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**

* **КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:** «Разработка компилятора LEI-2023»

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Выполнил студент | | | | | | Лахвич Ефим Иванович |
|  |  |  |  |  |  | (Ф.И.О.) |
| Руководитель проекта | | | | | | преп.-стажёр Север Александра Сергеевна |
|  |  |  |  |  |  | (учен. степень, звание, должность, подпись, Ф.И.О.) |
| Заведующий кафедрой | | | | | | к. ф.-м.н, доц. Смелов В.В. |
|  |  |  |  |  |  | (учен. степень, звание, должность, подпись, Ф.И.О.) |
| Консультанты | | | | | | преп.-стажёр Север Александра Сергеевна |
|  |  |  |  |  |  | (учен. степень, звание, должность, подпись, Ф.И.О.) |
| Нормоконтролер | | | | | | преп.-стажёр Север Александра Сергеевна |
|  |  |  |  |  |  | (учен. степень, звание, должность, подпись, Ф.И.О.) |
| Курсовой проект защищен с оценкой | | | | | |  |

Минск 2023

**Содержание**

[Введение 3](#_Toc154166092)

[1. Спецификация языка программирования 4](#_Toc154166093)

[1.1. Характеристика языка программирования 4](#_Toc154166094)

[1.2. Определение алфавита языка программирования 4](#_Toc154166095)

[1.3. Применяемые сепараторы 4](#_Toc154166096)

[1.4. Применяемые кодировки 5](#_Toc154166097)

[1.5. Типы данных 5](#_Toc154166098)

[1.6. Преобразование типов данных 7](#_Toc154166099)

[1.7. Идентификаторы 7](#_Toc154166100)

[1.8. Литералы 7](#_Toc154166101)

[1.9. Объявление данных 8](#_Toc154166102)

[1.10. Инициализация данных 8](#_Toc154166103)

[1.11. Инструкции языка 8](#_Toc154166104)

[1.12. Операции языка 9](#_Toc154166105)

[1.13. Выражение и их вычисление 10](#_Toc154166106)

[1.14. Конструкции языка 10](#_Toc154166107)

[1.15. Область видимости переменных 11](#_Toc154166108)

[1.16. Семантические проверки 11](#_Toc154166109)

[1.17. Распределение оперативной памяти на этапе выполнения 12](#_Toc154166110)

[1.18. Стандартная библиотека и её состав 12](#_Toc154166111)

[1.19. Ввод и вывод данных 13](#_Toc154166112)

[1.20. Точка входа 13](#_Toc154166113)

[1.21. Препроцессор 13](#_Toc154166114)

[1.22. Соглашения о вызовах 14](#_Toc154166115)

[1.23. Объектный код 14](#_Toc154166116)

[1.24. Классификация сообщений транслятора 14](#_Toc154166117)

[1.25. Контрольный пример 14](#_Toc154166118)

[2. Структура транслятора 15](#_Toc154166119)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 15](#_Toc154166120)

[2.2 Перечень входных параметров транслятора 16](#_Toc154166121)

[2.3 Протоколы, формируемые транслятором 16](#_Toc154166122)

[3 Разработка лексического анализатора 17](#_Toc154166123)

[3.1 Структура лексического анализатора 17](#_Toc154166124)

[3.2 Входные и выходные данные лексического анализатора 18](#_Toc154166125)

[3.3 Параметры лексического анализатора 18](#_Toc154166126)

[3.4 Алгоритм лексического анализа 18](#_Toc154166127)

[3.5 Контроль входных символов 19](#_Toc154166128)

[3.6 Удаление избыточных символов 20](#_Toc154166129)

[3.7 Перечень ключевых слов 20](#_Toc154166130)

[3.8 Основные структуры данных 22](#_Toc154166131)

[3.9 Структура и перечень сообщений лексического анализатора 23](#_Toc154166132)

[3.10 Принцип обработки ошибок 23](#_Toc154166133)

[3.11 Контрольный пример 24](#_Toc154166134)

[4. Разработка синтаксического анализатора 25](#_Toc154166135)

[4.1 Структура синтаксического анализатора 25](#_Toc154166136)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 25](#_Toc154166137)

[4.3 Построение конечного магазинного автомата 29](#_Toc154166138)

[4.4 Основные структуры данных 30](#_Toc154166139)

[4.5 Описание алгоритма синтаксического разбора 30](#_Toc154166140)

[4.6 Структура и перечень сообщений синтаксического анализатора 30](#_Toc154166141)

[4.7. Параметры синтаксического анализатора и режимы его работы 31](#_Toc154166142)

[4.8. Принцип обработки ошибок 31](#_Toc154166143)

[4.9. Контрольный пример 31](#_Toc154166144)

[5 Разработка семантического анализатора 32](#_Toc154166145)

[5.1 Структура семантического анализатора 32](#_Toc154166146)

[5.2 Функции семантического анализатора 32](#_Toc154166147)

[5.3 Структура и перечень сообщений семантического анализатора 32](#_Toc154166148)

[5.4 Принцип обработки ошибок 33](#_Toc154166149)

[5.5 Контрольный пример 34](#_Toc154166150)

[6 Вычисление выражений 35](#_Toc154166151)

[6.1 Выражения, допускаемые языком 35](#_Toc154166152)

[6.2 Польская запись и принцип её построения 35](#_Toc154166153)

[6.3 Программная реализация обработки выражений 36](#_Toc154166154)

[6.4 Контрольный пример 36](#_Toc154166155)

[7. Генерация кода 37](#_Toc154166156)

[7.1 Структура генератора кода 37](#_Toc154166157)

[7.2 Представление типов данных в оперативной памяти 37](#_Toc154166158)

[7.3 Статическая библиотека 38](#_Toc154166159)

[7.4 Особенности алгоритма генерации кода 38](#_Toc154166160)

[7.5 Входные параметры генератора кода 39](#_Toc154166161)

[7.6 Контрольный пример 39](#_Toc154166162)

[8. Тестирование транслятора 41](#_Toc154166163)

[8.1 Общие положения 41](#_Toc154166164)

[8.2 Результаты тестирования 41](#_Toc154166165)

[Заключение 43](#_Toc154166166)

[Список использованных источников 44](#_Toc154166167)

[Приложение A 45](#_Toc154166168)

[Приложение Б 47](#_Toc154166169)

[Приложение В 50](#_Toc154166170)

[Приложение Г 51](#_Toc154166171)

[Приложение Д 52](#_Toc154166172)

[Приложение Е 57](#_Toc154166173)

[Приложение Ж 58](#_Toc154166174)

[Приложение И 64](#_Toc154166175)

[Приложение К 65](#_Toc154166176)

**Введение**

Целью курсового проекта является разработка компилятора для своего языка программирования: LEI-2023. Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над числами.

Компилятор – это программа, задачей которого является перевод программы, написанной на одном из языков программирования в программу на язык ассемблера.

Компиляция состоит из двух частей: анализа и синтеза. Анализ – это разбиение исходной программы на составные части и создание ее промежуточного представления. Синтез – конструирование требуемой целевой программы из промежуточного представления. В данном курсовом проекте мой исходный код транслируется на язык ассемблера. Мой компилятор состоит из следующих составных частей:

– лексический и семантический анализаторы;

– синтаксический анализатор;

– генератор исходного кода на языке ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

– разбработка спецификации языка программирования;

– разбратка структуры транслятора;

– разработка лексического и семантического анализаторов;

– разработка синтаксического анализатора;

– преобразование выражений;

– генерация кода на язык ассемблера;

– тестирование транслятора.

Решения каждой из поставленных задач будут приведены в соответствующих главах курсового проекта, а именно:

В первой главе работы определена спецификация языка программирования, т.е. описан синтаксис и семантика языка.

Во второй главе работы представлена структура транслятора, т.е. перечислены компоненты транслятора, их назначение и принципы взаимодействия, перечень входных параметров, перечень протоколов, формируемых транслятором и их содержимое.

В третьей главе работы показана разработка лексического анализатора, порождающего таблицы лексем и идентификаторов.

В четвертой главе работы рассказывается о синтаксическом анализаторе, который выполняет синтаксический разбор текста с распечаткой протокола разбора и дерева разбора на основе таблицы лексем.

В пятой главе описан семантический анализатор, показана его работа (распечатка выданных сообщений в трёх примерах на разных этапах).

В шестой главе решены вопросы преобразования выражений, допускаемых языком и приведена часть протокола для контрольного примера, отображающая результаты преобразования выражений в польский формат.

В седьмой главе представлена генерация кода, где из промежуточного представления порождается код на целевом языке.

В восьмой главе описывается тестирование транслятора.

# **1. Спецификация языка программирования**

## **Характеристика языка программирования**

Язык программирования LEI-2023 является процедурным, строго типизированным, не объектно-ориентированным, компилируемым.

Процедурный язык программирования — язык высокого уровня, в котором используется метод разбиения программ на отдельные связанные между собой модули — подпрограммы (процедуры и функции).

* 1. **Определение алфавита языка программирования**

Символы, используемые на этапе выполнения: [a…z], [A…Z], [0…9], символы пробела, табуляции и перевода строки, спецсимволы: { } ( ) , ; : + - / \* & | ~ ! % > < =><= == != ! “” ‘’.

* 1. **Применяемые сепараторы**

Применяемые сепараторы в языке LEI-2023, приведены в таблице 1.1.

Таблица 1.1 — Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Область применения |
| пробел | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |
| ; | Разделитель программных конструкций |
| { } | Блок функции, блок условной и циклической конструкции. |
| = | Оператор присваивания |
| ( ) | Блок фактических или формальных параметров функции, а также приоритет арифметических операций |
| , | Разделитель параметров функций |
| +, -, \*,/ , % | Арифметические операции |
| &, |, ~, ^ | Побитовые операции |
| == <= >= != > < ! && || | Логические операции |
| “” ‘’ | Границы строки/символа |

Эффективное использование сепараторов в языке LEI-2023 способствует созданию легко читаемого, структурированного и однозначного программного кода

## **Применяемые кодировки**

Для написания программ язык LEI-2023 использует кодировку ASCII, содержащую английский алфавит, а также некоторые специальные символы, такие как { } [ ] ( ) , ; ~ + - / \* % & | ~ ! > < == <= >= !=! “” ‘’.

В алфавите языка LEI-2023 используется кодировка ASCII, таблица которой

представлена на рисунке 1.1.



Рисунок 1.1 – Алфавит входных символов

ASCII (American standard code for information interchange) — название таблицы (кодировки, набора), в которой некоторым распространённым печатным и непечатным символам сопоставлены числовые коды. Таблица была разработана и стандартизована в [США](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B5%D0%B4%D0%B8%D0%BD%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%A8%D1%82%D0%B0%D1%82%D1%8B_%D0%90%D0%BC%D0%B5%D1%80%D0%B8%D0%BA%D0%B8), в 1963 году.

## **Типы данных**

В языке LEI-2023 есть 3 типа данных: знаковый целочисленный, строковый и символьный. Описание типов данных, предусмотренных в данным языке представлено в таблице 1.2.

Таблица 1.2 – Типы данных языка LEI-2023

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| int | Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 2 байт.  Максимальное значение: 2147483647.  Минимальное значение: -2147483647.  Инициализация по умолчанию: значение 0. |

Продолжение таблицы 1.2

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| int | Поддерживаемые операции:  + (бинарный) – оператор сложения;  - (бинарный) – оператор вычитания;  \* (бинарный) – оператор умножения;  / (бинарный) – оператор деления;  %(бинарный) – оператор остаток от деления;  = (бинарный) – оператор присваивания;  & (бинарный) – поразрядный(побитовый) оператор И;  ^ (бинарный) – поразрядный(побитовый) оператор исключающее ИЛИ;  | (бинарный) – поразрядный(побитовый) оператор ИЛИ;  ~ (унарный) – поразрядный(побитовый) оператор инверсия.  В качестве условия условного оператора поддерживаются следующие логические операции:  > (бинарный) – оператор «больше»;  < (бинарный) – оператор «меньше»;  >= (бинарный) – оператор «больше либо равно»;  <= (бинарный) – оператор «меньше либо равно»;  == (бинарный) – оператор равенство;  != (бинарный) – оператор неравенство;  &&(бинарный) – оператор логического И;  || (бинарный) – оператор логического ИЛИ;  ! (унарный) – оператор логического отрицание. |
| string | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 255.  Инициализация по умолчанию: строка нулевой длины “”. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, сравнение операторами == и !=, а также использование библиотечных функций. |
| char | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 1.  Инициализация по умолчанию: символ нулевой длины “”. Операции над данными символьного типа: присваивание символьному идентификатору значения другого символьного идентификатора, символьного литерала или значения символьной функции, сравнение операторами == и !=, а также использование библиотечных функций. |

Окончание таблицы 1.2

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| bool | Фундаментальный тип данных. Используется для работы с булевой логикой, который занимает 1 байт.  Инициализация по умолчанию: значение 0 (false) Операции над данными логического типа: присваивание логическому идентификатору значения другого логического идентификатора, логического литерала или значения логической функции, любой оператор, что и у целочисленного типа. |

Эти типы данных обеспечивают разнообразные возможности для работы с числами, строками, символами и с булевой логикой. Строгая типизация и поддержка различных операций обеспечивают безопасность и эффективность программирования.

