МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ

ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра программной инженерии

Специальность 6-05-0612-01 Программная инженерия

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора RIV-2024»

Выполнил студент Романов Игорь Вячеславович

(Ф.И.О.)

Руководитель проекта Наркевич Аделина Сергеевна

Заведующий кафедрой к.т.н., доц. Смелов Владимир Владиславович

Консультанты Наркевич Аделина Сергеевна

Нормоконтролер Наркевич Аделина Сергеевна

Курсовой проект защищен с оценкой

Минск 2024

Содержание

[Введение 5](#_Toc185366932)

[1. Спецификация языка программирования 6](#_Toc185366933)

[1.1. Характеристика языка программирования 6](#_Toc185366934)

[1.2. Определение алфавита языка программирования 6](#_Toc185366935)

[1.3. Применяемые сепараторы 6](#_Toc185366936)

[1.4. Применяемые кодировки 7](#_Toc185366937)

[1.5. Типы данных 7](#_Toc185366938)

[1.6. Преобразование типов данных 8](#_Toc185366939)

[1.7. Идентификаторы 8](#_Toc185366940)

[1.8. Литералы 9](#_Toc185366941)

[1.9. Объявление данных 9](#_Toc185366942)

[1.10. Инициализация данных 10](#_Toc185366943)

[1.11. Инструкции языка 10](#_Toc185366944)

[1.12. Операции языка 10](#_Toc185366945)

[1.13. Выражения и их вычисление 11](#_Toc185366946)

[1.14. Конструкции языка 11](#_Toc185366947)

[1.15. Область видимости идентификаторов 12](#_Toc185366948)

[1.16. Семантические проверки 12](#_Toc185366949)

[1.17. Распределение оперативной памяти на этапе выполнения. 12](#_Toc185366950)

[1.18. Стандартная библиотека и ее состав 13](#_Toc185366951)

[1.19. Ввод и вывод данных 13](#_Toc185366952)

[1.20. Точка входа 13](#_Toc185366953)

[1.21. Препроцессор 13](#_Toc185366954)

[1.22. Соглашения о вызовах 13](#_Toc185366955)

[1.23. Объектный код 14](#_Toc185366956)

[1.24. Классификация сообщений транслятора 14](#_Toc185366957)

[1.25. Контрольный пример 14](#_Toc185366958)

[2. Структура транслятора 15](#_Toc185366959)

[2.1. Компоненты транслятора их назначение и принципы взаимодействия 15](#_Toc185366960)

[2.2. Перечень входных параметров транслятора 16](#_Toc185366961)

[2.3 Протоколы, формируемые транслятором 16](#_Toc185366962)

[3. Разработка лексического анализатора 17](#_Toc185366963)

[3.1. Структура лексического анализатора 17](#_Toc185366964)

[3.2. Контроль входных символов 17](#_Toc185366965)

[3.3. Удаление избыточных символов 18](#_Toc185366966)

[3.4. Перечень ключевых слов 19](#_Toc185366967)

[3.5. Основные структуры данных 20](#_Toc185366968)

[3.6. Структура и перечень сообщений лексического анализатора 21](#_Toc185366969)

[3.7. Принцип обработки ошибок 21](#_Toc185366970)

[3.8. Параметры лексического анализатора 22](#_Toc185366971)

[3.9. Алгоритм лексического анализа 22](#_Toc185366972)

[3.10. Контрольный пример 23](#_Toc185366973)

[4. Разработка синтаксического анализатора 24](#_Toc185366974)

[4.1. Структура синтаксического анализатора 24](#_Toc185366975)

[4.2. Контекстно-свободная грамматика, описывающая синтаксис языка 24](#_Toc185366976)

[4.3. Построение конечного магазинного автомата 25](#_Toc185366977)

[4.4. Основные структуры данных 26](#_Toc185366978)

[4.5. Описание алгоритма синтаксического разбора 28](#_Toc185366979)

[4.6. Структура и перечень сообщений синтаксического анализатора 28](#_Toc185366980)

[4.7. Параметры синтаксического анализатора и режимы его работы 28](#_Toc185366981)

[4.8. Принцип обработки ошибок 29](#_Toc185366982)

[4.9. Контрольный пример 29](#_Toc185366983)

[5. Разработка семантического анализатора 30](#_Toc185366984)

[5.1. Структура семантического анализатора 30](#_Toc185366985)

[5.2. Функции семантического анализатора 30](#_Toc185366986)

[5.3. Структура и перечень сообщений семантического анализатора 31](#_Toc185366987)

[5.4. Принцип обработки ошибок 32](#_Toc185366988)

[5.5. Контрольный пример 32](#_Toc185366989)

[6. Вычисление выражений 33](#_Toc185366990)

[6.1. Выражения, допускаемые языком 33](#_Toc185366991)

[6.2. Польская запись и принцип ее построения 33](#_Toc185366992)

[6.3. Программная реализация обработки выражений 34](#_Toc185366993)

[6.4. Контрольный пример 34](#_Toc185366994)

[7. Генерация кода 35](#_Toc185366995)

[7.1. Структура генератора кода 35](#_Toc185366996)

[7.2. Представление типов данных в оперативной памяти 35](#_Toc185366997)

[7.3. Статическая библиотека 36](#_Toc185366998)

[7.4. Особенности алгоритма генерации кода 36](#_Toc185366999)

[7.5. Входные параметры, управляющие генерацией кода 37](#_Toc185367000)

[7.6. Контрольный пример 37](#_Toc185367001)

[8. Тестирование транслятора 38](#_Toc185367002)

[8.1. Общие положения 38](#_Toc185367003)

[8.2 Результаты тестирования 38](#_Toc185367004)

[Заключение 40](#_Toc185367005)

[Список использованных источников 41](#_Toc185367006)

[Приложение А 42](#_Toc185367007)

[Приложение Б 43](#_Toc185367008)

[Приложение В 45](#_Toc185367009)

[Приложение Г 47](#_Toc185367010)

[Приложение Д 49](#_Toc185367011)

[Приложение Е 51](#_Toc185367012)

[Приложение Ж 54](#_Toc185367013)

**Введение**

Целью курсового проекта является разработка собственного языка программирования RIV-2024 и компилятора для его обработки. Этот язык предназначен для учебных целей и ориентирован на изучение принципов разработки языка программирования и транслятора. Основное внимание уделено созданию простого и понятного синтаксиса, который позволит эффективно выполнять базовые операции над данными и функциями.

Для создания собственного языка программирования необходимо определить его алфавит, синтаксис и семантику. Алфавит языка представляет собой базовый набор символов, разрешенных для использования в программе. Синтаксис описывает правила формирования корректных выражений, а семантика – их смысловое значение. Описание языка будет представлено в форме Бэкуса-Наура, что обеспечит формализацию правил и позволит избежать двусмысленностей при разработке.

Транслятор играет ключевую роль в процессе преобразования исходного текста программы, написанного на языке RIV-2024, в исполняемую форму. Для этого требуется реализация последовательных этапов: лексического анализа, синтаксического анализа, семантического анализа и генерации объектного кода. Каждый из этих этапов будет подробно рассмотрен и описан в проекте.

В соответствии с приведенной выше целью были поставлены задачи курсового проекта:

* разработать спецификацию языка программирования RIV-2024, включая описание его синтаксиса и семантики;
* определить структуру транслятора, описав его основные компоненты и их взаимодействие;
* разработать структуры основных анализаторов: лексического, синтаксического и семантического;
* реализовать процесс генерации объектного кода, демонстрирующий функциональные возможности языка;
* создать контрольный пример программы, написанной на языке RIV-2024, для проверки и демонстрации работоспособности транслятора;
* провести тестирование транслятора с целью выявления ошибок и подтверждения корректности его работы.

Решения каждой из поставленных целей будут приведены в соответствующих разделах курсового проекта.

**1. Спецификация языка программирования**

**1.1. Характеристика языка программирования**

Разрабатываемый язык программирования RIV-2024 – компилируемый, высокоуровневый, императивный, строго типизированный.

**1.2. Определение алфавита языка программирования**

Язык программирования RIV-2024 использует стандартную кодировку символов Windows-1251. Его структура представлена в форме Бэкуса-Наура в таблице 1.1.

Таблица 1.1 – Алфавит языка программирования RIV-2024

|  |
| --- |
| <строчная буква латинского алфавита>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z |
| <прописная буква латинского алфавита>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z |
| <цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <символ- сепаратор>::= ' '|,|(|)|{|}|;| "" | |
| <арифметическая операция>::= + | - | \* |
| <операции сравнения>::= > | < |
| <побитовая операция>::= &| "|" | ~ |

**1.3. Применяемые сепараторы**

Сепараторы используются для логического разделения конструкций, функций и лексем.

Сепараторы, используемые в языке программирования RIV-2024, и их назначение представлены в таблице 1.2.

Таблица 1.2 – Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| пробел | Разделитель лексем. Не допускается в ключевых словах и идентификаторах |
| ; (точка с запятой) | Разделитель строк |
| {} | Определитель блока функций |
| () | Определитель параметров функций |
| , (запятая) | Разделитель параметров функции |
| & | ~ | Выполнение побитовых операций |
| = | Оператор присвоения |
| + - \* | Арифметические операции |
| > < | Операции сравнения |

**1.4. Применяемые кодировки**

Для написания исходного кода программы на языке RIV-2024 используется кодировка Windows-1251. Символы, поддерживающиеся в кодировке Windows-1251 представлены на рисунке 1.1.



Рисунок 1.1 – Таблица кодировки Windows - 1251

**1.5. Типы данных**

Язык программирования RIV-2024 поддерживает использование следующих типов данных: целочисленный (1 байт), целочисленный (4 байта), символьный, строковый. Описание типов данных представлено в таблице 1.3.

