,0x80,0x00,0x00,0x00,0x02,0x00,0x00,0x00,0x4C,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x40,

0x2E,0x72,0x65,0x6C,0x6F,0x63,0x00,0x00,0x30,0x00,0x00,0x00,0x00,0x90,0x00,0x00,

0x00,0x02,0x00,0x00,0x00,0x4E,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x42,

};

#define a123\_3\_SIZE 200

#define a123\_ZEROS\_3 368

unsigned char a123\_4[] = {

0xE8,0x00,0x00,0x00,0x00,0x5E,0x83,0xEE,0x05,0x8B,0xFE,0x81,0xC7,0x00,0x40,0x00,

0x00,0x8B,0xCF,0x81,0xC1,0x00,0x02,0x00,0x00,0xEB,0x7C,0x50,0x68,0x00,0x71,0x40,

0x00,0x68,0x00,0x70,0x40,0x00,0xE8,0x99,0x20,0x00,0x00,0x83,0xC4,0x0C,0x6A,0x00,

0x6A,0x00,0x50,0x68,0x00,0x70,0x40,0x00,0xFF,0x35,0x09,0x71,0x40,0x00,0xE8,0x7B,

0x20,0x00,0x00,0xC3,0x55,0x8B,0xEC,0x6A,0x00,0x68,0x8D,0x71,0x40,0x00,0x6A,0x0F,

0x68,0x0E,0x71,0x40,0x00,0xFF,0x35,0x05,0x71,0x40,0x00,0xE8,0x58,0x20,0x00,0x00,

0x8D,0x35,0x0D,0x71,0x40,0x00,0x03,0x35,0x8D,0x71,0x40,0x00,0x83,0xEE,0x02,0xE8,

0x04,0x00,0x00,0x00,0x8B,0xE5,0x5D,0xC3,0x33,0xC0,0xBB,0x01,0x00,0x00,0x00,0x33,

0xC9,0x0F,0xB6,0x0E,0x85,0xC9,0x74,0x0E,0x83,0xE9,0x30,0x0F,0xAF,0xCB,0x03,0xC1,

0x6B,0xDB,0x0A,0x4E,0xEB,0xEB,0xC3,0x6A,0xF6,0xE8,0x14,0x20,0x00,0x00,0xA3,0x05,

0x71,0x40,0x00,0x6A,0xF5,0xE8,0x08,0x20,0x00,0x00,0xA3,0x09,0x71,0x40,0x00,0x33,

0xC0,0xC3,

};

#define a123\_4\_SIZE 178

#define a123\_ZEROS\_4 8192

unsigned char a123\_5[] = {

0xFF,0x25,0x08,0x40,0x40,0x00,0xFF,0x25,0x00,0x40,0x40,0x00,0xFF,0x25,0x04,0x40,

0x40,0x00,0xFF,0x25,0x10,0x40,0x40,

};

#define a123\_5\_SIZE 23

#define a123\_ZEROS\_5 311

unsigned char a123\_6[] = {

0x30,0x42,0x00,0x00,0x40,0x42,0x00,0x00,0x20,0x42,0x00,0x00,0x00,0x00,0x00,0x00,

0x5E,0x42,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xCA,0x22,0x65,0x67,

0x00,0x00,0x00,0x00,0x02,0x00,0x00,0x00,0x54,0x00,0x00,0x00,0x88,0x40,0x00,0x00,

0x88,0x26,0x00,0x00,0x00,0x00,0x00,0x00,0xCA,0x22,0x65,0x67,0x00,0x00,0x00,0x00,

0x0C,0x00,0x00,0x00,0x14,0x00,0x00,0x00,0xDC,0x40,0x00,0x00,0xDC,0x26,0x00,0x00,

0x00,0x00,0x00,0x00,0xCA,0x22,0x65,0x67,0x00,0x00,0x00,0x00,0x0D,0x00,0x00,0x00,

0xDC,0x00,0x00,0x00,0xF0,0x40,0x00,0x00,0xF0,0x26,0x00,0x00,0x00,0x00,0x00,0x00,

0xCA,0x22,0x65,0x67,0x00,0x00,0x00,0x00,0x0E,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x52,0x53,0x44,0x53,0x05,0x39,0xA1,0x32,

0x25,0x9B,0x91,0x44,0x8E,0x6F,0x3B,0x20,0xBC,0x79,0x50,0x25,0x22,0x00,0x00,0x00,

0x43,0x3A,0x5C,0x55,0x73,0x65,0x72,0x73,0x5C,0x4E,0x61,0x7A,0x61,0x72,0x5C,0x73,

0x6F,0x75,0x72,0x63,0x65,0x5C,0x72,0x65,0x70,0x6F,0x73,0x5C,0x50,0x72,0x6F,0x6A,

0x65,0x63,0x74,0x36,0x30,0x5C,0x52,0x65,0x6C,0x65,0x61,0x73,0x65,0x5C,0x50,0x72,

0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x2E,0x70,0x64,0x62,

};

#define a123\_6\_SIZE 219

#define a123\_ZEROS\_6 21

unsigned char a123\_7[] = {

0x47,0x43,0x54,0x4C,0x00,0x10,0x00,0x00,0xCA,0x20,0x00,0x00,0x2E,0x74,0x65,0x78,

0x74,0x24,0x6D,0x6E,0x00,0x00,0x00,0x00,0x00,0x40,0x00,0x00,0x18,0x00,0x00,0x00,

0x2E,0x69,0x64,0x61,0x74,0x61,0x24,0x35,0x00,0x00,0x00,0x00,0x18,0x40,0x00,0x00,

0x70,0x00,0x00,0x00,0x2E,0x72,0x64,0x61,0x74,0x61,0x00,0x00,0x88,0x40,0x00,0x00,

0x44,0x01,0x00,0x00,0x2E,0x72,0x64,0x61,0x74,0x61,0x24,0x7A,0x7A,0x7A,0x64,0x62,

0x67,0x00,0x00,0x00,0xCC,0x41,0x00,0x00,0x28,0x00,0x00,0x00,0x2E,0x69,0x64,0x61,

0x74,0x61,0x24,0x32,0x00,0x00,0x00,0x00,0xF4,0x41,0x00,0x00,0x14,0x00,0x00,0x00,

0x2E,0x69,0x64,0x61,0x74,0x61,0x24,0x33,0x00,0x00,0x00,0x00,0x08,0x42,0x00,0x00,

0x18,0x00,0x00,0x00,0x2E,0x69,0x64,0x61,0x74,0x61,0x24,0x34,0x00,0x00,0x00,0x00,

0x20,0x42,0x00,0x00,0x56,0x00,0x00,0x00,0x2E,0x69,0x64,0x61,0x74,0x61,0x24,0x36,

0x00,0x00,0x00,0x00,0x00,0x50,0x00,0x00,0x91,0x21,0x00,0x00,0x2E,0x64,0x61,0x74,

0x61,0x00,0x00,0x00,0x00,0x80,0x00,0x00,0x60,0x00,0x00,0x00,0x2E,0x72,0x73,0x72,

0x63,0x24,0x30,0x31,0x00,0x00,0x00,0x00,0x60,0x80,0x00,0x00,0x80,0x01,0x00,0x00,

0x2E,0x72,0x73,0x72,0x63,0x24,0x30,0x32,0x00,0x00,0x00,0x00,0x08,0x42,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x50,0x42,0x00,0x00,0x00,0x40,0x00,0x00,

0x18,0x42,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x6A,0x42,0x00,0x00,

0x10,0x40,

};

#define a123\_7\_SIZE 258

#define a123\_ZEROS\_7 22

unsigned char a123\_8[] = {

0x30,0x42,0x00,0x00,0x40,0x42,0x00,0x00,0x20,0x42,0x00,0x00,0x00,0x00,0x00,0x00,

0x5E,0x42,0x00,0x00,0x00,0x00,0x00,0x00,0xD5,0x02,0x47,0x65,0x74,0x53,0x74,0x64,

0x48,0x61,0x6E,0x64,0x6C,0x65,0x00,0x00,0x68,0x04,0x52,0x65,0x61,0x64,0x43,0x6F,

0x6E,0x73,0x6F,0x6C,0x65,0x41,0x00,0x00,0x0B,0x06,0x57,0x72,0x69,0x74,0x65,0x43,

0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x41,0x00,0x4B,0x45,0x52,0x4E,0x45,0x4C,0x33,0x32,

0x2E,0x64,0x6C,0x6C,0x00,0x00,0xE1,0x03,0x77,0x73,0x70,0x72,0x69,0x6E,0x74,0x66,

0x41,0x00,0x55,0x53,0x45,0x52,0x33,0x32,0x2E,0x64,0x6C,0x6C,

};

#define a123\_8\_SIZE 108

#define a123\_ZEROS\_8 8844

unsigned char a123\_9[] = {

0x25,0x64,0x0A,0x0D,

};

#define a123\_9\_SIZE 4

#define a123\_ZEROS\_9 266

unsigned char a123\_10[] = {

0x01,0x00,0x18,0x00,0x00,0x00,0x18,0x00,0x00,0x80,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x00,0x01,0x00,0x00,0x00,0x30,0x00,

0x00,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x01,0x00,0x09,0x04,0x00,0x00,0x48,0x00,0x00,0x00,0x60,0x80,0x00,0x00,0x7D,0x01,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x3C,0x3F,0x78,0x6D,0x6C,0x20,0x76,0x65,0x72,0x73,0x69,0x6F,0x6E,0x3D,

0x27,0x31,0x2E,0x30,0x27,0x20,0x65,0x6E,0x63,0x6F,0x64,0x69,0x6E,0x67,0x3D,0x27,

0x55,0x54,0x46,0x2D,0x38,0x27,0x20,0x73,0x74,0x61,0x6E,0x64,0x61,0x6C,0x6F,0x6E,

0x65,0x3D,0x27,0x79,0x65,0x73,0x27,0x3F,0x3E,0x0D,0x0A,0x3C,0x61,0x73,0x73,0x65,

0x6D,0x62,0x6C,0x79,0x20,0x78,0x6D,0x6C,0x6E,0x73,0x3D,0x27,0x75,0x72,0x6E,0x3A,

0x73,0x63,0x68,0x65,0x6D,0x61,0x73,0x2D,0x6D,0x69,0x63,0x72,0x6F,0x73,0x6F,0x66,

0x74,0x2D,0x63,0x6F,0x6D,0x3A,0x61,0x73,0x6D,0x2E,0x76,0x31,0x27,0x20,0x6D,0x61,

0x6E,0x69,0x66,0x65,0x73,0x74,0x56,0x65,0x72,0x73,0x69,0x6F,0x6E,0x3D,0x27,0x31,

0x2E,0x30,0x27,0x3E,0x0D,0x0A,0x20,0x20,0x3C,0x74,0x72,0x75,0x73,0x74,0x49,0x6E,

0x66,0x6F,0x20,0x78,0x6D,0x6C,0x6E,0x73,0x3D,0x22,0x75,0x72,0x6E,0x3A,0x73,0x63,

0x68,0x65,0x6D,0x61,0x73,0x2D,0x6D,0x69,0x63,0x72,0x6F,0x73,0x6F,0x66,0x74,0x2D,

0x63,0x6F,0x6D,0x3A,0x61,0x73,0x6D,0x2E,0x76,0x33,0x22,0x3E,0x0D,0x0A,0x20,0x20,

0x20,0x20,0x3C,0x73,0x65,0x63,0x75,0x72,0x69,0x74,0x79,0x3E,0x0D,0x0A,0x20,0x20,

0x20,0x20,0x20,0x20,0x3C,0x72,0x65,0x71,0x75,0x65,0x73,0x74,0x65,0x64,0x50,0x72,

0x69,0x76,0x69,0x6C,0x65,0x67,0x65,0x73,0x3E,0x0D,0x0A,0x20,0x20,0x20,0x20,0x20,

0x20,0x20,0x20,0x3C,0x72,0x65,0x71,0x75,0x65,0x73,0x74,0x65,0x64,0x45,0x78,0x65,

0x63,0x75,0x74,0x69,0x6F,0x6E,0x4C,0x65,0x76,0x65,0x6C,0x20,0x6C,0x65,0x76,0x65,

0x6C,0x3D,0x27,0x61,0x73,0x49,0x6E,0x76,0x6F,0x6B,0x65,0x72,0x27,0x20,0x75,0x69,

0x41,0x63,0x63,0x65,0x73,0x73,0x3D,0x27,0x66,0x61,0x6C,0x73,0x65,0x27,0x20,0x2F,

0x3E,0x0D,0x0A,0x20,0x20,0x20,0x20,0x20,0x20,0x3C,0x2F,0x72,0x65,0x71,0x75,0x65,

0x73,0x74,0x65,0x64,0x50,0x72,0x69,0x76,0x69,0x6C,0x65,0x67,0x65,0x73,0x3E,0x0D,

0x0A,0x20,0x20,0x20,0x20,0x3C,0x2F,0x73,0x65,0x63,0x75,0x72,0x69,0x74,0x79,0x3E,

0x0D,0x0A,0x20,0x20,0x3C,0x2F,0x74,0x72,0x75,0x73,0x74,0x49,0x6E,0x66,0x6F,0x3E,

0x0D,0x0A,0x3C,0x2F,0x61,0x73,0x73,0x65,0x6D,0x62,0x6C,0x79,0x3E,0x0D,0x0A,

};

#define a123\_10\_SIZE 463

#define a123\_ZEROS\_10 36

#define a123\_ZEROS\_11 464

unsigned char a123\_11[] = {

0x10,0x00,0x00,0x20,0x00,0x00,0x00,0x1D,0x30,0x22,0x30,0x34,0x30,0x3A,0x30,0x4A,

0x30,0x51,0x30,0x57,0x30,0x62,0x30,0x68,0x30,0x9F,0x30,0xAB,0x30,0x00,0x00,0x00,

0x30,0x00,0x00,0x10,0x00,0x00,0x00,0xB4,0x30,0xBA,0x30,0xC0,0x30,0xC6,0x30,

};

#define a123\_11\_SIZE 47

unsigned long long int a123\_array\_part\_count = 12;

unsigned char \* a123[12] = {

a123\_0

, a123\_1

, a123\_2

, a123\_3

, a123\_4

, a123\_5

, a123\_6

, a123\_7

, a123\_8

, a123\_9

, a123\_10

, a123\_11

};

unsigned long long int a123\_array\_part\_size[12] = {

a123\_0\_SIZE

, a123\_1\_SIZE

, a123\_2\_SIZE

, a123\_3\_SIZE

, a123\_4\_SIZE

, a123\_5\_SIZE

, a123\_6\_SIZE

, a123\_7\_SIZE

, a123\_8\_SIZE

, a123\_9\_SIZE

, a123\_10\_SIZE

, a123\_11\_SIZE

};

unsigned long long int a123\_zero\_part\_count = 12;

unsigned long long int a123\_zeros[12] = {

a123\_ZEROS\_0

, a123\_ZEROS\_1

, a123\_ZEROS\_2

, a123\_ZEROS\_3

, a123\_ZEROS\_4

, a123\_ZEROS\_5

, a123\_ZEROS\_6

, a123\_ZEROS\_7

, a123\_ZEROS\_8

, a123\_ZEROS\_9

, a123\_ZEROS\_10

, a123\_ZEROS\_11

};  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: mod.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

unsigned char\* makeModCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode) { // task

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_MOD);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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\_eax\_edx[] = { 0x8B, 0xC2 };

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\_eax\_edx, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_toAddrFromECX\_eax, 2);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192," ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_MOD][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx - 4]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cdq\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " idiv dword ptr [ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, edx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr [ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_MOD][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] %= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: mul.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_MUL);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_MUL][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx - 4]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;cdq\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " imul dword ptr [ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr [ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_MUL][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] \*= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: not.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_NOT);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_NOT][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp eax, 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sete al\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and eax, 1\r\n");

//

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr[ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_NOT][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex] = !opStack[opStackIndex];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: not\_equal.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_NOT\_EQUAL);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_mov\_eax\_stackTopByECX[] = { 0x8B, 0x01 };

const unsigned char code\_\_sub\_ecx\_4[] = { 0x83, 0xE9, 0x04 };

const unsigned char code\_\_cmp\_stackTopByECX\_eax[] = { 0x39, 0x01 };

const unsigned char code\_\_setne\_al[] = { 0x0F, 0x95, 0xC0 };

const unsigned char code\_\_and\_eax\_1[] = { 0x83, 0xE0, 0x01 };

const unsigned char code\_\_mov\_stackTopByECX\_eax[] = { 0x89, 0x01 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_eax\_stackTopByECX, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_sub\_ecx\_4, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_cmp\_stackTopByECX\_eax, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_setne\_al, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_and\_eax\_1, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_stackTopByECX\_eax, 2);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_NOT\_EQUAL][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " setne al\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and eax, 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr[ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_NOT\_EQUAL][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] = opStack[opStackIndex - 1] != opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: null\_statement.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_NULL\_STATEMENT);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;null statement (non-context)\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " //null statement (non-context)\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
// Generated sparse arrays

unsigned char o123\_0[] = {

0x4C,0x01,0x05,0x00,0x85,0x62,0x84,0x67,0x86,0x4B,0x00,0x00,0x1F,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x2E,0x74,0x65,0x78,0x74,0x24,0x6D,0x6E,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0xB2,0x20,0x00,0x00,0xDC,0x00,0x00,0x00,0x8E,0x21,0x00,0x00,

0x00,0x00,0x00,0x00,0x10,0x00,0x00,0x00,0x20,0x00,0x50,0x60,0x2E,0x64,0x61,0x74,

0x61,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x91,0x21,0x00,0x00,

0x2E,0x22,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x40,0x00,0x50,0xC0,0x2E,0x64,0x65,0x62,0x75,0x67,0x24,0x53,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x4C,0x05,0x00,0x00,0xBF,0x43,0x00,0x00,0x0C,0x49,0x00,0x00,

0x00,0x00,0x00,0x00,0x1E,0x00,0x00,0x00,0x40,0x00,0x10,0x42,0x2E,0x64,0x65,0x62,

0x75,0x67,0x24,0x54,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0x01,0x00,0x00,

0x38,0x4A,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x40,0x00,0x10,0x42,0x2E,0x64,0x72,0x65,0x63,0x74,0x76,0x65,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x0D,0x00,0x00,0x00,0x78,0x4B,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0A,0x00,0x00,0xE8,0x00,0x00,0x00,

0x00,0x5E,0x83,0xEE,0x05,0x8B,0xFE,0x81,0xC7,0x00,0x40,0x00,0x00,0x8B,0xCF,0x81,

0xC1,0x00,0x02,0x00,0x00,0xEB,0x7C,0x50,0x68,0x00,0x00,0x00,0x00,0x68,0x00,0x00,

0x00,0x00,0xE8,0x00,0x00,0x00,0x00,0x83,0xC4,0x0C,0x6A,0x00,0x6A,0x00,0x50,0x68,

0x00,0x00,0x00,0x00,0xFF,0x35,0x00,0x00,0x00,0x00,0xE8,0x00,0x00,0x00,0x00,0xC3,

0x55,0x8B,0xEC,0x6A,0x00,0x68,0x00,0x00,0x00,0x00,0x6A,0x0F,0x68,0x01,0x00,0x00,

0x00,0xFF,0x35,0x00,0x00,0x00,0x00,0xE8,0x00,0x00,0x00,0x00,0x8D,0x35,0x00,0x00,

0x00,0x00,0x03,0x35,0x00,0x00,0x00,0x00,0x83,0xEE,0x02,0xE8,0x04,0x00,0x00,0x00,

0x8B,0xE5,0x5D,0xC3,0x33,0xC0,0xBB,0x01,0x00,0x00,0x00,0x33,0xC9,0x0F,0xB6,0x0E,

0x85,0xC9,0x74,0x0E,0x83,0xE9,0x30,0x0F,0xAF,0xCB,0x03,0xC1,0x6B,0xDB,0x0A,0x4E,

0xEB,0xEB,0xC3,0x6A,0xF6,0xE8,0x00,0x00,0x00,0x00,0xA3,0x00,0x00,0x00,0x00,0x6A,

0xF5,0xE8,0x00,0x00,0x00,0x00,0xA3,0x00,0x00,0x00,0x00,0x33,0xC0,0xC3,

};

