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 YMA-2020»

Выполнил студент Ярмолик Максим Александрович

(Ф.И.О.)

Руководитель проекта ст.пр. Наркевич Аделина Сергеевна

(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 ст.пр. Наркевич Аделина Сергеевна

(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер ст.пр. Наркевич Аделина Сергеевна

(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2020

**Содержание**

[Введение 5](#_Toc59100349)

[Глава 1. Спецификация языка программирования 6](#_Toc59100350)

[1.1 Характеристика языка программирования 6](#_Toc59100351)

[1.2 Алфавит языка 6](#_Toc59100352)

[1.3 Применяемые сепараторы 7](#_Toc59100353)

[1.4 Применяемые кодировки 7](#_Toc59100354)

[1.5 Типы данных 8](#_Toc59100355)

[1.6 Преобразование типов данных 9](#_Toc59100356)

[1.7 Идентификаторы 9](#_Toc59100357)

[1.8 Литералы 10](#_Toc59100358)

[1.9 Объявления данных и область видимости 10](#_Toc59100359)

[1.10 Инициализация данных 10](#_Toc59100360)

[1.11 Инструкции языка 10](#_Toc59100361)

[1.12 Операции языка 11](#_Toc59100362)

[1.13 Выражения и их вычисления 11](#_Toc59100363)

[1.14 Программные конструкции языка 11](#_Toc59100364)

[1.15 Область видимости идентификаторов 12](#_Toc59100365)

[1.16 Семантические проверки 12](#_Toc59100366)

[1.17 Распределение оперативной памяти на этапе выполнения 13](#_Toc59100367)

[1.18 Стандартные библиотеки и их состав 13](#_Toc59100368)

[1.19 Ввод и вывод данных 14](#_Toc59100369)

[1.20 Точка входа 14](#_Toc59100370)

[1.21 Препроцессор 14](#_Toc59100371)

[1.22 Соглашения о вызовах 14](#_Toc59100372)

[1.23 Объектный код 14](#_Toc59100373)

[1.24 Классификация сообщений транслятора 14](#_Toc59100374)

[1.25 Контрольный пример 14](#_Toc59100375)

[2 Структура транслятора 15](#_Toc59100376)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 15](#_Toc59100377)

[2.2 Перечень входных параметров транслятора 16](#_Toc59100378)

[2.3 Перечень протоколов, формируемых транслятором и их содержимое 17](#_Toc59100379)

[3 Разработка лексического анализатора 18](#_Toc59100380)

[3.1 Структура лексического анализатора 18](#_Toc59100381)

[3.2. Контроль входных символов 19](#_Toc59100382)

[3.3 Удаление избыточных символов 19](#_Toc59100383)

[3.4 Перечень ключевых слов 20](#_Toc59100384)

[3.5 Основные структуры данных 21](#_Toc59100385)

[3.6 Принцип обработки ошибок 23](#_Toc59100386)

[3.7 Структура и перечень сообщений лексического анализатора 24](#_Toc59100387)

[3.8 Параметры лексического анализатора 24](#_Toc59100388)

[3.9 Алгоритм лексического анализа 24](#_Toc59100389)

[3.10 Контрольный пример 24](#_Toc59100390)

[4. Разработка синтаксического анализатора 25](#_Toc59100391)

[4.1 Структура синтаксического анализатора 25](#_Toc59100392)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 25](#_Toc59100393)

[4.3 Построение конечного магазинного автомата 26](#_Toc59100394)

[4.4 Основные структуры данных 27](#_Toc59100395)

[4.5 Описание алгоритма синтаксического разбора 27](#_Toc59100396)

[4.6 Структура и перечень сообщений синтаксического анализатора 28](#_Toc59100397)

[4.7. Параметры синтаксического анализатора и режимы его работы 28](#_Toc59100398)

[4.8. Принцип обработки ошибок 28](#_Toc59100399)

[4.9. Контрольный пример 29](#_Toc59100400)

[5 Разработка семантического анализатора 30](#_Toc59100401)

[5.1 Структура семантического анализатора 30](#_Toc59100402)

[5.2 Функции семантического анализатора 30](#_Toc59100403)

[5.3 Структура и перечень сообщений семантического анализатора 30](#_Toc59100404)

[5.4 Принцип обработки ошибок 31](#_Toc59100405)

[5.5 Контрольный пример 31](#_Toc59100406)

[6. Вычисление выражений 32](#_Toc59100407)

[6.1 Выражения, допускаемые языком 32](#_Toc59100408)

[6.2 Польская запись и принцип её построения 32](#_Toc59100409)

[6.3 Программная реализация обработки выражений 32](#_Toc59100410)

[6.4 Контрольный пример 33](#_Toc59100411)

[7. Генерация кода 34](#_Toc59100412)

[7.1 Структура генератора кода 34](#_Toc59100413)

[7.2 Представление типов данных в оперативной памяти 34](#_Toc59100414)

[7.3 Статическая библиотека 35](#_Toc59100415)

[7.4 Особенности алгоритма генерации кода 35](#_Toc59100416)

[7.5 Входные параметры генератора кода 35](#_Toc59100417)

[7.6 Контрольный пример 36](#_Toc59100418)

[8. Тестирование транслятора 37](#_Toc59100419)

[8.1 Тестирование проверки на допустимость символов 37](#_Toc59100420)

[8.2 Тестирование лексического анализатора 37](#_Toc59100421)

[8.3 Тестирование синтаксического анализатора 38](#_Toc59100422)

[8.4 Тестирование семантического анализатора 39](#_Toc59100423)

[Заключение 40](#_Toc59100424)

[Список использованных источников 42](#_Toc59100425)

[Приложение А 43](#_Toc59100426)

[Приложение Б 44](#_Toc59100427)

[Приложение В 45](#_Toc59100428)

[Приложение Г 49](#_Toc59100429)

[Приложение Д 51](#_Toc59100430)

# **Введение**

В данном курсовом проекте поставлена задача разработки собственного языка программирования и транслятора для него. Название языка – YMA-2020. Написание транслятора будет осуществляться на языке C++, при этом код на языке YMA-2020 будет транслироваться в язык ассемблера.

Задание на курсовой проект можно разделить на следующие задачи:

1. Разработка спецификации языка YMA-2020;
2. Разработка лексического анализатора;
3. Разработка синтаксического анализатора;
4. Разработка семантического анализатора;
5. Разбор арифметических выражений;
6. Разработка генератора кода;
7. Тестирование транслятора.