Таблица 1.3 – Типы данных

|  |  |
| --- | --- |
| Тип данных | Описание |
| Целочисленный (1 байт) byte | Целочисленный тип данных. Занимает 1 байт. Диапазон значений: от -128 до 127.  Возможные операции:   * арифметические (+, - \*); * логические (&, |, ~) * операции сравнения (>, <)   Инициализация по умолчанию: 0. |
|  |  |

Окончание таблицы 1.3

|  |  |
| --- | --- |
| Тип данных | Описание |
| Целочисленный (4 байта) int | Целочисленный тип данных. Занимает 4 байта. Диапазон значений: от -216 до 216 - 1.  Возможные операции:   * арифметические (+, - \*); * логические (&, |, ~) * операции сравнения (>, <)   Инициализация по умолчанию: 0 |
| Логический bool | Логический тип данных. Занимает 1 байт, значение 0 или false или любое отличное от 0 или true. Инициализация по умолчанию: 0. |
| Символьный char | Символьный тип данных. Занимает 1 байт. Диапазон значений: все символы кодировки Windows-1251. Инициализация по умолчанию: нуль-символ ('\0') |
| Строковый str | Строковый тип данных. 1 символ занимает 1 байт. Диапазон значений: все символы кодировки Windows-1251. Инициализация по умолчанию: строка нулевой длины (также содержит нуль-символ – признак конца строки).  Максимальная длина строки - 255. |

**1.6. Преобразование типов данных**

Все типы данных определены однозначно и не могут быть преобразованы в другие.

**1.7. Идентификаторы**

Идентификатор языка – это последовательность символов, используемая в качестве имени для идентификации сущностей в языке программирования. В RIV-2024 идентификаторы используются для именования переменных, функций и их параметров.

Формальное описание идентификатора представлено ниже в форме Бэкуса-Наура.

<идентификатор> ::= <буква> {<буква> | <цифра> | '\_'};

<буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z ;

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .

В языке RIV-2024 идентификаторы не могут содержать символов кириллицы, специальных символов за исключением нижнего подчеркивания; не могут начинаться с цифры.

Примеры правильных идентификаторов: myVar, name, full\_price.

Примеры неправильных идентификаторов: 0var, operation+.

**1.8. Литералы**

Литералы языка программирования – последовательность символов, используемая для именования неизменяемых значений в коде.

В языке RIV-2024 предусмотрены следующие типы литералов: целочисленный, строковый. Описание литералов представлено в таблице 1.4.

Таблица 1.4 – Литералы RIV-2024

|  |  |
| --- | --- |
| Литерал | Описание |
| Целочисленный | Может быть представлен в десятичном виде, где используются цифры от 0 до 9 в виде последовательности |
| Логический | Принимает значение 0 или false или любое отличное от 0 или true. |
| Строковый | Состоит из символов, заключенных в двойные кавычки. Максимальное число символов в литерале – 255. Использование двойных кавычек внутри строкового литерала не допускается. Применяется кодировка Windows-1251 |

Пример правильных литералов: "OK", 'a'.

Пример неправильных литералов: 'NOT OK', "n".

**1.9. Объявление данных**

При объявлении переменных в языке RIV-2024 необходимо учитывать следующие правила:

* необходимо объявлять переменную до её использования;
* при объявлении используется ключевое слово let;
* переменная может быть объявлена в любом программном блоке, если она уже не была объявлена в данной области видимости.

Объявление функции схоже с объявлением обычной переменной:

* сначала указывается возвращаемый функцией тип данных, разрешенный в языке программирования;
* потом используется ключевое слово function;
* далее определяется имя функции;
* в круглых скобках указываются параметры. Скобки остаются пустыми, если параметры не передаются;
* затем в фигурных скобках определяется блок функции;
* в блоке функции используется ключевого слово return для возврата значения.

**1.10. Инициализация данных**

По умолчанию, переменным различных типов данных присваиваются начальные значения. Например, целочисленные имеют значение 0, строковым присваивается пустая строка, ограниченная двойными кавычками.

Язык программирования RIV-2024 использует явную инициализацию. Пример инициализации: let int number = 13; let ch s = 'a'.

**1.11. Инструкции языка**

Инструкции, их описание и применения языка программирования RIV-2024 представлены в таблице 1.5.

Таблица 1.5 – Инструкции языка RIV-2024

|  |  |  |
| --- | --- | --- |
| Инструкция | Описание | Пример |
| Объявление переменной | let <тип данных> <идентификатор>; | let int number; |
| Объявление функции | <тип данных> function <идентификатор функции> (<список параметров>) {<блок функции>}; | int function myFunc(a, b) {… return…}; |
| Вызов функции | <идентификатор функции> ( <параметры функции> | <пусто>); | getSum (a, b);  writeMessage(); |
| Присваивание | <идентификатор> = <литерал>;  <идентификатор> = <выражение>;  <идентификатор1> = <идентификатор 2>; | number = 27;  number = 14 – 1;  number = number2; |
| Вывод данных | print(<идентификатор | литерал>); | print(number) |
| Разветвление | if (<условное выражение>) {<тело блока истины>} else {<тело блока лжи>} | if (number > 27) { print(1); } else { print(2);} |
| Инициализация | let <тип данных> <идентификатор> = <выражение>; | let int number = 13; |

**1.12. Операции языка**

Язык программирования RIV-2024 поддерживает различные типы операций. Их описание, назначение, приоритетность выполнения и свойства представлены в таблице 1.6.

Таблица 1.6 – Операции языка RIV-2024

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Назначение | Приоритетность операции | Свойства |
| () | Приоритетность операций | 0 | Ассоциативность, коммутативность, дистрибутивность |
| ~ | Инверсия | 1 | Унарный |
| \* | Умножение | 2 | Бинарные |
| + - | Сложение, вычитание | 3 | Бинарные |
| > < | Сравнение чисел | 4 | Антикоммутативность |
| & | | Побитовые операции | 5 | Бинарные |
| , | Запятая | 6 | Ассоциативность |

**1.13. Выражения и их вычисление**

Предусмотрены следующие правила составления выражений:

* рассматриваются слева направо;
* для изменения приоритета операции используются круглые скобки ();
* каждое выражение должно заканчиваться сепаратором;
* при совершении операций над типами операнды строго должно быть одного целочисленного типа: byte или int;
* может содержать вызов функции.

Далее каждое выражение приводится к виду обратной польской записи для генерации объектного кода на языке ассемблер.

**1.14. Конструкции языка**

Программные конструкции языка программирование RIV-2024 приведены в таблице 1.7.

Таблица 1.7 – Конструкции языка RIV-2024

|  |  |  |
| --- | --- | --- |
| Конструкция | Описание | Пример |
| Функция | <тип данных> function <идентификатор> (<тип\_данных> <идентификатор>, …)  {…  return <идентификатор | литерал>  } | int function getSum(int a, int b) {  return a+b;  } |
|  |  |  |

Окончание таблицы 1.7

|  |  |  |
| --- | --- | --- |
| Конструкция | Описание | Пример |
| Условный оператор | if (<условие>) {<инструкции языка>}else {<инструкции языка>} | if (number > 27){  print( "YES");}  else {  print("NO");  } |

**1.15. Область видимости идентификаторов**

Область видимости в языке RIV-2024 имеет следующие особенности: все идентификаторы, параметры и переменные, объявленные внутри блока функции, являются локальными и доступны лишь в ее пределах. Идентификаторы, объявленные вне функций, видны в любой точке кода. Для создания искусственной области видимости в языке RIV-2024 используются фигурные скобки { ... }.

**1.16. Семантические проверки**

Семантические проверки, предусмотренные в языке RIV-2024, представлены в таблице 1.8.

Таблица 1.8 – Семантические проверки

|  |  |
| --- | --- |
| № | Правило |
| 1 | Наличие точки входа в программу в main |
| 2 | Дублирование main |
| 3 | Каждый идентификатор должен быть объявлен только один раз (с учетом блочной структуры объявлений) |
| 4 | Идентификаторы усекаются до 9 символов |
| 5 | Не допускается совпадения названий идентификаторов с ключевыми словами |
| 6 | Все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции |
| 7 | Все идентификаторы должны быть обвялены в той же области видимости перед использованием |
| 8 | Инструкция if должны содержать логический оператор |
| 9 | Возвращаемое значение функции должно совпадать с типом функции |

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

Транслированный код использует два отдельных сегмента памяти: сегмент констант, где хранятся все литералы, и сегмент данных, предназначенный для переменных и параметров функций. Локальная область видимости в исходном коде достигается за счет использования определенных правил именования идентификаторов, включая префиксы, которые обеспечивают локальность на уровне исходного текста. В оттранслированном коде на языке ассемблера все переменные располагаются в глобальной области видимости.

**1.18. Стандартная библиотека и ее состав**

Язык RIV-2024 предлагает несколько функций стандартной библиотеки, представленных в таблице 1.9.

Таблица 1.9 – Стандартная библиотека языка RIV-2024

|  |  |
| --- | --- |
| Функция | Описание |
| byte abs(byte num) | Вычисляет модуль числа. Принимает на вход byte аргумент. Возвращает модуль числа |
| byte rest(byte num1, byte num2) | Вычисляет остаток от деления числа на число. Принимает делимое и делитель. Оба должны быть одного типа. Возвращает результат деления |
| void writestr(char\* str) | Вывод в консоль строковой переменной |
| void writechar(char ch) | Вывод в консоль символьной переменной |
| void writeint(int n) | Вывод в консоль целочисленной переменной |
| void writebool(bool b) | Вывод в консоль логической переменной: в случае значения 0 – false, в остальных случаях – true |

**1.19. Ввод и вывод данных**

Вывод данных осуществляется с помощью ключевого слова «print». В качестве аргумента могут использоваться числовые и строковые идентификаторы, а также литералы: print (<идентификатор>|<литерал>).

Ввод данных не предусмотрен.

**1.20. Точка входа**

Точкой входа в программе является ключевое слово «main». Точка входа не может отсутствовать, быть переопределена или продублирована.

**1.21. Препроцессор**

На этапе генерации кода в языке ассемблера стандартные библиотеки библиотека с использованием директивы includelib**.**

**1.22. Соглашения о вызовах**

Язык программирования RIV-2024 использует соглашение о вызовах stdcall. В данном соглашении параметры передаются через стек, порядок передачи параметров – справа налево, стек освобождает вызываемый код.

