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 ARV-2024»

Выполнил студент Ананьев Роман Васильевич  (Ф.И.О)

Руководитель проекта Волчек Дарья Ивановна

Заведующий кафедрой к.т.н., доц. Смелов В.В.

Консультанты Волчек Дарья Ивановна

Нормоконтролер Волчек Дарья Ивановна

Курсовой проект защищен с оценкой

**Содержание**

[Введение 5](#_Toc1666)

[Спецификация языка программирования 6](#_Toc7035)

[1.1 Характеристика языка программирования 6](#_Toc22884)

[1.2 Применяемые сепараторы 6](#_Toc27676)

[1.3 Применяемые кодировки 6](#_Toc17866)

[1. 4 Типы данных 7](#_Toc23988)

[1.5 Преобразование типов данных 8](#_Toc22471)

[1.6 Идентификаторы 8](#_Toc20997)

[1.7 Литералы 9](#_Toc24596)

[1.8 Объявление данных 9](#_Toc23806)

[1.9 Инициализация данных 9](#_Toc22701)

[1.10 Инструкции языка 10](#_Toc5961)

[1.11 Операции языка 11](#_Toc27565)

[1. 12 Выражения и их вычисления 11](#_Toc3654)

[1.13 Конструкции языка 12](#_Toc5598)

[1.14 Область видимости идентификаторов 12](#_Toc1244)

[1.15 Семантические проверки 12](#_Toc27336)

[1.16 Распределение оперативной памяти на этапе выполнения 13](#_Toc5056)

[1.17 Стандартная библиотека и её состав 13](#_Toc28238)

[1.18 Ввод и вывод данных 14](#_Toc27953)

[1.19 Точка входа 14](#_Toc22550)

[1.20 Препроцессор 14](#_Toc26412)

[1.21 Соглашения о вызовах 14](#_Toc30543)

[1.22 Объектный код 14](#_Toc18597)

[1.23 Классификация сообщений транслятора 14](#_Toc24982)

[1.24 Контрольный пример 15](#_Toc31063)

[Структура транслятора 16](#_Toc11781)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 16](#_Toc28954)

[2.2 Перечень параметров транслятора 17](#_Toc17839)

[2.3 Протоколы, формируемые транслятором 17](#_Toc27364)

[Разработка лексического анализатора 17](#_Toc23834)

[3.1 Структура лексического анализатора 18](#_Toc16610)

[3.2 Входные и выходные данные лексического анализатора 19](#_Toc1660)

[3.3 Параметры лексического анализатора 19](#_Toc16972)

[3.4 Алгоритм лексического анализа 19](#_Toc2740)

[3.5 Контроль входных символов 20](#_Toc6368)

[3.6 Удаление избыточных символов 21](#_Toc26471)

[3.7. Перечень ключевых слов 21](#_Toc17188)

[3.8 Основные структуры данных 22](#_Toc676)

[3.9 Структура и перечень сообщений лексического анализатора 23](#_Toc4585)

[3.10 Принцип обработки ошибок 23](#_Toc26873)

[3.11 Контрольный пример 23](#_Toc19103)

[Разработка синтаксического анализатора 24](#_Toc20705)

[4.1 Структура синтаксического анализатора 24](#_Toc23920)

[4.2 Контекстно свободная грамматика, описывающая синтаксис языка 24](#_Toc14064)

[4.3 Построение конечного магазинного автомата 26](#_Toc21843)

[4.4 Основные структуры данных 27](#_Toc16436)

[4.5 Описание алгоритма синтаксического разбора 27](#_Toc6499)

[4.6 Параметры синтаксического анализатора 28](#_Toc15506)

[4.7 Структура и перечень сообщений синтаксического анализатора 28](#_Toc4672)

[4.8 Принцип обработки ошибок 29](#_Toc12596)

[4.9 Контрольный пример 29](#_Toc6992)

[Разработка семантического анализатора 30](#_Toc21314)

[5.1 Структура семантического анализатора 30](#_Toc9998)

[5.2. Функции семантического анализатора 30](#_Toc23925)

[5.3 Структура и перечень сообщений семантического анализатора 31](#_Toc30412)

[5.4 Принцип обработки ошибок 31](#_Toc16209)

[5.5 Контрольный пример 32](#_Toc2454)

[Вычисление выражений 32](#_Toc23293)

[6.1 Выражения, допускаемые языком 33](#_Toc17173)

[6.2 Польская запись и принцип её построения 33](#_Toc28438)

[6.3 Программная реализация обработки выражений 34](#_Toc4772)

[6.4 Контрольный пример 35](#_Toc29687)

[Генерация кода 35](#_Toc25590)

[7.1 Структура генератора кода 36](#_Toc12353)

[7.2. Представление типов данных в оперативной памяти 36](#_Toc11771)

[7.3. Статическая библиотека 37](#_Toc9900)

[7.4. Особенности алгоритма генерации кода 37](#_Toc23019)

[7.5 Параметры, управляющие генерацией кода 38](#_Toc20332)

[7.6 Контрольный пример 38](#_Toc19354)

[Тестирование транслятора 39](#_Toc25006)

[8.1 Общие положения 39](#_Toc23942)

[8.2 Результаты тестирования 39](#_Toc22003)

[Заключение 40](#_Toc30270)

[Приложения 42](#_Toc20566)

[Приложение А 43](#_Toc18961)

[Приложение Б 44](#_Toc23046)

[Приложение В 51](#_Toc13096)

[Приложение Г 52](#_Toc19313)

[Приложение Д 54](#_Toc7029)

[Приложение Е 57](#_Toc5372)

[Приложение Ж 59](#_Toc3872)

[Приложение З 64](#_Toc16951)

[Приложение И 68](#_Toc859)

[Список использованной литературы 69](#_Toc13290)

**Введение**

Задачей курсового проекта является разработка транслятора для своего языка программирования: ARV-2024. Предназначен этот язык программирования для выполнения простейших операций над строками и числами.

Транслятор – это набор различных программ, которые способны конвертировать исходный код с одного языка программирования на другой. Основная функция транслятора – преобразовывать программу, написанную на языке программирования ARV-2024, в код, который может быть интерпретирован компьютером, в код на языке ассемблера.

Язык ассемблера – это низкоуровневый машинно-ориентированный язык программирования. Это система обозначений, используемая для представления программ, написанных машинным кодом, в удобной для чтения форме. Его команды непосредственно соответствуют отдельным командам машины или их последовательностям. Исходя из цели курсового проекта, были определены следующие задачи:

* разработка спецификации языка программирования;
* разработка структуры транслятора;
* разработка лексического анализатора;
* разработка синтаксического анализатора;
* разработка семантического анализатора;
* обработка выражений;
* генерация кода на язык Assembler;
* тестирование транслятора.

**1. Спецификация языка программирования**

**1.1 Характеристика языка программирования**

Язык программирования ARV-2024 является языком программирования высокого уровня. Этот язык строго типизируемый, то есть в языке отсутствует преобразование типов. Так же язык компилируемый, функциональный, но не является объектно-ориентированным.

Исходный код ARV-2024 может содержать символы латинского алфавита верхнего и малого регистров, цифры десятичной системы счисления от 0 до 9, символы табуляции и перевода строки, спецсимволы: {} () , ; : # + - / \* % & > < !.

**1.2 Применяемые сепараторы**

Сепараторы, используемые в языке программирования ARV-2024, приведены в таблице 1.1.

Таблица 1.1. Символы сепараторы

|  |  |
| --- | --- |
| Символ(ы) | Назначение |
| пробел | Разделитель цепочек, в именах идентификаторов и ключевых слов использовать нельзя |
| **{**  **}** | Блок функции и условного оператора |
| **(**  **)** | Блок фактических или формальных параметров функции, а так же приоритет арифметических операций |
| **,** | Разделитель параметров функций |
| **\*, &, !** | Логические операции |
| **>, <, !=, >=, <=, ==** | Операции сравнения: больше, меньше, не равно, больше или равно, меньше или равно |
| **;** | Разделитель программных конструкций |
| **=** | Оператор присваивания |
| **“”** | Запись строкового литерала |

Символы сепараторы служат в качестве разделителей операций языка.

**1.3 Применяемые кодировки**

В основе алфавита ARV-2024 лежит таблица символов Windows-1251. Исходный код ARV-2024 может содержать символы кириллицы, латинского алфавита верхнего и малого регистров, цифры десятичной системы счисления от 0 до 9, символы табуляции и перевода строки, спецсимволы: {} () , ; : # + - / \* % & > < !. Содержимое таблицы Windows-1251 представлено на рисунке 1.1.

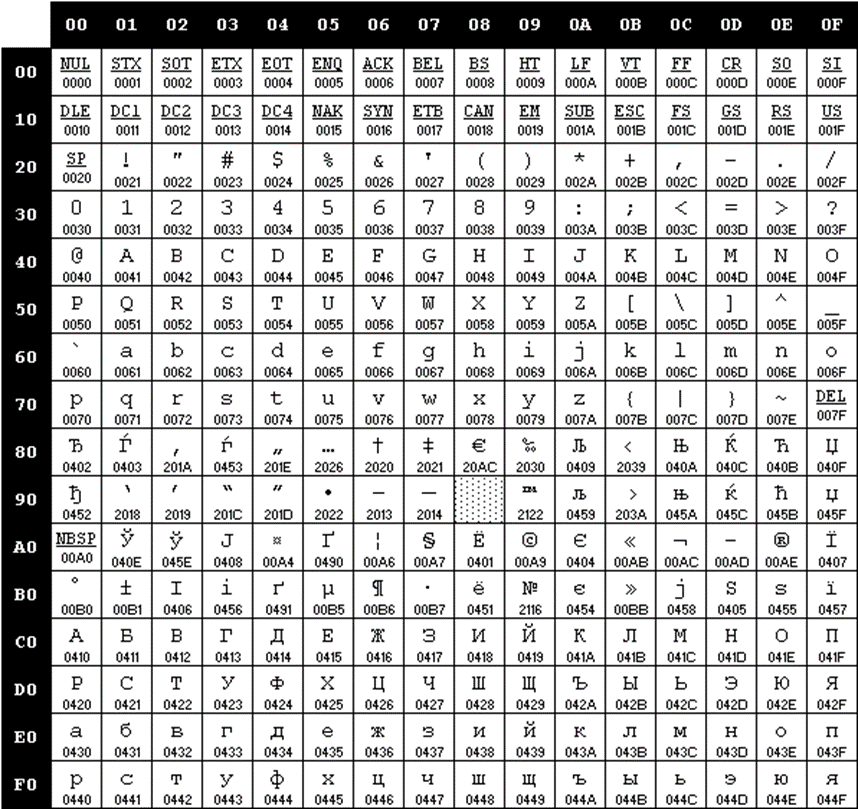


Рисунок 1.1 – Алфавит входных символов

Эта таблица символов является основой для кодирования и декодирования информации в ARV-2024.

