МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра программной инженерии

Специальность 6-05-0612-01 Программное инженерия

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора DAA-2024»

Выполнил студент Дубина Артём Александрович

(Ф.И.О.)

Руководитель проекта асс. Ромыш Александра Сергеевна

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

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

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

Консультанты асс. Ромыш Александра Сергеевна

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

Нормоконтролер асс. Ромыш Александра Сергеевна

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

Курсовой проект защищен с оценкой

Минск 2024

**Содержание**

[Введение 7](#_Toc185351169)

[**1.1 Характеристика языка программирования** 8](#_Toc185351170)

[**1.2 Определение алфавита языка программирования** 8](#_Toc185351171)

[**1.3 Применяемые сепараторы** 8](#_Toc185351172)

[**1.4 Применяемые кодировки** 9](#_Toc185351173)

[**1.5 Типы данных** 10](#_Toc185351174)

[**1.6 Преобразование типов данных** 11](#_Toc185351176)

[**1.7 Идентификаторы** 11](#_Toc185351177)

[**1.8 Литералы** 11](#_Toc185351179)

[**1.9 Объявления данных** 12](#_Toc185351181)

[**1.10 Инициализация данных** 13](#_Toc185351182)

[**1.11 Инструкции языка** 13](#_Toc185351183)

[**1.12 Операции языка** 14](#_Toc185351184)

[**1.13 Выражения и их вычисления** 14](#_Toc185351185)

[**1.15 Область видимости идентификаторов** 15](#_Toc185351187)

[**1.16 Семантические проверки** 15](#_Toc185351188)

[**1.17 Распределение оперативной памяти на этапе выполнения** 16](#_Toc185351189)

[**1.18 Стандартная библиотека и её состав** 16](#_Toc185351191)

[**1.19 Ввод и вывод данных** 17](#_Toc185351192)

[**1.20 Точка входа** 17](#_Toc185351193)

[**1.21 Препроцессор** 17](#_Toc185351194)

[**1.22 Соглашения о вызовах** 17](#_Toc185351195)

[**1.23 Объектный код** 17](#_Toc185351196)

[**1.24 Классификация сообщений транслятора** 17](#_Toc185351197)

[**1.25 Контрольный пример** 18](#_Toc185351198)

[2 Структура транслятора 19](#_Toc185351199)

[**2.1 Компоненты транслятора, их назначение и принципы взаимодействия** 19](#_Toc185351200)

[**2.2 Перечень входных параметров транслятора** 20](#_Toc185351201)

[**2.3 Перечень протоколов, формируемых транслятором, и их содержимое** 21](#_Toc185351202)

[3 Разработка лексического анализатора 22](#_Toc185351203)

[**3.1 Структура лексического анализатора** 22](#_Toc185351204)

[**3.2 Контроль входных символов** 22](#_Toc185351205)

[**3.3 Удаление избыточных символов** 23](#_Toc185351206)

[**3.4 Перечень ключевых слов** 24](#_Toc185351207)

[**3.5 Основные структуры данных** 25](#_Toc185351208)

[**3.6 Принцип обработки ошибок** 25](#_Toc185351210)

[**3.7 Структура и перечень сообщений лексического анализатора** 25](#_Toc185351211)

[**3.8 Параметры лексического анализатора и режимы его работы** 26](#_Toc185351212)

[**3.9 Алгоритм лексического анализа** 26](#_Toc185351213)

[**3.10 Контрольный пример** 27](#_Toc185351214)

[4 Разработка синтаксического анализатора 28](#_Toc185351215)

[**4.1 Структура синтаксического анализатора** 28](#_Toc185351216)

[**4.2 Контекстно свободная грамматика, описывающая синтаксис языка** 28](#_Toc185351217)

[**4.3 Построение конечного магазинного автомата** 30](#_Toc185351218)

[**4.4 Основные структуры данных** 31](#_Toc185351219)

[**4.5 Описание алгоритма синтаксического разбора** 31](#_Toc185351220)