**1.23. Объектный код**

Язык программирования RIV-2024 транслируется в исходный код на языке ассемблера.

**1.24. Классификация сообщений транслятора**

В соответствии с префиксами транслятор будет генерировать текст сообщений об ошибке и выводить их в файл протокола. В таблице 1.10 представлены возможные классификации сообщений транслятора.

Таблица 1.10 – Классификация сообщений транслятора

|  |  |
| --- | --- |
| Диапазон | Описание |
| 0 – 3 | Системные ошибки |
| 100 – 112 | Ошибки параметров |
| 113 – 124 | Ошибки лексического анализа |
| 600 – 606 | Синтаксические ошибки |
| 700 – 709 | Ошибки семантического анализа |

**1.25. Контрольный пример**

Контрольный пример, демонстрирующий возможности языка программирования RIV-2024 представлен в приложении A.

**2. Структура транслятора**

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

Транслятор – программа, преобразующая исходный код на одном языке программирования в исходный код на другом языке. Принцип работы транслятора приведен на рисунке 2.1.

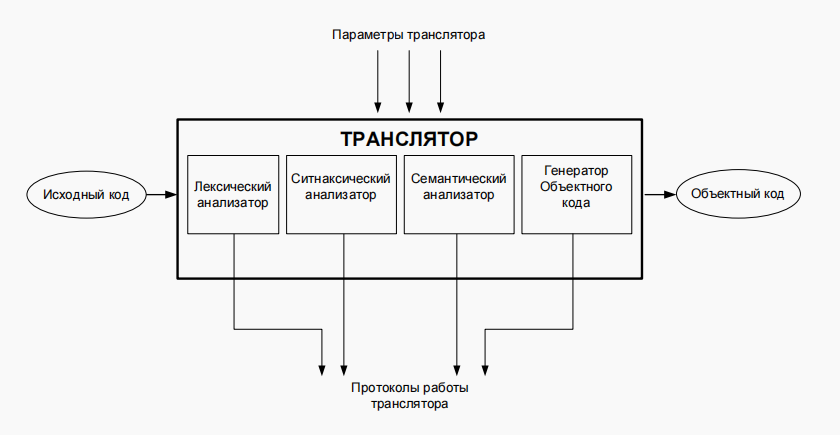
****

Рисунок 2.1 – Схема работы транслятора

Работу транслятора можно разделить на 4 логически связанные составляющие: лексический анализ, синтаксический анализ, семантический анализ, генерация в объектный код.

Лексический анализ – первая фаза трансляции. Одной из задач данной фазы является максимальное упрощение кода путём замены его на набор лексем и удалением лишних символов. На вход принимается файл с исходным кодом. В результате обработки создаются таблица лексем и таблица идентификаторов.

Синтаксический анализатор – этап трансляции, выполняющий проверку исходного кода на соответствие правилам заданной грамматики. Входными данными для данной фазы служат таблицы лексем и идентификаторов, сформированные на предыдущем этапе. Итогом работы синтаксического анализатора является построение дерева разбора, отражающего структуру исходного кода в соответствии с его грамматическими правилами.

Семантический анализатор отвечает за выполнение семантической проверки исходного кода, выявляя ошибки, которые невозможно обнаружить с помощью контекстно-свободной грамматики. На вход поступают таблицы лексем и идентификаторов. Семантический анализатор обеспечивает контроль корректности использования конструкций языка и их согласованности.

Генератор кода — это фаза трансляции, которая преобразует промежуточное представление программы в код на языке ассемблера. В процессе генерации используются таблицы лексем и идентификаторов. Результатом работы является выходной файл с расширением .asm.

**2.2. Перечень входных параметров транслятора**

Входные параметры, управляющие работой транслятора, представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора RIV-2024

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Значение по умолчанию |
| -in:<путь к in-файлу> | Обязательный. Файл с исходным кодом на языке RIV-2024 с расширением .txt | Не предусмотрено |
| -log:<путь к log-файлу> | Необязательный. Файл журнала для вывода протоколов работы программы | <имя in-файла>.log |
| -out:<путь к out-файлу> | Необязательный. Выходной файл с результатом работы транслятора | <имя in-файла>.out |

**2.3 Протоколы, формируемые транслятором**

В таблице 2.2 представлены протоколы, формируемые в результате работы транслятора.

Таблица 2.2 – Протоколы, формируемые транслятором RIV-2024

|  |  |
| --- | --- |
| Формируемый протокол | Назначение |
| Файл протокола, заданный параметром «-log:» | Файл с протоколом работы транслятора языка программирования RIV-2024. Содержит информацию про входные параметры, общем количестве символов и строк, протокол работы синтаксического анализатора, полученный на этапе синтаксического анализа. В случае ошибки текст, строка и позиция ошибки будет записан в файл. |
| Выходной файл «IT.txt» | Файл содержит таблицу идентификаторов, сформированную во время лексического анализа. |
| Выходной файл «LT.txt» | Файл содержит таблицу лексем, сформированную во время лексического анализа. |
| Выходной файл, заданный параметром «-asm: « | Результат работы транслятора – файл с исходным кодом на языке ассемблера |

**3. Разработка лексического анализатора**

**3.1. Структура лексического анализатора**

Лексический анализатор – это первая фаза транслятора, преобразующая исходный текст программы, заменяя лексические единицы языка лексемами. Схема работы лексического анализатора представлена на рисунке 3.1.

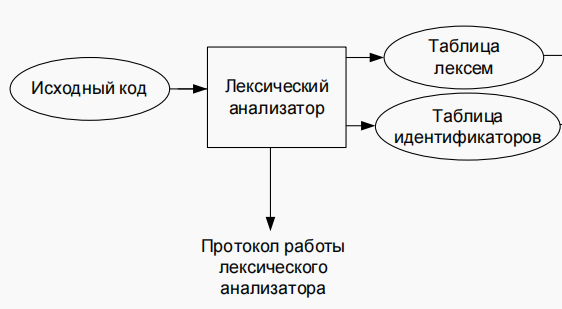


Рисунок 3.1 – Схема работы лексического анализа

На вход лексического анализатора подается исходный код. В результате формируется таблица лексем и идентификаторов. Работа лексического анализатора управляется таблицей разрешенных символов, списком ключевых слов. Их подробное описание будет приведено в соответствующих разделах ниже.

**3.2. Контроль входных символов**

На рисунке 3.2. приведена таблица допустимости символов, используемых для контроля правильности использования исходного текста.

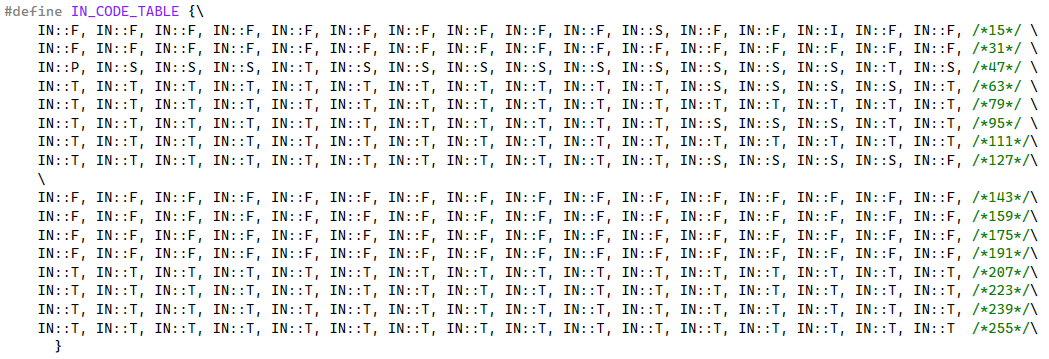


Рисунок 3.2 – Таблица допустимости символов

Таблица допустимости символов основана на кодировке Windows-1251. Индекс элемента соответствует его коду в таблице символов кодировке. Соответствие символов контроля и их назначение представлено в таблице 3.1.

Таблица 3.1 – Соответствие символов и их значений

|  |  |  |
| --- | --- | --- |
| Символ | Расшифровка | Описание |
| T | Разрешенный символ | Все используемые символы, разрешенные алфавитом языка |
| F | Запрещенный символ | Символы, запрещенные для использования |
| I | Игнорируемый символ | Символы, игнорируемые анализатором |
| S | Символ-сепаратор | Разграничивают символы алфавита языка между собой |
| P | Символ-пробел | Разделение токенов |

**3.3. Удаление избыточных символов**

До начала работы лексического анализатора текст исходного кода подлежит обработке и из него удаляются избыточные символы. Избыточными символами являются повторяющиеся символы пробела. Исключением являются пробелы, заключенные в кавычки.

Ниже приведен принцип удаления избыточных символов.

1. Создаем строковый буфер для хранения обработанного текста.
2. Поочередно считываем символы из файла с исходным кодом программы
3. Каждый символ проверяется по таблице допустимости, и зависимости от классификации допустимости происходит разветвление.
   1. Допустимый(T). Символ добавляется в буфер; в случае игнорирования (I) символ игнорируется.
   2. Запрещенный(F). Символ изменяется на «^» и добавляется в специально выделенный массив для хранения запрещенных символов.
   3. Игнорируемый(I). Символ пропускается.
   4. Пробел(P). Если пробел находится в начале строки, в конце строки или рядом с другим пробелом (и кавычки не открыты), он игнорируется. Иначе пробел добавляется в буфер.
   5. Сепаратор(S). Добавляется в буфер, но если символ добавляется перед пробелом или после него, то последний заменяется сепаратором.
4. Если строка не последняя в файле, то добавляется символ переноса строки.
5. В конце обработки добавляется нуль-символ и освобождается память.

**3.4. Перечень ключевых слов**

При лексическом анализе исходный код преобразуется, заменяя лексические единицы соответствующими лексемами. Таблица ключевых слов представлен в таблице 3.2.

