МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора LKE-2020»

Выполнил студент Лысков Кирилл Евгеньевич

(Ф.И.О.)

Руководитель проекта ст. пр. Наркевич Аделина Сергеевна

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

Заведующий кафедрой к.т.н., доц. Пацей Н.В.

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

Консультанты ст. пр. Наркевич Аделина Сергеевна

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

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

Нормоконтролер ст. пр. Наркевич Аделина Сергеевна

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

Курсовой проект защищен с оценкой

Минск 2020

Содержание

[Глава 1. Спецификация языка программирования 6](#_Toc501592483)

[1.1 Характеристика языка программирования 6](#_Toc501592484)

[1.2 Алфавит языка 6](#_Toc501592485)

[1.3 Применяемые сепараторы 6](#_Toc501592486)

[1.4 Применяемые кодировки 7](#_Toc501592487)

[1.5 Типы данных 7](#_Toc501592488)

[1.6 Преобразование типов данных 7](#_Toc501592489)

[1.7 Идентификаторы 7](#_Toc501592490)

[1.8 Литералы 8](#_Toc501592491)

[1.9 Объявления данных и область видимости 8](#_Toc501592492)

[1.10 Инициализация данных 8](#_Toc501592493)

[1.11 Инструкции языка 8](#_Toc501592494)

[1.12 Операции языка 9](#_Toc501592495)

[1.13 Выражения и их вычисления 10](#_Toc501592496)

[1.14 Программные конструкции языка 10](#_Toc501592497)

[1.15 Область видимости идентификаторов 10](#_Toc501592498)

[1.16 Семантические проверки 10](#_Toc501592499)

[1.17 Распределение оперативной памяти на этапе выполнения 11](#_Toc501592500)

[1.18 Стандартная библиотека и её состав 11](#_Toc501592501)

[1.19 Ввод и вывод данных 11](#_Toc501592502)

[1.20 Точка входа 12](#_Toc501592503)

[1.21 Препроцессор 12](#_Toc501592504)

[1.22 Соглашения о вызовах 12](#_Toc501592505)

[1.23 Объектный код 12](#_Toc501592506)

[1.24 Классификация сообщений транслятора 12](#_Toc501592507)

[1.25 Контрольный пример 12](#_Toc501592508)

[Глава 2. Структура транслятора 13](#_Toc501592509)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 13](#_Toc501592510)

[2.2 Перечень входных параметров транслятора 14](#_Toc501592511)

[2.3 Перечень протоколов, формируемых транслятором и их содержимое 14](#_Toc501592512)

[Глава 3. Разработка лексического анализатора 15](#_Toc501592513)

[3.1 Структура лексического анализатора 15](#_Toc501592514)

[3.2 Контроль входных символов 15](#_Toc501592515)

[3.3 Удаление избыточных символов 16](#_Toc501592516)

[3.4 Перечень ключевых слов, сепараторов, символов операций соответствующих им лексем 16](#_Toc501592517)

[3.6 Принцип обработки ошибок 18](#_Toc501592518)

[3.7 Структура и перечень сообщений лексического анализатора 18](#_Toc501592519)

[Глава 4. Разработка синтаксического анализатора 19](#_Toc501592520)

[4.1 Структура синтаксического анализатора. 19](#_Toc501592521)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 19](#_Toc501592522)

[4.3 Построение конченого магазинного автомата 20](#_Toc501592523)

[4.4 Основные структуры данных 21](#_Toc501592524)

[4.5 Описание алгоритма синтаксического разбора 21](#_Toc501592525)

[4.6 Структура и перечень сообщений синтаксического анализатора 21](#_Toc501592526)

[4.7 Параметры синтаксического анализатора и режимы его работы 22](#_Toc501592527)

[4.8 Принцип обработки ошибок 22](#_Toc501592528)

[4.9 Контрольный пример 22](#_Toc501592529)

[Глава 5. Разработка семантического анализатора 23](#_Toc501592530)

[5.1 Структура семантического анализатора 23](#_Toc501592532)

[5.2 Функции семантического анализатора 23](#_Toc501592533)

[5.3 Структура и перечень сообщений семантического анализатора 23](#_Toc501592534)

[5.4 Принцип обработки ошибок 24](#_Toc501592535)

[5.5 Контрольный пример 24](#_Toc501592536)

[Глава 6. Преобразование выражений 25](#_Toc501592537)

[6.1 Выражения, допускаемые языком 25](#_Toc501592538)

[6.2 Польская запись и принцип ее построения 25](#_Toc501592539)

[6.3 Программная реализация обработки выражений 26](#_Toc501592540)

[6.4 Контрольный пример 26](#_Toc501592541)

[Глава 7. Генерация кода 27](#_Toc501592542)

[7.1 Структура генератора кода 27](#_Toc501592543)

[7.2 Представление типов данных в памяти 27](#_Toc501592544)

[7.3 Статическая библиотека 28](#_Toc501592545)

[7.4 Особенности алгоритма генерации кода 28](#_Toc501592546)

[7.5 Контрольный пример 29](#_Toc501592547)

[Глава 8. Тестирование транслятора 30](#_Toc501592548)

[Заключение 34](#_Toc501592549)

[Приложение А 35](#_Toc501592550)

[Приложение Б 36](#_Toc501592551)

[Приложение Г 38](#_Toc501592552)

[Приложение Д 40](#_Toc501592553)

[Приложение Е 41](#_Toc501592554)

[Литература 44](#_Toc501592555)

**Введение**

Транслятор – это комплекс отдельных программ, позволяющих преобразовывать исходный код на одном языке программирования в исходный код на другом языке программирования.

Классический транслятор состоит из следующих частей:

* лексический анализатор;
* синтаксический анализатор;
* семантический анализатор;
* генератор кода, или интерпретатор.

Все части транслятора, взаимодействуя между собой, обрабатывают входной текст и строят для него эквивалентный текст на понятном компьютеру языке программирования.

# Глава 1. Спецификация языка программирования

* 1. **Характеристика языка программирования**

Язык LKE-2020 – это строго типизированный, процедурный, компилируемый язык. Он является транслируемым, не объектно-ориентированным. В LKE-2020 используется четыре типа данных: беззнаковый целочисленный тип данных unit (размер 4 байта), строковый тип данных note, логический тип данных idle, тип данных целочисленный массив []unit.

* 1. **Алфавит языка**

В основе алфавита LKE-2020 лежит таблица символов ACSII. Исходный код LKE-2020 может содержать символы латинского алфавита верхнего и малого регистров, цифры десятичной системы счисления от 0 до 9, символы кириллицы верхнего и малого регистров, знаковые символы, представленные в таблице 1.1.

Таблица 1.1 – Алфавит языка

|  |  |
| --- | --- |
| Название подгруппы | Символы подгруппы |
| Символы латинского алфавита | [a-z] && [A-Z] |
| Символы русского алфавита | [а-я] && [А-Я] |
| Знаковые символы и числовые символы | [! - #] && [% - ?] && символ "[" && символ "]" && символ  " " (пробел)  "\n" (символ новой строки) |

* 1. **Применяемые сепараторы**

Символы-сепараторы – символы, используемые для разделения отдельных лексических единиц или функциональных элементов в исходном коде программы. Символы, которые являются сепараторами представлены в таблице 1.2.

Таблица 1.2 - Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение сепаратора |
| " " (пробел) | Символы-сепараторы для разделения ключевых слов и идентификаторов |
| "\n" (символ новой строки) | Символы-сепараторы для разделения инструкций языка |
| : | Символы-сепараторы указывающие на открытие программного блока функции / цикла |
| ; | Символы-сепараторы указывающие на закрытие блока цикла / условного блока |
| . | Символы-сепараторы указывающие на закрытие блока функции / объявление глобальной переменной |
| , | Символы-сепараторы, указывающие на разделение параметров функции / неинициализированную переменную / продолжение условного блока |
| => | Символы-сепараторы указывающие на открытие условного блока |
| << | Символы-сепараторы указывающие на вывод значения выражения в консоль |
| >> | Символы-сепараторы указывающие на возврат значения выражения из функции |
| <  > | Символы-сепараторы, указывающие на параметры функции |
| (  ) | Символы-сепараторы, указывающие на аргументы при вызове функции / приоритетность операций (в выражениях) |
| [  ] | Символы-сепараторы, указывающие на индексацию массива (в выражениях) |

* 1. **Применяемые кодировки**

При написании исходного кода на языке программирования используется кодировка Windows-1251, представленная на рисунке 1.1.



Рис 1.1 – Используемая кодировка Windows-1251

* 1. **Типы данных**

Пользовательские типы данных не поддерживаются. Допускается использование фундаментальных типов данных определенных в таблице 1.3.

Таблица 1.3 - Фундаментальные типы данных

|  |  |
| --- | --- |
| Тип данных | Описание |
| unit | Является беззнаковым целочисленным типом данных. Этот тип данных занимает 4 байта. Инициализация по умолчанию: 0. Предназначен для арифметических операций над числами и операций сравнения чисел.  Предусмотрены следующие арифметические операции:  + - бинарная операция суммирования (unit + unit);  - - бинарная операция вычитания (unit - unit);  \* - бинарная операция умножения (unit \* unit);  / - бинарная операция деления (unit / unit);  % - бинарный операция остатка от деления двух операндов (unit % unit);  Предусмотрены следующие операции сравнения:  ? - бинарный оператор сравнения двух операндов на равенство (unit ? unit);  < - бинарный оператор сравнения меньше ли первый операнд чем второй (unit < unit);  > - бинарный оператор сравнения больше ли первый операнд чем второй (unit > unit);  ! - бинарный оператор сравнения двух операндов на неравенство (unit ! unit). |
| idle | Является логическим типом данных. Этот тип данных занимает 1 байт. Инициализация по умолчанию: 0. Предназначен для работы в условном блоке.  В языке предусмотрено преобразование логического типа в целочисленный, и, соответственно, все операции над целочисленным типом применимы и к логическому. |
| note | Является строковым типом данных. Предназначен для работы с символами, каждый символ занимает 1 байт.  Операции над данными строкового типа: возможно присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции. |
| []unit | Является сложным типом данных. Предназначен для хранения массива элементов целочисленного типа, каждый элемент занимает 4 байта.  Предусмотрена следующая операция:  [ ] - унарная операция индексирования, возвращает элемент из массива по его индексу и преобразует его в целочисленный тип (unit[ unit ]); |

* 1. **Преобразование типов данных**

В языке LKE-2020 поддерживается взаимное преобразование численных типов данных. Данные логического типа idle могут быть преобразованы в беззнаковый целочисленный тип unit, и обратно. Также используется преобразование элемента массива по его индексу оператором [ ] в целочисленный и логический типы. Преобразование строкового типа данных не поддерживается.

* 1. **Идентификаторы**

Для именования функций, параметров и переменных используются идентификаторы. Не предусмотрены зарезервированные идентификаторы. Максимальная длина идентификатора равна 15 и не должна превышать это значение. При превышении максимального значения длина идентификатора усекается до 15. Для составления имени идентификатора используется следующее регулярное выражение:

* [a..Z] ([a..Z] | [0-9])\*
  1. **Литералы**

Предусмотрены числовые (unit), логические (idle), строковые (note) литералы, литералы массива ([]unit) и строковые литералы для имени файла. Правила записи литералов определены в таблице 1.4.