**1.4 Типы данных**

В языке используется 2 основных типов данных, которые описываются в таблице 1.2.

Таблица 1.2. Описание типов данных

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| int | Фундаментальный тип данных. Предусмотрен для объявления целочисленных, беззнаковых данных (4 байта). Автоматически инициализируется нулевым значением.  Поддерживаемые операции:  \* (битовый) – логическое ИЛИ;  & (битовый) – логическое И;  ! (битовый) – инверсия;  + (бинарный) – оператор сложения;  В качестве операторов сравнения поддерживаются следующие операторы:  == (бинарный) – оператор “равенство”  > (бинарный) – оператор “больше”;  < (бинарный) – оператор “меньше”  != (бинарный) – оператор “не равно”  >= (бинарный) – оператор “больше либо равно”  <= (бинарный) – оператор “меньше либо равно” |
| str | Фундаментальный тип данных. Предусмотрен для объявления строк. (1 символ – 1 байт). Максимальный размер строки может быть 254 символа. Данные этого типа заключаются в “”. Автоматическая инициализация: символ конца строки ”\0”. |

Типы данных играют ключевую роль в структуре и функционировании языков программирования. Они обеспечивают безопасность типов, эффективное использование памяти и корректное выполнение операций.

**1.5 Преобразование типов данных**

В языке ARV-2024 не предусмотрена поддержка преобразования типов данных.

**1.6 Идентификаторы**

Для именования переменных, параметров и функций всегда используются идентификаторы. Идентификаторы могут содержать символы нижнего регистра. Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Идентификаторы не должны совпадать с ключевыми словами. Не предусмотрены зарезервированные идентификаторы.

Идентификатор должен составляться подобным образом:

* состоит из символов латинского алфавита [a-z]+ и непосредственно целое число состоящее из множества [0,1,2,...,9].

**1.7 Литералы**

Литерал — это элемент программы, который непосредственно представляет значение.

Все языки программирования, включая ARV-2024, имеют литералы. В ARV-2024 предусмотрены два типа литералов: целочисленные и строковые. Краткое описание литералов представлено в таблице 1.3.

Таблица 1.3. Литералы

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Литералы целого типа | Целочисленные литералы. Представляют непосредственно целое число состоящее из множества [0,1,2,...,9].Данные этого типа могут быть представлены в десятичной и шестнадцатеричной системе. |
| Строковые литералы | Строковые литералы. Представляют непосредственно строку состоящую из множества символов, заключенных в “” (двойные кавычки). |

Литералы представляют собой конкретные значения, которые не изменяются во время выполнения программы.

**1.8 Объявление данных**

Объявление всех типов данных, включая переменные и функции, в языке программирования ARV-2024 может происходить в любой части функции или ветвления. Объявление происходит с ключевого слово new. Дальше следует тип идентификатора, а после его название.

Пример:

new int a;

**1.9 Инициализация данных**

[Инициализация - это процесс присвоения начальных значений переменным перед выполнением программы](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B8%D1%86%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F). Способ инициализации переменной: без значения. При объявлении переменной без значения, ей будет присвоено значение по умолчанию. Для int нулевое значение, для str признак конца строки. Так же способы инициализации переменной представлены в таблице 1.4.

Таблица 1.4. Способы инициализации переменной

|  |  |
| --- | --- |
| Вид инициализации | Примечание |
| <тип данных> <идентификатор>; | Автоматическая инициализация: пере-менные типа int инициализируются нулём, переменные типа str – пустой строкой. |
| <идентификатор> = <значение>; | Присваивание переменной значения. |

Способ, которым выполняется инициализация, зависит от языка программирования, а также от типа, класса памяти и т. д. [инициализируемого объекта](https://wiki5.ru/wiki/Initialization_%28programming%29).

**1.10 Инструкции языка**

Все инструкции языка ARV-2024 представлены в таблице 1.5.

Таблица 1.5. Инструкции языка

|  |  |
| --- | --- |
| Инструкция | Запись |
| Объявление переменной | <тип данных> <идентификатор>; |
| Объявление с указанием области видимости | <тип данных> идентификатор функции идентификатор; |
| Присваивание | <идентификатор> = <значение>|<идентификатор>; |
| Объявление функции | <тип данных> function <идентификатор> (<тип данных> <идентификатор>, …) {…} |
| Блок инструкций | {  …  } |
| Вывод данных | print <выражение> |
| Вызов функций | <идентификатор функции>(<идентификатор | литерал>,…) |
| Условный оператор | if: <идентификатор>/<литерал>< логическая операция> <идентификатор>/ <литерал> # istrue{…} isfalse{…}# |

[Инструкции в языках программирования, также известные как операторы, являются наименьшими автономными частями языка программирования](https://bing.com/search?q=%d0%b8%d0%bd%d1%81%d1%82%d1%80%d1%83%d0%ba%d1%86%d0%b8%d0%b8+%d0%b2+%d1%8f%d0%b7%d1%8b%d0%ba%d0%b0%d1%85+%d0%bf%d1%80%d0%be%d0%b3%d1%80%d0%b0%d0%bc%d0%bc%d0%b8%d1%80%d0%be%d0%b2%d0%b0%d0%bd%d0%b8%d1%8f). [Они представляют собой команды или набор команд](https://bing.com/search?q=%d0%b8%d0%bd%d1%81%d1%82%d1%80%d1%83%d0%ba%d1%86%d0%b8%d0%b8+%d0%b2+%d1%8f%d0%b7%d1%8b%d0%ba%d0%b0%d1%85+%d0%bf%d1%80%d0%be%d0%b3%d1%80%d0%b0%d0%bc%d0%bc%d0%b8%d1%80%d0%be%d0%b2%d0%b0%d0%bd%d0%b8%d1%8f).

**1.11 Операции языка**

[Операции языка – это конструкции, которые аналогичны математическим операциям](https://bing.com/search?q=" \t "_blank). [Они представляют собой специальный способ записи некоторых действий](https://bing.com/search?q=). Операции языка ARV-2024 и их приоритет представлен в таблице 1.6.

Таблица 1.6. Операторы языка ARV-2024

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические | + **-** сложение |
| Сравнения | > - больше  < - меньше  != – не равно  >= – больше или равно  <= - меньше или равно  == - равно |
| Бинарные | \* - логическое ИЛИ  & - логическое И  ! - инверсия |

Поведение и синтаксис операций могут варьироваться в зависимости от конкретного языка программирования.

**1.12 Выражения и их вычисления**

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

* допускается использовать скобки для смены приоритета операций;
* выражение записывается в строку без переносов;
* использование двух подряд идущих операторов не допускается;
* допускается использовать в выражении вызов функции, вычисляющей и возвращающей целочисленное значение.

Перед генерацией кода каждое выражение приводится к записи в польской записи для удобства дальнейшего вычисления выражения на языке ассемблер

**1.13 Конструкции языка**

Программные конструкции языка программирования ARV-2024 представлены в таблице 1.7.

Таблица 1.7. Программные конструкции языка ARV-2024

|  |  |
| --- | --- |
| Главная функция (точка входа в приложение) | main  {…} |
| Функция | <тип данных> function <идентификатор> (<тип данных> <идентификатор>, …)  {  …  return <выражение>;  } |
| Условная конструкция condition | if: <условие>#  istrue {…}  isfalse {…}# |

Изучение характеристик конструкций языка программирования позволяет лучше понять, как эффективно использовать эти конструкции для реализации алгоритмов. Это включает в себя понимание того, как работают переменные, условные операторы, функции. Это знание помогает разработчикам писать более эффективный, читаемый и надежный код.

**1.14 Область видимости идентификаторов**

Областью видимости идентификатора по умолчанию является функция main. В случае, если идентификатор принадлежит какой-либо другой функции, область видимости переменной будет только внутри этой функции.

**1.15 Семантические проверки**

Таблица с перечнем семантических проверок, предусмотренных языком, приведена в таблице 1.8.

Таблица 1.8 – Семантические проверки

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Идентификаторы не должны повторяться |
| 2 | Тип данных передаваемых значений в функцию должен совпадать с типом параметров при её объявлении |
| 3 | В функцию должны быть переданы параметры |
| 4 | Тип данных результата выражения должен совпадать с типом данных идентификатора, которому оно присваивается |
| 5 | Индекс вызываемого элемента массива не должен превосходить размер массива |
| 6 | Функция main обязательно должна присутствовать и быть единственной |
| 7 | Параметру функции нельзя присвоить значение |

**1.16 Распределение оперативной памяти на этапе выполнения**

Все переменные хранятся в куче. Для таблиц лексем и идентификаторов выделяется динамическая память, которая очищается по завершению работы транслятора.

**1.17 Стандартная библиотека и её состав**

Функции стандартной библиотеки реализованы на языке программирования C++, а также представлены с описанием в таблице 1.9.

Таблица 1.9. Стандартные библиотеки

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| void outstr | - | string value | Функция выводит в консоль строковый литерал или переменную, переданный ей через параметры. |
| int lenght | int | string value | Функция возвращает длины строки. |
| void outnum | - | int | Функция выводит в консоль целочисленный литерал или переменную, переданный ей через параметры. |
| int strcmpp | int | string first, string last | Функция сравнивает две строки, если строки равны возвращает ноль, в противном случае – 1. |
| string strcopy | string | string buffer, string str | Функция копирует значение второго строкового параметра в первый |

Стандартная библиотека включает функции для работы со строками. Она является важным инструментом в арсенале любого разработчика.

**1.18 Ввод и вывод данных**

В языке ARV-2024 не реализованы средства ввода данных.

В стандартной библиотеке языка программирования предусмотрены функции outnum, outstr для вывода данных в стандартный поток вывода. Эти функции позволяют разработчикам выводить информацию, которую они хотят отобразить пользователю или для отладки своих программ.

**1.19 Точка входа**

Каждый исходный код на языке программирования ARV-2024 должен содержать главную функцию (точку входа) main. Повторение функции main не предусмотрено и вызывает ошибку.

**1.20 Препроцессор**

Наличие препроцессора в языке программирования ARV-2024 не предусмотрено.

**1.21 Соглашения о вызовах**

Вызов любых функций происходит при помощи соглашения о вызовах stdcall. Особенности stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

**1.22 Объектный код**

Язык программирования ARV-2024 транслируется в язык ассемблера.

**1.23 Классификация сообщений транслятора**

Их классификация сообщений транслятора представлена в таблице 1.10.

Таблица 1.10. Классификация сообщений транслятора

|  |  |
| --- | --- |
| Интервал | Описание ошибок |
| 0-103 | Системные ошибки, ошибки параметров |
| 200-299 | Ошибки лексического анализа |
| 300-599 | Ошибки синтаксического анализа |
| 600-900 | Ошибки семантического анализа |

При возникновении ошибки в коде программы на языке ARV-2024, транслятором генерируется исключение, которое укажет примерную причину неработоспособности исходного кода.