## **Преобразование типов данных**

В языке программирования LEI-2023 преобразование типов данных не поддерживается, т.е. язык является строго типизированным.

## **Идентификаторы**

Общее количество идентификаторов ограничено максимальным размером таблицы идентификаторов. Идентификаторы должны начинаться только с символов латинского алфавита, могут содержать цифры. Максимальная длина идентификатора равна восьми символам. Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Префикс занимает 8 дополнительных символов. В случае превышения заданной длины, идентификаторы усекаются до длины, равной 32 символов (24 символов на имя идентификатора, 8 символов на префикс). Данные правила действуют для всех типов идентификаторов. Идентификаторы не должны совпадать с ключевыми словами.

## **Литералы**

С помощью литералов осуществляется инициализация переменных. Все литералы являются rvalue. Типы литералов языка LEI-2023 представлены в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Литералы целого типа | Последовательность цифр:  0…9 (десятичное представление)  0х 0…F (шестнадцатеричное представление) |

Окончание таблицы 1.3

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Строковые литералы | Набор символов (от 1 до 255), заключённых в “ ” |
| Символьные литералы | Символ, заключенный в ‘ ’ |
| Логические литералы | Ключевое слово true/false |

Важно отметить, что при использовании строковых и символьных литералов недопустимо вводить литерал с длиной 0, то есть "".

## **Объявление данных**

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

Пример объявления числового типа с инициализацией:

int num1 = 1

Пример объявления переменной строкового типа с инициализацией:

string str1 = “hello world”

Пример объявления переменной символьного типа с инициализацией:

char ch1 = ‘c’

Для объявления функций используется ключевое слово fun, перед которым указывается тип функции, после которого имя функции. Далее обязателен список параметров и тело функции.

## **Инициализация данных**

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. При объявлении без инициализации предусмотрены значения по умолчанию: значение 0 для типа int и строка длины 0 (“”) для типа string и char.

## **Инструкции языка**

Инструкции языка LEI-2023 представлена в таблице 1.4.

Таблица 1.4 – Инструкции языка

|  |  |
| --- | --- |
| Инструкция | Форма записи |
| Объявление переменной | <тип данных> <идентификатор>. |
| Объявление переменной с явной инициализацией | <тип данных> <идентификатор> = <значение>|<выражение>;  Значение – литерал, идентификатор, вызов функции соответствующего типа данных |

Окончание таблицы 1.4

|  |  |
| --- | --- |
| Инструкция | Форма записи |
| Объявление функции | <тип данных> fun <идентификатор> (<тип данных> <идентификатор>, …)  {  / программный блок /  return <выражение>.  } |
| Вызов функции | <идентификатор> (<идентификатор>, ...) |
| Присвоение значения | <идентификатор> = <значение>|<выражение>; |
| Вывод данных | Write <идентификатор/литерал/функция> + …; |
| Вывод данных с переходом на новую строку | WriteLine <идентификатор/литерал/функция> + …; |
| Возвращаемое значение | return <выражение>. |

Эти инструкции позволяют создавать структурированные и функциональные программы, обеспечивая разнообразные возможности для работы с данными и функциями.

## **Операции языка**

В языке LEI-2023 предусмотрены следующие операции с данными. С приоритетами операции можно ознакомиться в представленном главе 6.1. Операции языка представлены в таблице 1.5.

Таблица 1.5 — Операции языка

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические операции языка | + – сложение  - – вычитание  \* – умножение  / – деление  % – остаток от деления  = – присваивание |
| Побитовые операции | & – поразрядное И  | – поразрядное ИЛИ  ~ – поразрядное инверсия  ^ – поразрядное исключающее ИЛИ |
| Логические операции | > – больше  < – меньше  >= – больше либо равно  <= – меньше либо равно  == – равенство  != – неравенство |

Окончание таблицы 1.5

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Логические операции | && – логическое И  || – логическое ИЛИ  ! – логическое отрицание |

С помощью этих операций можно манипулировать данными, создавать условия и реализовывать разнообразные вычисления в программах, написанных на языке LEI-2023.

## **Выражение и их вычисление**

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

## **Конструкции языка**

Программа на языке LEI-2023 следует структурному подходу и оформляется в виде функций пользователя и главной функции. Программные конструкции языка представлены в таблице 1.6

Таблица 1.6 — Основные конструкции языка

|  |  |
| --- | --- |
| Конструкция | Реализация |
| Главная функция | Main  { …} |
| Внешняя функция | <тип данных> fun <идентификатор> (<тип> <идентификатор>, ...)  {…  return <выражение>;  } |
| Условная конструкция | if (<выражение>)  then { … } else { … }  <выражение> - идентификатор, функция, литерал или само выражение (с несколькими идентификаторами/функциями/  литералами и оператором(-ами)). При истинности условия выполняется код внутри блока then, иначе – код внутри блока else. Блок else может отсутствовать. |

Окончание таблицы 1.6

|  |  |
| --- | --- |
| Конструкция | Реализация |
| Конструкция цикла | Конструкция цикла while с предусловием  while (<выражение>)  { … }  Конструкция цикла while с постусловием  do  { … }  while (<выражение>);  <выражение> - идентификатор, функция, литерал или само выражение (с несколькими идентификаторами/функциями/  литералами и оператором(-ами)). Цикл с предусловием проверяет условие перед каждой итерацией и выполняет код внутри блока, если условие истинно. Цикл с постусловием сначала выполняет код, а затем проверяет условие. Процесс повторяется до тех пор, пока условие становится ложным. |

Эти конструкции позволяют структурировать код для лучшей читаемости и обеспечивают эффективное выполнение программы. Важно выделять блоки и фрагменты кода, а также использовать отступы для улучшения читаемости.

## **Область видимости переменных**

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

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

## **Семантические проверки**

В языке программирования LEI-2023 выполняются следующие семантические проверки:

* наличие функции Main – точки входа в программу;
* единственность точки входа;
* переопределение идентификаторов;
* использование идентификаторов без их объявления;
* проверка соответствия типа функции и возвращаемого параметра;
* правильность передаваемых в функцию параметров: количество, типы;
* правильность строковых выражений;
* превышение размера строковых и числовых литералов;
* правильность составленного условия условного оператора.

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

Транслированный код использует две области памяти. В сегмент констант заносятся все литералы. В сегмент данных заносятся переменные и параметры функций. В сегмент кода заносится основная часть кода, где реализованы функции. 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 оттранслированным в язык ассемблера коде переменные имеют глобальную область видимости.

## **Стандартная библиотека и её состав**

В языке LEI-2023 предусмотрена стандартная библиотека. Функции, входящие в состав библиотеки, описаны в табл. 1.7. Стандартная библиотека подключается автоматически на этапе генерации кода.

Таблица 1.7 - Функции стандартной библиотеки языка LEI-2023

|  |  |
| --- | --- |
| Функция | Описание |
| int squere(int a) | Данная функция целочисленного типа принимает 1 параметр. Функция вычисляет корень числа и возвращает это число. |
| int pow (int a, int b) | Данная функция целочисленного типа принимает два целочисленных параметра. Функция возводит число a в степень b и возвращает результат. |
| int inputInt() | Функция для вывода из стандартного потока. Введённые данные из консоли будут интерпретированы как целочисленные. Возвращает введённые данные. |
| Char inputChar() | Функция для вывода из стандартного потока. Запрашивает ввод одного символа, возвращает этот символ. |
| string inputString() | Функция для вывода из стандартного потока. Возвращает введённые данные в виде строки. |

Для вывода предусмотрен оператор Write и WriteLine. Эти функции представлены в таблице 1.8.

Таблица 1.8 Дополнительные функции стандартной библиотеки

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void outnum(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| void outstr(char\* line) | Функция для вывода в стандартный поток значения строкового, символьного идентификатора/литерала. |

Окончание таблицы 1.8

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void outbool(char\* line) | Функция для вывода в стандартный поток значения логического идентификатора/литерала. |
| void outstrline(char\* line) | Функция для вывода в стандартный поток значения строкового, символьного идентификатора/литерала и перехода на новую строку. |
| void outnumline(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала и перехода на новую строку. |
| void outboolline(char\* line) | Функция для вывода в стандартный поток значения логического идентификатора/литерала и перехода на новую строку. |
| void system\_pause() | Функция которая запускается по окончанию программы и требует нажать любой символ для завершения программы |

Стандартная библиотека написана на языке С++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю.

## **Ввод и вывод данных**

Вывод данных осуществляется с помощью операторов Write и WriteLine. Допускается использование операторов Write и WriteLine с литералами, идентификаторами и с функциями.

Функции, управляющие выводом данных, реализованы на языке С++ и вызываются из транслированного кода, конечному пользователю недоступны. Пользовательские команды Write и WriteLine в транслированном коде будут заменены вызовом нужных библиотечных функций. Библиотека, содержащая нужные процедуры, подключается на этапе генерации кода.

## **Точка** **входа**

В языке LEI-2023 каждая программа должна содержать главную функцию (точку входа) Main, с первой инструкции которой начнётся последовательное выполнение команд программы.

## **Препроцессор**

Препроцессор, принимающий и выдающий некоторые данные на вход транслятору, в языке LEI-2023 отсутствует.

## **Соглашения о вызовах**

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

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

## **Объектный код**

LEI-2023 транслируется в язык ассемблера, а затем - в объектный код

## **Классификация сообщений транслятора**

Генерируемые транслятором сообщения определяют степень его информативности, то есть сообщения транслятора должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Сообщения транслятора приведены в таблице 1.9.

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 |

Больше информации о сообщение транслятора об ошибке находится в приложении Б.

## **Контрольный пример**

Контрольный пример демонстрирует главные особенности языка LEI-2023: его фундаментальные типы, процедуры, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

Также в приложении

# **2. Структура транслятора**

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

Транслятор преобразует программу, написанную на языке LEI-2023 в программу на языке ассемблера. Компонентами транслятора являются лексический, синтаксический и семантический анализаторы, а также генератор кода на язык ассемблера. Принцип их взаимодействия представлен на рисунке 2.1.



Рисунок 2.1 Структура транслятора языка программирования LEI-2023

Лексический анализ – первая фаза трансляции. Назначением лексического анализатора является нахождение ошибок лексики языка и формирование таблицы лексем и таблицы идентификаторов. Подробнее описан в 3 главе.

Синтаксический анализ – 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 анализ в свою очередь является проверкой исходной программы на семантическую согласованность с определением языка, т.е. проверяет правильность текста исходной программы с точки зрения семантики. Подробное описание представлено в 5 главе.

Генератор кода – этап транслятора, выполняющий генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. Генератор кода принимает на вход таблицы идентификаторов и лексем и транслирует код на языке LEI-2023, прошедший все предыдущие этапы, в код на языке Ассемблера. Более полно описан в главе 7.

## 

## **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 LEI-2023

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке LEI-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 |
| -out:<путь к out-файлу> | Выходной файл – результат работы транслятора. Содержит исходный код на языке ассемблера. | Значение по умолчанию:  <имя in-файла>.asm |

Параметры позволяют управлять процессом трансляции и формирования результатов. Обратите внимание, что параметр -in является обязательным и должен быть указан для передачи пути к файлу с исходным кодом. Параметры -log и -out являются необязательными и, если не указаны, будут использованы значения по умолчанию.

## **2.3 Протоколы, формируемые транслятором**

В ходе работы программы формируются протоколы работы лексического, синтаксического и семантического анализаторов, которые содержат в себе перечень протоколов работы. Эти протоколы сохраняются в файл с расширением по умолчанию .log.

# **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.

С структура лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1 Структура лексического анализатора

В приложении А приведен исходный код программы, который подвергается лексическому анализу. Процесс лексического анализа позволяет выделить ключевые элементы, такие как идентификаторы, числа и операторы, которые будут использованы на следующих этапах компиляции. Эти выделенные элементы затем передаются синтаксическому анализатору для дальнейшего анализа структуры программы.

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

Входные данные представляют собой текстовый файл, содержащий исходный код программы на языке LEI-2023. Выходные данные лексического анализатора включают два пользовательских типа структур:

- Таблица идентификаторов: Эта структура содержит информацию об идентификаторах, используемых в программе. Включает в себя тип данных и уникальный идентификатор для каждой переменной, объявленной в программе.

- Таблица лексем: Данная структура представляет собой набор лексем, выделенных в процессе лексического анализа. Каждая лексема представлена своим типом (ключевое слово, оператор, идентификатор, литерал и т.д.), а также информацией о её местоположении в исходном коде.

Назначение данных структур заключается в предоставлении основной информации для дальнейшего синтаксического, семантического анализа и генерации кода. Таблица идентификаторов обеспечивает необходимую информацию о переменных, функции и литералах, а таблица лексем предоставляет разбиение исходного кода на лексемы для облегчения последующих этапов компиляции.