#define o123\_0\_SIZE 398

#define o123\_ZEROS\_0 8192

unsigned char o123\_1[] = {

0x1D,0x00,0x00,0x00,0x14,0x00,0x00,0x00,0x06,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,

0x00,0x00,0x06,0x00,0x27,0x00,0x00,0x00,0x0D,0x00,0x00,0x00,0x14,0x00,0x34,0x00,

0x00,0x00,0x0E,0x00,0x00,0x00,0x06,0x00,0x3A,0x00,0x00,0x00,0x10,0x00,0x00,0x00,

0x06,0x00,0x3F,0x00,0x00,0x00,0x0C,0x00,0x00,0x00,0x14,0x00,0x4A,0x00,0x00,0x00,

0x11,0x00,0x00,0x00,0x06,0x00,0x51,0x00,0x00,0x00,0x12,0x00,0x00,0x00,0x06,0x00,

0x57,0x00,0x00,0x00,0x0F,0x00,0x00,0x00,0x06,0x00,0x5C,0x00,0x00,0x00,0x0B,0x00,

0x00,0x00,0x14,0x00,0x62,0x00,0x00,0x00,0x12,0x00,0x00,0x00,0x06,0x00,0x68,0x00,

0x00,0x00,0x11,0x00,0x00,0x00,0x06,0x00,0x9A,0x00,0x00,0x00,0x0A,0x00,0x00,0x00,

0x14,0x00,0x9F,0x00,0x00,0x00,0x0F,0x00,0x00,0x00,0x06,0x00,0xA6,0x00,0x00,0x00,

0x0A,0x00,0x00,0x00,0x14,0x00,0xAB,0x00,0x00,0x00,0x10,0x00,0x00,0x00,0x06,

};

#define o123\_1\_SIZE 159

#define o123\_ZEROS\_1 8449

unsigned char o123\_2[] = {

0x25,0x64,0x0A,0x0D,

};

#define o123\_2\_SIZE 4

#define o123\_ZEROS\_2 141

unsigned char o123\_3[] = {

0x04,0x00,0x00,0x00,0xF3,0x00,0x00,0x00,0x3C,0x00,0x00,0x00,0x00,0x43,0x3A,0x5C,

0x55,0x73,0x65,0x72,0x73,0x5C,0x4E,0x61,0x7A,0x61,0x72,0x5C,0x73,0x6F,0x75,0x72,

0x63,0x65,0x5C,0x72,0x65,0x70,0x6F,0x73,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,

0x36,0x30,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x5C,0x73,0x6F,0x75,

0x72,0x63,0x65,0x2E,0x61,0x73,0x6D,0x00,0xF4,0x00,0x00,0x00,0x18,0x00,0x00,0x00,

0x01,0x00,0x00,0x00,0x10,0x01,0xD3,0x4E,0x23,0x32,0x9C,0x94,0x3A,0xC3,0x61,0x14,

0xF6,0x0B,0x9E,0xC4,0xCE,0x14,0x00,0x00,0xF2,0x00,0x00,0x00,0xE8,0x01,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xB2,0x20,0x00,0x00,0x00,0x00,0x00,0x00,

0x3A,0x00,0x00,0x00,0xDC,0x01,0x00,0x00,0x05,0x00,0x00,0x00,0x23,0x00,0x00,0x80,

0x06,0x00,0x00,0x00,0x24,0x00,0x00,0x80,0x09,0x00,0x00,0x00,0x25,0x00,0x00,0x80,

0x0B,0x00,0x00,0x00,0x26,0x00,0x00,0x80,0x11,0x00,0x00,0x00,0x27,0x00,0x00,0x80,

0x13,0x00,0x00,0x00,0x28,0x00,0x00,0x80,0x19,0x00,0x00,0x00,0x29,0x00,0x00,0x80,

0x1B,0x00,0x00,0x00,0x2A,0x00,0x00,0x80,0x1B,0x00,0x00,0x00,0x2B,0x00,0x00,0x80,

0x1C,0x00,0x00,0x00,0x2C,0x00,0x00,0x80,0x21,0x00,0x00,0x00,0x2D,0x00,0x00,0x80,

0x26,0x00,0x00,0x00,0x2E,0x00,0x00,0x80,0x2B,0x00,0x00,0x00,0x2F,0x00,0x00,0x80,

0x2E,0x00,0x00,0x00,0x37,0x00,0x00,0x80,0x30,0x00,0x00,0x00,0x38,0x00,0x00,0x80,

0x32,0x00,0x00,0x00,0x39,0x00,0x00,0x80,0x33,0x00,0x00,0x00,0x3A,0x00,0x00,0x80,

0x38,0x00,0x00,0x00,0x3B,0x00,0x00,0x80,0x3E,0x00,0x00,0x00,0x3C,0x00,0x00,0x80,

0x43,0x00,0x00,0x00,0x3E,0x00,0x00,0x80,0x44,0x00,0x00,0x00,0x42,0x00,0x00,0x80,

0x44,0x00,0x00,0x00,0x43,0x00,0x00,0x80,0x45,0x00,0x00,0x00,0x44,0x00,0x00,0x80,

0x47,0x00,0x00,0x00,0x46,0x00,0x00,0x80,0x49,0x00,0x00,0x00,0x47,0x00,0x00,0x80,

0x4E,0x00,0x00,0x00,0x48,0x00,0x00,0x80,0x50,0x00,0x00,0x00,0x49,0x00,0x00,0x80,

0x55,0x00,0x00,0x00,0x4A,0x00,0x00,0x80,0x5B,0x00,0x00,0x00,0x4B,0x00,0x00,0x80,

0x60,0x00,0x00,0x00,0x4D,0x00,0x00,0x80,0x66,0x00,0x00,0x00,0x4E,0x00,0x00,0x80,

0x6C,0x00,0x00,0x00,0x4F,0x00,0x00,0x80,0x6F,0x00,0x00,0x00,0x50,0x00,0x00,0x80,

0x74,0x00,0x00,0x00,0x52,0x00,0x00,0x80,0x76,0x00,0x00,0x00,0x53,0x00,0x00,0x80,

0x77,0x00,0x00,0x00,0x54,0x00,0x00,0x80,0x78,0x00,0x00,0x00,0x57,0x00,0x00,0x80,

0x78,0x00,0x00,0x00,0x5A,0x00,0x00,0x80,0x7A,0x00,0x00,0x00,0x5B,0x00,0x00,0x80,

0x7F,0x00,0x00,0x00,0x5C,0x00,0x00,0x80,0x81,0x00,0x00,0x00,0x5F,0x00,0x00,0x80,

0x84,0x00,0x00,0x00,0x60,0x00,0x00,0x80,0x86,0x00,0x00,0x00,0x61,0x00,0x00,0x80,

0x88,0x00,0x00,0x00,0x62,0x00,0x00,0x80,0x8B,0x00,0x00,0x00,0x63,0x00,0x00,0x80,

0x8E,0x00,0x00,0x00,0x64,0x00,0x00,0x80,0x90,0x00,0x00,0x00,0x65,0x00,0x00,0x80,

0x93,0x00,0x00,0x00,0x66,0x00,0x00,0x80,0x94,0x00,0x00,0x00,0x67,0x00,0x00,0x80,

0x96,0x00,0x00,0x00,0x6A,0x00,0x00,0x80,0x97,0x00,0x00,0x00,0x6E,0x00,0x00,0x80,

0x99,0x00,0x00,0x00,0x6F,0x00,0x00,0x80,0x9E,0x00,0x00,0x00,0x70,0x00,0x00,0x80,

0xA3,0x00,0x00,0x00,0x71,0x00,0x00,0x80,0xA5,0x00,0x00,0x00,0x72,0x00,0x00,0x80,

0xAA,0x00,0x00,0x00,0x73,0x00,0x00,0x80,0xAF,0x00,0x00,0x00,0x87,0x00,0x00,0x80,

0xB1,0x00,0x00,0x00,0x88,0x00,0x00,0x80,0xF1,0x00,0x00,0x00,0xEA,0x02,0x00,0x00,

0x49,0x00,0x01,0x11,0x00,0x00,0x00,0x00,0x43,0x3A,0x5C,0x55,0x73,0x65,0x72,0x73,

0x5C,0x4E,0x61,0x7A,0x61,0x72,0x5C,0x73,0x6F,0x75,0x72,0x63,0x65,0x5C,0x72,0x65,

0x70,0x6F,0x73,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x5C,0x50,0x72,

0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x5C,0x52,0x65,0x6C,0x65,0x61,0x73,0x65,0x5C,

0x73,0x6F,0x75,0x72,0x63,0x65,0x2E,0x6F,0x62,0x6A,0x00,0x37,0x00,0x3C,0x11,0x03,

0x00,0x00,0x00,0x06,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0E,0x00,0x1C,

0x00,0xE4,0x74,0x00,0x00,0x4D,0x69,0x63,0x72,0x6F,0x73,0x6F,0x66,0x74,0x20,0x28,

0x52,0x29,0x20,0x4D,0x61,0x63,0x72,0x6F,0x20,0x41,0x73,0x73,0x65,0x6D,0x62,0x6C,

0x65,0x72,0x00,0x00,0xBB,0x00,0x3D,0x11,0x00,0x63,0x77,0x64,0x00,0x43,0x3A,0x5C,

0x55,0x73,0x65,0x72,0x73,0x5C,0x4E,0x61,0x7A,0x61,0x72,0x5C,0x73,0x6F,0x75,0x72,

0x63,0x65,0x5C,0x72,0x65,0x70,0x6F,0x73,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,

0x36,0x30,0x5C,0x50,0x72,0x6F,0x6A,0x65,0x63,0x74,0x36,0x30,0x00,0x65,0x78,0x65,

0x00,0x43,0x3A,0x5C,0x50,0x72,0x6F,0x67,0x72,0x61,0x6D,0x20,0x46,0x69,0x6C,0x65,

0x73,0x20,0x28,0x78,0x38,0x36,0x29,0x5C,0x4D,0x69,0x63,0x72,0x6F,0x73,0x6F,0x66,

0x74,0x20,0x56,0x69,0x73,0x75,0x61,0x6C,0x20,0x53,0x74,0x75,0x64,0x69,0x6F,0x5C,

0x32,0x30,0x31,0x39,0x5C,0x45,0x6E,0x74,0x65,0x72,0x70,0x72,0x69,0x73,0x65,0x5C,

0x56,0x43,0x5C,0x54,0x6F,0x6F,0x6C,0x73,0x5C,0x4D,0x53,0x56,0x43,0x5C,0x31,0x34,

0x2E,0x32,0x38,0x2E,0x32,0x39,0x39,0x31,0x30,0x5C,0x62,0x69,0x6E,0x5C,0x48,0x6F,

0x73,0x74,0x58,0x38,0x36,0x5C,0x78,0x38,0x36,0x5C,0x6D,0x6C,0x2E,0x65,0x78,0x65,

0x00,0x73,0x72,0x63,0x00,0x73,0x6F,0x75,0x72,0x63,0x65,0x2E,0x61,0x73,0x6D,0x00,

0x00,0x1A,0x00,0x0C,0x11,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x76,

0x61,0x6C,0x75,0x65,0x54,0x65,0x6D,0x70,0x5F,0x6D,0x73,0x67,0x00,0x1A,0x00,0x0C,

0x11,0x22,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x68,0x43,0x6F,0x6E,0x73,

0x6F,0x6C,0x65,0x49,0x6E,0x70,0x75,0x74,0x00,0x1B,0x00,0x0C,0x11,0x22,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x68,0x43,0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x4F,

0x75,0x74,0x70,0x75,0x74,0x00,0x2D,0x00,0x10,0x11,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x34,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x34,0x00,

0x00,0x00,0x06,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x67,0x65,0x74,

0x50,0x72,0x6F,0x63,0x00,0x02,0x00,0x06,0x00,0x2D,0x00,0x10,0x11,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x29,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x29,0x00,0x00,0x00,0x08,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x70,0x75,0x74,0x50,0x72,0x6F,0x63,0x00,0x02,0x00,0x06,0x00,0x15,0x00,0x05,0x11,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x69,0x6E,0x69,0x74,0x43,0x6F,0x6E,0x73,0x6F,

0x6C,0x65,0x00,0x19,0x00,0x0C,0x11,0x22,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x72,0x65,0x61,0x64,0x4F,0x75,0x74,0x43,0x6F,0x75,0x6E,0x74,0x00,0x13,0x00,

0x0C,0x11,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x62,0x75,0x66,0x66,

0x65,0x72,0x00,0x0F,0x00,0x05,0x11,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x73,0x74,

0x61,0x72,0x74,0x00,0x33,0x00,0x10,0x11,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x1F,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1F,0x00,0x00,0x00,

0x10,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x73,0x74,0x72,0x69,0x6E,

0x67,0x5F,0x74,0x6F,0x5F,0x69,0x6E,0x74,0x00,0x16,0x00,0x05,0x11,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x63,0x6F,0x6E,0x76,0x65,0x72,0x74,0x5F,0x6C,0x6F,0x6F,0x70,

0x00,0x0E,0x00,0x05,0x11,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x64,0x6F,0x6E,0x65,

0x00,0x02,0x00,0x06,0x00,0x17,0x00,0x0C,0x11,0x20,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x64,0x61,0x74,0x61,0x5F,0x73,0x74,0x61,0x72,0x74,0x00,0x1A,0x00,

0x0C,0x11,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x76,0x61,0x6C,0x75,

0x65,0x54,0x65,0x6D,0x70,0x5F,0x66,0x6D,0x74,0x00,0x00,0x00,0x00,0x70,0x00,0x00,

0x00,0x18,0x00,0x00,0x00,0x0B,0x00,0x74,0x00,0x00,0x00,0x18,0x00,0x00,0x00,0x0A,

0x00,0xA9,0x03,0x00,0x00,0x0E,0x00,0x00,0x00,0x0B,0x00,0xAD,0x03,0x00,0x00,0x0E,

0x00,0x00,0x00,0x0A,0x00,0xC5,0x03,0x00,0x00,0x0F,0x00,0x00,0x00,0x0B,0x00,0xC9,

0x03,0x00,0x00,0x0F,0x00,0x00,0x00,0x0A,0x00,0xE1,0x03,0x00,0x00,0x10,0x00,0x00,

0x00,0x0B,0x00,0xE5,0x03,0x00,0x00,0x10,0x00,0x00,0x00,0x0A,0x00,0x16,0x04,0x00,

0x00,0x16,0x00,0x00,0x00,0x0B,0x00,0x1A,0x04,0x00,0x00,0x16,0x00,0x00,0x00,0x0A,

0x00,0x49,0x04,0x00,0x00,0x15,0x00,0x00,0x00,0x0B,0x00,0x4D,0x04,0x00,0x00,0x15,

0x00,0x00,0x00,0x0A,0x00,0x60,0x04,0x00,0x00,0x19,0x00,0x00,0x00,0x0B,0x00,0x64,

0x04,0x00,0x00,0x19,0x00,0x00,0x00,0x0A,0x00,0x7B,0x04,0x00,0x00,0x11,0x00,0x00,

0x00,0x0B,0x00,0x7F,0x04,0x00,0x00,0x11,0x00,0x00,0x00,0x0A,0x00,0x96,0x04,0x00,

0x00,0x12,0x00,0x00,0x00,0x0B,0x00,0x9A,0x04,0x00,0x00,0x12,0x00,0x00,0x00,0x0A,

0x00,0xA7,0x04,0x00,0x00,0x13,0x00,0x00,0x00,0x0B,0x00,0xAB,0x04,0x00,0x00,0x13,

0x00,0x00,0x00,0x0A,0x00,0xD4,0x04,0x00,0x00,0x17,0x00,0x00,0x00,0x0B,0x00,0xD8,

0x04,0x00,0x00,0x17,0x00,0x00,0x00,0x0A,0x00,0xED,0x04,0x00,0x00,0x1A,0x00,0x00,

0x00,0x0B,0x00,0xF1,0x04,0x00,0x00,0x1A,0x00,0x00,0x00,0x0A,0x00,0x05,0x05,0x00,

0x00,0x1B,0x00,0x00,0x00,0x0B,0x00,0x09,0x05,0x00,0x00,0x1B,0x00,0x00,0x00,0x0A,

0x00,0x1D,0x05,0x00,0x00,0x1C,0x00,0x00,0x00,0x0B,0x00,0x21,0x05,0x00,0x00,0x1C,

0x00,0x00,0x00,0x0A,0x00,0x36,0x05,0x00,0x00,0x14,0x00,0x00,0x00,0x0B,0x00,0x3A,

0x05,0x00,0x00,0x14,0x00,0x00,0x00,0x0A,0x00,0x04,0x00,0x00,0x00,0x0A,0x00,0x01,

0x12,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,

0x00,0x00,0x00,0x01,0x00,0x00,0x10,0x00,0x00,0x06,0x00,0x0E,0x00,0x00,0x00,0xF2,

0xF1,0x1A,0x00,0x01,0x12,0x05,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,

0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,

0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x05,0x00,0x03,0x10,0x00,0x00,0x06,0x00,0x01,

0x12,0x00,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x00,

0x00,0x05,0x10,0x00,0x00,0x06,0x00,0x01,0x12,0x00,0x00,0x00,0x00,0x0E,0x00,0x08,

0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x00,0x00,0x07,0x10,0x00,0x00,0x0A,0x00,0x01,

0x12,0x01,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,

0x00,0x07,0x00,0x01,0x00,0x09,0x10,0x00,0x00,0x0E,0x00,0x01,0x12,0x02,0x00,0x00,

0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,

0x00,0x07,0x00,0x02,0x00,0x0B,0x10,0x00,0x00,0x0E,0x00,0x01,0x12,0x02,0x00,0x00,

0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,

0x00,0x07,0x00,0x02,0x00,0x0D,0x10,0x00,0x00,0x06,0x00,0x01,0x12,0x00,0x00,0x00,

0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x00,0x00,0x0F,0x10,0x00,

0x00,0x1A,0x00,0x01,0x12,0x05,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,

0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,

0x10,0x03,0x00,0x00,0x00,0x07,0x00,0x05,0x00,0x11,0x10,0x00,0x00,0x0A,0x00,0x01,

0x12,0x01,0x00,0x00,0x00,0x22,0x00,0x00,0x00,0x0E,0x00,0x08,0x10,0x03,0x00,0x00,

0x00,0x07,0x00,0x01,0x00,0x13,0x10,0x00,0x00,0x2F,0x45,0x4E,0x54,0x52,0x59,0x3A,

0x73,0x74,0x61,0x72,0x74,0x20,0x00,0x40,0x63,0x6F,0x6D,0x70,0x2E,0x69,0x64,0xE4,

0x74,0x03,0x01,0xFF,0xFF,0x00,0x00,0x03,0x00,0x40,0x66,0x65,0x61,0x74,0x2E,0x30,

0x30,0x10,0x00,0x00,0x00,0xFF,0xFF,0x00,0x00,0x03,0x00,0x2E,0x74,0x65,0x78,0x74,

0x24,0x6D,0x6E,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x03,0x01,0xB2,0x20,0x00,

0x00,0x10,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x2E,

0x64,0x61,0x74,0x61,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x00,0x00,0x00,0x03,

0x01,0x91,0x21,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x2E,0x64,0x65,0x62,0x75,0x67,0x24,0x53,0x00,0x00,0x00,0x00,0x03,

0x00,0x00,0x00,0x03,0x01,0x4C,0x05,0x00,0x00,0x1E,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x2E,0x64,0x65,0x62,0x75,0x67,0x24,0x54,0x00,

0x00,0x00,0x00,0x04,0x00,0x00,0x00,0x03,0x01,0x40,0x01,

};

#define o123\_3\_SIZE 2155

#define o123\_ZEROS\_3 20

unsigned char o123\_4[] = {

0x04,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x00,0x02,0x00,0x00,0x00,

0x00,0x00,0x14,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x00,0x02,0x00,

0x00,0x00,0x00,0x00,0x25,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x00,

0x02,0x00,0x00,0x00,0x00,0x00,0x37,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x20,0x00,0x02,0x00,0x00,0x00,0x00,0x00,0x42,0x00,0x00,0x00,0x00,0x20,0x00,0x00,

0x02,0x00,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0x00,0x50,0x00,0x00,0x00,0x05,0x21,

