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 SMD-2024»

Выполнил студент Шинкевич Марина Дмитриевна

(Ф.И.О.)

Руководитель проекта Волчек Дарья Ивановна

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

Консультанты

Нормоконтролер

Курсовой проект защищен с оценкой

Минск 2024

**Содержание**

[Введение 4](#_Toc184848060)

[1. Спецификация языка программирования SMD-2024 5](#_Toc184848061)

[1.1 Характеристика языка программирования 5](#_Toc184848062)

[1.2 Определение алфавита языка программирования 5](#_Toc184848063)

[1.3 Применяемые сепараторы 5](#_Toc184848064)

[1.4 Применяемые кодировки 5](#_Toc184848065)

[1.5 Типы данных 6](#_Toc184848066)

[1.6 Преобразование типов данных 7](#_Toc184848067)

[1.7. Идентификаторы 7](#_Toc184848068)

[1.8. Литералы 7](#_Toc184848069)

[1.9. Объявление данных 7](#_Toc184848070)

[1.10. Инициализация данных 8](#_Toc184848071)

[1.11. Инструкции языка 8](#_Toc184848072)

[1.12. Операции языка 9](#_Toc184848073)

[1.13. Выражения и их вычисление 9](#_Toc184848074)

[1.14. Конструкции языка 9](#_Toc184848075)

[1.15. Область видимости идентификаторов 10](#_Toc184848076)

[1.16. Семантические проверки 10](#_Toc184848077)

[1.17. Распределение оперативной памяти на этапе выполнения. 11](#_Toc184848078)

[1.19. Ввод и вывод данных 11](#_Toc184848079)

[1.20. Точка входа 12](#_Toc184848080)

[1.21. Препроцессор 12](#_Toc184848081)

[1.22. Соглашения о вызовах 12](#_Toc184848082)

[1.23. Объектный код 12](#_Toc184848083)

[1.24. Классификация сообщений транслятора 12](#_Toc184848084)

[1.25. Контрольный пример 12](#_Toc184848085)

[2. Структура транслятора 13](#_Toc184848086)

[2.1. Компоненты транслятора их назначение и принципы взаимодействия 13](#_Toc184848087)

[2.2. Перечень входных параметров транслятора 14](#_Toc184848088)

[2.3 Протоколы, формируемые транслятором 14](#_Toc184848089)

[4. Разработка синтаксического анализатора 21](#_Toc184848090)

[4.1. Структура синтаксического анализатора 21](#_Toc184848091)

[4.2. Контекстно-свободная грамматика, описывающая синтаксис языка 21](#_Toc184848092)

[4.3. Построение конечного магазинного автомата 23](#_Toc184848093)

[4.4. Основные структуры данных 24](#_Toc184848094)

[4.5. Описание алгоритма синтаксического разбора 24](#_Toc184848095)

[4.6. Структура и перечень сообщений синтаксического анализатора 25](#_Toc184848096)

[4.7. Параметры синтаксического анализатора и режимы его работы 26](#_Toc184848097)

[4.8. Принцип обработки ошибок 26](#_Toc184848098)

[4.9. Контрольный пример 26](#_Toc184848099)

[5. Разработка семантического анализатора 27](#_Toc184848100)

[5.1. Структура семантического анализатора 27](#_Toc184848101)

[5.2. Функции семантического анализатора 27](#_Toc184848102)

[5.3. Структура и перечень сообщений семантического анализатора 29](#_Toc184848103)

[5.4. Принцип обработки ошибок 30](#_Toc184848104)

[6. Вычисление выражений 32](#_Toc184848105)

[6.1. Выражения, допускаемые языком 32](#_Toc184848106)

[6.2. Польская запись и принцип ее построения 32](#_Toc184848107)

[6.3. Программная реализация обработки выражений 33](#_Toc184848108)

[6.4. Контрольный пример 34](#_Toc184848109)

[7. Генерация кода 35](#_Toc184848110)

[7.1. Структура генератора кода 35](#_Toc184848111)

[7.2. Представление типов данных в оперативной памяти 35](#_Toc184848112)

[7.3. Статическая библиотека 36](#_Toc184848113)

[7.4. Особенности алгоритма генерации кода 36](#_Toc184848114)

[7.5. Входные параметры, управляющие генерацией кода 36](#_Toc184848115)

[7.6. Контрольный пример 37](#_Toc184848116)

[8. Тестирование транслятора 38](#_Toc184848117)

[8.1. Общие положения 38](#_Toc184848118)

[8.2. Результаты тестирования 38](#_Toc184848119)

[Заключение 41](#_Toc184848120)

[Список использованных источников 42](#_Toc184848121)

[ПРИЛОЖЕНИЕ A 43](#_Toc184848122)

[ПРИЛОЖЕНИЕ В 45](#_Toc184848123)

[ПРИЛОЖЕНИЕ Г 53](#_Toc184848124)

[ПРИЛОЖЕНИЕ Д 55](#_Toc184848125)

[ПРИЛОЖЕНИЕ Е 57](#_Toc184848126)

[ПРИЛОЖЕНИЕ Ж 60](#_Toc184848127)

# **Введение**

Цель курсового проекта — создание транслятора для разработанного языка программирования SMD-2024, предназначенного для решения базовых задач. Транслятор для языка SMD-2024 включает следующие компоненты:

* лексический анализатор;
* синтаксический анализатор;
* семантический анализатор;
* преобразователь выражений в обратную польскую запись;
* генератор исходного кода на языке ассемблера.

Основные задачи курсового проекта включают в себя:

* разработка спецификации языка программирования SMD-2024;
* проектирование структуры компилятора;
* создание лексического анализатора;
* создание семантического анализатора;
* создание синтаксического анализатора;
* разработать преобразователь выражений в обратную польскую запись;
* генерация кода на языке ассемблера;
* тестирование компилятора.

Решение каждой задачи представлено в соответствующих главах проекта:

Первая глава описывает спецификацию языка, включая синтаксис и семантику.

Вторая глава описывает общую структуру транслятора.

Третья глава посвящена разработке лексического анализатора, который создает таблицы лексем и идентификаторов.

В четвертой главе изложена работа синтаксического анализатора, который выполняет разбор текста и генерирует протокол.

Пятая глава описывает семантический анализатор.

Шестая глава отображает результаты преобразования выражений в обратную польскую запись.

В седьмой главе рассматривается генерация кода на ассемблере.

Восьмая глава включает тестирование компилятора и описание различных исключительных ситуаций.

# **1. Спецификация языка программирования SMD-2024**

## **1.1 Характеристика языка программирования**

Язык программирования SMD-2024 – это универсальный язык высокого уровня. Он является компилируемым, процедурным языком со строгой типизацией, а также не объектно-ориентированным.

## **1.2 Определение алфавита языка программирования**

В языке SMD-2024 допустимы латинские буквы, цифры (0-9), а также символы-сепараторы, непечатные символы, такие как пробелы, переводы строки и табуляции. Русские символы можно использовать исключительно внутри строковых литералов.

## **1.3 Применяемые сепараторы**

Символы-сепараторы выполняют роль разделителей операций в языке. В таблице 1.1 перечислены сепараторы, используемые в языке программирования SMD-2024.

Таблица 1.1 — Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Область применения |
| ‘ ‘ | Допускается везде, кроме названий идентификаторов и ключевых слов |
| ; | Разделитель программных конструкций |
| {…} | Блок функции |
| = | Оператор присваивания |
| “…” | Строковый литерал |
| (…) | Блок параметров функции, а также приоритет арифметических операций |
| , | Разделитель параметров функций |
| +,-,\*,/ ,\,: | Арифметические операции |
| […] | Блок условной конструкции/цикла |
| > < | Используются в условиях цикла/условной конструкции. |

## **1.4 Применяемые кодировки**

В языке SMD-2024 используется 8-битная кодировка Windows-1251, представлена на рисунке 1.1.

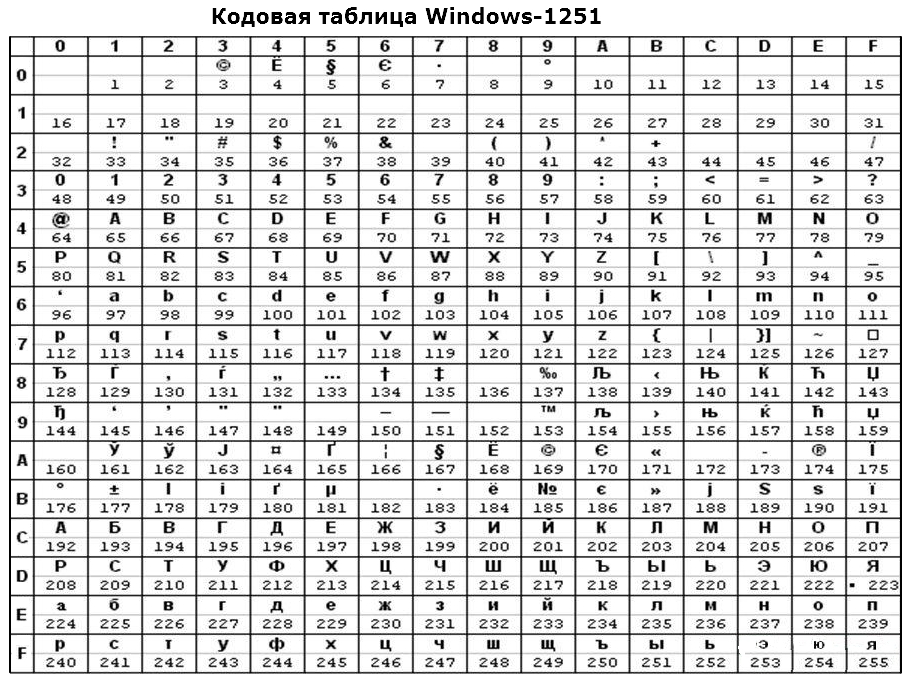


Рисунок 1.1 - Кодировка Windows 1251

## **1.5 Типы данных**

В языке SMD-2024 предусмотрены типы данных, представленные в таблице 1.2.

Таблица 1.2 – Типы данных языка SMD-2024

|  |  |  |
| --- | --- | --- |
| Название типа | Описание | Диапазон значений |
| IntNum | Целочисленный беззнаковый тип данных (1 байт). | от 0 до 255. |
| Symb | Символьный тип данных (1 байт). | Один разрешенный символ |
| Line | Используется для работы с символами, каждый символ в памяти занимает 1 байт.  Максимальное количество символов: 255. | От 1 до 255 символов |

## **1.6 Преобразование типов данных**

Преобразование типов данных в языке SMD-2024 не предусматривается, т.е. язык является строго типизированным.

**1.7. Идентификаторы**

Имена идентификаторов не могут совпадать с ключевыми словами языка SMD-2024. Запрещается использовать идентификаторы длиной более 15 символов. При использовании идентификаторов большей длины, они будут усекаться. В имени идентификатора допускаются только символы латинского алфавита и цифры.

**1.8. Литералы**

Литерал – это значение, которым инициализируется переменная в языке программирования. Литералы, которые доступны в языке SMD-2024 представлены в таблице 1.3.

Таблица 1.3 – Литералы

|  |  |  |
| --- | --- | --- |
| Тип литерала | Описание | Пример объявления |
| Целочисленный литерал | Целочисленные неотрицательные литералы, по умолчанию инициализируются 0. | Def IntNum numb;  numb =10;  10 – целочисленный литерал. |
| Символьный литерал | Символ, заключённый в ‘’ (одинарные кавычки), по умолчанию инициализируются пустой строкой. | Def Symb symbol;  symbol = ‘M’;  M – символьный литерал. |
| Строковый литерал | Строка, заключённая в “”(двойные кавычки), по умолчанию инициализируются пустой строкой. | Def Line str;  str = “String”;  String – строка . |

**1.9. Объявление данных**

Переменные и другие элементы в программе видны и доступны для использования только начиная с того места, где они объявлены. Если встретится использование переменной или функции до их объявления, это вызовет ошибку. Все переменные необходимо объявлять внутри определенного программного блока. Поскольку переменные, созданные в одной функции, недоступны в другой, возможно использовать одинаковые имена переменных в разных блоках.

Синтаксис:

Def <тип данных> <идентификатор> – объявление переменной

Def <тип данных> Func <идентификатор> (<тип данных> <идентификатор>, <тип данных> <идентификатор>) – объявление функции

{

<блок кода>

}

**1.10. Инициализация данных**

Инициализация переменной в языке SMD-2024 в момент объявления не предусмотрена. Переменные инициализируются после их объявления. При объявлении без инициализации предусмотрены значения по умолчанию: 0 для беззнакового целочисленного типа данных, “” (пустая строка) для строкового типа данных, ‘’ (пустая строка) для символьного типа данных.

Примеры объявления и инициализации:

Def Symb symbol;

symbol = ‘M’;

Def IntNum numb;

numb =10;

Def Line str;

str = “String”;

**1.11. Инструкции языка**

Инструкции языка SMD-2024 представлены в таблице 1.4.

Таблица 1.4 – Инструкции языка SMD-2024.

|  |  |
| --- | --- |
| Инструкция | Запись |
| Возврат из подпрограммы | Get <идентификатор> / <литерал>; |
| Условная инструкция | While(<условие>)[<блок кода>]; |
| Вывод данных | Show<идентификатор> / <литерал>; |
| Однострочный комментарий до конца строки | //<любой текст> |
| Объявление переменной | Def<тип данных> <идентификатор>; |
| Присваивание | <идентификатор> = <значение>/<идентификатор>; |
| Объявление функции | Def <тип данных> Func <идентификатор> (<тип данных> <идентификатор>, …) {<блок кода>}; |
| Внешняя функция | Out <тип данных> Func <идентификатор> (<тип данных> <идентификатор>, …); |
| Точка входа в программу | Main  {  …} |

**1.12. Операции языка**

Операции доступные в языке программирования SMD-2024 представлены в таблице 1.5. Выполнение операций с разными типами данных не допускается. При выполнении операции сдвига учитывается только младший бит оператора, поскольку сдвиг на более чем 255 (за исключением сдвига числа 0) вернет значение, превышающее максимальное для типа данных IntNum.

Таблица 1.5 – Операции языка SMD-2024

|  |  |  |  |
| --- | --- | --- | --- |
| Операция | Примечание | Типы данных | Пример |
| = | Присваивание | (IntNum, IntNum)  (Symb, Symb)  (Line, Line) | sum = 10;  str = “txt”;  sym= ‘g’ |
| < | Знак «меньше» для условной инструкции | (IntNum, IntNum) | While (a < b) […] |
| > | Знак «больше» для условной инструкции | (IntNum, IntNum) | While (b >a) […] |
| ( | Приоритет операций | - | number = (a + b) \* c; |
| ) |
| + | Суммирование | (IntNum, IntNum)  (Symb, Symb) | num= a + b; |
| - | Разность | (IntNum, IntNum)  (Symb, Symb) | diff = a – b; |
| \* | Умножение | (IntNum, IntNum) | mul = a \* b; |
| : | Деление | (IntNum, IntNum) | div = a:b; |
| \ | Сдвиг вправо | (IntNum, IntNum) | sr = a \ b; |
| / | Сдвиг влево | (IntNum, IntNum) | sl = a / b; |
| % | Остаток от деления | (IntNum, IntNum) | d = a%b; |

**1.13. Выражения и их вычисление**

В выражении допускается вызывать функции, которые уже определены в стандартной библиотеке. Вычисление выражений происходит только после выполнения оператора присваивания. Круглые скобки служат для изменения порядка выполнения операций. Нельзя использовать две арифметические операции подряд.

