**Содержание**

[Введение 3](#_Toc153622249)

[1. Спецификация языка программирования 4](#_Toc153622250)

[1.1 Характеристика языка программирования 4](#_Toc153622251)

[1.2 Определение алфавита языка программирования 4](#_Toc153622252)

[1.3 Применяемые сепараторы 4](#_Toc153622253)

[1.4 Применяемые кодировки 5](#_Toc153622254)

[1.5 Типы данных 5](#_Toc153622255)

[1.6 Преобразование типов данных 6](#_Toc153622256)

[1.7 Идентификаторы 7](#_Toc153622257)

[1.8 Литералы 7](#_Toc153622258)

[1.9 Объявление данных 8](#_Toc153622259)

[1.10 Инициализация данных 8](#_Toc153622260)

[1.11 Инструкции языка 8](#_Toc153622261)

[1.12 Операции языка 9](#_Toc153622262)

[1.13 Выражения и их вычисления 10](#_Toc153622263)

[1.14 Конструкции языка 10](#_Toc153622264)

[1.15 Область видимости идентификаторов 11](#_Toc153622265)

[1.16 Семантические проверки 11](#_Toc153622266)

[1.17 Распределение оперативной памяти на этапе выполнения 12](#_Toc153622267)

[1.18 Стандартная библиотека и ее состав 12](#_Toc153622268)

[1.20 Точка входа 13](#_Toc153622269)

[1.21 Препроцессор 13](#_Toc153622270)

[1.22 Соглашения о вызове 13](#_Toc153622271)

[1.23 Объектный код 13](#_Toc153622272)

[1.24 Классификация сообщений транслятора 13](#_Toc153622273)

[1.25 Контрольный пример 15](#_Toc153622274)

[2. Структура транслятора 16](#_Toc153622275)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 16](#_Toc153622276)

[2.2 Перечень входных параметров транслятора 17](#_Toc153622277)

[2.3 Протоколы, формируемые транслятором и их содержимое 17](#_Toc153622278)

[3 Разработка лексического анализатора 19](#_Toc153622279)

[3.1 Структура лексического анализатора 19](#_Toc153622280)

[3.2 Входные и выходные данные лексического анализатор 20](#_Toc153622281)

[3.3 Параметры лексического анализатора 20](#_Toc153622282)

[3.4 Алгоритм лексического анализа 20](#_Toc153622283)

[3.5 Контроль входных символов 20](#_Toc153622284)

[3.6 Удаление избыточных символов 21](#_Toc153622285)

[3.7 Перечень ключевых слов 22](#_Toc153622286)

[3.8 Основные структуры данных 24](#_Toc153622287)

[3.9 Структура и перечень сообщений лексического анализатора 25](#_Toc153622288)

[3.10 Принцип обработки ошибок 25](#_Toc153622289)

[3.11 Контрольный пример 25](#_Toc153622290)

[4. Разработка синтаксического анализатора 26](#_Toc153622291)

[4.1 Структура синтаксического анализатора 26](#_Toc153622292)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 26](#_Toc153622293)

[4.3 Построение конечного магазинного автомата 28](#_Toc153622294)

[4.4 Основные структуры данных 29](#_Toc153622295)

[4.5 Описание алгоритма синтаксического разбора 30](#_Toc153622296)

[4.6 Параметры синтаксического анализатора и режимы его работы 30](#_Toc153622297)

[4.7 Структура и перечень сообщений синтаксического анализатора 30](#_Toc153622298)

[4.8 Принцип обработки ошибок 31](#_Toc153622299)

[4.9 Контрольный пример 31](#_Toc153622300)

[5 Разработка семантического анализатора 32](#_Toc153622301)

[5.1 Структура семантического анализатора 32](#_Toc153622302)

[5.2 Функции семантического анализатора 32](#_Toc153622303)

[5.3 Структура и перечень сообщений семантического анализатора 32](#_Toc153622304)

[5.4 Принцип обработки ошибок 33](#_Toc153622305)

[5.5 Контрольный пример 33](#_Toc153622306)

[6. Вычисление выражений 34](#_Toc153622307)

[6.1 Выражения, допускаемые языком 34](#_Toc153622308)

[6.2 Польская запись и принцип её построения 34](#_Toc153622309)

[6.3 Программная реализация обработки выражений 34](#_Toc153622310)

[6.4 Контрольный пример 35](#_Toc153622311)

[7. Генерация кода 36](#_Toc153622312)

[7.1 Структура генератора кода 36](#_Toc153622313)

[7.2 Представление типов данных в оперативной памяти 36](#_Toc153622314)

[7.3 Статическая библиотека 37](#_Toc153622315)

[7.4 Особенности алгоритма генерации кода 37](#_Toc153622316)

[7.5 Параметры, управляющие генерацией кода 38](#_Toc153622317)

[7.6 Контрольный пример 38](#_Toc153622318)

[8. Тестирование транслятора 39](#_Toc153622319)

[8.1 Общие положения 39](#_Toc153622320)

[8.2 Результаты тестирования 39](#_Toc153622321)

[Заключение 42](#_Toc153622322)

[Список использованных источников 43](#_Toc153622323)

[Приложение А 44](#_Toc153622324)

[Приложение Б 45](#_Toc153622325)

[Приложение В 48](#_Toc153622326)

[Приложение Г 53](#_Toc153622327)

[Приложение Д 60](#_Toc153622328)

# Введение

Целью курсового проекта является разработка компилятора для своего языка программирования: GSA-2023. Данный язык программирования предназначен для выполнения сдвиговых, арифметических и логических операций над числами и строками.

Компилятор – это программа, задачей которой является преобразование программы, написанной на одном языке программирования, в программу на языке ассемблера.

Процесс компиляции состоит из двух ключевых этапов: анализа и генерации. Анализ включает в себя разбор исходной программы на отдельные компоненты и создание ее промежуточного представления. Генерация означает создание целевой программы на основе промежуточного представления. В данном курсовом проекте исходный код транслируется на язык ассемблера.

Компилятор GSA-2023 состоит из следующих составных частей:

* лексический анализатор;
* синтаксический анализатор;
* семантический анализатор;
* генератор исходного кода на языке ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

* разработка спецификации языка программирования;
* разработка структуры транслятора;
* разработка лексического анализатора;
* разработка синтаксического анализатора;
* преобразование выражений;
* генерация кода на язык ассемблер;
* тестирование транслятора.

В каждой главе курсового проекта будут представлены решения, соответствующие установленным задачам.

# 1. Спецификация языка программирования

## **1.1 Характеристика языка программирования**

GSA-2023 является компилируемым процедурным строго типизированным языком.

Поддерживает 2 типа данных: строковый (string) и целочисленный (byte). В стандартной библиотеке языка содержится 3 функции для работы со строками: atoi(string str)– преобразование строки в число и strcat (string str1, string str2) – конкатенация строк, length(string str1) – длина строки.

**1.2 Определение алфавита языка программирования**

В основе алфавита GSA-2023 лежит таблица символов ASCII, которая представлена на рисунке 1.1.

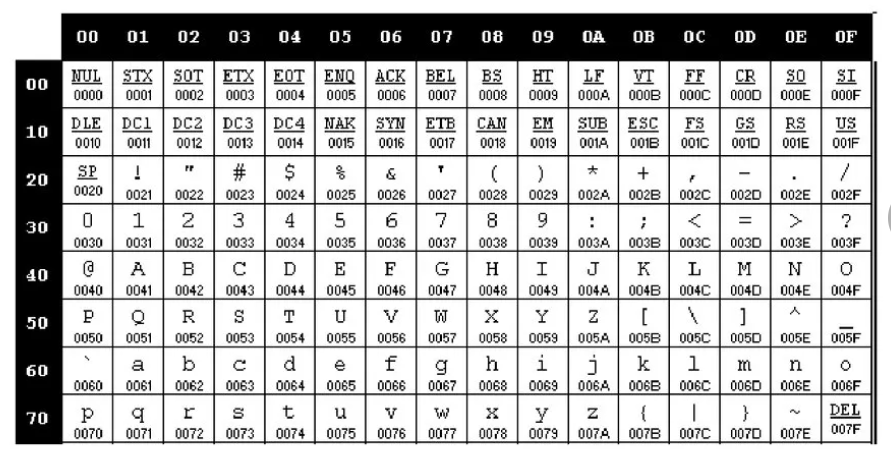


Рисунок 1.1 – Таблица кодировки ASCII

Разрешены латинские символы, символы-сепараторы, символы операций и другие.

Алфавит языка GSA-2023 состоит из следующих множеств символов:

Для записи инструкций языка используются символы: [a…z], [A…Z].

Для записи литералов (строковых и целочисленных) используют символы: [a…я], [a…z], [0…9].

В качестве сепараторов и специальных символов используются: [ ] ( ) , ; : # + - \* > < & !. “пробел”

**1.3 Применяемые сепараторы**

Сепараторы необходимы для разделения операций языка. Сепараторы, используемые в языке программирования GSA-2023, приведены в таблице 1.1.

Таблица 1.1 Применяемые сепараторы

|  |  |
| --- | --- |
| Символы | Назначение |
| **[** … **]** | Блок функции или условной конструкции/цикла |
| **(** … **)** | Блок фактических или формальных параметров функции, а также приоритет арифметических операций |
| **,** | Разделитель параметров функций |
| **#** | Символ, отделяющий условные конструкции/циклы |
| **+ - \*** | Арифметические операции |
| **> < & !** | Логические операции (операции сравнения: больше, меньше, проверка на равенство, на неравенство), используемые в условии цикла/условной конструкции. |
| **;** | Разделитель программных конструкций |
| }{ | Операторы сдвигов |
| ‘пробел’ | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |

Таким образом язык программирования GSA-2023 использует все приведённые выше символы.

**1.4 Применяемые кодировки**

При трансляции исходного кода применяется кодировка ASCII. Описание кодировки представлено в пункте 1.2.

## **1.5 Типы данных**

В языке GSA-2023 реализованы 2 фундаментальных типа данных: целочисленный и строковый, которые описываются в таблице 1.2.

Таблица 1.2 – Типы данных языка GSA-2023

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Строковый  (string) | Фундаментальный тип данных. По умолчанию инициализируется пустой строкой: ‘’.  Размер в байтах (для одного символа): 1 байт.  Допустимый диапазоны значений: от 0 до 255.  Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций. |

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Целочисленный  (byte) | Фундаментальный тип данных. Используется для работы с числовыми значениями.  Максимальное значение: 127.  Минимальное значение: -128  По умолчанию инициализируется 0.  Размер в байтах: 1.  Применяемые операции:  **+** (бинарный) – оператор сложения;  - (бинарный) – оператор вычитания;  **\*** (бинарный) – оператор умножения;  **/** (бинарный) – оператор деления;  **=** (бинарный) – оператор присваивания.  В качестве условия цикла/условного оператора поддерживаются следующие логические операции:  **>** (бинарный) – оператор «больше»;  < (бинарный) – оператор «меньше»;  ! (бинарный) – оператор проверки на неравенство. |

Окончание таблицы 1.2

В таблице представлены два типа данных: строковый (string) и целочисленный (byte). Строковый тип данных используется для работы с текстовыми значениями, имеет размер в один байт на символ и поддерживает операции присваивания, конкатенации и использование библиотечных функций. Целочисленный тип данных предназначен для работы с числовыми значениями, имеет размер в один байт и поддерживает математические операции и операторы сравнения.