0x00,0x00,0x02,0x00,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0x00,0x5E,0x00,0x00,0x00,

0x09,0x21,0x00,0x00,0x02,0x00,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0x00,0x6D,0x00,

0x00,0x00,0x8D,0x21,0x00,0x00,0x02,0x00,0x00,0x00,0x03,0x00,0x62,0x75,0x66,0x66,

0x65,0x72,0x00,0x00,0x0D,0x21,0x00,0x00,0x02,0x00,0x00,0x00,0x03,0x00,0x5F,0x73,

0x74,0x61,0x72,0x74,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x02,0x00,

0x00,0x00,0x00,0x00,0x7A,0x00,0x00,0x00,0x00,0x21,0x00,0x00,0x02,0x00,0x00,0x00,

0x03,0x00,0x00,0x00,0x00,0x00,0x88,0x00,0x00,0x00,0x1B,0x00,0x00,0x00,0x01,0x00,

0x20,0x00,0x02,0x00,0x00,0x00,0x00,0x00,0x93,0x00,0x00,0x00,0x44,0x00,0x00,0x00,

0x01,0x00,0x20,0x00,0x02,0x00,0x00,0x00,0x00,0x00,0x9E,0x00,0x00,0x00,0x78,0x00,

0x00,0x00,0x01,0x00,0x20,0x00,0x02,0x00,0x24,0x24,0x30,0x30,0x30,0x30,0x30,0x30,

0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0x00,0xAF,0x00,

0x00,0x00,0x97,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x06,0x00,0x00,0x00,0x00,0x00,

0xBB,0x00,0x00,0x00,0x81,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x06,0x00,0x64,0x6F,

0x6E,0x65,0x00,0x00,0x00,0x00,0x96,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x06,0x00,

0x00,0x00,0x00,0x00,0xC8,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x00,0x00,0x00,

0x03,0x00,0x2E,0x64,0x72,0x65,0x63,0x74,0x76,0x65,0x00,0x00,0x00,0x00,0x05,0x00,

0x00,0x00,0x03,0x01,0x0D,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0xD3,0x00,0x00,0x00,0x5F,0x47,0x65,0x74,0x53,0x74,

0x64,0x48,0x61,0x6E,0x64,0x6C,0x65,0x40,0x34,0x00,0x5F,0x52,0x65,0x61,0x64,0x43,

0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x41,0x40,0x32,0x30,0x00,0x5F,0x57,0x72,0x69,0x74,

0x65,0x43,0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x41,0x40,0x32,0x30,0x00,0x5F,0x77,0x73,

0x70,0x72,0x69,0x6E,0x74,0x66,0x41,0x00,0x76,0x61,0x6C,0x75,0x65,0x54,0x65,0x6D,

0x70,0x5F,0x6D,0x73,0x67,0x00,0x68,0x43,0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x49,0x6E,

0x70,0x75,0x74,0x00,0x68,0x43,0x6F,0x6E,0x73,0x6F,0x6C,0x65,0x4F,0x75,0x74,0x70,

0x75,0x74,0x00,0x72,0x65,0x61,0x64,0x4F,0x75,0x74,0x43,0x6F,0x75,0x6E,0x74,0x00,

0x76,0x61,0x6C,0x75,0x65,0x54,0x65,0x6D,0x70,0x5F,0x66,0x6D,0x74,0x00,0x5F,0x70,

0x75,0x74,0x50,0x72,0x6F,0x63,0x40,0x30,0x00,0x5F,0x67,0x65,0x74,0x50,0x72,0x6F,

0x63,0x40,0x30,0x00,0x5F,0x73,0x74,0x72,0x69,0x6E,0x67,0x5F,0x74,0x6F,0x5F,0x69,

0x6E,0x74,0x40,0x30,0x00,0x69,0x6E,0x69,0x74,0x43,0x6F,0x6E,0x73,0x6F,0x6C,0x65,

0x00,0x63,0x6F,0x6E,0x76,0x65,0x72,0x74,0x5F,0x6C,0x6F,0x6F,0x70,0x00,0x64,0x61,

0x74,0x61,0x5F,0x73,0x74,0x61,0x72,0x74,0x00,

};

#define o123\_4\_SIZE 585

unsigned long long int o123\_array\_part\_count = 5;

unsigned char\* o123[5] = {

o123\_0

, o123\_1

, o123\_2

, o123\_3

, o123\_4

};

unsigned long long int o123\_array\_part\_size[5] = {

o123\_0\_SIZE

, o123\_1\_SIZE

, o123\_2\_SIZE

, o123\_3\_SIZE

, o123\_4\_SIZE

};

unsigned long long int o123\_zero\_part\_count = 4;

unsigned long long int o123\_zeros[4] = {

o123\_ZEROS\_0

, o123\_ZEROS\_1

, o123\_ZEROS\_2

, o123\_ZEROS\_3

};  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: identifier\_or\_value.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

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

if ((\*lastLexemInfoInTable)->tokenType == VALUE\_LEXEME\_TYPE) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%lld\"\r\n", (\*lastLexemInfoInTable)->ifvalue);

//

currBytePtr += sprintf((char\*)currBytePtr, " add ecx, 4\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " mov eax, 0%08Xh\r\n", (int)(\*lastLexemInfoInTable)->ifvalue);

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr [ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%lld\"\r\n", (\*lastLexemInfoInTable)->ifvalue);

//

currBytePtr += sprintf((char\*)currBytePtr, " opStack[++opStackIndex] = opTemp = 0x%08X;\r\n", (int)(\*lastLexemInfoInTable)->ifvalue);

}

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);

}

variableIndex \*= VALUE\_SIZE;

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

unsigned char code\_\_mov\_eax\_edi[] = { 0x8B, 0xC7 };

unsigned char code\_\_add\_eax\_variableOffsetInDataSection[] = { 0x05, 0x00, 0x00, 0x00, 0x00 };

const unsigned char code\_\_mov\_eax\_valueByAdrressInEAX[] = { 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\_valueByAdrressInEAX, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_add\_ecx\_4, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_toAddrFromECX\_eax, 2);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", (\*lastLexemInfoInTable)->lexemStr);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, edi\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " add eax, 0%08Xh\r\n", (int)variableIndex);

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[eax]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr [ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", (\*lastLexemInfoInTable)->lexemStr);

//

currBytePtr += sprintf((char\*)currBytePtr, " opStack[++opStackIndex] = opTemp = data[0x%08X];\r\n", (int)variableIndex);

}

return ++ \* lastLexemInfoInTable, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: or.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_OR);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_mov\_eax\_stackTopByECX[] = { 0x8B, 0x01 };

const unsigned char code\_\_cmp\_eax\_0[] = { 0x83, 0xF8, 0x00 };

const unsigned char code\_\_setne\_al[] = { 0x0F, 0x95, 0xC0 };

const unsigned char code\_\_and\_eax\_1[] = { 0x83, 0xE0, 0x01 };

const unsigned char code\_\_sub\_ecx\_4[] = { 0x83, 0xE9, 0x04 };

//

const unsigned char code\_\_cmp\_stackTopByECX\_0[] = { 0x83, 0x39, 0x00 };

const unsigned char code\_\_setne\_dl[] = { 0x0F, 0x95, 0xC2 };

const unsigned char code\_\_and\_edx\_1[] = { 0x83, 0xE2, 0x01 };

//

const unsigned char code\_\_or\_eax\_edx[] = { 0x0B, 0xC2 };

//

const unsigned char code\_\_mov\_stackTopByECX\_eax[] = { 0x89, 0x01 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_eax\_stackTopByECX, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_cmp\_eax\_0, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_setne\_al, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_and\_eax\_1, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_sub\_ecx\_4, 3);

//

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_cmp\_stackTopByECX\_0, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_setne\_dl, 3);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_and\_edx\_1, 3);

//

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_or\_eax\_edx, 2);

//

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_stackTopByECX\_eax, 2);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_OR][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " cmp eax, 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " setne al\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and eax, 1\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

//

currBytePtr += sprintf((char\*)currBytePtr, " cmp dword ptr[ecx], 0\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " setne dl\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " and edx, 1\r\n");

//

currBytePtr += sprintf((char\*)currBytePtr, " or eax, edx\r\n");

//

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr[ecx], eax\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_OR][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] = opStack[opStackIndex - 1] || opStack[opStackIndex--];\r\n");

// currBytePtr += sprintf((char\*)currBytePtr, " //--opStackIndex;\r\n");

// currBytePtr += sprintf((char\*)currBytePtr, " //opTemp = opStack[opStackIndex];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: output.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_OUTPUT);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_mov\_eax\_stackTopByECX[] = { 0x8B, 0x01 };

const unsigned char code\_\_mov\_edx\_address[] = { 0xBA, 0x00, 0x00, 0x00, 0x00 };

const unsigned char code\_\_add\_edx\_esi[] = { 0x03, 0xD6 };

//const unsigned char code\_\_push\_ecx[] = { 0x51 };

//const unsigned char code\_\_push\_ebx[] = { 0x53 };

const unsigned char code\_\_push\_esi[] = { 0x56 };

const unsigned char code\_\_push\_edi[] = { 0x57 };

const unsigned char code\_\_call\_edx[] = { 0xFF, 0xD2 };

const unsigned char code\_\_pop\_edi[] = { 0x5F };

const unsigned char code\_\_pop\_esi[] = { 0x5E };

//const unsigned char code\_\_pop\_ebx[] = { 0x5B };

//const unsigned char code\_\_pop\_ecx[] = { 0x59 };

const unsigned char code\_\_mov\_ecx\_edi[] = { 0x8B, 0xCF };

const unsigned char code\_\_add\_ecx\_512[] = { 0x81, 0xC1, 0x00, 0x02, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_eax\_stackTopByECX, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_edx\_address, 5);

\*(unsigned int\*)&(currBytePtr[-4]) = (unsigned int)putProcOffset;

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_add\_edx\_esi, 2);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_push\_ecx, 1);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_push\_ebx, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_push\_esi, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_push\_edi, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_call\_edx, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_pop\_edi, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_pop\_esi, 1);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_pop\_ebx, 1);

//currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_pop\_ecx, 1);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_mov\_ecx\_edi, 2);

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_add\_ecx\_512, 6);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 9192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_OUTPUT][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " mov edx, 0%08Xh\r\n", (unsigned int)putProcOffset);

currBytePtr += sprintf((char\*)currBytePtr, " add edx, esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push ecx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;push ebx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " push edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " call edx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " pop edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " pop esi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;pop ebx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " ;pop ecx\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ecx, edi ; reset second stack\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add ecx, 512 ; reset second stack\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_OUTPUT][0]);

currBytePtr += sprintf((char\*)currBytePtr, " (void)printf(\"%%d\\r\\n\", opTemp = opStack[opStackIndex]), opStackIndex = 0;\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: preparer.hxx \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

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

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

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

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

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

int precedenceLevel(char\* lexemStr) {

if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_BITWISE\_NOT][0], MAX\_LEXEM\_SIZE)) {

return 6;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_NOT][0], MAX\_LEXEM\_SIZE)) {

return 6;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_BITWISE\_AND][0], MAX\_LEXEM\_SIZE)) {

return 5;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_AND][0], MAX\_LEXEM\_SIZE)) {

return 5;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_MUL][0], MAX\_LEXEM\_SIZE)) {

return 5;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_DIV][0], MAX\_LEXEM\_SIZE)) {

return 5;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_MOD][0], MAX\_LEXEM\_SIZE)) {

return 5;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_BITWISE\_OR][0], MAX\_LEXEM\_SIZE)) {

return 4;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_OR][0], MAX\_LEXEM\_SIZE)) {

return 4;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_ADD][0], MAX\_LEXEM\_SIZE)) {

return 4;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_SUB][0], MAX\_LEXEM\_SIZE)) {

return 4;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_EQUAL][0], MAX\_LEXEM\_SIZE)) {

return 3;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_NOT\_EQUAL][0], MAX\_LEXEM\_SIZE)) {

return 3;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_LESS][0], MAX\_LEXEM\_SIZE)) {

return 3;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_GREATER][0], MAX\_LEXEM\_SIZE)) {

return 3;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_LESS\_OR\_EQUAL][0], MAX\_LEXEM\_SIZE)) {

return 3;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_GREATER\_OR\_EQUAL][0], MAX\_LEXEM\_SIZE)) {

return 3;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_RLBIND][0], MAX\_LEXEM\_SIZE)) {

return 2;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_LRBIND][0], MAX\_LEXEM\_SIZE)) {

return 2;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_INPUT][0], MAX\_LEXEM\_SIZE)) {

return 1;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_OUTPUT][0], MAX\_LEXEM\_SIZE)) {

return 1;

}

return 0;

}

bool isLeftAssociative(char\* lexemStr) {

if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_BITWISE\_AND][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_AND][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_MUL][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_DIV][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_MOD][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_BITWISE\_OR][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_OR][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_ADD][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_SUB][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_EQUAL][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_NOT\_EQUAL][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_LESS][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_GREATER][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_LESS\_OR\_EQUAL][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_GREATER\_OR\_EQUAL][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_LRBIND][0], MAX\_LEXEM\_SIZE)) { // ! TODO: ...

return false;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_RLBIND][0], MAX\_LEXEM\_SIZE)) {

return false;

}

if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_BITWISE\_NOT][0], MAX\_LEXEM\_SIZE)) {

return false;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_NOT][0], MAX\_LEXEM\_SIZE)) {

return false;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_INPUT][0], MAX\_LEXEM\_SIZE)) {

return false;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_OUTPUT][0], MAX\_LEXEM\_SIZE)) {

return false;

}

return false;

}

bool isSplittingOperator(char\* lexemStr) {

if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_INPUT][0], MAX\_LEXEM\_SIZE)) {

return true;

}

else if (!strncmp(lexemStr, tokenStruct[MULTI\_TOKEN\_OUTPUT][0], MAX\_LEXEM\_SIZE)) {

return true;

}

return false;

}

void makePrepare4IdentifierOrValue(struct LexemInfo\*\* lastLexemInfoInTable, struct LexemInfo\*\* lastTempLexemInfoInTable) { //

if ((\*lastLexemInfoInTable)->tokenType == IDENTIFIER\_LEXEME\_TYPE || (\*lastLexemInfoInTable)->tokenType == VALUE\_LEXEME\_TYPE) {

int prevNonOpenParenthesesIndex = -1;

for (; !strncmp((\*lastLexemInfoInTable)[prevNonOpenParenthesesIndex].lexemStr, "(", MAX\_LEXEM\_SIZE); --prevNonOpenParenthesesIndex);

if (!strncmp((\*lastLexemInfoInTable)[1].lexemStr, tokenStruct[MULTI\_TOKEN\_RLBIND][0], MAX\_LEXEM\_SIZE)

||

!strncmp((\*lastLexemInfoInTable)[-1].lexemStr, tokenStruct[MULTI\_TOKEN\_LRBIND][0], MAX\_LEXEM\_SIZE)

||

//!strncmp((\*lastLexemInfoInTable)[-1].lexemStr, tokenStruct[MULTI\_TOKEN\_INPUT][0], MAX\_LEXEM\_SIZE)

//||

//!strncmp((\*lastLexemInfoInTable)[-2].lexemStr, tokenStruct[MULTI\_TOKEN\_INPUT][0], MAX\_LEXEM\_SIZE)

//||

!strncmp((\*lastLexemInfoInTable)[prevNonOpenParenthesesIndex].lexemStr, tokenStruct[MULTI\_TOKEN\_INPUT][0], MAX\_LEXEM\_SIZE)

) {

bool findComplete = false;

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

if (!strncmp((\*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;

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

if (passMakePrepareElementCount++ == ~0) {

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

exit(0);

}

// \*lastLexemInfoInTable += lastDataSectionLexemIndex;

// while (lastDataSectionLexemIndex--) {

//

// }

//

// for (; false && (\*lastLexemInfoInTable)->lexemStr[0] != '\0'; \*(\*lastTempLexemInfoInTable)++ = \*(\*lastLexemInfoInTable)++) {

// if (passMakePrepareElementCount) {

// --passMakePrepareElementCount;

// ++lexemInfoInTable;

// continue;

// }

// else {

// break;

// }

// }

lexemInfoTransformationTempStackSize = 0;

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

if (passMakePrepareElementCount) {

--passMakePrepareElementCount;

++lexemInfoInTable;

continue;

}

for (struct LexemInfo\* lastLexemInfoInTable\_ = NULL; lastLexemInfoInTable\_ != \*lastLexemInfoInTable;) {

lastLexemInfoInTable\_ = \*lastLexemInfoInTable;

makePrepare4IdentifierOrValue(lastLexemInfoInTable, lastTempLexemInfoInTable);

if (lastLexemInfoInTable\_ == \*lastLexemInfoInTable)

makePrepare4Operators(lastLexemInfoInTable, lastTempLexemInfoInTable);

if (lastLexemInfoInTable\_ == \*lastLexemInfoInTable)

makePrepare4LeftParenthesis(lastLexemInfoInTable, lastTempLexemInfoInTable);

if (lastLexemInfoInTable\_ == \*lastLexemInfoInTable)

makePrepare4RightParenthesis(lastLexemInfoInTable, lastTempLexemInfoInTable);

if (lastLexemInfoInTable\_ != \*lastLexemInfoInTable

&& (!nullStatementIndex || (lexemInfoInTable + nullStatementIndex == lastLexemInfoInTable\_))) {

if (nullStatementIndex != ~0) {

if (nullStatementIndex) {

// printf("Added null statement after %lld(lexem index)\r\n", nullStatementIndex);

makePrepareEnder(lastLexemInfoInTable, lastTempLexemInfoInTable);

(void)createMultiToken(lastTempLexemInfoInTable, MULTI\_TOKEN\_NULL\_STATEMENT);

}

nullStatementIndex = getNextEndOfExpressionIndex(lexemInfoInTable, nullStatementIndex);

}

}

}

makePrepareEnder(lastLexemInfoInTable, lastTempLexemInfoInTable);

if ((!nullStatementIndex || (lexemInfoInTable + nullStatementIndex == \*lastLexemInfoInTable))) {

if (nullStatementIndex != ~0) {

if (nullStatementIndex) {

// printf("Added null statement after %lld(lexem index)\r\n", nullStatementIndex);

makePrepareEnder(lastLexemInfoInTable, lastTempLexemInfoInTable);

(void)createMultiToken(lastTempLexemInfoInTable, MULTI\_TOKEN\_NULL\_STATEMENT);

}

nullStatementIndex = getNextEndOfExpressionIndex(lexemInfoInTable, nullStatementIndex);

}

}

}

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: repeat\_until.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

#include "string.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_REPEAT);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_REPEAT][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_REPEAT][0]);

}

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = \*\*lastLexemInfoInTable;

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL@REPEAT\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL\_\_REPEAT\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr);

}

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

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_REPEAT][0], MAX\_LEXEM\_SIZE)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;%s\r\n", tokenStruct[MULTI\_TOKEN\_UNTIL][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //%s\r\n", tokenStruct[MULTI\_TOKEN\_UNTIL][0]);

}

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

|| strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_UNTIL][0], MAX\_LEXEM\_SIZE)

|| strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI\_TOKEN\_REPEAT][0], MAX\_LEXEM\_SIZE)

) {

return currBytePtr;

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;after cond expresion (after \"%s\" after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_UNTIL][0], tokenStruct[MULTI\_TOKEN\_REPEAT][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " cmp eax, 0\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jnz LABEL@REPEAT\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //after cond expresion (after \"%s\" after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_UNTIL][0], tokenStruct[MULTI\_TOKEN\_REPEAT][0]);

//

currBytePtr += snprintf((char\*)currBytePtr, 8192, " if(opTemp != 0) goto LABEL\_\_REPEAT\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

}

lexemInfoTransformationTempStackSize -= 2;

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_RLBIND][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ebx, dword ptr[ecx - 4]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 8\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add ebx, edi\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov dword ptr [ebx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov ecx, edi ; reset second stack\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " add ecx, 512 ; reset second stack\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_RLBIND][0]);

currBytePtr += sprintf((char\*)currBytePtr, " lastBindDataIndex = opStack[opStackIndex - 1];\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " data[lastBindDataIndex] = opTemp = opStack[opStackIndex], opStackIndex = 0;\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: semantix.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

#include "string.h"