Глава 1. Спецификация языка программирования

* 1. Характеристика языка программирования

Язык программирования YMA-2020 является процедурным, строго типизируемым, транслируемым на язык assembler

* 1. Алфавит языка

Алфавит YMA-2020 состоит из символов, приведённых на Рисунке 1.1

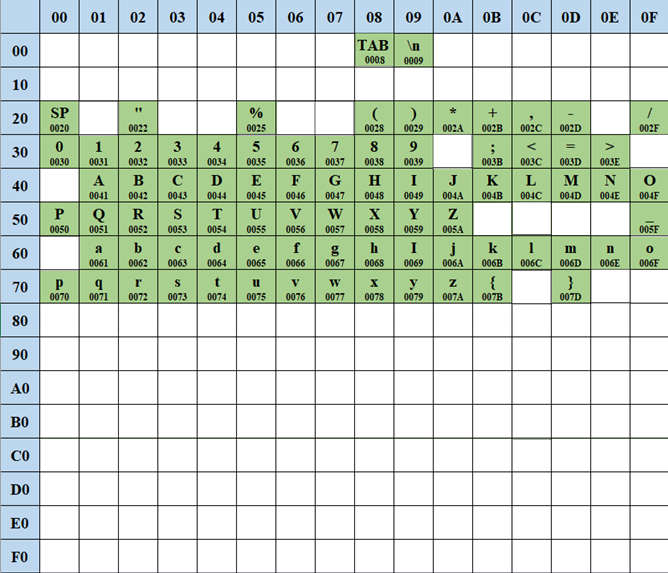


Рисунок 1.1 – Символы, разрешенные к использованию

* 1. Применяемые сепараторы

Сепараторы служат для разделения операций языка. Сепараторы, применяемые в языке программирования YMA-2020, приведены в таблице 1.1.

Таблица 1.1 – Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение сепаратора |
| ; | Разделитель инструкций |
| { } | Программный блок |
| ( ) | Параметры, приоритетность операций в выражениях |

Продолжение таблицы 1.1 – Применяемые сепараторы

|  |  |
| --- | --- |
| пробел | Допускается везде, кроме идентификаторов и ключевых слов. Служит для их разделения |
| , | Разделитель параметров в функции |

* 1. **Применяемые кодировки**

Кодировка, используемая для написания программ на языке YMA-2020 - стандартная кодировка Windows-1251, представленная на Рисунке 1.2.



Рисунок 1.2

* 1. Типы данных

Допускается использование фундаментальных типов данных определенных в таблице 1.2. Пользовательские типы данных не поддерживаются.

Таблица 1.2 – Фундаментальные типы данных

|  |  |
| --- | --- |
| Тип данных | Описание |
| **int** | Фундаментальный тип данных. Предусмотрен для объявления целочисленных данных (4 байта).  Максимальное значение: 2147483647.  Минимальное значение: -2147483648.  Автоматически инициализируется нулевым значением.  Возможные **арифметические** операции:  **+** (бинарный) – операция сложения;  - (бинарный) – операция вычитания;  **\*** (бинарный) – операция произведения;  **/** (бинарный) – операция деления;  **%** (бинарный) – операция деления нацело;  **=** (бинарный) – оператор присваивания.  Возможные варианты присваивания:  Присваивание целочисленному идентификатору значения другого целочисленного идентификатора, целочисленного литерала или значения целочисленной функции.  Возможные **логические** операции:  **>** (бинарный) – оператор «больше»;  **<** (бинарный) – оператор «меньше»; |
| **str** | Фундаментальный тип данных. Предусмотрен для объявления строк. (1 символ – 1 байт).  Автоматическая инициализация строкой нулевой длины. Максимальное количество символов в строке – 255.  Первый байт – длина строки.  Возможные операции:  **=** (бинарный) – оператор присваивания.  Возможные варианты присваивания:  Присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций. |

* 1. **Преобразование типов данных**

Преобразование типов данных в языке YMA-2020 не предусмотрено. Попытка преобразования типов данных приведет к семантической ошибке.

**1.7 Идентификаторы**

Идентификаторы могут выступать в качестве имен функций, параметров, переменных. Зарезервированные идентификаторы не предусмотрены. Идентификаторы не должны совпадать с ключевыми словами.

Имя идентификатора составляется по следующим правилам:

* состоит из символов [a..z] и [A..Z] и ‘ ’;
* длина идентификатора не должна превышать 20 символов. При превышении максимально допустимой длины применяется усечение.

**1.8 Литералы**

Предусмотрены целочисленные и строковые литералы. Все литералы являются rvalue. Правила записи приведены в Таблице 1.3.

Таблица 1.3 – Правила записи литералов

|  |  |
| --- | --- |
| Тип литерала | Описание литерала |
| Целочисленный | YMA-2020 поддерживает представление литералов в следующих системах счисления:  двоичная: [0, 1], начинается с префикса ‘b’;  восьмеричная: [0..7], начинается с префикса ‘o’;  десятичная: [0..9], без префикса;  шестнадцатеричная: [0..9], [A..F], начинается с префикса ‘h’.  При выходе за пределы будет произведен сдвиг. |
| Строковый | Символы, заключенные в “…” (двойные кавычки), число которых не превышает 255. |

**1.9** **Объявления данных и область видимости**

Для объявления переменной указывается тип данных, далее используется ключевое слово **var**, после чего имя идентификатора.

Для объявления функций используется ключевое слово **function**, перед которым указывается тип функции, а после – идентификатор.

Область видимости реализована сверху-вниз. Все функции и процедуры имеют глобальную область видимости и могут быть объявлены только в глобальной области видимости. Переменные не могут быть глобальными. Любые идентификаторы должны быть объявлены до их использования.

**1.10 Инициализация данных**

Объектами-инициализаторами могут быть только идентификаторы или литералы. При объявлении предусмотрены значения по умолчанию: значение 0 для типа **int** и строка длины 0 (“”) для типа **str**.

**1.11 Инструкции языка**

В языке программирования YMA-2020 применяются инструкции, представленные в Таблице 1.4.

Таблица 1.4 – Инструкции языка

|  |  |
| --- | --- |
| Инструкция языка | Синтаксис |
| Объявление переменной | <тип данных> var <идентификатор>; |
| Присваивание | <идентификатор> = <литерал>|<выражение>|<идентификатор>; |
| Вызов функций | <идентификатор функции> ([параметр 1] [, параметр 2]); |
| Вывод данных в консоль | write <литерал>|<идентификатор>; |
| Возврат из функции | return <литерал>|<идентификатор>; |

**1.12 Операции языка**

Язык программирования YMA-2020 может выполнять арифметические операции, представленные в таблице 1.5.