[**4.6 Структура и перечень сообщений синтаксического анализатора** 31](#_Toc185351221)

[**4.7 Параметры синтаксического анализатора и режимы его работы** 32](#_Toc185351223)

[**4.8 Принцип обработки ошибок** 32](#_Toc185351224)

[**4.9 Контрольный пример** 32](#_Toc185351226)

[5 Разработка семантического анализатора 33](#_Toc185351227)

[**5.1 Структура семантического анализатора** 33](#_Toc185351228)

[**5.2 Функции семантического анализатора** 33](#_Toc185351230)

[**5.3 Структура и перечень сообщений семантического анализатора** 33](#_Toc185351231)

[**5.4 Принцип обработки ошибок** 34](#_Toc185351233)

[**5.5 Контрольный пример** 34](#_Toc185351234)

[6 Преобразование выражений 35](#_Toc185351235)

[**6.1 Выражения, допускаемые языком** 35](#_Toc185351236)

[**6.2 Польская запись и принцип ее построения** 35](#_Toc185351238)

[**6.3 Программная реализация обработки выражений** 36](#_Toc185351240)

[**6.4 Контрольный пример** 36](#_Toc185351241)

[7 Генерация кода 37](#_Toc185351242)

[**7.1 Структура генератора кода** 37](#_Toc185351243)

[**7.2 Представление типов данных в оперативной памяти** 37](#_Toc185351244)

[**7.3 Статическая библиотека** 38](#_Toc185351245)

[**7.4 Особенности алгоритма генерации кода** 38](#_Toc185351246)

[**7.5 Входные параметры, управляющие генерацией кода** 39](#_Toc185351247)

[**7.6 Контрольный пример** 39](#_Toc185351248)

[8 Тестирование транслятора 40](#_Toc185351249)

[**8.1 Общие положения** 40](#_Toc185351250)

[**8.2 Результаты тестирования** 40](#_Toc185351251)

[Заключение 43](#_Toc185351252)

[Список использованных источников 44](#_Toc185351253)

[ПРИЛОЖЕНИЕ А 45](#_Toc185351254)

[ПРИЛОЖЕНИЕ Б 47](#_Toc185351255)

[ПРИЛОЖЕНИЕ В 49](#_Toc185351256)

[ПРИЛОЖЕНИЕ Г 51](#_Toc185351257)

[ПРИЛОЖЕНИЕ Д 62](#_Toc185351258)

[ПРИЛОЖЕНИЕ Е 67](#_Toc185351259)

# **Введение**

Задачей данного курсового проекта была поставлена разработка транслятора своего языка программирования DAA-2024. Он предназначен для работы с консолью, выполнения простейших действий, побитовых и арифметических операций над числами, а также для преобразования строк.

Основная задача транслятора заключается в преобразовании исходного кода, написанного на данном языке программирования, в формат, понятный компьютеру. Для реализации этой задачи выбран подход, при котором исходный код трансформируется в код на языке ассемблера. Ассемблер — это машинно-ориентированный язык, представляющий собой удобочитаемую запись машинных команд, доступную для понимания человеком.

Компилятор языка DAA-2024 состоит из следующих составных частей:

– лексический и семантический анализаторы;

– синтаксический анализатор;

– генератор исходного кода на языке ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

– разработка спецификации языка программирования;

– разработка структуры транслятора;

– разработка лексического и семантического анализаторов;

– разработка синтаксического анализатора;

– преобразование выражений;

– генерация кода на язык ассемблера;

– тестирование транслятора.

**1 Спецификация языка программирования**

**1.1 Характеристика языка программирования**

Язык программирования DAA-2024 является процедурным, строго типизированным, компилируемым. Он транслируется в язык ассемблера в 2 этапа: сначала исходный код транслируется в байт-код (промежуточное представление), а затем происходит перевод байт-кода в язык ассемблера.

Процедурный язык программирования — это язык высокого уровня, основанный на разбиении программы на модули, такие как процедуры и функции. Такой подход упрощает разработку и поддержку, делая код более структурированным.