//#include <iterator>

#include <regex>

//

//#define COLLISION\_II\_STATE 128

//#define COLLISION\_LL\_STATE 129

//#define COLLISION\_IL\_STATE 130

//#define COLLISION\_I\_STATE 132

//#define COLLISION\_L\_STATE 133

//

//#define NO\_IMPLEMENT\_CODE\_STATE 256

unsigned long long int getDataSectionLastLexemIndex(LexemInfo\* lexemInfoTable, Grammar\* grammar) {

int lexemIndex = 0;

const struct LexemInfo\* unexpectedLexemfailedTerminal = nullptr;

if (recursiveDescentParserRuleWithDebug("program\_\_\_\_part1", lexemIndex, lexemInfoTable, grammar, 0, &unexpectedLexemfailedTerminal)

&& lexemInfoTable[lexemIndex].lexemStr[0] != '\0') {

return lexemIndex;

}

printf("Error: No find data section end index!\r\n");

return ~0;

}

int checkingInternalCollisionInDeclarations(LexemInfo\* lexemInfoTable, Grammar\* grammar, char(\*identifierIdsTable)[MAX\_LEXEM\_SIZE], char \*\* errorMessagesPtrToLastBytePtr) {

// int returnState = SUCCESS\_STATE;

unsigned long long int lastDataSectionLexemIndex = 0;

if (~0 == (lastDataSectionLexemIndex = getDataSectionLastLexemIndex(lexemInfoTable, grammar))) { // TODO: ADD TO START CODE

\*errorMessagesPtrToLastBytePtr += sprintf(\*errorMessagesPtrToLastBytePtr, "Error get of data section last lexem index.\r\n");

return ~SUCCESS\_STATE;

}

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

char isDeclaredIdentifier = 0;

char isDeclaredIdentifierCollision = 0;

unsigned int lexemIndex = 0;

for (lexemIndex = 0; lexemIndex <= lastDataSectionLexemIndex; ++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 (!strncmp(lexemesInfoTable[lexemIndex + 1].lexemStr, tokenStruct[MULTI\_TOKEN\_COLON][0], MAX\_LEXEM\_SIZE)) {

if (isDeclaredLabel) {

isDeclaredLabelCollision = 1;

}

isLabel = 1;

isDeclaredLabel = 1;

}

if (lexemIndex && !strncmp(lexemesInfoTable[lexemIndex - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_GOTO][0], MAX\_LEXEM\_SIZE)) {

isLabel = 1;

}

}

// //tryToGetKeyWord(struct LexemInfo\* lexemInfoInTable);

// if (SUCCESS\_STATE != checkingCollisionInDeclarationsByKeyWords(identifierIdsTable[index])) {

// return COLLISION\_IK\_STATE;

// }

if (isDeclaredIdentifierCollision) {

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

\*errorMessagesPtrToLastBytePtr += snprintf(\*errorMessagesPtrToLastBytePtr, MAX\_LEXEM\_SIZE + strlen("Collision(identifier/identifier): #\r\n"), "Collision(identifier/identifier): %s\r\n", identifierIdsTable[index]);

return COLLISION\_II\_STATE;

}

if (isDeclaredLabelCollision) {

printf("Collision(label/label): %s\r\n", identifierIdsTable[index]);

\*errorMessagesPtrToLastBytePtr += snprintf(\*errorMessagesPtrToLastBytePtr, MAX\_LEXEM\_SIZE + strlen("Collision(label/label): #\r\n"), "Collision(label/label): %s\r\n", identifierIdsTable[index]);

return COLLISION\_LL\_STATE;

}

if (isDeclaredIdentifier && isLabel) {

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

\*errorMessagesPtrToLastBytePtr += snprintf(\*errorMessagesPtrToLastBytePtr, MAX\_LEXEM\_SIZE + strlen("Collision(identifier/label): #\r\n"), "Collision(identifier/label): %s\r\n", identifierIdsTable[index]);

return COLLISION\_IL\_STATE;

}

else if (!isDeclaredIdentifier && !isLabel && !isDeclaredLabel) {

printf("Undeclared identifier: %s\r\n", identifierIdsTable[index]);

\*errorMessagesPtrToLastBytePtr += snprintf(\*errorMessagesPtrToLastBytePtr, MAX\_LEXEM\_SIZE + strlen("Undeclared identifier: #\r\n"), "Undeclared identifier: %s\r\n", identifierIdsTable[index]);

return COLLISION\_I\_STATE;

}

else if (isLabel && !isDeclaredLabel) {

printf("Undeclared label: %s\r\n", identifierIdsTable[index]);

\*errorMessagesPtrToLastBytePtr += snprintf(\*errorMessagesPtrToLastBytePtr, MAX\_LEXEM\_SIZE + strlen("Undeclared label: #\r\n"), "Undeclared label: %s\r\n", identifierIdsTable[index]);

return COLLISION\_L\_STATE;

}

}

// if (returnState == SUCCESS\_STATE) {

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

// }

//

return SUCCESS\_STATE;

}

int checkingVariableInitialization(LexemInfo\* lexemInfoTable, Grammar\* grammar, char(\*identifierIdsTable)[MAX\_LEXEM\_SIZE], char\*\* errorMessagesPtrToLastBytePtr) {

int returnState = SUCCESS\_STATE;

unsigned long long int lastDataSectionLexemIndex = 0;

if (~0 == (lastDataSectionLexemIndex = getDataSectionLastLexemIndex(lexemInfoTable, grammar))) { // TODO: ADD TO START CODE

\*errorMessagesPtrToLastBytePtr += sprintf(\*errorMessagesPtrToLastBytePtr, "Error get of data section last lexem index.\r\n");

return ~SUCCESS\_STATE;

}

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

for (unsigned int lexemIndex = lastDataSectionLexemIndex; lexemesInfoTable[lexemIndex].lexemStr[0] != '\0'; ++lexemIndex) {

if (lexemesInfoTable[lexemIndex].tokenType != IDENTIFIER\_LEXEME\_TYPE || strncmp(identifierIdsTable[index], lexemesInfoTable[lexemIndex].lexemStr, MAX\_LEXEM\_SIZE)) {

continue;

}

if (!strncmp(lexemesInfoTable[lexemIndex + 1].lexemStr, tokenStruct[MULTI\_TOKEN\_COLON][0], MAX\_LEXEM\_SIZE)) {

continue;

}

if (lexemIndex && !strncmp(lexemesInfoTable[lexemIndex - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_GOTO][0], MAX\_LEXEM\_SIZE)) {

continue;

}

int prevNonOpenParenthesesIndex = -1;

for (; !strncmp(lexemesInfoTable[lexemIndex + prevNonOpenParenthesesIndex].lexemStr, "(", MAX\_LEXEM\_SIZE); --prevNonOpenParenthesesIndex);

if (!strncmp(lexemesInfoTable[lexemIndex + 1].lexemStr, tokenStruct[MULTI\_TOKEN\_RLBIND][0], MAX\_LEXEM\_SIZE)

||

!strncmp(lexemesInfoTable[lexemIndex - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_LRBIND][0], MAX\_LEXEM\_SIZE)

||

//!strncmp(lexemesInfoTable[-1].lexemStr, tokenStruct[MULTI\_TOKEN\_INPUT][0], MAX\_LEXEM\_SIZE)

//||

//!strncmp(lexemesInfoTable[-2].lexemStr, tokenStruct[MULTI\_TOKEN\_INPUT][0], MAX\_LEXEM\_SIZE)

//||

!strncmp(lexemesInfoTable[lexemIndex + prevNonOpenParenthesesIndex].lexemStr, tokenStruct[MULTI\_TOKEN\_INPUT][0], MAX\_LEXEM\_SIZE)

){

break;

}

printf("Uninitialized: %s\r\n", identifierIdsTable[index]);

\*errorMessagesPtrToLastBytePtr += snprintf(\*errorMessagesPtrToLastBytePtr, MAX\_LEXEM\_SIZE + strlen("Uninitialized: #\r\n"), "Uninitialized: %s\r\n", identifierIdsTable[index]);

returnState = UNINITIALIZED\_I\_STATE;

break;

}

}

if (returnState == SUCCESS\_STATE) {

printf("Variable initialization checking was successful!\r\n");

}

return returnState;

}

int checkingCollisionInDeclarationsByKeyWords(char(\*identifierIdsTable)[MAX\_LEXEM\_SIZE], char\*\* errorMessagesPtrToLastBytePtr) {

int returnState = SUCCESS\_STATE;

char keywords\_re[] = KEYWORDS\_RE;

char keywords\_[sizeof(keywords\_re)] = { '\0' };

prepareKeyWordIdGetter(keywords\_, keywords\_re);

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

if (std::regex\_match(std::string(identifierIdsTable[index]), std::regex(keywords\_re))) {

printf("Declaration matches keyword: %s\r\n", identifierIdsTable[index]);

\*errorMessagesPtrToLastBytePtr += snprintf(\*errorMessagesPtrToLastBytePtr, MAX\_LEXEM\_SIZE + strlen("Declaration matches keyword: #\r\n"), "Declaration matches keyword: %s\r\n", identifierIdsTable[index]);

returnState = COLLISION\_IK\_STATE;

}

}

printf("Declaration verification for keyword collision was successful!\r\n");

return SUCCESS\_STATE;

}

int semantixAnalyze(LexemInfo\* lexemInfoTable, Grammar\* grammar, char(\*identifierIdsTable)[MAX\_LEXEM\_SIZE], char\* errorMessagesPtrToLastBytePtr){

int returnState = SUCCESS\_STATE;

if ( SUCCESS\_STATE != (returnState = checkingInternalCollisionInDeclarations(lexemesInfoTable, grammar, identifierIdsTable, &errorMessagesPtrToLastBytePtr))

|| SUCCESS\_STATE != (returnState = checkingVariableInitialization(lexemesInfoTable, grammar, identifierIdsTable, &errorMessagesPtrToLastBytePtr))

|| SUCCESS\_STATE != (returnState = checkingCollisionInDeclarationsByKeyWords(identifierIdsTable, &errorMessagesPtrToLastBytePtr))

) {

return returnState;

}

return SUCCESS\_STATE;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: semicolon.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_SEMICOLON);

if (multitokenSize

&&

!lexemInfoTransformationTempStackSize // !

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0]);

}

\* 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) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_SEMICOLON][0]);

}

\* lastLexemInfoInTable += multitokenSize;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: sub.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_SUB);

if (multitokenSize) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_SUB][0]);

//

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub ecx, 4\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " sub dword ptr[ecx], eax\r\n");

currBytePtr += sprintf((char\*)currBytePtr, " mov eax, dword ptr[ecx]\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_SUB][0]);

currBytePtr += sprintf((char\*)currBytePtr, " opTemp = opStack[opStackIndex - 1] -= opStack[opStackIndex--];\r\n");

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: syntax.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include <iostream>

#include <fstream>

#include <iomanip>

#include <vector>

#include <map>

//#include <unordered\_map>

#include <string>

#include <set>

using namespace std;

Grammar grammar = {

CONFIGURABLE\_GRAMMAR

#if 0

{

{"labeled\_point", 2, {"ident", "tokenCOLON"}}, // !!!!!

{"goto\_label", 2, {"tokenGOTO","ident"}}, // !!!!!

{"program\_name", 1, {"ident\_terminal"}},

{"value\_type", 1, {T\_DATA\_TYPE\_0}},

{"other\_declaration\_ident", 2, {"tokenCOMMA", "ident"}},

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"other\_declaration\_ident","other\_declaration\_ident\_\_\_\_iteration\_after\_one", }},

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"tokenCOMMA", "ident"}},

{"value\_type\_\_ident", 2, {"value\_type", "ident"}},

{"declaration", 2, {"value\_type\_\_ident", "other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},

{"declaration", 2, {"value\_type", "ident"}},

//

{"unary\_operator", 1, {T\_NOT\_0}},

{"unary\_operator", 1, {T\_SUB\_0}},

{"unary\_operator", 1, {T\_ADD\_0}},

{"binary\_operator", 1, {T\_AND\_0}},

{"binary\_operator", 1, {T\_OR\_0}},

{"binary\_operator", 1, {T\_EQUAL\_0}},

{"binary\_operator", 1, {T\_NOT\_EQUAL\_0}},

{"binary\_operator", 1, {T\_LESS\_OR\_EQUAL\_0}},

{"binary\_operator", 1, {T\_GREATER\_OR\_EQUAL\_0}},

{"binary\_operator", 1, {T\_ADD\_0}},

{"binary\_operator", 1, {T\_SUB\_0}},

{"binary\_operator", 1, {T\_MUL\_0}},

{"binary\_operator", 1, {T\_DIV\_0}},

{"binary\_operator", 1, {T\_MOD\_0}},

{"binary\_action", 2, {"binary\_operator","expression"}},

//

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

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

{"left\_expression", 1, {"ident\_terminal"}},

{"left\_expression", 1, {"value\_terminal"}},

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action\_\_\_\_iteration\_after\_two"}},

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action"}},

{"expression", 2, {"left\_expression","binary\_action\_\_\_\_iteration\_after\_two"}},

{"expression", 2, {"left\_expression","binary\_action"}},

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

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

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

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

//

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

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

//

{"bind\_right\_to\_left", 2, {"ident","rl\_expression"}},

{"bind\_left\_to\_right", 2, {"lr\_expression","ident"}},

//

{"body\_for\_true", 2, {"statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenSEMICOLON"}},

{"body\_for\_true", 2, {"statement\_in\_while\_body","tokenSEMICOLON"}},

{"body\_for\_true", 1, {T\_SEMICOLON\_0}},

{"tokenELSE\_\_statement\_in\_while\_body", 2, {"tokenELSE","statement\_in\_while\_body"}},

{"tokenELSE\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"tokenELSE","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"}},

{"body\_for\_false", 2, {"tokenELSE\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenSEMICOLON"}},

{"body\_for\_false", 2, {"tokenELSE\_\_statement\_in\_while\_body","tokenSEMICOLON"}},

{"body\_for\_false", 2, {"tokenELSE","tokenSEMICOLON"}},

{"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN", 2, {"tokenIF","tokenGROUPEXPRESSIONBEGIN"}},

{"expression\_\_tokenGROUPEXPRESSIONEND", 2, {"expression","tokenGROUPEXPRESSIONEND"}},

{"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN","expression\_\_tokenGROUPEXPRESSIONEND"}},

{"body\_for\_true\_\_body\_for\_false", 2, {"body\_for\_true","body\_for\_false"}},

{"cond\_block", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND","body\_for\_true\_\_body\_for\_false"}},

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

//

{"cycle\_counter", 1, {"ident\_terminal"}},

{"rl\_expression", 2, {"tokenRLBIND","expression"}},

{"lr\_expression", 2, {"expression","tokenLRBIND"}},

{"cycle\_counter\_init", 2, {"cycle\_counter","rl\_expression"}},

{"cycle\_counter\_init", 2, {"lr\_expression","cycle\_counter"}},

{"cycle\_counter\_last\_value", 1, {"value\_terminal"}},

{"cycle\_body", 2, {"tokenDO","statement\_\_\_\_iteration\_after\_two"}},

{"cycle\_body", 2, {"tokenDO","statement"}},

{"tokenFOR\_\_cycle\_counter\_init", 2, {"tokenFOR","cycle\_counter\_init"}},

{"tokenTO\_\_cycle\_counter\_last\_value", 2, {"tokenTO","cycle\_counter\_last\_value"}},

{"tokenFOR\_\_cycle\_counter\_init\_\_tokenTO\_\_cycle\_counter\_last\_value", 2, {"tokenFOR\_\_cycle\_counter\_init","tokenTO\_\_cycle\_counter\_last\_value"}},

{"cycle\_body\_\_tokenSEMICOLON", 2, {"cycle\_body","tokenSEMICOLON"}},

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

//

{"continue\_while", 2, {"tokenCONTINUE","tokenWHILE"}},

{"exit\_while", 2, {"tokenEXIT","tokenWHILE"}},

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

{"tokenEND\_\_tokenWHILE", 2, {"tokenEND","tokenWHILE"}},

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

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

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

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

{"while\_cycle", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},

//

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

{"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two", 2, {"tokenREPEAT","statement\_\_\_\_iteration\_after\_two"}},

{"tokenREPEAT\_\_statement", 2, {"tokenREPEAT","statement"}},

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},

{"repeat\_until\_cycle", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},

//

{"input\_\_first\_part", 2, {"tokenGET","tokenGROUPEXPRESSIONBEGIN"}},

{"input\_\_second\_part", 2, {"ident","tokenGROUPEXPRESSIONEND"}},

{"input", 2, {"input\_\_first\_part","input\_\_second\_part"}},

//

{"output\_\_first\_part", 2, {"tokenPUT","tokenGROUPEXPRESSIONBEGIN"}},

{"output\_\_second\_part", 2, {"expression","tokenGROUPEXPRESSIONEND"}},

{"output", 2, {"output\_\_first\_part","output\_\_second\_part"}},

//

{"statement", 2, {"ident","rl\_expression"}},

{"statement", 2, {"lr\_expression","ident"}},

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

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

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

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

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

{"statement", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},

{"statement", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},

{"statement", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},

{"statement", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},

{"statement", 2, {"ident","tokenCOLON"}},

{"statement", 2, {"tokenGOTO","ident"}},

{"statement", 2, {"input\_\_first\_part","input\_\_second\_part"}},

{"statement", 2, {"output\_\_first\_part","output\_\_second\_part"}},

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement\_\_\_\_iteration\_after\_two"}},

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement"}},

//

{ "statement\_in\_while\_body", 2, {"ident","rl\_expression"} },

{ "statement\_in\_while\_body", 2, {"lr\_expression","ident"} },

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

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

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

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

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

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

{ "statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"} },

{ "statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"} },

{ "statement\_in\_while\_body", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"} },

{ "statement\_in\_while\_body", 2, {"ident","tokenCOLON"} },

{ "statement\_in\_while\_body", 2, {"tokenGOTO","ident"} },

{ "statement\_in\_while\_body", 2, {"input\_\_first\_part","input\_\_second\_part"} },

{ "statement\_in\_while\_body", 2, {"output\_\_first\_part","output\_\_second\_part"} },

{ "statement\_in\_while\_body", 2, {"tokenCONTINUE","tokenWHILE"} },

{ "statement\_in\_while\_body", 2, {"tokenEXIT","tokenWHILE"} },

{ "statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"} },

{ "statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body"} },

//

{"tokenNAME\_\_program\_name", 2, {"tokenNAME","program\_name"}},

{"tokenSEMICOLON\_\_tokenBODY", 2, {"tokenSEMICOLON","tokenBODY"}},

{"tokenDATA\_\_declaration", 2, {"tokenDATA","declaration"}},

{"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY", 2, {"tokenNAME\_\_program\_name","tokenSEMICOLON\_\_tokenBODY"}},

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA\_\_declaration"}},

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA"}},

{"statement\_\_tokenEND", 2, {"statement","tokenEND"}},

{"statement\_\_\_\_iteration\_after\_two\_\_tokenEND", 2, {"statement\_\_\_\_iteration\_after\_two","tokenEND"}},

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_\_\_iteration\_after\_two\_\_tokenEND"}},

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_tokenEND"}},

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","tokenEND"}},

{"program", 2, {"program\_\_\_\_part1","program\_\_\_\_part2"}},

//

{"tokenCOLON", 1, {T\_COLON\_0}},

{"tokenGOTO", 1, {T\_GOTO\_0}},

{"tokenINTEGER16", 1, {T\_DATA\_TYPE\_0}},

{"tokenCOMMA", 1, {T\_COMA\_0}},

{"tokenNOT", 1, {T\_NOT\_0}},

{"tokenAND", 1, {T\_AND\_0}},

{"tokenOR", 1, {T\_OR\_0}},

{"tokenEQUAL", 1, {T\_EQUAL\_0}},

{"tokenNOTEQUAL", 1, {T\_NOT\_EQUAL\_0}},

{"tokenLESSOREQUAL", 1, {T\_LESS\_OR\_EQUAL\_0}},

{"tokenGREATEROREQUAL", 1, {T\_GREATER\_OR\_EQUAL\_0}},

{"tokenPLUS", 1, {T\_ADD\_0}},

{"tokenMINUS", 1, {T\_SUB\_0}},

{"tokenMUL", 1, {T\_MUL\_0}},

{"tokenDIV", 1, {T\_DIV\_0}},

{"tokenMOD", 1, {T\_MOD\_0}},

{"tokenGROUPEXPRESSIONBEGIN", 1, {"("}},

{"tokenGROUPEXPRESSIONEND", 1, {")"}},

{"tokenRLBIND", 1, {T\_RLBIND\_0}},

{"tokenLRBIND", 1, {T\_LRBIND\_0}},

{"tokenELSE", 1, {T\_ELSE\_0}},

{"tokenIF", 1, {T\_IF\_0}},

{"tokenDO", 1, {T\_DO\_0}},

{"tokenFOR", 1, {T\_FOR\_0}},

{"tokenTO", 1, {T\_TO\_0}},

{"tokenWHILE", 1, {T\_WHILE\_0}},

{"tokenCONTINUE", 1, {T\_CONTINUE\_WHILE\_0}},

{"tokenEXIT", 1, {T\_EXIT\_WHILE\_0}},

{"tokenREPEAT", 1, {T\_REPEAT\_0}},

{"tokenUNTIL", 1, {T\_UNTIL\_0}},

{"tokenGET", 1, {T\_INPUT\_0}},

{"tokenPUT", 1, {T\_OUTPUT\_0}},

{"tokenNAME", 1, {T\_NAME\_0}},

{"tokenBODY", 1, {T\_BODY\_0}},

{"tokenDATA", 1, {T\_DATA\_0}},

{"tokenEND", 1, {T\_END\_0}},

{"tokenSEMICOLON", 1, {T\_SEMICOLON\_0}},

//

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

//

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

//

// { "label", 1, {"ident\_terminal"} },

//

{ "", 2, {"",""} }

},

176,

"program"

#endif

};

Grammar originalGrammar = {

ORIGINAL\_GRAMMAR

#if 0

{

{"labeled\_point", 2, {"ident", "tokenCOLON"}}, // !!!!!