Таблица 1.4 - Правила записи литералов

|  |  |
| --- | --- |
| Тип литерала | Описание литерала |
| Числовые | Максимально допустимое значение равно . Могут иметь десятичное представление ([0..9])+, восьмеричное 0o([0..7])+ или шестнадцатеричное 0x([0..9]|[a..f])+. При выходе за пределы допустимости будет выведена ошибка. Значение числового литерала может быть присвоено переменной целочисленного, либо логического типа. |
| Логические | Представлены в виде ключевых слов true и false. Значения которых 1 и 0 соответственно, могут быть присвоены переменной целочисленного, либо логического типа. |
| Строковые | Состоит из символов латинского алфавита, кириллицы, различных знаков языка, пробелов и специальных символов "/n" (символ новой строки), заключенных в "…" (двойные кавычки). Максимальное число которых не может превышать . В случае превышения длины литерала работа транслятора прекращается. |
| Массив | Состоит из числовых литералов, разделенных запятыми и заключенных в {…}. Максимально допустимое значение каждого числового литерала равно . |
| Имя файла | Состоит из символов латинского алфавита и символа ".", различных знаков, заключенных в '…' (одинарные кавычки). Максимальное число которых не может превышать . |

* 1. **Объявления данных и область видимости**

В языке программирования LKE-2020 переменная должны быть объявлена до ее использования. Переменная может быть объявлена в глобальной области видимости с ее обязательной инициализацией значением литерала. Также областью видимости может являться блок функции или блок цикла, в которой она была объявлена.

* 1. **Инициализация данных**

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. Объектами-инициализаторами могут быть идентификаторы, литералы, выражения, элементы массива и вызовы функций. По умолчанию переменные целочисленного и логического типов инициализируются нулем, строковые переменные инициализируются пустой строкой. Переменные типа массив, обязательно должны быть инициализированы при объявлении с помощью соответствующего литерала, стоящего справа от идентификатора.

* 1. **Инструкции языка**

В языке программирования LKE-2020 применяются инструкции, представленные в таблице 1.5.

Таблица 1.5 - Инструкции языка

|  |  |
| --- | --- |
| Инструкция языка | Синтаксис |
| Главная функция | main  :  <инструкции языка>  end  . |
| Вызов функций | [..] <идентификатор функции> (<идентификатор | литерал | идентификатор>,…) |
| Возврат из функции | release >> <идентификатор | литерал | выражение> |
| Объявление функции | order < <тип данных> <идентификатор>, … > <тип данных> <идентификатор функции>  :  <инструкции языка>  . |
| Объявление переменной | & <тип данных> <идентификатор> <= литерал | , | . > |
| Объявление массива | [ <литерал> ] unit <идентификатор> <литерал массива> |
| Присваивание | для unit, note, idle:  <идентификатор> = <литерал | выражение | идентификатор 2>; |
| Вывод данных | write << <идентификатор | литерал | выражение> |
| Цикл | since <идентификатор> = <литерал> till <идентификатор | литерал | выражение>  :  <тело цикла>  ; |
| Условный блок | check <идентификатор | литерал | выражение> => <инструкция языка> [, <check … | get> => инструкция языка ;] |
| Комментарий | # [текст] |

* 1. **Операции языка**

В языке программирования LKE-2020 определены арифметические операции, операции сравнения и операция индексирования. Наибольшую приоритетность арифметических операций имеют операции умножения, деления и остатка от деления, а сложение и вычитание меньшую. При одинаковом приоритете первой выполнится операция, расположенная левее. Изменить приоритетность арифметических операций можно с помощью круглых скобок. Операции сравнения применяются в условном блоке и имеют наименьшую приоритетность по сравнению с арифметическими. Применимые к числовым типам данных операции языка программирования LKE-2020 приведены в таблице 1.6.

Таблица 1.6 - Операции языка

|  |  |  |
| --- | --- | --- |
| Тип | Тип операндов | Операторы |
| Арифметические | Целочисленный Логический | () – приоритетность операций  + ̶ сложение  -  ̶ вычитание  \* ̶ умножение  / ̶ деление  % ̶ остаток от деления |
| Сравнения | Целочисленный Логический | ? – равенство  < – меньше  > – больше  ! – неравенство |
| Индексирования | Массив | [ ] – элемент массива по его индексу |

* 1. **Выражения и их вычисления**

Предусмотрены следующие правила составления выражений:

* + выражения читаются слева направо и записываются в одну строку;
  + реализация выражений происходит с помощью обратной польской записи;
  + для изменения приоритета арифметических операций используются круглые скобки;
  + составляются из идентификаторов, литералов, элементов массива, вызовов функций и других выражений числовых типов данных.
  1. **Программные конструкции языка**

Программные конструкции представлены в таблице 1.7.

Таблица 1.7 - Программные конструкции языка

|  |  |
| --- | --- |
| Конструкция | Представление в языке |
| Главная функция | main  :  <инструкции языка>  end  . |
| Функция | order < <тип данных> <идентификатор>, … > <тип данных> <идентификатор функции>  :  <инструкции языка>  . |
| Цикл | since <идентификатор> = <литерал> till <идентификатор | литерал | выражение>  :  <тело цикла>  ; |
| Условие | check <идентификатор | литерал | выражение> => <инструкция языка> [, <check … | get> => инструкция языка ;] |

* 1. **Область видимости идентификаторов**

Для использования идентификаторов внутри программных структур, переменные должны быть объявлены на внешнем или на том же уровне области видимости до их использования. Параметры видны только внутри функции, в которой объявлены.

* 1. **Семантические проверки**

Перечень семантических проверок, предусмотренных языком, приведен в таблице 1.8.

Таблица 1.8 - Перечень семантических проверок

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Наличие только одной функции main, как точки входа в программу |
| 2 | Наличие ключевого слова main |
| 3 | Использование директив с помощью символа «!» |
| 4 | Одноразовое объявление внешней функции с помощью директивы |
| 5 | Объявление функций только в глобальной области видимости |
| 6 | Использование ключевого слова release только внутри соответствующей функции |
| 7 | Отсутствие вложенных циклов в другие циклы |
| 8 | Одноразовое объявление переменной внутри области ее видимости |
| 9 | Одноразовое объявление функции |
| 10 | Использование оператора «[ ]» только для объявления или индексации массива. |
| 11 | Использование только числового значения в качестве индекса или размера массива |
| 12 | Использование директивы подключения библиотеки только с соответствующим литералом имени файла |
| 13 | Использование литерала имени файла только с соответствующей директивой |

* 1. **Распределение оперативной памяти на этапе выполнения**

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

* 1. **Стандартная библиотека и её состав**

В языке LKE-2020 присутствует стандартная библиотека, которая может быть подключена на этапе трансляции исходного кода в язык ассемблера с помощью обязательного указания директивы ! Adhere. Для использования внешних библиотечных функций в программе на языке LKE-2020, они должны быть объявлены с помощью директивы ! Form перед их вызовом. Содержимое библиотеки и описание функций представлено в таблице 1.9.

Таблица 1.9 - Стандартная библиотека языка LKE-2020

|  |  |
| --- | --- |
| Функция | Описание |
| Form <note a, note b> idle comprnotes | Целочисленная функция. Вычисляет и возвращает квадратный корень из числа a. |
| Form <note a > unit notelength | Целочисленная функция. Вычисляет и возвращает длину строки в целочисленном виде. |
| Form <note a, note b> idle copynote | Строковая функция. Возвращает результат копирования строки b, записанный в строку a. |
| Form <note a, note b> note cncatnotes | Строковая функция. Возвращает результат конкатенации строк a и b, записанный в строку a. |
| Form <unit a, unit b> unit powerunit | Целочисленная функция. Вычисляет и возвращает число a, возведенное в степень b. |

* 1. **Ввод и вывод данных**

Ввод данных не предусмотрен. Вывод данных осуществляется с помощью ключевого слова write. В качестве аргумента принимаются числовые и строковые идентификаторы и литералы, так же выражения и вызовы функции:

write << <идентификатор | литерал | выражение>;

* 1. **Точка входа**

В языке LKE-2020 точкой входа является ключевое слово “main”. Точка входа не может отсутствовать, также не может быть переопределена. В программе может быть только одна точка входа.

* 1. **Препроцессор**

В языке LKE-2020 предусмотрено использование директивы ! Adhere для подключения файла статической библиотеки и директивы ! Form для объявления внешней функции, описанной в статической библиотеке.

! Adhere <имя файла>

! Form < < тип данных> <идентификатор>, … > <тип данных> <идентификатор>

* 1. **Соглашения о вызовах**

При генерации кода используется соглашение stdcall, в котором все параметры передаются в стек справа налево. Освобождением памяти занимается вызываемая подпрограмма, которая очищает стек.

* 1. **Объектный код**

Целевым языком трансляции исходного кода, написанном на языке LKE-2020, является язык ассемблера.

* 1. **Классификация сообщений транслятора**

Транслятор генерирует сообщения о ошибках пользователю. В соответствии с префиксами будут различаться сообщения, представленные в таблице 1.10.

Таблица 1.10 - Классификация сообщений транслятора

|  |  |
| --- | --- |
| Префикс ошибки | Описание ошибки |
| |Lexical| ### | ### - код ошибки. Сообщение, генерируемое на этапе лексического анализа. |
| |Syntax| ### | ### - код ошибки. Сообщение, генерируемое на этапе синтаксического анализа. |
| |Semantic| ### | ### - код ошибки. Сообщение, генерируемое на этапе семантического анализа. |
| |SYSTEM| ### | ### - код ошибки. Сообщение, генерируемое при системной ошибке. |

* 1. **Контрольный пример**

Контрольный пример, написанный на языке LKE-2020, представлен в приложении А.

**Глава 2. Структура транслятора**

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

Исходный код, написанный на языке программирования LKE-2020, является для транслятора входными данными.

Как выходные данные используется объектный код и протоколы работы транслятора, описанные в пункте 2.3.

Компоненты транслятора приведены на рисунке 2.1.



Рис 2.1 - Структура транслятора LKE-2020

Первоначально на вход лексического анализатора передается исходный код. Анализатором проверяется исходный текст на недопустимые символы, выделяет литералы, идентификаторы и ключевые слова, а также формирует таблицы лексем и идентификаторов.

Далее наступает черед синтаксического анализатора, к нему на вход поступает таблица лексем, полученная на этапе лексического анализа. Если программа построена синтаксически правильно, то осуществляется переход к этапу трансляции стоящему далее, при ином раскладе работа транслятора останавливается.

Наборы функций, проверяющие правила на разных этапах работы транслятора представлены в семантическом анализаторе. Продолжение или остановка работы транслятора всецело зависит от критичности возникающих ошибок.

Генерация кода реализуется посредством чистой интерпретации, без создания промежуточного представления кода. В финале происходит генерация кода, во время исполнения которого формируется объектный код.

**2.2 Перечень входных параметров транслятора**

В таблице 2.1 представлены входные параметры, которые могут использоваться для представления работы транслятора.

Таблица 2.1 - Входные параметры транслятора

|  |  |  |
| --- | --- | --- |
| Параметр | Назначение | Тип |
| -in: | Указывает на файл с исходным кодом. Исходный код содержится в файле с расширением \*.txt | Обязательный |
| -out: | Указывает имя протокола. Если не указан явно, то имя протокола не формируется. | Не обязательный |
| -log: | Указывает имя протокола. Если не указан явно, то имя протокола формируется конкатенацией имени файла исходного кода и постфикса “.log” | Не обязательный |

**2.3 Перечень протоколов, формируемых транслятором и их содержимое**

По итогам своей работы транслятор формирует два протокола, согласно заданным входным параметрам. -log: <путь к файлу> - в этом файле находится информация о входных параметрах, количестве символов исходного кода, а также код и сообщение вызванной ошибки, при остановке работы транслятора; -out: <путь к файлу> - в этом файле находятся информация о таблице лексем, идентификаторов, работы синтаксического анализатора, дерево разбора.