**1.14. Конструкции языка**

Конструкции языка SMD-2024 дают возможность выполнять различные операции и решить базовые задачи. Доступные программные конструкции языка программирования SMD-2024 представлены в таблице 1.6.

Таблица 1.6 – Программные конструкции языка SMD-2024

|  |  |
| --- | --- |
| Конструкция | Запись |
| Точка входа в программу | Main  {  …  Get <идентификатор> / <литерал>;  } |
| Функция | Def <тип данных> Func <идентификатор> (<тип> <идентификатор>, …)  {… Get <идентификатор> / <литерал>; }; |
| Цикл | While(a>b)[ …]; |
| Условный оператор | if(a<b)[…]; |

**1.15. Область видимости идентификаторов**

Организация области видимости схожа с C++. Все идентификаторы должны быть доступны из текущей области видимости, но не предусмотрено объявление глобальных переменных, а также создание пользовательских областей видимости.

**1.16. Семантические проверки**

В языке SMD-2024 предусмотрены следующие семантические проверки:

1. Наличие одной функции Main;
2. Идентификаторы не должны быть объявлены повторно внутри одной функции.;
3. Типы данных операндов в выражении должны совпадать
4. Тип данных результата выражения должен соответствовать типу данных идентификатора, которому он присваивается.
5. Тип данных Line не может использоваться в качестве аргумента в условной конструкции.
6. Функции не должны подключаться более одного раза в рамках одной программы.
7. В функцию должно быть передано то число параметров, сколько ожидается
8. Деление на 0 в выражении.
9. Тип данных передаваемых в функцию значений должен соответствовать типу параметров, указанному при её объявлении или вызове.
10. Использование идентификатора до объявления или без него;
11. Тип возвращаемого значения должен соответствовать типу, указанному для функции при её объявлении или вызове.

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

Переменные размещаются в куче.

**1.18. Стандартная библиотека и ее состав**

Стандартная библиотека языка SMD-2024 включает в себя функции для получения даты и времени. Функции для библиотеки написаны на языке программирования C++. Для использования необходима подключить функцию с помощью ключевого слова Out, после этого можно использовать её как пользовательскую функцию. Функции стандартной библиотеки с описанием представлены в таблице 1.7.

Таблица 1.7 – Стандартная библиотека SMD-2024

|  |  |
| --- | --- |
| Функция | Описание |
| getCurrentDate() | Возвращает дату в формате ДДММГГГГ |
| getCurrentHour () | Возвращает час |
| getCurrentMinuts () | Возвращает минуты |

Так же в библиотеке присутствуют приватные функции. Их описание представлено в таблице 1.8.

Таблица 1.8 – Приватные функции стандартной библиотеки

|  |  |  |
| --- | --- | --- |
| Функция(C++) | Возвращаемое значение | Описание |
| void outputIntNum (unsigned int a) | – | Выводит число на экран Вызывается оператором Show |
| void outputSymb (char a) | – | Выводит символ на экран Вызывается оператором Show |
| void outputLine (void\* in) | – | Выводит строку на экран Вызывается оператором Show |

Приватные функции не могут быть вызваны явно и не требуют предварительного пользовательского подключения. Они вызываются специальными операторами языка.

**1.19. Ввод и вывод данных**

Вывод данных в языке SMD-2024 осуществляется с помощью ключевого слова Show.

Синтаксис:

Show “Any string”;

Show identifier;

Ввод данных для языка SMD-2024 не предусмотрен.

**1.20. Точка входа**

В языке SMD-2024 любая программа начинается с выполнения функции Main, которая является обязательной и служит точкой входа в программу.

**1.21. Препроцессор**

Препроцессор в языке программирования SMD-2024 не предусмотрен.

**1.22. Соглашения о вызовах**

Используется соглашение stdcall, то есть все параметры передаются в стек справа налево, память высвобождает вызываемый код.

**1.23. Объектный код**

Язык программирования SMD-2024 транслируется в ассемблер.

**1.24. Классификация сообщений транслятора**

Если в программе на языке SMD-2024 обнаруживается ошибка, транслятор записывает соответствующее сообщение в текущий протокольный файл. Подробная классификация этих сообщений представлена в таблице 1.9.

Таблица 1.9 – Классификация ошибок

|  |  |
| --- | --- |
| Диапазон ошибок | Описание |
| 0-99 | Системные ошибки. |
| 100-109 | Ошибки параметров |
| 110-119 | Ошибки открытия и чтения файлов |
| 120-129 | Ошибки лексического анализа. |
| 600-610 | Ошибки синтаксического анализа. |
| 700-710 | Ошибки семантического анализа. |

**1.25. Контрольный пример**

Контрольный пример представлен в Приложении А.

# **2. Структура транслятора**

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

**Транслятор** преобразует текст программы, написанный на языке программирования SMD-2024 в программу на языке ассемблера. Выходные файлы задаются с помощью входных параметров транслятора, описанных в разделе 2.2. Основные компоненты транслятора включают лексический, синтаксический и семантический анализаторы, а также генератор кода на языке ассемблера. Схема их взаимодействия показана на рисунке 2.1.

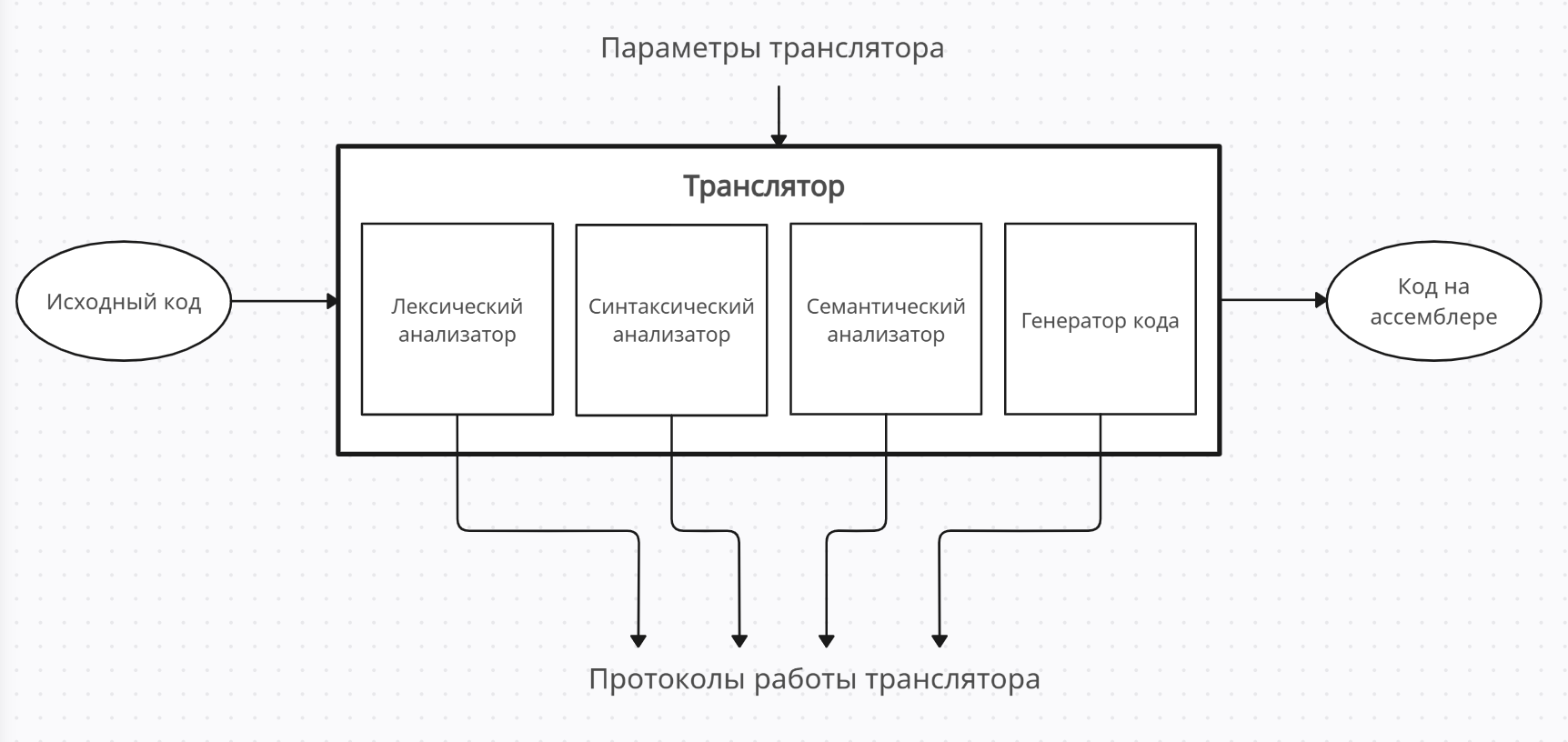
****

Рисунок 2.1 – Схема транслятора

Лексический анализ является начальным этапом трансляции, основной задачей которого является разбиение входного текста программы на токены, обнаружение лексических ошибок языка и создание таблиц лексем и идентификаторов.

Синтаксический анализ выполняет ключевую роль в трансляции. Он отвечает за распознавание синтаксических конструкций, выявление ошибок синтаксиса и построение дерева разбора. В процессе работы синтаксический анализатор использует таблицы лексем и идентификаторов, а также генерирует промежуточное представление программы на SMD-2024.

Семантический анализ проверяет исходную программу SMD-2024 на соответствие семантическим правилам языка, оценивая корректность программы с точки зрения её смысла и логики.

Генерация кода завершает процесс трансляции, преобразуя обработанный исходный код в ассемблерный код. На этом этапе используются данные, сформированные на предыдущих этапах, включая таблицы идентификаторов и лексем. Результатом работы генератора является код на языке Ассемблера.

**2.2. Перечень входных параметров транслятора**

Входные параметры представлены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка SMD-2024

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Значение по умолчанию |
| -in:<имя\_файла> | Входной файл может иметь любое расширение и должен содержать исходный код на языке SMD-2024. Этот параметр является обязательным: если он не указан, выполнение этапа трансляции не начнётся. | Не предусмотрено |
| -log:<имя\_файла> | Файл включает краткую информацию об исходном коде на языке SMD-2024. В него могут быть записаны таблицы идентификаторов, лексем и дерево разбора. | <имя\_файла>.log |
| -out:<имя\_файла> | В этот файл будет записан результат трансляции кода на язык assembler | <имя\_файла>.asm |
| t | Вывод дерева разбора синтаксического анализатора. | Не выводится |
| l | Вывод таблицы лексем | Не выводится |
| i | Вывод таблицы идентификаторов | Не выводится |

**2.3 Протоколы, формируемые транслятором**

В таблице 2.2 приведены протоколы, формируемые транслятором и их содержимое.

Таблица 2.2 Протоколы, формируемые транслятором языка SMD-2024

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл содержит информацию об исходном коде на языке SMD-2024. В него записывается протокол работы анализаторов, а также выявленные ошибки. |
| Выходной файл, c расширением ".asm" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

**3. Разработка лексического анализатора**

**3.1. Структура лексического анализатора**

Лексический анализ – первая фаза трансляции. Лексический анализ выполняется программой, называемой лексическим анализатором (или сканером). Лексический анализатор — это компонент компилятора, который отвечает за преобразование исходного кода программы в последовательность токенов. Структура лексического анализатора представлена на рисунке 3.1.

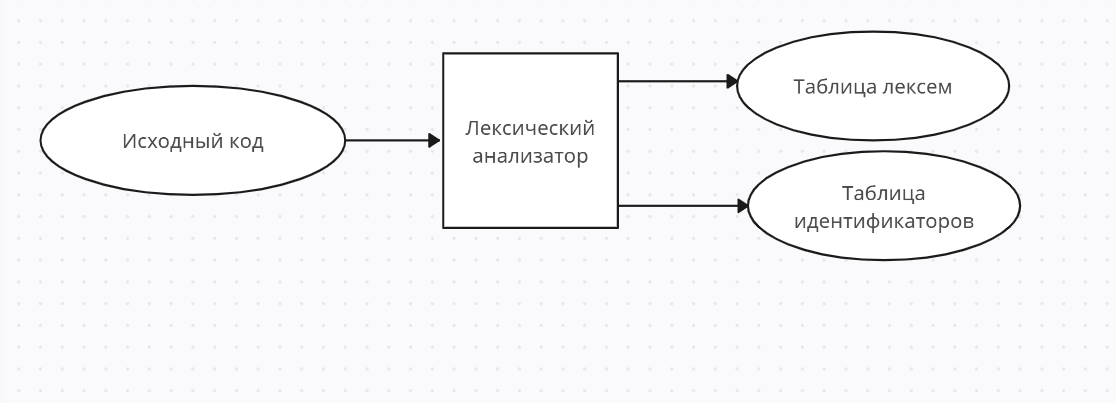


Рисунок 3.1 – Схема лексического анализатора

Результатом работы лексического анализатора являются таблица лексем и таблица идентификаторов.

**3.2. Контроль входных символов**

Исходный код на языке программирования SMD-2024 прежде чем транслироваться проверяется на допустимость символов. Принцип работы таблицы заключается в соответствии значения каждому элементу в шестнадцатеричной системе счисления значению в таблице Windows-1251. Это помогает программе быстро определять, как действовать с каждым входным символом, основываясь на его коде в таблице кодировки.

Таблица для проверки входных символов представлена в листинге 3.1.

#define IN\_CODE\_TABLE {\

IN::T, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::S, IN::N, IN::F, IN::F, IN::T,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::S, IN::F, IN::D, IN::O, IN::F, IN::L, IN::L, IN::C, IN::L, IN::L,\

IN::L, IN::L, IN::L, IN::L, IN::F, IN::L, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\

IN::T, IN::T, IN::L, IN::L, IN::L, IN::L, IN::L, IN::F, IN::F, IN::T, IN::T, IN::T, IN::T, IN::T,\

IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\

IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::L, IN::L, IN::L, IN::L, IN::T, IN::F, IN::T,\

IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\

IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::L, IN::F, IN::L,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F,\

}

Листинг 3.1 – Таблица для проверки входных символов

Описание значения символов: T – разрешённый символ, F – запрещённый символ, S – пробельный символ, С – символ одинарной кавычки, L – символ-разделитель, D – символ двойной кавычки, O – символ начала комментария, N – символ новой строки.

**3.3. Удаление избыточных символов**

**Избыточный символ** — это символ, который не несет полезной информации в контексте задачи и может быть удален без потери смысла данных. Удаление избыточных символов не выполняется, поскольку после проверки допустимости символов исходный код на языке программирования SMD-2024 сразу преобразуется в токены, которые добавляются в очередь.

**3.4. Перечень ключевых слов**

Лексемы — это элементы, обозначающие ключевые слова, операторы и разделители, которые служат для упрощения анализа исходного кода программы. Подробная информация о них представлена в таблице 3.1.

