

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий
Кафедра Программной инженерии
Специальность 1-40 01 01 Программное обеспечение информационных технологий
Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора GKV-2022»

Выполнил студент Гвоздовский Кирилл Владимирович
(Ф.И.О.)
Руководитель проекта асс. Мущук Артур Николаевич
(учен. степень, звание, должность, подпись, Ф.И.О.)
Заведующий кафедрой к.т.н. доц. Пацей Наталья Владимировна
(учен. степень, звание, должность, подпись, Ф.И.О.)
Консультант асс. Мущук Артур Николаевич
(учен. степень, звание, должность, подпись, Ф.И.О.)
Нормоконтролер асс. Мущук Артур Николаевич
(учен. степень, звание, должность, подпись, Ф.И.О.)
Курсовой проект защищен с оценкой _____

Минск 2022

Оглавление

| | |
|---|----|
| Введение | 4 |
| 1 Спецификация языка программирования..... | 5 |
| 1.1 Характеристика языка программирования..... | 5 |
| 1.2 Определение алфавита языка программирования..... | 5 |
| 1.3 Применяемые сепараторы..... | 6 |
| 1.4 Применяемые кодировки | 6 |
| 1.5 Типы данных | 6 |
| 1.6 Преобразование типов данных | 7 |
| 1.7 Идентификаторы | 7 |
| 1.8 Литералы..... | 7 |
| 1.9 Объявление данных | 8 |
| 1.10 Инициализация данных..... | 8 |
| 1.11 Инструкции языка..... | 8 |
| 1.12 Операции языка..... | 9 |
| 1.13 Выражения и их вычисление | 10 |
| 1.14 Конструкции языка | 10 |
| 1.15 Области видимости идентификаторов..... | 11 |
| 1.16 Семантические проверки | 11 |
| 1.17 Распределение оперативной памяти на этапе выполнения | 11 |
| 1.18 Стандартная библиотека и её состав | 11 |
| 1.19 Ввод и вывод данных | 12 |
| 1.20 Точка входа..... | 12 |
| 1.21 Препроцессор | 12 |
| 1.22 Соглашения о вызовах..... | 12 |
| 1.23 Объектный код | 13 |
| 1.24 Классификация сообщений транслятора..... | 13 |
| 1.25 Контрольный пример..... | 13 |
| 2 Структура транслятора..... | 14 |
| 2.1 Компоненты транслятора, их назначение и принципы взаимодействия | 14 |
| 2.2 Перечень входных параметров транслятора | 15 |
| 2.3 Перечень протоколов, формируемых транслятором и их содержимое | 15 |
| 3 Разработка лексического анализатора | 17 |
| 3.1 Структура лексического анализатора..... | 17 |
| 3.2. Контроль входных символов | 18 |
| 3.3 Удаление избыточных символов..... | 18 |
| 3.4 Перечень ключевых слов | 18 |
| 3.5 Основные структуры данных | 20 |
| 3.6 Принцип обработки ошибок..... | 21 |
| 3.7 Структура и перечень сообщений лексического анализатора | 22 |
| 3.8 Параметры лексического анализатора..... | 22 |
| 3.9 Алгоритм лексического анализа..... | 22 |
| | 22 |

| | |
|--|----|
| 3.10 Контрольный пример..... | 22 |
| 4. Разработка синтаксического анализатора | 23 |
| 4.1 Структура синтаксического анализатора | 23 |
| 4.2 Контекстно-свободная грамматика, описывающая синтаксис языка | 23 |
| 4.3 Построение конечного магазинного автомата..... | 25 |
| 4.4 Основные структуры данных | 26 |
| 4.5 Описание алгоритма синтаксического разбора | 26 |
| 4.6 Структура и перечень сообщений синтаксического анализатора | 27 |
| 4.7. Параметры синтаксического анализатора и режимы его работы | 27 |
| 4.8. Принцип обработки ошибок | 27 |
| 4.9. Контрольный пример..... | 27 |
| 5 Разработка семантического анализатора..... | 28 |
| 5.1 Структура семантического анализатора..... | 28 |
| 5.2 Функции семантического анализатора..... | 28 |
| 5.3 Структура и перечень сообщений семантического анализатора..... | 28 |
| 5.4 Принцип обработки ошибок | 29 |
| 5.5 Контрольный пример..... | 29 |
| 6. Вычисление выражений | 30 |
| 6.1 Выражения, допускаемые языком..... | 30 |
| 6.2 Польская запись и принцип её построения | 30 |
| 6.3 Программная реализация обработки выражений | 30 |
| 6.4 Контрольный пример..... | 31 |
| 7. Генерация кода | 32 |
| 7.1 Структура генератора кода | 32 |
| 7.2 Представление типов данных в оперативной памяти | 32 |
| 7.3 Статическая библиотека..... | 33 |
| 7.4 Особенности алгоритма генерации кода | 33 |
| 7.5 Входные параметры генератора кода | 33 |
| 7.6 Контрольный пример..... | 33 |
| 8. Тестирование транслятора | 35 |
| 8.1 Общие положения..... | 35 |
| 8.2 Результаты тестирования | 35 |
| Заключение | 38 |
| Список использованных источников..... | 39 |
| Приложение А | 40 |
| Приложение Б..... | 41 |
| Приложение В | 42 |
| Приложение Г | 50 |
| Приложение Д | 52 |

Введение

В данном курсовом проекте поставлена задача разработки собственного языка программирования и транслятора для него. Название языка – GKV-2022. Написание транслятора будет осуществляться на языке C++, при этом код на языке GKV-2022 будет транслироваться в язык ассемблера.

Задание на курсовой проект можно разделить на следующие задачи:

1. Разработка спецификации языка GKV-2022;
2. Разработка лексического анализатора;
3. Разработка синтаксического анализатора;
4. Разработка семантического анализатора;
5. Разбор арифметических выражений;
6. Разработка генератора кода;
7. Тестирование транслятора.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования GKV-2022 является процедурным, универсальным строго типизированным, не объектно-ориентированным, компилируемым.

1.2 Определение алфавита языка программирования

При написании программы на языке GKV-2022 используется таблица символов Windows-1251, представленная в рис.1.1.

| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|----|---------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|-------------------|--------------------|--------------------|-------------------|-------------------|-------------------|-------------------|
| 00 | <u>NUL</u> 0000 | <u>STX</u> 0001 | <u>SOT</u> 0002 | <u>ETX</u> 0003 | <u>EOT</u> 0004 | <u>ENQ</u> 0005 | <u>ACK</u> 0006 | <u>BEL</u> 0007 | <u>BS</u> 0008 | <u>HT</u> 0009 | <u>LF</u> 000A | <u>VT</u> 000B | <u>FF</u> 000C | <u>CR</u> 000D | <u>SO</u> 000E | <u>SI</u> 000F |
| 10 | <u>DLE</u> 0010 | <u>DC1</u> 0011 | <u>DC2</u> 0012 | <u>DC3</u> 0013 | <u>DC4</u> 0014 | <u>NAK</u> 0015 | <u>SYN</u> 0016 | <u>ETB</u> 0017 | <u>CAN</u> 0018 | <u>EM</u> 0019 | <u>SUB</u> 001A | <u>ESC</u> 001B | <u>FS</u> 001C | <u>GS</u> 001D | <u>RS</u> 001E | <u>US</u> 001F |
| 20 | <u>SP</u> 0020 | ! | " | # | \$ | % | & | ' | (|) | * | + | , | - | . | / |
| 30 | 0 0030 | 1 0031 | 2 0032 | 3 0033 | 4 0034 | 5 0035 | 6 0036 | 7 0037 | 8 0038 | 9 0039 | : | ; | < | = | > | ? |
| 40 | @ 0040 | A 0041 | B 0042 | C 0043 | D 0044 | E 0045 | F 0046 | G 0047 | H 0048 | I 0049 | J 004A | K 004B | L 004C | M 004D | N 004E | O 004F |
| 50 | P 0050 | Q 0051 | R 0052 | S 0053 | T 0054 | U 0055 | V 0056 | W 0057 | X 0058 | Y 0059 | Z 005A | [005B | \ 005C |] 005D | ^ 005E | _ 005F |
| 60 | ` 0060 | a 0061 | b 0062 | c 0063 | d 0064 | e 0065 | f 0066 | g 0067 | h 0068 | i 0069 | j 006A | k 006B | l 006C | m 006D | n 006E | o 006F |
| 70 | p 0070 | q 0071 | r 0072 | s 0073 | t 0074 | u 0075 | v 0076 | w 0077 | x 0078 | y 0079 | z 007A | { 007B | 007C | } 007D | ~ 007E | DEL 007F |
| 80 | Ъ 0402 | Ѓ 0403 | Ѕ 201A | Ї 0453 | Љ 201E | Њ 2026 | Ћ 2020 | Ќ 2021 | Є 20AC | Ў 2030 | Љ 0409 | < 2039 | Њ 040A | Ѓ 040C | Ѕ 040B | Ї 040F |
| 90 | ђ 0452 | ѵ 2018 | Ѷ 2019 | ѷ 201C | Ѹ 201D | • 2022 | — 2013 | — 2014 | ░ 2122 | ▒ 2122 | Љ 0459 | > 203A | Њ 045A | Ѓ 045C | Ѕ 045B | Ї 045F |
| A0 | <u>NBSP</u> 00A0 | Ў 040E | Ѷ 045E | Ј 0408 | • 00A4 | Ѓ 0490 | Ѕ 00A6 | Ї 00A7 | Љ 0401 | Є 00A9 | Ў 0404 | Љ 00AB | < 00AC | Њ 00AD | Ѓ 00AE | Ѕ 0407 |
| B0 | ° 00B0 | ± 00B1 | І 0406 | і 0456 | ґ 0491 | µ 00B5 | ¶ 00B6 | · 00B7 | ё 0451 | № 2116 | е 0454 | » 00BB | ј 0458 | ѕ 0405 | ѕ 0455 | ї 0457 |
| C0 | А 0410 | Б 0411 | В 0412 | Г 0413 | Д 0414 | Е 0415 | Ж 0416 | З 0417 | И 0418 | Й 0419 | К 041A | Л 041B | М 041C | Н 041D | О 041E | П 041F |
| D0 | Р 0420 | С 0421 | Т 0422 | У 0423 | Ф 0424 | Х 0425 | Ц 0426 | Ч 0427 | Ш 0428 | Щ 0429 | Ъ 042A | Ы 042B | Ь 042C | Э 042D | Ю 042E | Я 042F |
| E0 | а 0430 | б 0431 | в 0432 | г 0433 | д 0434 | е 0435 | ж 0436 | з 0437 | и 0438 | й 0439 | к 043A | л 043B | м 043C | н 043D | о 043E | п 043F |
| F0 | р 0440 | с 0441 | т 0442 | у 0443 | ф 0444 | х 0445 | ц 0446 | ч 0447 | ш 0448 | щ 0449 | ъ 044A | ы 044B | ь 044C | э 044D | ю 044E | я 044F |

Рисунок 1.1 Алфавит входных символов

Исходный код GKV-2022 может содержать символы латинского алфавита малого регистра, цифры десятичной системы счисления от 0 до 9, символы кириллицы разрешены только в строковых литералах.