# Глава 3. Разработка лексического анализатора

**3.1 Структура лексического анализатора**

Структура лексического анализатора представлена на рисунке 3.1.



Рис 3.1 - Структура лексического анализатора

Исходный код на языке LKE-2020 является входными данными;

Таблицы лексем и идентификаторов являются выходными данными;

**3.2 Контроль входных символов**

Таблица допустимости представлена на рисунке 3.2.

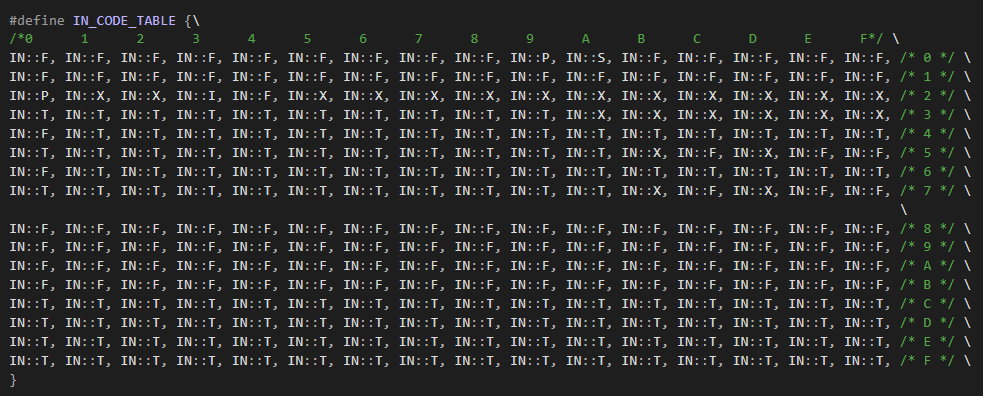


Рисунок 3.2 - Таблица допустимости входных символов

Таблица допустимости была сформирована на основе кодировки Windows-1251. Таблица необходима для проверки входных символов на допустимость. Символы могут быть разрешенными, запрещенными, игнорируемыми и др. Символы представлены в шестнадцатеричной системе счисления. В таблице записаны различные числовые значения, соответствующие символам в данной таблице:

«T» - разрешенные алфавитом символы;

«F» - запрещенные алфавитом символы;

«I» - символ комментария;

«S» - символ новой строки;

«P» - символы пробела и табуляции;

«X» - символы знаков;

**3.3 Удаление избыточных символов**

Избыточными символами представлены пробелы, символы табуляции, символы перехода на новую строку.

Алгоритм удаления избыточных символов:

1. Просматриваем текущий символ

1.1) Если «I», символ комментария, то просматриваем следующий символ.

1.1.1) Если «S», символ новой строки, то переходим к пункту 2.

1.2) Если «T», разрешенные символы, то записываем символ в результирующую строку.

1.3) Если «X», символы знаков, то просматриваем последний записанный в результирующую строку символ.

1.3.1) Если «P», то записываем символ «X» в результирующую строку вместо предыдущего.

1.3.2) Если не «P», то записываем символ «X» в результирующую строку.

1.4) Если «P», символ пробел и табуляции, то просматриваем последний записанный в результирующую строку символ.

1.4.1) Если «T», то записываем символ «P» в результирующую строку.

1.4.2) Если не «T», то переходим к пункту 2.

1.5) Если «S», символ новой строки, то просматриваем последний записанный в результирующую строку символ.

1.5.1) Если «P», то записываем символ «S» в результирующую строку вместо предыдущего и увеличиваем на единицу переменную, отвечающую за подсчет строк

1.5.2) Если «T» или «X», то увеличиваем на единицу переменную, отвечающую за подсчет строк и записываем символ «S» в результирующую строку.

1.5.3) Если «S», то переходим к пункту 2.

1.6) Если обнаружен конец файла, то переходим к пункту 3.

1. Перемещаем указатель на байт вправо и переходим к пункту 1.
2. Окончание алгоритма.

## 3.4 Перечень ключевых слов, сепараторов, символов операций соответствующих им лексем

Перечень ключевых слов, сепараторов, символов операций соответствующих им лексем представлен в таблице 3.1.

Таблица 3.1 - Перечень ключевых слов

|  |  |
| --- | --- |
| Цепочка | Лексема |
| unit | t |
| note | t |
| idle | t |
| order | \* |
| release | r |
| main | m |
| end | e |
| write | w |
| since | s |
| till | u |
| skip | b |
| check | c |
| get | g |
| Adhere | a |
| Form | f |
| true | l |
| false | l |
| + | v |
| - | v |
| \* | v |
| / | v |
| % | v |
| = | = |
| ( | ( |
| ) | ) |
| [ | [ |
| ] | ] |
| < | < |
| > | > |
| & | & |
| , | , |
| : | : |
| ; | ; |
| . | . |
| ! | ! |
| ? | ? |
| идентификатор | i |
| числовой литерал | l |
| строковый литерал | l |
| литерал массива | l |

**3.5 Основные структуры данных**

В приложении Б представлены основные структуры данных на этапе синтаксического анализа.

**3.6 Принцип обработки ошибок**

Все лексические ошибки являются критическими, так как они не позволят анализаторам или генератору правильно функционировать. В случае обнаружения такой ошибки транслятор прекращает свою работу, в консоль и в log-файл записывается ее код и сообщение. Подсчет количества ошибок в обоих случаях не ведется.

## 3.7 Структура и перечень сообщений лексического анализатора

Префикс сообщений “|Lexical|”. Перечень сообщений, генерируемых на этапе лексического анализа, представлен в таблице 3.2.

Таблица 3.2 - Сообщения лексического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 100 | Недопустимый символ в исходном коде |
| 101 | Недопустимый символ в строковом литерале |
| 110 | Лексема не найдена |
| 111 | Превышена длина строкового литерала |
| 123 | Переменная не объявлена |
| 124 | Неверно задан тип данных |
| 125 | Неверно объявлена функция |
| 130 | Превышено допустимое целочисленное значение |
| 131 | Слишком много значений инициализатора |
| 132 | Неверная инициализация массива |

**3.8 Параметры лексического анализатора и режим его работы**

Текст кода на языке LKE-2020 подается на вход. Параметры не определяют режим работы лексического анализатора.

**3.9 Контрольный пример**

На вход лексического анализатора подается программа на языке LKE-2020, описанная в пункте 1.25. Результат работы лексического анализатора представлен в приложении В.

# Глава 4. Разработка синтаксического анализатора

## 4.1 Структура синтаксического анализатора.

Структура синтаксического анализатора представлена на рисунке 4.1.



Рисунок 4.1 - Структура синтаксического анализатора

Таблицы лексем и идентификаторов являются входными данными.

Дерево разбора является выходными данными.

## 4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

Грамматика, описывающая язык LKE-2020 представлена в таблице 4.1.

Таблица 4.1 - грамматика языка LKE-2020

|  |  |
| --- | --- |
| Нетерминалы | Описание |
|  | Правила, описывающие общую структуру программы |
|  | Порождает правила, описывающие инструкции языка |
|  | Порождает правила, описывающие структуру условного блока |
|  | Порождает правила, описывающие условные инструкции |
|  | Порождает правила, описывающие выражения |
|  | Порождает правила, описывающие арифметические операции |
|  | Порождает правила, описывающие операции сравнения |
|  | Порождает правила, описывающие формальные параметры функции |
|  | Порождает правила, описывающие фактические параметры функции |

## 4.3 Построение конченого магазинного автомата

Принцип действия конечного магазинного автомата представлен на рисунке 4.2.



Рис. 4.2 - МП-автомат

Формальное описание МП-автомата:



 - множество состояний;

 - алфавит входных символов;

 - специальный алфавит магазинных символов;

-функция переходов автомата , где  - множество подмножеств ;

 - начальное состояние автомата;

- начальное состояние магазина (маркер дна);

- множество конечных состояний.

Конфигурация (текущее состояние автомата) описывается тройкой , где - текущее состояние автомата, - остаток цепочки,  - цепочка-содержимое магазина.

Начальное состояние , - начальное состояние автомата,  - входная цепочка, - маркер дна магазина.

Цепочка  является допустимой (распознается) автоматом , если  и .

Работа автомата 

1. состояние автомата 
2. читает символ  находящийся под головкой (сдвигает ленту);
3. не читает ничего (читает , не сдвигает ленту);
4. из  определяет новое состояние , если  или .
5. читает верхний (в стеке) символ  и записывает цепочку  т.к. , при этом, если , то верхний символ магазина просто удаляется.

работа автомата заканчивается 

## 4.4 Основные структуры данных

В приложении Г представлены основные структуры данных и правила перехода, используемые на этапе синтаксического анализа.

## 4.5 Описание алгоритма синтаксического разбора

Входные символы и лексемы в форме Грейбах находятся в ленте на входе конечного автомата.

1) Если лента не пустая, переходим далее следующему пункту, иначе переходим к пункту 5.

2) Если на верхушке магазина нетерминальный символ.

2.1) Если есть такое правило, то переходим к следующему пункту.

2.1.1) Если цепочка есть, возвращаем NS\_OK. Переходим к пункту 4.

2.1.2) Иначе восстанавливаем состояние. Переходим к пункту 4.

2.2) Иначе возвращаем ошибку. Переход к пункту 4.

3) Если на верхушке терминал и он совпадает с символом на ленте, то удаляем его из стека и продвигаем ленту. Переход к пункту 4.

4) Повторяем шаг, переходим к пункту 1.

5) Конец работы.

## 4.6 Структура и перечень сообщений синтаксического анализатора

Префикс сообщений “|Syntax|”. Перечень сообщений, генерируемых на этапе синтаксического анализа, представлен в таблице 4.3.

Таблица 4.3 - Перечень сообщений

|  |  |
| --- | --- |
| Код | Сообщение |
| 200 | Неверная структура программы |
| 201 | Ошибка в инструкции языка |
| 202 | Ошибка в условном блоке |
| 203 | Ошибка в условной инструкции |
| 204 | Ошибка в операндах |
| 205 | Ошибка в арифметической операции |
| 206 | Ошибка в операции сравнения |
| 207 | Ошибка в параметрах функции |
| 208 | Ошибка в параметрах вызываемой функции |

## 4.7 Параметры синтаксического анализатора и режимы его работы

Таблицы идентификаторов и лексем являются входными параметрами для синтаксического анализатора. Эти таблицы мы получаем в ходе лексического анализа. В конце после разбора формируется дерево разбора, которое выводится в протокол работы –out.

## 4.8 Принцип обработки ошибок

При обнаружении ошибки в цепочке какого-либо правила, синтаксический анализатор идет вверх по дереву разбора, пока не найдет верный вариант. Иначе запоминается самая глубокая ошибка, которая выводится в консоль и в протокол работы, транслятор прекращает свою работу.

## 4.9 Контрольный пример

Результатом работы синтаксического анализатора является дерево разбора. Результат работы синтаксического анализатора представлен в приложении Д.