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

|  |  |  |  |
| --- | --- | --- | --- |
| Тип цепочки | Примечание | Цепочка | Лексема |
| Тип данных | Целочисленный беззнаковый тип данных | IntNum | i |
| Строковый тип данных | Line | i |
| Символьный тип данных | Symb | i |
| Лексема | Объявление переменной | Def | v |
| Подключение функции библиотеки | Out | e |
| Оператор вывода | Show | p |
| Объявление функции | Func | f |
| Возврат значения из функции | Get | r |
| Инструкция цикла | While | u |
| Инструкция Условного оператора | if | o |
| Блок инструкции цикла | [ | [ |
| ] | ] |
| Блок функции | { | { |
| } | } |
| Изменение приоритетности в выражении и отделение параметров функций | ( | ( |
| ) | ) |
| Сепараторы | ; | ; |
| , | , |
| Оператор присваивания | = | v |
| Условный оператор | < | b |
| > | b |
| Оператор | Знаки арифметических операций | + | v |
| - | v |
| \* | v |
| \ | v |
| / | v |
| % | v |
| Идентификатор |  | [a-z|A-Z]+  [a-z|A-Z|0-9]\* | i |
| Литерал | Целочисленный литерал | [1-9]+[0-9]\* | l |

Окончание таблицы 3.1

|  |  |  |  |
| --- | --- | --- | --- |
|  | Символьный литерал | [a-z|A-Z|0-9]\*  кроме ‘ | l |
| Строковый литерал | [a-z|A-Z|0-9]\* кроме ” | l |
| Точка входа |  | Main | m |

Каждому выражению соответствует свой детерминированный конечный автомат (ДКА) — структура с конечным числом состояний, которая выполняет разбор выражения. Для обработки текста фраза подаётся на вход каждому автомату из массива, где её анализируют с помощью регулярного выражения, описывающего переходы между состояниями графа. Если анализ проходит успешно, выражение заносится в таблицу лексем. В случае, если оно представляет собой идентификатор или литерал, соответствующие данные дополнительно записываются в таблицу идентификаторов. Пример реализации таблицы лексем и таблицы идентификаторов приведён в приложении В. Также в приложении В находятся конечные автоматы, соответствующие лексемам языка SMD-2024.

**3.5. Основные структуры данных**

Ключевыми структурами данных лексического анализатора для языка SMD-2024 являются таблицы лексем и идентификаторов. Таблица лексем хранит информацию о каждой лексеме, включая её текстовое представление, уникальный номер, определённый в процессе анализа, координаты в исходном коде (номер строки и позицию), индекс связанного идентификатора в таблице идентификаторов. Таблица идентификаторов используется для хранения имени идентификатора, его номера из таблицы лексем, типа данных, тип идентификатора, уникальный номер, определённый в процессе анализа.

Полученные таблицы лексем и идентификаторов представлены в приложении В.

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

Структура сообщений об содержит номер ошибки, вид ошибки, её сообщение, а также номер строки и позиции возникшей ошибки.

Индексы ошибок, обнаруживаемых лексическим анализатором, находятся в диапазоне 120-129. Текст ошибки содержит в себе префикс [Lex]. Перечень сообщений лексического анализатора представлен в таблице 3.2.

Таблица 3.2 – Перечень сообщений лексического анализатора

|  |  |
| --- | --- |
| Код ошибки | Сообщение |
| 120 | Ошибка при разборе токена. |
| 121 | Используется идентификатор без объявления. |
| 122 | Идентификатор не имеет типа |

Окончание таблицы 3.2

|  |  |
| --- | --- |
| 124 | Отсутствует точка входа (Main) |
| 125 | Обнаружена вторая точка входа (Main) |

**3.7. Принцип обработки ошибок**

Любая ошибка является критической, приводит к остановке работы транслятора и записи диагностического сообщения в log-файл.

**3.8. Параметры лексического анализатора**

Входные параметры, используемые для отображения результатов работы лексического анализатора, передаются через аргументы командной строки и описаны в таблице 2.1.

**3.9. Алгоритм лексического анализа**

Алгоритм лексического анализа основан на последовательном анализе цепочек исходного кода, их распознавании и добавлении данных в таблицы лексем и идентификаторов. Лексический анализатор выделяет токены из исходного текста программы, используя конечные автоматы, которые можно представить в виде графов. Если подходящий автомат для токена не найден, фиксируется номер строки с ошибкой, и выводится соответствующее сообщение. В случае успешного распознавания токена дальнейшие действия зависят от его типа.

Регулярные выражения, представляющие собой комбинацию констант и операторов, используются для описания регулярных языков. Они задают множество строк и операций над ними, а их поведение может быть представлено в виде графа.

Если токен является литералом, он добавляется в таблицу идентификаторов в формате abi, где a — имя функции, в которой объявлен литерал, b — префикс "$LEX", а i — порядковый номер литерала плюс один. Для токена, который является идентификатором переменной, его имя дополняется названием функции, в которой он объявлен, и затем добавляется в таблицу идентификаторов. В случае идентификатора функции имя родительской функции не записывается.

Когда токен представляет ключевое слово (например, тип данных или вид идентификатора), соответствующая ему лексема заносится в таблицу лексем, а информация о типе данных или виде идентификатора сохраняется для дальнейшего использования.

Впоследствии, при встрече идентификатора, он добавляется в таблицу идентификаторов вместе с его типом данных, видом идентификатора и именем формата ab, где a — имя функции, в которой он объявлен, а b — имя самого идентификатора.

Граф конечного автомата для ключевого слова «Def» представлен на рисунке 3.2. S0 – начальное состояние, S3  – конечное состояние автомата.

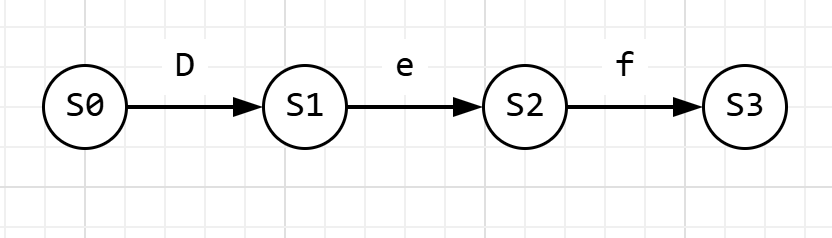
****

Рисунок 3.2 – Граф переходов для цепочки «Def»

**3.10. Контрольный пример**

Результат работы лексического анализатора – таблицы лексем и идентификаторов – представлен в приложении В.

**4. Разработка синтаксического анализатора**

**4.1. Структура синтаксического анализатора**

Синтаксический анализатор — это компонент компилятора, который отвечает за анализ структуры исходного кода на предмет соответствия синтаксическим правилам языка программирования. Синтаксический анализатор находится между лексическим анализатором и семантическим анализатором в цепочке обработки программного кода. Структура синтаксического анализатора показана на рисунке 4.1.

****

Рисунок 4.1 – Структура синтаксического анализатора SMD-2024

Входными данными для синтаксического анализа служат таблица лексем и таблица идентификаторов, которые формируются на этапе лексического анализа. Результатом работы синтаксического анализатора является дерево разбора.

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

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

1. T — это множество терминальных символов,
2. N — множество нетерминальных символов,
3. P — множество правил языка,
4. S — начальный символ грамматики, являющийся нетерминалом.

Данная грамматика имеет нормальную форму Грейбаха, так как не содержит леворекурсивных правил. Правила P имеют следующий формат:

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

Грамматика для языка SMD-2024 представлена в приложении Г.

TS — это терминальные символы, включая сепараторы, знаки арифметических операций и некоторые строчные буквы.

NS — это нетерминальные символы, обозначенные несколькими заглавными буквами латинского алфавита.

Таблица 4.1 - Описание грамматики

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| S | m{NrE;};  m{rE;};  dtfi(F){NrE;};S  dtfi(F){rE;};S  dtfi(){NrE;};S  dtfi(){rE;};S | Порождает правила, описывающее общую структуру программы |
| N | dti;  f(F);  i(F);N  ivE;  etfi(F);  o(B)[N]N  o(B)[N]  o(B)[N];  dti;N  ivE;N  etfi(F);N  pi;  pl;  pi;N  pi;N;  w(E)[N];  w(E)[N];N  pl;N  u(B)[N];  u(B)[N];N  etfi();N  etfi(); | Порождает правила, описывающие конструкции языка |

Окончание таблицы 4.1

|  |  |  |
| --- | --- | --- |
| E | i  l  (E)  i(W)  iM  lM  (E)M  i(W)M  i() | Порождает правила, описывающие выражения |
| F | ti  ti,F  i  i,F  l,F | Порождает правила, описывающие параметры локальной функции при её объявлении |
| W | i  l  i,W  l,W | Порождает правила, описывающие принимаемые параметры функции |
| B | ibi  ibl  lbi  lbl | Порождает правила, описывающие условное выражение в операторе цикла |
| M | vE  v(E)  v(E)M  vEM | Порождает правила, описывающие знаки арифметических операций |

Протокол и ошибки работы синтаксического анализатора выводятся в лог журнал.

**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

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | 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 |

Для вывода результата работы синтаксического анализатора нужно использовать флаг t.

**4.4. Основные структуры данных**

Основными структуры данных синтаксического анализатора языка SMD-2024 являются автомат с магазинной памятью и структура грамматики Грейбах, описывающей правила языка SMD-2024. Данные структуры представлены в приложении В.

## **4.5. Описание алгоритма синтаксического разбора**

Принцип работы автомата включает следующие этапы:

1. В магазин записывается стартовый символ.
2. Формируется входная лента на основе ранее созданных таблиц.
3. Автомат запускается.
4. Нетерминал раскрывается в соответствии с правилами грамматики, соответствующая цепочка записывается в магазин в обратном порядке.
5. Если терминалы в стеке и ленте совпадают, терминал выталкивается, лента сдвигается на одну позицию, переход к п.6. Если подходящих правил не найдено – ошибка синтаксического анализа, иначе возврат к п.4.
6. При обнаружении нетерминала в магазине выполняется переход к шагу 4.
7. Если стек опустошён, а входная лента пуста, синтаксический анализ считается успешным.

Блок-схема алгоритма представлена на рисунке 4.2.

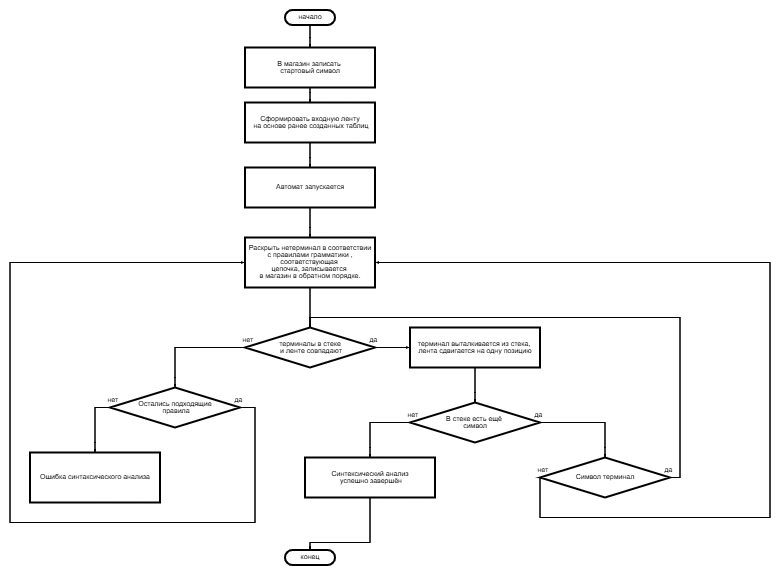


Рисунок 4.2 – Блок-схема алгоритма синтаксического разбора

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

Коды ошибок, обнаруживаемых синтаксическим анализатором, находятся в диапазоне 600–609, а текст ошибки начинается с префикса [Syn]. Список сообщений синтаксического анализатора представлен в таблице 4.3.

Список сообщений синтаксического анализатора языка SMD-2024 также приведён в таблице 4.3.

Табл. 4.3 - Перечень сообщений синтаксического анализатора

|  |  |
| --- | --- |
| Код ошибки | Сообщение |
| 600 | [Syn]: Неверная структура программы |
| 601 | [Syn]: Ошибочный оператор |

Окончание таблицы 4.3

|  |  |
| --- | --- |
| 602 | [Syn]: Ошибка в выражении" |
| 603 | [Syn]: Ошибка в параметрах функции |
| 604 | [Syn]: Ошибка в параметрах вызываемой функции |
| 605 | [Syn]: Ошибка знака в выражении |
| 606 | [Syn]: синтаксического анализа |
| 607 | [Syn]: Ошибка условной конструкции |
| 609 | [Syn]: Обнаружена синтаксическая ошибка (смотреть журнал Log) |

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

Для вывода результата работы синтаксического анализатора используются входные параметры, описанные в пункте 2.2Перечень входных параметров транслятора в таблице 2.1.

**4.8. Принцип обработки ошибок**

Процесс обработки ошибок включает следующие шаги:

1. Синтаксический анализатор последовательно проверяет правила и цепочки грамматики, пытаясь сопоставить их с конструкцией из таблицы лексем.
2. Если подходящая цепочка не обнаружена, фиксируется ошибка.
3. Сообщение об ошибке записывается в лог-файл, после чего компилятор прекращает дальнейшую работу.

**4.9. Контрольный пример**

Пример синтаксического разбора исходного кода на языке SMD-2024 можно найти в приложении Д. Также в этом приложении представлено дерево разбора данного кода. Протокол разбора представляет собой вывод результата пошаговой работы конечного автомата с магазинной памятью.

**5. Разработка семантического анализатора**

**5.1. Структура семантического анализатора**

Семантический анализатор — это компонент транслятора, который выполняет проверку исходного кода на семантические ошибки. Эти ошибки невозможно выявить с помощью регулярной или контекстно-свободной грамматики. В качестве входных данных используются таблица лексем и таблица идентификаторах.

Структура семантического анализатора языка SMD-2024 представлена на рисунке 5.1.

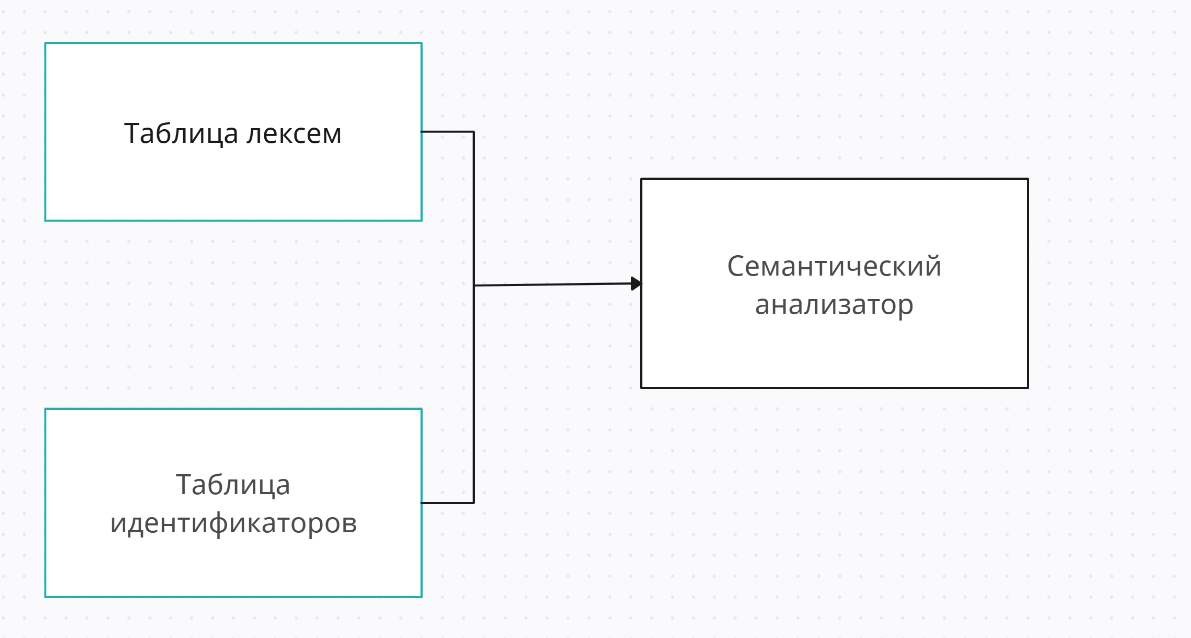


Рисунок 5.1 – Структура семантического анализатора языка SMD-2024

**5.2. Функции семантического анализатора**

Семантические проверки в компиляторе используются для проверки смысла и корректности программы на основании ее синтаксической структуры. Такие проверки могут быть выполнены на различных фазах компиляции: во время **лексического анализа, синтаксического анализа,** либо в отдельной фазе **семантического анализа.**

1) **Проверка типов данных:**

**Фаза выполнения:** Семантический анализатор.

**Описание:** Проверяется совместимость типов данных в выражениях, присваиваниях, параметрах функций и возвращаемых значениях.