## **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**

Алгоритм лексического анализа выполняется следующим образом:

- 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, где S0 – начальное, а S6 – конечное состояние автомата.

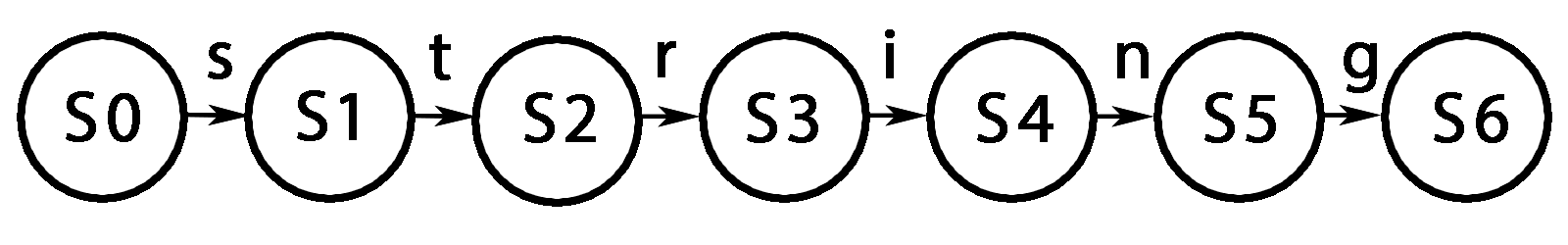


Рисунок 3.2 Пример графа переходов для цепочки string

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов.

## **3.5 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 3.3.

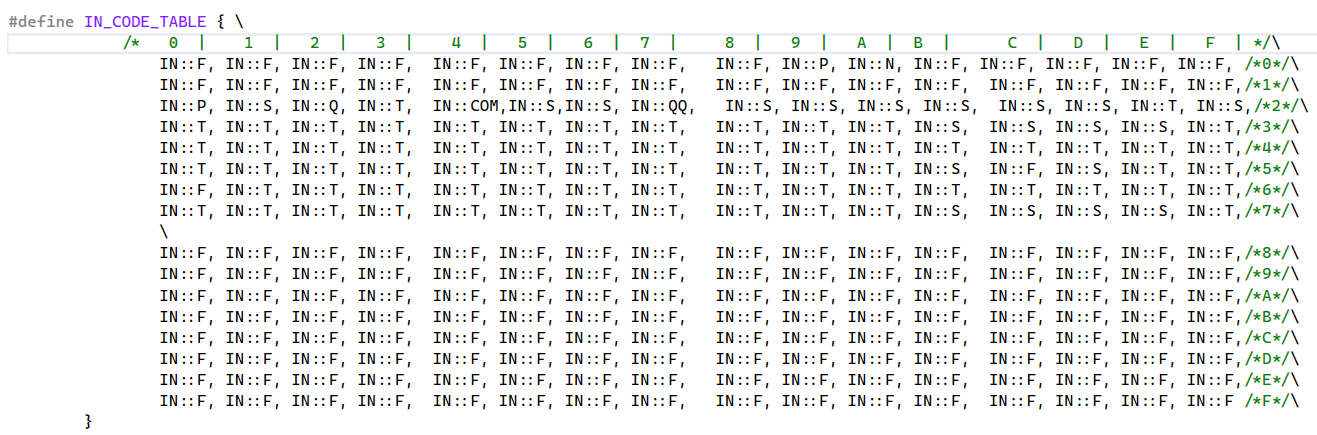


Рисунок 3.3. Таблица контроля входных символов

Расшифровка обозначений категорий:

* S – символы-сепараторы, такие как ( ) { } [ ] # < > ! & | / % \* + - = , ~ ;
* P - пробел, табуляция;
* N - новая строка;
* F - запрещённый символ;
* T - разрешённый символ;
* I - игнорируемый символ;
* Q - символ одинарных кавычек (");
* QQ - символ двойных кавычек (');
* COM - комментарий.

Эти категории обеспечивают лексическому анализатору необходимую информацию для правильного выделения лексем и разделения исходного кода на составляющие.

## **3.6 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.

## **3.7 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, char | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 8 символов. |
| Литерал | l | Литерал любого доступного типа. |
| function | f | Объявление функции. |
| Функции стандартной библиотеки | p | Ключевое слово для стандартных функций. |
| return | e | Выход из функции/процедуры. |
| Main | m | Главная функция. |
| Write | o | Вывод данных. |
| WriteLine | ^ | Вывод данных c переводом строки. |
| if | ? | Указывает начало условного оператора. |
| then | r | Истинная ветвь условного оператора. |
| else | w | Ложная ветвь условного оператора. |
| ; | ; | Разделение выражений. |
| , | , | Разделение параметров функций. |
| { | { | Начало тела функции, условной конструкции и тело цикла. |
| } | } | Конец тела функции, условной конструкции и тело цикла. |
| ( | ( | Передача параметров в функцию, приоритет операций. |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций. |

Окончание таблицы 3.2

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| = | = | Знак присваивания. |
| +  -  \*  /  %  &  |  ~  ^ | +  -  \*  /  %  &  |  ~  ^ | Знаки операций. |
| ==  !=  >  <  >=  <=  &&  ||  ! | 0  1  2  3  4  5  6  7  ! | Знаки логических операций. |

Пример реализации таблицы лексем представлен в приложении Г.

Структура конечного автомата изображены на рисунках 3.4

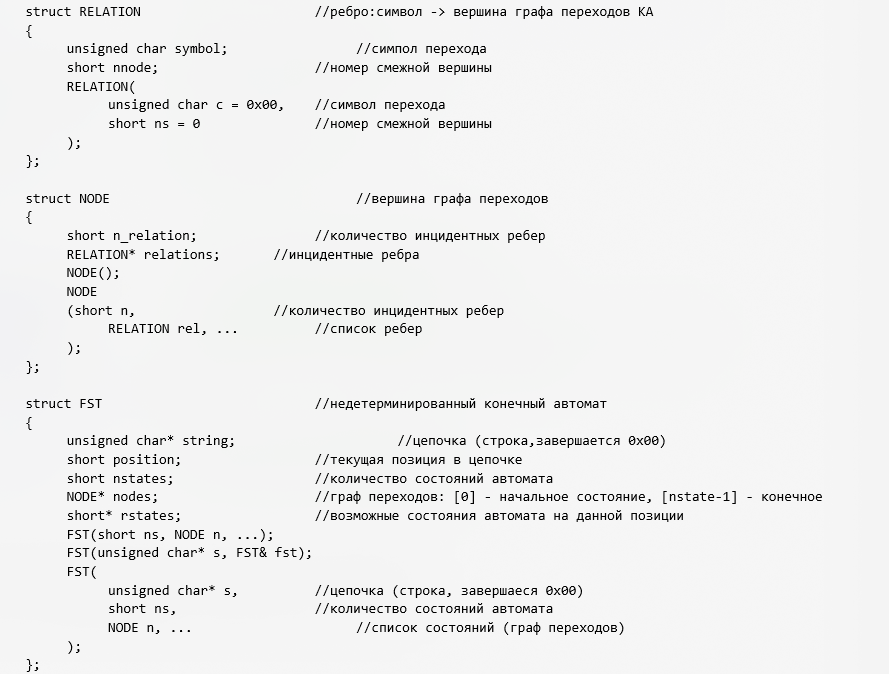


Рисунок 3.4 Структура конечного автомата

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.

Пример графа перехода конечного автомата изображены на рисунках 3.5

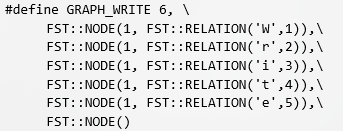


Рисунок 3.5 Реализация графа конечного автомата для токена Write

Остальные графы перехода конечного автомата представлены в приложении Д.

## **3.8 Основные структуры данных**

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде (sn), и номер в таблице идентификаторов, если лексема является идентификатором (idxTI). Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value). Код C++ со структурой таблицы идентификаторов представлен на рисунке 3.6.

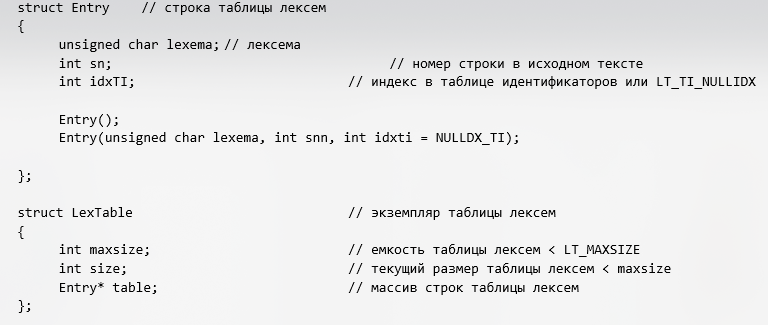


Рисунок 3.6 Структура таблицы лексем

Код C++ со структурой таблицы лексем представлен на рисунке 3.7.

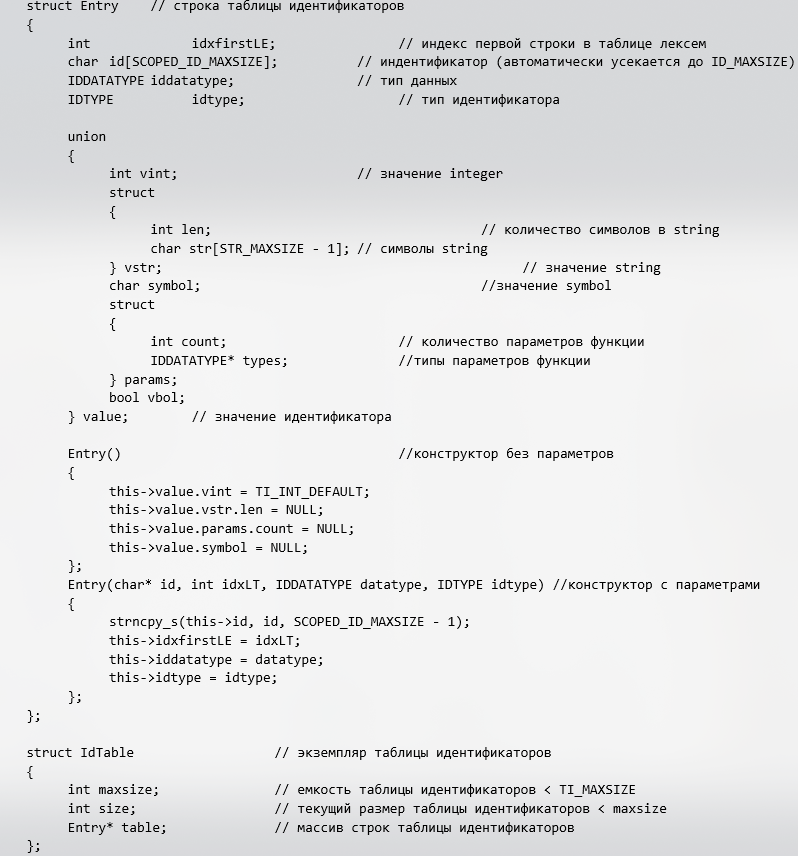


Рисунок 3.7 Структура таблицы идентификаторов

Пример реализации таблицы идентификаторов представлен в приложении В.

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

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

## **3.10 Принцип обработки ошибок**

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Перечень сообщений представлен на рисунке 3.8.

Таблицы 3.3 – список сообщений лексического анализатора

|  |  |
| --- | --- |
| Номер | Текст сообщений |
| 200 | Недопустимый символ в исходном файле(-in) |
| 201 | Неизвестная последовательность символов |
| 202 | Превышен размер таблицы лексем |
| 203 | Превышен размер таблицы идентификаторов |

Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. При возникновении сообщения, лексический анализатор игнорирует найденную ошибку и продолжает работу с исходным кодом.

## **3.11 Контрольный пример**

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

# **4. Разработка синтаксического анализатора**

## **4.1 Структура синтаксического анализатора**

Синтаксический анализатор (парсер) в компиляторе выполняет вторую стадию обработки исходного кода, выполняющий 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.

Пример реализации разбора синтаксического анализатора представлено в приложении Е.

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

В синтаксическом анализаторе транслятора языка LEI-2023 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:

1. , где ; (или , или );
2. , где — начальный символ, при этом если такое правило существует, то нетерминал  не встречается в правой части правил.