**1.6 Преобразование типов данных**

В языке программирования GSA-2023 присутствует преобразование строки в число с помощью функции стандартной библиотеки atoii(string), которая возвращает значение byte.

**1.7 Идентификаторы**

Для именования функций, параметров и переменных используются идентификаторы. Не предусмотрены зарезервированные идентификаторы. Имя идентификаторов не должно совпадать с ключевыми словами языка и с именами функций стандартной библиотеки.

Имя идентификатора составляется по следующим образом:

* состоит из символов латинского алфавита [a..z];
* максимальная длина идентификатора равна 8 символам;
* идентификаторы, объявленные внутри функционального блока, получают префикс идентичный имени функции, внутри которой они объявлены;
* в случае превышения заданной длины, идентификаторы усекаются до длины, равной 16 символов;
* зарезервированные идентификаторы не предусмотрены;
* идентификаторы не должны совпадать с ключевыми словами;
* имена идентификаторов-функций не должны совпадать с именами команд ассемблера;

**1.8 Литералы**

В языке существует 2 типа литералов: целого и строкового типов. Краткое описание литералов представлено в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание литерала |
| Целочисленные литералы в десятичном представлении | Последовательность цифр 0…9 с предшествующим знаком минус или без него (знак минус не должен отделяться пробелом). Ограничения: не могут начинаться с нуля, если их значение не равно нулю. Если литерал отрицательный, то перед ним не может быть знака «-». |
| Целочисленные литералы в восьмеричном представлении | Последовательность цифр 0…7 с предшествующим символом ‘q’. Ограничения: не могут начинаться с нуля, если их значение не равно нулю. Если литерал отрицательный, то перед ним не может быть знака «-». |
| .Строковые литералы | Набор символов (от 1 до 255), заключённых в двойные кавычки. Ограничения: Внутри литерала не допускается использование символов кириллицы, одинарных и двойных кавычек. |

Литералы – числовые значения или строки, представленные в программировании. Целочисленные литералы в десятичном представлении – последовательность цифр с возможным знаком минус. Строковые литералы – набор символов в двойных кавычках. Целочисленные литералы в восьмеричном представлении - цифры 0...7 с символом 'q' в начале [1].

## **1.9 Объявление данных**

В языке программирования GSA-2023 переменная должны быть объявлена до ее использования. Для объявления переменной используется ключевое слово new, после которого необходимо указать тип данных и имя идентификатора.

Объявление с инициализацией:

new byte a;

a = 3;

Объявление с инициализацией:

new string b = “test”;

Для объявления функции необходимо использовать ключевое слово function, предварительно указав тип функции. Далее необходимо указать имя функции, список параметров и ее тело. Для объявления параметра необходимо перед именем идентификатора указать тип данных:

string function Func1(string a, string b)

{

…

return a+b;

};

procedure function Func2(string c)

{

…

return

};

## **1.10 Инициализация данных**

По умолчанию переменные типа byte инициализируются нулём, а переменные типа string – пустой строкой (“”). В языке GSA-2023 также допустима инициализация при объявлении. При этом переменной присваивается значение литерала или идентификатора, находящееся справа от знака равенства.

## **1.11 Инструкции языка**

Все возможные инструкции языка программирования GSA-2023 представлены в общем виде в таблице 1.4.

Таблица 1.4 – Характеристика инструкций языка GSA-2023

|  |  |
| --- | --- |
| Инструкция | Описание |
| Объявление переменной с явной инициализацией | new <тип данных> <идентификатор> = <значение>;  Значение может быть только литералом или идентификатором |
| Возврат из функции или процедуры | Для функций, возвращающих значение:  return <идентификатор/литерал>;  Для процедур: return;**return;** |
| Вывод данных | **write** <идентификатор/литерал>; |

Окончание таблицы 1.4

|  |  |
| --- | --- |
| Вызов функции или процедуры | <идентификатор функции> (<список параметров>);  Список параметров может быть пустым. |
| Перевод строки | **newline**; |

Таким образом язык GSA-2023 использует следующий синтаксис, приведённый выше.

**1.12 Операции языка**

В языке GSA-2023 предусмотрены следующие операции с данными. Приоритетность операций сдвигов выше приоритета операции умножения, далее идут сложение и вычитание. Операции языка представлены в таблице 1.5.

Таблица 1.5 – Операции языка GSA-2023

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические | 1. + – сложение (1) 2. - – вычитание (1) 3. \* – умножение(2) |
| Строковые | 1. = – присваивание |
| Логические | 1. > – больше(1)  2. < – меньше(1)  3. ! – проверка на неравенство(1) |
| Сдвиговые | 1. } – cдвиг вправо(3)  2. { – сдвиг влево(3) |

В языке GSA-2023 операции арифметических, строковых, логических и сдвиговых операторов определены с соответствующими приоритетами, где сдвиги имеют более высокий приоритет.

**1.13 Выражения и их вычисления**

Выражения языка GSA-2023 подчиняются следующим правилам:

* Допускается использовать скобки для смены приоритета операций;
* Выражение записывается в строку без переносов;
* Использование двух подряд идущих операторов не допускается;
* 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 GSA-2023 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода. Ключевые программные конструкции языка программирования GSA-2023 представлены в таблице 1.6

Таблица 1.6 – Характеристика конструкций языка GSA-2023

|  |  |
| --- | --- |
| Конструкция | Реализация |
| Главная функция | **main**  **[**  …  **]** |
| Внешняя функция | <тип данных> **function** <идентификатор> **(**<тип> <идентификатор>, ...**)**  **[**…  **return** <идентификатор/литерал>;  **]** |
| Внешняя процедура | **procedure** **function** <идентификатор> **(** <тип> <идентификатор>, ...**)**  **[**…  **return**;  **]** |
| Цикл | **condition**: <идентификатор1> <оператор> <идентификатор2> **#**  **cycle** [ … **]** **#**  Цикл (операторы внутри блока **cycle**) выполняется, пока истинно условие “<идентификатор1> <оператор> <идентификатор2>”, имеющее тот же смысл, что и в примере выше. |
| Условная конструкция | condition: <идентификатор1> <оператор> <идентификатор2> #  istrue [ … ]  isfalse [ … ] #  <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 блока istrue, иначе – код внутри блока isfalse. Любой из блоков istrue, isfalse может отсутствовать, но не оба блока одновременно. При отсутствии одного из блоков, в зависимости от истинности или ложности условия программа может как выполнить один из заявленных блоков, так и передать управление инструкции, следующей в коде за закрывающим условную конструкцию символом ‘#’. |

В данной таблице представлены различные конструкции программирования и их реализация. Главная функция "main" является основной точкой входа в программу. Внешние функции и процедуры определяются с помощью ключевых слов "function" и "procedure" со списком параметров, а операторы внутри блока выполняются в зависимости от условий или в цикле. Условные конструкции позволяют выполнять различные кодовые блоки в зависимости от условий истинности или ложности.

**1.15 Область видимости идентификаторов**

Все идентификаторы обязаны быть объявленными внутри функции. Вне функции объявление идентификаторов недопустимы. 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**

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

Таблица 1.7 – Перечень семантических проверок

|  |  |
| --- | --- |
| Номер | Правило |
| 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; |

Следовательно, при несоблюдении хотя бы одного из правил, семантический анализатор будет сообщать об ошибке.

## **1.17 Распределение оперативной памяти на этапе выполнения**

При трансляции кода используется две области памяти: сегмент констант и сегмент данных. 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 ее состав**

В языке GSA-2023 присутствует стандартная библиотека, которая подключается автоматически при трансляции исходного кода в язык ассемблера. Все стандартные библиотеки реализованы на языке C++. Стандартные библиотеки подключены по умолчанию в программу. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Содержимое библиотеки и описание функций представлено в таблице 1.8.

Таблица 1.8 – Стандартная библиотека языка GSA-2023

|  |  |
| --- | --- |
| Функция | Характеристика |
| string concat(string a, string b) | Строковая функция, возвращающая результат конкатенации строк а и b, записанный в строку a |
| byte length(string a) | Целочисленная функция. Вычисляет и возвращает длину строки str |
| byte atoi(string а) | Целочисленная функция, возвращающая результат преобразования строки а в число. |

Вывод данных осуществляется с помощью ключевого слова write. В качестве аргумента принимаются литералы, идентификаторы и выражения. В языке программирования GSA-2023 ввод данных не поддерживается.

## **1.20 Точка входа**

Точкой входа является функция main. Точка входа в приложение не может отсутствовать, а также должна быть единственной.

**1.21 Препроцессор**

В языке GSA-2023 препроцессоры не предусмотрены.

## **1.22 Соглашения о вызове**

При генерации кода используется соглашение о вызове stdcall. Параметры функций передаются через стек справа налево. Память освобождает вызываемый код.

## **1.23 Объектный код**

Исходный код языка транслируется в язык ассемблера, а затем в объектный код.

**1.24 Классификация сообщений транслятора**

В случае возникновения ошибки в исходном коде программы на языке программирования GSA-2023 и выявлении её транслятором в файл протокола выводится сообщение. Классификация обрабатываемых ошибок приведена в таблице 1.9.

Таблица 1.9 – Классификация ошибок

|  |  |
| --- | --- |
| Номера ошибок | Характеристика |
| 0 – 200 | Системные ошибки |
| 200 – 299 | Ошибки лексического анализа |
| 300 – 399 | Ошибки семантического анализа |
| 600 – 699 | Ошибки синтаксического анализа |
| 400-499, 700-999 | Зарезервированные коды ошибок |

Соответственно в таблице представлены возможные ошибки, которые могут возникнуть на языке GSA-2023. Ошибки разделены на категории в зависимости от их характеристик и номеров.

**1.25 Контрольный пример**

Листинг исходного кода на языке GSA-2023 представлен в приложении А.