Таблица 1.5. Приоритетности операций языка программирования YMA-2020

|  |  |  |
| --- | --- | --- |
| Операция | Арифметическое назначение | Приоритетность операции |
| () | Приоритетность операций | 0 |
| + | Сложение | 1 |
| - | Вычитание | 1 |
| \* | Умножение | 2 |
| / | Деление | 2 |
| % | Взятие остатка от деления | 2 |
| > | Сравнение чисел | 3 |
| < | Сравнение чисел | 3 |

**1.13 Выражения и их вычисления**

Выражения относятся только к целому типу.

Предусмотрены следующие правила составления выражений:

* выражения записываются до ввода сепаратора ‘;’;
* выражение может содержать вызов функций;
* рассматриваются слева направо;
* для изменения приоритета операция используются круглые скобки ();
* реализация выражений происходит с помощью обратной польской записи.

**1.14 Программные конструкции языка**

Программные конструкции представлены в Таблице 1.7.

Таблица 1.6 – программные конструкции языка

|  |  |
| --- | --- |
| Конструкция | Представление в языке |
| Главная функция | main  {  <инструкции языка>  } |
| Функция | <тип данных> function <идентификатор>([<тип данных> param <идентификатор>][, <тип данных> param <идентификатор>])  {  <инструкции языка>  return <идентификатор>|<литерал>  } |
| Цикл | repeat(<идентификатор1>|<литерал>  <логический оператор><идентификатор2>|<литерал>)  then{<инструкции языка>}  else{<инструкции языка>} |

**1.15 Область видимости идентификаторов**

Все идентификаторы, объявленные внутри функции, являются локальными. Параметры видны только внутри функции, в которой объявлены.

**1.16 Семантические проверки**

Перечень семантических проверок, предусмотренных языком, приведен в Таблице 1.7.

Таблица 1.7 - Перечень семантических проверок

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Наличие функции main |
| 2 | Усечение слишком длинных идентификаторов до 20 символов |
| 3 | Сначала осуществляется проверка на ключевые слова, а затем на  идентификатор. Не допускаются идентификаторы, совпадающие с  ключевыми словами |
| 4 | Нет повторяющихся наименований функций |
| 5 | Нет повторяющихся объявлений идентификаторов |
| 6 | Предварительное объявление, применяемых функций |
| 7 | Предварительное объявление, применяемых идентификаторов. |
| 8 | Соответствие типов формальных и фактических параметров при вызове  функций |
| 9 | Усечение слишком длинного значения str-литерала |
| 10 | Округление слишком большого значения int-литерала |
| 11 | Если ошибка возникает на этапе лексического анализа, синтаксический  анализ не выполняется |
| 12 | Если 3 подряд фразы не разобраны, то работа транслятора останавливается |
| 13 | При возникновении ошибки в процессе синтаксического анализа,  ошибочная фраза игнорируется (предполагается, что ее нет) и  осуществляется попытка разбора следующей фразы. Граница фразы –  точка с запятой. |

**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, параметры функций. Локальная область видимости в исходном коде определяется за счет использования переменной, хранящей имя родительского блока, что и обуславливает их локальность на уровне ассемблерного кода, все глобальные переменные имеют имя родительского блока “global”.

**1.18 Стандартные библиотеки и их состав**

В языке YMA-2020 присутствует стандартная статическая библиотека:

StandLib, подключается автоматически при трансляции кода в ассемблер;

Функции стандартной библиотеки и их описание представлено в таблице 1.8.

Таблица 1.8 - Библиотека StandartLib языка YMA-2020

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| length\_str | integer | str parm1 - строка | Функция вычисляет длину строки parm1 |
| copy\_str | str\* parm1 | str parm1 – строка  str parm2 – строка | Функция выполняет копирование строки parm2 в строку parm1 |
| write | отсутствует | integer <идентификатор> | <целочисленный литерал> | Функция выводит на консоль <выражение> |
| write | отсутствует | str <идентификатор> | <строковый литерал> | Функция выводит на консоль <выражение> |

**1.19 Ввод и вывод данных**

Ввод данных в языке YMA-2020 не предусмотрен. Вывод данных осуществляется с помощью ключевых слова “write”.

## 1.20 Точка входа

Точкой входа в программе является ключевое слово “main”. Точка входа не может отсутствовать или быть переопределена.

**1.21 Препроцессор**

Препроцессор в языке программирования YMA-2020 не предусмотрен.

**1.22 Соглашения о вызовах**

Используется соглашение stdcall, то есть все параметры передаются в стек справа налево, память освобождает вызываемым кодом.

**1.23 Объектный код**

Язык YMA-2020 транслируется в ассемблер.

**1.24 Классификация сообщений транслятора**

Классификация ошибок представлена в таблице 1.9.

Таблица 1.9 Классификация ошибок

|  |  |
| --- | --- |
| Номера ошибок | Характеристика |
| 0 – 99 | Системные ошибки |
| 100 – 104 | Ошибки входных параметров |
| 105 – 109 | Ошибки при открытии файла |
| 110 – 119 | Ошибки при чтении файла |
| 120 – 140 | Ошибки лексического анализа |
| 600 – 610 | Ошибки синтаксического анализа |
| 700 – 720 | Ошибки семантического анализа |

**1.25 Контрольный пример**

Контрольный пример демонстрирует главные особенности языка YMA-2020: 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 YMA-2020 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, используется 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 YMA-2020 приведена на рисунке 1.



Рисунок 2.1 Структура транслятора языка программирования YMA-2020

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, преобразующий 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 YMA-2020

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке YMA-2020 , имеющий расширение .txt | Не предусмотрено |
| -log:<путь к log-файлу> | Файл журнала для вывода протоколов работы программы. | Значение по умолчанию:  <имя in-файла>.log |
| -out:<путь к out-файлу> | Выходной файл – результат работы транслятора. Содержит исходный код на языке асемблера. | Значение по умолчанию:  <имя in-файла>.asm |

## 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 YMA-2020

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования YMA-2020. 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, заданный параметром "-out:" | 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.