**1.24 Контрольный пример**

Контрольный пример представлен в Приложении [A](#_Приложение_А).

**2. Структура транслятора**

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

Транслятор преобразует программу, написанную на языке ARV-2024 в программу на языке ассемблера. Для указания выходных файлов используются входные параметры транслятора, которые описаны в пункте 2.2. Компонентами транслятора являются лексический, синтаксический и семантический анализаторы, а также генератор кода на язык ассемблера. Принцип их взаимодействия представлен на рисунке 2.1.

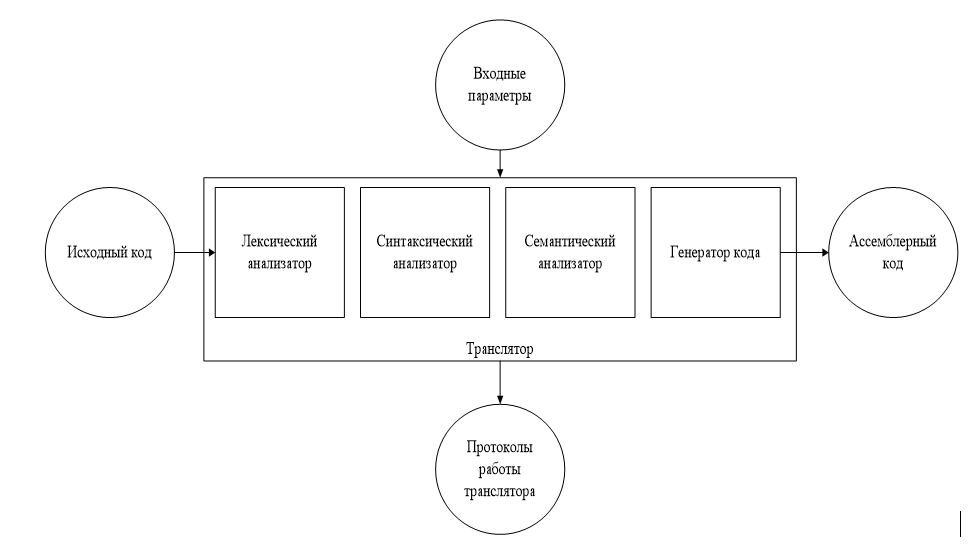


Рисунок 2.1 – Структура транслятора

Лексический анализ – первая фаза трансляции. Назначением лексического анализатора является нахождение ошибок лексики языка ARV-2024 и формирование таблицы лексем и таблицы идентификаторов.

Синтаксический анализ – это основная часть транслятора, предназначенная для распознавания синтаксических конструкций и формирования промежуточного кода ARV-2024. Для этого используются таблица лексем и идентификаторов. Синтаксический анализатор распознаёт синтаксические конструкции, выявляет синтаксические ошибки при их наличии и формирует дерево разбора

Семантический анализ в свою очередь является проверкой исходной программы ARV-2024 на семантическую согласованность с определением языка.

Генератор кода – этап транслятора, выполняющий генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. Генератор кода принимает на вход таблицы идентификаторов и лексем и транслирует код на языке ARV-2024, прошедший успешно все предыдущие этапы, в код на языке Ассемблера.

**2.2 Перечень параметров транслятора**

Входные параметры представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка ARV-2024

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Значение по умолчанию |
| -in:<имя\_файла> | Входной файл с любым расширением, в котором содержится исходный код на языке ARV-2024. Данный параметр должен быть указан обязательно. В случае если он не будет задан, то выполнение этапа трансляции не начнётся. | Не предусмотрено |
| -log:<имя\_файла> | Файл содержит в себе краткую информацию об исходном коде на языке ARV-2024. | <имя\_файла>.log |
| -out:<имя\_файла> | В этот файл будет записан результат обработки исходного кода | <имя\_файла>.out |

Таблица содержит перечень основных входных параметров транслятора ARV-2024.

**2.3 Протоколы, формируемые транслятором**

Результатом работы транслятора языка ARV-2024 является исходный код на языке ассемблера и протокол работы транслятора.

Таблица с перечнем протоколов, формируемых транслятором языка ARV-2024 и их назначением представлена в таблице 2.2.

Таблица 2.2 – Протоколы, формируемые транслятором языка ARV-2024

|  |  |
| --- | --- |
| Формируемый протокол | Описание протокола |
| Файл журнала, “log.txt” | Файл содержит в себе краткую информацию об исходном коде на языке ARV-2024. В этот файл могут быть выведены таблицы идентификаторов, лексем, а также сообщения о возникших ошибках. |
| “ASM.asm” | Содержит сгенерированный код на языке Ассемблера. |

Протокол содержащий основную информацию о процессе обработки исходного кода.

**3.** **Разработка лексического анализатора**

**3.1 Структура лексического анализатора**

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся исходный код входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка. Лексический анализатор производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов.

Структура лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1 — Структура лексического анализатора ARV-2024

Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т. д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация.

**3.2 Входные и выходные данные лексического анализатора**

[Лексический анализатор - это часть компилятора, которая читает символы программы на исходном языке и строит из них слова (лексемы) исходного языка](https://bing.com/search?q=%d0%b2%d1%85%d0%be%d0%b4%d0%bd%d1%8b%d0%b5+%d0%b8+%d0%b2%d1%8b%d1%85%d0%be%d0%b4%d0%bd%d1%8b%d0%b5+%d0%b4%d0%b0%d0%bd%d0%bd%d1%8b%d0%b5+%d0%bb%d0%b5%d0%ba%d1%81%d0%b8%d1%87%d0%b5%d1%81%d0%ba%d0%be%d0%b3%d0%be+%d0%b0%d0%bd%d0%b0%d0%bb%d0%b8%d0%b7%d0%b0%d1%82%d0%be%d1%80%d0%b0).

[**Входные данные** лексического анализатора - это текст исходной программы](https://bing.com/search?q=%d0%b2%d1%85%d0%be%d0%b4%d0%bd%d1%8b%d0%b5+%d0%b8+%d0%b2%d1%8b%d1%85%d0%be%d0%b4%d0%bd%d1%8b%d0%b5+%d0%b4%d0%b0%d0%bd%d0%bd%d1%8b%d0%b5+%d0%bb%d0%b5%d0%ba%d1%81%d0%b8%d1%87%d0%b5%d1%81%d0%ba%d0%be%d0%b3%d0%be+%d0%b0%d0%bd%d0%b0%d0%bb%d0%b8%d0%b7%d0%b0%d1%82%d0%be%d1%80%d0%b0). [Это может быть последовательность символов, представляющая собой код программы](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7).

[**Выходные данные** лексического анализатора - это списковая структура, содержащая лексемы в числовом представлении](https://studfile.net/preview/1582161/page:5/). [Это идентифицированные последовательности, называемые "токенами"](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7). Каждый токен можно представить в виде структуры, содержащей идентификатор токена (или идентификатор класса токена) и, если нужно, последовательность символов лексемы, выделенной из входного потока (строку, число и т. [д.)](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7).

**3.3 Параметры лексического анализатора**

Входным параметром лексического анализатора является структура IN, которая содержит исходный текст программы, написанный на языке ARV-2024, а также структура LOG, которая содержит файл протокола.

**3.4 Алгоритм лексического анализа**

Лексический анализ выполняется программой (входящей в состав транслятора), называемой лексическим анализатором. Цель лексического анализа — выделение и классификация лексем в тексте исходной программы. Лексический анализатор производит распознаёт и разбирает цепочки исходного текста программы. Это основывается на работе конечных автоматов, которые представлены в виде графов. Регулярные выражения — аналитический или формульный способ задания регулярных языков. Они состоят из констант и операторов, которые определяют множества строк и множество операций над ними. Любое регулярное выражение можно представить в виде графа.

Пример. Регулярное выражение для ключевого слова print.

Граф конечного автомата для этой лексемы представлен на рисунке 3.4. S0 – начальное состояние, S5 – конечное состояние автомата. В виде кода представлен на рисунке 3.5.

r

t

n

i

p

Рисунок 3.4 – Граф переходов для цепочки ‘print’

|  |
| --- |
| #define GRAPH\_PRINT 6, \  FST::NODE(1, FST::RELATION(‘p’, 1),\  FST::NODE(1, FST::RELATION(‘r’, 2),\  FST::NODE(1, FST::RELATION(‘i’, 3),\  FST::NODE(1, FST::RELATION(‘n’, 4),\  FST::NODE(1, FST::RELATION(‘t’, 15,\  FST::NODE() |

Листинг 3.5 – Граф переходов для цепочки “ print”

Для ускорения работы анализатора был добавлен просмотр первого символа слова, за счет этого отсеиваются неподходящие графы. Результат работы лексического анализатора – сформированные таблицы лексем и идентификаторов.

**3.5 Контроль входных символов**

Исходный код на языке программирования ARV-2024 прежде, чем транслироваться, проверяется на допустимость символов. То есть изначально из входного файла считывается по одному символу и проверяется является ли он разрешённым.

Таблица для контроля входных символов представлена на рисунке 3.2.

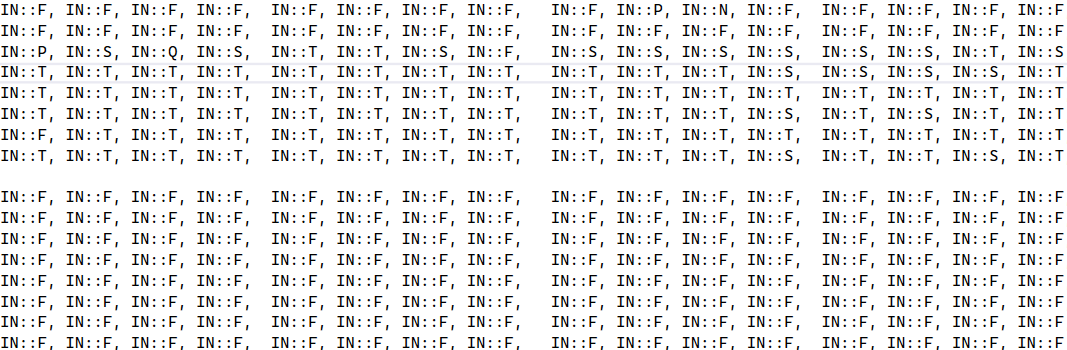


Рисунок 3.2. Таблица контроля входных символов

Принцип работы таблицы заключается в соответствии значения каждому элементу в шестнадцатеричной системе счисления значению в таблице ASCII.

Описание значения символов: T – разрешённый символ, F – запрещённый символ, P – пробел, табуляция, S – ( ) {} # < > ! & \* + - = , N – новая строка, Q – ", I – игнорируемый символ.