# 2. Структура транслятора

**2.1 Компоненты транслятора, их назначение и принципы взаимодействия**

В языке GSA-2023 исходный код проходит трансляцию в язык Assembler. Транслятор состоит из нескольких компонентов, которые взаимодействуют между собой и выполняют свои специфические функции, описанные в разделе 2.1. Чтобы получить ассемблерный код, требуется использовать выходные данные, сгенерированные лексическим анализатором [2]. Эти данные включают таблицу лексем и таблицу идентификаторов. Для указания выходных файлов, которые будут сгенерированы, используются определенные входные параметры, описанные в таблице. 2.1. Структура транслятора языка GSA-2023 приведена на рисунке 2.1.



Рисунок 2.1 – Структура транслятора языка программирования GSA-2023

Первая стадия компиляции – лексический анализ, для реализации используется лексический анализатор или сканер. Входные данные – последовательность символов исходного языка. Он выполняет предварительный анализ текста, преобразуя единый массив символов в отдельные слова или «токены»: идентификаторы, числа, операторы, ключевые слова, другие лексические единицы. Каждой лексеме присваивается тип, и она записывается в таблицу лексем, а также в таблицу идентификаторов, где хранится дополнительная информация.

Синтаксический анализатор – часть компилятора, выполняет синтаксический анализ или проверку соответствия исходного кода грамматическим правилам. Входными данными для синтаксического анализатора являются таблица лексем и таблица идентификаторов. Результатом работы синтаксического анализатора является дерево разбора.

Семантический анализатор – это часть транслятора, которая выполняет семантический анализ или проверку исходного кода на наличие ошибок, которые невозможно обнаружить при помощи грамматики. Входными данными для семантического анализатора являются таблица лексем и таблица идентификаторов.

Генератор кода – это часть транслятора, которая создает ассемблерный код на основе данных, полученных на предыдущих этапах компиляции. Входными данными для генератора кода являются таблица лексем и таблица идентификаторов. Генератор кода использует эти данные для создания файла 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 GSA-2023

|  |  |  |
| --- | --- | --- |
| Входной параметр | Назначение | Значение по умолчанию |
| -in:<путь к in-файлу> | Указывает на файл с исходным кодом на языке GSA-2023 , имеющий расширение .txt | Не предусмотрено |
| -log:<путь к log-файлу> | Файл журнала для вывода протоколов работы программы. | Значение по умолчанию:  <имя in-файла>.log |

В таблице 2.1 представлены входные параметры транслятора языка GSA-2023, которые используются для формирования файлов с результатами работы лексического, синтаксического и семантического анализаторов.

**2.3 Протоколы, формируемые транслятором и их содержимое**

Таблица протоколов, формируемых транслятором языка и их назначением представлена в таблице 2.2.

Таблица 2.2 – Протоколы, формируемые транслятором языка GSA-2023

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования GSA-2023. Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа. |
| Выходной файл, c расширением ".asm" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

Выходной протокол транслятора GSA-2023 включает подробную информацию о лексическом анализе, синтаксическом анализе и дереве разбора.

**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 [3].

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

|  |
| --- |
|  |

Рисунок 3.1 – Структура лексического анализатора

Лексический анализатор играет важную роль в компиляции программ, так как он выполняет предварительный разбор исходного кода, выделяя лексические единицы и преобразуя их в лексемы.

## **3.2 Входные и выходные данные лексического анализатор**

Входными параметрами являются данные исходного кода. Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. В качестве выходных параметров выступают таблица лексем и таблица идентификаторов, которые необходимы для продолжения компиляции, в качестве входных данных синтаксического анализатора.

## **3.3 Параметры лексического анализатора**

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

## **3.4 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 «string» представлен на рисунке 3.2.

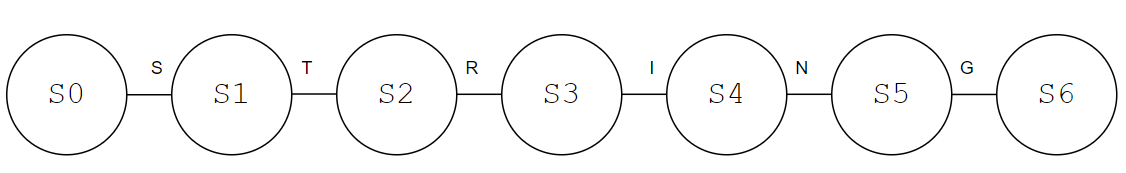


Рисунок 3.2 – Граф переходов конечного автомата

В графе на рисунке 3.2 S0 – начальное состояние, а S6 – конечное состояние автомата.

## **3.5 Контроль входных символов**

Таблица входных символов представлена на рисунке 3.3, категории входных символов представлены в таблице 3.4.



Рисунок 3.3. – Таблица контроля входных символов

Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям.

Таблица 3.1 – Соответствие символов и их значений в таблице

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

Следовательно, в таблице выше приведены входные символы языка программирования GSA-2023 и их значения.

## **3.6 Удаление избыточных символов**

Лишними символами в данном контексте являются символы табуляции и пробелы, которые не несут смысловой нагрузки в исходном коде программы. Эти излишние символы удаляются в процессе разделения исходного кода на токены.

Алгоритм удаления излишних символов может быть описан следующим образом:

* Исходный код программы считывается символ за символом.
* Если встречается пробел или символ табуляции, он рассматривается как разделительный символ.
* В отличие от других разделительных символов, пробелы и символы табуляции не добавляются в очередь лексем, а игнорируются.

## **3.7 Перечень ключевых слов**

Соответствие токенов и лексем приведено в таблице 3.2.

Таблица 3.2 – Соответствие токенов и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| byte, string | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 8 символов. |
| Литерал | l | Литерал любого доступного типа. |
| Восьмиричный литерал | q | Литерал в восьмиричном представлении. |
| function | f | Объявление функции. |
| procedure | p | Ключевое слово для процедур – функций, не возвращающих значения. Указывается перед словом function. |
| return | e | Выход из функции/процедуры. |
| main | m | Главная функция. |
| new | n | Объявление переменной. |
| writeline | o | Ввод данных. |
| condition: | ? | Указывает начало цикла/условного оператора. |
| istrue | r | Истинная ветвь условного оператора. |
| isfalse | w | Ложная ветвь условного оператора. |
| cycle | c | Указывает на начало тела цикла. |
| newline | ^ | Оператор вывода символа перевода строки. |
| # | # | Разделение конструкций в цикле/условном операторе. |
| ; | ; | Разделение выражений. |
| , | , | Разделение параметров функций. |
| [ | [ | Начало блока/тела функции. |
| ] | ] | Закрытие блока/тела функции. |
| }  { | }  { | Знаки сдвиговых операций. |
| ( | ( | Передача параметров в функцию, приоритет операций. |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций. |
| = | = | Знак присваивания. |
| + - \* / | + - \* / | Знаки операций. |
| >  <  ! | >  <  ! | Знаки логических операторов |

Лексический анализатор преобразует исходный текст, заменяя лексические единицы лексемами для создания промежуточного представления исходной программы.

Каждое выражение в исходном коде соответствует детерминированному конечному автомату, который используется для его разбора. Для каждого автомата в массиве подается токен, и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор выражения. Структура конечного автомата и пример графа переходов конечного автомата представлена в листингах 3.4 и 3.5 соответственно.

Пример реализации таблицы лексем представлен в Приложении Б.

Листинг 3.1 – Структура конечного автомата

|  |
| --- |
| struct RELATION{  char symbol;  short nnode;  RELATION(  char c,  short ns  );  };  struct NODE  {  short n\_relation;  RELATION\* relations;  NODE();  NODE(short n, RELATION rel, ...); };  struct FST //недетерминированный конечный автомат  {  char\* string;  short position;  short nstates;  NODE\* node;  short\* rstates;  FST(short ns, NODE n, ...);  FST(char\* s, FST& fst); };  bool execute(FST& fst); |

Если разбор проходит успешно, выражение записывается в таблицу лексем. Если выражение является идентификатором или литералом, связанная с ним информация также добавляется в таблицу идентификаторов.

Листинг 3.2– Пример реализации графа конечного автомата для токена condition

|  |
| --- |
| #define GRAPH\_CONDITION 11,\  FST::NODE(1,FST::RELATION('c',1)),\  FST::NODE(1,FST::RELATION('o',2)),\  FST::NODE(1,FST::RELATION('n',3)),\  FST::NODE(1,FST::RELATION('d',4)),\  FST::NODE(1,FST::RELATION('i',5)),\  FST::NODE(1,FST::RELATION('t',6)),\  FST::NODE(1,FST::RELATION('i',7)),\  FST::NODE(1,FST::RELATION('o',8)),\  FST::NODE(1,FST::RELATION('n',9)),\  FST::NODE(1,FST::RELATION(':',10)),\  FST::NODE() |

Граф перехода представлен в виде последовательности узлов FST::NODE, где каждый узел связан с другими узлами с помощью отношений FST::RELATION. Каждое отношение определяется символом и индексом узла, к которому осуществляется переход.

## **3.8 Основные структуры данных**

Структуры данных, используемые в лексическом анализаторе, включают таблицу лексем и таблицу идентификаторов. Таблица лексем содержит информацию о каждой лексеме, включая ее номер, саму лексему (lexema), номер строки в исходном коде (sn) и, если лексема является идентификатором, ее номер в таблице идентификаторов (idxTI). Код C++ со структурой таблицы лексем представлен в листинге 3.3. Код C++ со структурой таблицы идентификаторов представлен в листинге 3.4.

|  |
| --- |
| struct Entry  {  char lexema;  int sn;  int idxTI;  Entry();  Entry(char lexema, int snn, int idxti = NULLDX\_TI);  };  struct LexTable  {  int maxsize;  int size;  Entry\* table;  };  LexTable Create(int size);  void Add(LexTable& lextable, Entry entry);  void writeLexTable(std::ostream\* stream, LT::LexTable& lextable); void writeLexemsOnLines(std::ostream\* stream, LT::LexTable& lextable); |

Листинг 3.3 – Структура таблицы лексем

Данная таблица лексем предоставляет удобную структуру для хранения информации о лексемах, их позициях в исходном тексте и связях с таблицей идентификаторов.

Рисунок 3.4 – Структура таблицы идентификаторов

|  |
| --- |
| Entry() {  this->value.vint = NUM\_DEFAULT;  this->value.vstr.len = NULL;  this->value.params.count = NULL;};  Entry(char\* id, int idxLT, IDDATATYPE datatype, IDTYPE idtype)  {  strncpy\_s(this->id, id, SCOPED\_ID\_MAXSIZE - 1);  this->idxfirstLE = idxLT;  this->iddatatype = datatype;  this->idtype = idtype;};};  struct IdTable  {  int maxsize;  int size;  Entry\* table; };  IdTable Create(int size = NULL);  void Add( |

Окончание листинга 3.4

|  |
| --- |
| IdTable& idtable,  Entry entry);  int isId(  IdTable& idtable,  char id[SCOPED\_ID\_MAXSIZE]);  bool SetValue(IT::Entry\* entry, char\* value); bool SetValue(IT::IdTable& idtable, int index, char\* value);  void writeIdTable(std::ostream\* stream, IT::IdTable& idtable); |

Таблица идентификаторов содержит информацию об идентификаторах, включая их имена (id), номер первой лексемы, связанной с этим идентификатором (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value).