1.3 Применяемые сепараторы

Символы-сепараторы служат в качестве разделителей цепочек языка во время обработки исходного текста программы с целью разделения на токены. Они представлены в таблице 1.1.

Таблица 1.1. Сепараторы

| Символ(ы) | Назначение |
|-----------|---|
| ‘пробел’ | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |

Продолжение таблицы 1.1

| Символ(ы) | Назначение |
|-----------|---|
| { ... } | Блок функции или условной конструкции |
| (...) | Блок фактических или формальных параметров функции, а также приоритет арифметических операций |
| , | Разделитель параметров функций |
| + - * / % | Арифметические операции |
| ; | Разделитель программных конструкций |
| = | Оператор присваивания |

1.4 Применяемые кодировки

Для написания программ язык GKV-2022 использует кодировку Windows-1251, содержащую как латинские символы, так и кириллицу, а также некоторые специальные символы, такие как [] () , ; : # + - / * > < & ! { }.

1.5 Типы данных

В языке GKV-2022 реализованы два фундаментальных типа данных: целочисленный и строковый. Описание типов приведено в таблице 1.2.

Таблица 1.2 Типы данных языка GKV-2022

| Тип данных | Характеристика |
|---|---|
| Целочисленный тип данных integer | Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 4 байта. Инициализация по умолчанию: значение 0. Поддерживаемые операции: + (бинарный) – оператор сложения; - (бинарный) – оператор вычитания; * (бинарный) – оператор умножения; / (бинарный) – оператор деления; % (бинарный) – оператор определения остатка от деления; |

| | |
|--|---|
| | = (бинарный) – оператор присваивания. В качестве условия условного оператора поддерживаются следующие логические операции: > (бинарный) – оператор «больше»; < (бинарный) – оператор «меньше»; |
|--|---|

Продолжение таблицы 1.2

| Тип данных | Характеристика |
|--|---|
| Строковый тип данных stroka | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 255. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций. |

1.6 Преобразование типов данных

Преобразование типов данных в языке GKV-2022 не предусмотрено.

1.7 Идентификаторы

Общее количество идентификаторов ограничено максимальным размером таблицы идентификаторов. Идентификаторы должны содержать только символы нижнего регистра латинского алфавита. Максимальная длина идентификатора равна пяти символам. Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Префикс занимает 5 дополнительных символов. В случае превышения заданной длины, идентификаторы усекаются до длины, равной 10 символов (5 символов на имя идентификатора, 5 символов на префикс). Данные правила действуют для всех типов идентификаторов. Зарезервированные идентификаторы не предусмотрены. Идентификаторы не должны совпадать с ключевыми словами. Типы идентификаторов: имя переменной, имя функции, параметр функции. Имена идентификаторов-функций и имена идентификаторов-переменных не должны совпадать с именами команд ассемблера.

1.8 Литералы

С помощью литералов осуществляется инициализация переменных. Все литералы являются **rvalue**. Типы литералов языка GKV-2022 представлены в таблице 1.3.

Таблица 1.3 Литералы

| Литералы | Пояснение |
|----------|-----------|
|----------|-----------|

| | |
|--|---|
| Целочисленные литералы в десятичном представлении | Может состоять из чисел [0-9]. Минимальное значение равно -2147483647, максимальное 2147483647. При выходе за пределы значение будет укорочено до максимального значения. |
| Строковые литералы | Набор символов (от 1 до 255), заключённых в двойные кавычки |
| Целочисленные литералы в шестнадцатеричном представлении | Может состоять из чисел [0-9], а также символов [A-F]. Начинается с суффикса 16х. Минимальное значение равно -16х7FFFFFFF, максимальное 16х7FFFFFFF. Не содержит дробную часть. При выходе за пределы значение будет укорочено до максимального значения. |

Ограничения на строковые литералы языка GKV-2022: внутри литерала не допускается использование одинарных и двойных кавычек.

1.9 Объявление данных

Для объявления переменной указывается тип данных и имя идентификатора. Допускается инициализация при объявлении.

Пример объявления числового типа с инициализацией:

integer num1 = -1

integer num2 = 16хFFF

Пример объявления переменной символьного типа с инициализацией:

stroka str1= "hello world"

Для объявления функций и процедур используется ключевое слово **function**, перед которым указывается тип функции. Далее обязателен список параметров и тело функции.

1.10 Инициализация данных

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. Объектами-инициализаторами могут быть только идентификаторы или литералы. При объявлении без инициализации предусмотрено значение 0 для типа **integer** по умолчанию.

1.11 Инструкции языка

Инструкции языка GKV-2022 представлены в таблице 1.4.

Таблица 1.4 Инструкции языка GKV-2022

| Инструкция | Реализация |
|------------|------------|
|------------|------------|

| | |
|--|--|
| Объявление переменной | <тип данных> <идентификатор>; |
| Инструкция | Реализация |
| Объявление переменной с явной инициализацией | <тип данных> <идентификатор> = <значение>; Значение – инициализатор конкретного типа. Может быть только литералом или идентификатором |
| Возврат из функции или процедуры | Для функций, возвращающих значение: return <идентификатор/литерал>; Для процедур: return ; |
| Вывод данных | output <идентификатор/литерал>; |
| Вызов функции или процедуры | <идентификатор функции> (<список параметров>); Список параметров может быть пустым. |
| Присваивание | <идентификатор> = <выражение>; Выражением может быть идентификатор, литерал, или вызов функции соответствующего типа. Для целочисленного типа выражение может быть дополнено арифметическими операциями с любым количеством операндов с использованием скобок. Для строкового типа выражение может быть только идентификатором, литералом или вызовом функции, возвращающей значение строкового типа. |

1.12 Операции языка

В языке GKV-2022 предусмотрены следующие операции с данными. Приоритетность операции умножения выше приоритета операций сложения и вычитания. Для установки наивысшего приоритета используются круглые скобки. Операции языка представлены в таблице 1.5.

Таблица 1.5 Операции языка GKV-2022

| Тип оператора | Оператор |
|----------------|---|
| Арифметические | 1. + – сложение 2. - – вычитание 3. * – умножение 4. / – деление без остатка 5. % – остаток от деления 6. = – присваивание |
| Строковые | 1. = – присваивание |
| Логические | 1. > – больше 2. < – меньше |

1.13 Выражения и их вычисление

Вычисление выражений – одна из важнейших задач языков программирования. Всякое выражение составляется согласно следующим правилам:

1. Допускается использовать скобки для смены приоритета операций;
2. Выражение записывается в строку без переносов;
3. Использование двух подряд идущих операторов не допускается;
4. Допускается использовать в выражении вызов функции, вычисляющей и возвращающей целочисленное значение.

Перед генерацией кода каждое выражение приводится к записи в польской записи для удобства дальнейшего вычисления выражения на языке ассемблера.

1.14 Конструкции языка

Программа на языке GKV-2022 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода.

Программные конструкции языка представлены в таблице 1.6.

Таблица 1.6 Программные конструкции языка GKV-2022

| Конструкция | Реализация |
|----------------------|---|
| Главная функция | main {... return ; } |
| Внешняя функция | <тип данных> function <идентификатор> (<тип> <идентификатор>, ...) {... return <идентификатор/литерал>; } |
| Внешняя процедура | void function <идентификатор> (<тип> <идентификатор>, ...) {... return ; } |
| Условная конструкция | if: <идентификатор1> <оператор> <идентификатор2> блок1 {...} ^ блок2 {...} <идентификатор1>, <идентификатор2> - идентификаторы или литералы целочисленного типа (но не два литерала одновременно). <оператор> - один из операторов сравнения (> <), устанавливающий отношение между двумя операндами и организующий условие данной конструкции. При истинности условия выполняется код внутри блока1, иначе – код внутри блока блок2 . |

1.15 Области видимости идентификаторов

Область видимости: сверху вниз (как и в C++). Переменные, объявленные в одной функции, недоступны в другой. Все объявления и операции с переменными происходят внутри какого-либо блока. Каждая переменная или параметр функции получают префикс – название функции, внутри которой они находятся.

Все идентификаторы являются локальными и обязаны быть объявленными внутри какой-либо функции. Глобальных переменных нет. Параметры видны только внутри функции, в которой объявлены.

1.16 Семантические проверки

В языке программирования GKV-2022 выполняются следующие семантические проверки:

1. Наличие функции **main** – точки входа в программу;
2. Единственность точки входа;
3. Переопределение идентификаторов;
4. Использование идентификаторов без их объявления;
5. Проверка соответствия типа функции и возвращаемого параметра;
6. Правильность передаваемых в функцию параметров: количество, типы;
7. Правильность строковых выражений;
8. Превышение размера строковых и числовых литералов;
9. Правильность составленного условия условного оператора.

1.17 Распределение оперативной памяти на этапе выполнения

Транслированный код использует две области памяти. В сегмент констант заносятся все литералы. В сегмент данных заносятся переменные и параметры функций. Локальная область видимости в исходном коде определяется за счет использования правил именования идентификаторов и регулируется их префиксами, что и обуславливает их локальность на уровне исходного кода, несмотря на то, что в оттранслированном в язык ассемблера коде переменные имеют глобальную область видимости.

1.18 Стандартная библиотека и её состав

В языке GKV-2022 присутствует стандартная библиотека, которая подключается автоматически при трансляции исходного кода в язык ассемблера. Содержимое библиотеки и описание функций представлено в таблице 1.8.

Таблица 1.8 Стандартная библиотека языка GKV-2022

| Функция | Описание |
|---|--|
| integer len(stroka str1); | Строковая функция, высчитывает длину строки. |
| integer comp(stroka str1, str2); | Сравнивает две строки и возвращает 1, если строки равны и 0, если строки не равны. |

Стандартная библиотека написана на языке C++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю. Для вывода предусмотрен оператор **output**. Эти функции представлены в таблице 1.9.

Таблица 1.9 Дополнительные функции стандартной библиотеки

| Функция на языке C++ | Описание |
|-------------------------------------|---|
| <code>void outn(int value)</code> | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| <code>void outw(char* value)</code> | Функция для вывода в стандартный поток значения строкового идентификатора/литерала. |

1.19 Ввод и вывод данных

Вывод данных осуществляется с помощью оператора **output**. Допускается использование оператора **output** с литералами и идентификаторами.

Функции, управляющие выводом данных, реализованы на языке C++ и вызываются из транслированного кода, конечному пользователю недоступны. Пользовательская команда **output** в транслированном коде будут заменена вызовом нужных библиотечных функций. Библиотека, содержащая нужные процедуры, подключается на этапе генерации кода.