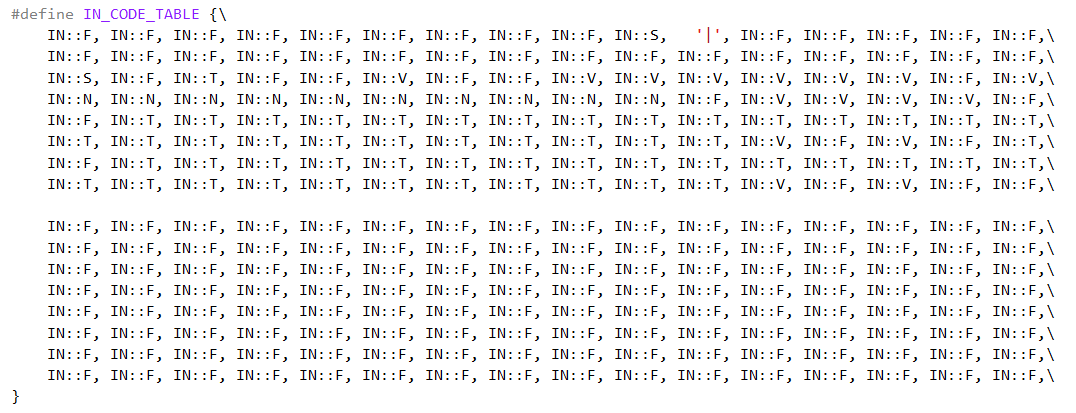
Рисунок 3.2. Таблица контроля входных символов

Таблица 3.1 Соответствие символов и их значений в таблице

|  |  |
| --- | --- |
| Значение в таблице входных символов | Символы |
| Разрешенный | T |
| Запрещенный | F |
| Игнорируемый | I |
| Пробел, табуляция | S |
| Операторы | V |
| Цифры | N |

## 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 |
| int, string | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 20 символов. |
| Литерал | l | Литерал любого доступного типа. |
| function | f | Объявление функции. |
| var | v | Объявление переменной |
| param | p | Объявление параметра функции |
| return | r | Выход из функции/процедуры. |
| main | m | Главная функция. |
| write | w | Вывод данных |
| if | ? | Уловный оператор. |
| then | : | Истинная ветвь условного оператора. |
| else | ! | Ложная ветвь условного оператора. |
| pereat | ~ | Указывает на начало тела цикла. |
| ; | ; | 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. |
| >  < | >  < | 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.

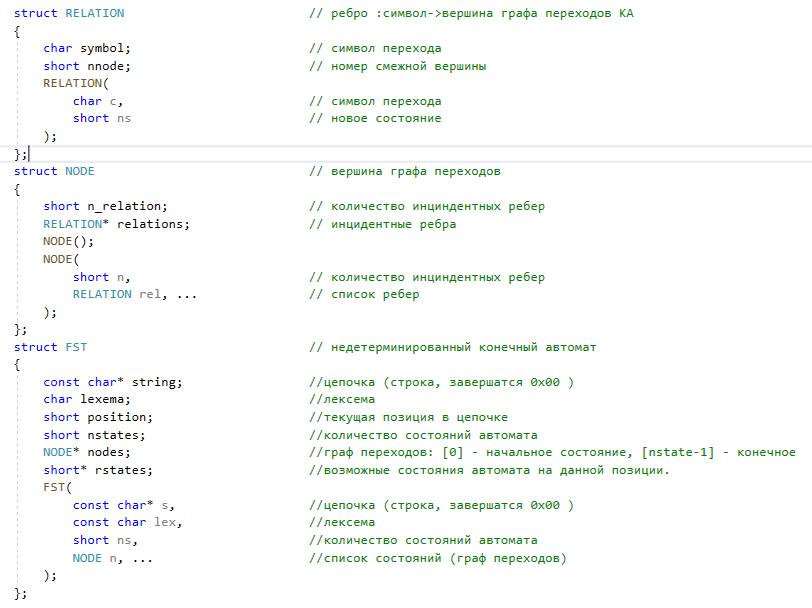


Рисунок 3.3 Структура конечного автомата

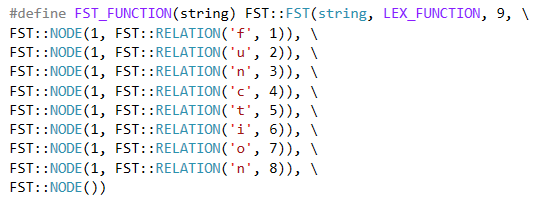


Рисунок 3.4 Пример реализации графа конечного автомата для токена function

## 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, знак оператора (sign), номер строки в исходном коде (sn), номер токена (st), 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 3.5. Код C++ со структурой таблицы идентификаторов представлен на рисунке 3.6.

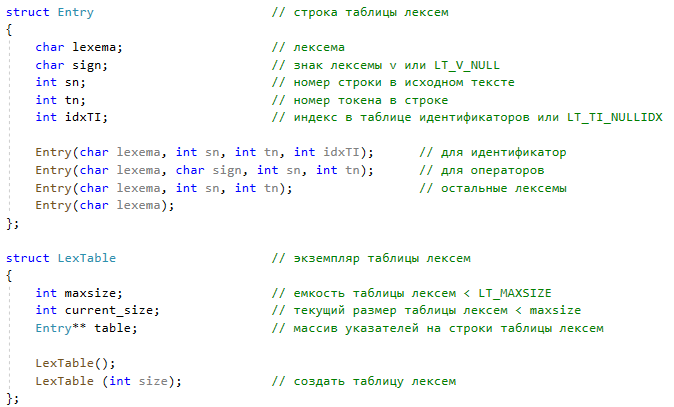


Рисунок 3.5 Структура таблицы лексем

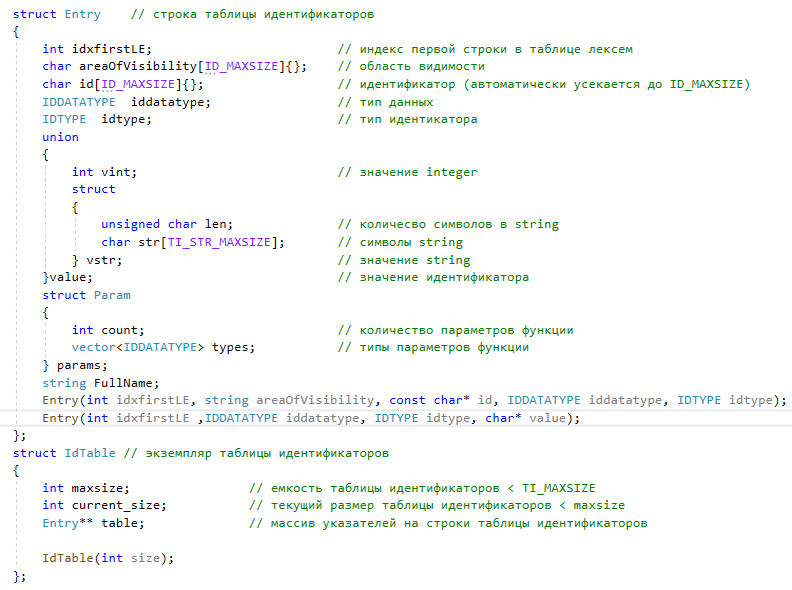


Рисунок 3.6 Структура таблицы идентификаторов

## 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 выбрасывает исключение – работа программы останавливается. Перечень сообщений представлен на рисунке 3.7.