**3.6 Удаление избыточных символов**

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на лексемы.

Описание алгоритма удаления избыточных символов:

1) Посимвольно считываем исходный код, занесенный в структуру In.

2) Встреча пробела или знака табуляции вне пределов строкового литерала является своего рода встречей символа-сепаратора.

3) В отличие от других символов-сепараторов не записываем в таблицу лексем эти символы, т.е. игнорируем.

**3.7. Перечень ключевых слов**

Лексемы – это символы, соответствующие ключевым словам, символам операций и сепараторам, необходимые для упрощения дальнейшей обработки исходного кода программы. Данное соответствие описано в таблице 3.1.

Таблица 3.1 Соответствие ключевых слов, символов операций и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| main | m | Главная функция. |
| = | = | Объявление переменной. |
| function | f | Объявление функции. |
| print | p | Ввод данных. |
| return | r | Возврат функцией значения |
| int, string | t | Названия типов данных языка. |
| идентификатор | i | Идентификатор |
| литерал | l | Литерал любого доступного типа. |
| if | ? | Оператор условного блока |
| ; | ; | Разделение выражений. |
| , | , | Разделение параметров функций. |

Окончание таблицы 3.1

|  |  |  |
| --- | --- | --- |
| +  -  \*  &  ! | +  -  \*  &  ! | Знаки арифметических операций. |
| >  <  >=  <=  ==  != | >  <  b  m  g  h | Знаки операторов сравнения |
| { | { | Начало блока/тела функции. |
| } | } | Закрытие блока/тела функции. |

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов.

Также в приложении [В](#_Приложение_В) находятся конечные автоматы, соответствующие лексемам языка ARV-2024.

**3.8 Основные структуры данных**

Основные структуры таблиц лексем и идентификаторов данных языка ARV-2024, используемых для хранения, представлены в приложении Б. В таблице лексем содержится лексема, её номер, полученный при разборе, номер строки в исходном коде и индекс таблицы идентификаторов. В таблице идентификаторов содержится имя идентификатора, номер в таблице лексем, тип данных, тип идентификатора и его значение.

**3.9 Структура и перечень сообщений лексического анализатора**

Структура и перечень сообщений лексического анализатора представлен в таблице 3.2.

Таблица 3.2. Классификация сообщений транслятора

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

Перечень сообщений лексического анализатора информирует о ошибках на стадии лексического анализа.

**3.10 Принцип обработки ошибок**

Когда возникает ошибка – работа транслятора прекращается, а ошибка записывается в log журнал.

**3.11 Контрольный пример**

Результат работы лексического анализатора – таблицы лексем и идентификаторов – представлен в приложении [Б](#_Приложение_Б).

**4. Разработка синтаксического анализатора**

**4.1 Структура синтаксического анализатора**

Синтаксический анализ – это фаза трансляции, выполняемая после лексического анализа и предназначенная для распознавания синтаксических конструкций ARV-2024. Структура синтаксического анализатора представлена на рисунке 4.1.

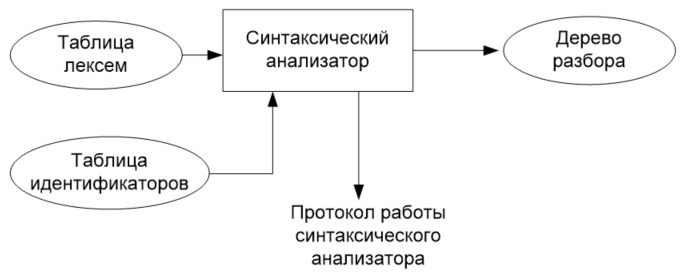


Рисунок 4.1 – Структура синтаксического анализатора ARV-2024

Входом для синтаксического анализа является таблица лексем и таблица идентификаторов, полученные после фазы лексического анализа. Выходом – дерево разбора.

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

В синтаксическом анализаторе транслятора языка ARV-2024 используется контекстно-свободная грамматика , где

T – множество терминальных символов (было описано в разделе 1.2 данной пояснительной записки),

N – множество нетерминальных символов (первый столбец таблицы 4.1),

P – множество правил языка (второй столбец таблицы 4.1),

S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, т.к. она не леворекурсивная (не содержит леворекурсивных правил) и правила  имеют вид:

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

Таблица 4.1 – Перечень правил, составляющих грамматику языка и описание нетерминальных символов ARV-2024

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| S | tfiPTS  pfiPGS  m{K} | Порождает правила, описывающее общую структуру программы |
| T | {eV;}  {KeV;} | Порождает правила, описывающие тело функции |
| G | {e;}  {Ke;} | Порождает правила, описывающие список параметров функции |
| E | ti, E  ti | Порождает правило, описывающее параметры функции |
| F | (N)  () | Порождающее правило вызова функции |
| R | rY#  wY#  cY#  rYwY#  wYrY# | Конструкция условного выражения |
| Y | {X} | Порождает правила, описывающие тело условного выражения |
| Z | iLi  iLl  lLi | Порождает правила, описывающие тело уловного выражения |
| L | <  >  g  ! | Порождает правила, описывающие условные операторы |
| A | +  -  \*  & | Арифметические операции |
| W | i  l  (W)  (W)AW  iF  iAW  lAW  iFAW  !W | Арифметические выражения |

Окончание таблицы

|  |  |  |
| --- | --- | --- |
| K | nti=V;K  nti;K  i=W;K  oV;K  ^;K  ?Z#RK  iF;K  nti=V;  i=W;  nti;  oV  ^;  ?Z#R  iF; | Синтаксические конструкции языка |

|  |  |  |
| --- | --- | --- |
| X | i=W;X  oV;X  ^;X  iF;X  i=W;  oV;  ^;  iF; | Синтаксические конструкции языка |

Грамматика языка ARV-2024 представлена в приложении Б.

TS – терминальные символы, которыми являются сепараторы, знаки арифметических операций и некоторые строчные буквы.

NS – нетерминальные символы, представленные несколькими заглавными буквами латинского алфавита.

**4.3 Построение конечного магазинного автомата**

Конечный автомат с магазинной памятью представляет собой семерку, описание которой 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 |

Окончание таблицы 4.2

|  |  |  |
| --- | --- | --- |
|  | Алфавит входных символов | Алфавит является множеством терминальных и нетерминальных символов, описание которых содержится в разделе 1.2 и в таблице 4.1. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека |
|  | 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 (нетерминальный символ А) |
|  | 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 |

Структура данного автомата показана в приложении [Г](#_Приложение_Г).

**4.4 Основные структуры данных**

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного конечного автомата и структуру грамматики Грейбах, описывающей правила языка ARV-2024. Данные структуры представлены в приложении [Г](#_Приложение_Г).

**4.5 Описание алгоритма синтаксического разбора**

Принцип работы автомата следующий:

1. В магазин записывается стартовый символ;
2. На основе полученных ранее таблиц формируется входная лента;
3. Запускается автомат;
4. Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке;
5. Если терминалы в стеке и в ленте совпадают, то данный терминал удаляется из ленты и стека. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другую цепочку нетерминала;
6. Если в магазине встретился нетерминал, переходим к пункту 4;
7. Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно и формируется дерево разбора.

**4.6 Параметры синтаксического анализатора**

Входным параметром синтаксического анализатора является структура LexTable, которая содержит сформированную таблицу лексем, полученную на этапе лексического анализа, потоки вывода протокола, а также правила контекстно-свободной грамматики в форме Грейбах.

Выходными параметрами являются трассировка прохода таблицы лексем и правила разбора, которые записываются в файл протокола.

**4.7 Структура и перечень сообщений синтаксического анализатора**

Перечень сообщений синтаксического анализатора представлен в таблице 4.3.

Таблица 4.3 – Перечень сообщений синтаксического анализатора

|  |  |
| --- | --- |
| Номер ошибки | Описание ошибки |
| 600 | Неверная сруктура программы |
| 601 | Не найден список параметров функции |
| 602 | Ошибка в теле функции |
| 603 | Ошибка в теле скрипта |
| 604 | Ошибка в списке параметров функции |
| 605 | Ошибка в вызове функции/выражении |
| 606 | Ошибка в списке фактических параметров функции |
| 607 | Ошибка при конструировании условного оператора |
| 608 | Ошибка в теле условного выражения |
| 609 | Ошибка в условии условного выражения |
| 610 | Неверный условный оператор |
| 611 | Неверный арифметический оператор |
| 612 | Неверное выражение. Ожидаются только идентификаторы / литералы |
| 613 | Ошибка в арифметическом выражении |
| 614 | Недопустимая синтаксическая конструкция |
| 615 | Недопустимая синтаксическая конструкция в теле условного оператора |

Перечень сообщений синтаксического анализатора информирует о ошибках на стадии лексического анализа.

**4.8 Принцип обработки ошибок**

Обработка ошибок происходит следующим образом:

1. Синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.
2. Если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка.
3. Все ошибки записываются в общую структуру ошибок.
4. В случае нахождения ошибки в протокол будет выведено диагностическое сообщение.

**4.9 Контрольный пример**

Пример разбора синтаксическим анализатором исходного кода на языке ARV-2024 представлен в приложении [Е](#_Приложение_Е).

**5. Разработка семантического анализатора**

**5.1 Структура семантического анализатора**

Семантический анализ происходит при выполнении фазы лексического анализа и реализуется в виде отдельных проверок текущих ситуаций в конкретных случаях: установки флага или нахождении в особом месте программы (оператор выхода из функции, оператор ветвления, вызов функции стандартной библиотеки). Структура семантического анализатора представлена на рисунке 5.1.

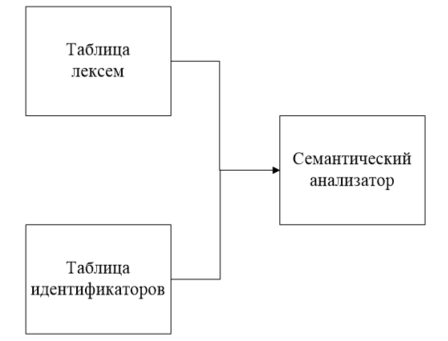


Рисунок 5.1 — структура семантического анализатора

[Семантический анализ используется для проверки программ на соответствие определенным семантическим правилам и для определения смысла программы](https://rdc.grfc.ru/2021/09/semantic_analysis/).

**5.2. Функции семантического анализатора**

Назначение семантического анализа – проверка смысловой правильности

конструкций языка программирования

Входные данные для семантического анализатора:

* таблица идентификаторов;
* дерево разбора – результат разбора синтаксических конструкций входного языка.

Основные действия семантического анализатора:

1) проверка соблюдения в исходной программе семантических правил

входного языка;

2) дополнение внутреннего представления программы в компиляторе

операторами и действиями, неявно предусмотренными семантикой

входного языка;

3) проверка элементарных семантических (смысловых) норм языка

программирования.