# Глава 5. Разработка семантического анализатора

1. 1. **Структура семантического анализатора**

Семантический анализатор состоит из набора функций для проверки правильности исходной программы. Функции анализатора выполняются на различных этапах работы транслятора. Структура семантического анализатора представлена на рисунке 5.1.

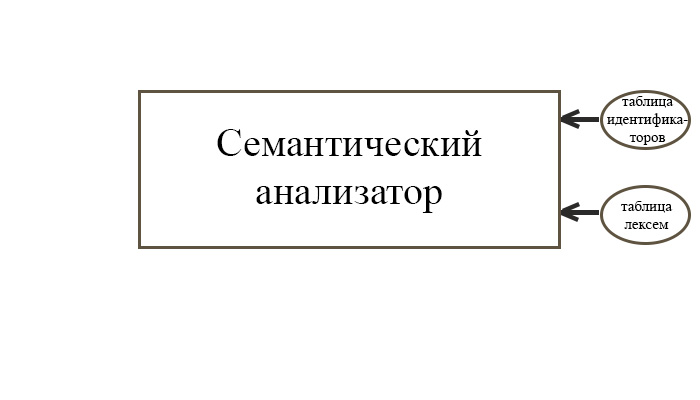


Рисунок 5.1 - Структура семантического анализатора

* 1. **Функции семантического анализатора**

Функции, представляющие проверку правил, представлены в таблице 5.1. Некоторые проверки встроены непосредственно в код этапов транслятора.

Таблица 5.1 - Семантические проверки

|  |  |
| --- | --- |
| Функция | Описание |
| TypeCheck | Проверка на соответствие типов идентификаторов и операндов в выражениях |
| ParamCheck | Проверка на соответствие типов и количества формальных и фактических параметров функций |
| FuncRet | Проверка на соответствие типа данных возвращаемого функцией значения |
| StdCheck | Проверка на подключение стандартной библиотеки |

* 1. **Структура и перечень сообщений семантического анализатора**

Префикс сообщений “|Semantic|”. Сообщения, генерируемые при выполнении семантических проверок, представлены в таблице 5.2.

Таблица 5.2 - Перечень сообщений

|  |  |
| --- | --- |
| Код | Сообщение |
| 112 | Дублирование функции main |
| 113 | Требуется объявление функции main |
| 114 | Неверно задана директива |
| 115 | Повторное использование директивы |
| 120 | Объявление вложенной функции |
| 121 | Требуется объявление функции |
| 122 | Использование вложенного цикла недопустимо |
| 126 | Повторное объявление переменной |
| 127 | Повторное объявление функции |
| 128 | Некорректно задан размер массива |
| 129 | Требуется объявление массива |
| 133 | Размер массива должен иметь целочисленный тип |
| 134 | Требуется имя файла |
| 135 | Неверное использование имени файла |
| 300 | Несовпадение типов операндов |
| 301 | Присваивание недопустимого типа |
| 302 | Несовпадение типов параметров функции |
| 303 | Несовпадение количества параметров функции |
| 304 | Тип возвращаемого значения не соответствует типу функции |
| 305 | Не удается открыть файл |
| 306 | Требуется подключение стандартной библиотеки |

* 1. **Принцип обработки ошибок**

В случае обнаружения критической семантической ошибки транслятор прекращает свою работу, в консоль и в протокол работы транслятора выводится соответствующее сообщение об ошибке, иначе происходит запись этой ошибки в протокол без прекращения работы транслятора.

* 1. **Контрольный пример**

Результат работы семантических функций описан в главе 8.

**Глава 6. Преобразование выражений**

**6.1 Выражения, допускаемые языком**

В языке LKE-2020 допускаются выражения с использованием числовых идентификаторов и литералов. Также предусмотрены следующие арифметические операции и операции сравнения:

* сложения: «+» ;
* вычитания: «-» ;
* умножения: «\*» ;
* деления: «/» ;
* остатка от деления: «%» ;
* равенство: «?» ;
* неравенство: «!» ;
* меньше: «<» ;
* больше: «>» ;

Также есть возможность изменять приоритет выполнения арифметических операций при помощи скобок:

* «(»;
* «)»;

За счет заключения арифметической операции в скобки происходит повышение ее приоритета, а значит при вычислении всего выражения операция с более высоким приоритетом будет вычисляться раньше операции с меньшим приоритетом.

Приоритетность операций представлена в таблице 6.1. Чем выше число, тем выше и приоритет.

Таблица 6.1 - Приоритетность операций

|  |  |
| --- | --- |
| Операция | Приоритет |
| «?», «!», «<», «>» | 1 |
| «(», «)» | 2 |
| «+», «-» | 3 |
| «\*», «/», «%» | 4 |

**6.2 Польская запись и принцип ее построения**

Польская запись -форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Приоритетность операций приведена в таблице 6.1. Известен следующий принцип 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**

После этапов лексического и синтаксического анализа происходит преобразование в польскую запись, перед этапом семантического анализа и этапом генерации кода на язык ассемблера.

**6.4 Контрольный пример**

Контрольный пример разбора выражения содержится в таблице 6.2.

Таблица 6.2 - Разбор выражения

|  |  |  |
| --- | --- | --- |
| Выражение | Стек | Результат |
| notelength ( start ) \* ( 33 % 10 + 156 / 27 ) |  |  |
| ( start ) \* ( 33 % 10 + 156 / 27 ) | notelength |  |
| start ) \* ( 33 % 10 + 156 / 27 ) | ( notelength |  |
| ) \* ( 33 % 10 + 156 / 27 ) | ( notelength | start |
| \* ( 33 % 10 + 156 / 27 ) | notelength | start |
| ( 33 % 10 + 156 / 27 ) | \* | start notelength @1 |
| 33 % 10 + 156 / 27 ) | ( \* | start notelength @1 |
| % 10 + 156 / 27 ) | ( \* | start notelength @1 33 |
| 10 + 156 / 27 ) | % ( \* | start notelength @1 33 |
| + 156 / 27 ) | % ( \* | start notelength @1 33 10 |
| 156 / 27 ) | + ( \* | start notelength @1 33 10 % |
| / 27 ) | + ( \* | start notelength @1 33 10 % 156 |
| 27 ) | / + ( \* | start notelength @1 33 10 % 156 |
| ) | / + ( \* | start notelength @1 33 10 % 156 27 |
|  | \* | start notelength @1 33 10 % 156 27 / + |
|  |  | start notelength @1 33 10 % 156 27 / + \* |

# Глава 7. Генерация кода

## 7.1 Структура генератора кода

Заключительным этапом трансляции языка LKE-2020 является генерация кода. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. Выходной файл на языке ассемблера, который будет являться результатом работы транслятора, в соответствие с таблицей лексем. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода LKE-2020 представлена на рисунке 7.1.



Рисунок 7.1 – Структура генератора кода

## 7.2 Представление типов данных в памяти

Элементы таблицы идентификаторов расположены сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке LKE-2020 и на языке ассемблера приведены в таблице 7.1.

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

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке LKE-2020 | Тип идентификатора на языке ассемблера | Пояснение |
| unit | DWORD | Хранит беззнаковый целочисленный тип данных. |
| note | BYTE | Хранит указатель на начало строки. Строка должна завешаться нулевым символом. |
| idle | BYTE | Хранит логический тип данных. |
| []unit | DWORD | Хранит указатель на первый элемент массива целочисленного типа данных. |

## 7.3 Статическая библиотека

В языке LKE-2020 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Она может быть подключена на этапе трансляции исходного кода в язык ассемблера с помощью директивы ! Adhere. Объявление функций статической библиотеки можно осуществить с помощью директивы ! Form.

Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю. Эти функции представлены в таблице 7.2.

Таблица 7.2 - Дополнительные функции стандартной библиотеки

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void writeunit(int u) | Функции для вывода в стандартный поток значения целочисленного идентификатора/литерала/выражения. |
| void writeidle(int i) | Функции для вывода в стандартный поток значения логического идентификатора/литерала. |
| void writenote(char\* b) | Функции для вывода в стандартный поток значения строкового идентификатора/литерала. |
| int comprnotes(char\* s1, char\* s2) | Логическая функция. Возвращает результат сравнения строки s1 со строкой s2. |
| int notelength (char\* s) | Целочисленная функция. Вычисляет и возвращает длину строки s. |
| int copynote(char\* s1, char\* s2) | Функция, копирующая значение строки s2 в строку s1. |
| char\* cncatnotes(char\* s1, char\* s2) | Строковая функция. Возвращает результат конкатенации строк a, b, записанный в строку a. |
| int powerunit(int u, int n) | Целочисленная функция. Вычисляет и возвращает число a, возведенное в степень b. |

## 7.4 Особенности алгоритма генерации кода

В языке LKE-2020 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2

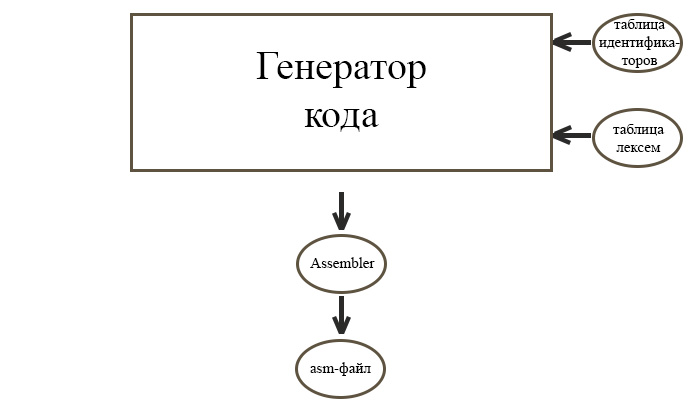


Рисунок 7.2 - Структура генератора кода

## 7.5 Контрольный пример

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

# Глава 8. Тестирование транслятора

В данной главе описаны возможные ошибки, возникающие на различных этапах работы транслятора. Результат тестирования представлен в таблице 8.1.