## **3.9 Структура и перечень сообщений лексического анализатора**

В процессе трансляции программы возникают ошибки, которые записываются в протокол. Формат протокола определяется входными параметрами. Если в процессе анализа обнаруживается ошибка, она регистрируется вместе с номером ошибки и диагностическим сообщением. Если число найденных ошибок более трёх, анализ останавливается, и дальнейшая обработка прекращается [4].

## **3.10 Принцип обработки ошибок**

Для обработки ошибок, лексический анализатор использует таблицу, содержащую информацию о сообщениях. Каждое сообщение в таблице содержит номер сообщения, номер строки и позицию в исходном коде, где было обнаружено, а также информацию об ошибке. Перечень сообщений представлен в таблице 3.3.

Таблица 3.3 – Сообщения лексического анализатора

|  |  |
| --- | --- |
| ERROR\_ENTRY 200 | "Недопустимый символ в исходном файле(-in)" |
| ERROR\_ENTRY 201 | "Неизвестная последовательность символов" |
| ERROR\_ENTRY 202 | "Превышен размер таблицы лексем" |
| ERROR\_ENTRY 203 | "Превышен размер таблицы идентификаторов" |

Когда возникает сообщение, лексический анализатор не продолжает работу и выдаёт сообщение о прекращении программы.

**3.11 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.

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

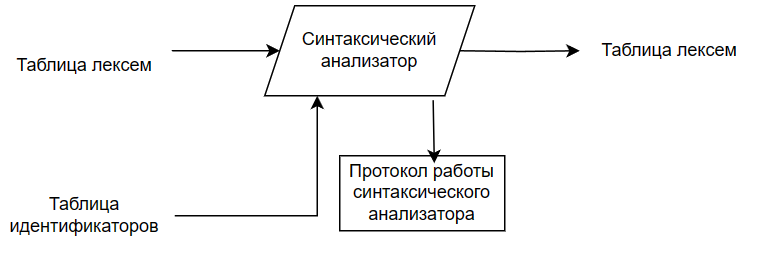


Рисунок 4.1 – Структура синтаксического анализатора.

Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией – дерево разбора

**4.2 Контекстно-свободная грамматика, описывающая синтаксис языка**

В синтаксическом анализаторе транслятора языка GSA-2023 используется контекстно-свободная грамматика, где

T – множество терминальных символов,

N – множество нетерминальных символов,

P – множество правил языка,

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:

1. , где ; (или , или );
2. , 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->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 |
| G | G->[e;]  G->[Ke;] | Правила для тела процедур |
| 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#  R->rYwY#  R->wYrY# | Правила составления цикла/условного оператора |
| Z | Z->iLi  Z->iLl  Z->lli | Правила для условия цикла/условного оператора |
| L | L-><  L->>  L->! | Правила для логических операторов |
| A | A->+  A->-  A->\*  A->/  A->{  A->} | Правила для арифметических и свдиговых операторов |
| V | V->l  V->i  V->q | Правила для простых выражений |
| Y | Y->[X] | Правила для тела цикла/условного выражения |

Окончание таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| 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 цикла/условного оператора |

В таблице 4.1 представлено описание нетерминальных символов и соответствующих правил переходов в контекстно-свободной грамматике языка GSA-2023.

## **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 |

Из данной таблицы можно сделать вывод, что магазинный автомат используется для анализа и обработки языка GSA-2023 с использованием контекстно-свободной грамматики. Автомат состоит из состояний, алфавитов символов, функции переходов и имеет начальное и конечные состояния. С помощью этих компонентов автомат выполняет анализ и трансляцию программного кода на языке GSA-2023.

**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 GSA-2023. Данные структуры в приложении В.

**4.5 Описание алгоритма синтаксического разбора**

Принцип работы автомата следующий:

* В начале, в магазин записывается стартовый символ.
* На основе предварительно полученных таблиц формируется входная последовательность символов, которая будет использоваться для разбора.
* Запускается работа магазинного конечного автомата.
* Автомат выбирает цепочку, соответствующую текущему нетерминальному символу, и записывает ее в магазин в обратном порядке.
* Если терминалы в стеке и во входной последовательности совпадают, то соответствующий терминал удаляется из стека и входной последовательности. Если терминалы не совпадают, происходит откат и возвращение к предыдущей цепочке.
* Если в магазине встречается нетерминальный символ, процесс переходит к пункту 4.
* Если в магазине достигнуто дно, а входная последовательность пуста, то синтаксический анализ считается успешно выполненным. В противном случае генерируется исключение, указывающее на наличие синтаксической ошибки.

**4.6 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.

**4.7 Структура и перечень сообщений синтаксического анализатора**

Перечень сообщений синтаксического анализатора представлен в таблице 4.3.

Таблица 4.3 – Сообщения синтаксического анализатора

|  |  |
| --- | --- |
| ERROR\_ENTRY 600 | "Синтаксическая ошибка: Неверная структура программы" |
| ERROR\_ENTRY 601 | "Синтаксическая ошибка: Не найден список параметров функции" |
| ERROR\_ENTRY 602 | "Синтаксическая ошибка: Ошибка в теле функции" |
| ERROR\_ENTRY 603 | "Синтаксическая ошибка: Ошибка в теле процедуры" |
| ERROR\_ENTRY 604 | "Синтаксическая ошибка: Ошибка в списке параметров функции |
| ERROR\_ENTRY 605 | "Синтаксическая ошибка: Ошибка в вызове функции/выражении" |

Каждая ошибка представлена макросом ERROR\_ENTRY, который содержит код ошибки и сообщение самой ошибки.

**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 результаты работы лексического и синтаксического анализаторов в качестве входных данных. Это включает таблицы лексем, идентификаторов и дерево разбора, полученное в результате синтаксического анализа. Семантический анализатор последовательно проводит различные проверки, чтобы обнаружить возможные ошибки. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.

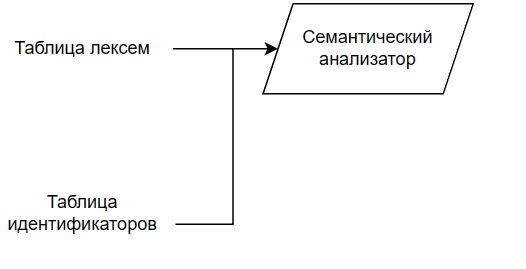


Рисунок 5.1. Структура семантического анализатора

Однако, некоторые проверки, такие как проверка единственности точки входа или предварительное объявление переменных, могут быть выполнены уже на этапе лексического анализа.

**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 в таблице 5.1.

Таблица 5.1 – Сообщения семантического анализатора

|  |  |
| --- | --- |
| ERROR\_ENTRY 300 | "Семантическая ошибка: Необъявленный идентификатор" |
| ERROR\_ENTRY 301 | "Семантическая ошибка: Отсутствует точка входа main" |
| ERROR\_ENTRY 302 | "Семантическая ошибка: Обнаружено несколько точек входа main" |
| ERROR\_ENTRY 303 | "Семантическая ошибка: В объявлении не указан тип идентификатора" |

Окончание таблицы 5.1

|  |  |
| --- | --- |
| ERROR\_ENTRY 304 | "Семантическая ошибка: В объявлении отсутствует ключевое слово new" |
| ERROR\_ENTRY 305 | "Семантическая ошибка: Попытка переопределения идентификатора" |

Данные ошибки семантики помогают и обрабатывать проблемы, связанные с некорректным использованием идентификаторов, отсутствием или повторным определением точки входа, отсутствия указания типов и другими семантическими правилами языка GSA-2023.

**5.4 Принцип обработки ошибок**

В процессе трансляции программы возникающие ошибки записываются в протокол, который определяется входными параметрами. При возникновении ошибок происходит их протоколирование с указанием номера ошибки и диагностического сообщения. Анализ программы продолжается до тех пор, пока не будут найдены все ошибки, и только после этого процесс анализа останавливается. [5]

## **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[  byte x = 9;  Writeline x;] | Ошибка N304: Семантическая ошибка: В объявлении отсутствует ключевое слово new Строка: 2 |
| main[  new byte x = 9;  new string y =x;] | Ошибка N314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 3 |
| main[  new byte x = 9;]  main[  new string y = "qwerty";] | Ошибка N302: Семантическая ошибка: Обнаружено несколько точек входа main Строка: 0 |

Из приведенных примеров можно сделать вывод, что в процессе трансляции программы на языке GSA-2023 возможны различные семантические ошибки, связанные с отсутствием ключевых слов, несоответствием типов данных и нарушением правил синтаксиса, таких как единственная точка входа. Ошибки записываются в протокол с указанием номера ошибки и диагностического сообщения. Анализ программы продолжается до обнаружения всех ошибок, после чего процесс анализа останавливается.

**6. Вычисление выражений**

**6.1 Выражения, допускаемые языком**

В языке GSA-2023 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 |
| ( ) | 0 |
| \* | 2 |
| + | 1 |
| - | 1 |
| } | 3 |
| { | 3 |

Из таблицы 6.1 в языке GSA-2023 можно сделать вывод, что при вычислении выражений целочисленного типа данных следует учитывать порядок операций, определенный в таблице. Операции сдвигов имеют наивысший приоритет, за ними следуют умножение, а затем сложение и вычитание.

**6.2 Польская запись и принцип её построения**

В языке GSA-2023 все выражения преобразуются в обратную польскую запись. Польская запись представляет собой альтернативный способ записи арифметических выражений, в котором операции располагаются перед или после операндов, без использования скобок. Алгоритм преобразования в обратную польскую запись включает следующие шаги:

* Исходное выражение рассматривается слева направо.
* Операнды копируются в результирующую строку.
* Операции помещаются в стек, если стек пуст.
* Если встречается операция, она выталкивает из стека все операции с более высоким или равным приоритетом в результирующую строку.
* Открывающая скобка помещается в стек.
* Закрывающая скобка выталкивает из стека все операции до открывающей скобки, а затем обе скобки удаляются.

Таким образом, в результате преобразования все выражения языка GSA-2023 представляются в виде обратной польской записи, что позволяет выполнять их вычисление без использования скобок и с легкостью определять порядок операций.

Пример приведения выражения к обратной польской записи представлен в пункте 6.4.