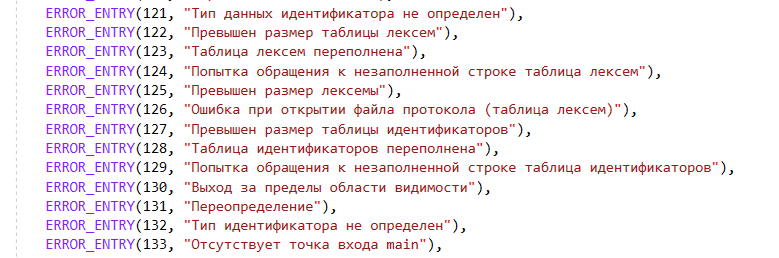


Рисунок 3.7 - Сообщения лексического анализатора

## 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.

## 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

1. Проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;
2. Для выделенной части входного потока выполняется функция распознавания лексемы;
3. При успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
4. Формирует протокол работы;
5. 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 «**str**» представлен на рисунке 3.2, где S0 – начальное, а S3 – конечное состояние автомата.

S T T R

Рисунок 3.2 Пример графа переходов для цепочки **str**

## 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 YMA-2020 используется контекстно-свободная грамматика , 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  имеют вид:

, где ; (или , или );

, где — начальный символ, при этом если такое правило существует, то нетерминал  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-> m{N}S  S-> tfi(F){NrU;}S  S-> λ | Стартовые правила, описывающее общую структуру программы |
| N | N-> tY;N  N-> i=E;N  N-> wU;N  N-> ~K{N}N  N-> ?KJN  N-> λ | Правила для операторов |
| U | U-> l  U-> i | Только литерал или идентификатор |
| J | J-> :{N}  J-> :{N}!{N} | Правила условного перехода |
| Y | Y-> vi | Правило определения переменной |
| K | K-> (U<U)  K-> (U>U) | Правила определения условия перехода |
| E | E-> iM  E-> lM  E-> (E)M  E-> i(W)M | Правила выражений |
| M | M-> #E  M-> λ | Правила арифметический операторов |
| F | F-> tP  F-> tP,F | Правила определения параметров функции |
| P | P-> pi | Правило определения параметров функции |
| W | W-> l  W-> i  W-> l,W  W-> i,W | Правила вызова функции с параметрами |

## 4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку. Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $) |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
|  | Начальное состояние магазина автомата | Символ маркера дна стека $ |
|  | 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 YMA-2020. 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.

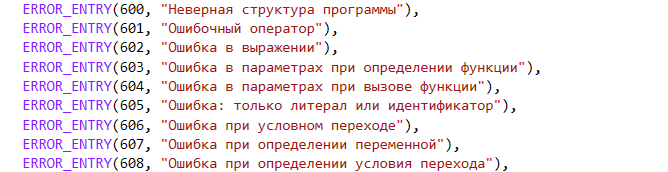


Рисунок 4.3 - Сообщения синтаксического анализатора

## 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 находятся более трёх ошибок, то анализ останавливается.

## 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.

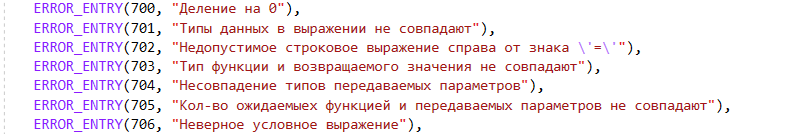


Рисунок 5.2 – Перечень сообщений семантического анализатора

## 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.

## 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

|  |  |
| --- | --- |
| Исходный код | Текст сообщения |
| main{  int var x;  x = 1 + “test1”;  } | Ошибка на этапе Семантического анализатора  Ошибка 701: Типы данных в выражении не совпадают, строка 3 |
| main{  int var x;  x = 5 / 0;  } | Ошибка на этапе Семантического анализатора  Ошибка 700: Деление на 0, строка 3, лексема 5 |
| main{  str var y;  y = "test2" + “test3”;  } | Ошибка на этапе Семантического анализатора  Ошибка 702: Недопустимое строковое выражение справа от знака '=', строка 3 |

# **6. Вычисление выражений**

## 6.1 Выражения, допускаемые языком

В языке YMA-2020 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 |
| % | 4 |
| / | 4 |
| \* | 4 |
| - | 3 |
| + | 3 |
| , | 2 |
| ( ) | 1 |

## 6.2 Польская запись и принцип её построения

Все выражения языка YMA-2020 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.

Таблица 6.2. Преобразование выражений к ПОЛИЗ

|  |  |
| --- | --- |
| Выражение | Обратная польская запись для выражения |
| l | l |
| i#l | il# |
| i#l | il# |

# **7. Генерация кода**

## 7.1 Структура генератора кода

В языке YMA-2020 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 YMA-2020 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 YMA-2020 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка YMA-2020 и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке YMA-2020 | Тип идентификатора на языке ассемблера | Пояснение |
| int | sdword | Хранит целочисленный тип данных. |
| str | dword | Хранит указатель на начало строки. Строка должна завершаться нулевым символом. |

## 7.3 Статическая библиотека

В языке YMA-2020 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 |
| void write\_str(char\* str) | Вывод на консоль строки str |
| void write\_int(int num) | Вывод на консоль целочисленной переменной num |
| int length\_str(char\* str) | Вычисление длины строки |
| char\* copy\_str(char\* str1, char\* str2) | Копирование строки str2 в строку str1 |

## 7.4 Особенности алгоритма генерации кода

В языке YMA-2020 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 YMA-2020. 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.