Таблица 4.1 Таблица правил переходов нетерминальных символов

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| S | S->tfiPTS  S->gfiPTS  S->m{K} | Стартовые правила, описывающее общую структуру программы |
| P | P->(E)  P->() | Правила для параметров объявляемых функций |
| T | T->{eW;}  T->{KeW;}  T->{e;}  T->{Ke;}  T->{K} | Правила для тела функций |
| E | E->ti,E  E->ti | Правила для списка параметров функции |
| F | F->(N)  F->() | Правила для вывозов функций(в т.ч. и в выражениях) |
| N | N->i  N->l  N->(W)  N->(-i)  N->iF  N->pF  N->~W  N->!W  N->iOW  N->lOW  N->(W)OW  N->(-i)OW  N->iFOW  N->pFOW  N->~WOW  N->!WOW  N->i, N  N->l, N  N->(W), N  N->(-i), N  N->iF, N  N->pF, N  N->~W, N  N->!W, N  N->iOW, N  N->lOW, N | Правила для параметров вызываемых функций |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| N | N->(W)OW, N  N->(-i)OW, N  N->iFOW, N  N->pFOW, N  N->~WOW, N  N->!WOW, N | Правила для параметров вызываемых функций |
| R | R->rY  R->wY  R->rYwY  R->wYrY | Правила составления условного оператора |
| O | O->+  O->-  O->\*  O->/  O->%  O->&  O->|  O->0  O->1  O->2  O->3  O->4  O->5  O->6  O->7 | Правила для арифметических, логических и побитовых операторов |
| V | V->l  V->i | Правила для простых выражений |
| W | W->i  W->l  W->(W)  W->(-i)  W->iF  W->pF  W->~W  W->!W  W->iOW  W->lOW  W->(W)OW  W->(-i)OW  W->iFOW  W->pFOW  W->~WOW  W->!WOW | Правила для сложных выражений |

Окончание таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| K | K->ti=W;  K->i=W;  K->ti;  K->iF;  K->pF;  K->oB;  K->^B;  K->?(W)R  K->h(W)Y  K->dYh(W);  K->ti=W;K  K->i=W;K  K->ti;K  K->iF;K  K->pF;K  K->oB;K  K->^B;K  K->?(W)RK  K->h(W)YK  K->dYh(W);K | Программные конструкции |
| U | U->iF  U->pF | Правила для идентификатора функции |
| Y | Y->{eW;}  Y->{KeW;}  Y->{e;}  Y->{Ke;}  Y->{K}  Y->{} | Тело конструкции |
| B | B->i  B->l  B->iF  B->pF  B->i+B  B->l+B  B->iF+B  B->pF+B | Правило для вывода значений |

Таблица 4.1 предоставляет правила для различных нетерминальных символов, охватывая различные аспекты языка LEI-2023. Например, правила для тел функций, списка параметров функции, условного оператора, арифметических, логических и побитовых операторов, а также различных выражений и конструкций языка.

Важно отметить, что грамматика описывает не только синтаксическую структуру языка, но и семантические аспекты, такие как операции с выражениями, условные операторы и программные конструкции.

## **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 |

Эти компоненты описывают структуру и функциональность магазинного автомата, который используется для обработки строк в соответствии с заданными грамматическими правилами.

## **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 LEI-2023. Правила языка LEI-2023 в приложении Д.

## **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**

Список сообщений синтаксического анализатора представлен на рисунке 4.3.

Таблица 4.3 – список сообщений синтаксического анализатора

|  |  |
| --- | --- |
| Номер | Текст сообщений |
| 600 | Синтаксическая ошибка. Неверная структура программы |
| 601 | Синтаксическая ошибка. Неверная структура тела функции |
| 602 | Синтаксическая ошибка. Неверная структура тела процедуры |
| 603 | Синтаксическая ошибка. Неверная структура списка параметров функции |
| 604 | Синтаксическая ошибка в списке параметров функции |
| 605 | Синтаксическая ошибка. Неверная структура списка фактических параметров функции |
| 606 | Синтаксическая ошибка в списке фактических параметров функции |
| 607 | Синтаксическая ошибка. Неверная условная конструкция |
| 608 | Синтаксическая ошибка в выражении. Такой оператор не существует |
| 609 | Синтаксическая ошибка в выражении |
| 610 | Синтаксическая ошибка в программе. Неверная структура операторы программы |
| 611 | Синтаксическая ошибка в операторе Write/Writeline. Неверно передано параметров оператора |

Окончание таблица 4.3

|  |  |
| --- | --- |
| Номер | Текст сообщений |
| 612 | Синтаксическая ошибка. Неверная структура тела конструкции |
| 613 | Синтаксическая ошибка в индентификаторе/литерале. |

Эти сообщения помогают идентифицировать различные синтаксические ошибки в программе, облегчая процесс поиска и устранения проблем.

## **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 синтаксического 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 синтаксического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью приведены в приложении Д.

# **5 Разработка семантического анализатора**

## **5.1 Структура семантического анализатора**

Семантический анализатор является важным звеном в компиляции программы, и его задача заключается в анализе смысла программы, выраженного в синтаксической структуре, и выявлении потенциальных ошибок, связанных с семантикой языка программирования.

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 ошибки, которые не могут быть выявлены на более ранних этапах компиляции. 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.

Этот этап компиляции является ключевым для обеспечения корректности и надежности исполняемого кода, а также для предотвращения потенциальных ошибок в ходе выполнения программы.

## **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 5.2.

Таблица 4.3 – список сообщений синтаксического анализатора

|  |  |
| --- | --- |
| Номер | Текст сообщений |
| 300 | Идентификатор не объявлен. |
| 301 | Отсутствует точка входа main. |
| 302 | Несколько точек входа main. |
| 303 | Не указан тип идентификатора при объявлении. |
| 304 | Отсутствует ключевое слово при объявлении. |
| 305 | Попытка переопределения идентификатора. |
| 306 | Превышено максимальное количество параметров функции. |
| 307 | Слишком много параметров в вызове функции. |
| 308 | Кол-во ожидаемых функцией и передаваемых параметров не совпадают. |
| 309 | Несовпадение типов передаваемых параметров. |
| 310 | Использование пустого строкового литерала недопустимо. |
| 311 | Обнаружен символ \'\"\'. Возможно, не закрыт строковый литерал. |
| 312 | Превышен размер строкового литерала. |
| 313 | Недопустимый целочисленный литерал. |
| 314 | Типы данных в выражении не совпадают с типом переменной. |
| 315 | Тип функции и возвращаемого значения не совпадают. |
| 316 | Недопустимое строковое выражение после =. |
| 317 | Не закрыт блок main. |
| 318 | Логическое сравнение со строками может быть только == или !=. |
| 319 | В выражений возник конфликт типов. Невозможно приведение типов. |
| 320 | Деление на ноль недопустимо. |
| 321 | Операция ! и ~ допустимо только для операнда целочисленного и логического типа. |
| 322 | В условном выражении возникла ошибка. |
| 323 | Не во всех случаях в функции имеет return. |
| 324 | Нельзя использовать рекурсию, если не во всех случаях в функции имеет return. |
| 325 | В Main нельзя использовать return. |

Эти сообщения семантического анализатора играют важную роль в выявлении и предотвращении ошибок в программе, обеспечивая правильное выполнение кода и повышая его надежность.

## **5.4 Принцип обработки ошибок**

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

## **5.5 Контрольный пример**

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

Таблица 5.1. Примеры диагностики ошибок

|  |  |
| --- | --- |
| Исходный код | Текст сообщения |
| Main {  int x = 69;  char y =x;  } | Семантическая ошибка 314: Типы данных в выражении не совпадают с типом переменной. Cтрока: 3, позиция: 12 |
| Main {  int x = 69;  }  Main {  string y = "qwerty";  } | Семантическая ошибка 302: Несколько точек входа main. |

Помимо приведенных примеров, семантический анализатор может выявлять и другие ошибки, такие как неопределенные переменные, использование недопустимых операторов, и другие нарушения семантики программы.

# **6 Вычисление выражений**

## **6.1 Выражения, допускаемые языком**

В языке LEI-2023 допускаются вычисления выражений любого типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| ( ) | 12 |
| !, ~ | 11 |
| /, \*, % | 10 |
| +, - | 9 |
| >=, <=, >, < | 8 |
| !=, == | 7 |
| & | 6 |
| ^ | 5 |
| | | 4 |
| && | 3 |
| || | 2 |
| , | 1 |

Таблица 6.1. Приоритеты операций

Обратите внимание, что операторы внутри таблицы упорядочены по убыванию приоритета, что означает, что операции с более высоким приоритетом будут выполняться раньше. Если в выражении операторы имеют одинаковый приоритет, то порядок их выполнения будет определен ассоциативностью (т.е. слева на право). Ассоциативность определяет направление вычислений в случае одинакового приоритета операторов.

## **6.2 Польская запись и принцип её построения**

Все выражения языка LEI-2023 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 Ж.

## **6.4 Контрольный пример**

Пример преобразования выражения из контрольного примера к обратной польской записи представлен в таблице 6.2.

Таблица 6.2. Преобразование выражения к ПОЛИЗ

|  |  |
| --- | --- |
| Выражение | Обратная польская запись для выражения |
| i = l + i(l, i(i - i, i)) + ~l \* (l) | i | i = l i i i i i - @ 2 l @ 2 + l ~ l \* + i | |

Преобразование выражения в формат польской записи необходимо для построения более простых алгоритмов их вычисления и преобразования к ассемблерному коду. В приложении И приведены изменённые таблицы лексем и идентификаторов, отображающие результаты преобразования выражений в польский формат.

# **7. Генерация кода**

## **7.1 Структура генератора кода**

В языке LEI-2023 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. Структура генератора кода LEI-2023 представлена на рисунке 7.1.



Рисунок 7.1 – Структура генератора кода

При возникновении ошибок в процессе трансляции, генерация кода не осуществляется, что обеспечивает предотвращение возможных ошибок в сгенерированном коде.

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

Таблица 7.1 – Соответствия типов идентификаторов языка и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке LEI-2023 | Тип идентификатора на языке ассемблера | Пояснение |
| int | sdword | Хранит целочисленный тип данных со знаком. |

Окончание таблицы 7.1

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке LEI-2023 | Тип идентификатора на языке ассемблера | Пояснение |
| bool | byte | Хранит булевой тип данных |
| string, char | byte | Хранит указатель на начало строки. Строка должна завешаться нулевым символом. |

Данное соответствие является ключевым для правильного выделения памяти и управления данными на уровне ассемблерного уровня, обеспечивая корректное выполнение программы на целевой архитектуре.

## **7.3 Статическая библиотека**

В языке LEI-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 outstr(char\* ptr) | Вывод на консоль строки ptr |
| void outbool(int value) | Вывод на консоль переменной value |
| void outnum(int value) | Вывод на консоль переменной value |
| void outstrline(char\* ptr) | Вывод на консоль строки ptr и перевод на новую строку |
| void outboolline(int value) | Вывод на консоль логической переменной value и перевод на новую строку |
| void outnumline(int value) | Вывод на консоль целочисленной переменной value и перевод на новую строку |
| void system\_pause() | Ожидание нажатия клавиши пользователем |
| int squere(int a) | Вычисление корня числа |
| int power(int a, int b) | Возведение числа a в степень b |

Генерация объявлений функций статической библиотеки автоматически в коде ассемблера обеспечивает прозрачность в использовании данных функций, а также позволяет компилятору корректно обрабатывать вызовы этих функций в процессе сборки программы на языке LEI-2023.

## **7.4 Особенности алгоритма генерации кода**

В языке LEI-2023 генерация кода строится на основе таблиц лексем и идентификаторов. Схема работы генератора кода представлена на рисунке 7.2



Рисунок 7.2 – Структура генератора кода

Генератор кода принимает эту информацию и формирует соответствующий выходной файл на языке ассемблера, который затем может быть выполнен для получения конечного результата программы.

## **7.5 Входные параметры генератора кода**

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке LEI-2023. Результаты работы генератора кода выводятся в файл с расширением .asm.

## **7.6 Контрольный пример**

Результат работы контрольного примера приведён на рисунке 7.3.

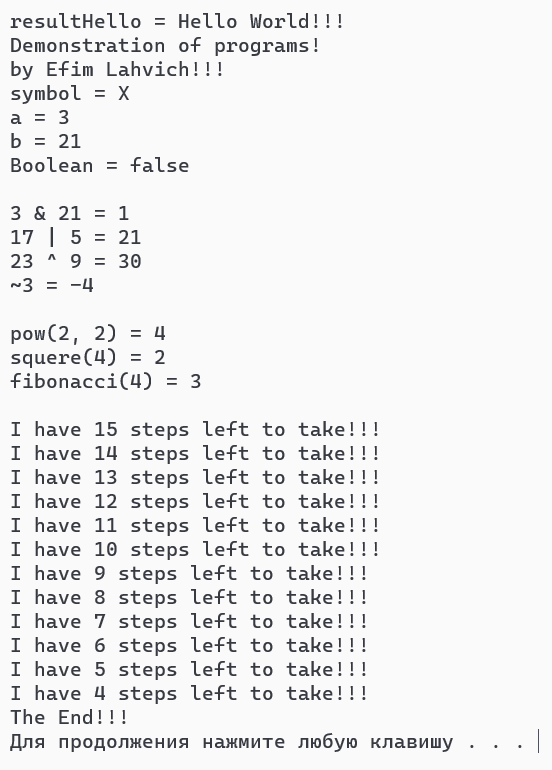


Рисунок 7.3 Результат работы программы на языке LEI-2023

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

# **8. Тестирование транслятора**

## **8.1 Общие положения**

При возникновении ошибки на каком-либо этапе трансляции, она обрабатывается в главном файле программы: ошибка выводится на консоль и записывается в файл логирования.