{"goto\_label", 2, {"tokenGOTO","ident"}}, // !!!!!

{"program\_name", 1, {"ident\_terminal"}},

{"value\_type", 1, {"INTEGER16"}},

{"other\_declaration\_ident", 2, {"tokenCOMMA", "ident"}},

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"other\_declaration\_ident","other\_declaration\_ident\_\_\_\_iteration\_after\_one", }},

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"tokenCOMMA", "ident"}},

{"value\_type\_\_ident", 2, {"value\_type", "ident"}},

{"declaration", 2, {"value\_type\_\_ident", "other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},

{"declaration", 2, {"value\_type", "ident"}},

//

{"unary\_operator", 1, {"NOT"}},

{"unary\_operator", 1, {"-"}},

{"unary\_operator", 1, {"+"}},

{"binary\_operator", 1, {"AND"}},

{"binary\_operator", 1, {"OR"}},

{"binary\_operator", 1, {"=="}},

{"binary\_operator", 1, {"!="}},

{"binary\_operator", 1, {"<="}},

{"binary\_operator", 1, {">="}},

{"binary\_operator", 1, {"+"}},

{"binary\_operator", 1, {"-"}},

{"binary\_operator", 1, {"\*"}},

{"binary\_operator", 1, {"DIV"}},

{"binary\_operator", 1, {"MOD"}},

{"binary\_action", 2, {"binary\_operator","expression"}},

//

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

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

{"left\_expression", 1, {"ident\_terminal"}},

{"left\_expression", 1, {"value\_terminal"}},

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action\_\_\_\_iteration\_after\_two"}},

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action"}},

{"expression", 2, {"left\_expression","binary\_action\_\_\_\_iteration\_after\_two"}},

{"expression", 2, {"left\_expression","binary\_action"}},

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

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

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

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

//

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

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

//

{"bind\_right\_to\_left", 2, {"ident","rl\_expression"}},

{"bind\_left\_to\_right", 2, {"lr\_expression","ident"}},

//

{"body\_for\_true", 2, {"statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenSEMICOLON"}},

{"body\_for\_true", 2, {"statement\_in\_while\_body","tokenSEMICOLON"}},

{"body\_for\_true", 1, {";"}},

{"tokenELSE\_\_statement\_in\_while\_body", 2, {"tokenELSE","statement\_in\_while\_body"}},

{"tokenELSE\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"tokenELSE","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"}},

{"body\_for\_false", 2, {"tokenELSE\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenSEMICOLON"}},

{"body\_for\_false", 2, {"tokenELSE\_\_statement\_in\_while\_body","tokenSEMICOLON"}},

{"body\_for\_false", 2, {"tokenELSE","tokenSEMICOLON"}},

{"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN", 2, {"tokenIF","tokenGROUPEXPRESSIONBEGIN"}},

{"expression\_\_tokenGROUPEXPRESSIONEND", 2, {"expression","tokenGROUPEXPRESSIONEND"}},

{"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN","expression\_\_tokenGROUPEXPRESSIONEND"}},

{"body\_for\_true\_\_body\_for\_false", 2, {"body\_for\_true","body\_for\_false"}},

{"cond\_block", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND","body\_for\_true\_\_body\_for\_false"}},

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

//

{"cycle\_counter", 1, {"ident\_terminal"}},

{"rl\_expression", 2, {"tokenRLBIND","expression"}},

{"lr\_expression", 2, {"expression","tokenLRBIND"}},

{"cycle\_counter\_init", 2, {"cycle\_counter","rl\_expression"}},

{"cycle\_counter\_init", 2, {"lr\_expression","cycle\_counter"}},

{"cycle\_counter\_last\_value", 1, {"value\_terminal"}},

{"cycle\_body", 2, {"tokenDO","statement\_\_\_\_iteration\_after\_two"}},

{"cycle\_body", 2, {"tokenDO","statement"}},

{"tokenFOR\_\_cycle\_counter\_init", 2, {"tokenFOR","cycle\_counter\_init"}},

{"tokenTO\_\_cycle\_counter\_last\_value", 2, {"tokenTO","cycle\_counter\_last\_value"}},

{"tokenFOR\_\_cycle\_counter\_init\_\_tokenTO\_\_cycle\_counter\_last\_value", 2, {"tokenFOR\_\_cycle\_counter\_init","tokenTO\_\_cycle\_counter\_last\_value"}},

{"cycle\_body\_\_tokenSEMICOLON", 2, {"cycle\_body","tokenSEMICOLON"}},

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

//

{"continue\_while", 2, {"tokenCONTINUE","tokenWHILE"}},

{"exit\_while", 2, {"tokenEXIT","tokenWHILE"}},

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

{"tokenEND\_\_tokenWHILE", 2, {"tokenEND","tokenWHILE"}},

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

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

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

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

{"while\_cycle", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},

//

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

{"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two", 2, {"tokenREPEAT","statement\_\_\_\_iteration\_after\_two"}},

{"tokenREPEAT\_\_statement", 2, {"tokenREPEAT","statement"}},

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},

{"repeat\_until\_cycle", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},

//

{"input\_\_first\_part", 2, {"tokenGET","tokenGROUPEXPRESSIONBEGIN"}},

{"input\_\_second\_part", 2, {"ident","tokenGROUPEXPRESSIONEND"}},

{"input", 2, {"input\_\_first\_part","input\_\_second\_part"}},

//

{"output\_\_first\_part", 2, {"tokenPUT","tokenGROUPEXPRESSIONBEGIN"}},

{"output\_\_second\_part", 2, {"expression","tokenGROUPEXPRESSIONEND"}},

{"output", 2, {"output\_\_first\_part","output\_\_second\_part"}},

//

{"statement", 2, {"ident","rl\_expression"}},

{"statement", 2, {"lr\_expression","ident"}},

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

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

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

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

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

{"statement", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},

{"statement", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},

{"statement", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},

{"statement", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},

{"statement", 2, {"ident","tokenCOLON"}},

{"statement", 2, {"tokenGOTO","ident"}},

{"statement", 2, {"input\_\_first\_part","input\_\_second\_part"}},

{"statement", 2, {"output\_\_first\_part","output\_\_second\_part"}},

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement\_\_\_\_iteration\_after\_two"}},

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement"}},

//

{"statement\_in\_while\_body", 2, {"ident","rl\_expression"} },

{"statement\_in\_while\_body", 2, {"lr\_expression","ident"} },

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

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

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

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

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

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

{"statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"} },

{"statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"} },

{"statement\_in\_while\_body", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"} },

{"statement\_in\_while\_body", 2, {"ident","tokenCOLON"} },

{"statement\_in\_while\_body", 2, {"tokenGOTO","ident"} },

{"statement\_in\_while\_body", 2, {"input\_\_first\_part","input\_\_second\_part"} },

{"statement\_in\_while\_body", 2, {"output\_\_first\_part","output\_\_second\_part"} },

{"statement\_in\_while\_body", 2, {"tokenCONTINUE","tokenWHILE"} },

{"statement\_in\_while\_body", 2, {"tokenEXIT","tokenWHILE"} },

{"statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"} },

{"statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body"} },

//

{"tokenNAME\_\_program\_name", 2, {"tokenNAME","program\_name"}},

{"tokenSEMICOLON\_\_tokenBODY", 2, {"tokenSEMICOLON","tokenBODY"}},

{"tokenDATA\_\_declaration", 2, {"tokenDATA","declaration"}},

{"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY", 2, {"tokenNAME\_\_program\_name","tokenSEMICOLON\_\_tokenBODY"}},

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA\_\_declaration"}},

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA"}},

{"statement\_\_tokenEND", 2, {"statement","tokenEND"}},

{"statement\_\_\_\_iteration\_after\_two\_\_tokenEND", 2, {"statement\_\_\_\_iteration\_after\_two","tokenEND"}},

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_\_\_iteration\_after\_two\_\_tokenEND"}},

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_tokenEND"}},

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","tokenEND"}},

{"program", 2, {"program\_\_\_\_part1","program\_\_\_\_part2"}},

//

{"tokenCOLON", 1, {":"}},

{"tokenGOTO", 1, {"GOTO"}},

{"tokenINTEGER16", 1, {"INTEGER16"}},

{"tokenCOMMA", 1, {","}},

{"tokenNOT", 1, {"NOT"}},

{"tokenAND", 1, {"AND"}},

{"tokenOR", 1, {"OR"}},

{"tokenEQUAL", 1, {"=="}},

{"tokenNOTEQUAL", 1, {"!="}},

{"tokenLESSOREQUAL", 1, {"<="}},

{"tokenGREATEROREQUAL", 1, {">="}},

{"tokenPLUS", 1, {"+"}},

{"tokenMINUS", 1, {"-"}},

{"tokenMUL", 1, {"\*"}},

{"tokenDIV", 1, {"DIV"}},

{"tokenMOD", 1, {"MOD"}},

{"tokenGROUPEXPRESSIONBEGIN", 1, {"("}},

{"tokenGROUPEXPRESSIONEND", 1, {")"}},

{"tokenRLBIND", 1, {"<<"}},

{"tokenLRBIND", 1, {">>"}},

{"tokenELSE", 1, {"ELSE"}},

{"tokenIF", 1, {"IF"}},

{"tokenDO", 1, {"DO"}},

{"tokenFOR", 1, {"FOR"}},

{"tokenTO", 1, {"TO"}},

{"tokenWHILE", 1, {"WHILE"}},

{"tokenCONTINUE", 1, {"CONTINUE"}},

{"tokenEXIT", 1, {"EXIT"}},

{"tokenREPEAT", 1, {"REPEAT"}},

{"tokenUNTIL", 1, {"UNTIL"}},

{"tokenGET", 1, {"GET"}},

{"tokenPUT", 1, {"PUT"}},

{"tokenNAME", 1, {"NAME"}},

{"tokenBODY", 1, {"BODY"}},

{"tokenDATA", 1, {"DATA"}},

{"tokenEND", 1, {"END"}},

{"tokenSEMICOLON", 1, {";"}},

//

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

//

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

//

// { "label", 1, {"ident\_terminal"} },

//

{ "", 2, {"",""} }

},

176,

"program"

#endif

};

#define DEBUG\_STATES

#define MAX\_LEXEMS 256

//#define MAX\_RULES 128

#define MAX\_SYMBOLS 64

typedef struct {

char symbols[MAX\_SYMBOLS][MAX\_TOKEN\_SIZE];

int count;

} SymbolSet;

typedef SymbolSet ParseInfoTable[MAX\_LEXEMS][MAX\_LEXEMS];

bool insertIntoSymbolSet(SymbolSet\* set, const char\* symbol) {

for (int i = 0; i < set->count; ++i) {

if (strcmp(set->symbols[i], symbol) == 0) {

// symbol already exists

return false;

}

}

strncpy(set->symbols[set->count], symbol, MAX\_TOKEN\_SIZE);

set->symbols[set->count][MAX\_TOKEN\_SIZE - 1] = '\0';

++set->count;

return true;

}

bool containsSymbolSet(const SymbolSet\* set, const char\* symbol) {

for (int i = 0; i < set->count; ++i) {

if (strcmp(set->symbols[i], symbol) == 0) {

return true;

}

}

return false;

}

// initialize with empty SymbolSets

ParseInfoTable parseInfoTable = { {{0}} };

struct ASTNode {

std::string value;

bool isTerminal;

std::vector<ASTNode\*> children;

ASTNode(const std::string& val, bool isTerminal) : isTerminal(isTerminal), value(val) {}

~ASTNode() {

for (ASTNode\* child : children) {

delete child;

}

}

};

ASTNode\* buildASTByCPPMap(const std::map<int, std::map<int, std::set<std::string>>>& parseInfoTable,

Grammar\* grammar,

int start,

int end,

const std::string& symbol) {

if (start > end) return nullptr;

ASTNode\* node = new ASTNode(symbol, false);

for (const Rule& rule : grammar->rules) {

if (rule.lhs != symbol) continue;

if (rule.rhs\_count == 1) {

//if (parseInfoTable.at(start).at(end).count(rule.rhs[0])) {

node->children.push\_back(new ASTNode(rule.rhs[0], true));

return node;

//}

}

else if (rule.rhs\_count == 2) {

for (int split = start; split < end; ++split) {

if (parseInfoTable.at(start).at(split).count(rule.rhs[0]) &&

parseInfoTable.at(split + 1).at(end).count(rule.rhs[1])) {

node->children.push\_back(buildASTByCPPMap(parseInfoTable, grammar, start, split, rule.rhs[0]));

node->children.push\_back(buildASTByCPPMap(parseInfoTable, grammar, split + 1, end, rule.rhs[1]));

return node;

}

}

}

}

return nullptr;

}

ASTNode\* buildAST(//const std::map<int, std::map<int, std::set<std::string>>>& parseInfoTable,

ParseInfoTable& parseInfoTable,

Grammar\* grammar,

int start,

int end,

const std::string& symbol) {

if (start > end) return nullptr;

ASTNode\* node = new ASTNode(symbol, false);

for (const Rule& rule : grammar->rules) {

if (rule.lhs != symbol) continue;

if (rule.rhs\_count == 1) {

//if (parseInfoTable.at(start).at(end).count(rule.rhs[0])) {

node->children.push\_back(new ASTNode(rule.rhs[0], true));

return node;

//}

}

else if (rule.rhs\_count == 2) {

for (int split = start; split < end; ++split) {

if (containsSymbolSet(&parseInfoTable[start][split], rule.rhs[0]) &&

containsSymbolSet(&parseInfoTable[split + 1][end], rule.rhs[1])) {

node->children.push\_back(buildAST(parseInfoTable, grammar, start, split, rule.rhs[0]));

node->children.push\_back(buildAST(parseInfoTable, grammar, split + 1, end, rule.rhs[1]));

return node;

}

}

}

}

return nullptr;

}

void printAST(struct LexemInfo\* lexemInfoTable, const ASTNode\* node, int depth = 0) {

static int lexemInfoTableIndexForPrintAST = 0; // ATTENTION: multithreading is not supported for this!

if (!node) {

return;

}

if (!depth) {

lexemInfoTableIndexForPrintAST = 0;

}

for (unsigned int depthIndex = 0; depthIndex <= depth; ++depthIndex) {

std::cout << " " << "|";

}

std::cout << "--";

if (node->isTerminal) {

std::cout << "\"" << lexemInfoTable[lexemInfoTableIndexForPrintAST++].lexemStr << "\"";

}

else {

std::cout << node->value;

}

std::cout << "\n";

for (const ASTNode\* child : node->children) {

printAST(lexemInfoTable, child, depth + 1);

}

}

void printASTToFile(struct LexemInfo\* lexemInfoTable, const ASTNode\* node, std::ofstream& outFile, int depth = 0) {

static int lexemInfoTableIndexForPrintAST = 0; // ATTENTION: multithreading is not supported for this!

if (!node) {

return;

}

if (!depth) {

lexemInfoTableIndexForPrintAST = 0;

}

for (unsigned int depthIndex = 0; depthIndex <= depth; ++depthIndex) {

outFile << " |";

}

outFile << "--";

if (node->isTerminal) {

outFile << "\"" << lexemInfoTable[lexemInfoTableIndexForPrintAST++].lexemStr << "\"";

}

else {

outFile << node->value;

}

outFile << "\n";

for (const ASTNode\* child : node->children) {

printASTToFile(lexemInfoTable, child, outFile, depth + 1);

}

}

void printAST\_\_OLD\_123(struct LexemInfo\* lexemInfoTable, const ASTNode\* node, int depth = 0) {

static int lexemInfoTableIndexForPrintAST = 0; // ATTENTION: multithreading is not supported for this!

if (!node) {

return;

}

if (!depth) {

lexemInfoTableIndexForPrintAST = 0;

}

for (unsigned int depthIndex = 0; depthIndex <= depth; ++depthIndex) {

std::cout << " " << "|";

}

std::cout << "--";

if (node->isTerminal) {

std::cout << "\"" << lexemInfoTable[lexemInfoTableIndexForPrintAST++].lexemStr << "\"";

}

else {

std::cout << node->value;

}

std::cout << "\n";

for (const ASTNode\* child : node->children) {

printAST(lexemInfoTable, child, depth + 1);

}

}

void displayParseInfoTable(const map<int, map<int, set<string>>>& parseInfoTable) {

constexpr int CELL\_WIDTH = 128;

cout << left << setw(CELL\_WIDTH) << "[i\\j]";

for (const auto& outerEntry : parseInfoTable) {

cout << setw(CELL\_WIDTH) << outerEntry.first;

}

cout << endl;

for (const auto& outerEntry : parseInfoTable) {

int i = outerEntry.first;

cout << setw(CELL\_WIDTH) << i;

for (const auto& innerEntry : parseInfoTable) {

int j = innerEntry.first;

if (parseInfoTable.at(i).find(j) != parseInfoTable.at(i).end()) {

const set<string>& rules = parseInfoTable.at(i).at(j);

string cellContent;

for (const string& rule : rules) {

cellContent += rule + ", ";

}

if (!cellContent.empty()) {

cellContent.pop\_back();

cellContent.pop\_back();

}

cout << setw(CELL\_WIDTH) << cellContent;

}

else {

cout << setw(CELL\_WIDTH) << "-";

}

}

cout << endl;

}

}

void saveParseInfoTableToFile(const map<int, map<int, set<string>>>& parseInfoTable, const string& filename) {

constexpr int CELL\_WIDTH = 128;

ofstream file(filename);

if (!file.is\_open()) {

cerr << "Error: Unable to open file " << filename << endl;

return;

}

file << left << setw(CELL\_WIDTH) << "[i\\j]";

for (const auto& outerEntry : parseInfoTable) {

file << setw(CELL\_WIDTH) << outerEntry.first;

}

file << endl;

for (const auto& outerEntry : parseInfoTable) {

int i = outerEntry.first;

file << setw(CELL\_WIDTH) << i;

for (const auto& innerEntry : parseInfoTable) {

int j = innerEntry.first;

if (parseInfoTable.at(i).find(j) != parseInfoTable.at(i).end()) {

const set<string>& rules = parseInfoTable.at(i).at(j);

string cellContent;

for (const string& rule : rules) {

cellContent += rule + ", ";

}

if (!cellContent.empty()) {

cellContent.pop\_back();

cellContent.pop\_back();

}

file << setw(CELL\_WIDTH) << cellContent;

}

else {

file << setw(CELL\_WIDTH) << "-";