**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 польской обратной записи

|  |  |
| --- | --- |
| Выражение | Обратная польская запись для выражения |
| i[6]=(((l[1]+l[7])-i[0])\*l[5]){l[8]; | i[6]=l[1]l[7]+i[0]-l[5]\*l[8]{ |
| i[4]=(i[4]-l[61])\*l[61] | i[4]=i[4]l[61]-l[61]\* |
| i[0]=(((l[13]-(l[32]+l[5]))+i[3])\*l[6]) | i[0]=l[13]l[32]l[5]+i[3]-l[6]\* |

Таким образом, исходя из вышеприведённой таблицы, преобразование выражений в обратную польскую запись в языке GSA-2023 упрощает алгоритмы их вычисления и преобразования к ассемблерному коду.

**7. Генерация кода**

**7.1 Структура генератора кода**

В языке GSA-2023 генерация кода является последним этапом процесса трансляции. Генерация кода на язык ассемблер является процессом преобразования программы, написанной на языке GSA-2023 в машинный код, который может быть выполнен препроцессором. Структура генератора кода GSA-2023 представлена на рисунке 7.1.



Рисунок 7.1 – Структура генератора кода

Генератор кода принимает таблицы лексем и идентификаторов, которые были получены в результате лексического анализа. На основе этих таблиц генератор создает выходной файл на языке ассемблера, который представляет собой окончательный результат работы транслятора. В случае возникновения ошибок в процессе трансляции, генерация кода не будет выполнена.

**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 GSA-2023 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка GSA-2023 и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке GSA-2023 | Тип идентификатора на языке ассемблера | Пояснение |
| byte | sdword | Хранит целочисленный тип данных. |

Окончание таблицы 7.1

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке GSA-2023 | Тип идентификатора на языке ассемблера | Пояснение |
| string | dword | Хранит указатель на начало строки. Строка должна завешаться нулевым символом. |

Следовательно, таблица 7.1 показывает соответствия между типами идентификаторов на языке GSA-2023 и языке ассемблера. Это важно при переводе кода с языка GSA-2023 на язык ассемблера, чтобы правильно определить типы данных и использовать соответствующие инструкции и регистры для работы с идентификаторами.

**7.3 Статическая библиотека**

В языке GSA-2023 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера.

Таблица 7.3 – Функции статической библиотеки

|  |  |
| --- | --- |
| Функция | Назначение |
| void outlstr(char\* str) | Вывод на консоль строки str |
| void outnum(int num) | Вывод на консоль целочисленной переменной num |
| int lenght(char\* buffer, char\* str) | Вычисление длины строки |
| char\* concat(char\* buffer, char\* str1, char\* str2) | Объединение строк str1 и str2 |
| int atoii(char\* ptr) | Преобразование строки в число |

В итоге, таблица 7.3 описывает функции, доступные в статической библиотеке языка GSA-2023. Эти функции могут быть полезны при работе с выводом на консоль, вычислениями строковых операций и преобразованием строк в числа.

**7.4 Особенности алгоритма генерации кода**

В языке GSA-2023 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2.

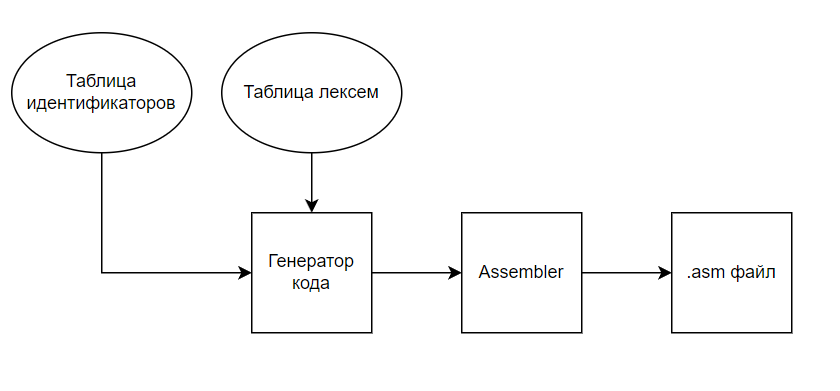


Рисунок 7.2 – Структура генератора кода

Таким образом данный генератор кода получает данные из таблицы лексем и таблицы идентификаторов, применяет к ним функции преобразующие код на ассемблер и записывает в файл .asm.

**7.5 Параметры, управляющие генерацией кода**

На вход генератору кода поступают таблицы лексем и идентификаторов исходного кода программы на языке GSA-2023. 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 7.3.

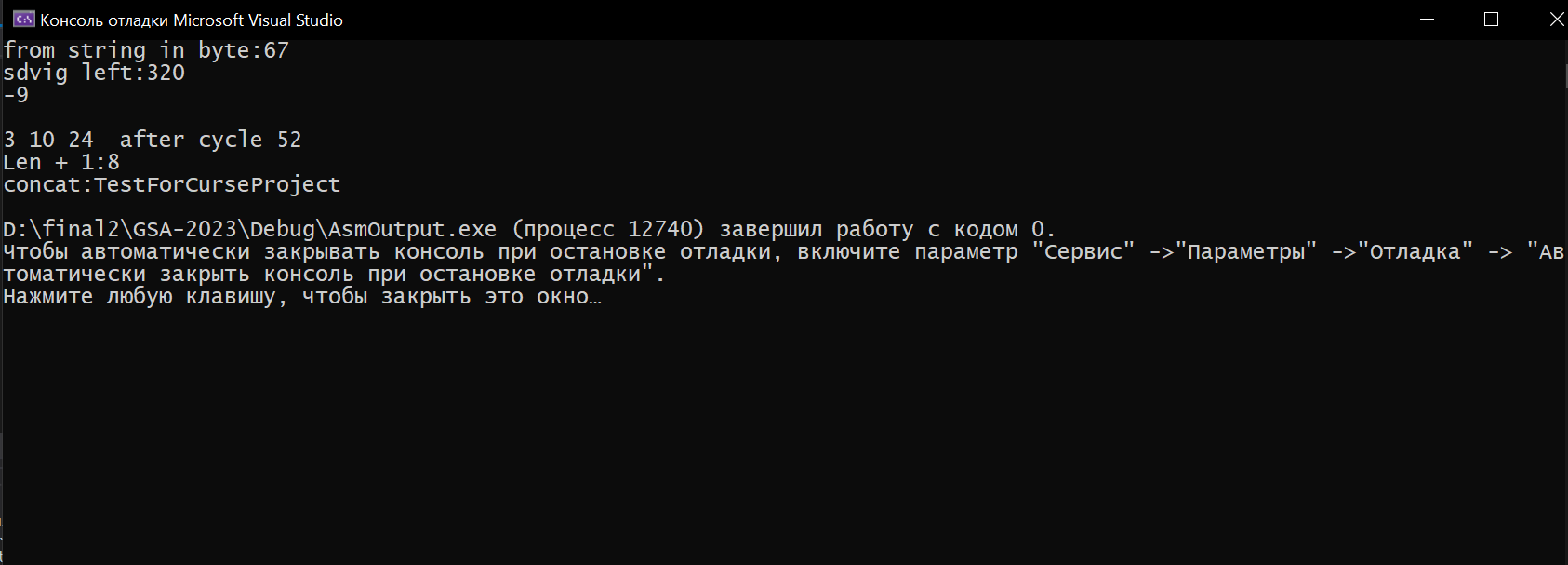


Рисунок 7.3. – Результат работы программы на языке GSA-2023

Таким образом данный контрольный пример демонстрирует все возможности языка GSA-2023, включая использование строкового и целочисленного типа данных, сдвиговых операций, функций статической библиотеки (конкатенация строк, перевод строки в число, длина строки) и цикла.

**8. Тестирование транслятора**

**8.1 Общие положения**

Тестирование должно покрывать как можно больше сценариев использования языка и его конструкций. Все тесты были представлены для типичных ошибок пользователей при использовании языка. Когда компилятор обнаруживает ошибку, он записывает информацию о ней в протокол, содержащий номер ошибки и диагностическое сообщение, помогающее разработчику понять причину ошибки компиляции. После обнаружения ошибки компилятор может продолжить анализ, чтобы найти другие возможные ошибки. Результаты тестирования записываются в файл .log.

**8.2 Результаты тестирования**

На этапе лексического анализа в языке GSA-2023 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  [new byte y8;] | Ошибка N201: Лексическая ошибка: Неизвестная последовательность символов Строка: 3 |

На этапе синтаксического анализа в языке GSA-2023 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 new [byte x; | Ошибка 600: строка 1, Синтаксическая ошибка: Неверная структура программы |
| string function fi([]  main[] | Ошибка 601: строка 1, Синтаксическая ошибка: Не найден список параметров функции |
| string function fi()  [newline; write]  main[] | Ошибка 602: строка 1, Синтаксическая ошибка: Ошибка в теле функции |
| procedure function fi()  [newline write;]  main[] | Ошибка 603: строка 1, Синтаксическая ошибка: Ошибка в теле процедуры |
| procedure function fi(byte byte)[]  main[] | Ошибка 604: строка 1, Синтаксическая ошибка: Ошибка в списке параметров функции |

Окончание таблицы 8.3

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| string function fi(byte x)[return 3;]  main [newline;fi(5,5; newline;] | Ошибка 605: строка 2, Синтаксическая ошибка: Ошибка в вызове функции/выражении |
| string function fi(byte x)[return 3;]  main [newline;fi(5,5,5 5);] | Ошибка 606: строка 2, Синтаксическая ошибка: Ошибка в списке фактических параметров функции |
| main [new byte x; condition: x > 2 # cycle #] | Ошибка 607: строка 1, Синтаксическая ошибка: Ошибка при конструировании цикла/условного выражения |
| main [new byte x; condition: x > 2 # cycle #] | Ошибка 608: строка 1, Синтаксическая ошибка: Ошибка в теле цикла/условного выражения |
| main [condition: 1 = 2 #] | Ошибка 609: строка 1, Синтаксическая ошибка: Ошибка в условии цикла/условного выражения |
| main [condition: 1 = 2 #] | Ошибка 610: строка 1, Синтаксическая ошибка: Неверный условный оператор |
| main [new byte x; x = x ! x;] | Ошибка 611: строка 1, Синтаксическая ошибка: Неверный арифметический оператор |
| main [new byte x; write new;] | Ошибка 612: строка 1, Синтаксическая ошибка: Неверное выражение. Ожидаются только идентификаторы/литералы |
| main [new byte x; x = 1 ++;] | Ошибка 613: строка 1, Синтаксическая ошибка: Ошибка в арифметическом выражении |
| main [newline; 4;] | Ошибка 614: строка 1, Синтаксическая ошибка: Недопустимая синтаксическая конструкция |
| main [new byte a; condition: a < 3 # istrue [newline; 3;] #] | Ошибка 615: строка 1, Синтаксическая ошибка: Недопустимая синтаксическая конструкция в теле цикла/условного выражения |