Таблица 3.2 –Таблица ключевых слов RIV-2024

|  |  |  |
| --- | --- | --- |
| Фраза RIV-2024 | Лексема | Примечание |
| byte  int ch str | t | Название типов данных |
| идентификатор | i | Максимальная длина идентифкатора – 9 |
| литералы | l | byte, int, ch или string значение |
| функция | f |  |
| let | d | объявление/инициализация переменной |
| return | r |  |
| print | p |  |
| main | m |  |
| if | s | условный оператор |
| else | e | условный оператор, альтернативный оператору if |
| ; | ; |  |
| , | , |  |
| { | { |  |
| } | } |  |
| ( | ( |  |
| ) | ) |  |
| + - \* & | | v | Арифметические и бинарные операции |
| ~ | ~ | Побитовая инверсия |
| = | = |  |
| > < | b |  |

Для распознавания вышеперечисленных цепочек используется механизм конечного автомата, в которых цепочки записаны в виде регулярных выражений. Пример записи регулярного выражения для идентификатора: [a-zA-Z\_][a-zA-Z0-9\_]\*.

Если цепочка успешно распознаётся, она записывается в таблицу лексем. В случае идентификаторов или литералов информация о них также заносится в таблицу идентификаторов. Пример реализованного конечного автомата для ключевого слова return языка RIV-2024 представлен в листинге 3.1.

|  |
| --- |
| #define FST\_FUNC FST::FST \_function(str,\  9,\  FST::NODE(1, FST::RELATION('f', 1)),\  FST::NODE(1, FST::RELATION('u', 2)),\  FST::NODE(1, FST::RELATION('n', 3)),\  FST::NODE(1, FST::RELATION('c', 4)),\  FST::NODE(1, FST::RELATION('t', 5)),\  FST::NODE(1, FST::RELATION('i', 6)),\  FST::NODE(1, FST::RELATION('o', 7)),\  FST::NODE(1, FST::RELATION('n', 8)),\  FST::NODE()\  ); |

Листинг 3.1 – Пример реализации графа конечного автомата RIV-2024

**3.5. Основные структуры данных**

Основными структурами данных, используемых для реализации лексического анализатора, являются таблица лексем и таблица идентификаторов.

В таблице лексем (LexTable) содержится номер строки исходного кода и лексемы, содержащиеся в этой строке. В таблице 3.3 представлены описание структуры, реализующей таблицу лексем.

Таблица 3.3 – Описание структуры таблицы лексем

|  |  |
| --- | --- |
| Поле | Назначение |
| maxsize | максимальное количество элементов в таблице |
| size | текущее количество элементов в таблице |
| table | массив элементов типа LT::Entry, являющихся  хранилищем для лексем |

Каждый элемент массива типа Entry, находящего в пространстве имен LT представляет отдельную лексему. В ней содержится информация о соответствующей лексем и ее номере в строке исходного кода. В таблице 3.4 представлено описание структуры LT::Entry.

Таблица 3.4 – Описание структуры Entry для LexTable

|  |  |
| --- | --- |
| Поле | Назначение |
| lexema[] | лексема |
| size | номера строка в исходном коде |
| idxTI | индекс в таблице идентификаторов. По умолчанию  устанавливается значение -1. |

Таблица идентификаторов (IdTable) имеет аналогичную структуру данных, как и у таблицы лексем. Отличием является массив Entry, находящийся уже в пространстве имен IT, который имеет другую реализацию для IdTable. Описание структуры IT::Entry представлено в таблице 3.5.

Таблица 3.5 – Описание структуры Entry для IdTable

|  |  |
| --- | --- |
| Поле | Назначение |
| idxfirstLE | индекс данного идентификатора в таблице лексем |
| line | номер текущей линии |
| id | имя идентификатора |
| iddatatype | тип данных языка RIV-2024 |
| idtype | Тип идентификатора. Принимает значения переменная, функция  или параметр |
| scope | Область видимости идентификатора |
| value | Объединение полей, используемых для хранения значений  литералов типа данных языка RIV-2024 |

Программная реализация структур LexTable и IdTable представлена в приложении Б.

**3.6. Структура и перечень сообщений лексического анализатора**

Сообщение об ошибке представляет собой уникальный номер, информацию о произошедшей ошибке, а также номер строки и столбца, где было обнаружена ошибка.

Перечень сообщений возможных ошибок лексического анализатора представлен в таблице 3.6.

Таблица 3.6 – Перечень сообщений лексического анализатора RIV-2024

|  |  |
| --- | --- |
| Номер | Сообщение |
| 113 | Недопустимый идентификатор |
| 114 | Дублирование идентификатора |
| 115 | Идентификатор не определён |
| 116 | Некорректное использование индекса |
| 117 | Недопустимый размер таблицы идентификаторов |
| 118 | Превышен размер таблицы идентификаторов |
| 119 | Превышена длина лексемы |
| 120 | Функция не может быть типа ch |
| 121 | Использование запрещенного символа |
| 122 | Кавычки не были закрыты |
| 123 | Литералы типа ch должен содержать только 1 символ |
| 124 | Превышен максимальный размер для литерала типа str(255) |

**3.7. Принцип обработки ошибок**

При обнаружении ошибки в исходном коде лексический анализатор немедленно прекращает выполнение, выводит сообщение об ошибке в консоль и записывает её в журнал.

Таким образом, за один запуск лексического анализа может быть выявлена только одна ошибка.

**3.8. Параметры лексического анализатора**

Входные параметры, использующиеся в работе лексического анализатора, их назначение и принцип применения описаны в таблице 3.7.

Таблица 3.7 – Входные параметры лексического анализатора RIV-2024

|  |  |  |
| --- | --- | --- |
| Параметр | Назначение | Принцип применения |
| Исходный код на языке RIV-2024 | Основной входной параметр, представляющий собой текст программы, подлежащий анализ | Построчно анализируется и разбивается на лексемы |
| Ключевые слова | Задает список зарезервированных слов, операторов и специальных символов, используемых в языке; нельзя использовать как идентификаторы. | Лексический анализатор использует ключевые слово для сравнения с обнаруженными токенами |
| Таблица допустимости входных символов | Определяет, какие символы считаются допустимыми в исходном коде. | Лексический анализатор различает символы, определенные в наборе, и игнорирует остальные |

**3.9. Алгоритм лексического анализа**

Ниже представлен алгоритм работы лексического анализатора RIV-2024.

1. Исходный код обрабатывается посимвольно, при этом каждый символ записывается в строковый буфер. Буфер наполняется до тех пор, пока не встретится символ-разделитель.
2. После накопления в строковом буфере, его содержимое передается на обработку конечным автоматом. Если автомат успешно разбирает буфер, формируется символьная лексема. В случае, если лексема является точкой входа в программу, литералом или идентификатором, процесс продолжается с пункта 3, иначе алгоритм возвращается к началу.
3. Тип данных идентификатора определяется на основе предыдущих лексем, которые обозначают тип идентификатора: параметр, переменная, функция
4. Если идентификатор только объявляется и явно указан его тип, в частично заполненной таблице идентификаторов выполняется поиск идентификатора с таким именем. Перед этим фиксируются тип и область видимости из стека. Если идентификатор найден, анализ завершается с выводом ошибки. В противном случае идентификатор добавляется в таблицу идентификаторов, а лексема идентификатора — в таблицу лексем с указанием индекса в таблице идентификаторов. Далее процесс возвращается к пункту 1.
5. При использовании ранее объявленного идентификатора программа выполняет его поиск в частично заполненной таблице. Если идентификатор отсутствует, процесс завершится с соответствующей ошибкой. Если найден, лексема получает индекс из таблицы идентификаторов и записывается в таблицу лексем. После этого процесс возвращается к пункту 1.
6. Если лексема является литералом, сначала определяется тип и значение литерала. Если литерал уже существует в таблице идентификаторов, новая запись не создается. В противном случае литерал добавляется в таблицу идентификаторов, а в таблицу лексем записывается ссылка на идентификатор. После этого процесс продолжается с пункта 1.
7. Если идентификатор является функцией, он записывается в таблицу идентификаторов с указанием типа возвращаемого значения. Все идентификаторы, перечисленные в круглых скобках, добавляются в качестве параметров функции. Для обозначения области видимости идентификаторов в стек помещается запись о функции, которая останется там до завершения ее объявления. После этого процесс возвращается к пункту 1.
8. Если исходный код еще не обработан полностью, алгоритм возвращается к пункту 1, иначе завершаем обработку.

**3.10. Контрольный пример**

Результат работы лексического анализатора, полученного при выполнении контрольного примера, представлен в приложении В.

**4. Разработка синтаксического анализатора**

**4.1. Структура синтаксического анализатора**

Синтаксический анализатор – вторая фаза работы компилятора, которая распознает синтаксические конструкции языка и формирует промежуточный код.

На вход подаются таблица лексем и таблица идентификаторов. В результате работы синтаксического анализатора формируется дерево разбора.

Параметрами, управляющими работой механизма, являются грамматика, описывающая синтаксис языка RIV-2024.

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

В синтаксическом анализаторе транслятора языка RIV-2024 используется контекстно-свободная грамматика G=⟨T,N,P,S⟩, где:

* T – множество терминальных символов (набор лексем, перечисленных в таблице 3.2);
* N – множество нетерминальных символов (первый столбец таблицы 4.1);
* P – множество правил языка (второй столбец таблицы 4.1);
* S – начальный символ грамматики, являющийся нетерминалом.

Контекстно-свободная грамматика G=⟨T,N,P,S⟩ имеет нормальную форму Грейбах, если она не леворекурсивная (не содержит леворекурсивных правил), а правила P имеют вид:

* A→aαA, где a∈T, α∈(N∪T)∗;
* S→λS, где S∈N – начальный символ, и если такое правило существует, то нетерминал не должен встречаться в правой части правил.

Полный перечень синтаксических правил в форме Грейбах перечислены в таблице 4.1.