}

}

file << endl;

}

file.close();

}

bool cykAlgorithmImplementation(struct LexemInfo\* lexemInfoTable, Grammar\* grammar, char \* astFileName) {

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

return false;

}

#if defined(\_DEBUG)

printf("ATTENTION: for better performance, use Release mode!\r\n");

#endif

#ifndef DEBUG\_STATES

cout << "cykParse in progress.....[please wait]";

#else

cout << "cykParse in progress.....[please wait]: ";

#endif

// 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";

if (!containsSymbolSet(&parseInfoTable[0][lexemIndex - 1], grammar->start\_symbol)) {

return false;

}

ASTNode\* astRoot = buildAST(parseInfoTable, grammar, 0, lexemIndex - 1, grammar->start\_symbol);

if (astRoot) {

std::cout << "Abstract Syntax Tree:\n";

printAST(lexemInfoTable, astRoot);

if (astFileName && astFileName[0] != '\0') {

std::ofstream astOFStream(astFileName, std::ofstream::out);

printASTToFile(lexemInfoTable, astRoot, astOFStream);

astOFStream.close();

printf("File \"%s\" saved.\n", astFileName);

}

delete astRoot;

}

else {

std::cout << "Failed to build AST.\n";

}

return true;

}

#define MAX\_STACK\_DEPTH 256

bool recursiveDescentParserRuleWithDebug(const char\* ruleName, int& lexemIndex, LexemInfo\* lexemInfoTable, Grammar\* grammar, int depth, const struct LexemInfo\*\* unexpectedLexemfailedTerminal) {

if (depth > MAX\_STACK\_DEPTH) {

printf("Error: Maximum recursion depth reached.\n");

return false;

}

char isError = false;

for (int i = 0; i < grammar->rule\_count; ++i) {

Rule& rule = grammar->rules[i];

if (strcmp(rule.lhs, ruleName) != 0) continue;

int savedIndex = lexemIndex;

if (rule.rhs\_count == 1) {

if (

lexemInfoTable[lexemIndex].tokenType == IDENTIFIER\_LEXEME\_TYPE && !strcmp(rule.rhs[0], "ident\_terminal")

|| lexemInfoTable[lexemIndex].tokenType == VALUE\_LEXEME\_TYPE && !strcmp(rule.rhs[0], "value\_terminal")

|| !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")

|| !strncmp(rule.rhs[0], lexemInfoTable[lexemIndex].lexemStr, MAX\_LEXEM\_SIZE)

) {

++lexemIndex;

return nullptr;

}

else {

currUnexpectedLexemfailedTerminalPtr = lexemInfoTable + lexemIndex;

}

}

else if (rule.rhs\_count == 2) {

if (nullptr == (returnUnexpectedLexemfailedTerminalPtr = recursiveDescentParserWithDebug\_(rule.rhs[0], lexemIndex, lexemInfoTable, grammar, depth + 1, unexpectedUnknownLexemfailedTerminal))

&& nullptr == (returnUnexpectedLexemfailedTerminalPtr = recursiveDescentParserWithDebug\_(rule.rhs[1], lexemIndex, lexemInfoTable, grammar, depth + 1, unexpectedUnknownLexemfailedTerminal))) {

return nullptr;

}

}

lexemIndex = savedIndex;

}

if (returnUnexpectedLexemfailedTerminalPtr != nullptr && returnUnexpectedLexemfailedTerminalPtr != unexpectedUnknownLexemfailedTerminal

&&( returnUnexpectedLexemfailedTerminalPtr->tokenType == IDENTIFIER\_LEXEME\_TYPE

|| returnUnexpectedLexemfailedTerminalPtr->tokenType == VALUE\_LEXEME\_TYPE

|| returnUnexpectedLexemfailedTerminalPtr->tokenType == KEYWORD\_LEXEME\_TYPE

)) {

return returnUnexpectedLexemfailedTerminalPtr;

}

if (currUnexpectedLexemfailedTerminalPtr != nullptr) {

return currUnexpectedLexemfailedTerminalPtr;

}

if(returnUnexpectedLexemfailedTerminalPtr != nullptr){

return returnUnexpectedLexemfailedTerminalPtr;

}

return unexpectedUnknownLexemfailedTerminal;

}

//

int syntaxAnalyze(LexemInfo\* lexemInfoTable, Grammar\* grammar, char syntaxlAnalyzeMode, char\* astFileName, char\* errorMessagesPtrToLastBytePtr) {

bool cykAlgorithmImplementationReturnValue = false;

if (syntaxlAnalyzeMode == SYNTAX\_ANALYZE\_BY\_CYK\_ALGORITHM) {

cykAlgorithmImplementationReturnValue = cykAlgorithmImplementation(lexemesInfoTable, grammar, astFileName);

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

if (cykAlgorithmImplementationReturnValue) {

return SUCCESS\_STATE;

}

else {

writeBytesToFile(astFileName, (unsigned char\*)"Error of AST build", strlen("Error of AST build"));

}

}

else if (astFileName && astFileName[0] != '\0') {

writeBytesToFile(astFileName, (unsigned char\*)"AST build no support.", strlen("AST build no support."));

}

if (cykAlgorithmImplementationReturnValue == false || syntaxlAnalyzeMode == SYNTAX\_ANALYZE\_BY\_RECURSIVE\_DESCENT) {

int lexemIndex = 0;

const struct LexemInfo\* unexpectedLexemfailedTerminal = nullptr;

if (recursiveDescentParserRuleWithDebug(grammar->start\_symbol, lexemIndex, lexemInfoTable, grammar, 0, &unexpectedLexemfailedTerminal)) {

if (lexemInfoTable[lexemIndex].lexemStr[0] == '\0') {

printf("Parse successful.\n");

printf("%d.\n", lexemIndex);

return SUCCESS\_STATE;

}

else {

printf("Parse failed: Extra tokens remain.\r\n");

errorMessagesPtrToLastBytePtr += sprintf(errorMessagesPtrToLastBytePtr, "Parse failed: Extra tokens remain.\r\n");

return ~SUCCESS\_STATE;

}

}

else {

if (unexpectedLexemfailedTerminal) {

printf("Parse failed.\r\n");

printf(" (The predicted terminal does not match the expected one.\r\n Possible unexpected terminal \"%s\" on line %lld at position %lld\r\n ..., but this is not certain.)\r\n", unexpectedLexemfailedTerminal->lexemStr, unexpectedLexemfailedTerminal->row, unexpectedLexemfailedTerminal->col);

errorMessagesPtrToLastBytePtr += sprintf(errorMessagesPtrToLastBytePtr, "Parse failed.\r\n");

errorMessagesPtrToLastBytePtr += snprintf(errorMessagesPtrToLastBytePtr, MAX\_LEXEM\_SIZE + 128 + strlen(" (The predicted terminal does not match the expected one.\r\n Possible unexpected terminal \"#\" on line # at position #\r\n ..., but this is not certain.)\r\n"), " (The predicted terminal does not match the expected one.\r\n Possible unexpected terminal \"%s\" on line %lld at position %lld\r\n ..., but this is not certain.)\r\n", unexpectedLexemfailedTerminal->lexemStr, unexpectedLexemfailedTerminal->row, unexpectedLexemfailedTerminal->col);

}

else {

printf("Parse failed: unexpected terminal.\r\n");

errorMessagesPtrToLastBytePtr += sprintf(errorMessagesPtrToLastBytePtr, "Parse failed: unexpected terminal.\r\n");

}

return ~SUCCESS\_STATE;

}

}

return ~SUCCESS\_STATE;

}

bool syntaxlAnalyze\_(LexemInfo\* lexemInfoTable, Grammar\* grammar, char syntaxlAnalyzeMode, char\* astFileName, char\* errorMessagesPtrToLastBytePtr) {

bool cykAlgorithmImplementationReturnValue = false;

if (syntaxlAnalyzeMode == SYNTAX\_ANALYZE\_BY\_CYK\_ALGORITHM) {

bool cykAlgorithmImplementationReturnValue = cykAlgorithmImplementation(lexemesInfoTable, grammar, astFileName);

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

if(!cykAlgorithmImplementationReturnValue) {

writeBytesToFile(astFileName, (unsigned char\*)"Error of AST build", strlen("Error of AST build"));

}

}

else if(astFileName && astFileName[0] != '\0') {

writeBytesToFile(astFileName, (unsigned char\*)"AST build no support.", strlen("AST build no support."));

}

if (cykAlgorithmImplementationReturnValue && syntaxlAnalyzeMode == SYNTAX\_ANALYZE\_BY\_RECURSIVE\_DESCENT) {

int lexemIndex = 0;

const struct LexemInfo unexpectedUnknownLexemfailedTerminal("unknown", 0, 0, 0, ~0, ~0); //

const struct LexemInfo\* returnUnexpectedLexemfailedTerminal = nullptr;

if (nullptr == (returnUnexpectedLexemfailedTerminal = recursiveDescentParserWithDebug\_(grammar->start\_symbol, lexemIndex, lexemInfoTable, grammar, 0, &unexpectedUnknownLexemfailedTerminal))) {

if (lexemInfoTable[lexemIndex].lexemStr[0] == '\0') {

printf("Parse successful.\n");

printf("%d.\n", lexemIndex);

return true;

}

else {

printf("Parse failed: Extra tokens remain.\n");

return false;

}

}

else {

if (returnUnexpectedLexemfailedTerminal->lexemStr[1]) {

printf("Parse failed.\r\n");

printf(" (The predicted terminal does not match the expected one.\r\n Possible unexpected terminal \"%s\" on line %lld at position %lld\r\n ..., but this is not certain.)\r\n", returnUnexpectedLexemfailedTerminal->lexemStr, returnUnexpectedLexemfailedTerminal->row, returnUnexpectedLexemfailedTerminal->col);

}

else {

printf("Parse failed: unexpected terminal.\r\n");

}

return false;

}

return false;

}

return false;

}

// OLD //

bool cykAlgorithmImplementationByCPPMap(struct LexemInfo\* lexemInfoTable, Grammar\* grammar) {

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

return false;

}

#if defined(\_DEBUG)

printf("ATTENTION: for better performance, use Release mode!\r\n");

#endif

#ifndef DEBUG\_STATES

cout << "cykParse in progress.....[please wait]";

#else

cout << "cykParse in progress.....[please wait]: ";

#endif

map<int, map<int, set<string>>> parseInfoTable;

int lexemIndex = 0;

for (--lexemIndex; lexemInfoTable[++lexemIndex].lexemStr[0];) {

#ifdef DEBUG\_STATES

printf("\rcykParse in progress.....[please wait]: %02d %16s", lexemIndex, lexemInfoTable[lexemIndex].lexemStr);

#endif

// Iterate over the rules

for (int xIndex = 0; xIndex < grammar->rule\_count; ++xIndex) {

string&& lhs = grammar->rules[xIndex].lhs;

Rule& rule = grammar->rules[xIndex];

// If a terminal is found

if (rule.rhs\_count == 1 && (

lexemInfoTable[lexemIndex].tokenType == IDENTIFIER\_LEXEME\_TYPE && !strcmp(rule.rhs[0], "ident\_terminal")

|| lexemInfoTable[lexemIndex].tokenType == VALUE\_LEXEME\_TYPE && !strcmp(rule.rhs[0], "value\_terminal")

|| !strncmp(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 = buildASTByCPPMap(parseInfoTable, grammar, 0, lexemIndex - 1, grammar->start\_symbol);

if (astRoot) {

std::cout << "Abstract Syntax Tree:\n";

printAST(lexemInfoTable, astRoot);

delete astRoot; // Не забуваємо звільняти пам'ять

}

else {

std::cout << "Failed to build AST.\n";

}

//return parseInfoTable[0][lexemIndex - 1].find(grammar->start\_symbol) != parseInfoTable[0][lexemIndex - 1].end(); // return !!parseInfoTable[0][lexemIndex - 1].size();

return true;

}

#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  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: while.cpp \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

#include "stdio.h"

#include "string.h"

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

unsigned char multitokenSize = detectMultiToken(\*lastLexemInfoInTable, MULTI\_TOKEN\_WHILE);

if (multitokenSize) {

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = \*\*lastLexemInfoInTable;

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++] = \*\*lastLexemInfoInTable;

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

//

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL@WHILE\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //\"%s\"\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

//

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL\_\_WHILE\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr);

}

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;

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;after cond expresion (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //after cond expresion (after \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

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

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

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++ - 1].lexemStr, MAX\_LEXEM\_SIZE);

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

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize++ - 1].lexemStr, MAX\_LEXEM\_SIZE);

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, " cmp eax, 0\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jz LABEL@AFTER\_WHILE\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " if (opTemp == 0) goto LABEL\_\_AFTER\_WHILE\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr);

}

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

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_THEN][0], MAX\_LEXEM\_SIZE)

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI\_TOKEN\_IF][0], MAX\_LEXEM\_SIZE)

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr, tokenStruct[MULTI\_TOKEN\_WHILE][0], MAX\_LEXEM\_SIZE)

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 6].lexemStr, tokenStruct[MULTI\_TOKEN\_WHILE][0], MAX\_LEXEM\_SIZE)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_jmp\_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jmp\_offset, 5);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;continue while (in \"then\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //continue while (in \"then\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

\*(unsigned int\*)(currBytePtr - 4) = (unsigned int)((unsigned char\*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 6].ifvalue - currBytePtr);

}

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr, tokenStruct[MULTI\_TOKEN\_CONTINUE\_WHILE][0], MAX\_LEXEM\_SIZE);

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@WHILE\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 6].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_WHILE\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 6].lexemStr);

}

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)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_jmp\_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jmp\_offset, 5);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;continue while (in \"else\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //continue while (in \"else\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

//lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].ifvalue = (unsigned long long int)(currBytePtr - 4);

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

\*(unsigned int\*)(currBytePtr - 4) = (unsigned int)((unsigned char\*)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].ifvalue - currBytePtr);

}

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr, tokenStruct[MULTI\_TOKEN\_CONTINUE\_WHILE][0], MAX\_LEXEM\_SIZE);

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@WHILE\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_WHILE\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr);

}

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)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_jmp\_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jmp\_offset, 5);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;continue while (in \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //continue while (in \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

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

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

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

}

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI\_TOKEN\_CONTINUE\_WHILE][0], MAX\_LEXEM\_SIZE);

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@WHILE\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_WHILE\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr);

}

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

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_THEN][0], MAX\_LEXEM\_SIZE)

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI\_TOKEN\_IF][0], MAX\_LEXEM\_SIZE)

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr, tokenStruct[MULTI\_TOKEN\_WHILE][0], MAX\_LEXEM\_SIZE)

&& !strncmp(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 6].lexemStr, tokenStruct[MULTI\_TOKEN\_WHILE][0], MAX\_LEXEM\_SIZE)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_jmp\_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jmp\_offset, 5);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;exit while (in \"then\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //exit while (in \"then\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].ifvalue = (unsigned long long int)(currBytePtr - 4);

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr, tokenStruct[MULTI\_TOKEN\_EXIT\_WHILE][0], MAX\_LEXEM\_SIZE);

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@AFTER\_WHILE\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_AFTER\_WHILE\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 5].lexemStr);

}

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)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_jmp\_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jmp\_offset, 5);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;exit while (in \"else\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //exit while (in \"else\"-part of %s-operator)\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

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

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 2].lexemStr, tokenStruct[MULTI\_TOKEN\_EXIT\_WHILE][0], MAX\_LEXEM\_SIZE);

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@AFTER\_WHILE\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_AFTER\_WHILE\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr);

}

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)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_jmp\_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 }; // jmp

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jmp\_offset, 5);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;exit while (in \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //exit while (in \"%s\")\r\n", tokenStruct[MULTI\_TOKEN\_WHILE][0]);

}

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

strncpy(lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 1].lexemStr, tokenStruct[MULTI\_TOKEN\_EXIT\_WHILE][0], MAX\_LEXEM\_SIZE);

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@AFTER\_WHILE\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_AFTER\_WHILE\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr);

}

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

}

return currBytePtr;

}

unsigned char\* makePostWhileCode\_(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode, unsigned char depthOfСontext) {

// if (!strncmp(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);

// }

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

const unsigned char code\_\_jmp\_offset[] = { 0xE9, 0x00, 0x00, 0x00, 0x00 };

currBytePtr = outBytes2Code(currBytePtr, (unsigned char\*)code\_\_jmp\_offset, 5);

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == C\_CODER\_MODE) {

//

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

\*(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 (!strncmp(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);

}

}

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " jmp LABEL@WHILE\_%016llX\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr);

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL@AFTER\_WHILE\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr);

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += snprintf((char\*)currBytePtr, 8192, " goto LABEL\_\_WHILE\_%016llX;\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 4].lexemStr);

currBytePtr += snprintf((char\*)currBytePtr, 8192, " LABEL\_\_AFTER\_WHILE\_%016llX:\r\n", (unsigned long long int)lexemInfoTransformationTempStack[lexemInfoTransformationTempStackSize - 3].lexemStr);

}

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)

) {

if (generatorMode == MACHINE\_X86\_WIN32\_CODER\_MODE) {

//

}

else if (generatorMode == ASSEMBLY\_X86\_WIN32\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " ;end of while\r\n");

}

else if (generatorMode == C\_CODER\_MODE) {

currBytePtr += sprintf((char\*)currBytePtr, "\r\n");

currBytePtr += snprintf((char\*)currBytePtr, 8192, " //end of while\r\n");

}

currBytePtr = makePostWhileCode\_(lastLexemInfoInTable, currBytePtr, generatorMode, 0);

lexemInfoTransformationTempStackSize -= 4;

return \*lastLexemInfoInTable += multitokenSize, currBytePtr;

}

return currBytePtr;

}  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: add.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define ADD\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeAddCode(B, C, M);

unsigned char\* makeAddCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: and.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define AND\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeAndCode(B, C, M);

unsigned char\* makeAndCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: bitwise\_and.h \*

\* (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);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: bitwise\_or.h \*

\* (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);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: cw\_lex.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define PATH\_NAME\_LENGH 2048

#define MAX\_PARAMETERS\_SIZE 4096

#define PARAMETERS\_COUNT 32

//#define INPUT\_FILENAME\_PARAMETER 0

#define INPUT\_FILENAME\_WITH\_EXTENSION\_PARAMETER 1

#define OUT\_LEXEMES\_SEQUENSE\_FILENAME\_WITH\_EXTENSION\_PARAMETER 2

#define OUT\_LEXEME\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER 3

#define OUT\_AST\_FILENAME\_WITH\_EXTENSION\_PARAMETER 4

#define OUT\_SYNTAX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER 5

#define OUT\_SEMANTIX\_ERROR\_FILENAME\_WITH\_EXTENSION\_PARAMETER 6

#define OUT\_PREPARED\_LEXEMES\_SEQUENSE\_FILENAME\_WITH\_EXTENSION\_PARAMETER 7

#define OUT\_C\_FILENAME\_WITH\_EXTENSION\_PARAMETER 8

#define OUT\_ASSEMBLY\_FILENAME\_WITH\_EXTENSION\_PARAMETER 9

#define OUT\_OBJECT\_FILENAME\_WITH\_EXTENSION\_PARAMETER 10

#define OUT\_BINARY\_FILENAME\_WITH\_EXTENSION\_PARAMETER 11

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

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

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

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

//#include "stdio.h"

//#include "stdlib.h"

//#include "string.h"

#define DEFAULT\_INPUT\_FILENAME "../test\_programs/file2.cwl" // TODO: move!

extern unsigned long long int mode;

extern char parameters[PARAMETERS\_COUNT][MAX\_PARAMETERS\_SIZE];

void comandLineParser(int argc, char\* argv[], unsigned long long int\* mode, char(\*parameters)[MAX\_PARAMETERS\_SIZE]);

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

size\_t loadSource(char\*\* text, char\* fileName);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: config.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

//#define LEXICAL\_ANALISIS\_MODE 1

//#define SEMANTIC\_ANALISIS\_MODE 2

//#define FULL\_COMPILER\_MODE 4

//#define DEBUG\_MODE 512

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

//#define DEFAULT\_MODE (DEBUG\_MODE | LEXICAL\_ANALISIS\_MODE | SYNTAX\_ANALISIS\_MODE | SEMANTIC\_ANALISIS\_MODE | MAKE\_ASSEMBLY | MAKE\_BINARY)