**Функция:** void operands

**Входные данные:**

t — структура, содержащая таблицы лексем (лексический анализ) и идентификаторов.

**Выходные данные:** Нет (при обнаружении ошибки выбрасывается исключение).

**Краткое описание:** Функция проверяет корректность типов данных в выражениях. Например:

Совместимость типов при присваивании (int x = 5 + "string"; — ошибка).

Проверка операторов для строк (допускается только конкатенация через +) и символов (разрешены только операции + и -).

2) **Проверка параметров функций**

**Фаза выполнения:** Семантический анализатор.

**Описание:** Проверяет соответствие типов и количества аргументов при вызове функции.

**Функция:** void parameters

**Входные данные:**

t — структура с таблицами лексем и идентификаторов.

**Выходные данные:** Нет.

**Краткое описание:**

Сравнивает типы и количество фактических аргументов (указанных при вызове функции) с формальными (заданными в объявлении функции).

Если типы или количество аргументов не совпадают, выбрасывается исключение.

3) **Проверка возвращаемого значения функции**

**Фаза выполнения:** Семантический анализатор.

**Описание:** Проверяет, соответствует ли тип возвращаемого значения функции типу, указанному в объявлении.

**Функция:** void returns

**Входные данные:**

t — структура с таблицами лексем и идентификаторов.

**Выходные данные:** Нет.

**Краткое описание:**

Проверяет, что выражение после ключевого слова Get имеет тот же тип данных, что и указанный в декларации функции.

Если типы не совпадают, выбрасывается ошибка.

4) **Проверка области видимости идентификаторов**

**Фаза выполнения:** Синтаксический анализатор или семантический анализатор.

**Описание:** Убедиться, что переменные и функции используются только в тех местах программы, где они видимы и допустимы.

**Функция:** Проверяется в процессе построения дерева разбора или в семантическом анализе.

**Входные данные:** Таблица идентификаторов и их области видимости.

**Выходные данные:** Нет.

**Краткое описание:**

Если идентификатор используется вне области видимости или до объявления, генерируется ошибка.

5) **Проверка уникальности идентификаторов в одной области видимости**

**Фаза выполнения:** Лексический анализатор (при построении таблицы идентификаторов) или семантический анализатор.

**Описание:** Убедиться, что имена переменных, функций и других сущностей уникальны в рамках одной области видимости.

**Функция:** Проверка при добавлении идентификатора в таблицу.

**Входные данные:** Идентификатор, таблица идентификаторов.

**Выходные данные:** Нет.

**Краткое описание:**

Проверяет, существует ли идентификатор с таким именем в текущей области видимости.

Если уже существует, выбрасывается ошибка.

6) **Проверка корректности выражений в циклах**

**Фаза выполнения:** Семантический анализатор.

**Описание:** Проверяет, что условие цикла имеет булев тип данных и не содержит некорректных операций.

**Функция:** void operands

**Входные данные:**

Таблица лексем.

**Выходные данные:** Нет.

**Краткое описание:**

Если условие содержит недопустимый тип (например, строку), генерируется ошибка.

7) **Проверка использования операторов**

**Фаза выполнения:** Семантический анализатор.

**Описание:** Убедиться, что операторы (например, +, -, \*, /) применяются только к допустимым типам данных.

**Функция:** Проверяется в функции void operands

**Входные данные:** Таблица лексем и идентификаторов.

**Выходные данные:** Нет.

**Краткое описание:**

Если оператор используется для неподходящих типов (например, 5 / "text";), выбрасывается ошибка.

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

Все ошибка семантического анализатора имеют идентификатор свыше 700. Текст ошибки содержит в себе префикс [Sem]. Перечень сообщений семантического анализатора представлен в таблице 5.1

Таблица 5.1 – Перечень сообщений семантического анализатора

|  |  |
| --- | --- |
| Код ошибки | Сообщение |
| 700 | [Sem]: Повторное объявление идентификатора |
| 701 | [Sem]: Ошибка в возвращаемом значении |
| 702 | [Sem]: Ошибка в передаваемых значениях в функции: количество параметров не совпадает |
| 703 | [Sem]: Ошибка в передаваемых значениях в функции: типы параметров не совпадают |
| 704 | [Sem]: Нарушены типы данных в выражении |

Окончание таблицы 5.1

|  |  |
| --- | --- |
| 705 | [Sem]: Ошибка экспорта: в библиотеке нет такой функции [Sem]: Ошибка экспорта: неверные параметры |
| 706 | [Sem]: Ошибка экспорта: неверные параметры |
| 707 | [Sem]: Ошибка экспорта: ошибочный тип возвращаемого значения |
| 708 | [Sem]: Ошибочный оператор: строки можно только складывать |
| 709 | [Sem]: Ошибочные параметры условной конструкции: строки не могут быть параметрами условной конструкции |
| 710 | [Sem]: Ошибочный опреатор: для типа char разрешены только операции + и - |

**5.4. Принцип обработки ошибок**

При обнаружении хотя бы одной ошибки транслятор завершит свою работу c

Записью информации об ошибке в лог файл.

**5.5. Контрольный пример**

Контрольный пример заключается в тестировании функций семантического анализатора при наличии соответствующих ошибок в исходном коде. Ошибки, диагностируемые семантическим анализатором:

1)Код программы:

Main{

Out IntNum Func getCurrentDat(IntNum a);

….

}

Генерируемая ошибка представлена на рисунке 5.2



Рисунок 5.2 Сообщение об ошибке в коде под пунктом 1

2)Код программы:

Main{

Out IntNum Func getCurrentDate(IntNum a);

…

Def IntNum num1;

num1 = getCurrentDate(1,1);

…

}

Генерируемая ошибка представлена на рисунке 5.3



Рисунок 5.3 Сообщение об ошибки в коде под пунктом 2

3)Код программы:

Main{

…

Def Line str1;

Def Line str2;

str1 = "Date";

str2 = "Hours";

Def Line str25;

str25=str1/str2;

…

}

Генерируемая ошибка представлена на рисунке 5.4

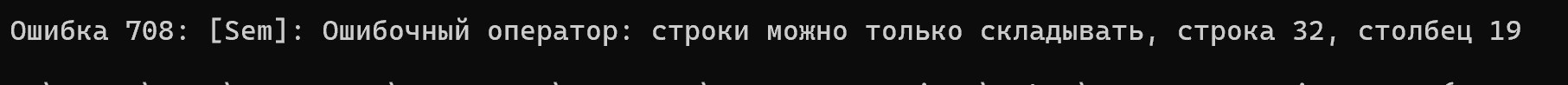


Рисунок 5.4 Сообщение об ошибке в коде под пунктом 3

**6. Вычисление выражений**

**6.1. Выражения, допускаемые языком**

В языке SMD-2024 допускаются вычисления выражений целочисленного типа данных, а также допускаются односоставные логические выражения с переменными или литералами целочисленного типа. Применение логических и арифметических операций в одном выражении не допускается. Такие операции, как сложение и вычитание доступны для символьного типа данных. Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритет операций в языке SMD-2024

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| ( | 1 |
| ) | 1 |
| + | 2 |
| - | 2 |
| \* | 3 |
| : | 3 |
| % | 3 |
| / | 3 |
| \ | 3 |

Примеры выражений из контрольного кода:

1. Арифметическое выражение:

b = a + b;

1. Условное выражение:

if (a < b) [

b = a + b;

];

1. Циклическое выражение:

While (x < y) [

x = x \* 3;

];

1. Функциональное выражение:

func = Demo(3);

1. Вывод данных:

Show "Demo Function";

Show func;

1. Побитовые операции:

l = n \ 1;

m = n / 1;

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

Обратная польская запись — это способ записи арифметических и логических выражений, при котором операнды предшествуют операторам. Основная цель применения — упрощение обработки выражений компьютером. Она устраняет необходимость использования скобок для определения порядка выполнения операций, так как их последовательность определяется автоматически при разборе записи.

Алгоритм преобразования выражений в польскую запись:

1. исходная строка просматривается слева направо;
2. операнды переносятся в результирующую строку в порядке их следования;
3. операция записывается в стек, если стек пуст или в вершине стека лежит отрывающая скобка;
4. операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
5. открывающая скобка помещается в стек;
6. закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются;
7. по концу разбора исходной строки все операции, оставшиеся в стеке, выталкиваются в результирующую строку;
8. если идентификатор является именем функции, то он заменяется на специальный символ «@».

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Исходная строка | Результирующая строка | Стек |
| x-y\*7/(g+8) |  |  |
| -y\*7/(g+8) | x |  |
| y\*7/(g+8) | x | - |
| \*7/(g+8) | xy | - |
| 7/(g+8) | xy | -\* |
| /(g+8) | xy7 | -\* |
| (g+8) | xy7\* | -/ |
| g+8) | xy7\* | -/( |
| +8) | xy7\*g | -/( |
| 8) | xy7\*g | -/(+ |
| ) | xy7\*g2 | -/(+ |
|  | xy7\*g2+ | -/ |
|  | xy7\*g2+/ | - |
|  | xy7\*g2+/- |  |

В результате успешного разбора стек оказывается пустым, а результирующая строка — полностью сформированной.

**6.3. Программная реализация обработки выражений**

Программная реализация алгоритма преобразования выражений к польской записи представлена в приложении Е.

**6.4. Контрольный пример**

В приложении Е приведена таблица лексем, содержащая результаты преобразования выражений в обратную польскую запись (ПОЛИЗ).

**7. Генерация кода**

**7.1. Структура генератора кода**

Генератор кода — это компонент транслятора, который преобразует данные, полученные на предыдущих этапах, в ассемблерный код. Входными данными для генератора служит таблица идентификаторов, содержащая информацию о переменных, функциях и других элементах программы. На основе этой информации создаётся файл с ассемблерным кодом, представляющий транслированную версию исходной программы. Итогом работы генератора является файл, содержащий результат компиляции в виде ассемблерных инструкций. Структура генератора кода SMD-2024 представлена на рисунке 7.1.

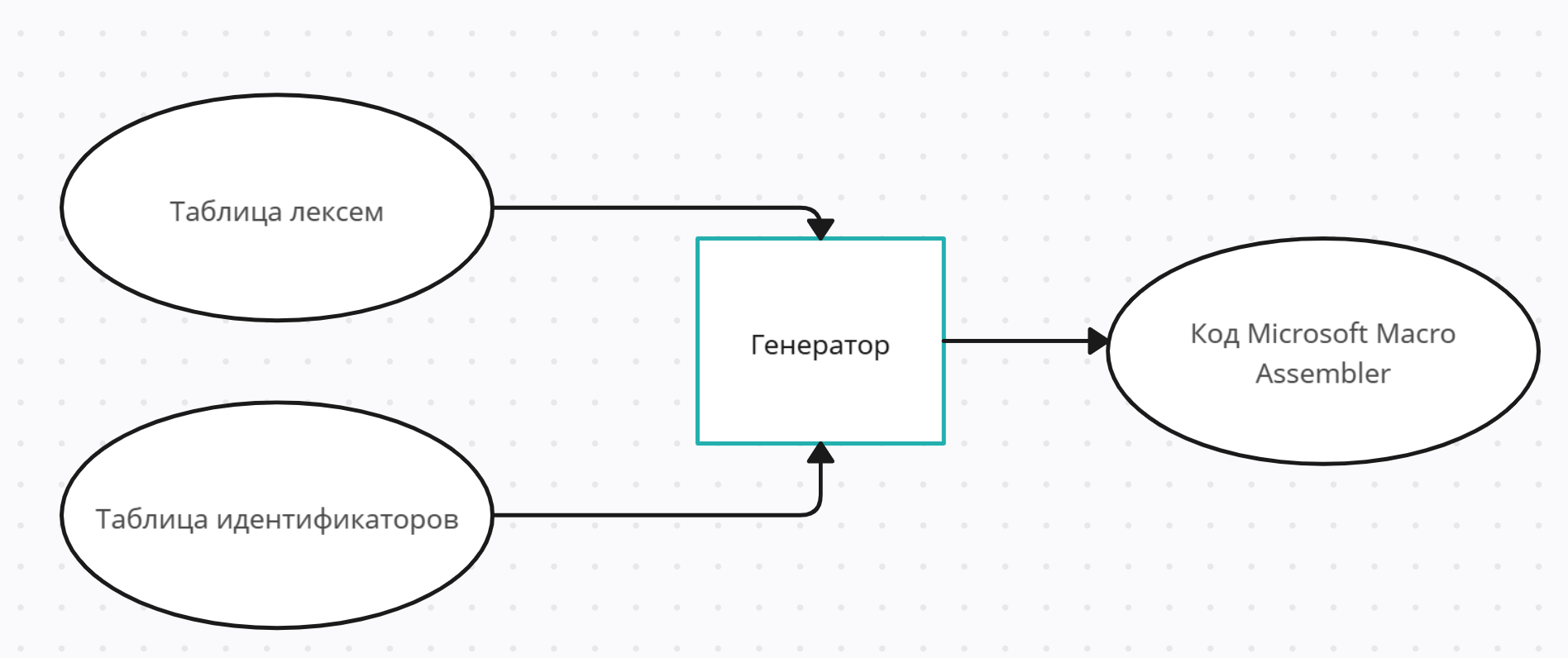
****

Рисунок 7.1 – Структура генератора кода

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

В языке SMD-2024 элементы таблицы идентификаторов распределены по различным сегментам ассемблера: сегменту данных и сегменту констант. Идентификаторы языка находятся в сегменте данных, а литералы размещаются в сегменте констант. В таблице 7.1 приведены соответствия между типами данных идентификаторов в языке SMD-2024 и языке ассемблера.