Таблица 4.1 – Синтаксические правила языка RIV-2024

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочка | Описание |
| S | m{N} | Стартовый символ порождает всю структуру исходного кода. |
| tfi(N){N};S |
| m{N};s |
| N | dti; | Операторы программы |
| dti;N |
| dtN |
| dti=E;N |
| i=E; |
| i=E;N |
| rE; |
| rE;N |
| pE; |
| pE;N |

Окончание таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочка | Описание |
|  | s(N){N} |  |
| s(N){N}N |
| s(N){N}e{N} |
| s(N){N}N e{N}N |
| E | i | Выражения |
| l |
| iM |
| lM |
| (E) |
| i(W) |
| i() |
| (E)M |
| i(W)M |
| ivi |
| ~E |
| EvE |
| (E)vE |
| R(W) |
| M(W) |
| W | i | Подвыражения |
| l |
| i,W |
| l,W |
| F | ti | Параметры функции |
| ti,F |
| M | vE | Выражение (арифметические операции и операции сравнения) |
| vEM |
| (E)v |
| (E)v(E) |
| B | ibi | Операции сравнения |
| ibl |
| lbi |
| lbl |

**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 3.2 и 4.1 соответственно |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $) |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1 |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
|  | Начальное состояние магазина автомата | Символ маркера дна стека $ |
|  | Множество конечных состояний | Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты |

**4.4. Основные структуры данных**

В таблице 4.3 представлено описание структур, характеризующих контекстно-свободную грамматику.

Таблица 4.3 – Описание структуры данных синтаксического анализатора

|  |  |
| --- | --- |
| Структура | Описание |
| Greibach | Все символы (алфавит) грамматики представляются в формате GRBALPHABET (short). Причем терминалы – положительные значения, нетерминалы – отрицательные значения.  Структура включает: |

Окончание таблицы 4.3

|  |  |
| --- | --- |
| Структура | Описание |
|  | * множество правил: переменная rules (типа структура Rule); * количество правил: переменная size (short); * стартовый символ грамматики: startN (GRBALPHABET); * служебный символ (дно стека и последняя лексема таблице лексем): stbottomT (GRBALPHABET); * два конструктора; * методы getRule: позволяет получить номер правила или -1 (к точке возврата) и правило (второй параметр типа Rule) по левому символу правила (первый параметр); позволяет получить правило (возвращает к точке вызова параметр типа Rule) по его номеру. |
| Rule | Структура: представление одного правила, имеющего вид:  A→xxx|yyy|… .  Структура включает:   * нетерминал – левый символ правила: nn (GRBALPHABET); * идентификатор ошибки, связанной с правилом: iderror (int) – код ошибки в подсистеме Error; * количество цепочек в правой стороне правила: size (short); * цепочки-правые стороны правила: chains (типа Rule::Chain); * два конструктора; * метод getCRule: позволяет получить правило в виде строки вида N->цепочка (в символьном ASCII-виде, для отображения); * метод getNextChain: позволяет найти следующую за заданным   номером (3й параметр j типа short)цепочку (параметр pchain типа Rule::Chain) и ее номер (к точке возврата типа short). |
| Rule::Chain | Структура: представление цепочки – правой стороны правила.  Структура включает:   * размер цепочки: size (short) в символах; * цепочка: nt (GRBALPHABET); * два конструктора; * метод getCChain: позволяет получить строку-цепочку в * символьном виде для отображения; * методы T и N: преобразовывают ASCI-символы в GRBALPHABET-символы (терминалы и нетерминалы); * методы isT и isN: проверяют является GRBALPHABET-символ терминалом или нетерминалом; * метод aplphabet\_to\_char: преобразует заданный (параметр) GRBALPHABET-символ в ASCII-символ. |

Реализация структур данных семантического анализа на языке C++ представлена в приложении Г.

**4.5. Описание алгоритма синтаксического разбора**

**Алгоритм работы синтаксического разбора описан ниже.**

1. Первым делом инициализируется магазин, в который помещается стартовый символ.
2. Формируется входная лента по полученной ранее таблице лексем.
3. Встретившийся нетерминальный символ раскрывается в соответствии с правилами, и реверс полученной цепочки записывается в магазин.
4. Если терминалы на вершине стека и в ленте совпадают, то лента продвигается, а терминал удаляется с вершины стека. Иначе происходит возврат к предыдущему сохраненному состоянию и выбирается другое правило.
5. Если в магазине встретился нетерминал, то осуществляется переход в пункт 3.
6. Если достигнуто дно стека и при этом входная цепочка пуста, то значит синтаксический анализ выполнен успешно. Если нет, то генерируется исключение.

**4.6. Структура и перечень сообщений синтаксического анализатора**

**Перечень возможных ошибок при работе синтаксического анализатора представлен в таблице 4.4.**

Таблица 4.4 – Перечень сообщений синтаксического анализатора

|  |  |
| --- | --- |
| **Код** | **Сообщение** |
| **600** | **Неверная структура программы** |
| **601** | **Ошибочный оператор** |
| **602** | **Ошибка в выражении** |
| **603** | **Ошибка в подвыражении** |
| **604** | **Ошибка в параметрах функции** |
| **605** | **Ошибка в арифметических операторах** |
| **606** | **Ошибка в операторах сравнения** |

**4.7. Параметры синтаксического анализатора и режимы его работы**

**На вход синтаксического анализатора подаются таблица лексем и таблица идентификаторов. Кроме этого, используется описание грамматики в форме Грейбах. После завершения работы синтаксического разбора получается дерево разбора и** в журнал протокола записывается результат работы автомата с магазинной памятью.

**4.8. Принцип обработки ошибок**

**Ниже представлен принцип обработки возможных ошибок синтаксического анализатора.**

1. Синтаксический анализатор последовательно проверяет все правила и их цепочки в грамматике, чтобы найти подходящее соответствие конструкции, представленной в таблице лексем.
2. Если подходящая цепочка не найдена, формируется ошибка, соответствующая данной ситуации.
3. Все обнаруженные ошибки фиксируются в общей структуре, предназначенной для хранения ошибок.
4. После завершения процесса трассировки в протокол выводится диагностическое сообщение о найденных ошибках.

**4.9. Контрольный пример**

**Результат работы протокола синтаксического разбора и дерево разбора представлены в приложении Д.**

**5. Разработка семантического анализатора**

**5.1. Структура семантического анализатора**

Семантический анализ – третья фаза работы транслятора, предназначенный для смысловой проверки исходного текста на соответствие семантическим правилам языка.

На вход принимает использует синтаксическое дерево и информацию из таблицы идентификаторов. Результат работы семантического анализатора записывается в файл протокола. Схема структура семантического анализатора представлена на рисунке 5.1.

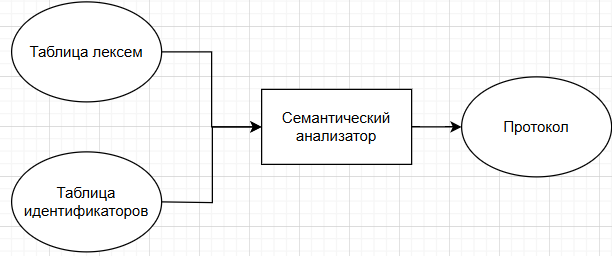


Рисунок 5.1 – Структура семантического анализатора

**5.2. Функции семантического анализатора**

Полный перечень семантических проверок с указанием их фаз приведён ниже в таблице 5.1.

Таблица 5.1 – Перечень семантических проверок

|  |  |  |
| --- | --- | --- |
| № | Правило | Фаза |
| 1 | Наличие main | Синтаксический |
| 2 | Дублирование main | Лексический |
| 3 | Идентификаторы усекаются до 9 символов | Лексический |
| 4 | Все идентификаторы должны быть обвялены в той же области видимости перед использованием | Лексический |
| 5 | Не допускается совпадения названий идентификаторов с ключевыми словами | Лексический |
| 6 | Соответствие типов формальных и фактических параметров при вызове функций | Семантический |
| 7 | Соответствие типов формальных и фактических параметров при вызове операторов | Семантический |

Окончание таблицы 5.1

|  |  |  |
| --- | --- | --- |
| № | Правило | Фаза |
| 8 | Инструкция if и else должны содержать логический оператор | Семантический |
| 9 | Все переменные должны инициализироваться | Лексический |
| 10 | Возвращаемое значение функции должно совпадать с типом функции | Семантический |

Семантический анализатор имеет набор функций, реализующий проверку некоторых семантических правил. На вход подается таблица лексем и таблица идентификаторов, и позиция, с которой начинается проверка. Перечень функций с их описанием представлен в таблице 5.2.

Таблица 5.2 – Перечень функции семантического анализатора

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| **functions(lex);** | **Обработка функций, их параметров и возвращаемого значения** |
| **literals(lex);** | **Проверка типа литералов** |
| **operands(lex);** | **Проверка типа операндов** |

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

При работе семантического анализатора могут возникать различные ошибки.

Перечень кодов ошибок и их сообщений представлен в таблице 5.3.

Таблица 5.3 – Перечень ошибок семантического анализа

|  |  |
| --- | --- |
| Код | Сообщение |
| 700 | Инструкция if и else должны содержать логический оператор |
| 701 | Все переменные должны инициализироваться |
| 702 | Возвращаемое значение функции должно совпадать с типом функции |
| 703 | Типы данных, используемые в выражении, не совпадают |
| 704 | Попытка вызвать операцию, неподдерживаемую типом данных |
| 705 | Невозможно использовать идентификатор функции кроме как для её вызова |
| 706 | Попытка присвоить идентификатору значение функции, которое отлично от типа данных идентификатора |
| 707 | Попытка присвоить идентификатору значение функции, которое отлично от типа данных идентификатора |
| 708 | Функции стандартной библиотеки должны принимать два параметра типа данных byte |
| 709 | print обязательно должно принимать какое-либо значение |

**5.4. Принцип обработки ошибок**

Ошибки, возникающие в процессе семантического анализа, фиксируются в протокол, заданный входным параметром. При выявлении ошибок, программа немедленно завершает работу транслятора.

Таким образом, за один запуск семантического анализа может быть выявлена только одна ошибка.

**5.5. Контрольный пример**

Перечень контрольных примеров, демонстрирующих работу семантического анализатора, представлен в таблице 5.4.