Семантический анализ в языке GSA-2023 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 |
| main [ a = 1 ] | Ошибка N300: Семантическая ошибка: Необъявленный идентификатор Строка: 1 |
| string function fi()[] | Ошибка N301: Семантическая ошибка: Отсутствует точка входа main Строка: 0 |
| main[]  main[] | Ошибка N302: Семантическая ошибка: Обнаружено несколько точек входа main Строка: 0 |
| main[a = 1;] | Ошибка N304: Семантическая ошибка: В объявлении отсутствует ключевое слово new Строка: 1 |
| main[new byte t;  new string t;] | Ошибка N305: Семантическая ошибка: Попытка переопределения идентификатора Строка: 3 |
| procedure function fi()[]  main[fi(“a”,”b”,”c”,”d”] | Ошибка N307: Семантическая ошибка: Слишком много параметров в вызове Строка: 1 |
| string function fi(string x, string y, string z, string s)  main[] | Ошибка N306: Семантическая ошибка: Превышено максимальное количество параметров функции Строка: 1 |
| string function fi(string x)[return "a";]  main[fi("a", "b");] | Ошибка N308: Семантическая ошибка: Кол-во ожидаемых функцией и передаваемых параметров не совпадают Строка: 2 |
| string function fi(string x)[return "a";]  main[fi("a", "b");] | Ошибка N309: Семантическая ошибка: Несовпадение типов передаваемых параметров Строка: 2 |
| main[new string x="";] | Ошибка N310: Семантическая ошибка: Использование пустого строкового литерала недопустимо Строка: 1 |
| main[new string x=";] | Ошибка N311: Семантическая ошибка: Обнаружен символ '"'. Возможно, не закрыт строковый литерал Строка: 1 |
| main[new byte x=99999999999999999;] | Ошибка N313: Семантическая ошибка: Недопустимый целочисленный литерал Строка: 1 |
| main[new byte x; x = 5 + "abc";] | Ошибка N314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 1 |
| string function fi()[return 5;]  main[newline;] | Ошибка N315: Семантическая ошибка: Тип функции и возвращаемого значения не совпадают Строка: 1 |
| main[new string x; x = "abc" + "d";] | Ошибка N316: Семантическая ошибка: Недопустимое строковое выражение справа от знака '=' Строка: 1 |
| main  [condition: "string"& 6#  istrue[write "string";]] | Ошибка N317: Семантическая ошибка: Неверное условное выражение Строка: 1 |

Таким образом данный раздел предоставляет набор тестов для проверки лексического, синтаксического и семантического анализаторов.

# Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования GSA-2023 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

* Сформулирована спецификация языка GSA-2023;
* Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
* Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
* Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
* Осуществлена программная реализация синтаксического анализатора;
* Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
* Разработан транслятор кода на язык ассемблера;
* Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка GSA-2023 включает:

* 2 типа данных;
* Поддержка операторов вывода и перевода строки;
* Возможность вызова функций стандартной библиотеки;
* Наличие 3 арифметических операторов для вычисления выражений;
* Поддержка функций, процедур, операторов цикла и условия;
* 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. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. – 429 с.
2. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768 с.
3. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c.
4. Принципы работы транслятора [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/435520/. – Дата доступа:29.11.2023.
5. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с.

# Приложение А

Листинг А.1 – Исходный код на языке GSA-2023

|  |
| --- |
| byte function min(byte x, byte y)  [  new byte res;  condition: x < y #  istrue [res = x;]  isfalse [res = y;]#  return res;]  procedure function stand (string a, string b)  [  new byte k;  k = lenght(a)+1;  write "Lenght of a + 1:";  write k;  newline;  new string str;  str = concat(a,b);  write "concat:";  write str;  newline;  return;]  main  [  new byte x = 11;  new byte y = 2;  new string strx = "For";  new string stry = "test";  new string strz = "70";  new byte e;  write "from string in byte:";  e = atoii(strz);  write e;  newline;  new byte result;  result = x{1{2;  write "sdvig left:";  write result;  newline;  new byte t;  write "min of 2 & 11:";  t = min (x,y);  write t;  newline;  new byte ab = 3;  new byte d = q120;  new byte h = q15;  new byte summ;  summ = d + h;  write "sum in hex: ";  write summ;  newline;  condition: ab ! 52 #  cycle [  write ab;  write " ";  ab = (ab + 2)\*2;]#  write " after cycle ";  write ab;  newline;  write "Result of stand:";  newline;  stand(strx,stry);]] |

# Приложение Б

Листинг Б.1 Таблица идентификаторов контрольного примера

| N |СТРОКА В ТЛ| ТИП ИДЕНТИФИКАТОРА | ИМЯ | ЗНАЧЕНИЕ (ПАРАМЕТРЫ)

| 0 | 2 | number function | min | P0:NUMBER | P1:NUMBER |

| 1 | 5 | number parameter | minx |

| 2 | 8 | number parameter | miny |

| 3 | 13 | number variable | minres |0

| 4 | 41 | proc function | stand | P0:STRING | P1:STRING |

| 5 | 44 | string parameter | standa |

| 6 | 47 | string parameter | standb |

| 7 | 52 | number variable | standk |0

| 8 | 56 | number LIB FUNC | lenght | P0:STRING |

| 9 | 68 | string variable | standstr |[0]

| 10 | 72 | string LIB FUNC | concat | P0:STRING | P1:STRING|

| 11 | 91 | number variable | mainx |0

| 12 | 93 | number literal | LTRL1 |9

| 13 | 97 | number variable | mainy |0

| 14 | 99 | number literal | LTRL2 |-9

| 15 | 103 | string variable | mainstrx |[0]

| 16 | 105 | string literal | LTRL3 |[4]Just

| 17 | 109 | string variable | mainstry |[0]

| 18 | 111 | string literal | LTRL4 |[6]string

| 19 | 115 | number variable | maint |0

| 20 | 133 | number variable | mainab |0

| 21 | 135 | number literal | LTRL5 |3

| 22 | 140 | number literal | LTRL6 |52

| 23 | 148 | string literal | LTRL7 |[1]

| 24 | 155 | number literal | LTRL8 |2

Листинг Б.2 Таблица лексем контрольного примера

|  |  |
| --- | --- |
| | N | ЛЕКСЕМА | СТРОКА | ИНДЕКС В ТИ |  | 0 | t | 1 | |  | 1 | f | 1 | |  | 2 | i | 1 | 0 |  | 3 | ( | 1 | |  | 4 | t | 1 | |  | 5 | i | 1 | 1 |  | 6 | , | 1 | |  | 7 | t | 1 | |  | 8 | i | 1 | 2 |  | 9 | ) | 1 | |  | 10 | [ | 2 | |  | 11 | n | 3 | |  | 12 | t | 3 | |  | 13 | i | 3 | 3 |  | 14 | ; | 3 | |  | 15 | ? | 4 | |  | 16 | i | 4 | 1 |  | 17 | < | 4 | |  | 18 | i | 4 | 2 |  | 19 | # | 4 | |  | 20 | w | 5 | |  | 21 | [ | 5 | |  | 22 | i | 5 | 3 |  | 23 | = | 5 | |  | 24 | i | 5 | 1 |  | 25 | ; | 5 | |  | 26 | ] | 5 | |  | 27 | r | 6 | |  | 28 | [ | 6 | |  | 29 | i | 6 | 3 |  | 30 | = | 6 | |  | 31 | i | 6 | 2 |  | 32 | ; | 6 | |  | 33 | ] | 6 | |  | 34 | # | 6 | |  | 35 | e | 7 | |  | 36 | i | 7 | 3 |  | 37 | ; | 7 | |  | 38 | ] | 8 | |  | 39 | p | 9 | |  | 40 | f | 9 | |  | 41 | i | 9 | 4 |  | 42 | ( | 9 | |  | 43 | t | 9 | |  | 44 | i | 9 | 5 |  | 45 | , | 9 | |  | 46 | t | 9 | |  | 47 | i | 9 | 6 |  | 48 | ) | 9 | |  | 49 | [ | 10 | |  | 50 | n | 11 | |  | 51 | t | 11 | |  | 52 | i | 11 | 7 |  | 53 | ; | 11 | |  | 54 | i | 12 | 7 |  | 55 | = | 12 | |  | 56 | i | 12 | 8 |  | 57 | ( | 12 | |  | 58 | i | 12 | 5 |  | 59 | ) | 12 | |  | 60 | + | 12 | |  | 61 | l | 12 | 9 |  | 62 | ; | 12 | |  | 63 | n | 13 | |  | 64 | t | 13 | |  | 65 | i | 13 | 10 |  | 66 | ; | 13 | |  | 67 | i | 14 | 10 |  | 68 | = | 14 | |  | 69 | i | 14 | 11 |  | 70 | ( | 14 | |  | 71 | i | 14 | 5 |  | 72 | , | 14 | |  | 73 | i | 14 | 6 |  | 74 | ) | 14 | |  | 75 | ; | 14 | |  | 76 | e | 15 | |  | 77 | ; | 15 | |  | 78 | ] | 16 | |  | 79 | m | 17 | |  | 80 | [ | 18 | |  | 81 | n | 19 | |  | 82 | t | 19 | |  | 83 | i | 19 | 12 |  | 84 | = | 19 | |  | 85 | l | 19 | 13 |  | 86 | ; | 19 | |  | 87 | n | 20 | |  | 88 | t | 20 | |  | 89 | i | 20 | 14 |  | 90 | = | 20 | |  | 91 | l | 20 | 15 |  | 92 | ; | 20 | |  | 93 | n | 21 | |  | 94 | t | 21 | |  | 95 | i | 21 | 16 |  | 96 | = | 21 | |  | 97 | l | 21 | 17 |  | 98 | ; | 21 | |  | 99 | n | 22 | |  |100 | t | 22 | | | |101 | i | 22 | 18 |  |102 | = | 22 | |  |103 | l | 22 | 19 |  |104 | ; | 22 | |  |105 | n | 23 | |  |106 | t | 23 | |  |107 | i | 23 | 20 |  |108 | = | 23 | |  |109 | l | 23 | 21 |  |110 | ; | 23 | |  |111 | n | 24 | |  |112 | t | 24 | |  |113 | i | 24 | 22 |  |114 | ; | 24 | |  |115 | i | 25 | 22 |  |116 | = | 25 | |  |117 | i | 25 | 23 |  |118 | ( | 25 | |  |119 | i | 25 | 20 |  |120 | ) | 25 | |  |121 | ; | 25 | |  |122 | n | 26 | |  |123 | t | 26 | |  |124 | i | 26 | 24 |  |125 | ; | 26 | |  |126 | i | 27 | 24 |  |127 | = | 27 | |  |128 | i | 27 | 12 |  |129 | { | 27 | |  |130 | l | 27 | 9 |  |131 | { | 27 | |  |132 | l | 27 | 15 |  |133 | ; | 27 | |  |134 | n | 28 | |  |135 | t | 28 | |  |136 | i | 28 | 25 |  |137 | ; | 28 | |  |138 | i | 29 | 25 |  |139 | = | 29 | |  |140 | i | 29 | 0 |  |141 | ( | 29 | |  |142 | i | 29 | 12 |  |143 | , | 29 | |  |144 | i | 29 | 14 |  |145 | ) | 29 | |  |146 | ; | 29 | |  |147 | o | 30 | |  |148 | i | 30 | 25 |  |149 | ; | 30 | |  |150 | ^ | 31 | |  |151 | ; | 31 | |  |152 | n | 32 | |  |153 | t | 32 | |  |154 | i | 32 | 26 |  |155 | = | 32 | |  |156 | l | 32 | 27 |  |157 | ; | 32 | |  |158 | n | 33 | |  |159 | t | 33 | |  |160 | i | 33 | 28 |  |161 | = | 33 | |  |162 | l | 33 | 29 |  |163 | ; | 33 | |  |164 | n | 34 | |  |165 | t | 34 | |  |166 | i | 34 | 30 |  |167 | = | 34 | |  |168 | l | 34 | 31 |  |169 | ; | 34 | |  |170 | n | 35 | |  |171 | t | 35 | |  |172 | i | 35 | 32 |  |173 | ; | 35 | |  |174 | i | 36 | 32 |  |175 | = | 36 | |  |176 | i | 36 | 28 |  |177 | + | 36 | |  |178 | i | 36 | 30 |  |179 | ; | 36 | |  |180 | ? | 37 | |  |181 | i | 37 | 26 |  |182 | ! | 37 | |  |183 | l | 37 | 33 |  |184 | # | 37 | |  |185 | c | 38 | |  |186 | [ | 38 | |  |187 | o | 39 | |  |188 | i | 39 | 26 |  |189 | ; | 39 | |  |190 | o | 40 | |  |191 | l | 40 | 34 |  |192 | ; | 40 | |  |193 | i | 41 | 26 |  |194 | = | 41 | |  |195 | ( | 41 | |  |196 | i | 41 | 26 |  |197 | + | 41 | |  |198 | l | 41 | 15 |  |199 | ) | 41 | |  |200 | \* | 41 | |  |201 | l | 41 | 15 |  |202 | ; | 41 | |  |203 | ] | 42 | | |