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

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке SMD-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| Symb | BYTE | Хранит символьный тип данных. |
| Line | DWORD | Хранит указатель на начало строки. |
| IntNum | DWORD | Хранит целочисленный тип данных без знака. |
| Лексема | BYTE  DWORD  DWORD | Литералы: символьные,  целочисленные, строковые |

**7.3. Статическая библиотека**

Статическая библиотека была разработана с использованием языка программирования C++ и размещена в проекте SMD-2024Lib . В свойствах этого проекта был указан тип конфигурации — «статическая библиотека (.lib)». Для интеграции библиотеки в ассемблерный код используется директива includelib, применяемая на этапе генерации машинного кода. После этого необходимо объявить имена функций, предоставляемых библиотекой. Состав статической библиотеки представлен в пункте 1.18.

**7.4. Особенности алгоритма генерации кода**

Алгоритм генерации кода выглядит следующим образом:

1. Генерирует заголовочную информацию: модель памяти, подключение библиотек.
2. Генерация прототипов функции.
3. Генерация сегмента стека.
4. Генерируем сегмент данных, сегмент констант.
5. Генерация сегмента кода.
6. Генерация пользовательских функций.
7. После генерации всех пользовательских функций, генерируется функция начала программы main.
8. Генерация кода, соответствующего окончанию процедуры main.

На рисунке 7.2 представлена обобщенная блок-схема генерации кода.

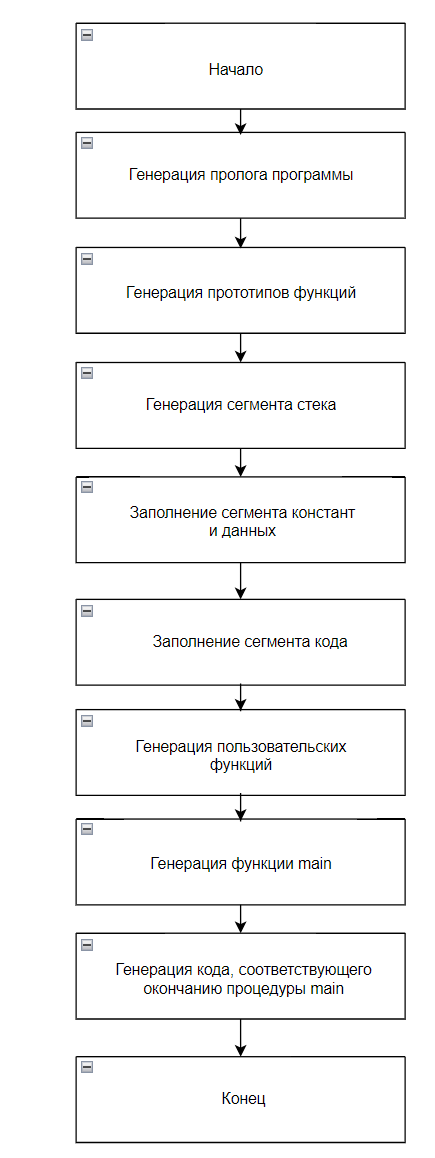


Рисунок 7.1 – Обобщенная блок-схема генерации кода

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

В языке SMD-2024 генератор кода принимает в качестве входных данных таблицу идентификаторов и таблицу лексем, которые используются для создания кода ассемблера. Результат работы генератора сохраняется в файл, заданный с помощью параметра -out.

**7.6. Контрольный пример**

Сгенерированный код сохраняется в файл с параметром -out. Пример сгенерированного кода представлен в приложении Ж.

**8. Тестирование транслятора**

**8.1. Общие положения**

Тесты создаются таким образом, чтобы охватить все аспекты работы компилятора, включая анализ лексем, синтаксический анализ, семантическую проверку и генерацию кода. Ошибка, выявленная в ходе выполнения программы выводится на консоль и записывается в журнал log.

**8.2. Результаты тестирования**

Итоги тестирования всех анализаторов приведены в таблице 8.1.

Таблица 8.1 – Итоги тестирования

|  |  |  |
| --- | --- | --- |
| Исходный код | Диагностическое сообщение | Анализатор |
| Def IntNum x;  x=10s; | Ошибка 120: [Lex]: Ошибка при разборе токена., строка 47, столбец 6 | Лексический |
| Main{  X=10;  …  } | Ошибка 121: [Lex]: Используется идентификатор без объявления., строка 47, столбец 2 | Лексический |
| Def x;  x=10; | Ошибка 122: [Lex]: Идентификатор не имеет типа, строка 46, столбец 6 | Лексический |
| {  Out IntNum Func getCurrentDate(IntNum a);  Out IntNum Func getCurrentMinuts(IntNum a);  Out IntNum Func getCurrentHour(IntNum a);  ……} | Ошибка 124: [Lex]: Отсутствует точка входа (Main) | Лексический |
| Main{}  Main{} | Ошибка 125: [Lex]: Обнаружена вторая точка входа (Main), строка 17, столбец 1 | Лексический |
| Def IntNum Demo(IntNum a) {} | Ошибка 602: [Syn]: Ошибка в выражении, строка 1, столбец 16 | Синтаксический |
| Def IntNum b;  b == 5; | 602: строка 4,[Syn]: Ошибка в выражении | Синтаксический |

Продоложение таблицы 8.1

|  |  |  |
| --- | --- | --- |
|  | 601: строка 4,[Syn]: Ошибочный оператор |  |
| Main{  Def IntNum Demo(IntNum a) {}  } | 600: строка 1,[Syn]: Неверная структура программы | Синтаксический |
| Def IntNum num1;  Def IntNum num1; | Ошибка 700: [Sem]: Повторное объявление идентификатора, строка 24, столбец 12 | Семантический |
| Def IntNum Func Demo(IntNum a) {  Def IntNum b;  b = 5;  if (a < b)  [  b = a + b;  ];  Get "Ytgj";  }; | Ошибка 701: [Sem]: Ошибка в возвращаемом значении, строка 12, столбец 12 | Семантический |
| num1 = getCurrentDate(); | Ошибка 702: [Sem]: Ошибка в передаваемых значениях в функции: количество параметров не совпадает, строка 24, столбец 24 | Семантический |
| Def IntNum Func Demo(IntNum a) {  Def IntNum b;  b = 5;  if (a < b)  [  b = a + b;  ];  Get b;  };  Main{  Show "Demo Function";  Def IntNum func;  func = Demo("wedrfgyh");  Show func} | Ошибка 703: [Sem]: Ошибка в передаваемых значениях в функции: типы параметров не совпадают, строка 42, столбец 24 | Семантический |

Окончание таблицы 8.1

|  |  |  |
| --- | --- | --- |
| Def IntNum x;  x = "fghj"; | Ошибка 704: [Sem]: Нарушены типы данных в выражении, строка 48, столбец 12 | Семантический |
| Out IntNum Func getCurrentDat(IntNum a); | Ошибка 705: [Sem]: Ошибка экспорта: в библиотеке нет такой функции, строка 19, столбец 18 | Семантический |
| Out IntNum Func getCurrentMinuts(Symb a); | Ошибка 706: [Sem]: Ошибка экспорта: неверные параметры, строка 20, столбец 41 | Семантический |
| Out Symb Func getCurrentMinuts(IntNum a); | Ошибка 707: [Sem]: Ошибка экспорта: ошибочный тип возвращаемого значения, строка 20, столбец 16 | Семантический |
| Def Line str1;  str1="ghjk";  Def Line str2;  str2="ghjk";  Def Line str3;  str3=str1/str2; | Ошибка 708: [Sem]: Ошибочный оператор: строки можно только складывать, строка 34, столбец 18 | Семантический |
| While("fghj" < "bg")[  x = x \* 3;  Show x;  ]; | Ошибка 709: [Sem]: Ошибочные параметры условной конструкции: строки не могут быть параметрами условной конструкции, строка 42, столбец 14 | Семантический |
| Def Symb a;  a= 'd';  Def Symb b;  b= 'f';  Def Symb v;  v= a/b; | Ошибка 710: [Sem]: Ошибочный опреатор: для типа char разрешены только операции + и -, строка 35, столбец 5 | Семантический |

# **Заключение**

В процессе выполнения курсовой работы был создан транслятор для языка SMD-2024, который переводит его в ассемблер, а также подготовлена пояснительная записка с описанием спецификации языка.

Таким образом, были выполнены следующие ключевые задачи:

1. Сформулирована спецификация языка SMD-2024;
2. Разработаны конечные автоматы;
3. Реализован лексический анализатор, способный распознавать допустимые строки языка;
4. Создана контекстно-свободная грамматика, приведённая к нормальной форме Грейбаха, для описания синтаксически корректных конструкций языка;
5. Реализован синтаксический анализатор;
6. Разработан семантический анализатор для проверки логической корректности инструкций;
7. Создан транслятор, который преобразует код в язык ассемблера;
8. Проведено тестирование всех компонентов системы.

В результате выполнения курсового проекта по дисциплине «Конструирование программного обеспечения» были приобретены навыки разработки трансляторов и интерпретаторов, а также знания теории формальных грамматик.

# **Список использованных источников**

1. Карпов Ю. Теория и технология программирования. Основы построения трансляторов, 2005. – 272с.
2. Введение в теорию трансляторов [Электронный ресурс]. – Режим доступа: http://bourabai.ru/tpoi/compilers.htm. – Дата доступа: 20.10.2024.
3. Википедия: Обратная польская запись [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Reverse\_Polish\_notation. – Дата доступа: 15.11.2024.
4. MASM для x86 [Электронный ресурс]. – Режим доступа: <https://docs.microsoft.com/en-us/cpp/assembler/masm/masm-for-x64-ml64-exe?view=msvc-160>. – Дата доступа: 29.11.2024.

# ПРИЛОЖЕНИЕ A

Def IntNum Func Demo(IntNum a) {

Def IntNum b;

b = 5;

if (a < b)

[

b = a + b;

];

Get b;

};

Main{

Out IntNum Func getCurrentDate(IntNum a);

Out IntNum Func getCurrentMinuts(IntNum a);

Out IntNum Func getCurrentHour(IntNum a);

Def IntNum num1;

num1 = getCurrentDate(1);

Def IntNum num2;

num2 = getCurrentHour(1);

Def IntNum num3;

num3 = getCurrentMinuts(1);

Def Line str1;

Def Line str2;

Def Line str3;

str1 = "Date";

Show str1;

Show num1;

str2 = "Hours";

Show str2;

Show num2;

str3 = "Minuts";

Show str3;

Show num3;

Show "Demo Function";

Def IntNum func;

func = Demo(2);

Show func;

Def IntNum x;

x = 10;

Def IntNum y;

y = 200;

While(x < y)[

x = x \* 3;

Show x;

];

Def Symb sym;

sym = 'k';

Show "Symbol";

Show sym;

Def IntNum n;

n = 18;

Def IntNum m;

m = n / 1;

Def IntNum l;

l = n \1;

Show "18<<1:";

Show l;

Show "18>>1:";

Show m;

Get 0;

}

Листинг 1 – Контрольный пример

ПРИЛОЖЕНИЕ Б

#define IN\_CODE\_TABLE {\

IN::T, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::S, IN::N, IN::F, IN::F, IN::T,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::S, IN::F, IN::D, IN::O, IN::F, IN::L, IN::L, IN::C, IN::L, IN::L,\

IN::L, IN::L, IN::L, IN::L, IN::F, IN::L, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\

IN::T, IN::T, IN::L, IN::L, IN::L, IN::L, IN::L, IN::F, IN::F, IN::T, IN::T, IN::T, IN::T, IN::T,\

IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\

IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::L, IN::L, IN::L, IN::L, IN::T, IN::F, IN::T,\

IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\

IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::L, IN::F, IN::L,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F, IN::F,\