Таблица 5.4 – Перечень контрольных примеров

|  |  |
| --- | --- |
| Код | Сообщение |
| byte function getInv(byte as) {  return ~as;  }  main {  getInv = 3;  } | Ошибка 706: Невозможно использовать идентификатор функции кроме как для её вызова  Строка: 7, позиция: 0 |
| main {  let byte bb = rest('l', 'r');  } | Ошибка 708: Функции стандартной библиотеки должны принимать два параметра типа данных byte  Строка: 3, позиция: 2 |

**6. Вычисление выражений**

**6.1. Выражения, допускаемые языком**

В языке RIV-2024 допускаются выражения, применяемые к целочисленным типам данных, с поддержкой вызова функций. Приоритет операция представлен в таблице 6.1.

Таблица 6.1. – Приоритеты операций

|  |  |
| --- | --- |
| Операция | Приоритет |
| ( ) | 0 |
| ~ | 1 |
| \* | 2 |
| +, - | 3 |
| >=, <=, >, < | 4 |
| !=, == | 5 |
| & | 6 |
| | | 7 |
| , | 8 |
| Примечание – операции с приоритетом «0» имеют высший приоритет | |

**6.2. Польская запись и принцип ее построения**

Польская запись – это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок.

Обратная польская нотация - это форма записи математических выражений, в которой операторы расположены после своих операндов.

Принципы построения польской записи:

* элементы исходного выражения читаются слева направо;
* механизм поочередно извлекает лексему из выражения;
* если текущий символ – идентификатор или литерал, то записываем его в результирующую строку. При этом, если идентификатор является функцией, записывается специальный токен «@» и количество передаваемых параметров;
* открывающаяся скобка автоматически заносится в стек операций;
* закрывающаяся скобка выталкивает все операции из стека в финальную;
* строку и самоуничтожается вместе с открывающейся скобкой;
* операция выталкивает все операции с большим приоритетом в результирующую строку;
* когда встречается символ конца строки кода («;») – все операции выталкиваются из стека в финальную строку;
* элементы таблицы лексем, которые остались свободными, заполняются символом-заполнителем «#».

**6.3. Программная реализация обработки выражений**

Программная реализация преобразования выражений в обратную польскую запись представлена в приложении Е.

**6.4. Контрольный пример**

В таблице 6.2 представлен результат преобразования выражений в обратную польскую запись.

Таблица 6.2 – Преобразование выражений в ПОЛИЗ

|  |  |
| --- | --- |
| Выражение | Обратная польская запись для выражения |
| dti=(ivi)vi | dti=iiviv### |
| dti=(lvl)lv(lvl); | dti=llvlllvv####; |
| dti=(((ivi)vl)vl)v(lv(lvl)); | dti=iivlvlvlllvvv##########; |

**7. Генерация кода**

**7.1. Структура генератора кода**

Целевым языком для трансляции исходного кода был выбран Microsoft Macro Assembler.

Генерация основывается на том, что каждый оператор представляет собой заранее заданный набор команд, который будет генерироваться при каждом использовании этого оператора. Функции на исходном языке программирования RIV-2024 транслируются в процедуры на ассемблере, возврат значений происходит через регистры и стек. При генерации условного оператора используются метки и переходы по ним.

Схема структуры генератора кода RIV-2024 представлена на рисунке 7.1.

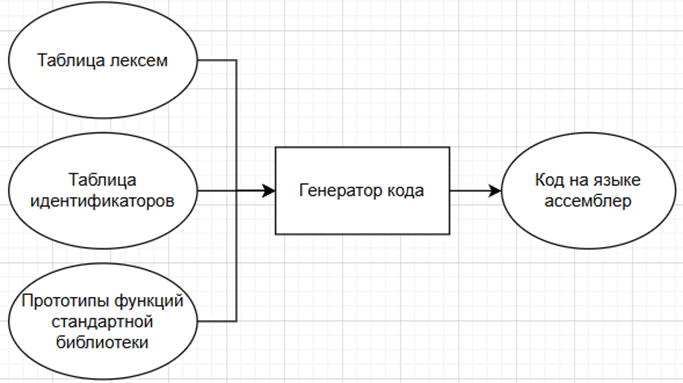


Рисунок 7.1 – Структура генератора кода

**7.2. Представление типов данных в оперативной памяти**

Элементы таблицы идентификаторов расположены сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке RIV-2024 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка и ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке RIV-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| byte | sbyte | Хранит целочисленный тип данных размером один байт |
| ch | byte | Хранит символьный тип данных размером один байт |
| bool | byte | Хранит логическое значение true, false размером 1 байт |

Окончание таблицы 7.1

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке RIV-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| int | dword | Хранит целочисленный тип данных размером 4 байта |
| str | db | Каждый символ строки типа str хранится в поле размером 1 байт |

**7.3. Статическая библиотека**

В языке RIV-2024 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически. Перечень функций стандартной библиотеки представлен в таблице 7.2

Таблица 7.2 – Функции статических библиотек

|  |  |
| --- | --- |
| Функция | Назначение |
| int module(int num1) | Вычисление модуля числа |
| int rest(int num1, int num2) | Вычисление остатка от деления числа num1 на число num2 |
| void writestr | Вывод в консоль строковой переменной |
| void writechar | Вывод в консоль символьной переменной |
| void writeint | Вывод в консоль целочисленной переменной |
| void writebool | Вывод в консоль логической переменной: в случае значения 0 – false, в остальных случаях – true |

**7.4. Особенности алгоритма генерации кода**

В языке RIV-2024 генерация кода строится на основе таблиц лексем и идентификаторов.

Генерация исходного кода в ассемблере осуществляется с использованием заранее подготовленных функций и макросов, что позволяет обеспечить структурированность, оптимизацию и надежность результирующего кода. Заранее подготовленные функции реализуют стандартные операции, такие как вывод данных, арифметические и логические вычисления, механизм условного оператора. Эти функции включаются в объектный код через вызовы, что упрощает процесс генерации и снижает вероятность ошибок.

Общая схема работы генератора кода представлена на рисунке 7.2.

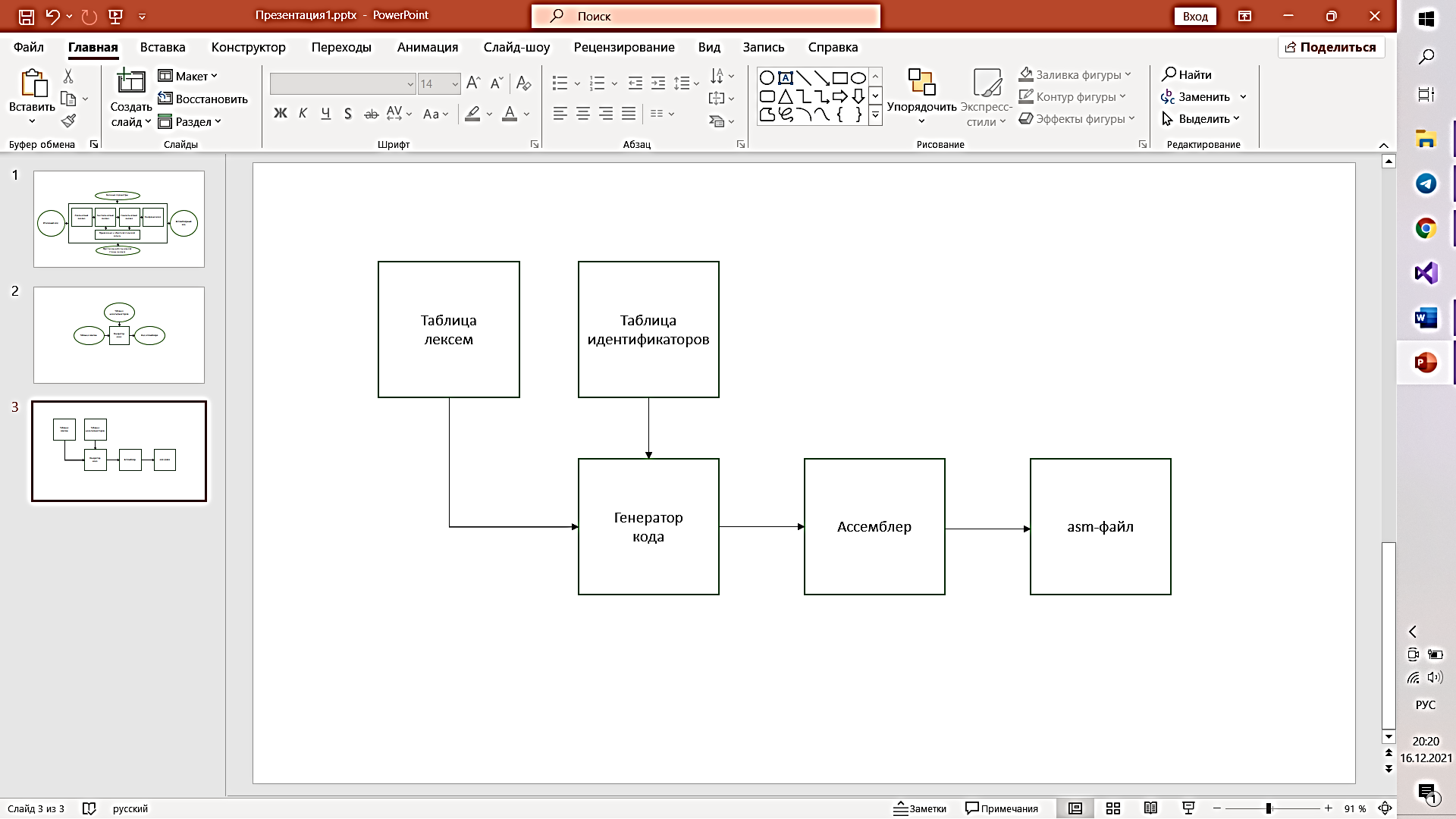


Рисунок 7.2 – Структура генератор кода

**7.5. Входные параметры, управляющие генерацией кода**

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке RIV-2024. Результаты работы генератора кода выводятся в файл с расширением .asm.

**7.6. Контрольный пример**

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

**8. Тестирование транслятора**

**8.1. Общие положения**

Тесты используются для проверки транслятора на работоспособность и для выявления недочетов и ошибок, а также последующего их исправления.