#define TOKENS\_RE ";|<-|\\+|-|\\\*|,|eq|noteq|:|\\(|\\)|less|gr|[\_0-9A-Za-z]+|[^ \t\r\f\v\n]"

#define KEYWORDS\_RE ";|<-|\\+|-|\\\*|,|eq|noteq|:|\\(|\\)|startprogram|variable|startblok|endblok|EXIT|CONTINUE|get|put|if|else|for|to|downto|do|while|repeat|until|goto|/|%|<=|>=|!|and|or|int\_2"

#define IDENTIFIERS\_RE "\_[A-Z][A-Z][A-Z]"

#define UNSIGNEDVALUES\_RE "0|[1-9][0-9]\*"

// first column of the cw term paper option

#define PROGRAM\_FORMAT \

{"tokenNAME\_\_program\_name", 2, {"tokenNAME","program\_name"}},\

{"tokenSEMICOLON\_\_tokenBODY", 2, {"tokenSEMICOLON","tokenBODY"}},\

{"tokenDATA\_\_declaration", 2, {"tokenDATA","declaration"}},\

{"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY", 2, {"tokenNAME\_\_program\_name","tokenSEMICOLON\_\_tokenBODY"}},\

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA\_\_declaration"}},\

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA"}},\

{"statement\_\_tokenEND", 2, {"statement","tokenEND"}},\

{"statement\_\_\_\_iteration\_after\_two\_\_tokenEND", 2, {"statement\_\_\_\_iteration\_after\_two","tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_\_\_iteration\_after\_two\_\_tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","tokenEND"}},\

{"program", 2, {"program\_\_\_\_part1","program\_\_\_\_part2"}},

#define T\_NAME\_0 "startprogram"

#define T\_NAME\_1 ""

#define T\_NAME\_2 ""

#define T\_NAME\_3 ""

#define T\_BODY\_0 "variable"

#define T\_BODY\_1 ""

#define T\_BODY\_2 ""

#define T\_BODY\_3 ""

#define T\_DATA\_0 "startblok"

#define T\_DATA\_1 ""

#define T\_DATA\_2 ""

#define T\_DATA\_3 ""

#define T\_DATA\_TYPE\_0 "int\_2"

#define T\_DATA\_TYPE\_1 ""

#define T\_DATA\_TYPE\_2 ""

#define T\_DATA\_TYPE\_3 ""

//

#define T\_BITWISE\_NOT\_0 ""

#define T\_BITWISE\_NOT\_1 ""

#define T\_BITWISE\_NOT\_2 ""

#define T\_BITWISE\_NOT\_3 ""

#define T\_BITWISE\_AND\_0 ""

#define T\_BITWISE\_AND\_1 ""

#define T\_BITWISE\_AND\_2 ""

#define T\_BITWISE\_AND\_3 ""

#define T\_BITWISE\_OR\_0 ""

#define T\_BITWISE\_OR\_1 ""

#define T\_BITWISE\_OR\_2 ""

#define T\_BITWISE\_OR\_3 ""

#define T\_NOT\_0 "not"

#define T\_NOT\_1 ""

#define T\_NOT\_2 ""

#define T\_NOT\_3 ""

#define T\_AND\_0 "and"

#define T\_AND\_1 ""

#define T\_AND\_2 ""

#define T\_AND\_3 ""

#define T\_OR\_0 "or"

#define T\_OR\_1 ""

#define T\_OR\_2 ""

#define T\_OR\_3 ""

//

#define T\_EQUAL\_0 "eq"

#define T\_EQUAL\_1 ""

#define T\_EQUAL\_2 ""

#define T\_EQUAL\_3 ""

#define T\_NOT\_EQUAL\_0 "noteq"

#define T\_NOT\_EQUAL\_1 ""

#define T\_NOT\_EQUAL\_2 ""

#define T\_NOT\_EQUAL\_3 ""

#define T\_LESS\_0 "less"

#define T\_LESS\_1 ""

#define T\_LESS\_2 ""

#define T\_LESS\_3 ""

#define T\_GREATER\_0 "gr"

#define T\_GREATER\_1 ""

#define T\_GREATER\_2 ""

#define T\_GREATER\_3 ""

#define T\_LESS\_OR\_EQUAL\_0 ""

#define T\_LESS\_OR\_EQUAL\_1 ""

#define T\_LESS\_OR\_EQUAL\_2 ""

#define T\_LESS\_OR\_EQUAL\_3 ""

#define T\_GREATER\_OR\_EQUAL\_0 ""

#define T\_GREATER\_OR\_EQUAL\_1 ""

#define T\_GREATER\_OR\_EQUAL\_2 ""

#define T\_GREATER\_OR\_EQUAL\_3 ""

//

#define T\_ADD\_0 "+"

#define T\_ADD\_1 ""

#define T\_ADD\_2 ""

#define T\_ADD\_3 ""

#define T\_SUB\_0 "-"

#define T\_SUB\_1 ""

#define T\_SUB\_2 ""

#define T\_SUB\_3 ""

#define T\_MUL\_0 "\*"

#define T\_MUL\_1 ""

#define T\_MUL\_2 ""

#define T\_MUL\_3 ""

#define T\_DIV\_0 "/"

#define T\_DIV\_1 ""

#define T\_DIV\_2 ""

#define T\_DIV\_3 ""

#define T\_MOD\_0 "%"

#define T\_MOD\_1 ""

#define T\_MOD\_2 ""

#define T\_MOD\_3 ""

//

#define T\_BIND\_RIGHT\_TO\_LEFT\_0 "<-"

#define T\_BIND\_RIGHT\_TO\_LEFT\_1 ""

#define T\_BIND\_RIGHT\_TO\_LEFT\_2 ""

#define T\_BIND\_RIGHT\_TO\_LEFT\_3 ""

#define T\_BIND\_LEFT\_TO\_RIGHT\_0 ""

#define T\_BIND\_LEFT\_TO\_RIGHT\_1 ""

#define T\_BIND\_LEFT\_TO\_RIGHT\_2 ""

#define T\_BIND\_LEFT\_TO\_RIGHT\_3 ""

//

#define T\_COMA\_0 ","

#define T\_COMA\_1 ""

#define T\_COMA\_2 ""

#define T\_COMA\_3 ""

#define T\_COLON\_0 ":"

#define T\_COLON\_1 ""

#define T\_COLON\_2 ""

#define T\_COLON\_3 ""

#define T\_GOTO\_0 "goto"

#define T\_GOTO\_1 ""

#define T\_GOTO\_2 ""

#define T\_GOTO\_3 ""

//

#define T\_IF\_0 "if"

#define T\_IF\_1 "("

#define T\_IF\_2 ""

#define T\_IF\_3 ""

#define T\_THEN\_0 ")"

#define T\_THEN\_1 ""

#define T\_THEN\_2 ""

#define T\_THEN\_3 ""

#define T\_ELSE\_0 "else"

#define T\_ELSE\_1 ""

#define T\_ELSE\_2 ""

#define T\_ELSE\_3 ""

//

#define T\_FOR\_0 "for"

#define T\_FOR\_1 ""

#define T\_FOR\_2 ""

#define T\_FOR\_3 ""

#define T\_TO\_0 "to"

#define T\_TO\_1 ""

#define T\_TO\_2 ""

#define T\_TO\_3 ""

#define T\_DOWNTO\_0 "downto"

#define T\_DOWNTO\_1 ""

#define T\_DOWNTO\_2 ""

#define T\_DOWNTO\_3 ""

#define T\_DO\_0 "do"

#define T\_DO\_1 ""

#define T\_DO\_2 ""

#define T\_DO\_3 ""

//

#define T\_WHILE\_0 "while"

#define T\_WHILE\_1 ""

#define T\_WHILE\_2 ""

#define T\_WHILE\_3 ""

#define T\_CONTINUE\_WHILE\_0 "continue"

#define T\_CONTINUE\_WHILE\_1 "while"

#define T\_CONTINUE\_WHILE\_2 ""

#define T\_CONTINUE\_WHILE\_3 ""

#define T\_EXIT\_WHILE\_0 "exit"

#define T\_EXIT\_WHILE\_1 "while"

#define T\_EXIT\_WHILE\_2 ""

#define T\_EXIT\_WHILE\_3 ""

#define T\_END\_WHILE\_0 "end"

#define T\_END\_WHILE\_1 "while"

#define T\_END\_WHILE\_2 ""

#define T\_END\_WHILE\_3 ""

//

#define T\_REPEAT\_0 "repeat"

#define T\_REPEAT\_1 ""

#define T\_REPEAT\_2 ""

#define T\_REPEAT\_3 ""

#define T\_UNTIL\_0 "until"

#define T\_UNTIL\_1 ""

#define T\_UNTIL\_2 ""

#define T\_UNTIL\_3 ""

//

#define T\_INPUT\_0 "get"

#define T\_INPUT\_1 ""

#define T\_INPUT\_2 ""

#define T\_INPUT\_3 ""

#define T\_OUTPUT\_0 "put"

#define T\_OUTPUT\_1 ""

#define T\_OUTPUT\_2 ""

#define T\_OUTPUT\_3 ""

//

#define T\_RLBIND\_0 "<-"

#define T\_RLBIND\_1 ""

#define T\_RLBIND\_2 ""

#define T\_RLBIND\_3 ""

#define T\_LRBIND\_0 ""

#define T\_LRBIND\_1 ""

#define T\_LRBIND\_2 ""

#define T\_LRBIND\_3 ""

//

#define T\_SEMICOLON\_0 ";"

#define T\_SEMICOLON\_1 ""

#define T\_SEMICOLON\_2 ""

#define T\_SEMICOLON\_3 ""

//

#define T\_BEGIN\_0 "startblok"

#define T\_BEGIN\_1 ""

#define T\_BEGIN\_2 ""

#define T\_BEGIN\_3 ""

#define T\_END\_0 "endblok"

#define T\_END\_1 ""

#define T\_END\_2 ""

#define T\_END\_3 ""

//

#define T\_NULL\_STATEMENT\_0 "NULL"

#define T\_NULL\_STATEMENT\_1 "STATEMENT"

#define T\_NULL\_STATEMENT\_2 ""

#define T\_NULL\_STATEMENT\_3 ""

#ifndef TOKEN\_STRUCT\_NAME\_

#define TOKEN\_STRUCT\_NAME\_

DECLENUM(TokenStructName,

MULTI\_TOKEN\_BITWISE\_NOT,

MULTI\_TOKEN\_BITWISE\_AND,

MULTI\_TOKEN\_BITWISE\_OR,

MULTI\_TOKEN\_NOT,

MULTI\_TOKEN\_AND,

MULTI\_TOKEN\_OR,

MULTI\_TOKEN\_EQUAL,

MULTI\_TOKEN\_NOT\_EQUAL,

MULTI\_TOKEN\_LESS,

MULTI\_TOKEN\_GREATER,

MULTI\_TOKEN\_LESS\_OR\_EQUAL,

MULTI\_TOKEN\_GREATER\_OR\_EQUAL,

MULTI\_TOKEN\_ADD,

MULTI\_TOKEN\_SUB,

MULTI\_TOKEN\_MUL,

MULTI\_TOKEN\_DIV,

MULTI\_TOKEN\_MOD,

MULTI\_TOKEN\_BIND\_RIGHT\_TO\_LEFT,

MULTI\_TOKEN\_BIND\_LEFT\_TO\_RIGHT,

MULTI\_TOKEN\_COLON,

MULTI\_TOKEN\_GOTO,

MULTI\_TOKEN\_IF,

// MULTI\_TOKEN\_IF\_, // don't change this!

MULTI\_TOKEN\_THEN,

// MULTI\_TOKEN\_THEN\_, // don't change this!

MULTI\_TOKEN\_ELSE,

MULTI\_TOKEN\_FOR,

MULTI\_TOKEN\_TO,

MULTI\_TOKEN\_DOWNTO,

MULTI\_TOKEN\_DO,

//

MULTI\_TOKEN\_WHILE,

/\*while special statement\*/MULTI\_TOKEN\_CONTINUE\_WHILE,

/\*while special statement\*/MULTI\_TOKEN\_EXIT\_WHILE,

MULTI\_TOKEN\_END\_WHILE,

//

//

MULTI\_TOKEN\_REPEAT,

MULTI\_TOKEN\_UNTIL,

//

//

MULTI\_TOKEN\_INPUT,

MULTI\_TOKEN\_OUTPUT,

//

//

MULTI\_TOKEN\_RLBIND,

MULTI\_TOKEN\_LRBIND,

//

MULTI\_TOKEN\_SEMICOLON,

MULTI\_TOKEN\_BEGIN,

MULTI\_TOKEN\_END,

//

MULTI\_TOKEN\_NULL\_STATEMENT

);

//#define PROCESS\_TOKENS(...) HANDLE\_TOKENS(\_\_VA\_ARGS\_\_)

//#define TOKENS\_FOR\_MULTI\_TOKEN(A, B, C, D) A, B, C, D

//#define TOKENS\_FOR\_MULTI\_TOKEN\_BITWISE\_NOT TOKENS\_FOR\_MULTI\_TOKEN("~", "", "", "")

#define INIT\_TOKEN\_STRUCT\_NAME() static void intitTokenStruct(){\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BITWISE\_NOT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BITWISE\_AND)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BITWISE\_OR)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, NOT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, AND)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, OR)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, EQUAL)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, NOT\_EQUAL)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, LESS)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, GREATER)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, LESS\_OR\_EQUAL)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, GREATER\_OR\_EQUAL)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, ADD)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, SUB)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, MUL)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, DIV)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, MOD)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BIND\_RIGHT\_TO\_LEFT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BIND\_LEFT\_TO\_RIGHT)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, COLON)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, GOTO)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, IF)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, THEN)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, ELSE)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, FOR)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, TO)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, DOWNTO)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, DO)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, WHILE)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, CONTINUE\_WHILE)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, EXIT\_WHILE)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, END\_WHILE)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, REPEAT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, UNTIL)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, INPUT)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, OUTPUT)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, RLBIND)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, LRBIND)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, SEMICOLON)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, BEGIN)\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, END)\

\

SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(tokenStruct, NULL\_STATEMENT)\

} char intitTokenStruct\_ = (intitTokenStruct(), 0);

#define MAX\_TOKEN\_STRUCT\_ELEMENT\_COUNT GET\_ENUM\_SIZE(TokenStructName)

#define MAX\_TOKEN\_STRUCT\_ELEMENT\_PART\_COUNT 4

#endif

extern char\* tokenStruct[MAX\_TOKEN\_STRUCT\_ELEMENT\_COUNT][MAX\_TOKEN\_STRUCT\_ELEMENT\_PART\_COUNT];

#define CONFIGURABLE\_GRAMMAR {\

{"labeled\_point", 2, {"ident", "tokenCOLON"}},\

{"goto\_label", 2, {"tokenGOTO","ident"}},\

{"program\_name", 1, {"ident\_terminal"}},\

{"value\_type", 1, {T\_DATA\_TYPE\_0}},\

{"other\_declaration\_ident", 2, {"tokenCOMMA", "ident"}},\

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"other\_declaration\_ident","other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},\

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"tokenCOMMA", "ident"}},\

{"value\_type\_\_ident", 2, {"value\_type", "ident"}},\

{"declaration", 2, {"value\_type\_\_ident", "other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},\

{"declaration", 2, {"value\_type", "ident"}},\

\

{"unary\_operator", 1, {T\_NOT\_0}},\

{"unary\_operator", 1, {T\_SUB\_0}},\

{"unary\_operator", 1, {T\_ADD\_0}},\

{"binary\_operator", 1, {T\_AND\_0}},\

{"binary\_operator", 1, {T\_OR\_0}},\

{"binary\_operator", 1, {T\_EQUAL\_0}},\

{"binary\_operator", 1, {T\_NOT\_EQUAL\_0}},\

{"binary\_operator", 1, {T\_LESS\_OR\_EQUAL\_0}},\

{"binary\_operator", 1, {T\_GREATER\_OR\_EQUAL\_0}},\

{"binary\_operator", 1, {T\_ADD\_0}},\

{"binary\_operator", 1, {T\_SUB\_0}},\

{"binary\_operator", 1, {T\_MUL\_0}},\

{"binary\_operator", 1, {T\_DIV\_0}},\

{"binary\_operator", 1, {T\_MOD\_0}},\

{"binary\_action", 2, {"binary\_operator","expression"}},\

\

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

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

{"left\_expression", 1, {"ident\_terminal"}},\

{"left\_expression", 1, {"value\_terminal"}},\

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action\_\_\_\_iteration\_after\_two"}},\

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action"}},\

{"expression", 2, {"left\_expression","binary\_action\_\_\_\_iteration\_after\_two"}},\

{"expression", 2, {"left\_expression","binary\_action"}},\

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

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

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

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

\

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

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

\

{"bind\_right\_to\_left", 2, {"ident","rl\_expression"}},\

{"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, {"tokenENDWHILE\_END","tokenENDWHILE\_WHILE"}},\

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

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

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

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

{"while\_cycle", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},\

\

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

{"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two", 2, {"tokenREPEAT","statement\_\_\_\_iteration\_after\_two"}},\

{"tokenREPEAT\_\_statement", 2, {"tokenREPEAT","statement"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

\

{"input\_\_first\_part", 2, {"tokenGET","tokenGROUPEXPRESSIONBEGIN"}},\

{"input\_\_second\_part", 2, {"ident","tokenGROUPEXPRESSIONEND"}},\

{"input", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

\

{"output\_\_first\_part", 2, {"tokenPUT","tokenGROUPEXPRESSIONBEGIN"}},\

{"output\_\_second\_part", 2, {"expression","tokenGROUPEXPRESSIONEND"}},\

{"output", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

\

{"statement", 2, {"ident","rl\_expression"}},\

{"statement", 2, {"lr\_expression","ident"}},\

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

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

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

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

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

{"statement", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},\

{"statement", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"ident","tokenCOLON"}},\

{"statement", 2, {"tokenGOTO","ident"}},\

{"statement", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

{"statement", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement\_\_\_\_iteration\_after\_two"}},\

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement"}},\

\

{ "statement\_in\_while\_body", 2, {"ident","rl\_expression"}},\

{ "statement\_in\_while\_body", 2, {"lr\_expression","ident"}},\

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

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

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

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

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

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

{ "statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{ "statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{ "statement\_in\_while\_body", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

{ "statement\_in\_while\_body", 2, {"ident","tokenCOLON"}},\

{ "statement\_in\_while\_body", 2, {"tokenGOTO","ident"}},\

{ "statement\_in\_while\_body", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

{ "statement\_in\_while\_body", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

{ "statement\_in\_while\_body", 2, {"tokenCONTINUE","tokenWHILE"}},\

{ "statement\_in\_while\_body", 2, {"tokenEXIT","tokenWHILE"}},\

{ "statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"}},\

{ "statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body"}},\

\

PROGRAM\_FORMAT\

\

{"tokenCOLON", 1, {T\_COLON\_0}},\

{"tokenGOTO", 1, {T\_GOTO\_0}},\

{"tokenINTEGER16", 1, {T\_DATA\_TYPE\_0}},\

{"tokenCOMMA", 1, {T\_COMA\_0}},\

{"tokenNOT", 1, {T\_NOT\_0}},\

{"tokenAND", 1, {T\_AND\_0}},\

{"tokenOR", 1, {T\_OR\_0}},\

{"tokenEQUAL", 1, {T\_EQUAL\_0}},\

{"tokenNOTEQUAL", 1, {T\_NOT\_EQUAL\_0}},\

{"tokenLESSOREQUAL", 1, {T\_LESS\_OR\_EQUAL\_0}},\

{"tokenGREATEROREQUAL", 1, {T\_GREATER\_OR\_EQUAL\_0}},\

{"tokenPLUS", 1, {T\_ADD\_0}},\

{"tokenMINUS", 1, {T\_SUB\_0}},\

{"tokenMUL", 1, {T\_MUL\_0}},\

{"tokenDIV", 1, {T\_DIV\_0}},\

{"tokenMOD", 1, {T\_MOD\_0}},\

{"tokenGROUPEXPRESSIONBEGIN", 1, {"("}},\

{"tokenGROUPEXPRESSIONEND", 1, {")"}},\

{"tokenRLBIND", 1, {T\_RLBIND\_0}},\

{"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}},\

{"tokenENDWHILE\_END", 1, {T\_END\_WHILE\_0}},\

{"tokenENDWHILE\_WHILE", 1, {T\_END\_WHILE\_1}},\

{"tokenREPEAT", 1, {T\_REPEAT\_0}},\

{"tokenUNTIL", 1, {T\_UNTIL\_0}},\

{"tokenGET", 1, {T\_INPUT\_0}},\

{"tokenPUT", 1, {T\_OUTPUT\_0}},\

{"tokenNAME", 1, {T\_NAME\_0}},\

{"tokenBODY", 1, {T\_BODY\_0}},\

{"tokenDATA", 1, {T\_DATA\_0}},\

{"tokenEND", 1, {T\_END\_0}},\

{"tokenSEMICOLON", 1, {T\_SEMICOLON\_0}},\

\

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

\

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

\

{"", 2, {"",""}}\

},\

178,\

"program"