## **8.2 Результаты тестирования**

Результаты тестирования приведены в таблице 8.1

Таблица 8.1 – Результаты тестирования транслятора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| мэйн | Недопустимый символ в исходном файле(-in). Строка 1, позиция 1 |
| int fun f(int a, int b){  return a+b;  } | Семантическая ошибка 301: Отсутствует точка входа main. |
| Main {  int x = 69;  }  Main {  string y = "qwerty";  } | Семантическая ошибка 302: Несколько точек входа main. |
| Main{  int i = 2;  int i = 2;  } | Семантическая ошибка 305: Попытка переопределения идентификатора. Cтрока: 3, позиция: 9 |
| int fun f(int a, int b){  if(a==b||b ==0||a== 0)  then {return a\*2;}  f(a-1, b-1);  }  Main{  f(2, 6);  } | Семантическая ошибка 324: Нельзя использовать рекурсию, если не во всех случаях в функции имеет return. Cтрока: 1, позиция: 8 |
| int fun f(int a, int b){  if(a+b>0)  then {return true;}  return a+b;  }  Main{  f(2, 6);  } | Семантическая ошибка 315: Тип функции и возвращаемого значения не совпадают. Cтрока: 3, позиция: 11 |

Окончание таблицы 8.1

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| Main{  int i = 50000000000;  } | Семантическая ошибка 313: Недопустимый целочисленный литерал. Cтрока: 2, позиция: 1 |
| int fun f(int a, int b, int c, int d, int e, int f, int g, int t, int u, int i){  return a+b+c+d+e+f+g+t+u+i  }  Main{  f(1,2,3,4,5,6,7,8,9,10);  } | Семантическая ошибка 306: Превышено максимальное количество параметров функции Cтрока: 1, позиция: 8 |
| Main{  int i = 2  } | Ошибка 610: строка 2 позиция 9. Синтаксическая ошибка в программе. Неверная структура операторы программы. |
| Main{  int i = 2;  ;  } | Ошибка 610: строка 3 позиция 1. Синтаксическая ошибка в программе. Неверная структура операторы программы. |
| Main{  int i = 2\0;  } | Семантическая ошибка 320: Деление на ноль недопустимо. Cтрока: 2, позиция: 10 |

Результаты тестирования демонстрируют разнообразные сценарии обработки ошибок транслятором языка LEI-2023. Каждое диагностическое сообщение указывает на конкретный тип ошибки и предоставляет информацию о месте в исходном коде, где ошибка была обнаружена.

# **Заключение**

В ходе выполнения курсовой работы был разработан транслятор для языка программирования LEI-2023. Таким образом, были выполнены основные задачи данной курсовой работы:

* Сформулирована спецификация языка LEI-2023;
* разработан лексический анализатор;
* разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика;
* разработан синтаксический анализатор;
* разработан семантический анализатор;
* разработан транслятор кода на язык ассемблера;
* проведено тестирование транслятора.

Окончательная версия языка LEI-2023 включает:

* 3 типа данных;
* поддержка операторов вывода;
* возможность вызова функций стандартной библиотеки;
* наличие 5 арифметических операторов для вычисления выражений;
* поддержка функций, процедур, оператора 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. 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 A**

Исходный код программы на языке LEI-2023

|  |
| --- |
| string fun Hello()  {  return "Hello World!!!";  }  void fun Print()  {  WriteLine "Demonstration of programs!";  }  int fun fibonacci(int n)  {  if(n == 0)  then {return 0;}  if(n == 1 || n == 2)  then {return 1;}  return fibonacci(n-1)+fibonacci(n-2);  }  Main  {  string str = Hello();  WriteLine "resultHello = " + str;  Print();  str = "by Efim Lahvich!!!";  WriteLine str;  char symbol = 'X';  WriteLine "symbol = " + symbol;  int a = 3;  WriteLine "a = " + a;  int b = 0x015;  WriteLine "b = " + b;  bool Boolean = false;  WriteLine "Boolean = " + Boolean;  WriteLine " ";  int AND = a & b;  int OR = 17 | 5;  int XOR = 23 ^ 9;  int NOT = ~a;  WriteLine a + " & " + b + " = " + AND;  WriteLine "17 | 5 = " + OR;  WriteLine "23 ^ 9 = " + XOR;  WriteLine "~" + a + " = " + NOT;  WriteLine " ";  b = squere(4);  WriteLine "pow(2, 2) = " + pow(2, 2);  WriteLine "squere(4) = " + b;  WriteLine "fibonacci(4) = " + fibonacci(4);  WriteLine " ";  a = 0 + 3;  b = 15;  while(a != b)  {  WriteLine "I have " + b + " steps left to take!!!";  b = b - 1;  }  WriteLine "The End!!!";  } |

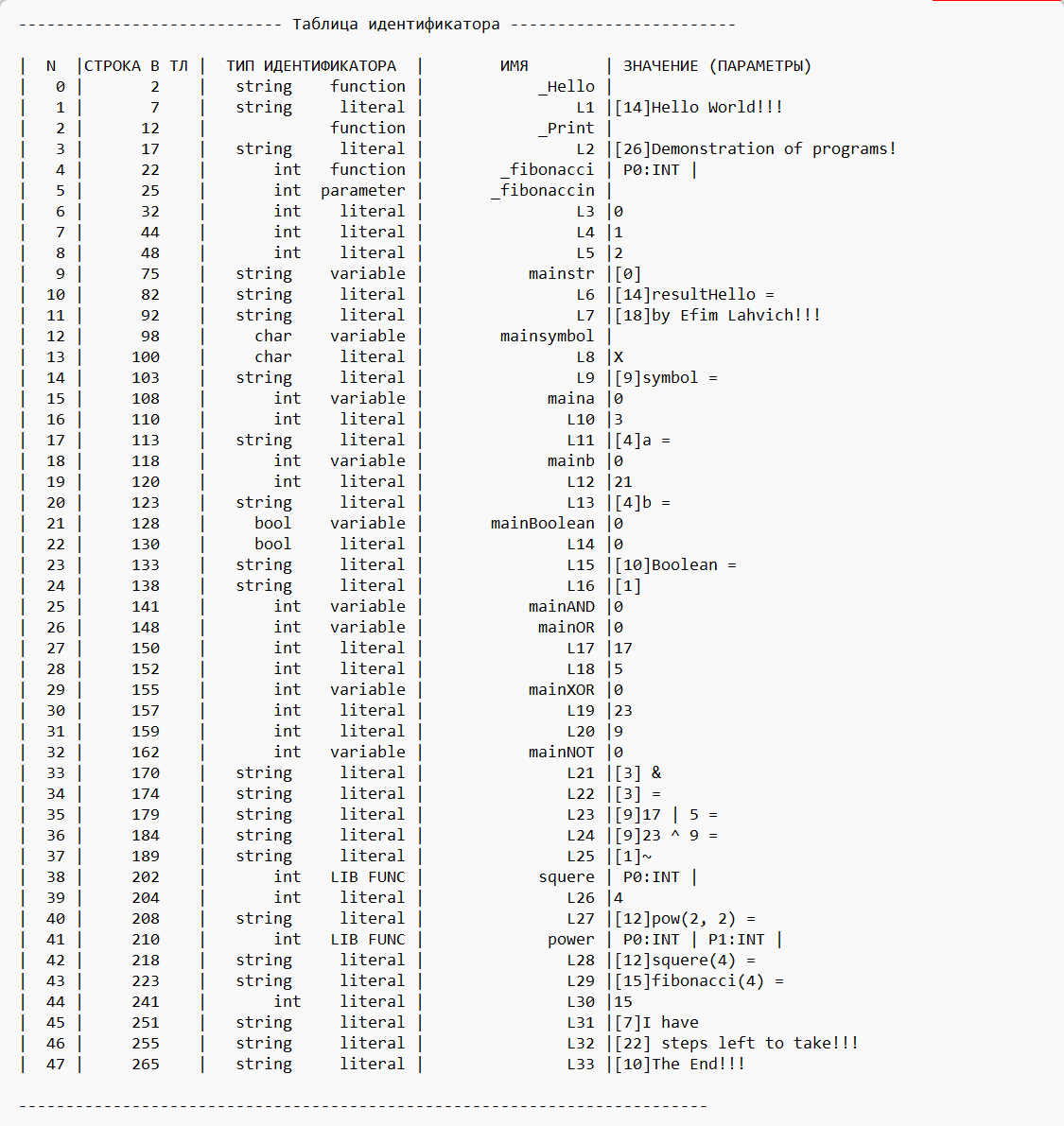
# **Приложение Б**

Таблица ошибок языка LEI-2023

|  |
| --- |
| ERROR errors[ERROR\_MAX\_ENTRY] =  {  //системные ошибки  ERROR\_ENTRY(0, "SYSTEM Недопустимый код ошибки"),  ERROR\_ENTRY(1, "SYSTEM Системный сбой"),  ERROR\_ENTRY\_NODEF(2), ERROR\_ENTRY\_NODEF(3), ERROR\_ENTRY\_NODEF(4), ERROR\_ENTRY\_NODEF(5),  ERROR\_ENTRY\_NODEF(6), ERROR\_ENTRY\_NODEF(7), ERROR\_ENTRY\_NODEF(8), ERROR\_ENTRY\_NODEF(9),  ERROR\_ENTRY\_NODEF10(10), ERROR\_ENTRY\_NODEF10(20), ERROR\_ENTRY\_NODEF10(30), ERROR\_ENTRY\_NODEF10(40),  ERROR\_ENTRY\_NODEF10(50), ERROR\_ENTRY\_NODEF10(60), ERROR\_ENTRY\_NODEF10(70), ERROR\_ENTRY\_NODEF10(80), ERROR\_ENTRY\_NODEF10(90),  ERROR\_ENTRY(100, "SYSTEM Параметр -in должен быть задан"),  ERROR\_ENTRY(101, "SYSTEM Превышена длина входного параметра"),  ERROR\_ENTRY(102, "SYSTEM Ошибка при открытии файла с исходным кодом(-in)"),  ERROR\_ENTRY(103, "SYSTEM Ошибка при создании файла протокола(-log)"),  ERROR\_ENTRY\_NODEF(104), ERROR\_ENTRY\_NODEF(105), ERROR\_ENTRY\_NODEF(106), ERROR\_ENTRY\_NODEF(107), ERROR\_ENTRY\_NODEF(108), ERROR\_ENTRY\_NODEF(109),  ERROR\_ENTRY\_NODEF10(110), ERROR\_ENTRY\_NODEF10(120), ERROR\_ENTRY\_NODEF10(130), ERROR\_ENTRY\_NODEF10(140),  ERROR\_ENTRY\_NODEF10(150), ERROR\_ENTRY\_NODEF10(160), ERROR\_ENTRY\_NODEF10(170), ERROR\_ENTRY\_NODEF10(180), ERROR\_ENTRY\_NODEF10(190),  //лексические ошибки  ERROR\_ENTRY(200, "ERROR 200 Недопустимый символ в исходном файле(-in)"),  ERROR\_ENTRY(201, "ERROR 201 Неизвестная последовательность символов"),  ERROR\_ENTRY(202, "ERROR 202 Превышен размер таблицы лексем"),  ERROR\_ENTRY(203, "ERROR 203 Превышен размер таблицы идентификаторов"),  ERROR\_ENTRY\_NODEF(204), ERROR\_ENTRY\_NODEF(205), ERROR\_ENTRY\_NODEF(206), ERROR\_ENTRY\_NODEF(207), ERROR\_ENTRY\_NODEF(208), ERROR\_ENTRY\_NODEF(209),  ERROR\_ENTRY\_NODEF10(210), ERROR\_ENTRY\_NODEF10(220), ERROR\_ENTRY\_NODEF10(230), ERROR\_ENTRY\_NODEF10(240),  ERROR\_ENTRY\_NODEF10(250), ERROR\_ENTRY\_NODEF10(260), ERROR\_ENTRY\_NODEF10(270), ERROR\_ENTRY\_NODEF10(280), ERROR\_ENTRY\_NODEF10(290),  //семантические ошибки  ERROR\_ENTRY(300, "ERROR 300 Идентификатор не объявлен"),  ERROR\_ENTRY(301, "ERROR 301 Отсутствует точка входа main"),  ERROR\_ENTRY(302, "ERROR 302 Несколько точек входа main"),  ERROR\_ENTRY(303, "ERROR 303 Не указан тип идентификатора при объявлении"),  ERROR\_ENTRY(304, "ERROR 304 Отсутствует ключевое слово при объявлении"),  ERROR\_ENTRY(305, "ERROR 305 Попытка переопределения идентификатора"),  ERROR\_ENTRY(306, "ERROR 306 Превышено максимальное количество параметров функции"),  ERROR\_ENTRY(307, "ERROR 307 Слишком много параметров в вызове функции"),  ERROR\_ENTRY(308, "ERROR 308 Кол-во ожидаемых функцией и передаваемых параметров не совпадают"),  ERROR\_ENTRY(309, "ERROR 309 Несовпадение типов передаваемых параметров"),  ERROR\_ENTRY(310, "ERROR 310 Использование пустого строкового литерала недопустимо"),  ERROR\_ENTRY(311, "ERROR 311 Обнаружен символ \'\"\'. Возможно, не закрыт строковый литерал"),  ERROR\_ENTRY(312, "ERROR 312 Превышен размер строкового литерала"),  ERROR\_ENTRY(313, "ERROR 313 Недопустимый целочисленный литерал"),  ERROR\_ENTRY(314, "ERROR 314 Типы данных в выражении не совпадают с типом переменной"),  ERROR\_ENTRY(315, "ERROR 315 Тип функции и возвращаемого значения не совпадают"),  ERROR\_ENTRY(316, "ERROR 316 Недопустимое строковое выражение после \'=\'"),  ERROR\_ENTRY(317, "ERROR 317 Не закрыт блок main"),  ERROR\_ENTRY(318, "ERROR 318 Логическое сравнение со строками может быть только == или !="),  ERROR\_ENTRY(319, "ERROR 319 В выражений возник конфликт типов. Невозможно приведение типов"),  ERROR\_ENTRY(320, "ERROR 320 Деление на ноль недопустимо"),  ERROR\_ENTRY(321, "ERROR 321 Операция ! и ~ допустимо только для операнда целочисленного типа"),  ERROR\_ENTRY(322, "ERROR 322 В условном выражении возникла ошибка"),  ERROR\_ENTRY\_NODEF(323), ERROR\_ENTRY\_NODEF(324), ERROR\_ENTRY\_NODEF(325),  ERROR\_ENTRY\_NODEF(326), ERROR\_ENTRY\_NODEF(327), ERROR\_ENTRY\_NODEF(328), ERROR\_ENTRY\_NODEF(329),  ERROR\_ENTRY\_NODEF10(330),ERROR\_ENTRY\_NODEF10(340),ERROR\_ENTRY\_NODEF10(350),  ERROR\_ENTRY\_NODEF10(360),ERROR\_ENTRY\_NODEF10(370),ERROR\_ENTRY\_NODEF10(380),ERROR\_ENTRY\_NODEF10(390),  ERROR\_ENTRY\_NODEF100(400), ERROR\_ENTRY\_NODEF100(500),  //синтаксические ошибки  ERROR\_ENTRY(600, "ERROR 600 Ошибка. Неверная структура программы"),  ERROR\_ENTRY(601, "ERROR 601 Ошибка. Не найден список параметров функции"),  ERROR\_ENTRY(602, "ERROR 602 Ошибка в теле функции"),  ERROR\_ENTRY(603, "ERROR 603 Ошибка в теле процедуры"),  ERROR\_ENTRY(604, "ERROR 604 Ошибка в списке параметров функции"),  ERROR\_ENTRY(605, "ERROR 605 Ошибка в вызове функции/выражении"),  ERROR\_ENTRY(606, "ERROR 606 Ошибка в списке фактических параметров функции"),  ERROR\_ENTRY(607, "ERROR 607 Ошибка. Неверный арифметический оператор"),  ERROR\_ENTRY(608, "ERROR 608 Ошибка. Неверное выражение. Ожидаются только идентификаторы/литералы"),  ERROR\_ENTRY(609, "ERROR 609 Ошибка в арифметическом выражении"),  ERROR\_ENTRY(610, "ERROR 610 Ошибка. Недопустимая синтаксическая конструкция"),  ERROR\_ENTRY\_NODEF(611), ERROR\_ENTRY\_NODEF(612), ERROR\_ENTRY\_NODEF(613), ERROR\_ENTRY\_NODEF(614), ERROR\_ENTRY\_NODEF(615),  ERROR\_ENTRY\_NODEF(616), ERROR\_ENTRY\_NODEF(617), ERROR\_ENTRY\_NODEF(618), ERROR\_ENTRY\_NODEF(619),  ERROR\_ENTRY\_NODEF10(620), ERROR\_ENTRY\_NODEF10(630), ERROR\_ENTRY\_NODEF10(640), ERROR\_ENTRY\_NODEF10(650),  ERROR\_ENTRY\_NODEF10(660), ERROR\_ENTRY\_NODEF10(670), ERROR\_ENTRY\_NODEF10(680), ERROR\_ENTRY\_NODEF10(690),  ERROR\_ENTRY\_NODEF100(700), ERROR\_ENTRY\_NODEF100(800), ERROR\_ENTRY\_NODEF100(900)  }; |