При возникновении ошибки транслятор немедленно завершает свою работу и выводит сообщение об ошибке с соответствующим ей кодом в консоль и в файл протокола .log.

**8.2 Результаты тестирования**

В таблице 8.1 приведены ошибки, генерируемые в процессе считывания входного файла, а также в процессе лексического, синтаксического и семантического анализов.

Таблица 8.1 – Результаты тестирования транслятора

|  |  |  |
| --- | --- | --- |
| Исходный код | Диагностическое сообщение | Этап |
| byte function test(byte a) {  return a;  }  byte function test(byte b) {  return b;  }  main {  …  } | Ошибка 114: Дублирование идентификатора  Строка: 5, позиция: 19 | Лексический |
| main {  let byte flag = 3;  let byte flag = 4;  } | Ошибка 114: Дублирование идентификатора  Строка: 3, позиция: 14 | Лексический |
| main {  print(flag);  } | Ошибка 115: Идентификатор не определён  Строка: 2, позиция: 11 | Лексический |
| main {  …  }  main {  …  } | Ошибка 116: Дублирование main | Лексический |

Продолжение таблицы 8.1

|  |  |  |
| --- | --- | --- |
| Исходный код | Диагностическое сообщение | Этап |
| ch function test(char a) {  return a;  }  main {  …  } | Ошибка 120: Функция не может быть типа char  Строка: 1, позиция: 17 | Лексический |
| let byte flag = 2;  print(flag); | 600: строка 1, Неверная структура программы | Синтаксический |

**Заключение**

В ходе выполнения курсового проекта была достигнута основная цель – разработан собственный язык программирования RIV-2024 и компилятор для его обработки. Выполненные задачи позволили изучить основные аспекты создания языка программирования и транслятора, включая определение алфавита, синтаксиса и семантики языка, а также реализацию основных этапов трансляции.

В рамках проекта были реализованы следующие ключевые этапы:

* разработана спецификация языка RIV-2024;
* реализованы лексический, синтаксический и семантический анализаторы, обеспечивающие обработку исходного текста программы.
* разработан процесс генерации объектного кода, демонстрирующий функциональные возможности языка.
* создан и протестирован контрольный пример программы, написанной на языке RIV-2024, для проверки работоспособности всех компонентов транслятора.

Окончательная версия языка RIV-2024 включает:

* 5 типов данных;
* поддержка оператора вывода строки;
* возможность вызова функций стандартной библиотеки;
* наличие 3 арифметических и 3 логических операторов для вычисления выражений;
* поддержка функций и 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. – 768 с.

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

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

4. Ирвин, К. Язык ассемблера для процессоров Intel, 4-е издание.: Пер. с англ. – М.: Вильямс, 2005. – 912с.

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

# **Приложение А**

|  |
| --- |
| byte function getAnd(byte a, byte b) {  return a&b;  }  int function getMul(int a, int b) {  return a\*b;  }  main {  let byte yesor = 3 | 5;  let byte inversion = ~4;  let byte yesand = getAnd(4, 5);  print(yesor);  print(inversion);  print(yesand);  let byte rv = rest(yesor, yesand);  let byte mv = module(inversion);  print(rv);  print(mv);  let int iv = getMul(155, 198);  print(iv);  let ch symbol = 'I';  let str stroka = "dont mind";  print(symbol);  print("Вывод строки");  let bool flag = 0;  if (inversion > 2) {  flag = 1;  print(flag);  }  else {  print(flag);  }  print(yesor);  } |

Листинг А.1 - Исходный код на языке RIV-2024

# **Приложение Б**

|  |
| --- |
| struct Entry  {  char lexema[LEXEMA\_FIXSIZE];  int sn;  int idxTI;  };  struct lexTable  {  int maxsize;  int size;  Entry\* table;  }; |

Листинг Б.1 – Реализация таблицы лексем

|  |
| --- |
| enum IDDATATYPE { INT = 1, STR = 2, BYTE = 3, CH = 4, BOOL = 5};  //V - операция, F - функция, P - параметр, L - лексема, SF - стандартная библиотека  enum IDTYPE { V = 1, F = 2, P = 3, L = 4, SF = 5, M = 6};  struct Entry // строка таблицы идентификаторов  {  int idxfirstLE; // индекс первой строки в таблице лексем  int line; //текущая линия  char id[ID\_MAXSIZE]; // идентификатор (автоматически усекается до ID\_MAXSIZE)  IDDATATYPE iddatatype; // тип данных  IDTYPE idtype; // тип идентификатора (переменная, функция, параметр, локальная переменная)  Entry\* scope;  union  {  int vint; // значение integer  char vbyte; // значения byte  char vchar; //значения char  bool vbool;  struct  {  int len; // кол-во символов в string  char str[TI\_STR\_MAXSIZE - 1]; // символы string  } vstr[TI\_STR\_MAXSIZE]; // значение string  } value; // значение идентификатора  };  struct idTable // экземпляр таблицы идентификаторов  {  int maxsize; // емкость таблицы идентификаторов < TI\_MAXSIZE  int size; // текущий размер таблицы идентификаторов < maxsize  Entry\* table; // массив строк таблицы идентификаторов  }; |

Листинг Б.1 – Реализация таблицы идентификаторов

# **Приложение В**

|  |
| --- |
| 1 tfi(ti,ti){  2 rivi;  3 }  5 tfi(ti,ti){  6 rivi;  7 }  9 m{  11 dti=lvl;  12 dti=~l;  13 dti=i(l,l);  15 p(i);  16 p(i);  17 p(i);  19 dti=R(i,i);  20 dti=M(i);  22 p(i);  23 p(i);  25 dti=i(l,l);  26 p(i);  27 dti=l;  28 dti=l;  29 p(i);  30 p(l);  32 dti=l;  34 s(i>l){  35 i=l;  36 p(i);  37 }  38 e{  39 p(i);  40 }  42 p(i);  44 } |

Листинг В.1 –Таблица лексем

|  |
| --- |
| id datatype idtype Line Scope value  getAnd\_1 BYTE F 1 - 0  a\_1 BYTE P 1 getAnd 0  b\_1 BYTE P 1 getAnd 0  getMul\_5 INT F 5 - 0  a\_5 INT P 5 getMul 0  b\_5 INT P 5 getMul 0  main\_9 INT M 9 - 0  yesor\_11 BYTE V 11 main 0  L0\_11 BYTE L 11 - 3  L1\_11 BYTE L 11 - 5  inversion\_12 BYTE V 12 main 0  L2\_12 BYTE L 12 - 4  yesand\_13 BYTE V 13 main 0  L3\_13 BYTE L 13 - 4  L4\_13 BYTE L 13 - 5  rv\_19 BYTE V 19 main 0  rest\_19 BYTE SF 19 main 0  mv\_20 BYTE V 20 main 0  module\_20 BYTE SF 20 main 0  iv\_25 INT V 25 main 0  L5\_25 INT L 25 - 155  L6\_25 INT L 25 - 198  symbol\_27 CH V 27 main ' '  C8\_27 CH L 27 main 'I'  stroka\_28 STR V 28 main  L9\_28 STR L 28 main "dont mind"  L11\_30 STR L 30 main "Вывод строки"  flag\_32 BOOl V 32 main 0  L12\_32 BYTE L 32 - 0  L13\_34 BYTE L 34 - 2  L14\_35 BYTE L 35 - 1 |

Листинг В.2 –Таблица идентификаторов

# **Приложение Г**

|  |
| --- |
| struct Rule //правило в грамматике Грейбах  {  GRBALPHABET nn; //нетерминалы (левый символ правила) < 0  int iderror; //идентификатор диагностического сообщения  short size; //количество цепочек - правых частей правила  struct Chain //цепочка(правая часть правила)  {  short size; //длина цепочки  GRBALPHABET\* nt; //цепочка терминалов (>0) и нетермеминалов (<0)  Chain()  {  this->size = 0; //количество символов в цепочке  this->nt = 0; //символы (терминал или нетерминал)  };  Chain(short psize, GRBALPHABET s, ...); //кол-во символов в цепочке/символы(терминалы и нетерминалы)  char\* getCChain(char\* b); //получить правую сторону правила  static GRBALPHABET T(char t) { return GRBALPHABET(t); }; //терминал  static GRBALPHABET N(char n) { return -GRBALPHABET(n); }; //не терминал  static bool isT(GRBALPHABET s) { return s > 0; }; //терминал?  static bool isN(GRBALPHABET s) { return !isT(s); }; //не терминал?  static char alphabet\_to\_char(GRBALPHABET s) //GRBALPHABET->char (обращение по указателю к полю символа)  {  return isT(s) ? char(s) : char(-s);  };  }\*chains; //массив цепочек - правых частей правила  Rule()  {  this->nn = 0x00;  this->size = 0;  }  Rule(GRBALPHABET pnn, //(нетерминал(< 0);  int iderroe, //идентификатор диагностического сообщ - я  short psize, //кол - во цепочек - правых частей правила  Chain c, ... //множество цепочек - правых частей правила)  );  char\* getCRule(char\* b, short nchain); //получить парвило в виде N->цепочка (для распечатки) (буфер;номер цепочки(правой части) в правиле)  short getNextChain(GRBALPHABET t, Rule::Chain& pchain, short j); //получить следующую за j подходящую цепочку, вернуть её номер или -1 (первый символ цепочки;возвращаемая цепочка; номер цепочки)  }; |

Листинг Г.1 – Реализация структур Rule и Rule::Chain

|  |
| --- |
| struct Greibach //грамматика Грейбах  {  short size; //количество правил  GRBALPHABET startN; //стартовый символ  GRBALPHABET stbottomT; //дно стека  Rule\* rules; //множество правил  Greibach() { this->size = 0; this->startN = 0; this->stbottomT = 0; this->rules = 0; };  Greibach(GRBALPHABET pstartN, GRBALPHABET pstbottomT, short psize, Rule r, ...);  short getRule(GRBALPHABET pnn, Rule& prule);  Rule getRule(short n);  }; |