IN::F, IN::F, IN::F,\

}

Листинг 2 – Таблица проверки входных символов

## ПРИЛОЖЕНИЕ В

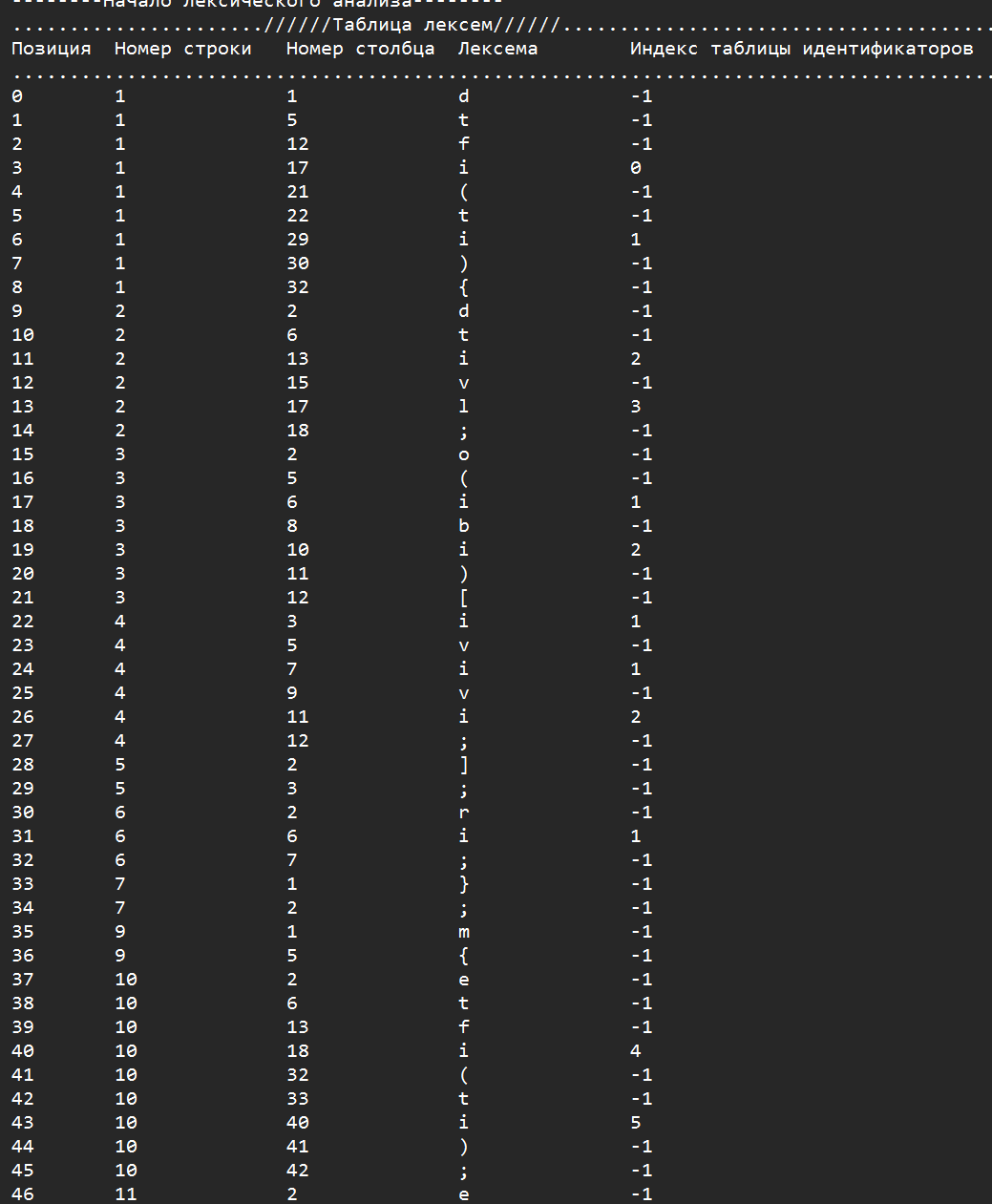


Рисунок 1– Таблица лексем

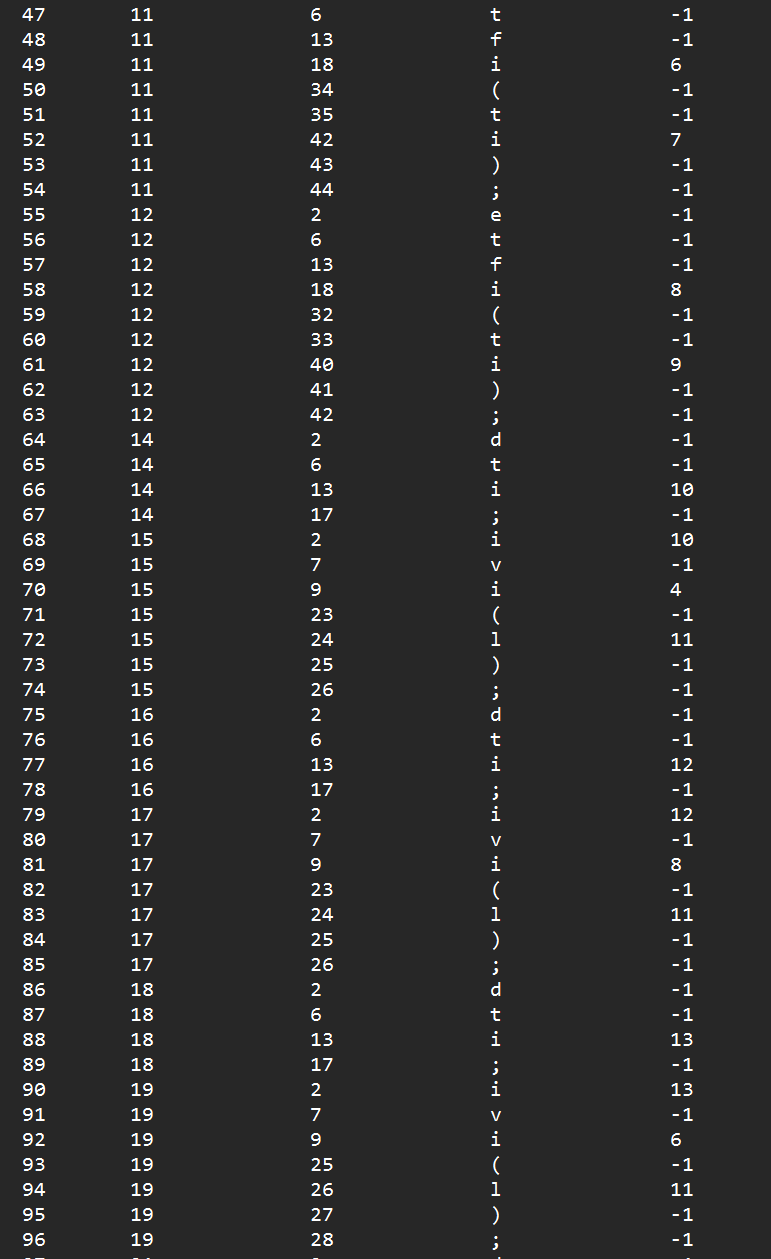


Рисунок 2– Таблица лексем (продолжение)

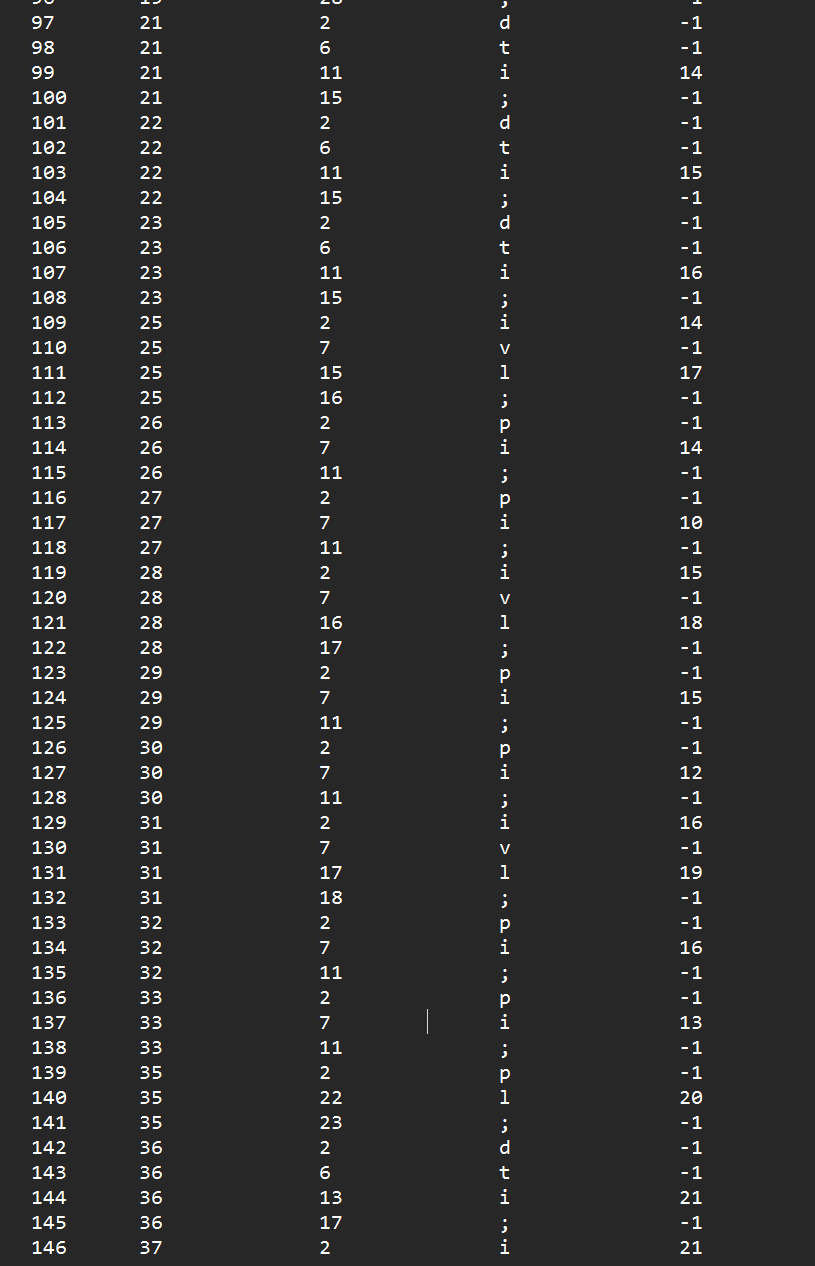


Рисунок 3– Таблица лексем (продолжение)

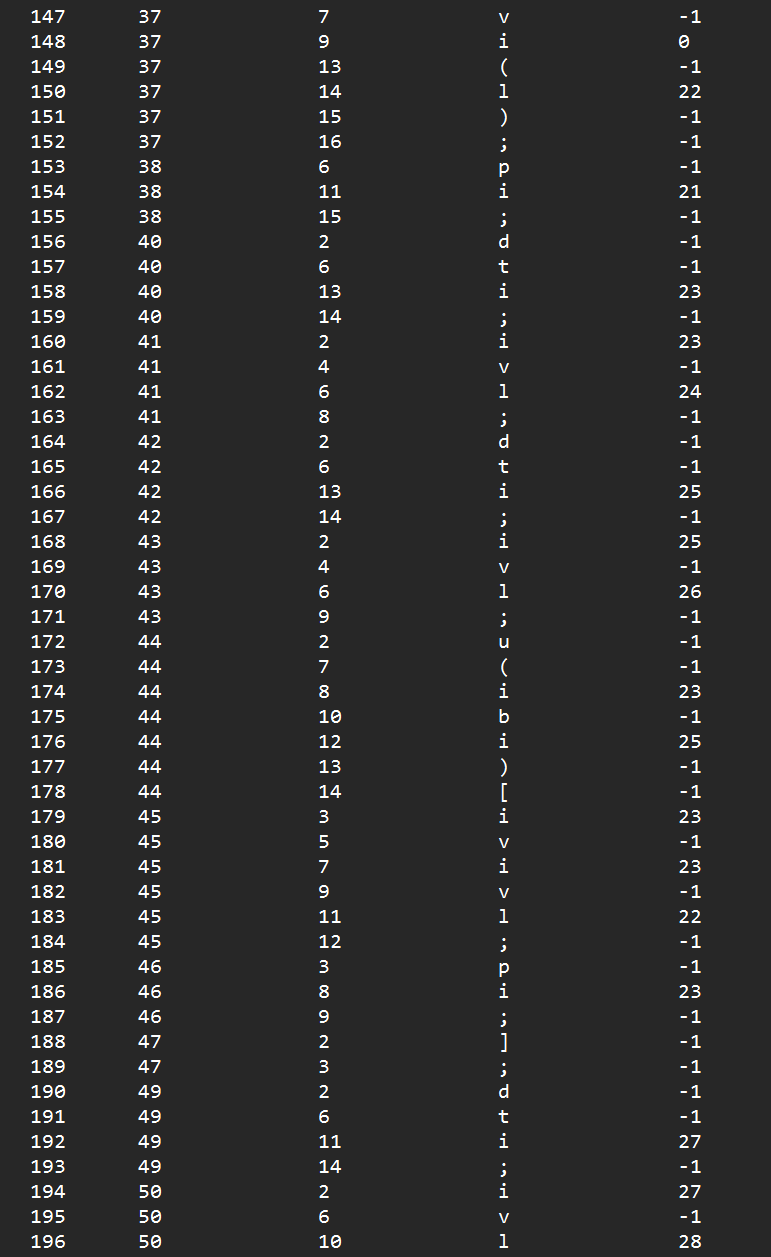


Рисунок 3– Таблица лексем (продолжение)

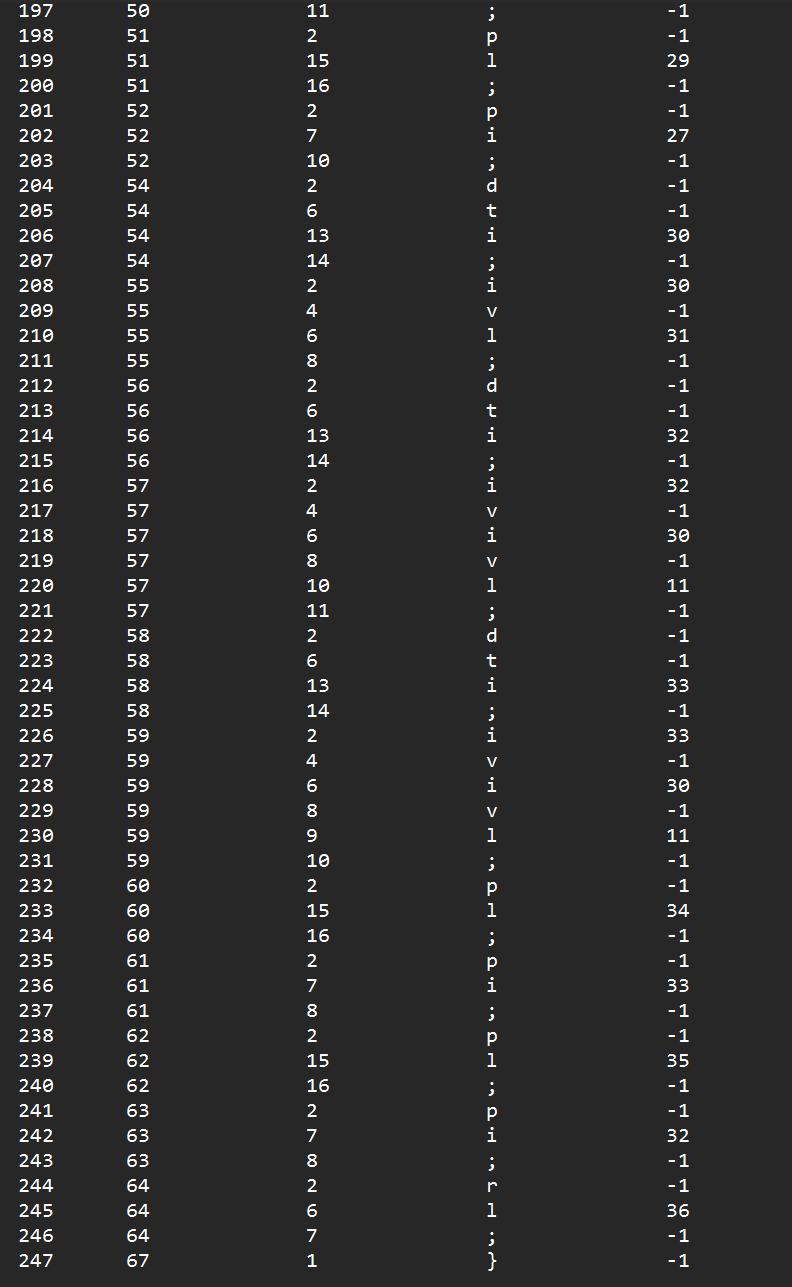


Рисунок 4– Таблица лексем (конец)

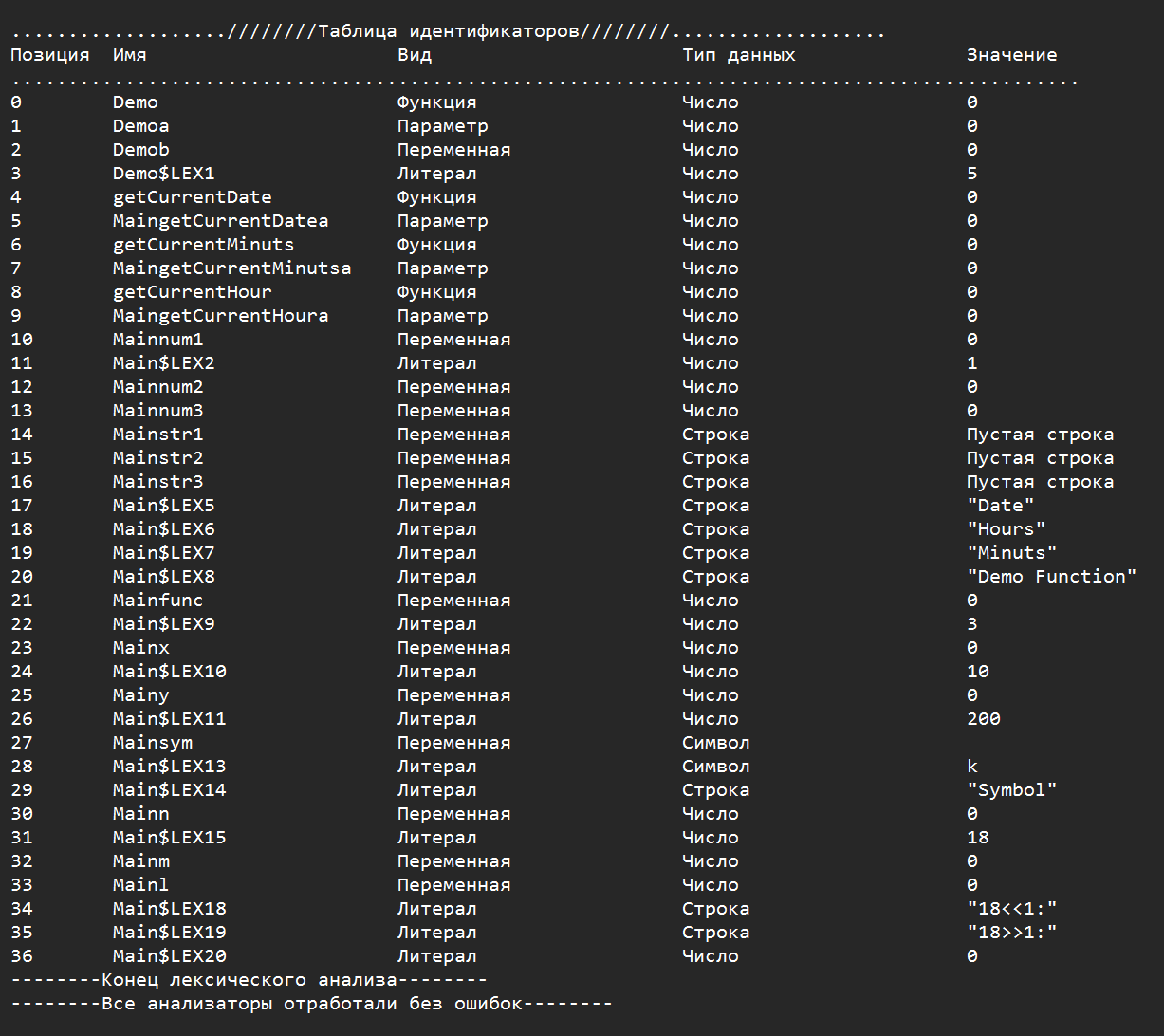


Рисунок 5 – Таблица идентификаторов

FST l\_Line(

str,

5, //количество состояний

NODE(1, RELATION('L', 1)),

NODE(1, RELATION('i', 2)),

NODE(1, RELATION('n', 3)),

NODE(1, RELATION('e', 4)),

NODE()