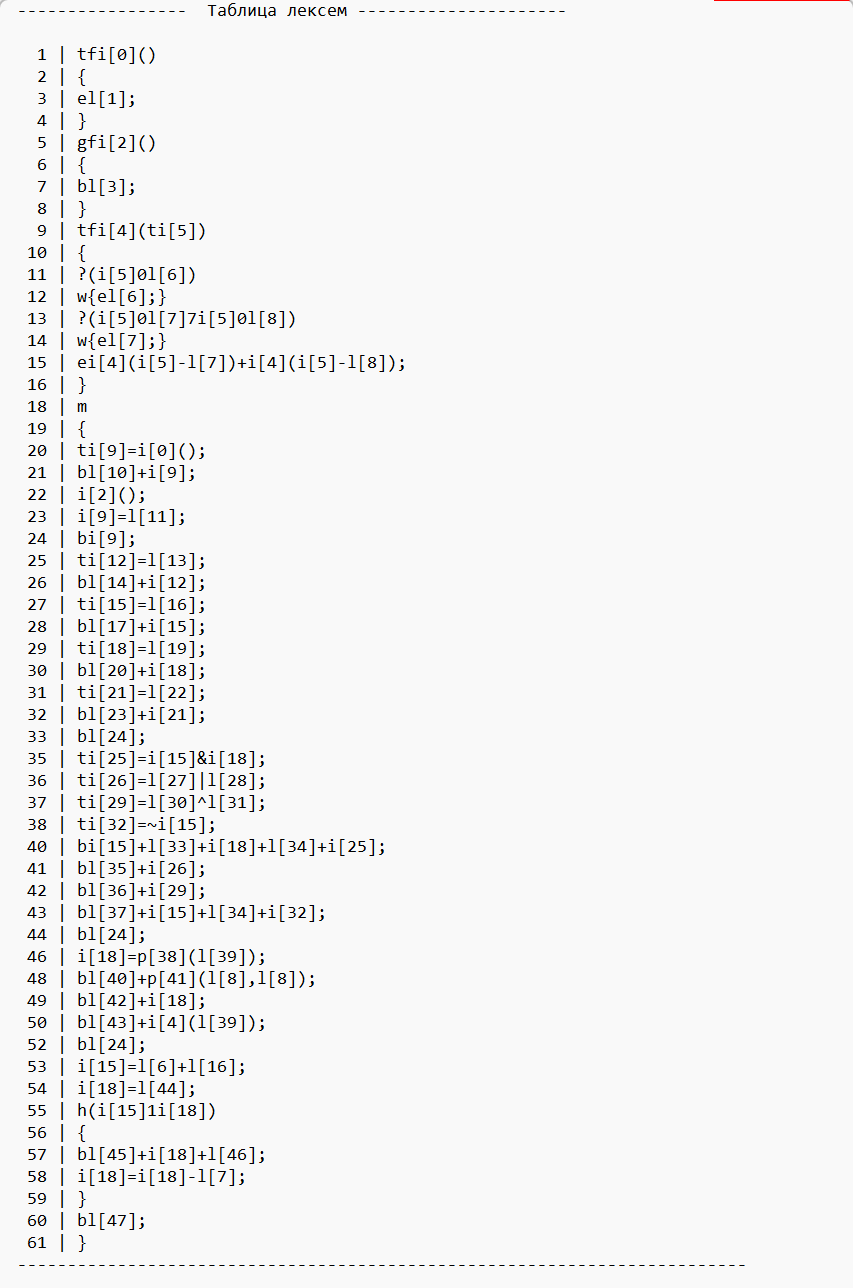
# **Приложение В**

Таблица идентификаторов контрольного примера



# **Приложение Г**

Лексемы соответствующие контрольному примеру



# **Приложение Д**

Грамматика языка

|  |
| --- |
| Greibach greibach(NS('S'), TS('$'), 14,  /\*1\*/ Rule(NS('S'), GRB\_ERROR\_SERIES, 3, // Объявление функции  // S → tfiPTS | gfiPTS | m{K}  Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('P'), NS('T'), NS('S')),  Rule::Chain(6, TS('g'), TS('f'), TS('i'), NS('P'), NS('T'), NS('S')),  Rule::Chain(4, TS('m'), TS('{'), NS('K'), TS('}'))  ),  /\*2\*/ Rule(NS('T'), GRB\_ERROR\_SERIES + 2, 4, // Тело функции  // T → {eW;} | {KeW;} | {e;} | {Kew;}  Rule::Chain(5, TS('{'), TS('e'), NS('W'), TS(';'), TS('}')),  Rule::Chain(6, TS('{'), NS('K'), TS('e'), NS('W'), TS(';'), TS('}')),  Rule::Chain(4, TS('{'), TS('e'), TS(';'), TS('}')),  Rule::Chain(5, TS('{'), NS('K'), TS('e'), TS(';'), TS('}'))  ),  /\*3\*/ Rule(NS('P'), GRB\_ERROR\_SERIES + 1, 2, // Тело параметров функции при его объявлений  // P → () | (E)  Rule::Chain(3, TS('('), NS('E'), TS(')')),  Rule::Chain(2, TS('('), TS(')'))  ),  /\*4\*/ Rule(NS('E'), GRB\_ERROR\_SERIES + 3, 2, // Параметры функции при его объявлений  // E → ti,E | ti  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('E')),  Rule::Chain(2, TS('t'), TS('i'))  ),  /\*5\*/ Rule(NS('F'), GRB\_ERROR\_SERIES + 4, 2, // Тело параметров функции  // F → (N) | ()  Rule::Chain(3, TS('('), NS('N'), TS(')')),  Rule::Chain(2, TS('('), TS(')'))  ),  /\*6\*/ Rule(NS('N'), GRB\_ERROR\_SERIES + 5, 32, // Параметры функции (некоторые параметры могут быть выражением)  // N → i | l | (W) | (-i) | iF | pF | ~W | !W  // N → iOW | lOW | (W)OW | (-i)OW | iFOW | pFOW | ~WOW | !WOW  // N → i,N | l,N | (W),N | (-i),N | iF,N | pF,N | ~W,N | !W,N  // N → iOW,N | lOW,N | (W)OW,N | (-i)OW,N | iFOW,N | pFOW,N | ~WOW,N | !WOW,N  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('('), NS('W'), TS(')')),  Rule::Chain(4, TS('('), TS('-'), TS('i'), TS(')')),  Rule::Chain(2, TS('i'), NS('F')),  Rule::Chain(2, TS('p'), NS('F')),  Rule::Chain(2, TS('~'), NS('W')),  Rule::Chain(2, TS('!'), NS('W')),  Rule::Chain(3, TS('i'), NS('O'), NS('W')),  Rule::Chain(3, TS('l'), NS('O'), NS('W')),  Rule::Chain(5, TS('('), NS('W'), TS(')'), NS('O'), NS('W')),  Rule::Chain(6, TS('('), TS('-'), TS('i'), TS(')'), NS('O'), NS('W')),  Rule::Chain(4, TS('i'), NS('F'), NS('O'), NS('W')),  Rule::Chain(4, TS('p'), NS('F'), NS('O'), NS('W')),  Rule::Chain(4, TS('~'), NS('W'), NS('O'), NS('W')),  Rule::Chain(4, TS('!'), NS('W'), NS('O'), NS('W')),  Rule::Chain(3, TS('i'), TS(','), NS('N')),  Rule::Chain(3, TS('l'), TS(','), NS('N')),  Rule::Chain(5, TS('('), NS('W'), TS(')'), TS(','), NS('N')),  Rule::Chain(6, TS('('), TS('-'), TS('i'), TS(')'), TS(','), NS('N')),  Rule::Chain(4, TS('i'), NS('F'), TS(','), NS('N')),  Rule::Chain(4, TS('p'), NS('F'), TS(','), NS('N')),  Rule::Chain(4, TS('~'), NS('W'), TS(','), NS('N')),  Rule::Chain(4, TS('!'), NS('W'), TS(','), NS('N')),  Rule::Chain(5, TS('i'), NS('O'), NS('W'), TS(','), NS('N')),  Rule::Chain(5, TS('l'), NS('O'), NS('W'), TS(','), NS('N')),  Rule::Chain(7, TS('('), NS('W'), TS(')'), NS('O'), NS('W'), TS(','), NS('N')),  Rule::Chain(8, TS('('), TS('-'), TS('i'), TS(')'), NS('O'), NS('W'), TS(','), NS('N')),  Rule::Chain(6, TS('i'), NS('F'), NS('O'), NS('W'), TS(','), NS('N')),  Rule::Chain(6, TS('p'), NS('F'), NS('O'), NS('W'), TS(','), NS('N')),  Rule::Chain(6, TS('~'), NS('W'), NS('O'), NS('W'), TS(','), NS('N')),  Rule::Chain(6, TS('!'), NS('W'), NS('O'), NS('W'), TS(','), NS('N'))  ),  /\*7\*/ Rule(NS('R'), GRB\_ERROR\_SERIES + 6, 4, // Условная конструкция  // R → rY | wY | rYwY | wYrY  Rule::Chain(2, TS('r'), NS('Y')),  Rule::Chain(2, TS('w'), NS('Y')),  Rule::Chain(4, TS('r'), NS('Y'), TS('w'), NS('Y')),  Rule::Chain(4, TS('w'), NS('Y'), TS('r'), NS('Y'))  ),  /\*8\*/ Rule(NS('O'), GRB\_ERROR\_SERIES + 9, 15, // Операторы  // O → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 // логические  // O → + | - | \* | / | % | & | | // арифметические и побитовые  Rule::Chain(1, TS('0')),  Rule::Chain(1, TS('1')),  Rule::Chain(1, TS('2')),  Rule::Chain(1, TS('3')),  Rule::Chain(1, TS('4')),  Rule::Chain(1, TS('5')),  Rule::Chain(1, TS('6')),  Rule::Chain(1, TS('7')),  Rule::Chain(1, TS('+')),  Rule::Chain(1, TS('-')),  Rule::Chain(1, TS('\*')),  Rule::Chain(1, TS('%')),  Rule::Chain(1, TS('/')),  Rule::Chain(1, TS('&')),  Rule::Chain(1, TS('|'))  ),  /\*9\*/ Rule(NS('W'), GRB\_ERROR\_SERIES + 11, 16, // Выражение  // W → i | l | (W) | (-i) | iF | pF | ~W | !W |  // W → iOW | lOW | (W)OW | (-i)OW | iFOW | pFOW | ~WOW | !WOW |  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('('), NS('W'), TS(')')),  Rule::Chain(4, TS('('), TS('-'), TS('i'), TS(')')),  Rule::Chain(2, TS('i'), NS('F')),  Rule::Chain(2, TS('p'), NS('F')),  Rule::Chain(2, TS('~'), NS('W')),  Rule::Chain(2, TS('!'), NS('W')),  Rule::Chain(3, TS('i'), NS('O'), NS('W')),  Rule::Chain(3, TS('l'), NS('O'), NS('W')),  Rule::Chain(6, TS('('), TS('-'), TS('i'), TS(')'), NS('O'), NS('W')),  Rule::Chain(5, TS('('), NS('W'), TS(')'), NS('O'), NS('W')),  Rule::Chain(4, TS('i'), NS('F'), NS('O'), NS('W')),  Rule::Chain(4, TS('p'), NS('F'), NS('O'), NS('W')),  Rule::Chain(4, TS('~'), NS('W'), NS('O'), NS('W')),  Rule::Chain(4, TS('!'), NS('W'), NS('O'), NS('W'))  ),  /\*10\*/ Rule(NS('K'), GRB\_ERROR\_SERIES + 12, 20, //Операторы программы  // Z → ti=W; | i=W; | ti; | iF; | pF; | oB; | ^B; | ?(W)R | h(W)Y | dYh(W);  // Z → ti=W;K | i=W;K | ti;K | iF;K | pF;K | oB;K | ^B;K | ?(W)RK | h(W)YK | dYh(W);K  Rule::Chain(5, TS('t'), TS('i'), TS('='), NS('W'), TS(';')),  Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')),  Rule::Chain(3, TS('t'), TS('i'), TS(';')),  Rule::Chain(3, TS('i'), NS('F'), TS(';')),  Rule::Chain(3, TS('p'), NS('F'), TS(';')),  Rule::Chain(3, TS('o'), NS('B'), TS(';')),  Rule::Chain(3, TS('^'), NS('B'), TS(';')),  Rule::Chain(5, TS('?'), TS('('), NS('W'), TS(')'), NS('R')),  Rule::Chain(5, TS('h'), TS('('), NS('W'), TS(')'), NS('Y')),  Rule::Chain(7, TS('d'), NS('Y'), TS('h'), TS('('), NS('W'), TS(')'), TS(';')),  Rule::Chain(6, TS('t'), TS('i'), TS('='), NS('W'), TS(';'), NS('K')),  Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('K')),  Rule::Chain(4, TS('t'), TS('i'), TS(';'), NS('K')),  Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('K')),  Rule::Chain(4, TS('p'), NS('F'), TS(';'), NS('K')),  Rule::Chain(4, TS('o'), NS('B'), TS(';'), NS('K')),  Rule::Chain(4, TS('^'), NS('B'), TS(';'), NS('K')),  Rule::Chain(6, TS('?'), TS('('), NS('W'), TS(')'), NS('R'), NS('K')),  Rule::Chain(6, TS('h'), TS('('), NS('W'), TS(')'), NS('Y'), NS('K')),  Rule::Chain(8, TS('d'), NS('Y'), TS('h'), TS('('), NS('W'), TS(')'), TS(';'), NS('K'))  ),  /\*11\*/ Rule(NS('B'), GRB\_ERROR\_SERIES + 12, 8, //Выводы;  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(2, TS('i'), NS('F')),  Rule::Chain(2, TS('p'), NS('F')),  Rule::Chain(3, TS('i'), TS('+'), NS('B')),  Rule::Chain(3, TS('l'), TS('+'), NS('B')),  Rule::Chain(4, TS('p'), NS('F'), TS('+'), NS('B')),  Rule::Chain(4, TS('p'), NS('F'), TS('+'), NS('B'))  ),  /\*12\*/ Rule(NS('Y'), GRB\_ERROR\_SERIES + 16, 6, // Тело конструкции  Rule::Chain(6, TS('{'), NS('K'), TS('e'), NS('W'), TS(';'), TS('}')),  Rule::Chain(5, TS('{'), NS('K'), TS('e'), TS(';'), TS('}')),  Rule::Chain(3, TS('{'), NS('K'), TS('}')),  Rule::Chain(5, TS('{'), TS('e'), NS('W'), TS(';'), TS('}')),  Rule::Chain(4, TS('{'), TS('e'), TS(';'), TS('}')),  Rule::Chain(2, TS('{'), TS('}'))  ),  /\*13\*/ Rule(NS('V'), GRB\_ERROR\_SERIES + 15, 2, // Переменные  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l'))  ),  /\*14\*/ Rule(NS('U'), GRB\_ERROR\_SERIES + 3, 2, // Функция  Rule::Chain(2, TS('i'), NS('F')),  Rule::Chain(2, TS('p'), NS('F'))  )  ); |