# Приложение В

Листинг В.1 – Грамматика языка GSA-2023

reibach greibach(NS('S'), TS('$'), 16,

Rule(NS('S'), GRB\_ERROR\_SERIES, 3, // Неверная структура программы

Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('P'), NS('T'), NS('S')), Rule::Chain(6, TS('p'), TS('f'), TS('i'), NS('P'), NS('G'), NS('S')), Rule::Chain(4, TS('m'), TS('['), NS('K'), TS(']'))

) Rule(NS('T'), GRB\_ERROR\_SERIES + 2, 2, // Ошибка в теле функции

Rule::Chain(5, TS('['), TS('e'), NS('V'), TS(';'), TS(']')),

Rule::Chain(6, TS('['), NS('K'), TS('e'), NS('V'), TS(';'), TS(']'))

),

Rule(NS('G'), GRB\_ERROR\_SERIES + 3, 2, // Ошибка в теле процедуры

Rule::Chain(4, TS('['), TS('e'), TS(';'), TS(']')),

Rule::Chain(5, TS('['), NS('K'), TS('e'), TS(';'), TS(']'))

Rule(NS('P'), GRB\_ERROR\_SERIES + 1, 2, // Не найден список параметров функции

Rule::Chain(3, TS('('), NS('E'), TS(')')),

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

),

Rule(NS('E'), GRB\_ERROR\_SERIES + 4, 2, // Ошибка в списке параметров функции

Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('E')),

Rule::Chain(2, TS('t'), TS('i'))

),

Rule(NS('F'), GRB\_ERROR\_SERIES + 5, 2, // Ошибка в вызове функции

Rule::Chain(3, TS('('), NS('N'), TS(')')),

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

),

Rule(NS('N'), GRB\_ERROR\_SERIES + 6, 4, // Ошибка в списке параметров функции

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

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

Rule::Chain(3, TS('i'), TS(','), NS('N')),

Rule::Chain(3, TS('l'), TS(','), NS('N'))

),

Rule(NS('R'), GRB\_ERROR\_SERIES + 7, 5, // Ошибка при констуировании цикла/условного выражения

Rule::Chain(3, TS('r'), NS('Y'), TS('#')),

Rule::Chain(3, TS('w'), NS('Y'), TS('#')),

Rule::Chain(3, TS('c'), NS('Y'), TS('#')),

Rule::Chain(5, TS('r'), NS('Y'), TS('w'), NS('Y'), TS('#')),

Rule::Chain(5, TS('w'), NS('Y'), TS('r'), NS('Y'), TS('#'))

),

Rule(NS('Y'), GRB\_ERROR\_SERIES + 8, 1, // Ошибка в теле цикла/условного выражения

Rule::Chain(3, TS('['), NS('X'), TS(']'))

),

Rule(NS('Z'), GRB\_ERROR\_SERIES + 9, 3, // Ошибка в условии цикла/условного выражения

Rule::Chain(3, TS('i'), NS('L'), TS('i')),

Rule::Chain(3, TS('i'), NS('L'), TS('l')),

Rule::Chain(3, TS('l'), NS('L'), TS('i'))

),

Rule(NS('L'), GRB\_ERROR\_SERIES + 10, 4,// Неверный условный оператор

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

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

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

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

),

Rule(NS('A'), GRB\_ERROR\_SERIES + 11, 4, // Неверный арифметический оператор

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

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

Rule::Chain(1, TS('\*')),

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

),

Rule(NS('V'), GRB\_ERROR\_SERIES + 12, 2, // Неверное выражение. Rule::Chain(1, TS('l')),

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

),

Rule(NS('W'), GRB\_ERROR\_SERIES + 13, 8,// Ошибка в арифметичском выражении

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

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

Rule::Chain(3, TS('('), NS('W'), TS(')')),

Rule::Chain(5, TS('('), NS('W'), TS(')'), NS('A'), NS('W')),

Rule::Chain(2, TS('i'), NS('F')),

Rule::Chain(3, TS('i'), NS('A'), NS('W')),

Rule::Chain(3, TS('l'), NS('A'), NS('W')),

Rule::Chain(4, TS('i'), NS('F'), NS('A'), NS('W'))

),

Rule(NS('K'), GRB\_ERROR\_SERIES + 14, 14,// Недопустимая синтаксическая конструкция

Rule::Chain(7, TS('n'), TS('t'), TS('i'), TS('='), NS('V'), TS(';'), NS('K')), // декларация + присваивание

Rule::Chain(5, TS('n'), TS('t'), TS('i'), TS(';'), NS('K')), // декларация

Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('K')), // присваивание

Rule::Chain(4, TS('o'), NS('V'), TS(';'), NS('K')), // вывод

Rule::Chain(3, TS('^'), TS(';'), NS('K')), // перевод строки

Rule::Chain(5, TS('?'), NS('Z'), TS('#'), NS('R'), NS('K')), // condition

Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('K')), // вызов функции

Rule::Chain(6, TS('n'), TS('t'), TS('i'), TS('='), NS('V'), TS(';')), // декларация + присваивание

Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')), // присваивание

Rule::Chain(4, TS('n'), TS('t'), TS('i'), TS(';')), // декларация

Rule::Chain(3, TS('o'), NS('V'), TS(';')), // вывод

Rule::Chain(2, TS('^'), TS(';')), // перевод строки

Rule::Chain(4, TS('?'), NS('Z'), TS('#'), NS('R')), // condition

Rule::Chain(3, TS('i'), NS('F'), TS(';')) // вызов функции

),

Rule(NS('X'), GRB\_ERROR\_SERIES + 15, 8, // Недопустимая синтаксическая конструкция в теле цикла/условного выражения

Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('X')), //присваивание

Rule::Chain(4, TS('o'), NS('V'), TS(';'), NS('X')), // вывод

Rule::Chain(3, TS('^'), TS(';'), NS('X')), // перевод строки

Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('X')), // вызов функции

Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')), // присваивание

Rule::Chain(3, TS('o'), NS('V'), TS(';')), // вывод

Rule::Chain(2, TS('^'), TS(';')), // перевод строки

Rule::Chain(3, TS('i'), NS('F'), TS(';')) //

Листинг В.2 Структура магазинного автомата

struct Mfst { //магазинный автомат

enum RC\_STEP //шаг автомата

{

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

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

NS\_NORULECHAIN,//не найдена подходящая цепочка правила NS\_ERROR, //неизвестный нетерминальный символ грамматики

TS\_OK, //текущий символ ленты == вершине стека, продвинулась лента TS\_NOK,//текущий символ ленты != вершине стека, восстановлено состояние

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

SURPRISE //неожиданный код возврата ( ошибка в step)

};

struct MfstDiagnosis //диагностика

{

short lenta\_position; //позиция на ленте

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

short nrule; //номер правила

short nrule\_chain; //номер цепочки правила

MfstDiagnosis();

MfstDiagnosis( //диагностика

short plenta\_position, //позиция на ленте

RC\_STEP prc\_step, //код завершения шага

short pnrule, //номер правила

short pnrule\_chain //номер цепочки правила

);

}

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

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

short lenta\_position; //текущая позиция на ленте

short nrule; //номер текущего правила

short nrulechain; //номер текущей цепочки, текущего правила

short lenta\_size; //размер ленты

GRB::Greibach grebach; //грамматика Грейбах

Lexer::LEX lex; //результат работы лексического анализатора

MFSTSTSTACK st; //стек автомата

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

Mfst();

Mfst(

Lexer::LEX plex, //результат работы лексического анализатора

GRB::Greibach pgrebach //грамматика Грейбах

);

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

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

char\* getDiagnosis(short n, char\*buf);//получить n-ую строку диагностики или 0х00

bool savestate(const Log::LOG &log);//сохранить состояние автомата

bool reststate(const Log::LOG &log);//восстановить состояние автомата

bool push\_chain( //поместить уепочку правила в стек

GRB::Rule::Chain chain //цепочка правила

);

RC\_STEP step(const Log::LOG &log);//выполнить шаг автомата

bool start(const Log::LOG &log); //запустить автомат

bool savediagnois(

RC\_STEP pprc\_step); //код завершения шага

void printrules(const Log::LOG &log);//вывести последовательность правил

struct Deducation //вывод

{

short size; //количество шагов в выводе

short\* nrules; //номера правил нрамматики

short\* nrulechains;//номера цепочек правил грамматики (nrules)

Deducation() { size = 0; nrules = 0; nrulechains = 0; };