вывод все нечетных двузначных чисел

11

13

15

17

19

…

85

87

89

91

93

95

97

99

Рисунок 7.2 Результат работы программы на языке YMA-2020

# **8. Тестирование транслятора**

## 8.1 Тестирование проверки на допустимость символов

В языке YMA-2020 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 |
| main { ~ } | Ошибка при чтении из входного файла  Ошибка 110: Недопустимый символ в исходном файле (-in), строка 1, столбец 8 |
| main  {  write "test  } | Ошибка при чтении из входного файла  Ошибка 111: Нет закрывающей кавычки (-in), строка 3, столбец 15 |

## 8.2 Тестирование лексического анализатора

На этапе лексического анализа в языке YMA-2020 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 |
| main  {  test123  } | Ошибка на этапе лексического анализатора  Ошибка 120: Нераспознанная лексема, строка 3, лексема 1 |
| main  {  var test;  } | Ошибка на этапе лексического анализатора  Ошибка 121: Тип данных идентификатора не определен, строка 3, лексема 2 |
| main  {  int var test;  int var test;  } | Ошибка на этапе лексического анализатора  Ошибка 131: Переопределение, строка 4, лексема 3 |
| main  {  int test;  } | Ошибка на этапе лексического анализатора  Ошибка 132: Тип идентификатора не определен, строка 3, лексема 2 |
| int function FindMaxLen(str param x, str param y){} | Ошибка 133: Отсутствует точка входа main |

Продолжение таблицы 8.2 - Тестирование лексического анализатора

|  |  |
| --- | --- |
| main {}  main  {  int var test;  } | Ошибка 134: Обнаружено несколько точек входа main |

## 8.3 Тестирование синтаксического анализатора

На этапе синтаксического анализа в языке YMA-2020 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 |
| function main  {  int var test;  } | 600: строка 1, Неверная структура программы |
| main  {  return 1;  } | 601: строка 4, Ошибочный оператор |
| main  {  int var test;  test = 1 + - 2;  } | 602: строка 4, Ошибка в выражении |
| int function FindMaxLen(str param x, )  {  }  main {} | 603: строка 1, Ошибка в параметрах при определении функции |
| int function FindMax(int param x)  { return 1;}  main  {  int var res;  res = FindMax(1, )  } | 604: строка 6, Ошибка в параметрах при вызове функции |

Продолжение таблицы 8.3 - Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| main  {  if (1 > 0)  then  } | 606: строка 4, Ошибка при условном переходе |
| main  {  int str var test;  } | 607: строка 3, Ошибка при определении переменной |
| main  {  if {1 > 0)  then {}  } | 608: строка 3, Ошибка при определении условия перехода |

## 8.4 Тестирование семантического анализатора

Семантический анализ в языке YMA-2020 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 | Текст сообщения |
| main{  int var x;  x = 1 + “test1”;  } | Ошибка на этапе Семантического анализатора  Ошибка 701: Типы данных в выражении не совпадают, строка 3 |
| main{  int var x;  x = 5 / 0;  } | Ошибка на этапе Семантического анализатора  Ошибка 700: Деление на 0, строка 3, лексема 5 |
| main{  str var y;  y = "test2" + “test3”;  } | Ошибка на этапе Семантического анализатора  Ошибка 702: Недопустимое строковое выражение справа от знака '=', строка 3 |
| str function Test(int param x)  {  return 0;  }  main {} | Ошибка на этапе Семантического анализатора  Ошибка 703: Тип функции и возвращаемое значение не совпадают, строка 3 |

Продолжение таблицы 8.4 - Тестирование семантического анализатора

|  |  |
| --- | --- |
| int function Test(int param x)  {return 0;}  main  {  int var temp;  temp = Test("test");  } | Ошибка на этапе Семантического анализатора  Ошибка 704: Несовпадение типов передаваемых параметров, строка 6, лексема 5 |
| int function Test(int param x)  {return 0;}  main  {  int var temp;  temp = Test(1, 2);  } | Ошибка на этапе Семантического анализатора  Ошибка 705: Кол-во ожидаемых функцией и передаваемых параметров не совпадают, строка 6 |
| main  {  if ("test1" > "test2")  then{}  } | Ошибка на этапе Семантического анализатора  Ошибка 706: Неверное условное выражение, строка 3 |

# **Заключение**

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования YMA-2020 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

Сформулирована спецификация языка YMA-2020;

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

Окончательная версия языка YMA-2020 включает:

1. 2 типа данных;
2. Поддержка оператора вывода;
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. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768с.

2. Ахо, А. Теория синтаксического анализа, перевода и компиляции /А. Ахо, Дж. Ульман. – Москва : Мир, 1998. – Т. 2 : Компиляция. - 487 с.

3. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.

4. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c.

5. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с

# **Приложение А**

Листинг 1 – исходный код программы на языке YMA-2020

main

{

write "вывод все нечетных двузначных чисел";

int var i; i = b1010;

repeat(i < h64)

{

int var result;

result = i % 2;

if(result > 0)

then

{

write i;

}

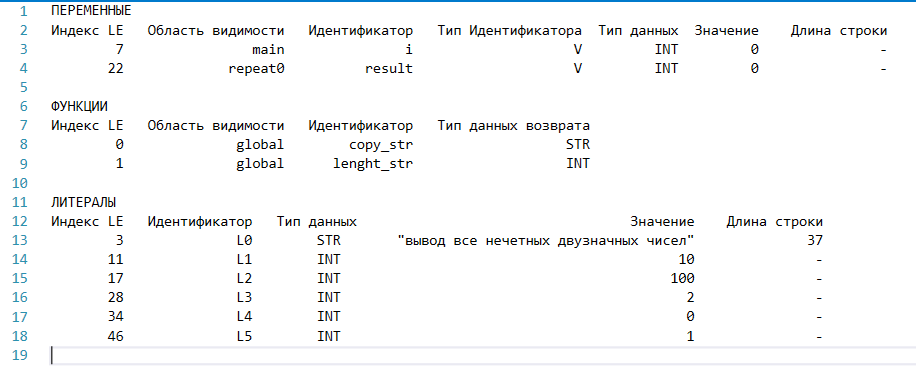
i = i + 1;

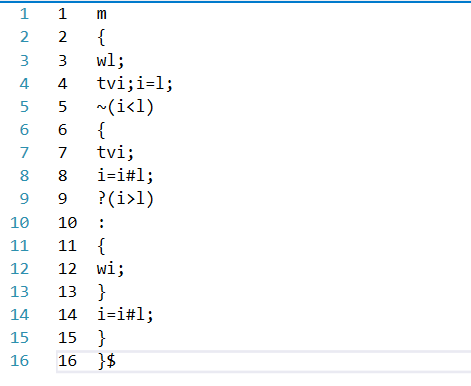
}

}

# **Приложение Б**

Листинг 1. Таблица идентификаторов контрольного примера



Листинг 2. Таблица лексем после контрольного прим

# **Приложение В**

Листинг 1. Грамматика языка YMA-2020

Greibach greibach(NS('S'), TS('$'), 11,

Rule(NS('S'), GRB\_ERROR\_SERIES + 0, //Неверная структура программы

3, // m{N}S | tfi(F){NrU;}S | пустой переход

Rule::Chain(5, TS('m'), TS('{'), NS('N'), TS('}'), NS('S')),

Rule::Chain(13, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('U'), TS(';'), TS('}'), NS('S')),