1.20 Точка входа

В языке GKV-2022 каждая программа должна содержать главную функцию (точку входа) **main**, с первой инструкции которой начнётся последовательное выполнение команд программы.

1.21 Препроцессор

Препроцессор, принимающий и выдающий некоторые данные на вход транслятору, в языке GKV-2022 отсутствует.

1.22 Соглашения о вызовах

В языке вызов функций происходит по соглашению о вызовах `stdcall`. Особенности `stdcall`:

- все параметры функции передаются через стек;
- память высвобождает вызываемый код;
- занесение в стек параметров идёт справа налево.

1.23 Объектный код

Язык GKV-2022 транслируется в язык ассемблера, а затем - в объектный код.

1.24 Классификация сообщений транслятора

Генерируемые транслятором сообщения определяют степень его информативности, то есть сообщения транслятора должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Сообщения транслятора приведены в таблице 1.10, а также в приложении А.

Таблица 1.10 Классификация ошибок

| Номера ошибок | Характеристика |
|---------------|--------------------------------|
| 0 – 200 | Системные ошибки |
| 200 – 299 | Ошибки лексического анализа |
| 300 – 399 | Ошибки семантического анализа |
| 600 – 699 | Ошибки синтаксического анализа |

1.25 Контрольный пример

Контрольный пример демонстрирует главные особенности языка GKV-2022: его фундаментальные типы, основные структуры, функции, процедуры, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

В языке GKV-2022 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые им функции, которые представлены в пункте 2.1. Для того чтобы получить ассемблерный код, используются выходные данные работы лексического анализатора, а именно таблица лексем и таблица идентификаторов. Для указания выходных файлов используются входные параметры транслятора, которые описаны в таблице 2.1. Структура транслятора языка GKV-2022 приведена на рисунке 1.

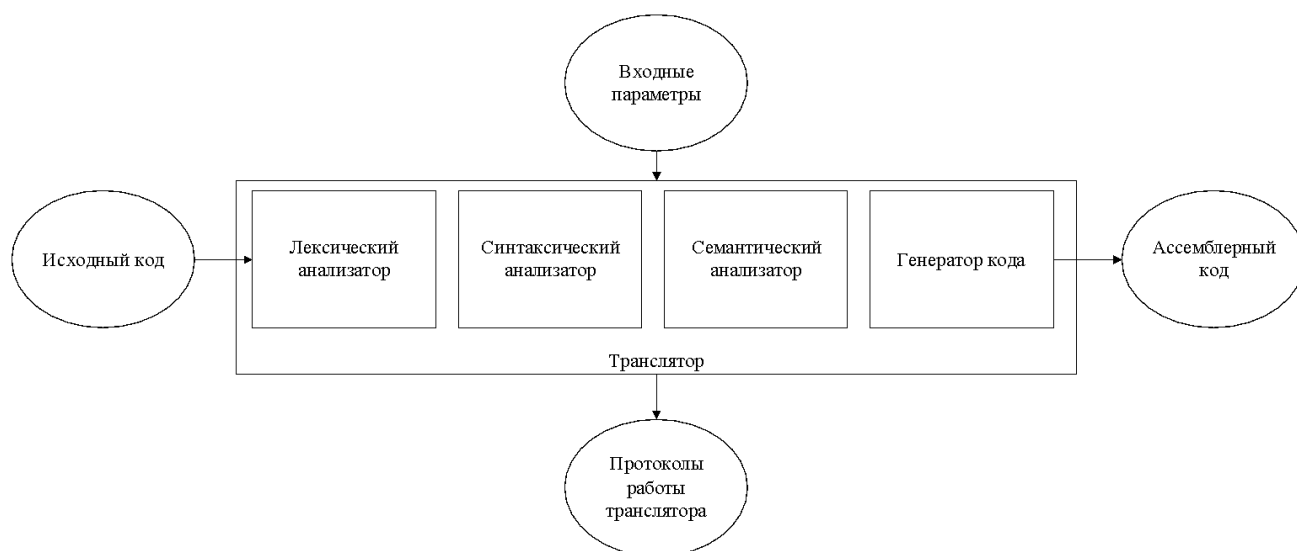


Рисунок 2.1 Структура транслятора языка программирования GKV-2022

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся последовательность символов входного языка. Он производит предварительный разбор текста, преобразуя единый массив текстовых символов в массив отдельных слов (в теории компиляции вместо термина «слово» часто используют термин «токен»). Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация. Таблица лексем (ТЛ) и таблица идентификаторов (ТИ) являются входом для следующей фазы компилятора – синтаксического анализа (разбора, парсера).

Цели лексического анализатора:

- убрать все лишние пробелы;
- выполнить распознавание лексем;
- построить таблицу лексем и таблицу идентификаторов;
- при неуспешном распознавании или обнаружении некоторых ошибок во входном тексте выдать сообщение об ошибке.

Синтаксический анализатор – часть компилятора, выполняющая синтаксический анализ, то есть проверку исходного кода на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией является дерево разбора

Семантический анализатор – часть транслятора, выполняющая семантический анализ, то есть проверку исходного кода на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Генератор кода – часть транслятора, выполняющая генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

2.2 Перечень входных параметров транслятора

Для формирования файлов с результатами работы лексического, синтаксического и семантического анализаторов используются входные параметры транслятора, которые приведены в таблице 2.1.

Таблица 2.1 Входные параметры транслятора языка GKV-2022

| Входной параметр | Описание параметра | Значение по умолчанию |
|-------------------------|---|---|
| -in:<путь к in-файлу> | Файл с исходным кодом на языке GKV-2022 , имеющий расширение .txt | Не предусмотрено |
| -log:<путь к log-файлу> | Файл журнала для вывода протоколов работы программы. | Значение по умолчанию: <имя in-файла>.log |

2.3 Перечень протоколов, формируемых транслятором и их содержимое

В ходе работы программы формируются протоколы работы лексического, синтаксического и семантического анализаторов, которые содержат в себе перечень протоколов работы. В таблице 2.2 приведены протоколы, формируемые транслятором и их содержимое.

Таблица 2.2 Протоколы, формируемые транслятором языка GKV-2022

| Формируемый протокол | Описание выходного протокола |
|---|--|
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования GKV-2022 . Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа, а также результат работы алгоритма преобразования выражений к польской записи. |
| Выходной файл, с расширением ".asm" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

3 Разработка лексического анализатора

3.1 Структура лексического анализатора

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся исходный код входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка,. Лексический анализатор производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов.

Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация.

Функции лексического анализатора:

- удаление «пустых» символов и комментариев. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними (альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации);

- распознавание идентификаторов и ключевых слов;
- распознавание констант;
- распознавание разделителей и знаков операций.

Исходный код программы представлен в приложении А, структура лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1 Структура лексического анализатора

3.2. Контроль входных символов

Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. Таблица входных символов представлена на рисунке 3.2, категории входных символов представлены в таблице 3.1.

```
#define IN_CODE_TABLE {\n
/*      0      1      2      3      4      5      6      7      8      9      A      B      C      D      E      F*/\n
/*0*/ IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::T, IN::I, '|', IN::T, IN::T, IN::I, IN::F, IN::F,\n
/*1*/ IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\n
/*2*/ IN::S, IN::T, IN::P, IN::F, IN::F, IN::S, IN::T, IN::P, IN::S, IN::S, IN::S, IN::S, IN::S, IN::S, IN::T, IN::S,\n
/*3*/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::S, IN::S, IN::S, IN::S, IN::S, IN::T,\n
/*4*/ IN::F, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
/*5*/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::S, IN::S, IN::S, IN::S, IN::T,\n
/*6*/ IN::F, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
/*7*/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::S, IN::S, IN::S, IN::S, IN::F,\n
\n
/*8*/ IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\n
/*9*/ IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::T, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\n
/*A*/ IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\n
/*B*/ IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\n
/*C*/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
/*D*/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
/*E*/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\n
/*F*/ IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T\n
}
```

Рисунок 3.2. Таблица контроля входных символов

Таблица 3.1 Соответствие символов и их значений в таблице

| Значение в таблице входных символов | Символы |
|-------------------------------------|---------|
| Разрешенный | T |
| Запрещенный | F |
| Игнорируемый | I |
| Сепаратор | S |
| Пробел, табуляция | P |

3.3 Удаление избыточных символов

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

1. Посимвольно считываем файл с исходным кодом программы;
2. Встреча пробела или знака табуляции является своего рода встречей символа-сепаратора;
3. В отличие от других символов-сепараторов не записываем в очередь лексем эти символы, т.е. игнорируем.

3.4 Перечень ключевых слов

Лексический анализатор преобразует исходный текст, заменяя лексические

единицы лексемами для создания промежуточного представления исходной программы. Соответствие токенов и лексем приведено в таблице 3.2.

Таблица 3.2 Соответствие токенов и сепараторов с лексемами

| Токен | Лексема | Пояснение |
|---------------------------|---------|--|
| integer, stroka | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 5 символов. |
| Литерал | l | Литерал любого доступного типа. |
| Шестнадцатеричный литерал | 16x | Литерал в шестнадцатеричном представлении. |
| function | f | Объявление функции. |
| return | r | Выход из функции/процедуры. |
| main | m | Главная функция. |
| output | @ | Ввод данных |
| condition: | ? | Указывает начало цикла/условного оператора. |
| istrue | r | Истинная ветвь условного оператора |
| isfalse | w | Ложная ветвь условного оператора |
| ^ | ^ | Разделение конструкций в цикле/условном операторе |
| ; | ; | Разделение выражений |
| , | , | Разделение параметров функций |
| { | { | Начало блока/тела функции |
| } | } | Закрытие блока/тела функции |
| (| (| Передача параметров в функцию, приоритет операций |
|) |) | Закрытие блока для передачи параметров, приоритет операций |
| = | = | Знак присваивания |
| + | + | Знаки операций |
| - | - | |
| * | * | |
| / | / | |
| % | % | |

Продолжение таблицы 3.2

| | | |
|---|---|-----------------------------|
| > | > | Знаки логических операторов |
| < | < | |

Пример реализации таблицы лексем представлен в приложении Б.