# **Приложение Е**

Разбор исходного кода синтаксическим анализатором

|  |  |
| --- | --- |
|  |  |

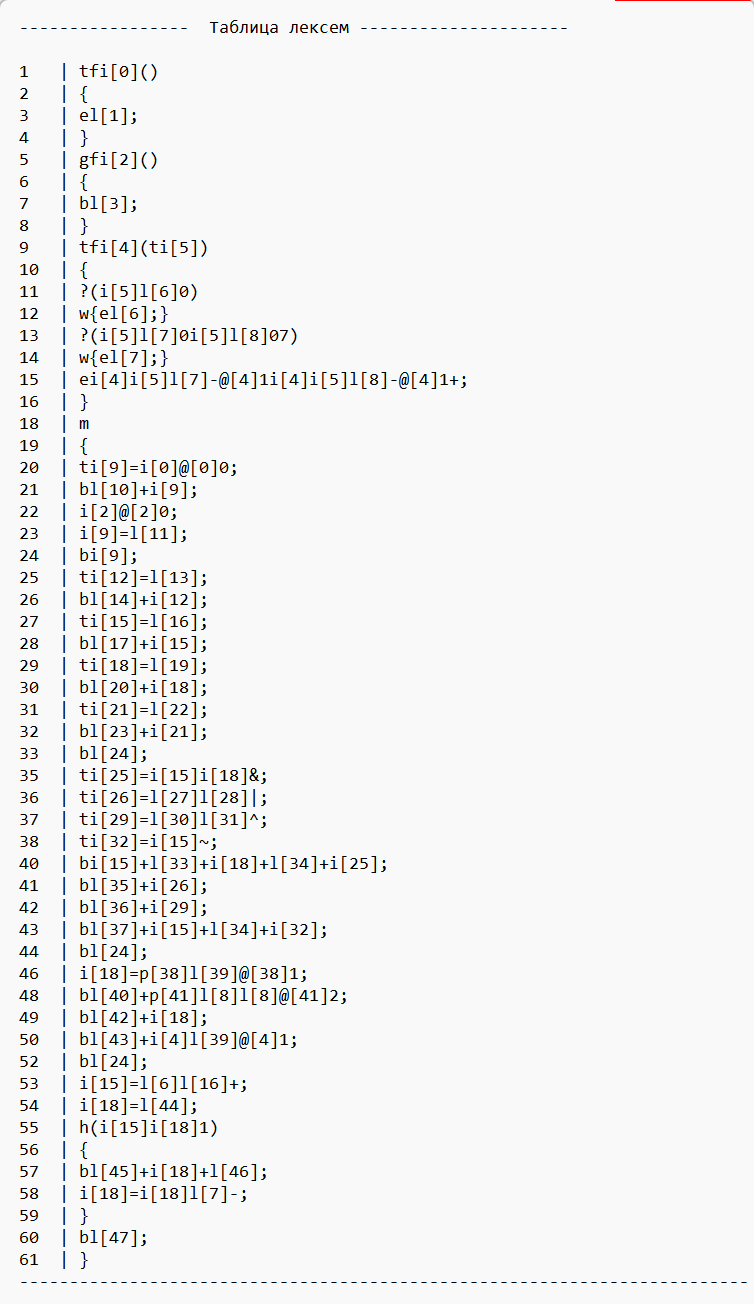
# **Приложение Ж**

Программная реализация механизма преобразования в ПОЛИЗ

|  |
| --- |
| int getPriority(unsigned char e)  {  switch (e)  {  case LEX\_LEFTHESIS: case LEX\_RIGHTTHESIS: return 0;  case LEX\_COMMA: return 1;  case LEX\_LOGIC\_OR: return 2;  case LEX\_LOGIC\_AND: return 3;  case LEX\_OR: return 4;  case LEX\_XOR: return 5;  case LEX\_AND: return 6;  case LEX\_LOGIC\_EQUALS: case LEX\_LOGIC\_NOT\_EQUALS: return 7;  case LEX\_LOGIC\_LESS: case LEX\_LOGIC\_LESS\_EQ: case LEX\_LOGIC\_MORE: case LEX\_LOGIC\_MORE\_EQ: return 8;  case LEX\_PLUS: case LEX\_MINUS: return 9;  case LEX\_STAR: case LEX\_DIRSLASH: case LEX\_PROCENT: return 10;  case LEX\_NOT: case LEX\_LOGIC\_NOT: return 11;  }  }  int searchExpression(Lex::LEX lex) {  bool fl = false, inEqual = false, inIf = false;  int inThesis = 0;  Lex::LEX result;  result.lextable = LT::Create(lex.lextable.size);  result.idtable = IT::Create(lex.idtable.size);  for (int i = 0, k = 0; i < lex.lextable.size; i++)  {  if (inEqual)  {  if (lex.lextable.table[i].lexema == LEX\_SEPARATOR)  {  inEqual = false;  }  continue;  }  if (inIf)  {  if (lex.lextable.table[i].lexema == LEX\_RIGHTTHESIS)  {  if (inThesis == 0)  {  inIf = false;  continue;  }  inThesis--;  }  if (lex.lextable.table[i].lexema == LEX\_LEFTHESIS)  {  inThesis++;  }  continue;  }  if (lex.lextable.table[i].lexema == LEX\_EQUAL || lex.lextable.table[i].lexema == LEX\_RETURN) {  lex.lextable.size = PolishNotation(++i, lex, Semantic::Equals);  inEqual = true;  }  else if(lex.lextable.table[i].lexema == LEX\_ID  && lex.lextable.table[i-1].lexema != LEX\_FUNCTION  && (lex.idtable.table[lex.lextable.table[i].idxTI].idtype == IT::IDTYPE::F  || lex.idtable.table[lex.lextable.table[i].idxTI].idtype == IT::IDTYPE::S)) {  lex.lextable.size = PolishNotation((i+=2), lex, Semantic::Fun);  }  else if (lex.lextable.table[i].lexema == LEX\_IF || lex.lextable.table[i].lexema == LEX\_WHILE) {  lex.lextable.size = PolishNotation((i+=2), lex, Semantic::If);  inIf = true;  }    }  return lex.lextable.size;  }  int PolishNotation(int lextable\_pos, Lex::LEX& lex, Semantic::Is n)  {  stack<LT::Entry> stack;  queue<LT::Entry> queue;  LT::Entry temp; temp.idxTI = -1; temp.lexema = '#'; temp.sn = -1;  LT::Entry tilda; tilda.idxTI = -1; tilda.lexema = '`'; tilda.sn = -1;  LT::Entry endFun; endFun.idxTI = -1; endFun.lexema = '|';endFun.sn = -1;  int countLex = 0;  int countParm = 0;  int posLex = lextable\_pos;  bool findFunc = false;  bool flagthethis = false;  int inThesis = 0;  char\* buf = new char[2];  for (int i = lextable\_pos;  ((lex.lextable.table[i].lexema != LEX\_COMMA && lex.lextable.table[i].lexema != LEX\_RIGHTTHESIS) && Semantic::Fun == n)  || (lex.lextable.table[i].lexema != LEX\_SEPARATOR && Semantic::Equals == n)  || ((lex.lextable.table[i].lexema != LEX\_RIGHTTHESIS || inThesis != 0) && Semantic::If == n);  i++, countLex++)  {  switch (lex.lextable.table[i].lexema)  {  case LEX\_STDFUNC:  case LEX\_ID:  {  if(lex.idtable.table[lex.lextable.table[i].idxTI].idtype == IT::IDTYPE::F || lex.idtable.table[lex.lextable.table[i].idxTI].idtype == IT::IDTYPE::S)  {  queue.push(lex.lextable.table[i++]);  queue.push(lex.lextable.table[i++]);  countLex += 2;  stack.push(endFun);  while(lex.lextable.table[i].lexema != LEX\_RIGHTTHESIS)  {  if (lex.lextable.table[i].lexema == LEX\_RIGHTTHESIS)  {  queue.push(lex.lextable.table[i]);  break;  }  if (lex.lextable.table[i + 1].lexema != LEX\_COMMA && lex.lextable.table[i + 1].lexema != LEX\_RIGHTTHESIS)  {  lex.lextable.size = PolishNotation(i, lex, Semantic::Fun);  int countThesis = 0;  int start = i;  while ((lex.lextable.table[i].lexema != LEX\_COMMA && lex.lextable.table[i].lexema != LEX\_RIGHTTHESIS) || countThesis != 0)  {  if(lex.lextable.table[i].lexema == LEX\_LEFTHESIS)  countThesis++;  if (lex.lextable.table[i].lexema == LEX\_RIGHTTHESIS)  countThesis--;  i++;  }  int end = i;  for (int j = end-1; j >= start; j--)  {  countLex++;  stack.push(lex.lextable.table[j]);  }  }  else  {  stack.push(lex.lextable.table[i++]);  countLex++;  }  if (lex.lextable.table[i].lexema == LEX\_COMMA)  {  countLex++;  stack.push(lex.lextable.table[i]);  }  else  break;  i++;  }  while (stack.top().lexema != '|')  {  queue.push(stack.top());  stack.pop();  }  stack.pop();  queue.push(lex.lextable.table[i]);  }  else  queue.push(lex.lextable.table[i]);  continue;  }  case LEX\_LITERAL:  {  queue.push(lex.lextable.table[i]);  continue;  }  case LEX\_LEFTHESIS:  {  if (lex.lextable.table[i + 1].lexema == LEX\_MINUS && lex.lextable.table[i + 2].lexema == LEX\_ID && !findFunc)  {  flagthethis = true;  continue;  }  stack.push(lex.lextable.table[i]);  if (Semantic::If == n)  inThesis++;  continue;  }  case LEX\_RIGHTTHESIS:  {  while (stack.top().lexema != LEX\_LEFTHESIS)  {  queue.push(stack.top());  stack.pop();    }  if (Semantic::If == n)  inThesis--;  if (stack.empty())  return 0;  stack.pop();  continue;  }  case LEX\_PLUS: case LEX\_MINUS: case LEX\_STAR: case LEX\_DIRSLASH: case LEX\_PROCENT: case LEX\_AND: case LEX\_OR: case LEX\_NOT:  case LEX\_LOGIC\_OR: case LEX\_LOGIC\_AND: case LEX\_LOGIC\_NOT: case LEX\_LOGIC\_EQUALS: case LEX\_LOGIC\_NOT\_EQUALS: case LEX\_LOGIC\_LESS:  case LEX\_LOGIC\_LESS\_EQ: case LEX\_LOGIC\_MORE: case LEX\_LOGIC\_MORE\_EQ:  {  if (flagthethis)  {  tilda.sn = lex.lextable.table[i].sn;  lex.lextable.table[i] = tilda;  queue.push(lex.lextable.table[i + 1]);  queue.push(lex.lextable.table[i]);  flagthethis = false;  i += 2;  countLex += 2;  continue;  }  while (!stack.empty() && getPriority(lex.lextable.table[i].lexema) <= getPriority(stack.top().lexema))  {  queue.push(stack.top());  stack.pop();  }  stack.push(lex.lextable.table[i]);  continue;  }  }  }  while (!stack.empty())  {  if (stack.top().lexema == LEX\_LEFTHESIS || stack.top().lexema == LEX\_RIGHTTHESIS)  return 0;  queue.push(stack.top());  stack.pop();  }  cout << ++countExpression << ". - ";  while (countLex != 0)  {  if (!queue.empty()) {  lex.lextable.table[posLex++] = queue.front();  if(lex.lextable.table[posLex - 1].lexema != '#')  cout << lex.lextable.table[posLex - 1].lexema << " ";  queue.pop();    }  else  {  lex.lextable.table[posLex++] = temp;  }  countLex--;  }  cout << "\n";  for (int i = 0; i < posLex; i++)  {  if (lex.lextable.table[i].lexema == LEX\_LITERAL)  lex.idtable.table[lex.lextable.table[i].idxTI].idxfirstLE = i;  }  return lex.lextable.size;  } |