Семантический анализатор выполняет проверку на основные правила языка (семантики языка), которые описаны в разделе 1.16.

**5.3 Структура и перечень сообщений семантического анализатора**

Сообщения, формируемые семантическим анализатором, представлены в таблице 5.1.

Таблица 5.1 – Описание компонентов магазинного автомата

|  |  |
| --- | --- |
| Номер ошибки | Описание ошибки |
| 300 | Необъявленный идентификатор |
| 301 | Отсутствует точка входа main |
| 302 | Обнаружено несколько точек входа main |
| 303 | В объявлении не указан тип идентификатор |
| 304 | В объявлении отсутствует ключевое слово new |
| 305 | Попытка переопределения идентификатора |
| 306 | Превышено максимальное количество параметров функции |
| 307 | Слишком много параметров в вызове |
| 308 | Кол-во ожидаемых функцией и передаваемых параметров не совпадает |
| 309 | Несовпадение типов передаваемых параметров |
| 310 | Использование пустого строкового литерала недопустимо |
| 311 | Обнаружен символ “. Возможно не закрыт строковый литерал |
| 312 | Превышен размер строкового литерала |
| 313 | Недопустимый целочисленный литерал |
| 314 | Типы данных в выражении не совпадают |
| 315 | Тип функции и возвращаемого значения не совпадают |
| 316 | Недопустимое строковое выражение справа от знака ‘=’ |
| 317 | Неверное условное выражение |
| 318 | Деление на ноль |

Перечень сообщений семантического анализатора информирует о ошибках на стадии семантического анализа.

**5.4 Принцип обработки ошибок**

Семантический анализатор проверяет, что объявления и утверждения программы семантически верны. Например: соответствие типов данных в выражении, совпадение фактических и формальных параметров функции.

Обработка ошибок происходит следующим образом:

1. Анализатор перебирает таблицу лексем.
2. Проверяет исключительные ситуации, которые могут быть связаны с данной лексемой.
3. В случае нахождения ошибки, она записывается в log журнал и дублируется в консоль.

**5.5 Контрольный пример**

Контрольный пример находится в приложении А.

**6.** **Вычисление выражений**

**6.1 Выражения, допускаемые языком**

В языке ARV-2024 допускаются выражения, применимые к целочисленным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, \* , ! , & , и (), и вызовы функций как операнды арифметических выражений.

Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритет операций в языке ARV-2024

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| ( ) | 3 |
| ! | 2 |
| \* | 2 |
| & | 2 |
| + | 1 |

[Приоритет операций в языках программирования определяет порядок, в котором выполняются операции в выражении](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8).

**6.2 Польская запись и принцип её построения**

Выражения в языке ARV-2024 преобразовываются к обратной польской записи.

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

Обратная польская запись — это форма записи математических выражений, в которой операторы расположены после своих операндов. Выражение в обратной польской нотации читается слева направо: операция выполняется над двумя операндами, непосредственно стоящими перед знаком этой операции.

Алгоритм построения:

– читаем очередной символ;

– если он является идентификатором или литералом, то добавляем его к выходной строке;

– если символ является символом функции, то помещаем его в стек;

– если символ является открывающей скобкой, то она помещается в стек;

– исходная строка просматривается слева направо;

– если символ является закрывающей скобкой, то выталкиваем из стека в выходную строку все символы пока не встретим открывающую скобку. При этом обе скобки удаляются и не попадают в выходную строку;

– как только входная лента закончится все символы из стека выталкиваются в выходную строку;

– в случае если встречаются операции, то выталкиваем из стека в выходную строку все операции, которые имеют выше приоритетность чем последняя операция;

– также, если идентификатор является именем функции, то он заменяется на спецсимвол «@».

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Исходная строка | Результирующая строка | Стек |
| i\*i-(i(i,i)-(i-l))\*l |  |  |
| \*i-(i(i,i)-(i-l))\*l | i |  |
| i-(i(i,i)-(i-l))\*l | i | \* |
| -(i(i,i)-(i-l))\*l | ii | \* |
| (i(i,i)-(i-l))\*l | ii\* | - |
| i(i,i)-(i-l))\*l | ii\* | -( |
| (i,i)-(i-l))\*l | ii\*i | -(@ |
| i,i)-(i-l))\*l | ii\*i | -(@( |
| ,i)-(i-l))\*l | ii\*ii | -(@( |
| i)-(i-l))\*l | ii\*ii | -(@( |
| )-(i-l))\*l | ii\*iii | -(@( |
| -(i-l))\*l | ii\*iii@2 | -( |
| (i-l))\*l | ii\*iii@2 | -(- |
| i-l))\*l | ii\*iii@2 | -(-( |
| -l))\*l | ii\*iii@2i | -(-( |
| l))\*l | ii\*iii@2i | -(-(- |
| ))\*l | ii\*iii@2il | -(-(- |
| )\*l | ii\*iii@2il- | -(- |
| \*l | ii\*iii@2il-- | - |
| l | ii\*iii@2il-- | -\* |
|  | ii\*iii@2il--l | -\* |
|  | ii\*iii@2il--l\* | - |
|  | ii\*iii@2il--l\*- |  |

[Польская запись упрощает вычисления и улучшает эффективность обработки выражений](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D1%81%D0%BA%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C).

**6.3 Программная реализация обработки выражений**

Программная реализация алгоритма преобразования выражений в обратный польский формат основана функции PolishNotation. Функция принимает как параметр адрес таблицы лексем и содержит цикл, в ходе которого перебираются все лексемы исходного кода. Если последовательность лексем соответствует началу выражения, проводится точечное преобразование выражений к польской нотации.

**6.4 Контрольный пример**

Пример преобразования выражений из контрольных примеров к обратной польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления и преобразования к ассемблерному коду. В приложении [И](#_Приложение_И) приведены изменённая таблицы лексем, отображающая результаты преобразования выражений в польский формат.

**7. Генерация кода**

**7.1 Структура генератора кода**

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка.

Структура генератора кода представлена на рисунке 7.1.



Рисунок 7.1 — Структура генератора кода

На вход генератора подаются таблицы лексем и идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

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

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблера – .data и .const. Идентификаторы языка ARV-2024 размещены в сегменте данных(.data). Литералы – в сегменте констант (.const). Соответствия между типами данных идентификаторов на языке ARV-2024 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка ARV-2024 и языка Ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке ARV-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| int | SDWORD | Хранит целочисленный тип данных без знака. |
| str | BYTE | Хранит указатель на начало строки. |
| literal | BYTE  SDWORD | Литералы: символьные,  целочисленные |

Различные типы данных используют разное количество байтов и имеют свои особенности представления.

**7.3. Статическая библиотека**

В языке ARV-2024 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++.

Функции из стандартной библиотеки содержатся в проекте GenLib, в свойствах которого указан тип конфигурации «статическая библиотека». Подключение библиотеки происходит с помощью includelib на этапе генерации кода.

Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически.

Все функции статической библиотеки представлены в пункте 1.17.

* 1. **Особенности алгоритма генерации кода**

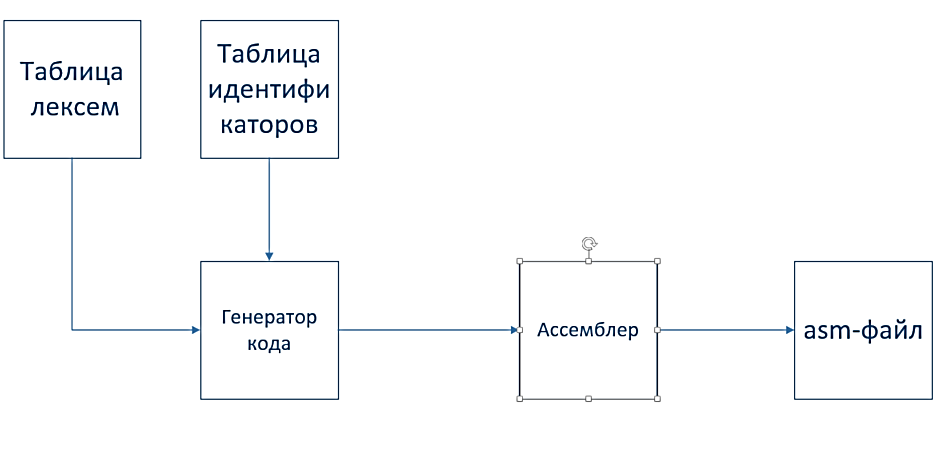
В языке ARV-2024 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2.

Рисунок 7.2 – Структура генератора кода

Генерация кода языка ARV-2024 происходит поэтапно. Сразу записывается типичные данные (модель памяти, соглашение о вызовах), потом прототипы функций статической библиотеки для языка программирования ARV-2024. В блок ‘const’ записываются все литералы, а после в блок ‘data’ записываются все переменные. После этого записываются блоки всех функций, все действия внутри этого блока транслируются в код ассемблера с помощью конструкций switch и case. В завершении в файл с расширением записывается конец главной процедуры ‘main’ и завершение программы.

**7.5 Параметры, управляющие генерацией кода**

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке ARV-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 приведён в приложении [З](#_Приложение_З).

**8. Тестирование транслятора**

**8.1 Общие положения**

В языке ARV-2024 не разрешается использовать запрещённые входным алфавитом символы. Допустимость входных символов проверяется при считывании данных из файла, если символ запрещён, то компилятор генерирует исключение.

При этапе лексического анализа, компилятор пытается распознать входящее слово. Если слово не распознано, то компилятор генерирует исключение.

Во время проверки входных данных синтаксическим анализатором, проверяется порядок использования лексем, то есть их структура. Если подходящее правило не найдено, то синтаксический анализатор диагностирует ошибку и генерирует исключение.

Так же в языке программирования ARV-2024 используется семантический анализатор. Он в свою очередь проверяет входящие данные на семантику. При неправильной семантике он также генерирует исключение.

**8.2 Результаты тестирования**

Результат тестирования входных символов на допустимость представлен в таблице 8.1.

Таблица 8.1 –Тестирование фазы проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| мmain | Ошибка N200: Лексическая ошибка: Недопустимый символ в исходном файле(-in) Строка: 3 Позиция в строке: 1 |

Результаты тестирования синтаксического анализатора показаны в таблице 8.2.

Таблица 8.2 – Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| new int n;; | Ошибка 614: строка 4, Синтаксическая ошибка: Недопустимая синтаксическая конструкция |
| int function(int a,, int b)  {  …  } | Ошибка 600: строка 2, Синтаксическая ошибка: Неверная структура программы |

Итоги тестирования семантического анализатора приведены в таблице 8.3.