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному

графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. Структура конечного автомата и пример графа перехода конечного автомата изображены на рисунках 3.3 и 3.4 соответственно.

```
namespace FST {
    struct RELATION {
        unsigned char symbol;
        short nnode;           //номер вершины

        RELATION(unsigned char c = 0x00, short ns = NULL);
    };

    struct NODE {
        short n_relation;
        RELATION* relations;

        NODE();
        NODE(short n, RELATION rel, ...);
    };

    struct FST {
        unsigned char* string;
        short position;
        short nstates;         // количество состояний автомата
        NODE* nodes;
        short* rstates;        // возможные состояния автомата на данной позиции

        FST(unsigned char* s, short ns, NODE n, ...);
    };
}
```

Рисунок 3.3 Структура конечного автомата

```
#define FST_INTEGER 8, \
    FST::NODE(1, FST::RELATION('i', 1)), \
    FST::NODE(1, FST::RELATION('n', 2)), \
    FST::NODE(1, FST::RELATION('t', 3)), \
    FST::NODE(1, FST::RELATION('e', 4)), \
    FST::NODE(1, FST::RELATION('g', 5)), \
    FST::NODE(1, FST::RELATION('e', 6)), \
    FST::NODE(1, FST::RELATION('r', 7)), \
    FST::NODE()
```

Рисунок 3.4 Пример реализации графа конечного автомата для токена integer

3.5 Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде (sn), и номер в таблице идентификаторов, если лексема является идентификатором (idxTI). Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value). Код C++ со структурой таблицы лексем представлен на рисунке 3.3. Код C++ со структурой таблицы идентификаторов представлен на рисунке 3.4.

```

struct Entry    // строка таблицы лексем
{
    unsigned char lexema;    // лексема
    int sn;                // номер строки в исходном тексте
    int idxTI = TI_NULLIDX;    // индекс в таблице идентификаторов или LT_TI_NULLIDX
};

struct LexTable    // экземпляр таблицы лексем
{
    int max_size;        // емкость таблицы лексем < LT_MAXSIZE
    int size;            // текущий размер таблицы лексем < max_size
    Entry* table;        // массив строк таблицы лексем
};

```

Рисунок 3.3 Структура таблицы лексем

```

struct Entry    // строка таблицы идентификаторов
{
    int idxfirstLE;        // индекс первой строки в таблице лексем
    unsigned char id[ID_MAXSIZE];    // идентификатор (автоматически усекается до ID_MAXSIZE)
    IDDATATYPE iddatatype = NUL;    // тип данных
    IDTYPE idtype;        // тип идентификатора
    int parm = 0;
    int nums = 0; // 0 - 10 CC 1 - 16 CC
    union
    {
        int vint;        // значение integer
        struct
        {
            int len;        // количество символов в string
            unsigned char str[TI_STR_MAXSIZE - 1];    // символы string
        } vstr;        // значение string
    } value;        // значение идентификатора
};

struct IdTable    // экземпляр таблицы идентификаторов
{
    int max_size;        // емкость таблицы идентификаторов < TI_MAXSIZE
    int size;            // текущий размер таблицы идентификаторов < maxsize
    Entry* table;        // массив строк таблицы идентификаторов
};

```

Рисунок 3.4 Структура таблицы идентификаторов

3.6 Принцип обработки ошибок

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. При возникновении сообщения, лексический анализатор игнорирует найденную ошибку и продолжает работу с исходным кодом. Перечень сообщений представлен на рисунке 3.5.

```

-----
ERROR_ENTRY(200, "Лексическая ошибка: Недопустимый символ в исходном файле(-in)",
ERROR_ENTRY(201, "Лексическая ошибка: Неизвестная последовательность символов"),
ERROR_ENTRY(202, "Лексическая ошибка: Превышен размер таблицы лексем"),
ERROR_ENTRY(203, "Лексическая ошибка: Превышен размер таблицы идентификаторов"),
-----

```

Рисунок 3.5 - Сообщения лексического анализатора

3.7 Структура и перечень сообщений лексического анализатора

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Если в процессе анализа находятся более трёх ошибок, то анализ останавливается.

3.8 Параметры лексического анализатора

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся как в файл журнала, так и в командную строку.

3.9 Алгоритм лексического анализа

- проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;
- для выделенной части входного потока выполняется функция распознавания лексемы;
- при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
- формирует протокол работы;
- при неуспешном распознавании выдается сообщение об ошибке.

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «**stroka**» представлен на рисунке 3.2, где S0 – начальное, а S6 – конечное состояние автомата.

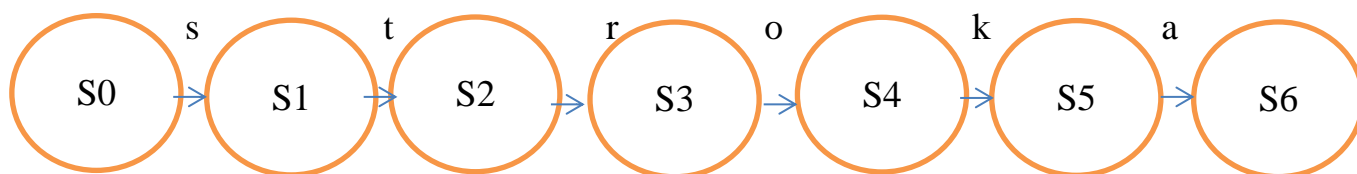


Рисунок 3.2 Пример графа переходов для цепочки stroka

3.10 Контрольный пример

Результат работы лексического анализатора в виде таблиц лексем и идентификаторов, соответствующих контрольному примеру, представлен в приложении Б.

4. Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализатор: часть компилятора, выполняющая синтаксический анализ, то есть исходный код проверяется на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией – дерево разбора

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

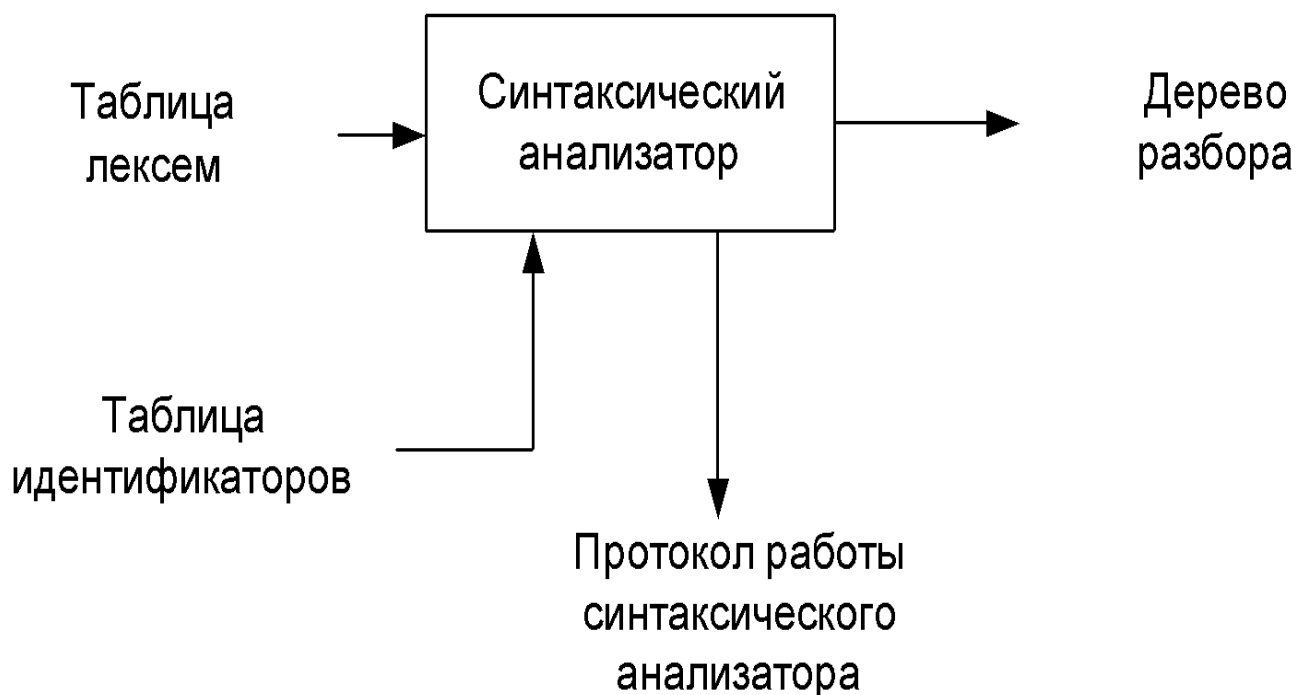


Рисунок 4.1 Структура синтаксического анализатора.

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка GKV-2022 используется контекстно-свободная грамматика $G = \langle T, N, P, S \rangle$, где

T – множество терминальных символов (было описано в разделе 1.2 данной пояснительной записки),

N – множество нетерминальных символов (первый столбец таблицы 4.1),

P – множество правил языка (второй столбец таблицы 4.1),

S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, т.к. она не леворекурсивная (не содержит леворекурсивных правил) и правила P имеют вид:

1) $A \rightarrow a\alpha$, где $a \in T, \alpha \in (T \cup N) \cup \{\lambda\}$; (или $\alpha \in (T \cup N)^*$, или $\alpha \in V^*$);

2) $S \rightarrow \lambda$, где $S \in N$ — начальный символ, при этом если такое правило существует, то нетерминал S не встречается в правой части правил. Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 Таблица правил переходов нетерминальных символов

| Символ | Правила | Какие правила порождает |
|--------|--------------------------------------|--|
| S | S->tfiPTS S->pfiPGS S->m[K] | Стартовые правила, описывающее общую структуру программы |
| P | P->(E) P->() | Правила для параметров объявляемых функций |
| T | T->[eV;] T->[KeV;] | Правила для тела функций |
| E | E->ti,E E->ti | Правила для списка параметров функции |
| F | F->(N) F->() | Правила для вывозов функций(в т.ч. и в выражениях) |
| N | N->i N>l N->l,N N->I,N | Правила для параметров вызываемых функций |
| R | R->rY# R>wY# R>cY# | Правила составления условного оператора |
| Z | Z->iLi Z->iLl Z->lli | Правила для условия условного оператора |
| L | L->< L->> | Правила для логических операторов |
| A | A->+ A->- A->* A->/ A->% | Правила для арифметических операторов |

Продолжение таблицы 4.1

| Символ | Правила | Какие правила порождает |
|--------|--------------|-------------------------------|
| V | V->l V->i | Правила для простых выражений |

| | | |
|---|---|--|
| | V->q | |
| Y | Y->[X] | Правила для тела условного выражения |
| W | W->l W->i W->(W) W->(W)AW W->iF W->iAW W->lAW W->iFAW | Правила для сложных выражений |
| K | K->nti=V;K K->nti;K K->i=W;K K->oV;K K->^;K K->&Z#RK K->iF;K K->nti=V; K->nti; K->i=W; K->oV; K->^; K->&Z#R K->iF; | Программные конструкции |
| X | X->i=W;X X->oV;X X->^;X X->iF;X X->i=W; X->oV; X->^; X->iF; | Программные конструкции внутри условного оператора |

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку $M = \langle Q, V, Z, \delta, q_0, z_0, F \rangle$. Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

| Компонента | Определение | Описание |
|------------|---|--|
| Q | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
| V | Алфавит входных символов | Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1. |
| Z | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ \$) |
| δ | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
| q_0 | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
| z_0 | Начальное состояние магазина автомата | Символ маркера дна стека \$ |
| F | Множество конечных состояний | Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты |

4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора представляются в виде структуры магазинного конечного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка GKV-2022 . Данные структуры в приложении В.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:

1. В магазин записывается стартовый символ;
2. На основе полученных ранее таблиц формируется входная лента;
3. Запускается автомат;
4. Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке;

5. Если терминалы в стеке и в ленте совпадают, то данный терминал удаляется из ленты и стека. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другую цепочку нетерминала;
6. Если в магазине встретился нетерминал, переходим к пункту 4;
7. Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе генерируется исключение.