Листинг Г.2 – Реализация структуры Greibach

# **Приложение Д**

|  |
| --- |
| 0 всего строк 44, синтаксический анализ выполнен без ошибок  0 : S->tfi(F){N}S  4 : F->ti,F  7 : F->ti  11 : N->rE;  12 : E->iM  13 : M->vE  14 : E->i  17 : S->tfi(F){N}S  21 : F->ti,F  24 : F->ti  28 : N->rE;  29 : E->iM  30 : M->vE  31 : E->i  34 : S->m{N}  36 : N->dtN  38 : N->i=E;N  40 : E->lM  41 : M->vE  42 : E->l  44 : N->dtN  46 : N->i=E;N  48 : E->~E  49 : E->l  51 : N->dtN  53 : N->i=E;N  55 : E->i(W)  57 : W->l,W  59 : W->l  62 : N->pE;N  63 : E->(E)  64 : E->i  67 : N->pE;N  68 : E->(E)  69 : E->i  72 : N->pE;N  73 : E->(E)  74 : E->i  77 : N->dtN  79 : N->i=E;N  81 : E->R(W)  83 : W->i,W  85 : W->i  88 : N->dtN  90 : N->i=E;N  92 : E->M(W)  94 : W->i  97 : N->pE;N  98 : E->(E)  99 : E->i  102 : N->pE;N  103 : E->(E)  104 : E->i  107 : N->dtN  109 : N->i=E;N  111 : E->i(W)  113 : W->l,W  115 : W->l  118 : N->pE;N  119 : E->(E)  120 : E->i  123 : N->dtN  125 : N->i=E;N  127 : E->l  129 : N->dtN  131 : N->i=E;N  133 : E->l  135 : N->pE;N  136 : E->(E)  137 : E->i  140 : N->pE;N  141 : E->(E)  142 : E->l  145 : N->dtN  147 : N->i=E;N  149 : E->l  151 : N->s(E){N}e{N}N  153 : E->iM  154 : M->>E  155 : E->l  158 : N->i=E;N  160 : E->l  162 : N->pE;  163 : E->(E)  164 : E->i  170 : N->pE;  171 : E->(E)  172 : E->i  176 : N->pE;  177 : E->(E)  178 : E->i |

Листинг Д.1 – Дерево разбора

# **Приложение Е**

|  |
| --- |
| #include "PolishNotation.h"  map<char, int>Priorities = {  {'(',0},  {')',0},  {'~',1},  {'\*',2},  {'/',2},  {'+',3},  {'-',3},  {'>',4},  {'<',4},  {'&',6},  {'|',7},  {',',8},  };  namespace PN {  bool polishNotation(int i, LA::LEX& lex)  {  std::stack<LT::Entry> stack;  std::queue<LT::Entry> queue;  LT::Entry aggregate\_symbol;  aggregate\_symbol.idxTI = -1;  aggregate\_symbol.lexema[0] = '#';  aggregate\_symbol.sn = lex.lexTable.table[i].sn;  LT::Entry function\_symbol;  function\_symbol.idxTI = LT\_TI\_NULLIDX;  function\_symbol.lexema[0] = '@';  function\_symbol.sn = lex.lexTable.table[i].sn;  int idx;  int lexem\_counter = 0;  int parm\_counter = 0;  int lexem\_position = i;  char\* buf = new char[i];  bool is\_function = false;  for (i; lex.lexTable.table[i].lexema[0] != LEX\_SEMICOLON; i++, lexem\_counter++) {  switch (lex.lexTable.table[i].lexema[0]) {  case LEX\_ID:  case LEX\_LITERAL:  if (lex.idTable.table[lex.lexTable.table[i].idxTI].idtype == IT::F) {  is\_function = true;  idx = lex.lexTable.table[i].idxTI;  }  else {  if (is\_function)  parm\_counter++;  queue.push(lex.lexTable.table[i]);  }  continue;  case TILDE:  case LEFTTHESIS:  stack.push(lex.lexTable.table[i]);  continue;  case RIGHTTHESIS:  while (stack.top().lexema[0] != LEFTTHESIS) {  queue.push(stack.top());  stack.pop();  if (stack.empty())  return false;  }  if (!is\_function)  stack.pop();  else {  function\_symbol.idxTI = idx;  idx = LT\_TI\_NULLIDX;  lex.lexTable.table[i] = function\_symbol;  queue.push(lex.lexTable.table[i]);  \_itoa\_s(parm\_counter, buf, 2, 10);  stack.top().lexema[0] = buf[0];  stack.top().idxTI = LT\_TI\_NULLIDX;  stack.top().sn = function\_symbol.sn;  queue.push(stack.top());  stack.pop();  parm\_counter = 0;  is\_function = false;  }  continue;  case LEX\_OPERATION:  while (!stack.empty() && Priorities[lex.lexTable.table[i].lexema[0]] > Priorities[stack.top().lexema[0]]) { //!!!!!!!!!!!!!!  queue.push(stack.top());  stack.pop();  }  stack.push(lex.lexTable.table[i]);  continue;    case LEX\_REST:  case LEX\_MODULE:  is\_function = true;  idx = lex.lexTable.table[i].idxTI;  continue;  }  }  while (!stack.empty()) {  if (stack.top().lexema[0] == LEFTTHESIS || stack.top().lexema[0] == RIGHTTHESIS)  return false;  queue.push(stack.top());  stack.pop();  }  while (lexem\_counter != 0) {  if (!queue.empty()) {  lex.lexTable.table[lexem\_position++] = queue.front();  queue.pop();  }  else  lex.lexTable.table[lexem\_position++] = aggregate\_symbol;  lexem\_counter--;  }  for (int i = 0; i < lexem\_position; i++) {  if (lex.lexTable.table[i].lexema[0] == LEX\_OPERATION || lex.lexTable.table[i].lexema[0] == LEX\_LITERAL)  lex.idTable.table[lex.lexTable.table[i].idxTI].idxfirstLE = i;  }  return true;  }  bool startPolish(LA::LEX& lex)  {  bool result = false;  for (int i = 0; i < lex.lexTable.size; i++) {  if (lex.lexTable.table[i].lexema[0] == '=') {  result = polishNotation(i + 1, lex);  if (!result) {  return false;  }  }  }  return true;  }  } |

Листинг Е.1 – Реализация преобразования в ПОЛИЗ

# **Приложение Ж**

|  |
| --- |
| .586P  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib ../Debug/LIB.lib  ExitProcess PROTO : DWORD  SetConsoleCP PROTO : DWORD  SetConsoleOutputCP PROTO : DWORD  module PROTO : BYTE  rest PROTO : BYTE, : BYTE  writestr PROTO : DWORD  writeint PROTO : BYTE  writebool PROTO : BYTE  writechar PROTO : BYTE  .stack 4096  .const  L0 SBYTE 3 ; byte  L1 SBYTE 5 ; byte  L2 SBYTE 4 ; byte  L3 SBYTE 4 ; byte  L4 SBYTE 5 ; byte  L5 DWORD 155 ; integer (4 bytes)  L6 DWORD 198 ; integer (4 bytes)  C8 BYTE 73 ; symbol  L9 DB "dont mind", 0 ; text  L11 DB "Вывод строки", 0 ; text  L12 SBYTE 0 ; byte  L13 SBYTE 2 ; byte  L14 SBYTE 1 ; byte  .data  yesor SBYTE 0 ; byte  inversion SBYTE 0 ; byte  yesand SBYTE 0 ; byte  rv SBYTE 0 ; byte  mv SBYTE 0 ; byte  iv SDWORD 0 ; int  symbol BYTE 0 ; ch  stroka DWORD 0 ; str  flag BYTE 0 ; boolean  .code  FgetAnd PROC uses ebx ecx edi esi, a : SBYTE, b : SBYTE  ; return  movsx eax, a  mov a, al  ; Logical AND  mov al, b  and al, a  mov b, al  ret  FgetAnd ENDP  FgetMul PROC uses ebx ecx edi esi, a : DWORD, b : DWORD  ; return  mov eax, a  ; Multiplication  mov eax, b  imul eax, a  mov b, eax  ret  FgetMul ENDP  main PROC  Invoke SetConsoleCP, 1251  Invoke SetConsoleOutputCP, 1251  ; string #11 : i=llv  mov al, L0  mov yesor, al  ; Logical OR  mov al, L0  or al, L1  mov yesor, al  ; string #12 : i=l~  mov al, L2  mov inversion, al  ; Logical NOT  mov al, L2  not al  mov inversion, al  ; string #13 : i=ll@2##  movsx eax, L4  push eax  movsx eax, L3  push eax  CALL FgetAnd  mov yesand, al  movsx eax, yesor  push eax  CALL writeint  movsx eax, inversion  push eax  CALL writeint  movsx eax, yesand  push eax  CALL writeint  ; string #19 : i=ii@2##  movsx eax, yesand  push eax  movsx eax, yesor  push eax  CALL rest  mov rv, al  ; string #20 : i=i@1#  movsx eax, inversion  push eax  CALL module  mov mv, al  movsx eax, rv  push eax  CALL writeint  movsx eax, mv  push eax  CALL writeint  ; string #25 : i=ll@2##  mov eax, L6  push eax  mov eax, L5  push eax  CALL FgetMul  mov iv, eax  mov eax, iv  push eax  CALL writeint  ; string #27 : i=l  mov al, C8  mov symbol, al  ; string #28 : i=l  push offset L9  pop eax  mov stroka, eax  push eax  movzx eax, symbol  push eax  CALL writechar  pop eax  push offset L11  CALL writestr  ; string #32 : i=l  mov al, L12  mov flag, al  If\_Begin1:  movzx eax, inversion  movzx ebx, L13  cmp eax, ebx  jg If\_End1  jmp If\_Else  If\_End1:  ; string #35 : i=l  mov al, L14  mov flag, al  push eax  movzx eax, flag  push eax  CALL writebool  pop eax  jmp If\_End2  If\_Else:  push eax  movzx eax, flag  push eax  CALL writebool  pop eax  If\_End2:  movsx eax, yesor  push eax  CALL writeint  push -1  call ExitProcess  main ENDP  end main |

Листинг Е.1 –Результат генерации исходного кода в ассемблер