Таблица 8.1 - Тестирование транслятора

|  |  |
| --- | --- |
| Исходный код | Генерируемая ошибка |
| main :  &unit $\_var,  end. | Ошибка 100: |Lexical| Недопустимый символ в исходном коде,  Строка 2 |
| main :  &note str = "text  end. | Ошибка 101: |Lexical| Недопустимый символ в строковом литерале,  Строка 2 |
| main :  &unit переменная = 1  end. | Ошибка 110: |Lexical| Лексема не найдена,  Строка 2 |
| main :  &note string = "too big string…/n"  end. | Ошибка 111: |Lexical| Превышена длина строкового литерала,  Строка 2 |
| main :  &unit x = 0  end.  main :  write << 1  end. | Ошибка 112: |Semantic| Дублирование функции main,  Строка 4 |
| order <> unit function :  release >> 24. | Ошибка 113: |Semantic| Требуется объявление функции main,  Строка 1 |
| Adhere 'std.lib'  main :  write << "text"  end. | Ошибка 114: |Semantic| Неверно задана директива,  Строка 1 |
| ! Form <note t>unit notelength  ! Form <unit x>idle notelength  main :  &unit u,  end. | Ошибка 115: |Semantic| Повторное использование директивы,  Строка 2 |
| main :  order <note s> unit print :  write << s  release >> 0.  end. | Ошибка 120: |Semantic| Объявление вложенной функции,  Строка 2 |
| main :  release >> 4  end. | Ошибка 121: |Semantic| Требуется объявление функции,  Строка 2 позиция 13 |
| main :  &unit i = 0  since i till 5 :  since i = i+1 till 10:  write << i;  ;  end. | Ошибка 122: |Semantic| Использование вложенного цикла недопустимо,  Строка 4 |
| main :  write << a  end. | Ошибка 123: |Lexical| Переменная не объявлена,  Строка 2 |
| main :  & var = 100  write << var  end. | Ошибка 124: |Lexical| Неверно задан тип данных,  Строка 2 |
| order <> [3]unit getArr :  [3]unit arr {1, 3, 5}  release >> arr.  main :  write << getArr()  end. | Ошибка 125: |Lexical| Неверно объявлена функция,  Строка 1 |
| main :  & unit x = 10  & unit x = 1000  write << x  end. | Ошибка 126: |Semantic| Повторное объявление переменной,  Строка 3 |
| order <>idle func :  release >> 1.  order <>note func :  release >> "./n".  main :  write << func()  end. | Ошибка 127: |Semantic| Повторное объявление функции,  Строка 3 |
| main :  [0]unit array {}  write << array[0]  end. | Ошибка 128: |Semantic| Некорректно задан размер массива,  Строка 2 |
| main :  write << array[1]  end. | Ошибка 129: |Semantic| Требуется объявление массива,  Строка 2 |
| main :  & unit value = 99359439595  write << value  end. | Ошибка 130: |Lexical| Превышено допустимое целочисленное значение,  Строка 2 |
| main :  [2]unit numbers {4, 43, 0X52, 63}  write << numbers[0]  end. | Ошибка 131: |Lexical| Слишком много значений инициализатора,  Строка 2 |
| main :  &unit wrong {4, 43, 0X52, 63}  write << wrong  end. | Ошибка 132: |Lexical| Неверная инициализация массива,  Строка 2 |
| main :  ["two"]unit arr {64, 15}  write << arr[0]  end. | Ошибка 133: |Semantic| Размер массива должен иметь целочисленный тип,  Строка 2 |
| !Adhere "File.txt"  main :  write << "start"  end. | Ошибка 134: |Semantic| Требуется имя файла,  Строка 1 |
| main :  write << 'End'  end. | Ошибка 135: |Semantic| Неверное использование имени файла,  Строка 2 |
| main :  write << "main"  . | Ошибка 200: |Syntax| Неверная структура программы,  Строка 2 |
| main :  write ("text")  end. | Ошибка 201: |Syntax| Ошибка в инструкции языка,  Строка 2 |
| main :  check 5 < 10  => write << "yes",  get >> write "no";  end. | Ошибка 202: |Syntax| Ошибка в условном блоке,  Строка 4 |
| main :  check 1 ? 0  => & note n = "okay"  end. | Ошибка 203: |Syntax| Ошибка в условной инструкции,  Строка 3 |
| main :  &unit y = 12 – write  end. | Ошибка 204: |Syntax| Ошибка в операндах,  Строка 2 |
| main :  &unit x = 36 % \* 7  end. | Ошибка 205: |Syntax| Ошибка в арифметической операции,  Строка 2 |
| main :  &idle b = 4 < ? 5  end. | Ошибка 206: |Syntax| Ошибка в операции сравнения,  Строка 2 |
| order < x , unit i> note func  : release >> "ret".  main :  write << func(0, 1)  end. | Ошибка 207: |Syntax| Ошибка в параметрах функции,  Строка 1 |
| order <unit x> note get  : release >> "String".  main :  write << get(check)  end. | Ошибка 208: |Syntax| Ошибка в параметрах вызываемой функции,  Строка 4 |
| main :  &unit z = "false" + "true"  end. | Ошибка 300: |Semantic| Несовпадение типов операндов,  Строка 2 |
| main :  &unit integer = "string"  end. | Ошибка 301: |Semantic| Присваивание недопустимого типа,  Строка 2 |
| order <unit x> idle bool  : release >> x.  main :  write << bool("true")  end. | Ошибка 302: |Semantic| Несовпадение типов параметров функции,  Строка 4 |
| order <unit x> idle use  : release >> x + 1.  main :  write << use(10, 20)  end. | Ошибка 303: |Semantic| Несовпадение количества параметров функции,  Строка 4 |
| order<> note hundred  : release >> 100.  main :  write << hundred()  end. | Ошибка 304: |Semantic| Тип возвращаемого значения не соответствует типу функции,  Строка 2 |
| !Adhere 'Nothing.std'  main :  write << "Hello/n"  end. | Ошибка 305: |Semantic| Не удается открыть файл,  Строка 1 |
| ! Form <note t>unit notelength  main :  write << notelength("Hello")  end. | Ошибка 306: |Semantic| Требуется подключение стандартной библиотеки,  Строка 1 |

# Заключение

* реализованы 5 арифметических операторов для вычисления выражений;
* поддерживаются 4 оператора сравнения для условных конструкций;
* определены четыре фундаментальных типа данных;
* поддерживается работа с целочисленными массивами;
* реализован цикл с условием и безусловным выходом;
* реализована условная конструкция с ветвлением;
* присутствует возможность писать комментарии к коду;
* поддерживается создание пользовательских функций;
* определены 8 внешних функций для двух типов данных;
* присутствует подключаемая стандартная библиотека;
* реализовано использование директив препроцессора;
* обрабатывается 50 возможных исключительных ситуаций;
* реализован оператор вывода;
* более 4000 строк кода.

# Приложение А

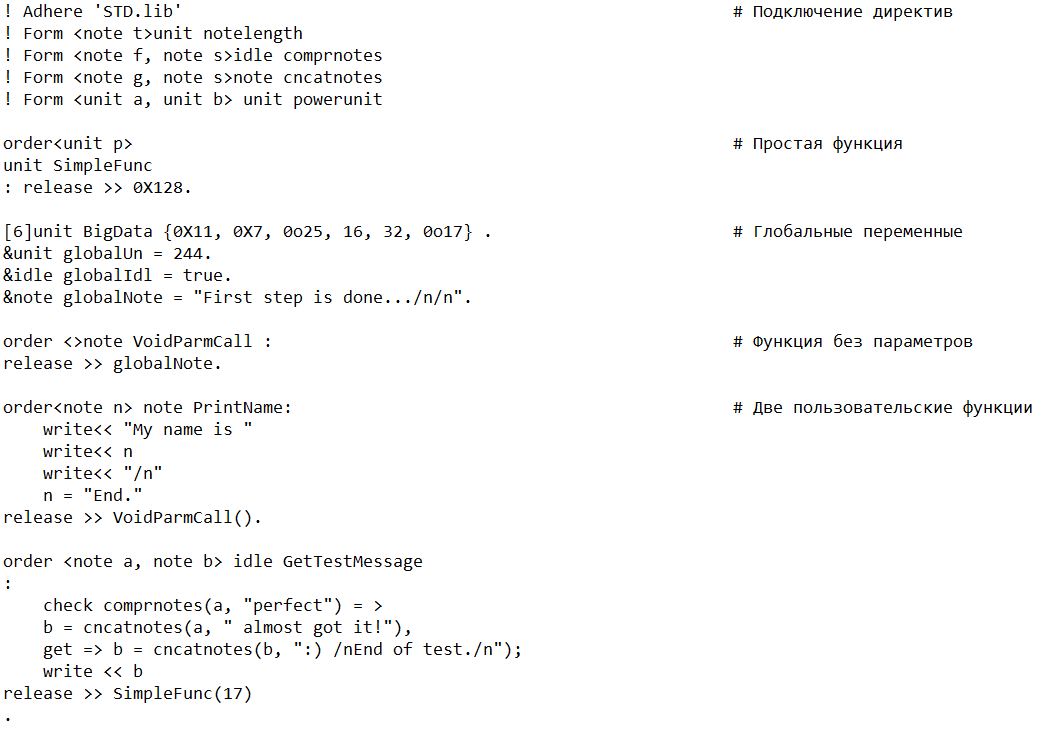
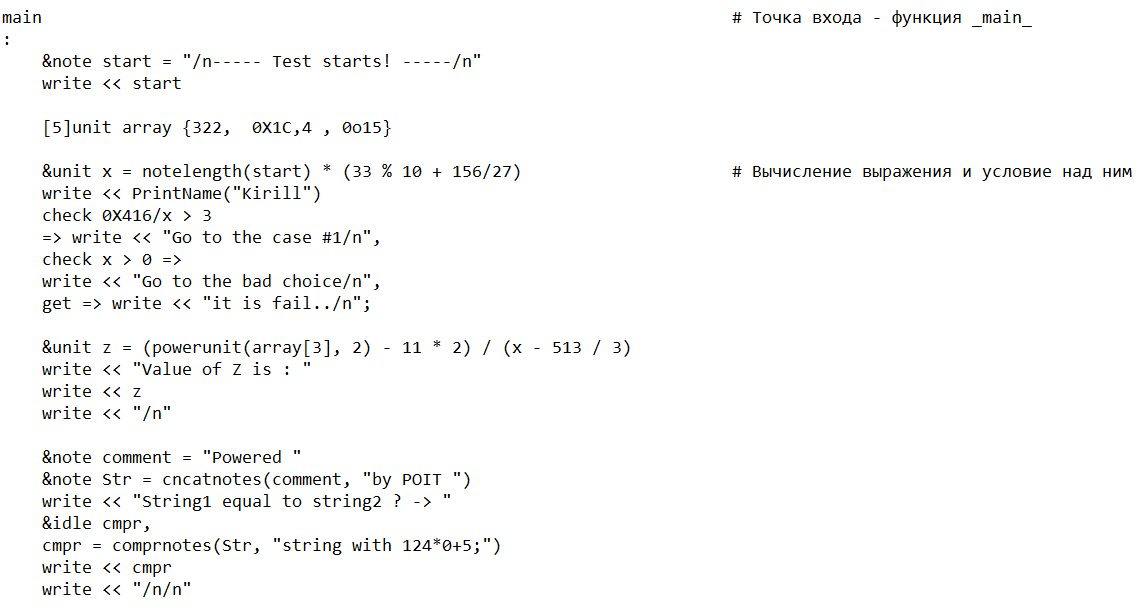


Рис. 8.1 - Исходный код



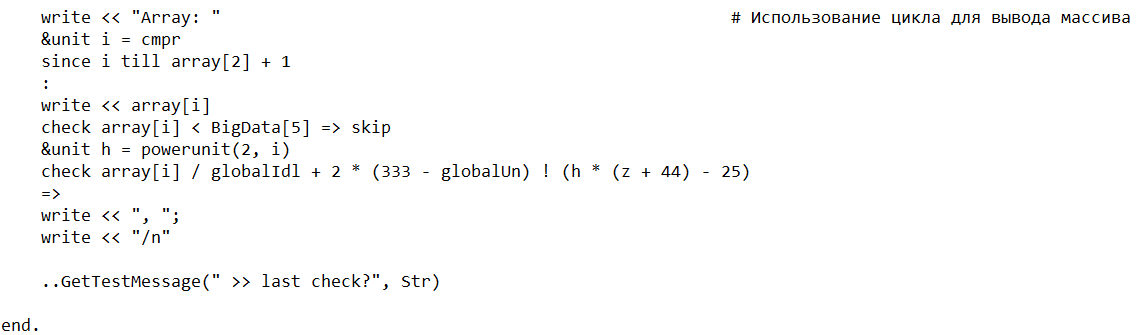


Рис. 8.2 - Исходный код

# Приложение Б

enum IDDATATYPE { UNIT = 1, NOTE, IDLE, ARRAY, MAIN};

enum IDTYPE { V = 1, F, P, L, O };

struct Entry

{

int idxfirstLEX;

char id[ID\_SIZE];

char realm[ID\_SIZE];

int block;