Таблица 8.3 –Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| new int n;  new int n; | Ошибка N305: Семантическая ошибка: Попытка переопределения идентификатора Строка: 6 |
| new string s = "; | Ошибка N311: Семантическая ошибка: Обнаружен символ ' " '. Возможно, не закрыт строковый литерал Строка: 6 |

**Заключение**

В результате курсовой работы был разработан транслятор, который состоит из лексического, синтаксического и семантического анализаторов, а также и генератор кода для языка программирования ARV-2024. По завершению курсового проекта, все поставленные задачи были выполнены успешно:

1. Сформулирована спецификация языка ARV-2024;
2. Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
3. Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
4. Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
5. Осуществлена программная реализация синтаксического анализатора;
6. Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
7. Разработан транслятор кода на язык ассемблера;
8. Проведено тестирование всех вышеперечисленных компонентов.

Выполнение курсового проекта позволило укрепить свои знания, а также узнать более подробно о разных аспектах языках C++ и ассемблера. Также это закрепило понимание механизма обработки различных языков программирования.

**Список использованных источников**

1. Курс лекций по предмету «Конструирование программного обеспечения» Наркевич А.С.
2. Ахо, А. Теория синтаксического анализа, перевода и компиляции /А. Ахо, Дж. Ульман. – Москва: Мир, 1998. – Т. 2: Компиляция. - 487 с.
3. Ирвин К. Р. Язык ассемблера для процессоров Intel / К. Р. Ирвин. – M.: Вильямс, 2005. – 912с.
4. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва: Вильямс, 2003. - 429 с.

**Приложения**

**Приложение А**

Контрольный пример:

|  |
| --- |
| int function max(int a, int b)  {  new int res = 0;  if: a > b #  istrue {  res = a;  }  isfalse {  res = b;  } #  return res;  }  main  {  new int a = 16xF;  print a;  newline;  new int c;  c = max(2, 5);  new string str = "max(2, 5): ";  print str;  print c;  newline;  new int aa = 4;  new int ab = 2;  new int cc;  cc = aa \* ab;  print "4 or 2: ";  print cc;  newline;  new int cd;  cd = 4 & 6;  print "4 and 6: ";  print cd;  newline;  new int ce;  ce = ! 3;  print "not 3: ";  print ce;  newline;    new string stra = "ded";  new string strb = "ded";  new string strc = "deed";  new string strd = "deb";  new int resaa;  print "strcmp(ded, ded): ";  resaa = strcmpp(stra, strb);  print resaa;  newline;    print "strcmp(ded, deed): ";  resaa = strcmpp(stra, strc);  print resaa;  newline;    print "strcmp(ded, deeb): ";  resaa = strcmpp(stra, strd);  print resaa;  newline;    resaa = lenght("123456");  print "lenght(123456): ";  print resaa;  newline;  newline;  } |

**Приложение Б**

Таблица лексем:

|  |
| --- |
| | N | ЛЕКСЕМА | СТРОКА | ИНДЕКС В ТИ |  | 0 | t | 1 | |  | 1 | f | 1 | |  | 2 | i | 1 | 0 |  | 3 | ( | 1 | |  | 4 | t | 1 | |  | 5 | i | 1 | 1 |  | 6 | , | 1 | |  | 7 | t | 1 | |  | 8 | i | 1 | 2 |  | 9 | ) | 1 | |  | 10 | { | 2 | |  | 11 | n | 3 | |  | 12 | t | 3 | |  | 13 | i | 3 | 3 |  | 14 | = | 3 | |  | 15 | l | 3 | 4 |  | 16 | ; | 3 | |  | 17 | ? | 4 | |  | 18 | i | 4 | 1 |  | 19 | > | 4 | |  | 20 | i | 4 | 2 |  | 21 | # | 4 | |  | 22 | w | 5 | |  | 23 | { | 5 | |  | 24 | i | 6 | 3 |  | 25 | = | 6 | |  | 26 | i | 6 | 1 |  | 27 | ; | 6 | |  | 28 | } | 7 | |  | 29 | r | 8 | |  | 30 | { | 8 | |  | 31 | i | 9 | 3 |  | 32 | = | 9 | |  | 33 | i | 9 | 2 |  | 34 | ; | 9 | |  | 35 | } | 10 | |  | 36 | # | 10 | |  | 37 | e | 11 | |  | 38 | i | 11 | 3 |  | 39 | ; | 11 | |  | 40 | } | 12 | |  | 41 | m | 14 | |  | 42 | { | 15 | |  | 43 | n | 16 | |  | 44 | t | 16 | |  | 45 | i | 16 | 5 |  | 46 | = | 16 | |  | 47 | l | 16 | 6 |  | 48 | ; | 16 | |  | 49 | o | 17 | |  | 50 | i | 17 | 5 |  | 51 | ; | 17 | |  | 52 | ^ | 18 | |  | 53 | ; | 18 | |  | 54 | n | 20 | |  | 55 | t | 20 | |  | 56 | i | 20 | 7 |  | 57 | ; | 20 | |  | 58 | i | 21 | 7 |  | 59 | = | 21 | |  | 60 | i | 21 | 0 |  | 61 | ( | 21 | |  | 62 | l | 21 | 8 |  | 63 | , | 21 | |  | 64 | l | 21 | 9 |  | 65 | ) | 21 | |  | 66 | ; | 21 | |  | 67 | n | 22 | |  | 68 | t | 22 | |  | 69 | i | 22 | 10 |  | 70 | = | 22 | |  | 71 | l | 22 | 11 |  | 72 | ; | 22 | |  | 73 | o | 23 | |  | 74 | i | 23 | 10 |  | 75 | ; | 23 | |  | 76 | o | 24 | |  | 77 | i | 24 | 7 |  | 78 | ; | 24 | |  | 79 | ^ | 25 | |  | 80 | ; | 25 | |  | 81 | n | 27 | |  | 82 | t | 27 | |  | 83 | i | 27 | 12 |  | 84 | = | 27 | |  | 85 | l | 27 | 13 |  | 86 | ; | 27 | |  | 87 | n | 28 | |  | 88 | t | 28 | |  | 89 | i | 28 | 14 |  | 90 | = | 28 | |  | 91 | l | 28 | 8 |  | 92 | ; | 28 | |  | 93 | n | 29 | |  | 94 | t | 29 | |  | 95 | i | 29 | 15 |  | 96 | ; | 29 | |  | 97 | i | 30 | 15 |  | 98 | = | 30 | |  | 99 | i | 30 | 12 |  |100 | \* | 30 | |  |101 | i | 30 | 14 |  |102 | ; | 30 | |  |103 | o | 31 | |  |104 | l | 31 | 16 |  |105 | ; | 31 | |  |106 | o | 32 | |  |107 | i | 32 | 15 |  |108 | ; | 32 | |  |109 | ^ | 33 | |  |110 | ; | 33 | |  |111 | n | 35 | |  |112 | t | 35 | |  |113 | i | 35 | 17 |  |114 | ; | 35 | |  |115 | i | 36 | 17 |  |116 | = | 36 | |  |117 | l | 36 | 13 |  |118 | & | 36 | |  |119 | l | 36 | 18 |  |120 | ; | 36 | |  |121 | o | 37 | |  |122 | l | 37 | 19 |  |123 | ; | 37 | |  |124 | o | 38 | |  |125 | i | 38 | 17 |  |126 | ; | 38 | |  |127 | ^ | 39 | |  |128 | ; | 39 | |  |129 | n | 41 | |  |130 | t | 41 | |  |131 | i | 41 | 20 |  |132 | ; | 41 | |  |133 | i | 42 | 20 |  |134 | = | 42 | |  |135 | ! | 42 | |  |136 | l | 42 | 21 |  |137 | ; | 42 | |  |138 | o | 43 | |  |139 | l | 43 | 22 |  |140 | ; | 43 | |  |141 | o | 44 | |  |142 | i | 44 | 20 |  |143 | ; | 44 | |  |144 | ^ | 45 | |  |145 | ; | 45 | |  |146 | n | 47 | |  |147 | t | 47 | |  |148 | i | 47 | 23 |  |149 | = | 47 | |  |150 | l | 47 | 24 |  |151 | ; | 47 | |  |152 | n | 48 | |  |153 | t | 48 | |  |154 | i | 48 | 25 |  |155 | = | 48 | |  |156 | l | 48 | 24 |  |157 | ; | 48 | |  |158 | n | 49 | |  |159 | t | 49 | |  |160 | i | 49 | 26 |  |161 | = | 49 | |  |162 | l | 49 | 27 |  |163 | ; | 49 | |  |164 | n | 50 | |  |165 | t | 50 | |  |166 | i | 50 | 28 |  |167 | = | 50 | |  |168 | l | 50 | 29 |  |169 | ; | 50 | |  |170 | n | 51 | |  |171 | t | 51 | |  |172 | i | 51 | 30 |  |173 | ; | 51 | |  |174 | o | 53 | |  |175 | l | 53 | 31 |  |176 | ; | 53 | |  |177 | i | 54 | 30 |  |178 | = | 54 | |  |179 | i | 54 | 32 |  |180 | ( | 54 | |  |181 | i | 54 | 23 |  |182 | , | 54 | |  |183 | i | 54 | 25 |  |184 | ) | 54 | |  |185 | ; | 54 | |  |186 | o | 55 | |  |187 | i | 55 | 30 |  |188 | ; | 55 | |  |189 | ^ | 56 | |  |190 | ; | 56 | |  |191 | o | 58 | |  |192 | l | 58 | 33 |  |193 | ; | 58 | |  |194 | i | 59 | 30 |  |195 | = | 59 | |  |196 | i | 59 | 32 |  |197 | ( | 59 | |  |198 | i | 59 | 23 |  |199 | , | 59 | |  |200 | i | 59 | 26 |  |201 | ) | 59 | |  |202 | ; | 59 | |  |203 | o | 60 | |  |204 | i | 60 | 30 |  |205 | ; | 60 | |  |206 | ^ | 61 | |  |207 | ; | 61 | |  |208 | o | 63 | |  |209 | l | 63 | 34 |  |210 | ; | 63 | |  |211 | i | 64 | 30 |  |212 | = | 64 | |  |213 | i | 64 | 32 |  |214 | ( | 64 | |  |215 | i | 64 | 23 |  |216 | , | 64 | |  |217 | i | 64 | 28 |  |218 | ) | 64 | |  |219 | ; | 64 | |  |220 | o | 65 | |  |221 | i | 65 | 30 |  |222 | ; | 65 | |  |223 | ^ | 66 | |  |224 | ; | 66 | |  |225 | i | 68 | 30 |  |226 | = | 68 | |  |227 | i | 68 | 35 |  |228 | ( | 68 | |  |229 | l | 68 | 36 |  |230 | ) | 68 | |  |231 | ; | 68 | |  |232 | o | 69 | |  |233 | l | 69 | 37 |  |234 | ; | 69 | |  |235 | o | 70 | |  |236 | i | 70 | 30 |  |237 | ; | 70 | |  |238 | ^ | 71 | |  |239 | ; | 71 | |  |240 | ^ | 73 | |  |241 | ; | 73 | |  |242 | } | 74 | | |