} deducation;

bool savededucation();}; //сохранить дерево вывода

Листинг В.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 pstbootomT, //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 В.4 – Разбор исходного кода синтаксическим анализатором

Шаг :Правило Входная лента Стек

0 :S->tfiPTS tfi(ti,ti)[nti;?i<i#w[i=i S$

1 : SAVESTATE: 1

1 : tfi(ti,ti)[nti;?i<i#w[i=i tfiPTS$

2 : fi(ti,ti)[nti;?i<i#w[i=i; fiPTS$

3 : i(ti,ti)[nti;?i<i#w[i=i;] iPTS$

4 : (ti,ti)[nti;?i<i#w[i=i;]r PTS$

5 :P->(E) (ti,ti)[nti;?i<i#w[i=i;]r PTS$

6 : SAVESTATE: 2

6 : (ti,ti)[nti;?i<i#w[i=i;]r (E)TS$

7 : ti,ti)[nti;?i<i#w[i=i;]r[ E)TS$

8 :E->ti,E ti,ti)[nti;?i<i#w[i=i;]r[ E)TS$

9 : SAVESTATE: 3

9 : ti,ti)[nti;?i<i#w[i=i;]r[ ti,E)TS$

10 : i,ti)[nti;?i<i#w[i=i;]r[i i,E)TS$

11 : ,ti)[nti;?i<i#w[i=i;]r[i= ,E)TS$

12 : ti)[nti;?i<i#w[i=i;]r[i=i E)TS$

13 :E->ti,E ti)[nti;?i<i#w[i=i;]r[i=i E)TS$

14 : SAVESTATE: 4

14 : ti)[nti;?i<i#w[i=i;]r[i=i ti,E)TS$

15 : i)[nti;?i<i#w[i=i;]r[i=i; i,E)TS$

16 : )[nti;?i<i#w[i=i;]r[i=i;] ,E)TS$

17 : TS\_NOK/NS\_NORULECHАIN

17 : RESSTATE

890 : SAVESTATE: 81

890 : i(i,i);] iF;]$

891 : (i,i);] F;]$

892 :F->(N) (i,i);] F;]$

893 : SAVESTATE: 82

893 : (i,i);] (N);]$

894 : i,i);] N);]$

895 :N->i i,i);] N);]$

896 : SAVESTATE: 83

896 : i,i);] i);]$

897 : ,i);] );]$

898 : TS\_NOK/NS\_NORULECHАIN

898 : RESSTATE

898 : i,i);] N);]$

899 :N->i,N i,i);] N);]$

900 : SAVESTATE: 83

900 : i,i);] i,N);]$

901 : ,i);] ,N);]$

902 : i);] N);]$

903 :N->i i);] N);]$

904 : SAVESTATE: 84

904 : i);] i);]$

905 : );] );]$

906 : ;] ;]$

907 : ] ]$

908 : $

909 : LENTA\_END

910 : ------>LENTA\_END

# Приложение Г

Листинг Г.1 Программная реализация механизма преобразования в ПОЛИЗ

bool setPolishNotation(IT::IdTable& idtable, Log::LOG& log, int lextable\_pos, ltvec& v)

vector < LT::Entry > result; //результирующий вектор

stack < LT::Entry > s; // стек для сохранения операторов

bool ignore = false; // флаг вызова функции

for (unsigned i = 0; i < v.size(); i++){

if (ignore) {

result.push\_back(v[i]);

if (v[i].lexema == LEX\_RIGHTTHESIS)

ignore = false;

continue;}

int priority = getPriority(v[i]); // его приоритет

if (v[i].lexema == LEX\_LEFTHESIS || 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){

if (s.empty() || v[i].lexema == LEX\_LEFTHESIS)

{s.push(v[i]);

continue;}

if (v[i].lexema == LEX\_RIGHTTHESIS)

{//выталкивание элементов до скобки

while (!s.empty() && s.top().lexema != LEX\_LEFTHESIS)

{result.push\_back(s.top());

s.pop();}

if (!s.empty() && s.top().lexema == LEX\_LEFTHESIS)

s.pop();

continue;}

while (!s.empty() && getPriority(s.top()) >= priority){

result.push\_back(s.top());

s.pop();}

s.push(v[i]);}

if (v[i].lexema == LEX\_LITERAL|| v[i].lexema == LEX\_ID)

if (idtable.table[v[i].idxTI].idtype == IT::IDTYPE::F || idtable.table[v[i].idxTI].idtype == IT::IDTYPE::S)

ignore = true;

result.push\_back(v[i]); }

if (v[i].lexema != LEX\_LEFTHESIS & 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){

Log::writeError(log.stream, Error::GetError(1));

return false;}}

while (!s.empty()) { result.push\_back(s.top()); s.pop(); }

v = result;

return true;}

Листинг Г.2 Таблица лексем после преобразования выражений в ПОЛИЗ

|  |  |
| --- | --- |
| 1 | tfi[0]()[  3 | nti[1]=l[2];  4 | nti[3];  5 | i[3]=i[1]l[4]\*;  6 | ei[3];  7 | ]  8 | tfi[5](ti[6],ti[7])  9 | [  10 | nti[8];  11 | ?i[6]&i[7]#  12 | w[i[8]=l[9];]  13 | r[i[8]=i[0]();]#  14 | ei[8];  15 | ]  16 | pfi[10](ti[11],ti[12])  17 | [  18 | nti[13];  19 | i[13]=i[14](i[11])l[9]+;  20 | ol[15];  21 | oi[13];  22 | ^;  23 | nti[16];  24 | i[16]=i[17](i[11],i[12]);  25 | ol[18];  26 | oi[16];  27 | ^;  29 | e;  30 | ]  31 | m  32 | [  33 | nti[19]=l[9];  34 | nti[20]=l[9];  35 | nti[21];  36 | i[21]=i[5](i[19],i[20]);  37 | ol[22];  38 | oi[21];  39 | ^;  40 | nti[23];  41 | i[23]=i[19]i[20]+;  42 | oi[23];  43 | nti[24]=l[25];  44 | nti[26]=l[27]; | 45 | nti[28]=l[29];  46 | nti[30];  47 | ol[31];  48 | i[30]=i[32](i[28]);  49 | oi[30];  50 | ^;  51 | nti[33];  52 | i[33]=i[19]l[9]{l[4]{;  53 | ol[34];  54 | oi[33];  55 | nti[35];  56 | i[35]=i[19]i[20]+l[9]l[36](-(+i[19]l[4]+\*i[5](i[19],i[20])-)l[37]i[0]()+\*;  57 | oi[35];  58 | ^;  59 | nti[38]=l[2];  60 | nti[39]=l[40];  61 | nti[41]=l[42];  62 | nti[43];  63 | i[43]=i[39]i[41]+;  64 | oi[43];  65 | ^;  66 | ?i[38]!l[44]#  67 | c[  68 | oi[38];  69 | ol[45];  70 | i[38]=i[38]l[4]+l[4]\*;  71 | ]#  72 | ol[46];  73 | oi[38];  74 | ^;  75 | i[10](i[24],i[26]);  76 | ] |

# Приложение Д

Листинг Д.1 – Код на языке ассемблер

|  |  |
| --- | --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib "../Debug/GenLib.lib  ExitProcess PROTO:DWORD  .stack 4096  outnum PROTO : DWORD  outstr PROTO : DWORD  concat PROTO : DWORD, : DWORD, : DWORD  lenght PROTO : DWORD, : DWORD  atoii PROTO : DWORD, : DWORD  .const  newline byte 13, 10, 0  LTRL1 sdword 1  LTRL2 sdword 11  LTRL3 sdword 2  LTRL4 byte 'Just', 0  LTRL5 byte 'string', 0  LTRL6 byte '70', 0  LTRL7 sdword 3  LTRL8 sdword 80  LTRL9 sdword 13  LTRL10 sdword 52  LTRL11 byte ' ', 0  .data  temp sdword ?  buffer byte 256 dup(0)  minres sdword 0  standk sdword 0  standstr dword ?  mainx sdword 0  mainy sdword 0  mainstrx dword ?  mainstry dword ?  mainstrz dword ?  maine sdword 0  mainresult sdword 0  maint sdword 0  mainab sdword 0  maind sdword 0  mainh sdword 0  mainsumm sdword 0  .code  ;----------- min ------------  min PROC,  minx : sdword, miny : sdword  ; --- save registers ---  push ebx  push edx  ; ----------------------  mov edx, minx  cmp edx, miny  jl right1  jg wrong1  right1:  push minx  pop ebx  mov minres, ebx  jmp next1  wrong1:  push miny  pop ebx  mov minres, ebx  next1:  ; --- restore registers ---  pop edx  pop ebx  ; -------------------------  mov eax, minres  ret  min ENDP  ;------------------------------  ;----------- stand ------------  stand PROC,  standa : dword, standb : dword  ; --- save registers ---  push ebx  push edx  ; ----------------------  push standa  push offset buffer  call lenght  push eax  push LTRL1  pop ebx  pop eax  add eax, ebx  push eax  pop ebx  mov standk, ebx  push standb  push standa  push offset buffer  call concat  mov standstr, eax  ; --- restore registers ---  pop edx  pop ebx  ; -------------------------  ret  stand ENDP  ;------------------------------  ;----------- MAIN ------------  main PROC  push LTRL2 | pop ebx  mov mainx, ebx  push LTRL3  pop ebx  mov mainy, ebx  mov mainstrx, offset LTRL4  mov mainstry, offset LTRL5  mov mainstrz, offset LTRL6  push mainstrz  push offset buffer  call atoii  push eax  pop ebx  mov maine, ebx  push mainx  push LTRL1  pop ebx  pop eax  mov cl, bl  shl eax, cl  push eax  push LTRL3  pop ebx  pop eax  mov cl, bl  shl eax, cl  push eax  pop ebx  mov mainresult, ebx  push mainy  push mainx  call min  push eax  pop ebx  mov maint, ebx  push maint  call outnum  push offset newline  call outstr  push LTRL7  pop ebx  mov mainab, ebx  push LTRL8  pop ebx  mov maind, ebx  push LTRL9  pop ebx  mov mainh, ebx  push maind  push mainh  pop ebx  pop eax  add eax, ebx  push eax  pop ebx  mov mainsumm, ebx  mov edx, mainab  cmp edx, LTRL10  jnz cycle2  jmp cyclenext2  cycle2:  push mainab  call outnum  push offset LTRL11  call outstr  push mainab  push LTRL3  pop ebx  pop eax  add eax, ebx  push eax  push LTRL3  pop ebx  pop eax  imul eax, ebx  push eax  pop ebx  mov mainab, ebx  mov edx, mainab  cmp edx, LTRL10  jnz cycle2  cyclenext2:  push mainstry  push mainstrx  call stand  push 0  call ExitProcess  main ENDP  end main |