IDDATATYPE iddatatype;

IDTYPE idtype;

union

{

int vint;

struct

{

char len;

char str[TI\_STR\_MAXSIZE - 1];

}vstr[TI\_STR\_MAXSIZE];

struct Array

{

int len;

int val[TI\_ARR\_MAXSIZE - 1];

}varr[TI\_ARR\_MAXSIZE];

}value;

};

struct IdTable

{

int maxsize;

int size;

Entry\* table;

};

Рис. 9 - Структуры данных, используемые на фазе лексического анализа

**Приложение В**

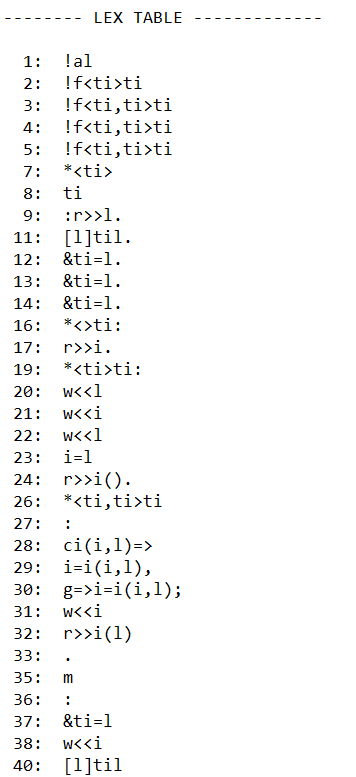


Рис. 10.1 - Таблица лексем

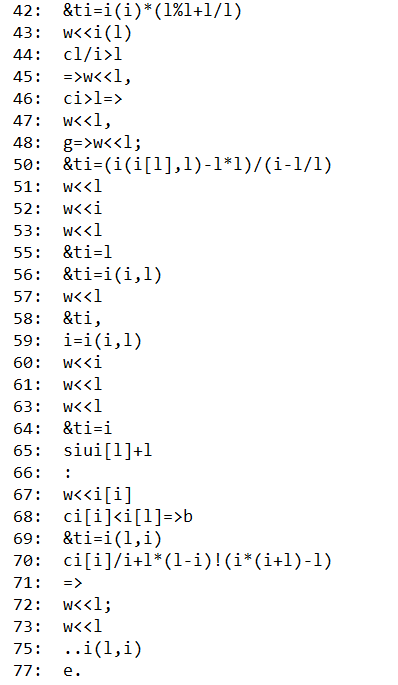


Рис. 10.2 - Таблица лексем

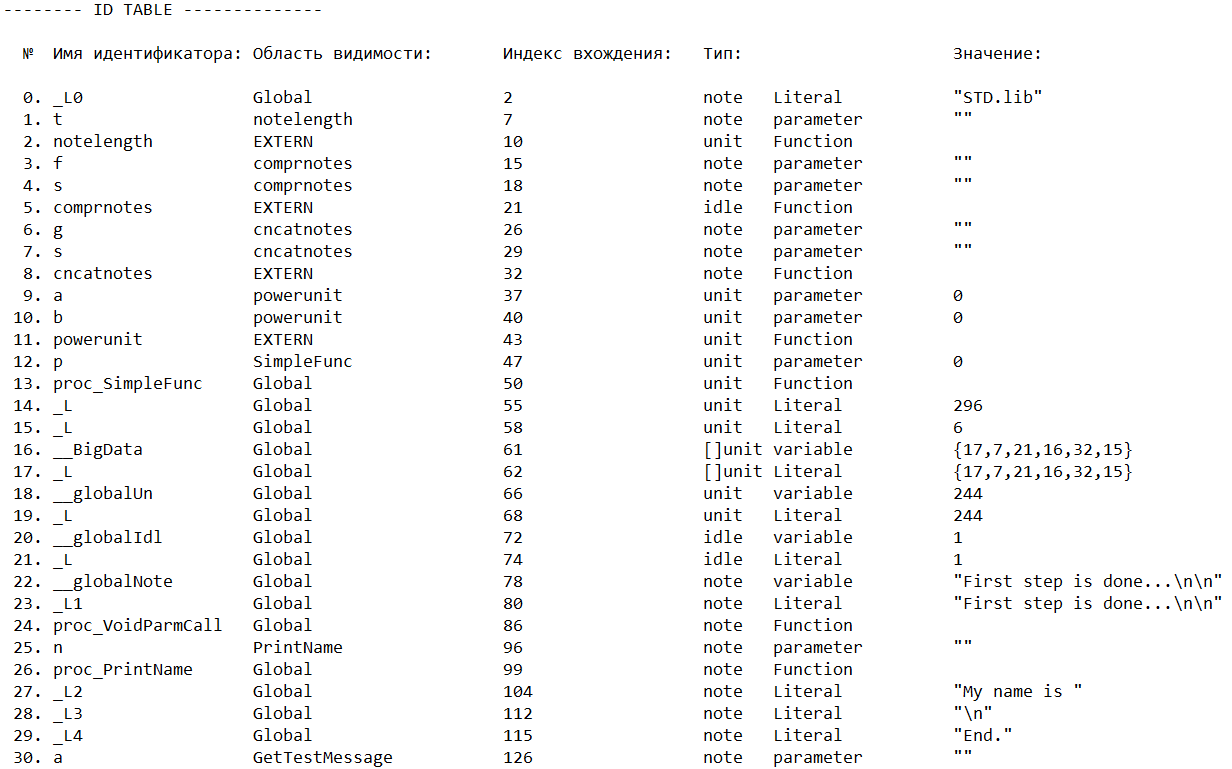
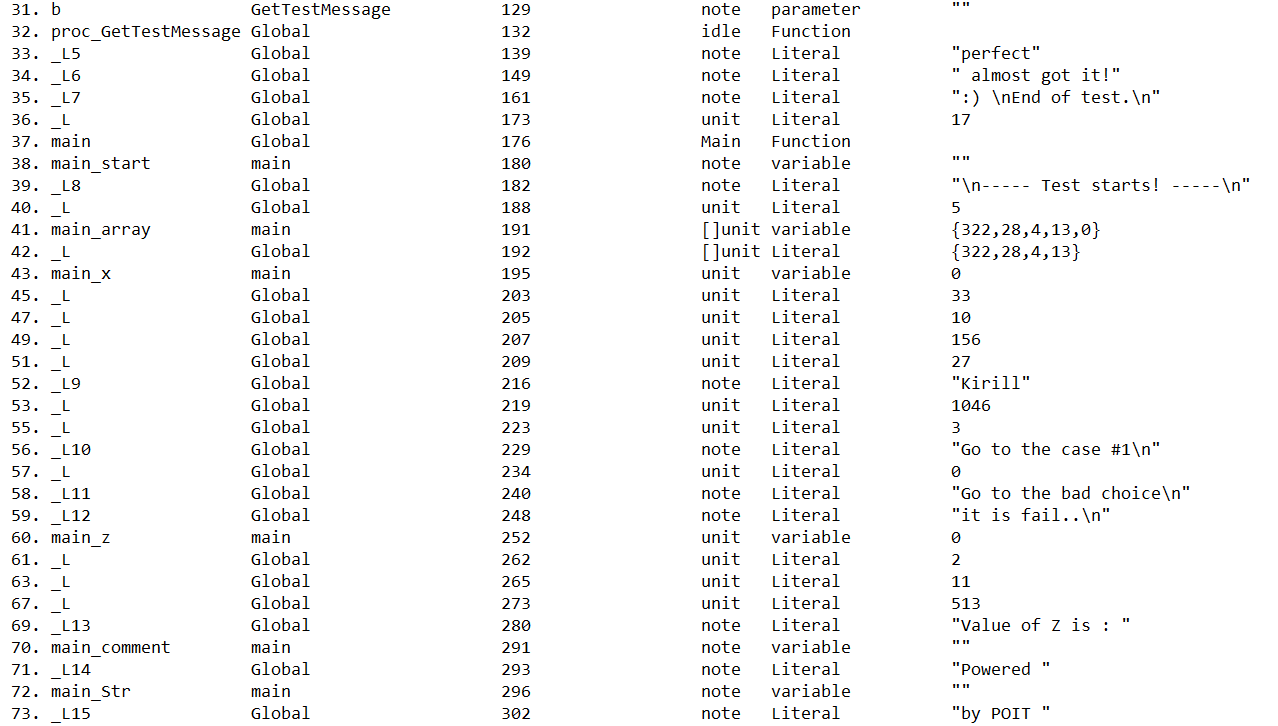


Рис. 11.1 - Таблица идентификаторов



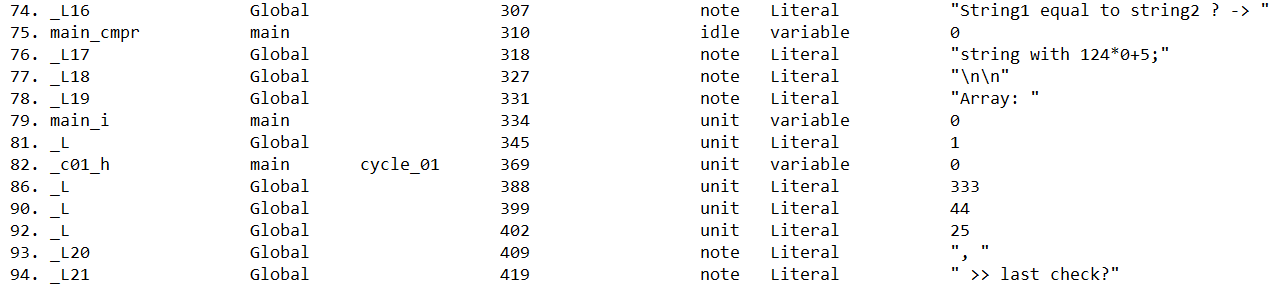


Рис. 11.1 - Таблица идентификаторов

# Приложение Г

Greibach\* grb = new Greibach(N('S'), T('$'), 9,

Rule(N('S'), 200, 14,

Rule::Chain(4, T('!'), T('a'), T('l'), N('S')),

Rule::Chain(7, T('!'), T('f'), T('<'), T('>'), T('t'), T('i'), N('S')),

Rule::Chain(8, T('!'), T('f'), T('<'), N('F'), T('>'), T('t'), T('i'), N('S')),

Rule::Chain(5, T('m'), T(':'), N('N'), T('e'), T('.')),

Rule::Chain(7, T('&'), T('t'), T('i'), T('='), T('l'), T('.'), N('S')),

Rule::Chain(8, T('['), T('l'), T(']'), T('t'), T('i'), T('l'), T('.'), N('S')),

Rule::Chain(11, T('\*'), T('<'), T('>'), T('t'), T('i'), T(':'), T('r'), T('>'), T('>'), N('E'), T('.')),

Rule::Chain(12, T('\*'), T('<'), N('F'), T('>'), T('t'), T('i'), T(':'), T('r'), T('>'), T('>'), N('E'), T('.')),

Rule::Chain(12, T('\*'), T('<'), T('>'), T('t'), T('i'), T(':'), N('N'), T('r'), T('>'), T('>'), N('E'), T('.')),