4.6 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен на рисунке 4.3.

```
ERROR_ENTRY(600, "Синтаксическая ошибка: Неверная структура программы"),  
ERROR_ENTRY(601, "Синтаксическая ошибка: Ошибка в теле функции"),  
ERROR_ENTRY(602, "Синтаксическая ошибка: Ошибка в выражении"),  
ERROR_ENTRY(603, "Синтаксическая ошибка: Ошибка в параметрах функции"),  
ERROR_ENTRY(604, "Синтаксическая ошибка: Ошибка в параметрах вызываемой функции"),  
ERROR_ENTRY_NODEF(605),  
ERROR_ENTRY(606, "Синтаксическая ошибка: Неверная структура условия"),
```

Рисунок 4.3 - Сообщения синтаксического анализатора

4.7. Параметры синтаксического анализатора и режимы его работы

Входной информацией для синтаксического анализатора является таблица лексем и идентификаторов. Кроме того используется описание грамматики в форме Грейбах. Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

4.8. Принцип обработки ошибок

Синтаксический анализатор выполняет разбор исходной последовательности лексем до тех пор, пока не дойдёт до конца цепочки лексем или не найдёт ошибку. Тогда анализ останавливается и выводится сообщение об ошибке (если она найдена). Если в процессе анализа находится ошибка, то процесс останавливается и выводится сообщение об ошибке.

4.9. Контрольный пример

Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью приведены в приложении В.

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет необходимые ошибки. Некоторые проверки (такие как проверка на единственность точки входа, проверка на предварительное объявление переменной) осуществляются в процессе лексического анализа. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1. Структура семантического анализатора

5.2 Функции семантического анализатора

Семантический анализатор выполняет проверку на основные правила языка (семантики языка), которые описаны в разделе 1.16.

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены на рисунке 5.1.

```
ERROR_ENTRY(301, "Семантическая ошибка: Отсутствует точка входа main"),
ERROR_ENTRY(302, "Семантическая ошибка: Обнаружено несколько точек входа main"),
ERROR_ENTRY_NODEF(303),
ERROR_ENTRY_NODEF(304),
ERROR_ENTRY(305, "Семантическая ошибка: Попытка переопределения идентификатора"),
ERROR_ENTRY_NODEF(306),
ERROR_ENTRY_NODEF(307),
ERROR_ENTRY(308, "Семантическая ошибка: Кол-во ожидаемых и переданных функция и параметров не совпадают"),
ERROR_ENTRY(309, "Семантическая ошибка: Несовпадение типов передаваемых параметров"),
ERROR_ENTRY(310, "Семантическая ошибка: Использование пустого строкового литерала недопустимо"),
ERROR_ENTRY(311, "Семантическая ошибка: Не закрыт строковый литерал"),
ERROR_ENTRY_NODEF(312),
ERROR_ENTRY(313, "Семантическая ошибка: Недопустимый размер целочисленного литерал"),
ERROR_ENTRY(314, "Семантическая ошибка: Типы данных в выражении не совпадают"),
ERROR_ENTRY(315, "Семантическая ошибка: Тип функции и возвращаемого значения не совпадают"),
ERROR_ENTRY(316, "Семантическая ошибка: Недопустимое строковое выражение справа от знака '='"),
ERROR_ENTRY(317, "Семантическая ошибка: Неверное условное выражение"),
```

Рисунок 5.2 – Перечень сообщений семантического анализатора

5.4 Принцип обработки ошибок

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Анализ останавливается после того, как будут найдены все ошибки.

5.5 Контрольный пример

Соответствие примеров некоторых ошибок в исходном коде и диагностических сообщений об ошибках приведено в таблице 5.1.

Таблица 5.1. Примеры диагностики ошибок

| | |
|--|--|
| <pre>main{ integer x = 9; stroka y = x; }</pre> | Ошибка N314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 3 |
| <pre>main{ integer x = 9; } main{ stroka y = "qwerty"; }</pre> | Ошибка N302: Семантическая ошибка: Обнаружено несколько точек входа main Строка: 0 |

6. Вычисление выражений

6.1 Выражения, допускаемые языком

В языке GKV-2022 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритеты операций

| Операция | Значение приоритета |
|----------|---------------------|
| () | 3 |
| * | 2 |
| / | 2 |
| % | 2 |
| + | 1 |
| - | 1 |

6.2 Польская запись и принцип её построения

Все выражения языка GKV-2022 преобразовываются к обратной польской записи.

Польская запись - это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок. Существует два типа польской записи: прямая и обратная, также известные как префиксная и постфиксная. Отличие их от классического, инфиксного способа заключается в том, что знаки операций пишутся не между, а, соответственно, до или после аргументов. Алгоритм построения польской записи:

- исходная строка: выражение;
- результирующая строка: польская запись;
- стек: пустой;
- исходная строка просматривается слева направо;
- операнды переносятся в результирующую строку;
- операция записывается в стек, если стек пуст;
- операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
- отрывающая скобка помещается в стек;
- закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются.

6.3 Программная реализация обработки выражений

Программная реализация алгоритма преобразования выражений к польской записи представлена в приложении Г.

6.4 Контрольный пример

Пример преобразования выражений из контрольных примеров к обратной польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления и преобразования к ассемблерному коду. В приложении Г приведены изменённые таблицы лексем и идентификаторов, отображающие результаты преобразования выражений в польский формат.

Таблица 6.2. Преобразование выражений к ПОЛИЗ

| Выражение | Обратная польская запись для выражения |
|--|--|
| $i[2]=(((l[3]+l[4])-i[0])*l[5])/l[6];$ | $i[2]=l[3]l[4]+i[0]-l[5]*l[6]/$ |
| $i[23]=(i[23]+l[26])*l[26]$ | $i[23]=i[23]l[26]+l[26]*$ |
| $i[3]=(((l[4]+l[5])-i[0])*l[6])$ | $i[3]=l[4]l[5]+i[0]-l[6]*$ |

7. Генерация кода

7.1 Структура генератора кода

В языке GKV-2022 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода GKV-2022 представлена на рисунке 7.1.

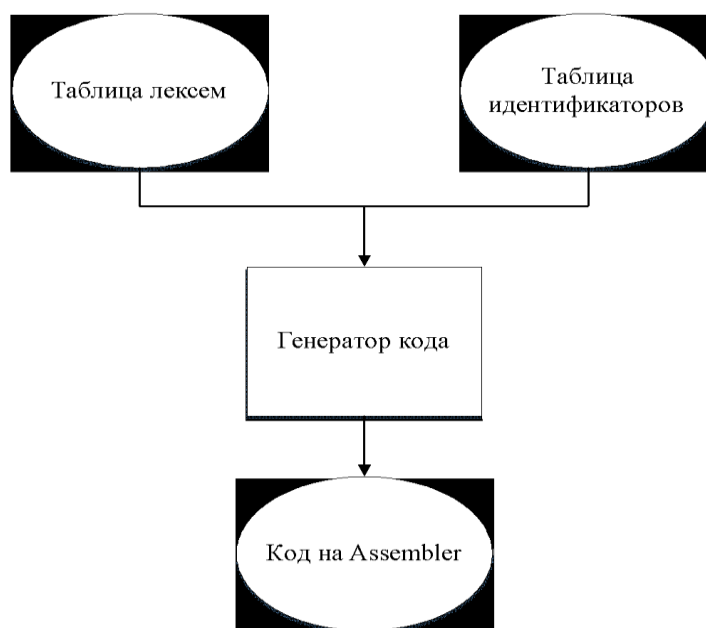


Рисунок 7.1 – Структура генератора кода

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены сегментах `.data` и `.const` языка ассемблера. Соответствия между типами данных идентификаторов на языке GKV-2022 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка GKV-2022 и языка ассемблера

| Тип идентификатора на языке GKV-2022 | Тип идентификатора на языке ассемблера | Пояснение |
|--------------------------------------|--|--|
| integer | sdword | Хранит целочисленный тип данных. |
| stroka | byte | Хранит указатель на начало строки. Строка должна завершаться нулевым символом. |

7.3 Статическая библиотека

В языке GKV-2022 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически.

Таблица 7.3 – Функции статической библиотеки

| Функция | Назначение |
|--|---|
| int outw(char* value) | Вывод на консоль строки value |
| int outn(int value) | Вывод на консоль целочисленной переменной value |
| int len(char* source) | Вычисление длины строки |
| int comp(char* source1, char* source2) | сравнение строк source и str2 |

7.4 Особенности алгоритма генерации кода

В языке GKV-2022 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке



Рисунок 7.2 – Структура генератора кода

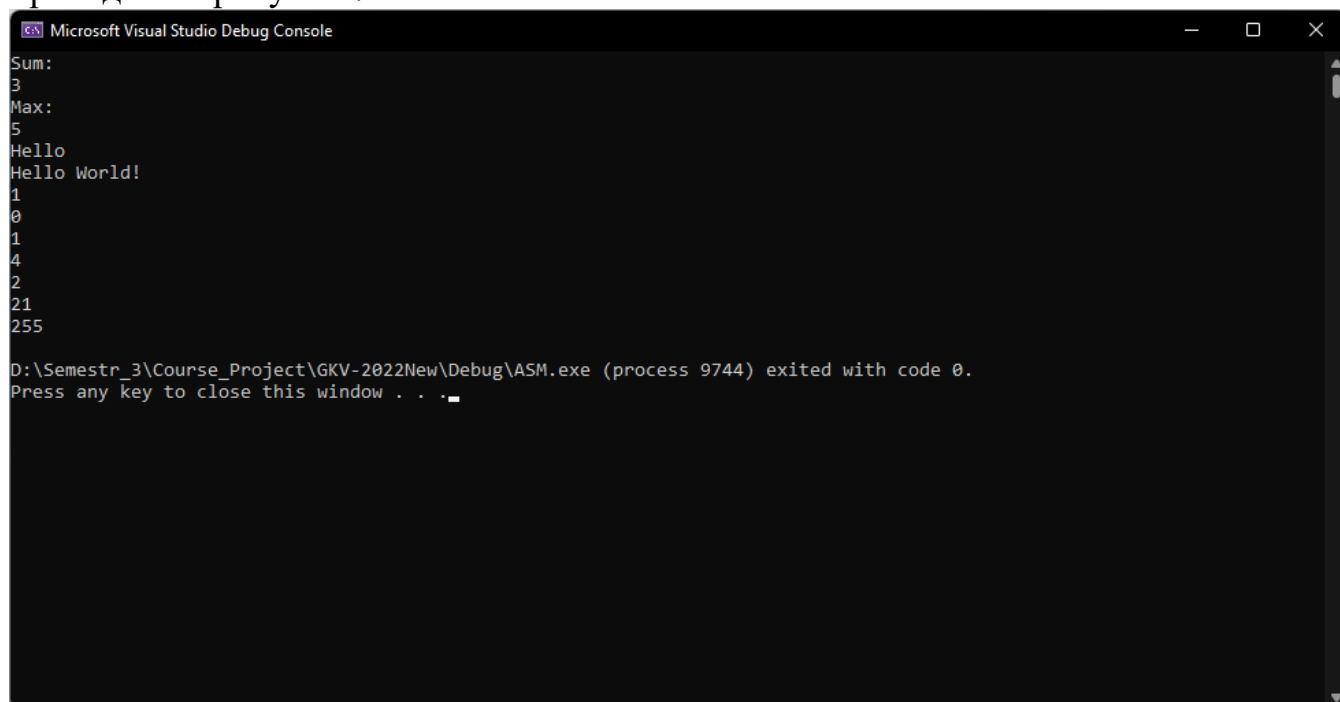
7.5 Входные параметры генератора кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке GKV-2022. Результаты работы генератора кода выводятся в файл с расширением .asm.