Rule::Chain()

),

Rule(NS('N'), GRB\_ERROR\_SERIES + 1, //ошибочный оператор

6, // tY;N | i=E;N | wU;N | ~K{N}N | ?KJN | пустой переход

Rule::Chain(4, TS('t'), NS('Y'), TS(';'), NS('N')),

Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),

Rule::Chain(4, TS('w'), NS('E'), TS(';'), NS('N')),

Rule::Chain(6, TS('~'), NS('K'), TS('{'), NS('N'), TS('}'), NS('N')),

Rule::Chain(4, TS('?'), NS('K'), NS('J'), NS('N')),

Rule::Chain()

),

Rule(NS('U'), GRB\_ERROR\_SERIES + 5, // Только литерал или идентификатор

2, // l | i

Rule::Chain(1, TS('l')),

Rule::Chain(1, TS('i'))

),

Rule(NS('J'), GRB\_ERROR\_SERIES + 6, // Ошибка при условном переходе

2, // :{N} | :{N}!{N}

Rule::Chain(4, TS(':'), TS('{'), NS('N'), TS('}')),

Rule::Chain(8, TS(':'), TS('{'), NS('N'), TS('}'), TS('!'), TS('{'), NS('N'), TS('}'))

),

Rule(NS('Y'), GRB\_ERROR\_SERIES + 7, // Ошибка при определении переменной

1, // vi

Rule::Chain(2, TS('v'), TS('i'))

),

Rule(NS('K'), GRB\_ERROR\_SERIES + 8, // Ошибка при определении условия перехода

2, // (U<U) | (U>U)

Rule::Chain(5, TS('('), NS('U'), TS('<'), NS('U'), TS(')')),

Rule::Chain(5, TS('('), NS('U'), TS('>'), NS('U'), TS(')'))

),

Rule(NS('E'), GRB\_ERROR\_SERIES + 2, // Ошибка в выражении

4, // iM | lM | (E)M | i(W)M

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(NS('M'), GRB\_ERROR\_SERIES + 2, // Ошибка в выражении

2, // #E | пустой переход

Rule::Chain(2, TS('#'), NS('E')),

Rule::Chain()

),

Rule(NS('F'), GRB\_ERROR\_SERIES + 3, // Ошибка в параметрах при определении функции

2, // tP | tP,F

Rule::Chain(2, TS('t'), NS('P')),

Rule::Chain(4, TS('t'), NS('P'), TS(','), NS('F'))

),

Rule(NS('P'), GRB\_ERROR\_SERIES + 3, // Ошибка в параметрах при определении функции

1, // pi

Rule::Chain(2, TS('p'), TS('i'))

),

Rule(NS('W'), GRB\_ERROR\_SERIES + 4, // Ошибка в параметрах при вызове функции

4, // i | l | i,W | l,W

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'))

)

);

Листинг 2. Структура магазинного автомата

struct MFST // магазинный автомат

{

enum RC\_STEP //код возврата функции step

{

NS\_OK, // найдено правило и цепочка, цепочка записана в стек

NS\_NORULE, // не найдено правило грамматики (ошибка в грамматике)

NS\_NORULECHAIN, // не найдена походящая цепочка правила (ошибка в исходном коде)

NS\_ERROR, // неизвесный нетерминальный символ грамматики

TS\_OK, // тек. символ ленты == вершине стека, продвинулась лента, pop стека

TS\_NOK, // тек. символ ленты != вершине стека, восстановленно состояние

LENTA\_END, // теущая позиция ленты >= lenta\_size

SURPRISE // неожиданный код возврата (ошибка в step)

};

struct MFST\_Diagnosis // диагностика

{

short posInLent; // позиция на ленте

RC\_STEP rc\_step; // код завершения шага

short ruleNum; // номер правила

short nrule\_chain; // номер цепочки правила

MFST\_Diagnosis();

MFST\_Diagnosis(short posInLent, RC\_STEP rc\_step, short ruleNum, short ruleChainNum);

} diagnosis[MFST\_DIAGN\_NUMBER]; // последние самые глубокие сообщения

GRBALPHABET\* lenta; // перекодированная (TS/NS) лента (из LEX)

short currentPosInLent; // текущая позиция на ленте

short currentRule; // номер текущего правила

short currentRuleChain; // номер текущей цепочки, текущего правила

short lenta\_size; // размер ленты

GRB::Greibach grebach; // грамматика Грейбах

LT::LexTable lexTable;

MFSTSTSTACK st; // стек автомата

std::stack<MFSTState> storestate; // стек для сохранения состояний

MFST();

MFST(const LT::LexTable& lexTable,GRB::Greibach grebach);

char\* getCSt(char\* buf); //получить содержиое стека

char\* getCLenta(char\* buf, short pos, short n = 25); //лента: n символов, начиная с pos

char\* getDiagnosis(short n, char\* buf); //получить n-ую строку диагностики или '\0'

bool savestate(std::ostream\* stream); //сохранить состояние автомата

bool resetstate(std::ostream\* stream); //восстановить состояние автомата

bool push\_chain(GRB::Rule::Chain chain);

RC\_STEP step(std::ostream\* stream); //выполнить шаг автомата

bool start(std::ostream\* stream); //запустить автомат

bool savedDiagnosis(RC\_STEP prc\_step);

void printRules(std::ostream\* stream); //вывести последовательность правил

struct Deducation // вывод

{

short stepsCount; // количество шагов в выводе

short\* nRules; // номер правила грамматики

short\* nChainsOfRules; // номер цепочек правил грамматики

Deducation()

{

this->stepsCount = 0;

this->nRules = 0;

this->nChainsOfRules = 0;

}

}deducation;

bool saveoutputTree(); // сохранить дерево вывода

};

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()

{

size = 0;

startN = 0;

stbottomT = 0;

rules = 0;

};

Greibach(GRBALPHABET pstartN, // стартовый символ

GRBALPHABET pstbottomT, // 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->m{N}S m{wl;tvi;i=l;~(i<l){tvi;i S$