Таблица идентификаторов:

|  |
| --- |
| | N |СТРОКА В ТЛ| ТИП ИДЕНТИФИКАТОРА | ИМЯ | ЗНАЧЕНИЕ (ПАРАМЕТРЫ)  | 0 | 2 | number function | max | P0:NUMBER | P1:NUMBER |  | 1 | 5 | number parameter | maxa |  | 2 | 8 | number parameter | maxb |  | 3 | 13 | number variable | maxres |0  | 4 | 15 | number literal | LTRL1 |0  | 5 | 45 | number variable | maina |0  | 6 | 47 | number literal | LTRL2 |15  | 7 | 56 | number variable | mainc |0  | 8 | 62 | number literal | LTRL3 |2  | 9 | 64 | number literal | LTRL4 |5  | 10 | 69 | string variable | mainstr |[0]  | 11 | 71 | string literal | LTRL5 |[11]max(2, 5):  | 12 | 83 | number variable | mainaa |0  | 13 | 85 | number literal | LTRL6 |4  | 14 | 89 | number variable | mainab |0  | 15 | 95 | number variable | maincc |0  | 16 | 104 | string literal | LTRL7 |[8]4 or 2:  | 17 | 113 | number variable | maincd |0  | 18 | 119 | number literal | LTRL8 |6  | 19 | 122 | string literal | LTRL9 |[9]4 and 6:  | 20 | 131 | number variable | maince |0  | 21 | 136 | number literal | LTRL10 |3  | 22 | 139 | string literal | LTRL11 |[7]not 3:  | 23 | 148 | string variable | mainstra |[0]  | 24 | 150 | string literal | LTRL12 |[3]ded  | 25 | 154 | string variable | mainstrb |[0]  | 26 | 160 | string variable | mainstrc |[0]  | 27 | 162 | string literal | LTRL13 |[4]deed  | 28 | 166 | string variable | mainstrd |[0]  | 29 | 168 | string literal | LTRL14 |[3]deb  | 30 | 172 | number variable | mainresaa |0  | 31 | 175 | string literal | LTRL15 |[18]strcmp(ded, ded):  | 32 | 179 | number LIB FUNC | strcmpp | P0:STRING | P1:STRING |  | 33 | 192 | string literal | LTRL16 |[19]strcmp(ded, deed):  | 34 | 209 | string literal | LTRL17 |[19]strcmp(ded, deeb):  | 35 | 227 | number LIB FUNC | lenght | P0:STRING |  | 36 | 229 | string literal | LTRL18 |[6]123456  | 37 | 233 | string literal | LTRL19 |[16]lenght(123456): |

**Приложение В**

|  |
| --- |
| Rule(NS('S'), GRB\_ERROR\_SERIES, 3, // Неверная структура программы  Rule::Chain(6, TS('t'), TS('f'), TS('i'), NS('P'), NS('T'), NS('S')),  Rule::Chain(6, TS('p'), TS('f'), TS('i'), NS('P'), NS('G'), NS('S')),  Rule::Chain(4, TS('m'), TS('{'), NS('K'), TS('}'))  ),  Rule(NS('T'), GRB\_ERROR\_SERIES + 2, 2, // Ошибка в теле функции  Rule::Chain(5, TS('{'), TS('e'), NS('V'), TS(';'), TS('}')),  Rule::Chain(6, TS('{'), NS('K'), TS('e'), NS('V'), TS(';'), TS('}'))  ),  Rule(NS('G'), GRB\_ERROR\_SERIES + 3, 2, // Ошибка в теле скрипта  Rule::Chain(4, TS('{'), TS('e'), TS(';'), TS('}')),  Rule::Chain(5, TS('{'), NS('K'), TS('e'), TS(';'), TS('}'))  ),…………………………  Rule(NS('X'), GRB\_ERROR\_SERIES + 15, 8, // Недопустимая синтаксическая конструкция в теле условного выражения  Rule::Chain(5, TS('i'), TS('='), NS('W'), TS(';'), NS('X')), // присваивание  Rule::Chain(4, TS('o'), NS('V'), TS(';'), NS('X')), // вывод  Rule::Chain(3, TS('^'), TS(';'), NS('X')), // перевод строки  Rule::Chain(4, TS('i'), NS('F'), TS(';'), NS('X')), // вызов функции  Rule::Chain(4, TS('i'), TS('='), NS('W'), TS(';')), // присваивание  Rule::Chain(3, TS('o'), NS('V'), TS(';')), // вывод  Rule::Chain(2, TS('^'), TS(';')), // перевод строки  Rule::Chain(3, TS('i'), NS('F'), TS(';')) // вызов функции  )  ); |

**Приложение Г**

Файл Greibach.h:

|  |
| --- |
| namespace GRB  {  struct Rule // правило в грамматике Грейбах  {  GRBALPHABET nn; // нетерминал (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 нетерминалов (<0)  Chain() { size = 0; nt = 0; };  Chain(  short psize, // количество символов в цепочке  GRBALPHABET s, ... // символы (терминал или нетерминал)  );  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); }; // нетерминал  static bool isT(GRBALPHABET s) { return s > 0; } // терминал?  static bool isN(GRBALPHABET s) { return !isT(s); } // нетерминал?  static char alphabet\_to\_char(GRBALPHABET s) { return isT(s) ? char(s) : char(-s); };//GRBALPHABET->char  }\*chains; //массив цепочек - правых частей правила  Rule() { nn = 0x00; size = 0; }  Rule(  GRBALPHABET pnn, // нетерминал (<0)  int iderror, // идентификатор диагностического сообщения  short psize, // количество цепочек - правых частей правила  Chain c, ... // множество цепочек - правых частей правила  );  char\* getCRule( // получить правило в виде N->цепочка(для распечатки)  char\* b, // буфер  short nchain // номер цепочки(правой части) в правиле  );  short getNextChain( // получить следующую за j подходящую цепочку, вернуть её номер или -1  GRBALPHABET t, // первый символ цепочки  Rule::Chain& pchain, // возвращаемая цепочка  short j // номер цепочки  );  };  struct Greibach // грамматика Грейбах  {  short size; // количество правил  GRBALPHABET startN; // стартовый символ  GRBALPHABET stbottomT; // дно стека  Rule\* rules; // множество правил  Greibach() { short size = 0; startN = 0; stbottomT = 0; rules = 0; };  Greibach(  GRBALPHABET pstartN, // стартовый символ  GRBALPHABET pstbootomT, // дно стека  short psize, // количество правил  Rule r, ... // правила  );  short getRule( // получить правило, возвращается номер правила или -1  GRBALPHABET pnn, // левый символ правила  Rule& prule // возвращаемое правило грамматики  );  Rule getRule(short n); // получить правило по номеру  };  Greibach getGreibach(); // получить грамматику  }; |

**Приложение Д**

Файл MFST.h:

|  |
| --- |
| #pragma once  #include "pch.h"  #include "MFST.h"  #include "Greibach.h"  #define MFST\_DIAGN\_MAXSIZE 2\*ERROR\_MAXSIZE\_MESSAGE  #define MFST\_DIAGN\_NUMBER 3  typedef std::stack <short> MFSTSTSTACK; // стек автомата  namespace MFST  {  struct MfstState // состояние автомата(для сохранения)  {  short lenta\_position; // позиция на ленте  short nrule; // номер текущего правила  short nrulechain; // номер текущей цепочки  MFSTSTSTACK st; // стек автомата  MfstState();  MfstState(  short pposition, // позиция на ленте  MFSTSTSTACK pst, // стек автомата  short pnrulechain // номер текущей цепочки, текущего правила  );  MfstState(  short pposition, // позиция на ленте  MFSTSTSTACK pst, // стек автомата  short pnrule, // номер текущего правила  short pnrulechain // номер текущей цепочки, текущего правила  );  };  struct Mfst // магазинный автомат  {  enum RC\_STEP // шаг автомата  {  NS\_OK, // найдено правило и цепочка, цепочка записана в стек  NS\_NORULE, // не найдено правило грамматики (ошибки в грамматике)  NS\_NORULECHAIN, // не найдена подходящая цепочка правила (ошибка в исходном коде)  NS\_ERROR, // неизвестный нетерминальный символ грамматики  TS\_OK, // текущий символ ленты == вершине стека, продвинулась лента, рор стека  TS\_NOK, // текущий символ ленты != вершине стека, восстановлено состояние  LENTA\_END, // текущая позиция ленты >= lenta\_size  SURPRISE // неожиданный код возврата ( ошибка в step)  };  struct MfstDiagnosis // диагностика  {  short lenta\_position; // позиция на ленте  RC\_STEP rc\_step; // код завершения шага  short nrule; // номер правила  short nrule\_chain; // номер цепочки правила  MfstDiagnosis();  MfstDiagnosis( // диагностика  short plenta\_position, // позиция на ленте  RC\_STEP prc\_step, // код завершения шага  short pnrule, // номер правила  short pnrule\_chain // номер цепочки правила  );  }  diagnosis[MFST\_DIAGN\_NUMBER]; // последние самые глубокие сообщения  GRBALPHABET\* lenta; // перекодированная(TS/NS) лента (из LEX)  short lenta\_position; // текущая позиция на ленте  short nrule; // номер текущего правила  short nrulechain; // номер текущей цепочки, текущего правила  short lenta\_size; // размер ленты  GRB::Greibach grebach; // грамматика Грейбах  Lexer::LEX lex; // результат работы лексического анализатора  MFSTSTSTACK st; // стек автомата  std::stack<MfstState> storestate; // стек для сохранения состояний  Mfst();  Mfst(  Lexer::LEX plex, // результат работы лексического анализатора  GRB::Greibach pgrebach // грамматика Грейбах  );  char\* getCSt(char\* buf); // получить содержимое стека  char\* getCLenta(char\* buf, short pos, short n = 25);// лента: n символов с pos  char\* getDiagnosis(short n, char\* buf); // получить n-ую строку диагностики или 0х00  bool savestate(const Log::LOG& log); // сохранить состояние автомата  bool reststate(const Log::LOG& log); // восстановить состояние автомата  bool push\_chain( // поместить уепочку правила в стек  GRB::Rule::Chain chain // цепочка правила  );  RC\_STEP step(const Log::LOG& log); // выполнить шаг автомата  bool start(const Log::LOG& log); // запустить автомат  bool savediagnois(  RC\_STEP pprc\_step // код завершения шага  );  void printrules(const Log::LOG& log); // вывести последовательность правил  struct Deducation // вывод  {  short size; // количество шагов в выводе  short\* nrules; // номера правил грамматики  short\* nrulechains;// номера цепочек правил грамматики (nrules)  Deducation() { size = 0; nrules = 0; nrulechains = 0; };  } deducation;  bool savededucation(); // сохранить дерево вывода  };  }; |