7.6 Контрольный пример

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении Д. Результат работы контрольного примера

приведён на рисунке 7.2.

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard window controls (minimize, maximize, close). The console output is as follows:

```
Sum:  
3  
Max:  
5  
Hello  
Hello World!  
1  
0  
1  
4  
2  
21  
255  
  
D:\Semestr_3\Course_Project\GKV-2022New\Debug\ASM.exe (process 9744) exited with code 0.  
Press any key to close this window . . .
```

Рисунок 7.2 Результат работы программы на языке GKV-2022

8. Тестирование транслятора

8.1 Общие положения

В основе тестов лежит проверка работоспособности всех анализаторов. При обнаружении компилятором ошибки она будет обрабатываться одним из анализаторов в зависимости от типа ошибки. Все сообщения об ошибках будут храниться в файле text.txt.log.

8.2 Результаты тестирования

В языке GKV-2022 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 - Тестирование проверки на допустимость символов

| Исходный код | Диагностическое сообщение |
|--|--|
| <pre>... main { ё integer a = 2; ... }</pre> | Ошибка 111: Не доступный символ в исходном файле(-in) Строка: 13 Позиция в строке: 2 |

На этапе лексического анализа в языке GKV-2022 могут возникнуть ошибки, описанные в пункте 3.7. Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 - Тестирование лексического анализатора

| Исходный код | Диагностическое сообщение |
|--|---|
| <pre>... main {... integer x1a1 = 1; ... }</pre> | Ошибка 201: Лексический анализатор: Неверное имя идентификатора строка 14 позиция 4 |

На этапе синтаксического анализа в языке GKV-2022 могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 - Тестирование синтаксического анализатора

| Исходный код | Диагностическое сообщение |
|--|--|
| main { integer x; } } | Ошибка 601: Синтаксическая ошибка: Ошибка в теле функции. 601: строка 14, Синтаксическая ошибка: Ошибка в теле функции |
| stroka function fi({ } main{ } | Ошибка 603: Синтаксическая ошибка: Ошибка в параметрах функции 603: строка 12, Синтаксическая ошибка: Ошибка в параметрах функции |
| stroka function fi() { output } | Ошибка 602: Синтаксическая ошибка: Ошибка в выражении 602: строка 15, Синтаксическая ошибка: Ошибка в выражении |
| main { sum(4, 5; } | Ошибка 600: Синтаксическая ошибка: Неверная структура программы 600: строка 12, Синтаксическая ошибка: Неверная структура программы |
| main { integer x = 1; integer y = 2; if: x > y } | Ошибка 606: Синтаксическая ошибка: Неверная структура условия 606: строка 19, Синтаксическая ошибка: Неверная структура условия |

Семантический анализ в языке GKV-2022 содержит множество проверок по семантическим правилам, описанным в пункте 1.16. Итоги тестирования семантического анализатора на корректное обнаружение семантических ошибок приведены в таблице 8.4.

Таблица 8.4 - Тестирование семантического анализатора

| Исходный код | Диагностическое сообщение |
|--|---|
| stroka function fi(){ } | Ошибка 301: Семантическая ошибка: Отсутствует точка входа main |
| main{ } main{ } | Ошибка 302: Семантическая ошибка: Обнаружено несколько точек входа main |
| main { integer x = 1; integer x = 2; } | Ошибка 305: Семантическая ошибка: Попытка переопределения идентификатора строка 4 позиция 0 |

Продолжение таблицы 8.4

| Исходный код | Диагностическое сообщение |
|--|--|
| integer function fi(integer x){return x;} main{fi();} | Ошибка 308: Семантическая ошибка: Кол-во ожидаемых и переданных функция и параметров не совпадают строка 3 позиция 0 |
| integer function fi(integer x){return x;} main { stroka a = "1"; fi(a); } | Ошибка 309: Семантическая ошибка: Несовпадение типов передаваемых параметров строка 6 позиция 0 |
| main{ stroka x = ""; } | Ошибка 310: Семантическая ошибка: Использование пустого строкового литерала недопустимо строка 2 позиция 0 |
| main{ stroka x = " } | Ошибка 311: Семантическая ошибка: Не закрыт строковый литерал строка 2 позиция 0 |
| stroka function fi() { integer x = 5; return x; } | Ошибка 315: Семантическая ошибка: Тип функции и возвращаемого значения не совпадают строка 4 |
| main { stroka x; x = "abc" + "d"; } | Ошибка 316: Семантическая ошибка: Недопустимое строковое выражение справа от знака '=' строка 4 |

Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования GKV-2022 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

1. Сформулирована спецификация языка GKV-2022;
2. Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
3. Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
4. Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
5. Осуществлена программная реализация синтаксического анализатора;
6. Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
7. Разработан транслятор кода на язык ассемблера;
8. Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка GKV-2022 включает:

1. 2 типа данных;
2. Поддержка оператора вывода строки;
3. Возможность вызова функций стандартной библиотеки;
4. Наличие 5 арифметических операторов для вычисления выражений;
5. Поддержка функций, процедур, операторов цикла и условия;
6. Структурированная и классифицированная система для обработки ошибок пользователя.

Проделанная работа позволила получить необходимое представление о структурах и процессах, использующихся при построении трансляторов, а также основные различия и преимущества тех или иных средств трансляции.

Список использованных источников

1. Курс лекций по КПО Наркевич А.С.
2. Построение компиляторов / Никлаус Вирт 2010. – 194 с.
3. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.
4. Язык программирования C++. Лекции и упражнения [6-е издание] / Стивен Прата 2019 – 1094 с.

Приложение А

Листинг 1 – Исходный код программы на языке GKV-2022

```
integer function max(integer x, integer y){
integer res;
    if: x>y
    {res = x;}
    ^{ res = y;}
    return res;
}
integer function sum(integer x, integer y){
    x = x + y;
    return x;
}
main
{
    output "Sum:";
    integer test1 = sum(2, 1);
    output test1;
    output "Max:";
    integer test2 = max(2, 5);
    output test2;
    stroka hi = "Hello World!";
    stroka h1 = "Hello";
    integer a = comp(hi, hi);
    integer a1 = comp(hi, h1);
    output h1;
    output hi;
    output a;
    output a1;
    integer test3 = 7 % 3;
    integer test4 = 7 - 3;
    integer test5 = 7 / 3;
    integer test6 = 7 * 3;
    output test3;
    output test4;
    output test5;
    output test6;
    return;
}
```


Приложение Б

Листинг 1 Таблица идентификаторов контрольного примера

| -----Таблица индентификаторов----- | | | | | | |
|------------------------------------|---------|------------|---------------------|------------------|------------|------------------|
| № | Имя | тип данных | тип индификатора | первое вхождение | содержание | |
| 1 | max | integer | Функция | 2 2 | | - |
| 2 | maxx | integer | Параметр | 5 | | - |
| 3 | maxy | integer | Параметр | 8 | | - |
| 4 | maxres | integer | Переменная | 12 | | - |
| 5 | > | - | Логический оператор | 17 | | - |
| 6 | sum | integer | Функция | 2 38 | | - |
| 7 | sumx | integer | Параметр | 41 | | - |
| 8 | sumy | integer | Параметр | 44 | | - |
| 9 | + | - | Оператор | 50 | | - |
| 10 | L1 | stroka | Литерал | 60 | | [4]Sum: |
| 11 | test1 | integer | Переменная | 63 | | - |
| 12 | L2 | integer | Литерал | 67 | | 2 |
| 13 | L3 | integer | Литерал | 69 | | 1 |
| 14 | L4 | stroka | Литерал | 76 | | [4]Max: |
| 15 | test2 | integer | Переменная | 79 | | - |
| 16 | L5 | integer | Литерал | 85 | | 5 |
| 17 | hi | stroka | Переменная | 92 | | - |
| 18 | L6 | stroka | Литерал | 94 | | [12]Hello World! |
| 19 | h1 | stroka | Переменная | 97 | | - |
| 20 | L7 | stroka | Литерал | 99 | | [5]Hello |
| 21 | a | integer | Переменная | 102 | | - |
| 22 | comp | integer | Функция | 2 104 | | - |
| 23 | source1 | stroka | Параметр | -1 | | - |
| 24 | source2 | stroka | Параметр | -1 | | - |
| 25 | a1 | integer | Переменная | 112 | | - |
| 26 | test3 | integer | Переменная | 134 | | - |
| 27 | L8 | integer | Литерал | 136 | | 7 |
| 28 | % | - | Оператор | 137 | | - |
| 29 | L9 | integer | Литерал | 138 | | 3 |
| 30 | test4 | integer | Переменная | 141 | | - |
| 31 | - | - | Оператор | 144 | | - |
| 32 | test5 | integer | Переменная | 148 | | - |
| 33 | / | - | Оператор | 151 | | - |
| 34 | test6 | integer | Переменная | 155 | | - |
| 35 | * | - | Оператор | 158 | | - |
| 36 | b | integer | Переменная | 174 | | - |
| 37 | L10 | integer | Литерал | 176 | | 255 |