# **Приложение И**

Лексемы соответствующие контрольному примеру после преобразования выражений в ПОЛИЗ



# **Приложение К**

Результат генерации кода контрольного примера в Ассемблер

|  |
| --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib "..\Debug\Library.lib"  ExitProcess PROTO:DWORD  .stack 4096  outnum PROTO : DWORD  outstr PROTO : DWORD  outstrline PROTO : DWORD  outnumline PROTO : DWORD  outbool PROTO : DWORD  outboolline PROTO : DWORD  inttostr PROTO : DWORD  system\_pause PROTO  squere PROTO : DWORD  strLen PROTO : DWORD  inputInt PROTO  inputString PROTO  inputChar PROTO  power PROTO : DWORD, : DWORD  codetochar PROTO : DWORD  .const  null\_division BYTE 'ERROR: DIVISION BY ZERO', 0  overflow BYTE 'ERROR: VARIABLE OVERFLOW', 0  L1 BYTE 'Hello World!!!', 0  L2 BYTE 'Demonstration of programs!', 0  L3 SDWORD 0  L4 SDWORD 1  L5 SDWORD 2  L6 BYTE 'resultHello = ', 0  L7 BYTE 'by Efim Lahvich!!!', 0  L8 BYTE 'X', 0  L9 BYTE 'symbol = ', 0  L10 SDWORD 3  L11 BYTE 'a = ', 0  L12 SDWORD 21  L13 BYTE 'b = ', 0  L14 BYTE 0  L15 BYTE 'Boolean = ', 0  L16 BYTE ' ', 0  L17 SDWORD 17  L18 SDWORD 5  L19 SDWORD 23  L20 SDWORD 9  L21 BYTE ' & ', 0  L22 BYTE ' = ', 0  L23 BYTE '17 | 5 = ', 0  L24 BYTE '23 ^ 9 = ', 0  L25 BYTE '~', 0  L26 SDWORD 4  L27 BYTE 'pow(2, 2) = ', 0  L28 BYTE 'squere(4) = ', 0  L29 BYTE 'fibonacci(4) = ', 0  L30 SDWORD 15  L31 BYTE 'I have ', 0  L32 BYTE ' steps left to take!!!', 0  L33 BYTE 'The End!!!', 0  .data  mainstr DWORD ?  mainsymbol DWORD ?  maina SDWORD 0  mainb SDWORD 0  mainBoolean DWORD 0  mainAND SDWORD 0  mainOR SDWORD 0  mainXOR SDWORD 0  mainNOT SDWORD 0  .code  \_Hello PROC  push ebx  push edx  push ecx  push offset L1  pop eax  jmp return\_Hello  return\_Hello:  pop ecx  pop edx  pop ebx  ret  SOMETHINGWRONG:  push offset null\_division  call outstrline  call system\_pause  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstrline  call system\_pause  push -2  call ExitProcess  \_Hello ENDP  \_Print PROC  push ebx  push edx  push ecx  push offset L2  call outstrline  return\_Print:  pop ecx  pop edx  pop ebx  ret  SOMETHINGWRONG:  push offset null\_division  call outstrline  call system\_pause  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstrline  call system\_pause  push -2  call ExitProcess  \_Print ENDP  \_fibonacci PROC \_fibonaccin : SDWORD  push ebx  push edx  push ecx  if1do:  push \_fibonaccin  push L3  pop eax  pop ebx  cmp eax, ebx  je if1Equal1True  mov eax, 0  push eax  jmp if1Equal1End  if1Equal1True:  mov eax, 1  push eax  if1Equal1End:  pop eax  cmp eax, 0  jne if1true  je if1End  if1true:  push L3  pop eax  jmp return\_fibonacci  jmp if1End  if1End:  if2do:  push \_fibonaccin  push L4  pop eax  pop ebx  cmp eax, ebx  je if2Equal1True  mov eax, 0  push eax  jmp if2Equal1End  if2Equal1True:  mov eax, 1  push eax  if2Equal1End:  push \_fibonaccin  push L5  pop eax  pop ebx  cmp eax, ebx  je if2Equal2True  mov eax, 0  push eax  jmp if2Equal2End  if2Equal2True:  mov eax, 1  push eax  if2Equal2End:  pop eax  pop ebx  cmp eax, 0  jne if2Or1True  cmp ebx, 0  jne if2Or1True  mov eax, 0  push eax  jmp if2Or1End  if2Or1True:  mov eax, 1  push eax  if2Or1End:  pop eax  cmp eax, 0  jne if2true  je if2End  if2true:  push L4  pop eax  jmp return\_fibonacci  jmp if2End  if2End:  push \_fibonaccin  push L4  pop ebx  pop eax  sub eax, ebx  push eax  call \_fibonacci  push eax  push \_fibonaccin  push L5  pop ebx  pop eax  sub eax, ebx  push eax  call \_fibonacci  push eax  pop eax  pop ebx  add eax, ebx  jo EXIT\_OVERFLOW  push eax  pop eax  jmp return\_fibonacci  return\_fibonacci:  pop ecx  pop edx  pop ebx  ret  SOMETHINGWRONG:  push offset null\_division  call outstrline  call system\_pause  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstrline  call system\_pause  push -2  call ExitProcess  \_fibonacci ENDP  main PROC  call \_Hello  push eax  pop mainstr  push offset L6  call outstr  push mainstr  call outstrline  call \_Print  push eax  push offset L7  pop mainstr  push mainstr  call outstrline  push offset L8  pop mainsymbol  push offset L9  call outstr  push mainsymbol  call outstrline  push L10  pop maina  push offset L11  call outstr  push maina  call outnumline  push L12  pop mainb  push offset L13  call outstr  push mainb  call outnumline  xor eax, eax  mov al, L14  push eax  pop mainBoolean  push offset L15  call outstr  push mainBoolean  call outboolline  push offset L16  call outstrline  push maina  push mainb  pop ebx  pop eax  and eax, ebx  push eax  pop mainAND  push L17  push L18  pop ebx  pop eax  or eax, ebx  push eax  pop mainOR  push L19  push L20  pop ebx  pop eax  xor eax, ebx  push eax  pop mainXOR  push maina  pop eax  not eax  push eax  pop mainNOT  push maina  call outnum  push offset L21  call outstr  push mainb  call outnum  push offset L22  call outstr  push mainAND  call outnumline  push offset L23  call outstr  push mainOR  call outnumline  push offset L24  call outstr  push mainXOR  call outnumline  push offset L25  call outstr  push maina  call outnum  push offset L22  call outstr  push mainNOT  call outnumline  push offset L16  call outstrline  push L26  call squere  push eax  pop mainb  push offset L27  call outstr  push L5  push L5  call power  push eax  call outnum  push offset L28  call outstr  push mainb  call outnumline  push offset L29  call outstr  push L26  call \_fibonacci  push eax  call outnum  push offset L16  call outstrline  push L3  push L10  pop eax  pop ebx  add eax, ebx  jo EXIT\_OVERFLOW  push eax  pop maina  push L30  pop mainb  while1do:  push maina  push mainb  pop eax  pop ebx  cmp eax, ebx  jne while1NotEqual1True  mov eax, 0  push eax  jmp while1NotEqual1End  while1NotEqual1True:  mov eax, 1  push eax  while1NotEqual1End:  pop eax  cmp eax, 0  je while1End  push offset L31  call outstr  push mainb  call outnum  push offset L32  call outstrline  push mainb  push L4  pop ebx  pop eax  sub eax, ebx  push eax  pop mainb  jmp while1do  while1End:  push offset L33  call outstrline  call system\_pause  push 0  call ExitProcess  SOMETHINGWRONG:  push offset null\_division  call outstrline  call system\_pause  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstrline  call system\_pause  push -2  call ExitProcess  main ENDP  end main |