);

FST l\_Func(

str,

5, //количество состояний

NODE(1, RELATION('F', 1)),

NODE(1, RELATION('u', 2)),

NODE(1, RELATION('n', 3)),

NODE(1, RELATION('c', 4)),

NODE()

);

FST l\_Def(

str,

4, //количество состояний

NODE(1, RELATION('D', 1)),

NODE(1, RELATION('e', 2)),

NODE(1, RELATION('f', 3)),

NODE()

);

FST l\_Get(

str,

4, //количество состояний

NODE(1, RELATION('G', 1)),

NODE(1, RELATION('e', 2)),

NODE(1, RELATION('t', 3)),

NODE()

);

FST l\_Show(

str,

5, //количество состояний

NODE(1, RELATION('S', 1)),

NODE(1, RELATION('h', 2)),

NODE(1, RELATION('o', 3)),

NODE(1, RELATION('w', 4)),

NODE()

);

FST l\_prints(

str,

5, //количество состояний

NODE(1, RELATION('s', 1)),

NODE(1, RELATION('a', 2)),

NODE(1, RELATION('y', 3)),

NODE(1, RELATION('s', 4)),

NODE()

);

FST l\_main(

str,

5, //количество состояний

NODE(1, RELATION('M', 1)),

NODE(1, RELATION('a', 2)),

NODE(1, RELATION('i', 3)),

NODE(1, RELATION('n', 4)),

NODE()

);

FST l\_conditional(

str,

3, //количество состояний

NODE(1, RELATION('i', 1)),

NODE(1, RELATION('f', 2)),

NODE()

);

FST l\_semicolon(

str,

2, //количество состояний

NODE(1, RELATION(';', 1)),

NODE()

);

FST l\_comma(

str,

2, //количество состояний

NODE(1, RELATION(',', 1)),

NODE()

);

FST l\_braceleft(

str,

2, //количество состояний

NODE(1, RELATION('{', 1)),

NODE()

);

FST l\_braceright(

str,

2, //количество состояний

NODE(1, RELATION('}', 1)),

NODE()

);

FST l\_lefthesis(

str,

2, //количество состояний

NODE(1, RELATION('(', 1)),

NODE()

);

FST l\_cycleStart(

str,

2, //количество состояний

NODE(1, RELATION('[', 1)),

NODE()

);

FST l\_cycleEnd(

str,

2, //количество состояний

NODE(1, RELATION(']', 1)),

NODE()

);

FST l\_conditionalStart(

str,

2, //количество состояний

NODE(1, RELATION('[', 1)),

NODE()

);

FST l\_conditionalEnd(

str,

2, //количество состояний

NODE(1, RELATION(']', 1)),

NODE()

);

FST l\_righthesis(

str,

2, //количество состояний

NODE(1, RELATION(')', 1)),

NODE()

);

FST l\_verb(

str,

2, //количество состояний

NODE(8, RELATION('+', 1), RELATION('-', 1), RELATION('\*', 1),

RELATION('/', 1), RELATION(':', 1), RELATION('\\', 1), RELATION('%', 1), RELATION('=', 1)),

NODE()

);

FST l\_boolVerb(

str,

2, //количество состояний

NODE(4, RELATION('!', 1), RELATION('<', 1), RELATION('>', 1)),

NODE()

);

Листинг 3 – Пример конечных автоматов

# ПРИЛОЖЕНИЕГ

namespace GRB

{

Greibach greibach(NS('S'), TS('$'),

9,

Rule(NS('S'), GRB\_ERROR\_SERIES + 0,

6, //todo m{NrE;}; tfi(F){NrE;};S m{NrE;};S tfi(F){NrE;};

Rule::Chain(7, TS('m'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}')),//+

Rule::Chain(6, TS('m'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}')),//+

Rule::Chain(15, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),//+

Rule::Chain(14, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),

Rule::Chain(14, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')), // функция

Rule::Chain(13, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S'))

),

Rule(NS('N'), GRB\_ERROR\_SERIES + 1,

22, //todo dti; rE; i=E; dtfi(F); dti;N rE;N i=E;N dtfi(F);N pl;N pi;N pl; pi; pi(W);

Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),//+

//Rule::Chain(5, TS('d'), TS('t'), TS('i'), NS('M'), TS(';')),//

Rule::Chain(5, TS('f'), TS('('), NS('F'), TS(')'), TS(';')),//+

Rule::Chain(6, TS('i'), TS('('), NS('F'), TS(')'), TS(';'), NS('N')),//+

Rule::Chain(8, TS('o'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), NS('N')),//+

Rule::Chain(5, TS('o'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']')),//+

Rule::Chain(8, TS('o'), TS('('), NS('B'), TS(')'), TS('['), TS('N'), TS(']'), TS(';')),//+

Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')), //+

Rule::Chain(4, TS('i'), TS('v'), NS('E'), TS(';')), //+

Rule::Chain(5, TS('i'), TS('v'), NS('E'), TS(';'), NS('N')),

Rule::Chain(9, TS('e'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';'), NS('N')), //+

Rule::Chain(8, TS('e'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';')),//+

Rule::Chain(8, TS('e'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS(';'), NS('N')),

Rule::Chain(7, TS('e'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS(';')),

Rule::Chain(4, TS('p'), TS('i'), TS(';'), NS('N')),//+

Rule::Chain(3, TS('p'), TS('i'), TS(';')),//+

Rule::Chain(4, TS('p'), TS('l'), TS(';'), NS('N')),//+

Rule::Chain(3, TS('p'), TS('l'), TS(';')),//+

Rule::Chain(8, TS('o'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), TS(';')), // if (a < b) [ N ];

Rule::Chain(5, TS('p'), TS('i'), TS(';'), NS('N'), TS(';')),

Rule::Chain(8, TS('u'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), TS(';')),

Rule::Chain(9, TS('u'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), TS(';'), NS('N')),

Rule::Chain(9, TS('w'), TS('('), NS('E'), TS(')'), TS('['), NS('N'), TS(']'), TS(';')), // While(E)[N];

Rule::Chain(10, TS('w'), TS('('), NS('E'), TS(')'), TS('['), NS('N'), TS(']'), TS(';'), NS('N')) // While(E)[N];N

),

Rule(NS('E'), GRB\_ERROR\_SERIES + 2,

9, //todo i l (E) i(W) iM lM (E)M i(W)M

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

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

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

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

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

Rule::Chain(2, TS('i'), NS('M')),//+

Rule::Chain(2, TS('l'), NS('M')),//+

Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),//+

Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M'))

),

Rule(NS('M'), GRB\_ERROR\_SERIES + 3,

4, //todo vE vEM v(E) v(E)M

Rule::Chain(2, TS('v'), NS('E')),//+

Rule::Chain(4, TS('v'), TS('('), NS('E'), TS(')')),

Rule::Chain(5, TS('v'), TS('('), NS('E'), TS(')'), NS('M')),

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

),

Rule(NS('F'), GRB\_ERROR\_SERIES + 4,

6, //todo ti ti,F

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

Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F')),//+

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

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

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

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

),

Rule(NS('W'), GRB\_ERROR\_SERIES + 5,

4, //todo i l i,W l,W

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

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

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

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

),

Rule(NS('B'), GRB\_ERROR\_SERIES + 6,

4, //todo ibi ibl lbi lbl

Rule::Chain(3, TS('i'), TS('b'), TS('i')),

Rule::Chain(3, TS('i'), TS('b'), TS('l')),

Rule::Chain(3, TS('l'), TS('b'), TS('i')),

Rule::Chain(3, TS('l'), TS('b'), TS('l'))

)

);

Листинг 4 – Правила грамматики

## ПРИЛОЖЕНИЕД

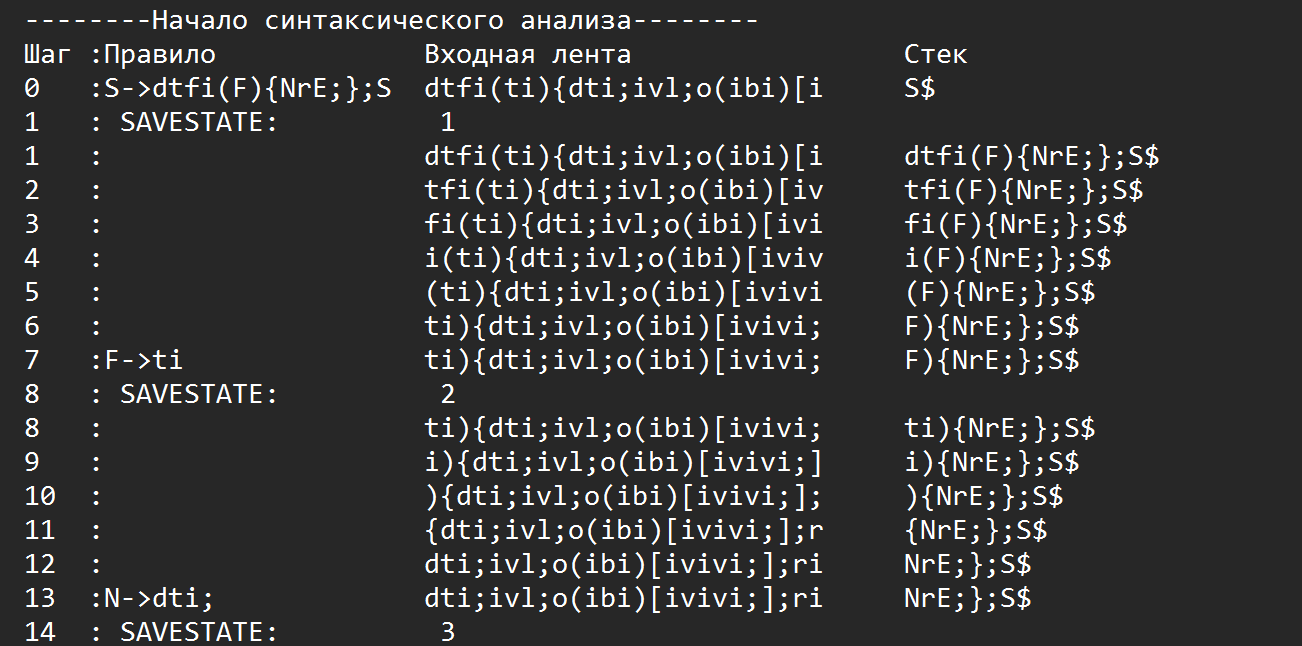


Рисунок 6 - Начало синтаксического анализа

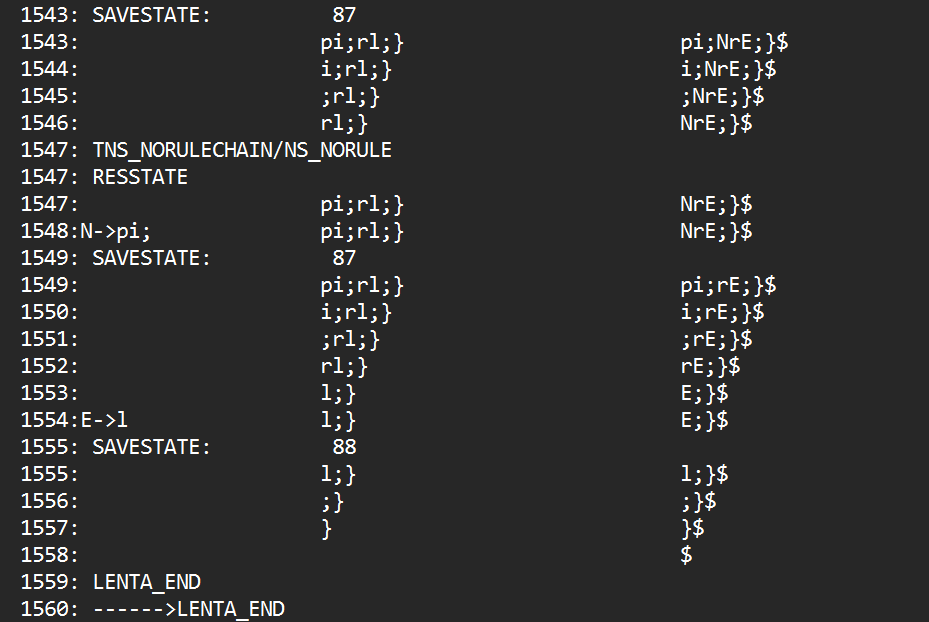


Рисунок 7 – Конец синтаксического анализа

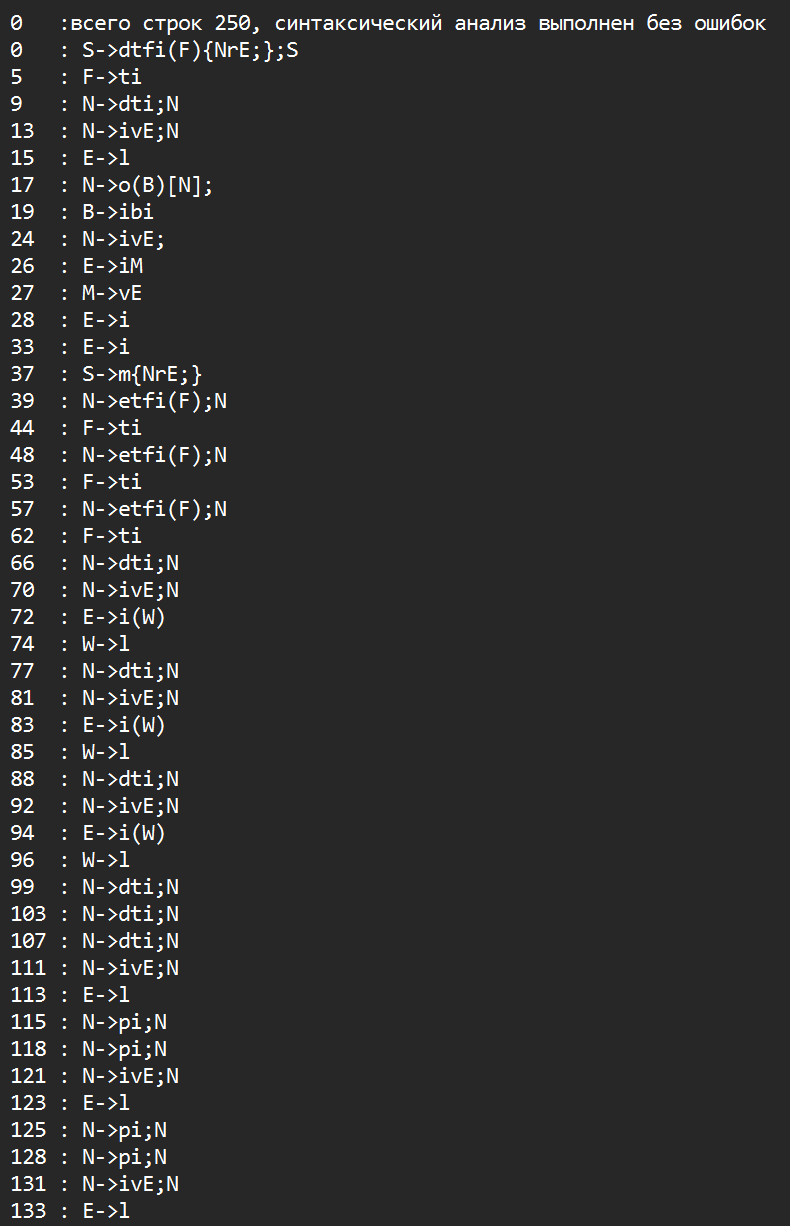


Рисунок 8 – Пример разбора синтаксическим анализатором

## ПРИЛОЖЕНИЕЕ

namespace PolishNotation {

template <typename T>

struct container : T

{

using T::T;

using T::c;