Rule::Chain(13, T('\*'), T('<'), N('F'), T('>'), T('t'), T('i'), T(':'), N('N'), T('r'), T('>'), T('>'), N('E'), T('.')),

Rule::Chain(12, T('\*'), T('<'), T('>'), T('t'), T('i'), T(':'), T('r'), T('>'), T('>'), N('E'), T('.'), N('S')),

Rule::Chain(13, T('\*'), T('<'), N('F'), T('>'), T('t'), T('i'), T(':'), T('r'), T('>'), T('>'), N('E'), T('.'), N('S')),

Rule::Chain(13, T('\*'), T('<'), T('>'), T('t'), T('i'), T(':'), N('N'), T('r'), T('>'), T('>'), N('E'), T('.'), N('S')),

Rule::Chain(14, T('\*'), T('<'), N('F'), T('>'), T('t'), T('i'), T(':'), N('N'), T('r'), T('>'), T('>'), N('E'), T('.'), N('S'))),

Rule(N('N'), 201, 32,

Rule::Chain(6, T('.'), T('.'), T('i'), T('('), T(')'), N('N')),

Rule::Chain(7, T('.'), T('.'), T('i'), T('('), N('W'), T(')'), N('N')),

Rule::Chain(6, T('&'), T('t'), T('i'), T('='), N('E'), N('N')),

Rule::Chain(5, T('&'), T('t'), T('i'), T(','), N('N')),

Rule::Chain(4, T('i'), T('='), N('E'), N('N')),

Rule::Chain(5, T('w'), T('<'), T('<'), N('E'), N('N')),

Rule::Chain(5, T('i'), T('='), N('E'), N('L'), N('N')),

Rule::Chain(6, T('w'), T('<'), T('<'), N('E'), N('L'), N('N')),

Rule::Chain(7, T('['), T('l'), T(']'), T('t'), T('i'), T('l'), N('N')),

Rule::Chain(8, T('s'), T('i'), T('u'), N('E'), T(':'), N('N'), T(';'), N('N')),

Rule::Chain(10, T('s'), T('i'), T('='), N('E'), T('u'), N('E'), T(':'), N('N'), T(';'), N('N')),

Rule::Chain(12, T('s'), T('&'), T('t'), T('i'), T('='), N('E'), T('u'), N('E'), T(':'), N('N'), T(';'), N('N')),

Rule::Chain(6, T('c'), N('E'), T('='), T('>'), N('A'), N('N')),

Rule::Chain(7, T('c'), N('E'), N('L'), T('='), T('>'), N('A'), N('N')),

Rule::Chain(4, T('c'), N('E'), N('V'), N('N')),

Rule::Chain(5, T('c'), N('E'), N('L'), N('V'), N('N')),

Rule::Chain(5, T('.'), T('.'), T('i'), T('('), T(')')),

Rule::Chain(6, T('.'), T('.'), T('i'), T('('), N('W'), T(')')),

Rule::Chain(5, T('&'), T('t'), T('i'), T('='), T('l')),

Rule::Chain(4, T('&'), T('t'), T('i'), T(',')),

Rule::Chain(3, T('i'), T('='), N('E')),

Rule::Chain(4, T('w'), T('<'), T('<'), N('E')),

Rule::Chain(4, T('i'), T('='), N('E'), N('L')),

Rule::Chain(5, T('w'), T('<'), T('<'), N('E'), N('L')),

Rule::Chain(6, T('['), T('l'), T(']'), T('t'), T('i'), T('l')),

Rule::Chain(7, T('s'), T('i'), T('u'), N('E'), T(':'), N('N'), T(';')),

Rule::Chain(9, T('s'), T('i'), T('='), N('E'), T('u'), N('E'), T(':'), N('N'), T(';')),

Rule::Chain(11, T('s'), T('&'), T('t'), T('i'), T('='), N('E'), T('u'), N('E'), T(':'), N('N'), T(';')),

Rule::Chain(5, T('c'), N('E'), T('='), T('>'), N('A')),

Rule::Chain(6, T('c'), N('E'), N('L'), T('='), T('>'), N('A')),

Rule::Chain(3, T('c'), N('E'), N('V')),

Rule::Chain(4, T('c'), N('E'), N('L'), N('V'))),

Rule(N('V'), 202, 4,

Rule::Chain(4, T('='), T('>'), N('A'), T(';')),

Rule::Chain(9, T('='), T('>'), N('A'), T(','), T('g'), T('='), T('>'), N('A'), T(';')),

Rule::Chain(7, T('='), T('>'), N('A'), T(','), T('c'), N('E'), N('V')),

Rule::Chain(8, T('='), T('>'), N('A'), T(','), T('c'), N('E'), N('L'), N('V'))),

Rule(N('A'), 203, 7,

Rule::Chain(1, T('b')),

Rule::Chain(5, T('.'), T('.'), T('i'), T('('), T(')')),

Rule::Chain(6, T('.'), T('.'), T('i'), T('('), N('W'), T(')')),

Rule::Chain(3, T('i'), T('='), N('E')),

Rule::Chain(4, T('w'), T('<'), T('<'), N('E')),

Rule::Chain(4, T('i'), T('='), N('E'), N('L')),

Rule::Chain(5, T('w'), T('<'), T('<'), N('E'), N('L'))),

Rule(N('E'), 204, 14,

Rule::Chain(1, T('i')),

Rule::Chain(1, T('l')),

Rule::Chain(4, T('i'), T('['), T('i'), T(']')),

Rule::Chain(4, T('i'), T('['), T('l'), T(']')),

Rule::Chain(3, T('('), N('E'), T(')')),

Rule::Chain(3, T('i'), T('('), T(')')),

Rule::Chain(4, T('i'), T('('), N('W'), T(')')),

Rule::Chain(2, T('i'), N('M')),

Rule::Chain(2, T('l'), N('M')),

Rule::Chain(5, T('i'), T('['), T('i'), T(']'), N('M')),

Rule::Chain(5, T('i'), T('['), T('l'), T(']'), N('M')),

Rule::Chain(4, T('('), N('E'), T(')'), N('M')),

Rule::Chain(4, T('i'), T('('), T(')'), N('M')),

Rule::Chain(5, T('i'), T('('), N('W'), T(')'), N('M'))),

Rule(N('M'), 205, 2,

Rule::Chain(2, T('v'), N('E')),

Rule::Chain(3, T('v'), N('E'), N('M'))),

Rule(N('L'), 206, 4,

Rule::Chain(2, T('?'), N('E')),

Rule::Chain(2, T('!'), N('E')),

Rule::Chain(2, T('<'), N('E')),

Rule::Chain(2, T('>'), N('E'))),

Rule(N('F'), 207, 2,

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

Rule::Chain(4, T('t'), T('i'), T(','), N('F'))),

Rule(N('W'), 208, 8,

Rule::Chain(1, T('i')),

Rule::Chain(1, T('l')),

Rule::Chain(3, T('i'), T(','), N('W')),

Rule::Chain(3, T('l'), T(','), N('W')),

Rule::Chain(4, T('i'), T('['), T('i'), T(']')),

Rule::Chain(4, T('i'), T('['), T('l'), T(']')),

Rule::Chain(6, T('i'), T('['), T('i'), T(']'), T(','), N('W')),

Rule::Chain(6, T('i'), T('['), T('l'), T(']'), T(','), N('W'))));

Рис.11 - Структура данных грамматики Грейбах

# Приложение Д

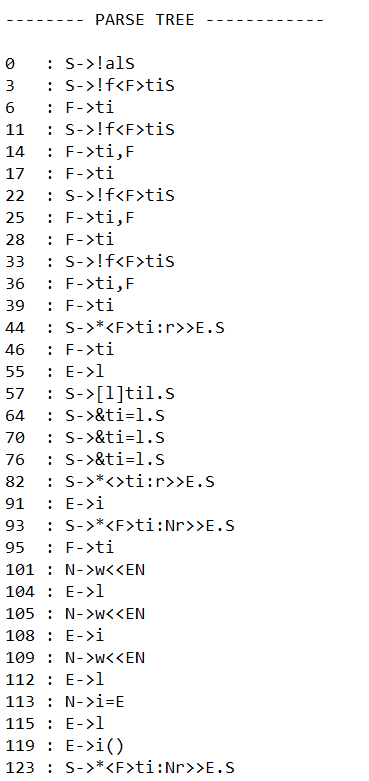


Рис. 12.1 - Дерево разбора

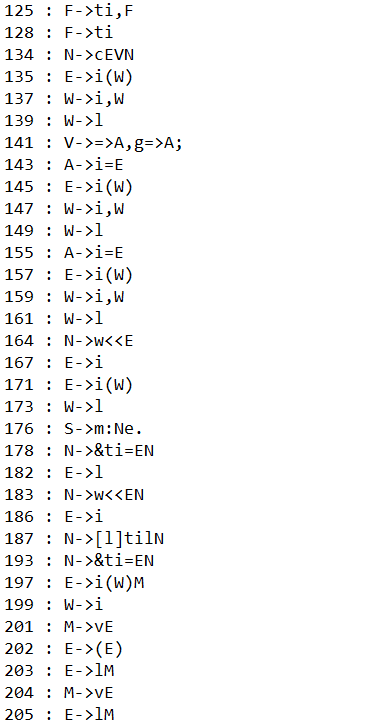


Рис. 12.2 - Дерево разбора

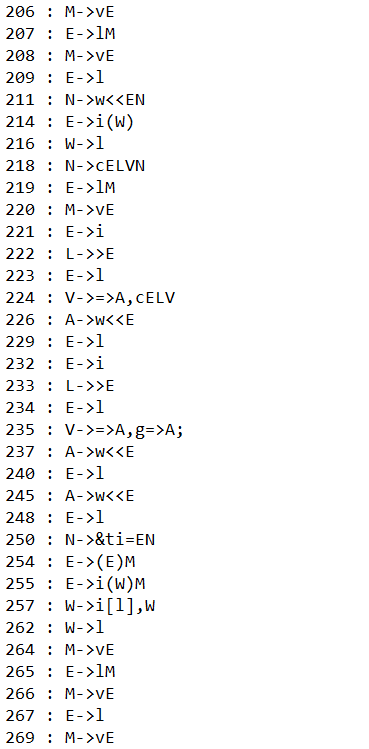


Рис. 12.3 - Дерево разбора

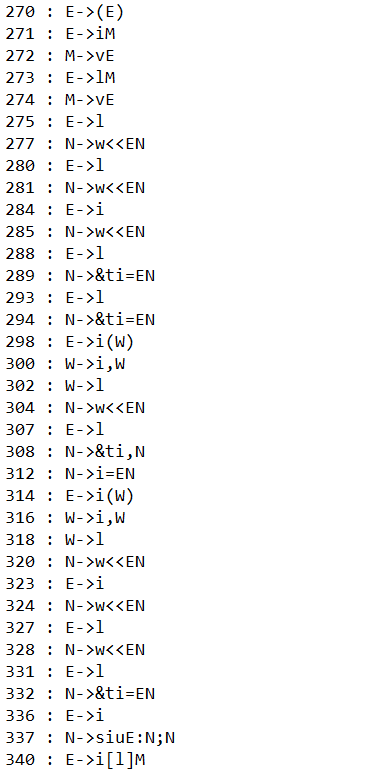
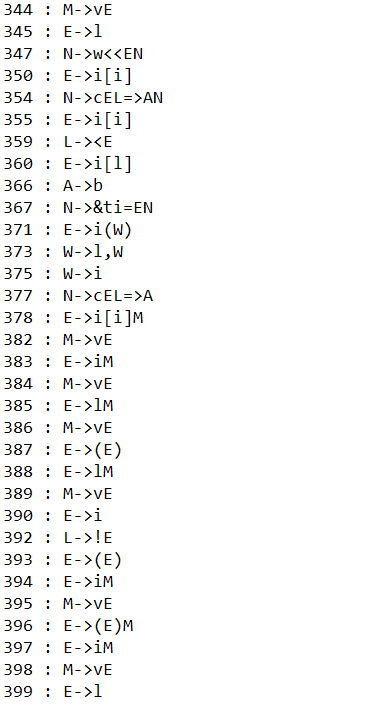