1 : SAVESTATE: 1

1 : m{wl;tvi;i=l;~(i<l){tvi;i m{N}S$

2 : {wl;tvi;i=l;~(i<l){tvi;i= {N}S$

3 : wl;tvi;i=l;~(i<l){tvi;i=i N}S$

4 :N->wE;N wl;tvi;i=l;~(i<l){tvi;i=i N}S$

5 : SAVESTATE: 2

5 : wl;tvi;i=l;~(i<l){tvi;i=i wE;N}S$

6 : l;tvi;i=l;~(i<l){tvi;i=i# E;N}S$

7 :E->lM l;tvi;i=l;~(i<l){tvi;i=i# E;N}S$

8 : SAVESTATE: 3

8 : l;tvi;i=l;~(i<l){tvi;i=i# lM;N}S$

9 : ;tvi;i=l;~(i<l){tvi;i=i#l M;N}S$

10 :M-> ;tvi;i=l;~(i<l){tvi;i=i#l M;N}S$

11 : SAVESTATE: 4

11 : ;tvi;i=l;~(i<l){tvi;i=i#l ;N}S$

12 : tvi;i=l;~(i<l){tvi;i=i#l; N}S$

13 :N->tY;N tvi;i=l;~(i<l){tvi;i=i#l; N}S$

14 : SAVESTATE: 5

14 : tvi;i=l;~(i<l){tvi;i=i#l; tY;N}S$

15 : vi;i=l;~(i<l){tvi;i=i#l;? Y;N}S$

16 :Y->vi vi;i=l;~(i<l){tvi;i=i#l;? Y;N}S$

17 : SAVESTATE: 6

17 : vi;i=l;~(i<l){tvi;i=i#l;? vi;N}S$

18 : i;i=l;~(i<l){tvi;i=i#l;?( i;N}S$

19 : ;i=l;~(i<l){tvi;i=i#l;?(i ;N}S$

20 : i=l;~(i<l){tvi;i=i#l;?(i> N}S$

Листинг 4 (продолжение). Разбор исходного кода синтаксическим анализатором

114 : #l;}}$ M;N}N}S$

115 :M->#E #l;}}$ M;N}N}S$

116 : SAVESTATE: 32

116 : #l;}}$ #E;N}N}S$

117 : l;}}$ E;N}N}S$

118 :E->lM l;}}$ E;N}N}S$

119 : SAVESTATE: 33

119 : l;}}$ lM;N}N}S$

120 : ;}}$ M;N}N}S$

121 :M-> ;}}$ M;N}N}S$

122 : SAVESTATE: 34

122 : ;}}$ ;N}N}S$

123 : }}$ N}N}S$

124 :N-> }}$ N}N}S$

125 : SAVESTATE: 35

125 : }}$ }N}S$

126 : }$ N}S$

127 :N-> }$ N}S$

128 : SAVESTATE: 36

128 : }$ }S$

129 : $ S$

130 :S-> $ S$

131 : SAVESTATE: 37

131 : $ $

132 :

133 : LENTA\_END

134 : ------>LENTA\_END

# **Приложение Г**

Листинг 1. Программная реализация механизма преобразования в ПОЛИЗ

bool polishNotation(int lextable\_pos, LT::LexTable& lextable, IT::IdTable& idtable)

{

stack<LT::Entry\*> stk; //создаем стек для хранения временных операций

queue<LT::Entry\*> result;

bool function = false;

int quantityParm = 0;

int i = ++lextable\_pos;

for (; lextable.table[i]->lexema != LEX\_SEMICOLON && (lextable.table[i]->lexema != LEX\_RIGHTHESIS || !stk.empty()); i++)

{

switch (lextable.table[i]->lexema)

{

case LEX\_ID: //операнды

case LEX\_LITERAL:

if (idtable.table[lextable.table[i]->idxTI]->idtype == IT::IDTYPE::F)

{

quantityParm = 0;

function = true;

result.push(lextable.table[i]);

break;

}

if (function && !quantityParm)

quantityParm++;

result.push(lextable.table[i]);

break;

case LEX\_OPERATOR:

if (stk.empty() || stk.top()->lexema == LEX\_LEFTHESIS)

stk.push(lextable.table[i]);

else

{

int prioritet = priority(lextable.table[i]->sign);

if (priority(stk.top()->sign) >= prioritet)

{

result.push(stk.top());

stk.pop();

}

stk.push(lextable.table[i]);

}

break;

case LEX\_LEFTHESIS:

stk.push(lextable.table[i]);

break;

case LEX\_RIGHTHESIS:

while (stk.top()->lexema != LEX\_LEFTHESIS)

{

result.push(stk.top());

stk.pop();

}

stk.pop();

if (function)

{

result.push(new LT::Entry('@'));

result.push(new LT::Entry('0' + quantityParm));

function = false;

}

break;

case LEX\_COMMA:

if (function)

quantityParm++;

while (stk.top()->lexema != LEX\_LEFTHESIS)

{

result.push(stk.top());

stk.pop();

}

break;

case LEX\_MORE:

case LEX\_LESS:

result.push(lextable.table[i]);

break;

}

}

while (!stk.empty())

{

result.push(stk.top());

stk.pop();

}

for (int j = lextable\_pos; j < i; j++)

{

if (!result.empty())

{

lextable.table[j] = result.front();

lextable.table[j]->sn = lextable.table[j - 1]->sn;

lextable.table[j]->tn = lextable.table[j - 1]->tn + 1;

result.pop();

}

else

{

lextable.table[j] = new LT::Entry('@', lextable.table[j]->sn = lextable.table[j - 1]->sn, lextable.table[j]->tn = lextable.table[j - 1]->tn + 1);

}

}

return true;

}

# **Приложение Д**

.586 ; система команд (процессор Pentium)

.model flat, stdcall ; модель памяти, соглашение о вызовах

includelib kernel32.lib

includelib libucrt.lib

includelib StandLib.lib

ExitProcess PROTO: dword ; прототип функции для завершения процесса Windows

EXTRN copy\_str: proc

EXTRN lenght\_str: proc

EXTRN write\_int: proc

EXTRN write\_str : proc

.stack 4096

.const ; сегмент констант - литералы

L0 byte "вывод все нечетных двузначных чисел", 0

L1 sdword 10

L2 sdword 100

L3 sdword 2

L4 sdword 0

L5 sdword 1

.data ; сегмент данных - переменные и параметры

i\_main sdword 0

result\_repeat0 sdword 0

.code ; сегмент кода

;----------- MAIN ------------

main PROC

push offset L0

call write\_str

push L1

pop i\_main

cyclenext0:

mov edx, i\_main

cmp edx, L2

jg cycle0

push i\_main

push L3

pop ebx

pop eax

cdq

idiv ebx

push edx

pop result\_repeat0

mov edx, result\_repeat0

cmp edx, L4

jg true1

jmp next1

true1:

push i\_main

call write\_int

next1:

push i\_main

push L5

pop ebx

pop eax

add eax, ebx

push eax

pop i\_main

jmp cyclenext0

cycle0:

push 0

call ExitProcess

main ENDP

end main