Приложение В

Листинг 1 Грамматика языка GKV-2022

```

Greibach greibach(
    NS('S'), // стартовый символ
    TS('$'), //дно стека
    11, // количество правил
    Rule(
        NS('S'),
        GRB_ERROR_SERIES + 0, // неверная структура
        программы
        6, //
        Rule::Chain(2, TS('m'), NS('T')),
        Rule::Chain(7, TS('t'), TS('f'), TS('i'), TS('('),
        TS(')'), NS('T'), NS('S')),
        Rule::Chain(8, TS('t'), TS('f'), TS('i'), TS('('),
        NS('F'), TS(')'), NS('T'), NS('S')),
        Rule::Chain(7, TS('n'), TS('f'), TS('i'), TS('('),
        TS(')'), NS('T'), NS('S')),
        Rule::Chain(8, TS('n'), TS('f'), TS('i'), TS('('),
        NS('F'), TS(')'), NS('T'), NS('S')),
        Rule::Chain(3, TS('m'), NS('T'), NS('S'))
    ),
    Rule(
        NS('T'),
        GRB_ERROR_SERIES + 0, // неверная структура
        программы
        4, //
        Rule::Chain(5, TS('{'), NS('N'), TS('r'), TS(';'),
        TS('}')),
        Rule::Chain(6, TS('{'), NS('N'), TS('r'), NS('E'),
        TS(';'), TS('}')),
        Rule::Chain(3, TS('{'), NS('N'), TS('r'), TS(';'),
        TS('}')),
        Rule::Chain(2, TS('{'), TS('r'), TS(';'), TS('}')),
        Rule::Chain(5, TS('{'), TS('r'), NS('E'), TS(';'),
        TS('}'))
    ),
    Rule(
        NS('N'),
        GRB_ERROR_SERIES + 1, // конструкции в функциях
        20, //
        Rule::Chain(4, TS('t'), TS('i'), TS(';'), NS('N')),
        Rule::Chain(6, TS('i'), TS('('), NS('W'), TS(')'),
        TS(';'), NS('N')),
        Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'),
        TS(';')),
        Rule::Chain(5, TS('i'), TS('('), TS(')'), TS(';'),
        NS('N')),
        Rule::Chain(4, TS('i'), TS('('), TS(')'), TS(';')),
        Rule::Chain(6, TS('t'), TS('i'), TS('='), NS('E'),
        TS(';'), NS('N')),

```

```

Rule::Chain(5, TS('t'), TS('i'), TS('='), NS('E'),
TS(';')),
Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),
Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';')),
NS('N')),
Rule::Chain(4, TS('p'), TS('i'), TS(';'), NS('N')),
Rule::Chain(4, TS('p'), TS('l'), TS(';'), NS('N')),
Rule::Chain(4, TS('p'), NS('E'), TS(';'), NS('N')),
Rule::Chain(3, TS('p'), NS('E'), TS(';')),
Rule::Chain(3, TS('t'), TS('i'), TS(';')),
Rule::Chain(3, TS('p'), TS('i'), TS(';')),
Rule::Chain(3, TS('p'), TS('l'), TS(';')),
Rule::Chain(4, TS('w'), NS('I'), TS('|'), NS('X')),
Rule::Chain(4, TS('o'), NS('I'), TS('|'), NS('O')),
Rule::Chain(5, TS('w'), NS('I'), TS('|'), NS('X')),
NS('N')),
Rule::Chain(5, TS('o'), NS('I'), TS('|'), NS('O')),
NS('N'))
),
Rule(
NS('B'),
GRB_ERROR_SERIES + 6,      // конструкции условия
10,                        //
Rule::Chain(5, TS('t'), TS('i'), TS('='), NS('E'),
TS(';')),
Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),
Rule::Chain(3, TS('p'), NS('E'), TS(';')),
Rule::Chain(3, TS('t'), TS('i'), TS(';')),
Rule::Chain(3, TS('p'), TS('i'), TS(';')),
Rule::Chain(3, TS('p'), TS('l'), TS(';')),
Rule::Chain(4, TS('w'), NS('I'), TS('|'), NS('X')),
Rule::Chain(4, TS('o'), NS('I'), TS('|'), NS('O')),
Rule::Chain(5, TS('w'), NS('I'), TS('|'), NS('X')),
NS('N')),
Rule::Chain(5, TS('o'), NS('I'), TS('|'), NS('O')),
NS('N'))
),
Rule(
NS('X'),
GRB_ERROR_SERIES + 6,      // конструкции условия
2,
Rule::Chain(7, TS('{'), NS('N'), TS('}'), TS('^'),
TS('{'), NS('N'), TS('}')),
Rule::Chain(3, TS('{'), NS('N'), TS('}'))
),
Rule(
NS('O'),
GRB_ERROR_SERIES + 7,
1,                        // цикл
//
Rule::Chain(3, TS('{'), NS('N'), TS('}'))
),
Rule(
NS('E'),

```

```

GRB_ERROR_SERIES + 2,      // ошибка в выражении
8,                          //
Rule::Chain(1, TS('i')),
Rule::Chain(1, TS('l')),
Rule::Chain(3, TS('('), NS('E'), TS(')')),
Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),
Rule::Chain(3, TS('i'), TS('('), TS(')')),
Rule::Chain(2, TS('i'), NS('M')),
Rule::Chain(2, TS('l'), NS('M')),
Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),
Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'),
NS('M')),
Rule::Chain(4, TS('i'), TS('('), TS(')'), NS('M'))
),
Rule(
NS('F'),
GRB_ERROR_SERIES + 3,      // ошибка в параметрах
функции
2,                          //
Rule::Chain(2, TS('t'), TS('i')),
Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F'))
),
Rule(
NS('W'),
GRB_ERROR_SERIES + 4,      // ошибка в параметрах
вызываемой функции
4,                          //
Rule::Chain(1, TS('i')),
Rule::Chain(1, TS('l')),
Rule::Chain(3, TS('i'), TS(','), NS('W')),
Rule::Chain(3, TS('l'), TS(','), NS('W'))
),
Rule(
NS('M'),
GRB_ERROR_SERIES + 2,      // оператор
2,                          //
Rule::Chain(2, TS('v'), NS('E')),
Rule::Chain(3, TS('v'), NS('E'), NS('M'))
),
Rule(
NS('I'),
GRB_ERROR_SERIES + 6,      // условие
2,                          //
Rule::Chain(4, TS(':'), NS('E'), TS('q'), NS('E')),
Rule::Chain(6, TS(':'), NS('E'), TS('q'), NS('E'),
TS('&'), NS('I'))
)

```

Листинг 2 Структура магазинного автомата

```

struct Mfst {
    enum RC_STEP {
        NS_OK,
        NS_NORULE,
        NS_NORULECHAIN,
        NS_ERROR,
        TS_OK,
        TS_NOK,
        LENTA_END,
        SURPRISE,
    };

    struct MfstDiagnosis {
        short lenta_position;
        RC_STEP rc_step;
        short nrule;
        short nrule_chain;
        MfstDiagnosis();
        MfstDiagnosis(short plenta_position, RC_STEP
prc_step, short pnrule, short pnrule_chain);
    }diagnosis[MFST_DIAGN_NUMBER];

    class my_stack_MfstState :public std::stack<MfstState> {
    public:
        using std::stack<MfstState>::c;
    };

    GRBALPHABET* lenta;
    short lenta_position;
    short nrule;
    short nrulechain;
    short lenta_size;
    GRB::Greibach greibach;
    LT::LexTable lex;
    bool more = false;
    Log::LOG log;
    MFSTSTSTACK st;
    my_stack_MfstState storestate;
    Mfst();
    Mfst(LT::LexTable& plex, GRB::Greibach pgreibach);
    char* getCSt(char* buf);
    char* getCLenta(char* buf, short pos, short n = 25);
    char* getDiagnosis(short n, char* buf);
    bool savestate();
    bool resetstate();
    bool push_chain(GRB::Rule::Chain chain);
    RC_STEP step();
    bool start();
    bool saveddiagnosis(RC_STEP pprc_step);
    void printrules();

    struct Deduction {

```

```
        short size;  
        short* nrules;  
        short* nrulechains;  
        Deduction() {  
            size = 0;  
            nrules = 0;  
            nrulechains = 0;  
        }  
    }deduction;  
  
    bool savededucation();
```

Листинг 3 Структура грамматики Грейбах

```
struct Greibach //грамматика Грейбах
{
    short size; //количество правил
    GRBALPHABET startN; //стартовый символ
    GRBALPHABET stbottomT; //дно стека
    Rule* rules; //множество правил
    Greibach() { short size = 0; startN = 0; stbottomT = 0;
rules = 0; };
    Greibach(
        GRBALPHABET pstartN, //стартовый символ
        GRBALPHABET pstbottom, //дно стека
        short psize, //количество правил
        Rule r, ... //правила
    );
    short getRule( //получить правило, возвращается номер
правила или -1
        GRBALPHABET pnn, //левый символ правила
        Rule& prule //возвращаемое правило
грамматики
    );
    Rule getRule(short n); //получить правило по номеру
};
```

Листинг 4 Разбор исходного кода синтаксическим анализатором

| Шаг | Правило | Входная лента |
|------------|---------------|---------------------------|
| Стек | | |
| 0 | : S->tfi()TS | SS\$ |
| 0 | : SAVESTATE: | 1 |
| 0 | : | |
| tfi()TS\$ | | |
| 1 | : | fi(ti,ti){ti;w:iqi {i=i;} |
| fi()TS\$ | | |
| 2 | : | i(ti,ti){ti;w:iqi {i=i;}^ |
| i()TS\$ | | |
| 3 | : | (ti,ti){ti;w:iqi {i=i;}^{ |
| ()TS\$ | | |
| 4 | : | ti,ti){ti;w:iqi {i=i;}^{i |
|)TS\$ | | |
| 5 | : 2 | |
| 5 | : RESSTATE | |
| 5 | : | SS\$ |
| 6 | : S->tfi(F)TS | SS\$ |
| 6 | : SAVESTATE: | 1 |
| 6 | : | |
| tfi(F)TS\$ | | |
| 7 | : | fi(ti,ti){ti;w:iqi {i=i;} |
| fi(F)TS\$ | | |
| 8 | : | i(ti,ti){ti;w:iqi {i=i;}^ |
| i(F)TS\$ | | |
| 9 | : | (ti,ti){ti;w:iqi {i=i;}^{ |
| (F)TS\$ | | |
| 10 | : | ti,ti){ti;w:iqi {i=i;}^{i |
| F)TS\$ | | |
| 11 | : F->ti | ti,ti){ti;w:iqi {i=i;}^{i |
| F)TS\$ | | |
| 11 | : SAVESTATE: | 2 |
| 11 | : | ti,ti){ti;w:iqi {i=i;}^{i |
| ti)TS\$ | | |
| 12 | : | i,ti){ti;w:iqi {i=i;}^{i= |
| i)TS\$ | | |
| 13 | : | ,ti){ti;w:iqi {i=i;}^{i=i |
|)TS\$ | | |
| 14 | : 2 | |
| 14 | : RESSTATE | |
| 14 | : | ti,ti){ti;w:iqi {i=i;}^{i |
| F)TS\$ | | |
| 15 | : F->ti,F | ti,ti){ti;w:iqi {i=i;}^{i |
| F)TS\$ | | |
| 15 | : SAVESTATE: | 2 |
| 15 | : | ti,ti){ti;w:iqi {i=i;}^{i |
| ti,F)TS\$ | | |
| 16 | : | i,ti){ti;w:iqi {i=i;}^{i= |
| i,F)TS\$ | | |
| 17 | : | ,ti){ti;w:iqi {i=i;}^{i=i |
| ,F)TS\$ | | |
| 18 | : | ti){ti;w:iqi {i=i;}^{i=i; |


```

F)TS$
19 : F->ti ti){ti;w:iqi|{i=i;}^{i=i;
F)TS$
19 : SAVESTATE: 3 ti){ti;w:iqi|{i=i;}^{i=i;
19 : ti)TS$
20 : i){ti;w:iqi|{i=i;}^{i=i;}
i)TS$