Рис. 12.4 - Дерево разбора



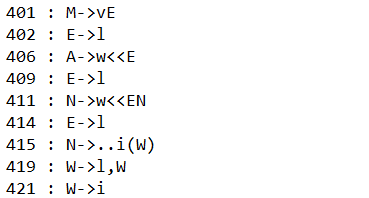


Рис. 12.5 - Дерево разбора

## Приложение Е

Таблица 1 – Результат генерации кода

|  |
| --- |
| .586  .model flat, stdcall  includelib kernel32.lib  includelib libucrt.lib  includelib STD.lib  .stack 4096  .const  ExitProcess PROTO : DWORD  writenote PROTO : DWORD  writeidle PROTO : DWORD  writeunit PROTO : DWORD  copynote PROTO : DWORD, : DWORD  notelength PROTO : DWORD  comprnotes PROTO : DWORD, : DWORD  cncatnotes PROTO : DWORD, : DWORD  powerunit PROTO : DWORD, : DWORD  overflow db 'ERROR: VARIABLE OVERFLOW', 0  null\_division db 'ERROR: DIVISION BY ZERO', 0  \_\_BigData DWORD 17, 7, 21, 16, 32, 15  \_L1 BYTE "First step is done...", 0dh, 0ah, 0dh, 0ah, 0  \_L2 BYTE "My name is ", 0  \_L3 BYTE 0dh, 0ah, 0  \_L4 BYTE "End.", 0  \_L5 BYTE "perfect", 0  \_L6 BYTE " almost got it!", 0  \_L7 BYTE ":) ", 0dh, 0ah, "End of test.", 0dh, 0ah, 0  \_L8 BYTE 0dh, 0ah, "----- Test starts! -----", 0dh, 0ah, 0  main\_array DWORD 322, 28, 4, 13, 0  \_L9 BYTE "Kirill", 0  \_L10 BYTE "Go to the case #1", 0dh, 0ah, 0  \_L11 BYTE "Go to the bad choice", 0dh, 0ah, 0  \_L12 BYTE "it is fail..", 0dh, 0ah, 0  \_L13 BYTE "Value of Z is : ", 0  \_L14 BYTE "Powered ", 0  \_L15 BYTE "by POIT ", 0  \_L16 BYTE "String1 equal to string2 ? -> ", 0  \_L17 BYTE "string with 124\*0+5;", 0  \_L18 BYTE 0dh, 0ah, 0dh, 0ah, 0  \_L19 BYTE "Array: ", 0  \_L20 BYTE ", ", 0  \_L21 BYTE " >> last check?", 0  .data  \_\_globalUn DWORD 244  \_\_globalIdl BYTE 1  \_\_globalNote BYTE "First step is done...", 0dh, 0ah, 0dh, 0ah, 232 DUP(0)  main\_start BYTE 255 DUP(0)  main\_x DWORD 0  main\_z DWORD 0  main\_comment BYTE 255 DUP(0)  main\_Str BYTE 255 DUP(0)  main\_cmpr BYTE 0  main\_i DWORD 0  \_c01\_h DWORD 0  .code  proc\_SimpleFunc PROC p: DWORD  push 296  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call writenote  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call writenote  push -2  call ExitProcess  EXIT:  pop eax  ret 4  proc\_SimpleFunc ENDP  proc\_VoidParmCall PROC  push offset \_\_globalNote  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call writenote  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call writenote  push -2  call ExitProcess  EXIT:  pop eax  ret 0  proc\_VoidParmCall ENDP  proc\_PrintName PROC n: DWORD  push offset \_L2  call writenote  push n  call writenote  push offset \_L3  call writenote  push offset \_L4  pop n  call proc\_VoidParmCall  push eax  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call writenote  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call writenote  push -2  call ExitProcess  EXIT:  pop eax  ret 4  proc\_PrintName ENDP  proc\_GetTestMessage PROC a: DWORD, b: DWORD  push offset \_L5  push a  call comprnotes  push eax  pop eax  test eax, eax  jz NEXT0  push offset \_L6  push a  call cncatnotes  push eax  pop b  jmp ENDC0  NEXT0:  push offset \_L7  push b  call cncatnotes  push eax  pop b  NEXT1:  ENDC0:  push b  call writenote  push 17  call proc\_SimpleFunc  push eax  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call writenote  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call writenote  push -2  call ExitProcess  EXIT:  pop eax  test eax, eax  jz IDLE0  mov eax, 1  jmp ENDI0  IDLE0:  mov eax, 0  ENDI0:  ret 8  proc\_GetTestMessage ENDP  main PROC  push offset \_L8  push offset main\_start  call copynote  push offset main\_start  call writenote  push offset main\_start  call notelength  push eax  push 33  push 10  pop ebx  pop eax  mov edx, 0  test ebx,ebx  jz EXIT\_DIV\_ON\_NULL  cdq  idiv ebx  push edx  push 156  push 27  pop ebx  pop eax  test ebx, ebx  jz EXIT\_DIV\_ON\_NULL  cdq  idiv ebx  push eax  pop ebx  pop eax  add eax, ebx  jo EXIT\_OVERFLOW  push eax  pop ebx  pop eax  imul eax, ebx  jo EXIT\_OVERFLOW  push eax  pop main\_x  push offset \_L9  call proc\_PrintName  push eax  call writenote  push 1046  push main\_x  pop ebx  pop eax  test ebx, ebx  jz EXIT\_DIV\_ON\_NULL  cdq  idiv ebx  push eax  push 3  pop ebx  pop eax  cmp eax, ebx  jle NGRTRX0  mov eax, 1  jmp ENDX0  NGRTRX0:  mov eax, 0  ENDX0:  push eax  pop eax  test eax, eax  jz NEXT2  push offset \_L10  call writenote  jmp ENDC1  NEXT2:  push main\_x  push 0  pop ebx  pop eax  cmp eax, ebx  jle NGRTRX1  mov eax, 1  jmp ENDX1  NGRTRX1:  mov eax, 0  ENDX1:  push eax  pop eax  test eax, eax  jz NEXT3  push offset \_L11  call writenote  jmp ENDC1  NEXT3:  push offset \_L12  call writenote  NEXT4:  ENDC1:  push 2  push 3  mov ebx, offset main\_array  pop eax  imul eax, 4  add ebx, eax  jo EXIT\_OVERFLOW  push [ebx]  call powerunit  push eax  push 11  push 2  pop ebx  pop eax  imul eax, ebx  jo EXIT\_OVERFLOW  push eax  pop ebx  pop eax  sub eax, ebx  jo EXIT\_OVERFLOW  push eax  push main\_x  push 513  push 3  pop ebx  pop eax  test ebx, ebx  jz EXIT\_DIV\_ON\_NULL  cdq  idiv ebx  push eax  pop ebx  pop eax  sub eax, ebx  jo EXIT\_OVERFLOW  push eax  pop ebx  pop eax  test ebx, ebx  jz EXIT\_DIV\_ON\_NULL  cdq  idiv ebx  push eax  pop main\_z  push offset \_L13  call writenote  push main\_z  call writeunit  push offset \_L3  call writenote  push offset \_L14  push offset main\_comment  call copynote  push offset \_L15  push offset main\_comment  call cncatnotes  push eax  push offset main\_Str  call copynote  push offset \_L16  call writenote  push offset \_L17  push offset main\_Str  call comprnotes  push eax  pop eax  test eax, eax  jz IDLE1  mov eax, 1  jmp ENDI1  IDLE1:  mov eax, 0  ENDI1:  mov main\_cmpr, al  movzx eax, main\_cmpr  push eax  call writeidle  push offset \_L18  call writenote  push offset \_L19  call writenote  movzx eax, main\_cmpr  push eax  pop main\_i  push 2  mov ebx, offset main\_array  pop eax  imul eax, 4  add ebx, eax  jo EXIT\_OVERFLOW  push [ebx]  push 1  pop ebx  pop eax  add eax, ebx  jo EXIT\_OVERFLOW  push eax  push main\_i  CYCLE0:  pop eax  pop ecx  cmp eax, ecx  jz ENDS0  push ecx  push main\_i  mov ebx, offset main\_array  pop eax  imul eax, 4  add ebx, eax  jo EXIT\_OVERFLOW  push [ebx]  call writeunit  push main\_i  mov ebx, offset main\_array  pop eax  imul eax, 4  add ebx, eax  jo EXIT\_OVERFLOW  push [ebx]  push 5  mov ebx, offset \_\_BigData  pop eax  imul eax, 4  add ebx, eax  jo EXIT\_OVERFLOW  push [ebx]  pop ebx  pop eax  cmp eax, ebx  jge NLESSX2  mov eax, 1  jmp ENDX2  NLESSX2:  mov eax, 0  ENDX2:  push eax  pop eax  test eax, eax  jz NEXT5  jmp ENDS0  NEXT5:  ENDC2:  push main\_i  push 2  call powerunit  push eax  pop \_c01\_h  push main\_i  mov ebx, offset main\_array  pop eax  imul eax, 4  add ebx, eax  jo EXIT\_OVERFLOW  push [ebx]  movzx eax, \_\_globalIdl  push eax  pop ebx  pop eax  test ebx, ebx  jz EXIT\_DIV\_ON\_NULL  cdq  idiv ebx  push eax  push 2  push 333  push \_\_globalUn  pop ebx  pop eax  sub eax, ebx  jo EXIT\_OVERFLOW  push eax  pop ebx  pop eax  imul eax, ebx  jo EXIT\_OVERFLOW  push eax  pop ebx  pop eax  add eax, ebx  jo EXIT\_OVERFLOW  push eax  push \_c01\_h  push main\_z  push 44  pop ebx  pop eax  add eax, ebx  jo EXIT\_OVERFLOW  push eax  pop ebx  pop eax  imul eax, ebx  jo EXIT\_OVERFLOW  push eax  push 25  pop ebx  pop eax  sub eax, ebx  jo EXIT\_OVERFLOW  push eax  pop ebx  pop eax  cmp eax, ebx  jz EQUALX3  mov eax, 1  jmp ENDX3  EQUALX3:  mov eax, 0  ENDX3:  push eax  pop eax  test eax, eax  jz NEXT6  push offset \_L20  call writenote  NEXT6:  ENDC3:  push main\_i  pop eax  inc eax  push eax  pop main\_i  push eax  jmp CYCLE0  ENDS0:  push offset \_L3  call writenote  push offset main\_Str  push offset \_L21  call proc\_GetTestMessage  push eax  pop eax  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call writenote  push -1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call writenote  push -2  call ExitProcess  EXIT:  push 0  call ExitProcess  main ENDP  end main |

# Литература

1. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768с.

2. Смелов, В.В. Курс лекций по предмету языки программирования – 2016

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

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