#define ORIGINAL\_GRAMMAR {\

{"labeled\_point", 2, {"ident", "tokenCOLON"}},\

{"goto\_label", 2, {"tokenGOTO","ident"}},\

{"program\_name", 1, {"ident\_terminal"}},\

{"value\_type", 1, {"INTEGER16"}},\

{"other\_declaration\_ident", 2, {"tokenCOMMA", "ident"}},\

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"other\_declaration\_ident","other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},\

{"other\_declaration\_ident\_\_\_\_iteration\_after\_one", 2, {"tokenCOMMA", "ident"}},\

{"value\_type\_\_ident", 2, {"value\_type", "ident"}},\

{"declaration", 2, {"value\_type\_\_ident", "other\_declaration\_ident\_\_\_\_iteration\_after\_one"}},\

{"declaration", 2, {"value\_type", "ident"}},\

\

{"unary\_operator", 1, {"NOT"}},\

{"unary\_operator", 1, {"-"}},\

{"unary\_operator", 1, {"+"}},\

{"binary\_operator", 1, {"AND"}},\

{"binary\_operator", 1, {"OR"}},\

{"binary\_operator", 1, {"=="}},\

{"binary\_operator", 1, {"!="}},\

{"binary\_operator", 1, {"<="}},\

{"binary\_operator", 1, {">="}},\

{"binary\_operator", 1, {"+"}},\

{"binary\_operator", 1, {"-"}},\

{"binary\_operator", 1, {"\*"}},\

{"binary\_operator", 1, {"DIV"}},\

{"binary\_operator", 1, {"MOD"}},\

{"binary\_action", 2, {"binary\_operator","expression"}},\

\

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

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

{"left\_expression", 1, {"ident\_terminal"}},\

{"left\_expression", 1, {"value\_terminal"}},\

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action\_\_\_\_iteration\_after\_two"}},\

{"binary\_action\_\_\_\_iteration\_after\_two", 2, {"binary\_action","binary\_action"}},\

{"expression", 2, {"left\_expression","binary\_action\_\_\_\_iteration\_after\_two"}},\

{"expression", 2, {"left\_expression","binary\_action"}},\

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

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

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

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

\

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

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

\

{"bind\_right\_to\_left", 2, {"ident","rl\_expression"}},\

{"bind\_left\_to\_right", 2, {"lr\_expression","ident"}},\

\

{"body\_for\_true", 2, {"statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenSEMICOLON"}},\

{"body\_for\_true", 2, {"statement\_in\_while\_body","tokenSEMICOLON"}},\

{"body\_for\_true", 1, {";"}},\

{"tokenELSE\_\_statement\_in\_while\_body", 2, {"tokenELSE","statement\_in\_while\_body"}},\

{"tokenELSE\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"tokenELSE","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"}},\

{"body\_for\_false", 2, {"tokenELSE\_\_statement\_in\_while\_body\_\_\_\_iteration\_after\_two","tokenSEMICOLON"}},\

{"body\_for\_false", 2, {"tokenELSE\_\_statement\_in\_while\_body","tokenSEMICOLON"}},\

{"body\_for\_false", 2, {"tokenELSE","tokenSEMICOLON"}},\

{"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN", 2, {"tokenIF","tokenGROUPEXPRESSIONBEGIN"}},\

{"expression\_\_tokenGROUPEXPRESSIONEND", 2, {"expression","tokenGROUPEXPRESSIONEND"}},\

{"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN","expression\_\_tokenGROUPEXPRESSIONEND"}},\

{"body\_for\_true\_\_body\_for\_false", 2, {"body\_for\_true","body\_for\_false"}},\

{"cond\_block", 2, {"tokenIF\_\_tokenGROUPEXPRESSIONBEGIN\_\_expression\_\_tokenGROUPEXPRESSIONEND","body\_for\_true\_\_body\_for\_false"}},\

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

\

{"cycle\_counter", 1, {"ident\_terminal"}},\

{"rl\_expression", 2, {"tokenRLBIND","expression"}},\

{"lr\_expression", 2, {"expression","tokenLRBIND"}},\

{"cycle\_counter\_init", 2, {"cycle\_counter","rl\_expression"}},\

{"cycle\_counter\_init", 2, {"lr\_expression","cycle\_counter"}},\

{"cycle\_counter\_last\_value", 1, {"value\_terminal"}},\

{"cycle\_body", 2, {"tokenDO","statement\_\_\_\_iteration\_after\_two"}},\

{"cycle\_body", 2, {"tokenDO","statement"}},\

{"tokenFOR\_\_cycle\_counter\_init", 2, {"tokenFOR","cycle\_counter\_init"}},\

{"tokenTO\_\_cycle\_counter\_last\_value", 2, {"tokenTO","cycle\_counter\_last\_value"}},\

{"tokenFOR\_\_cycle\_counter\_init\_\_tokenTO\_\_cycle\_counter\_last\_value", 2, {"tokenFOR\_\_cycle\_counter\_init","tokenTO\_\_cycle\_counter\_last\_value"}},\

{"cycle\_body\_\_tokenSEMICOLON", 2, {"cycle\_body","tokenSEMICOLON"}},\

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

\

{"continue\_while", 2, {"tokenCONTINUE","tokenWHILE"}},\

{"exit\_while", 2, {"tokenEXIT","tokenWHILE"}},\

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

{"tokenEND\_\_tokenWHILE", 2, {"tokenEND","tokenWHILE"}},\

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

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

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

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

{"while\_cycle", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},\

\

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

{"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two", 2, {"tokenREPEAT","statement\_\_\_\_iteration\_after\_two"}},\

{"tokenREPEAT\_\_statement", 2, {"tokenREPEAT","statement"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"repeat\_until\_cycle", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

\

{"input\_\_first\_part", 2, {"tokenGET","tokenGROUPEXPRESSIONBEGIN"}},\

{"input\_\_second\_part", 2, {"ident","tokenGROUPEXPRESSIONEND"}},\

{"input", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

\

{"output\_\_first\_part", 2, {"tokenPUT","tokenGROUPEXPRESSIONBEGIN"}},\

{"output\_\_second\_part", 2, {"expression","tokenGROUPEXPRESSIONEND"}},\

{"output", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

\

{"statement", 2, {"ident","rl\_expression"}},\

{"statement", 2, {"lr\_expression","ident"}},\

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

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

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

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

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

{"statement", 2, {"tokenWHILE\_\_expression","tokenEND\_\_tokenWHILE"}},\

{"statement", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

{"statement", 2, {"ident","tokenCOLON"}},\

{"statement", 2, {"tokenGOTO","ident"}},\

{"statement", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

{"statement", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement\_\_\_\_iteration\_after\_two"}},\

{"statement\_\_\_\_iteration\_after\_two", 2, {"statement","statement"}},\

\

{"statement\_in\_while\_body", 2, {"ident","rl\_expression"}},\

{"statement\_in\_while\_body", 2, {"lr\_expression","ident"}},\

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

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

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

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

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

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

{"statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement\_\_\_\_iteration\_after\_two","tokenUNTIL\_\_expression"}},\

{"statement\_in\_while\_body", 2, {"tokenREPEAT\_\_statement","tokenUNTIL\_\_expression"}},\

{"statement\_in\_while\_body", 2, {"tokenREPEAT","tokenUNTIL\_\_expression"}},\

{"statement\_in\_while\_body", 2, {"ident","tokenCOLON"}},\

{"statement\_in\_while\_body", 2, {"tokenGOTO","ident"}},\

{"statement\_in\_while\_body", 2, {"input\_\_first\_part","input\_\_second\_part"}},\

{"statement\_in\_while\_body", 2, {"output\_\_first\_part","output\_\_second\_part"}},\

{"statement\_in\_while\_body", 2, {"tokenCONTINUE","tokenWHILE"}},\

{"statement\_in\_while\_body", 2, {"tokenEXIT","tokenWHILE"}},\

{"statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body\_\_\_\_iteration\_after\_two"}},\

{"statement\_in\_while\_body\_\_\_\_iteration\_after\_two", 2, {"statement\_in\_while\_body","statement\_in\_while\_body"}},\

\

{"tokenNAME\_\_program\_name", 2, {"tokenNAME","program\_name"}},\

{"tokenSEMICOLON\_\_tokenBODY", 2, {"tokenSEMICOLON","tokenBODY"}},\

{"tokenDATA\_\_declaration", 2, {"tokenDATA","declaration"}},\

{"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY", 2, {"tokenNAME\_\_program\_name","tokenSEMICOLON\_\_tokenBODY"}},\

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA\_\_declaration"}},\

{"program\_\_\_\_part1", 2, {"tokenNAME\_\_program\_name\_\_tokenSEMICOLON\_\_tokenBODY","tokenDATA"}},\

{"statement\_\_tokenEND", 2, {"statement","tokenEND"}},\

{"statement\_\_\_\_iteration\_after\_two\_\_tokenEND", 2, {"statement\_\_\_\_iteration\_after\_two","tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_\_\_iteration\_after\_two\_\_tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","statement\_\_tokenEND"}},\

{"program\_\_\_\_part2", 2, {"tokenSEMICOLON","tokenEND"}},\

{"program", 2, {"program\_\_\_\_part1","program\_\_\_\_part2"}},\

\

{"tokenCOLON", 1, {":"}},\

{"tokenGOTO", 1, {"GOTO"}},\

{"tokenINTEGER16", 1, {"INTEGER16"}},\

{"tokenCOMMA", 1, {","}},\

{"tokenNOT", 1, {"NOT"}},\

{"tokenAND", 1, {"AND"}},\

{"tokenOR", 1, {"OR"}},\

{"tokenEQUAL", 1, {"=="}},\

{"tokenNOTEQUAL", 1, {"!="}},\

{"tokenLESSOREQUAL", 1, {"<="}},\

{"tokenGREATEROREQUAL", 1, {">="}},\

{"tokenPLUS", 1, {"+"}},\

{"tokenMINUS", 1, {"-"}},\

{"tokenMUL", 1, {"\*"}},\

{"tokenDIV", 1, {"DIV"}},\

{"tokenMOD", 1, {"MOD"}},\

{"tokenGROUPEXPRESSIONBEGIN", 1, {"("}},\

{"tokenGROUPEXPRESSIONEND", 1, {")"}},\

{"tokenRLBIND", 1, {"<<"}},\

{"tokenLRBIND", 1, {">>"}},\

{"tokenELSE", 1, {"ELSE"}},\

{"tokenIF", 1, {"IF"}},\

{"tokenDO", 1, {"DO"}},\

{"tokenFOR", 1, {"FOR"}},\

{"tokenTO", 1, {"TO"}},\

{"tokenWHILE", 1, {"WHILE"}},\

{"tokenCONTINUE", 1, {"CONTINUE"}},\

{"tokenEXIT", 1, {"EXIT"}},\

{"tokenREPEAT", 1, {"REPEAT"}},\

{"tokenUNTIL", 1, {"UNTIL"}},\

{"tokenGET", 1, {"GET"}},\

{"tokenPUT", 1, {"PUT"}},\

{"tokenNAME", 1, {"NAME"}},\

{"tokenBODY", 1, {"BODY"}},\

{"tokenDATA", 1, {"DATA"}},\

{"tokenEND", 1, {"END"}},\

{"tokenSEMICOLON", 1, {";"}},\

\

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

\

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

\

{"", 2, {"",""}}\

\

},\

176,\

"program"

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

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

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

#define DEFAULT\_MODE (DEBUG\_MODE | LEXICAL\_ANALYZE\_MODE | SYNTAX\_ANALYZE\_MODE | SEMANTIX\_ANALYZE\_MODE | MAKE\_ASSEMBLY | MAKE\_BINARY)  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: def.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define SUCCESS\_STATE 0

#define LEXICAL\_ANALYZE\_MODE 1 // lexicalAnalyze

#define MAKE\_LEXEMES\_SEQUENSE 2 // ADD MODE

#define SYNTAX\_ANALYZE\_MODE 4

#define MAKE\_AST 8 // ADD MODE

#define SEMANTIX\_ANALYZE\_MODE 16 // ADD MODE

#define MAKE\_PREPARE 32 // ADD MODE

#define MAKE\_C 64 // ADD MODE

#define MAKE\_ASSEMBLY 128 // ADD MODE

#define MAKE\_OBJECT 256 // ADD MODE

#define MAKE\_BINARY 512 // ADD MODE

#define RUN\_BINARY 1024 // ADD MODE

#define UNDEFINED\_MODE 16384

#define INTERACTIVE\_MODE 32768

#define SKIP\_INTERACTIVE\_IN\_INTERACTIVE\_MODE 65536

#define FULL\_COMPILER\_MODE 2048 // ?

#define DEBUG\_MODE 4096

//#define DECLENUM(NAME, ...) typedef enum {\_\_VA\_ARGS\_\_, size##NAME} NAME;

#define DECLENUM(NAME, ...) enum NAME {\_\_VA\_ARGS\_\_, size##NAME};

#define GET\_ENUM\_SIZE(NAME) size##NAME

#define SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(ARRAY, NAME)\

ARRAY[MULTI\_TOKEN\_##NAME][0] = (char\*)T\_##NAME##\_0;\

ARRAY[MULTI\_TOKEN\_##NAME][1] = (char\*)T\_##NAME##\_1;\

ARRAY[MULTI\_TOKEN\_##NAME][2] = (char\*)T\_##NAME##\_2;\

ARRAY[MULTI\_TOKEN\_##NAME][3] = (char\*)T\_##NAME##\_3;

//#define EXPAND\_MACRO(...) \_\_VA\_ARGS\_\_ // Проміжний макрос для розгортання

//

//#define SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY\_(ARRAY, QUADRUPLE\_ELEMENT\_INDEX, ...)\

//SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY(ARRAY, QUADRUPLE\_ELEMENT\_INDEX, EXPAND\_MACRO(\_\_VA\_ARGS\_\_) )

//

//#define TOKENS\_FOR\_MULTI\_TOKEN\_BITWISE\_NOT "~", "", "", ""

//

//SET\_QUADRUPLE\_STR\_MACRO\_IN\_ARRAY\_(tokenStruct, MULTI\_TOKEN\_BITWISE\_NOT, TOKENS\_FOR\_MULTI\_TOKEN\_BITWISE\_NOT)

//#define MAX\_TEXT\_SIZE 8192

//#define MAX\_WORD\_COUNT (MAX\_TEXT\_SIZE / 5)

//#define MAX\_LEXEM\_SIZE 1024

//#define MAX\_VARIABLES\_COUNT 256

//#define MAX\_KEYWORD\_COUNT 64

//

//#define KEYWORD\_LEXEME\_TYPE 1

//#define IDENTIFIER\_LEXEME\_TYPE 2 // #define LABEL\_LEXEME\_TYPE 8

//#define VALUE\_LEXEME\_TYPE 4

//#define UNEXPEXTED\_LEXEME\_TYPE 127

//

//#define LEXICAL\_ANALISIS\_MODE 1

//#define SEMANTIC\_ANALISIS\_MODE 2

//#define FULL\_COMPILER\_MODE 4

//

//#define DEBUG\_MODE 512

//

//+!///#define MAX\_PARAMETERS\_SIZE 4096

//+!///#define PARAMETERS\_COUNT 4

//+!///#define INPUT\_FILENAME\_PARAMETER 0

//

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

#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: div.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define DIV\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeDivCode(B, C, M);

unsigned char\* makeDivCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: else.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define ELSE\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeElseCode(B, C, M);\

if (A ==\* B) C = makeSemicolonAfterElseCode(B, C, M);

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

unsigned char\* makeSemicolonAfterElseCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: equal.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define EQUAL\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeIsEqualCode(B, C, M);

unsigned char\* makeIsEqualCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: for.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define FOR\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeForCycleCode(B, C, M);\

if (A ==\* B) C = makeToOrDowntoCycleCode(B, C, M);\

if (A ==\* B) C = makeDoCycleCode(B, C, M);\

if (A ==\* B) C = makeSemicolonAfterForCycleCode(B, C, M);

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

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

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

unsigned char\* makeSemicolonAfterForCycleCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: generator.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

// TODO: CHANGE BY fRESET() TO END

#define C\_CODER\_MODE 0x01

#define ASSEMBLY\_X86\_WIN32\_CODER\_MODE 0x02

#define MACHINE\_X86\_WIN32\_CODER\_MODE 0x04

extern unsigned char generatorMode;

#define CODEGEN\_DATA\_TYPE int

#define START\_DATA\_OFFSET 512

#define OUT\_DATA\_OFFSET (START\_DATA\_OFFSET + 512)

#define M1 1024

#define M2 1024

//unsigned long long int dataOffsetMinusCodeOffset = 0x00003000;

#define dataOffsetMinusCodeOffset 0x00004000ull

//unsigned long long int codeOffset = 0x000004AF;

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

#define baseOperationObjectOffset 0x0000018Bull

#define baseOperationOffset 0x000004AFull

#define putProcOffset 0x0000001Bull

#define getProcOffset 0x00000044ull

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

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

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

#define MAX\_ACCESSORY\_STACK\_SIZE 128

extern struct NonContainedLexemInfo lexemInfoTransformationTempStack[MAX\_ACCESSORY\_STACK\_SIZE];

extern unsigned long long int lexemInfoTransformationTempStackSize;

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

#if 1

unsigned char\* getObjectCodeBytePtr(unsigned char\* baseBytePtr, unsigned char generatorMode);

unsigned char\* getImageCodeBytePtr(unsigned char\* baseBytePtr, unsigned char generatorMode);

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

void viewCode(unsigned char\* outCodePtr, unsigned long long int outCodePrintSize, unsigned char align);

#endif

unsigned long long int buildTemplateForCodeObject(unsigned char\* byteImage);

unsigned long long int buildTemplateForCodeImage(unsigned char\* byteImage);

void writeBytesToFile(const char\* output\_file, unsigned char\* byteImage, unsigned long long int imageSize);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: goto\_lable.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <string>

#include <map>

//#include <utility>

#include <stack>

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

#define LABEL\_GOTO\_LABELE\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeLabelCode(B, C, M);\

if (A ==\* B) C = makeGotoLabelCode(B, C, M);

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

unsigned char\* makeGotoLabelCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: greater\_or\_equal.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define GREATER\_OR\_EQUAL\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeIsGreaterOrEqualCode(B, C, M);

unsigned char\* makeIsGreaterOrEqualCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: if\_then.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define IF\_THEN\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeIfCode(B, C, M);\

if (A ==\* B) C = makeThenCode(B, C, M);\

if (A ==\* B) C = makeSemicolonAfterThenCode(B, C, M);

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

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

unsigned char\* makeSemicolonAfterThenCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: input.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define INPUT\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeGetCode(B, C, M);

unsigned char\* makeGetCode(struct LexemInfo\*\* lastLexemInfoInTable, unsigned char\* currBytePtr, unsigned char generatorMode);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* 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);  
  
#define \_CRT\_SECURE\_NO\_WARNINGS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* N.Kozak // Lviv'2024-2025 // cw\_sp2\_\_2024\_2025 \*

\* file: less\_or\_equal.h \*

\* (draft!) \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define LESS\_OR\_EQUAL\_CODER(A, B, C, M, R)\

if (A ==\* B) C = makeIsLessOrEqualCode(B, C, M);

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

**Додаток Д.** Дерево граматичного розбору