**Приложение Е**

Разбор синтаксическим анализатором:

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

**Приложение Ж**

Файл PolishNotation.cpp:

|  |
| --- |
| #include "pch.h"  #include "Header.h"  #include <cstring>  using namespace std;  namespace Polish  {  int getPriority(LT::Entry& e)  {  switch (e.lexema) // ( ) \* / ! % + - { }  {  case LEX\_LEFTHESIS:  case LEX\_RIGHTTHESIS:  case LEX\_NOT: return 0;  case LEX\_PLUS:  case LEX\_OR:  case LEX\_AND:  case LEX\_MINUS: return 1;  case LEX\_MOD: return 2;  case LEX\_LEFT:  case LEX\_RIGHT: return 3;  default: return -1;  }  }  bool PolishNotation(Lexer::LEX& tbls, Log::LOG& log)  {  unsigned curExprBegin = 0;  ltvec v; // вектор таблицы лексем  LT::LexTable new\_table = LT::Create(tbls.lextable.maxsize);  intvec vpositions = getExprPositions(tbls); // начало выражений  for (int i = 0; i < tbls.lextable.size; i++)  {  if (curExprBegin < vpositions.size() && i == vpositions[curExprBegin]) // дошли до следующего начала выражения  {  int lexcount = fillVector(vpositions[curExprBegin], tbls.lextable, v); // заполняем вектор строками ТЛ в прямом порядке  if (lexcount > 1)  {  bool rc = setPolishNotation(tbls.idtable, log, vpositions[curExprBegin], v); // получаем вектор co строками ТЛ в польской нотации  if (!rc)  return false;  }  addToTable(new\_table, tbls.idtable, v); // добавляем постpоенный вектор к ТЛ + корректировка ТИ  i += lexcount - 1;  curExprBegin++;  continue;  }  if (tbls.lextable.table[i].lexema == LEX\_ID || tbls.lextable.table[i].lexema == LEX\_LITERAL) // если идентификатор или литерал  {  int firstind = Lexer::getIndexInLT(new\_table, tbls.lextable.table[i].idxTI);  if (firstind == -1)  firstind = new\_table.size;  tbls.idtable.table[tbls.lextable.table[i].idxTI].idxfirstLE = firstind;  }  LT::Add(new\_table, tbls.lextable.table[i]); // добавляем в таблицу лексем  }  tbls.lextable = new\_table;  return true;  }  int \_\_cdecl fillVector(int posExprBegin, LT::LexTable& lextable, ltvec& v) // заполнение вектора  {  v.clear();  for (int i = posExprBegin; i < lextable.size; i++)  {  if (lextable.table[i].lexema == LEX\_SEPARATOR)  break;  else  v.push\_back(LT::Entry(lextable.table[i]));  }  return v.size();  }  void addToTable(LT::LexTable& new\_table, IT::IdTable& idtable, ltvec& v) // добавление в таблицу лексем и идентификаторов  {  for (unsigned i = 0; i < v.size(); i++)  {  LT::Add(new\_table, v[i]);  if (v[i].lexema == LEX\_ID || v[i].lexema == LEX\_LITERAL) // обратная свзяь между ТИ и ТЛ  {  int firstind = Lexer::getIndexInLT(new\_table, v[i].idxTI);  idtable.table[v[i].idxTI].idxfirstLE = firstind;  }  }  }  intvec getExprPositions(Lexer::LEX& tbls) // найти позиции выражени  {  intvec v;  bool f\_begin = false; // признак найденного конца выражения  bool f\_end = false; // признак найденного начала выражения  int begin = 0; int end = 0;  for (int i = 0; i < tbls.lextable.size; i++)  {  if (tbls.lextable.table[i].lexema == LEX\_EQUAL) // начало выражения  {  begin = i + 1;  f\_begin = true;  continue;  }  if (f\_begin && tbls.lextable.table[i].lexema == LEX\_SEPARATOR) // конец выражения  {  end = i;  f\_end = true;  continue;  }  if (f\_begin && f\_end) // добавить начало и конец выражения в вектор  {  v.push\_back(begin);  f\_begin = f\_end = false;  }  }  return v;  }  bool \_\_cdecl setPolishNotation(IT::IdTable& idtable, Log::LOG& log, int lextable\_pos, ltvec& v)  {  vector < LT::Entry > result; //результирующий вектор  stack < LT::Entry > s; // стек для сохранения операторов  bool ignore = false; // флаг вызова функции  for (unsigned i = 0; i < v.size(); i++)  {  if (ignore) // вызов функции считаем подставляемым значением и заносим в результат  {  result.push\_back(v[i]);  if (v[i].lexema == LEX\_RIGHTTHESIS)  ignore = false;  continue;  }  int priority = getPriority(v[i]); // его приоритет  if (v[i].lexema == LEX\_LEFTHESIS || v[i].lexema == LEX\_RIGHTTHESIS || v[i].lexema == LEX\_PLUS || v[i].lexema == LEX\_MINUS || v[i].lexema  == LEX\_OR || v[i].lexema == LEX\_AND || v[i].lexema == LEX\_LEFT || v[i].lexema == LEX\_RIGHT || v[i].lexema == LEX\_MOD || v[i].lexema == LEX\_NOT)  {  if (s.empty() || v[i].lexema == LEX\_LEFTHESIS)  {  s.push(v[i]);  continue;  }  if (v[i].lexema == LEX\_RIGHTTHESIS)  {  while (!s.empty() && s.top().lexema != LEX\_LEFTHESIS) //выталкивание элементов до скобки  {  result.push\_back(s.top());  s.pop();  }  if (!s.empty() && s.top().lexema == LEX\_LEFTHESIS)  s.pop();  continue;  }  while (!s.empty() && getPriority(s.top()) >= priority) //выталкивание элем с большим/равным приоритетом в результат  {  result.push\_back(s.top());  s.pop();  }  s.push(v[i]);  }  if (v[i].lexema == LEX\_LITERAL || v[i].lexema == LEX\_ID) // идентификатор, идентификатор функции или литерал  {  if (idtable.table[v[i].idxTI].idtype == IT::IDTYPE::F || idtable.table[v[i].idxTI].idtype == IT::IDTYPE::S)  ignore = true;  result.push\_back(v[i]); // операнд заносим в результирующий вектор  }  if (v[i].lexema != LEX\_LEFTHESIS && v[i].lexema != LEX\_RIGHTTHESIS && v[i].lexema != LEX\_PLUS && v[i].lexema  != LEX\_MINUS && v[i].lexema != LEX\_OR && v[i].lexema != LEX\_AND && v[i].lexema != LEX\_ID && v[i].lexema  != LEX\_LITERAL && v[i].lexema != LEX\_LEFT && v[i].lexema != LEX\_RIGHT && v[i].lexema != LEX\_MOD && v[i].lexema != LEX\_NOT)  {  Log::writeError(log.stream, Error::GetError(1));  return false;  }  }  while (!s.empty()) { result.push\_back(s.top()); s.pop(); }  v = result;  return true;  }  } |

**Приложение З**

Файл AssembledCode.asm :

|  |
| --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib "../Debug/GenLib.lib  ExitProcess PROTO:DWORD  .stack 4096  outnum PROTO : DWORD  outstr PROTO : DWORD  strcopy PROTO : DWORD, : DWORD  lenght PROTO : DWORD, : DWORD  atoii PROTO : DWORD, : DWORD  .const  newline byte 13, 10, 0  LTRL1 sdword 10  LTRL2 sdword 20  LTRL3 byte 'min(10, 20) + 10 = ', 0  LTRL4 byte '123', 0  LTRL5 byte 'string -> int: ', 0  LTRL6 byte 'cycle: ', 0  LTRL7 sdword 1  LTRL8 byte '; ', 0  LTRL9 sdword 3  .data  temp sdword ?  buffer byte 256 dup(0)  minres sdword 0  mainaa sdword 0  mainab sdword 0  mainac sdword 0  mainnumb dword ?  mainad sdword 0  .code  ;------------- min --------------  min PROC,  mina : sdword, minb : sdword  ; -------- save registers -------  push ebx  push edx  ; -------------------------------  mov edx, mina  cmp edx, minb  jg right1  jl wrong1  right1:  push minb  pop ebx  mov minres, ebx  jmp next1  wrong1:  push mina  pop ebx  mov minres, ebx  next1:  ; ------ restore registers ------  pop edx  pop ebx  ; -------------------------------  mov eax, minres  ret  min ENDP  ;---------------  ;------------- MAIN --------------  main PROC  push LTRL1  pop ebx  mov mainaa, ebx  push LTRL2  pop ebx  mov mainab, ebx  push mainab  push mainaa  call min  push eax  push LTRL1  pop ebx  pop eax  add eax, ebx  push eax  pop ebx  mov mainac, ebx  push offset LTRL3  call outstr  push mainac  call outnum  push offset newline  call outstr  mov mainnumb, offset LTRL4  push mainnumb  push offset buffer  call atoii  push eax  pop ebx  mov mainad, ebx  push offset LTRL5  call outstr  push mainad  call outnum  push offset newline  call outstr  push offset LTRL6  call outstr  mov edx, mainad  cmp edx, LTRL7  jg cycle2  jmp cyclenext2  cycle2:  push mainad  call outnum  push offset LTRL8  call outstr  push mainad  push LTRL9  pop ebx  pop eax  cdq  idiv ebx  push eax  pop ebx  mov mainad, ebx  mov edx, mainad  cmp edx, LTRL7  jg cycle2  cyclenext2:  push offset newline  call outstr  push offset newline  call outstr  push 0  call ExitProcess  main ENDP  end main |

**Приложение И**

Результат преобразования выражений:

|  |
| --- |
| 1 | tfi[0](ti[1],ti[2])  2 | [  3 | nti[3];  4 | ?i[1]>i[2]#  5 | w  6 | [  7 | i[3]=i[2];  8 | ]  9 | r  10 | [  11 | i[3]=i[1];  12 | ]#  13 | ei[3];  14 | ]  16 | m  17 | [  18 | nti[4]=l[5];  19 | nti[6]=l[7];  20 | nti[8];  21 | i[8]=i[0](i[4],i[6])l[5]+;  22 | ol[9];  23 | oi[8];  24 | ^;  26 | nti[10]=l[11];  27 | nti[12];  28 | i[12]=i[13](i[10]);  29 | ol[14];  30 | oi[12];  31 | ^;  33 | ol[15];  34 | ?i[12]>l[16]#  35 | c  36 | [  37 | oi[12];  38 | ol[17];  39 | i[12]=i[12]l[18]/;  40 | ]#  41 | ^;  45 | ^;  46 | ] |