};

std::string toString(int n) {

char buf[40];

sprintf\_s(buf, "%d", n);

return buf;

}

bool find\_elem(std::stack<char> stack, size\_t size, char elem) {

for (size\_t i = 0; i < size; i++)

if (stack.top() == elem)

return true;

else

stack.pop();

return false;

}

int get\_priority(char a)

{

switch (a)

{

case '(':

return 0;

case ')':

return 0;

case ',':

return 1;

case '-':

return 2;

case '+':

return 2;

case '\*':

return 3;

case '%':

return 3;

case '/':

return 3;

case '\\':

return 3;

case ':':

return 3;

default: {

return 0;

}

}

}

void fixIt(LT::LexTable& lextable, const std::string& str, size\_t length, size\_t pos, const std::vector<int>& ids) {

for (size\_t i = 0, q = 0; i < str.size(); i++) {

lextable.table[pos + i].lexema = str[i];

if (lextable.table[pos + i].lexema == LEX\_ID || lextable.table[pos + i].lexema == LEX\_LITERAL) {

lextable.table[pos + i].idxTI = ids[q];

q++;

}

else

lextable.table[pos + i].idxTI = LT\_TI\_NULLIDX;

}

int temp = str.size() + pos;

for (size\_t i = 0; i < length - str.size(); i++) {

lextable.table[temp + i].idxTI = LT\_TI\_NULLIDX;

lextable.table[temp + i].lexema = '!';

lextable.table[temp + i].sn = -1;

}

}

bool PolishNotation(int lextable\_pos, LT::LexTable& lextable, IT::IdTable& idtable)

{

container<std::stack<char>> stack;

std::string PolishString;

std::vector<int> ids;

int operators\_count = 0, operands\_count = 0, iterator = 0, right\_counter = 0, left\_counter = 0, params\_counter = 0;

for (int i = lextable\_pos; i < lextable.size; i++, iterator++) {

char lexem = lextable.table[i].lexema;

char data = lextable.table[i].data;

size\_t stack\_size = stack.size();

if (idtable.table[lextable.table[i].idxTI].idtype == IT::IDTYPE::F) {

stack.push('@');

operands\_count--;

}

switch (lexem) {

case LEX\_OPERATOR:

{

if (!stack.empty() && stack.top() != LEX\_LEFTHESIS) {

while (!stack.empty() && get\_priority(data) <= get\_priority(stack.top())) {

PolishString += stack.top();

stack.pop();

}

}

stack.push(data);

operators\_count++;

break;

}

case LEX\_COMMA:

{

while (!stack.empty()) {

if (stack.top() == LEX\_LEFTHESIS)

break;

PolishString += stack.top();

stack.pop();

}

operands\_count--;

break;

}

case LEX\_LEFTHESIS:

{

left\_counter++;

stack.push(lexem);

break;

}

case LEX\_RIGHTHESIS:

{

right\_counter++;

if (!find\_elem(stack, stack\_size, LEX\_LEFTHESIS))

return false;

while (stack.top() != LEX\_LEFTHESIS) {

PolishString += stack.top();

stack.pop();

}

stack.pop();

if (!stack.empty() && stack.top() == '@') {

PolishString += stack.top() + toString(params\_counter - 1);

params\_counter = 0;

stack.pop();

}

break;

}

case LEX\_SEMICOLON:

{

if (operators\_count != 0 && operands\_count != 0)

if ((!stack.empty() && (stack.top() == LEX\_RIGHTHESIS || stack.top() == LEX\_LEFTHESIS))

|| right\_counter != left\_counter || operands\_count - operators\_count != 1)

return false;

while (!stack.empty()) {

PolishString += stack.top();

stack.pop();

}

fixIt(lextable, PolishString, iterator, lextable\_pos, ids);

return true;

break;

}

case LEX\_ID: {

if (std::find(stack.c.begin(), stack.c.begin(), '@') != stack.c.end())

params\_counter++;

PolishString += lexem;

if (lextable.table[i].idxTI != LT\_TI\_NULLIDX)

ids.push\_back(lextable.table[i].idxTI);

operands\_count++;

break;

}

case LEX\_LITERAL: {

if (std::find(stack.c.begin(), stack.c.begin(), '@') != stack.c.end())

params\_counter++;

PolishString += lexem;

if (lextable.table[i].idxTI != LT\_TI\_NULLIDX)

ids.push\_back(lextable.table[i].idxTI);

operands\_count++;

break;

}

}

}

return true;

}

void OutputLexTableFormatted(const LT::LexTable& lextable, const std::string& filename) {

std::ofstream outfile(filename, std::ios::out);

if (!outfile.is\_open()) {

throw ERROR\_THROW(120); // Ошибка при открытии файла

}

outfile << "------------------- Таблица лексем --------------------" << std::endl;

int lineIndex = 0;

std::string currentLine;

// Перебираем все лексемы

for (int i = 0; i < lextable.size; i++) {

if (lextable.table[i].lexema == '!') continue; // Пропускаем удалённые элементы

if (lextable.table[i].sn != -1) {

// Добавляем лексему с её идентификатором в текущую строку

currentLine += lextable.table[i].lexema;

}

else {

// Если лексема не имеет связи с идентификатором, просто выводим её

currentLine += lextable.table[i].lexema + " ";

}

// Если находим символ новой строки, записываем текущую строку и начинаем новую

if (lextable.table[i].lexema == LEX\_SEMICOLON|| lextable.table[i].lexema == LEX\_LEFTBRACE || lextable.table[i].lexema == LEX\_LEFT\_SQUAREBRACE) {

outfile << lineIndex + 1 << ": " << currentLine << std::endl;

currentLine.clear(); // Очищаем строку для следующей

lineIndex++; // Переходим к следующей строке

}

}

// Если осталась незаписанная строка (на случай, если нет семиколона в конце)

if (!currentLine.empty()) {

outfile << lineIndex + 1 << ": " << currentLine << std::endl;

}

outfile.close();

}

// Функция для обработки польской записи и записи в файл

void DoPolish(LEX::LEX t, const std::string& outputFilename) {

for (int i = 0; i < t.lextable.size; i++) {

if (t.lextable.table[i].lexema == LEX\_EQUAL) {

if (!PolishNotation(i + 1, t.lextable, t.idtable)) {

throw ERROR\_THROW(130); // Ошибка польской записи

}

}

}

for (int i = 0; i < t.lextable.size; i++) {

if (t.lextable.table[i].lexema == '+' || t.lextable.table[i].lexema == '-' ||

t.lextable.table[i].lexema == '\*' || t.lextable.table[i].lexema == '/' ||

t.lextable.table[i].lexema == '\\' || t.lextable.table[i].lexema == ':' ||

t.lextable.table[i].lexema == '%') {

t.lextable.table[i].data = t.lextable.table[i].lexema;

t.lextable.table[i].lexema = LEX\_OPERATOR;

}

}

// Вывод таблицы в формате с несколькими лексемами на одной строке

OutputLexTableFormatted(t.lextable, outputFilename);

}

}

Листинг 5– Программная реализация польской записи

------------------- Таблица лексем --------------------

1: dtfi(ti){

2: dti;

3: ivl;

4: o(ibi)[

5: iviiv;

6: ];

7: ri;

8: };

9: m{

10: etfi(ti);

11: etfi(ti);

12: etfi(ti);

13: dti;

14: ivil@1;

15: dti;

16: ivil@1;

17: dti;

18: ivil@1;

19: dti;

20: dti;

21: dti;

22: ivl;

23: pi;

24: pi;

25: ivl;

26: pi;

27: pi;

28: ivl;

29: pi;

30: pi;

31: pl;

32: dti;

33: ivil@1;

34: pi;

35: dti;

36: ivl;

37: dti;

38: ivl;

39: u(ibi)[

40: ivilv;

41: pi;

42: ];

43: dti;

44: ivl;

45: pl;

46: pi;

47: dti;

48: ivl;

49: dti;

50: ivilv;

51: dti;

52: ivilv;

53: pl;

54: pi;

55: pl;

56: pi;

57: rl;

58: }

Листинг 6 – Таблица лексем после польской записи

## ПРИЛОЖЕНИЕЖ

.586

.model flat, stdcall

includelib libucrt.lib

includelib kernel32.lib

includelib ../Debug/SMD\_2024Lib.lib

ExitProcess PROTO :DWORD

getCurrentDate PROTO :DWORD

getCurrentMinuts PROTO :DWORD

getCurrentHour PROTO :DWORD

outputIntNum PROTO :DWORD

outputSymb PROTO :BYTE

outputLine PROTO :DWORD

.stack 4096

.const

divideOnZeroExeption BYTE "Попытка деления на ноль.", 0 ;STR, для вывода ошибки при делении на ноль

Demo$LEX1 DWORD 5 ;INT

Main$LEX2 DWORD 1 ;INT

Main$LEX5 BYTE "Date", 0 ;STR

Main$LEX6 BYTE "Hours", 0 ;STR

Main$LEX7 BYTE "Minuts", 0 ;STR

Main$LEX8 BYTE "Demo Function", 0 ;STR

Main$LEX9 DWORD 2 ;INT

Main$LEX10 DWORD 10 ;INT

Main$LEX11 DWORD 200 ;INT

Main$LEX12 DWORD 3 ;INT

Main$LEX13 BYTE 'k' ;CHR

Main$LEX14 BYTE "Symbol", 0 ;STR

Main$LEX15 DWORD 18 ;INT

Main$LEX18 BYTE "18<<1:", 0 ;STR

Main$LEX19 BYTE "18>>1:", 0 ;STR

Main$LEX20 DWORD 0 ;INT

.data

Demob DWORD 0 ;INT

Mainnum1 DWORD 0 ;INT

Mainnum2 DWORD 0 ;INT

Mainnum3 DWORD 0 ;INT

Mainstr1 DWORD 0 ;STR

Mainstr2 DWORD 0 ;STR

Mainstr3 DWORD 0 ;STR

Mainfunc DWORD 0 ;INT

Mainx DWORD 0 ;INT

Mainy DWORD 0 ;INT

Mainsym BYTE 0 ;CHR

Mainn DWORD 0 ;INT

Mainm DWORD 0 ;INT

Mainl DWORD 0 ;INT

.code

$Demo PROC uses ebx ecx edi esi , Demoa: DWORD

; String #4 :ivl

push Demo$LEX1

pop Demob

If17Start:

mov eax, Demoa

mov ebx, Demob

cmp eax, ebx

ja If17End

; String #9 :iviiv

push Demoa

push Demob

pop ebx

pop eax

add eax, ebx

push eax

pop Demob

If17End:

mov eax, Demob

ret

$Demo ENDP

main PROC

; String #23 :ivil@1

invoke getCurrentDate, Main$LEX2

push eax ;результат функции

pop Mainnum1

; String #25 :ivil@1

invoke getCurrentHour, Main$LEX2

push eax ;результат функции

pop Mainnum2

; String #27 :ivil@1

invoke getCurrentMinuts, Main$LEX2

push eax ;результат функции

pop Mainnum3

; String #35 :ivl

push offset Main$LEX5

pop Mainstr1

push Mainstr1

CALL outputLine

push Mainnum1

CALL outputIntNum

; String #38 :ivl

push offset Main$LEX6

pop Mainstr2

push Mainstr2

CALL outputLine

push Mainnum2

CALL outputIntNum

; String #41 :ivl

push offset Main$LEX7

pop Mainstr3

push Mainstr3

CALL outputLine

push Mainnum3

CALL outputIntNum

push offset Main$LEX8

CALL outputLine

; String #47 :ivil@1

invoke $Demo, Main$LEX9

push eax ;результат функции

pop Mainfunc

push Mainfunc

CALL outputIntNum

; String #50 :ivl

push Main$LEX10

pop Mainx

; String #52 :ivl

push Main$LEX11

pop Mainy

While174Start:

mov eax, Mainx

mov ebx, Mainy

cmp eax, ebx

ja While174End

; String #55 :ivilv

push Mainx

push Main$LEX12

pop ebx

pop eax

mul ebx

push eax

pop Mainx

push Mainx

CALL outputIntNum

jmp While174Start

While174End:

; String #61 :ivl

movzx eax, Main$LEX13

push eax

pop eax

mov Mainsym, al

push offset Main$LEX14

CALL outputLine

push eax

movzx eax, Mainsym

push eax

CALL outputSymb

pop eax

; String #66 :ivl

push Main$LEX15

pop Mainn

; String #68 :ivilv

push Mainn

push Main$LEX2

pop ebx

pop eax

push ecx ; сохраняем данные регистра ecx

mov ecx, ebx

SHL eax, cl

pop ecx

push eax

pop Mainm

; String #70 :ivilv

push Mainn

push Main$LEX2

pop ebx

pop eax

push ecx ; сохраняем данные регистра ecx

mov ecx, ebx

SHR eax, cl

pop ecx

push eax

pop Mainl

push offset Main$LEX18

CALL outputLine

push Mainl

CALL outputIntNum

push offset Main$LEX19

CALL outputLine

push Mainm

CALL outputIntNum

mov eax, Main$LEX20

jmp endPoint

div\_by\_0:

push offset divideOnZeroExeption

CALL outputLine

endPoint:

invoke ExitProcess, eax

main ENDP

end main

Листинг 7 – Сгенерированный код на ассемблере