```

Листинг 4 (прод.) Разбор исходного кода синтаксическим анализатором

```

975 : RESSTATE
975 : i;r;}}}}}}}}}}}}}}}}}}}}}}
E;Nr;}$
976 : E->iM i;r;}}}}}}}}}}}}}}}}}}}}}}
E;Nr;}$
976 : SAVESTATE: 78
976 : i;r;}}}}}}}}}}}}}}}}}}}}}}
iM;Nr;}$
977 : ;r;}}}}}}}}}}}}}}}}}}}}}}
M;Nr;}$
978 : NS_NRCHAIN/NS_NR
978 : RESSTATE
978 : i;r;}}}}}}}}}}}}}}}}}}}}}}
E;Nr;}$
979 : NS_NRCHAIN/NS_NR
979 : RESSTATE
979 : pi;r;}}}}}}}}}}}}}}}}}}}}}}
Nr;}$
980 : N->pE; pi;r;}}}}}}}}}}}}}}}}}}}}}}
Nr;}$
980 : SAVESTATE: 77
980 : pi;r;}}}}}}}}}}}}}}}}}}}}}}
pE;r;}$
981 : i;r;}}}}}}}}}}}}}}}}}}}}}}
E;r;}$
982 : E->i i;r;}}}}}}}}}}}}}}}}}}}}}}
E;r;}$
982 : SAVESTATE: 78
982 : i;r;}}}}}}}}}}}}}}}}}}}}}}
i;r;}$
983 : ;r;}}}}}}}}}}}}}}}}}}}}}}
;r;}$
984 : r;}}}}}}}}}}}}}}}}}}}}}}
r;}$
985 : ;}}}}}}}}}}}}}}}}}}}}}}
;}$
986 : }}}}}}}}}}}}}}}}}}}}}}}}}}}}}}} }$
987 : }}}}}}}}}}}}}}}}}}}}}}}}}}}}}}} $
988 : 6
989 : ----->LENTA END

```

Приложение Г

Листинг 1 Программная реализация механизма преобразования в ПОЛИЗ

```
bool __cdecl setPolishNotation(IT::IdTable& idtable, Log::LOG&
log, int lextable_pos, ltvec& v)
{
    //результатирующий вектор
    vector < LT::Entry > result;
    // стек для сохранения операторов
    stack < LT::Entry > s;
    // флаг вызова функции
    bool ignore = false;

    for (unsigned i = 0; i < v.size(); i++)
    {
        if (ignore)    // вызов функции считаем подставляемым
            значением и заносим в результат
        {
            result.push_back(v[i]);
            if (v[i].lexema == LEX_RIGHTTHESIS) {
                ignore = false;
            }
            continue;
        }
        int priority = getPriority(v[i], idtable); // его
        приоритет

        if (v[i].lexema == LEX_LEFTTHESIS || v[i].lexema ==
LEX_RIGHTTHESIS || v[i].lexema == LEX_PLUS || v[i].lexema ==
LEX_MINUS || v[i].lexema == LEX_STAR || v[i].lexema == LEX_DIRSLASH
|| v[i].lexema == LEX_LEFTBRACE || v[i].lexema == LEX_BRACELET)
        {
            if (s.empty() /*|| v[i].lexema ==
LEX_LEFTTHESIS*/)
            {
                s.push(v[i]);
                continue;
            }

            if (v[i].lexema == LEX_RIGHTTHESIS)
            {
                //выталкивание элементов до скобки
                while (!s.empty() && s.top().lexema !=
LEX_LEFTTHESIS)
                {
                    result.push_back(s.top());
                    s.pop();
                }
                if (!s.empty() && s.top().lexema ==
LEX_LEFTTHESIS)
                    s.pop();
                continue;
            }
        }
    }
}
```

```

//выталкивание элем с большим/равным приоритетом
в результат
while (!s.empty() && getPriority(s.top(),
idtable) >= priority)
{
    if(s.top().lexema != LEX_LEFTTHESIS)
        result.push_back(s.top());
    s.pop();
}
s.push(v[i]);

if (v[i].lexema == LEX_LITERAL || v[i].lexema ==
LEX_ID) // идентификатор, идентификатор функции или литерал
{
    if (idtable.table[v[i].idxTI].idtype ==
IT::IDTYPE::F)
        ignore = true;
    result.push_back(v[i]); // операнд заносим в
результатирующий вектор
}
if (v[i].lexema != LEX_LEFTTHESIS && v[i].lexema !=
LEX_RIGHTTHESIS && v[i].lexema != LEX_PLUS && v[i].lexema !=
LEX_MINUS && v[i].lexema != LEX_STAR && v[i].lexema != LEX_DIRSLASH
&& v[i].lexema != LEX_ID && v[i].lexema != LEX_LITERAL &&
v[i].lexema != LEX_LEFTBRACE && v[i].lexema != LEX_BRACELET &&
v[i].lexema != LEX_ENDIF)
{
    Log::WriteError(log, Error::geterror(1));
    return false;
}

while (!s.empty()) { result.push_back(s.top()); s.pop(); }
v = result;
return true;
}

```

Приложение Д

Листинг 1 Результат генерации кода контрольного примера в Ассемблере

| | |
|---|---|
| <pre> .586 .model flat, stdcall includelib libucrt.lib includelib kernel32.lib includelib "../Debug/Lib.lib" ExitProcess PROTO:DWORD .stack 4096 ExitProcess PROTO :DWORD outn PROTO : SDWORD outw PROTO : DWORD len PROTO : DWORD comp PROTO : DWORD, : DWORD stcmp PROTO : DWORD, : DWORD .const newline byte 13, 10, 0 L1 byte 'Sum:', 0 L2 sdword 2 L3 sdword 1 L4 byte 'Max:', 0 L5 sdword 5 L6 byte 'Hello World!', 0 L7 byte 'Hello', 0 L8 sdword 7 L9 sdword 3 L10 sdword 255 .data temp sdword ? buffer byte 256 dup(0) maxres sdword 0 test1 sdword 0 test2 sdword 0 hi dword ? h1 dword ? a sdword 0 a1 sdword 0 test3 sdword 0 test4 sdword 0 test5 sdword 0 test6 sdword 0 b sdword 0 left dword ? rig dword ? result sdword ? result_str byte 4 dup(0) .code int_to_char PROC uses eax ebx ecx edi esi, </pre> | <pre> mov cl, '-' mov[edi], cl inc edi plus : push dx inc esi test eax, eax jz fin cdq idiv ebx jmp plus fin : mov ecx, esi write : pop bx add bl, '0' mov[edi], bl inc edi loop write ret int_to_char ENDP ;----- max ----- max PROC, maxx : sdword, maxy : sdword ; --- save registers --- push ebx push edx ; ----- push maxx pop ebx mov left, ebx push maxy pop ebx mov rig, ebx mov edx, left cmp edx, rig jg right1 jl wrong1 right1: push maxx pop ebx mov maxres, ebx </pre> |
|---|---|

| | |
|---|--|
| <pre> pstr: dword, intfield : sdword mov edi, pstr mov esi, 0 mov eax, intfield cdq mov ebx, 10 idiv ebx test eax, 80000000h jz plus neg eax neg edx </pre> | <pre> jmp next1 wrong1: push maxy pop ebx mov maxres, ebx next1: </pre> |
| <pre> ; --- restore registers --- pop edx pop ebx ; ----- mov eax, maxres ret max ENDP ;----- ;----- sum ----- sum PROC, sumx : sdword, sumy : sdword ; --- save registers --- push ebx push edx ; ----- push sumx push sumy pop ebx pop eax add eax, ebx push eax pop ebx mov sumx, ebx ; --- restore registers --- pop edx pop ebx ; ----- mov eax, sumx ret sum ENDP ;----- ;----- MAIN ----- main PROC INVOKE outw, offset L1 push L3 push L2 </pre> | <pre> call sum push eax pop ebx mov test1, ebx mov eax, test1 mov result, eax INVOKE int_to_char, offset result_str, result INVOKE outw, offset result_str INVOKE outw, offset L4 push L5 push L2 call max push eax pop ebx mov test2, ebx mov eax, test2 mov result, eax INVOKE int_to_char, offset result_str, result INVOKE outw, offset result_str mov hi, offset L6 mov h1, offset L7 push hi push hi call comp push eax pop ebx mov a, ebx </pre> |

```

push h1
push hi
call comp
push eax

pop ebx
mov al, ebx

INVOKE outw, h1

INVOKE outw, hi

mov eax, a
mov result, eax
INVOKE int_to_char, offset
result_str, result
INVOKE outw, offset result_str

mov eax, al
mov result, eax
INVOKE int_to_char, offset
result_str, result
INVOKE outw, offset result_str

push L8
push L9
pop ebx
pop eax
cdq
mov edx, 0
idiv ebx
push edx

pop ebx
mov test3, ebx

push L8
push L9
pop ebx
pop eax
sub eax, ebx
push eax

pop ebx
mov test4, ebx

push L8
push L9
pop ebx
pop eax
cdq

```

```

idiv ebx
push eax

pop ebx
mov test5, ebx

push L8
push L9
pop ebx
pop eax
imul eax, ebx
push eax

pop ebx
mov test6, ebx

mov eax, test3
mov result, eax
INVOKE int_to_char, offset
result_str, result
INVOKE outw, offset result_str

mov eax, test4
mov result, eax
INVOKE int_to_char, offset
result_str, result
INVOKE outw, offset result_str

mov eax, test5
mov result, eax
INVOKE int_to_char, offset
result_str, result
INVOKE outw, offset result_str

mov eax, test6
mov result, eax
INVOKE int_to_char, offset
result_str, result
INVOKE outw, offset result_str

push L10

pop ebx
mov b, ebx

mov eax, b
mov result, eax
INVOKE int_to_char, offset
result_str, result
INVOKE outw, offset result_str
INVOKE ExitProcess, 0
main ENDP
end main

```