Программа состоит из команд, описывающих шаги решения задачи, сгруппированных в процедуры или функции. Это позволяет повторно использовать код, улучшая модульность и масштабируемость.

Строго типизированный язык программирования — это язык, где переменные имеют строго определённые типы данных. Он запрещает смешивание различных типов в выражениях и исключает выполнение автоматических неявных преобразований типов.

Компилируемый язык программирования — это язык, исходный код которого преобразуется компилятором в машинный код или промежуточное представление на другом языке. Этот процесс позволяет программе выполняться непосредственно на целевой платформе, обеспечивая высокую производительность.

DAA-2024 предназначен для работы с консолью, выполнения простейших действий, арифметических и побитовых операций над числами, а также для преобразования строк.

**1.2 Определение алфавита языка программирования**

Алфавит языка программирования — это фиксированный набор основных символов, из которых должен составляться любой текст на этом языке. С помощью этих символов могут быть записаны идентификаторы, выражения и операторы языка.

Алфавит языка DAA-2024 состоит из следующих множеств символов:

* латинские символы верхнего и нижнего регистра: {A,B,C,…,Z; a,b,c,…,z};
* цифры: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
* знаки арифметических и побитовых операций: {+, -, \*, /, %, &, |, ~};
* знаки-сепараторы и слова-сепараторы: {(), {}, [], ;, :, =, !, <, >}.

**1.3 Применяемые сепараторы**

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

Таблица 1.1 — Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение сепаратора |
| { } | Программный блок |
| ; | Разделитель инструкций |
| ( ) | Параметры, приоритетность операций |
| < > | Блок условной конструкции |
| , | Разделитель параметров в функции |
| = | Оператор проверки на равенство. Используется в условных конструкциях |
| : | Оператор присваивания |
| +, -, \*, / | Арифметические операции |
| &, |, ~ | Побитовые операции |

Символы-сепараторы — это символы, предназначенные для разделения различных лексических единиц или функциональных компонентов в исходном коде программы.

**1.4 Применяемые кодировки**

В основе алфавита DAA-2024 лежит таблица символов Windows-1251, которая представлена на рисунке 1.1.



Рисунок 1.1 — Таблица кодировки Windows-1251

Windows-1251 — набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для русских версий Microsoft Windows до 11-й версии. Windows-1251 выгодно отличается от других 8‑битных кириллических кодировок (таких как CP866, KOI8-R и ISO 8859-5) наличием практически всех символов, использующихся в русской типографике для обычного текста (отсутствует только значок ударения).

**1.5 Типы данных**

В языке DAA-2024 допускается использование фундаментальных типов данных. Есть 3 типа данных: беззнаковый целочисленный, логический и строковый.

Описание типов данных, предусмотренных в данным языке представлено в таблице 1.2.

Таблица 1.2 — Фундаментальные типы данных

|  |  |
| --- | --- |
| Тип данных | Описание |
| arinteger (беззнаковый целочисленный) | Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 2 байта.  Диапазон значений: 0 — 2 147 483 647.  Инициализация по умолчанию: 0.  Поддерживаемые операции:  + (бинарный) — оператор сложения;  - (бинарный) — оператор вычитания;  \* (бинарный) — оператор умножения;  / (бинарный) — оператор деления;  & (бинарный) — оператор побитового И;  | (бинарный) — оператор побитового ИЛИ;  ~ (бинарный) — оператор инверсии;  : (бинарный) — оператор присваивания.  = (бинарный) — оператор проверки на равенство. |
| boolean  (логический) | Фундаментальный тип данных, предназначенный для объявления логических переменных, которые могут иметь одно из двух значений — false или true (в таблице идентификаторов эти значения представлены как 0 и 1 соответственно).  В памяти такой тип занимает 1 байт.  Инициализация по умолчанию: false.  Операции с данными логического типа не предусмотрены. |

Окончание таблицы 1.2

|  |  |
| --- | --- |
| string (строковый) | Фундаментальный тип данных, предназначенный для объявления строк. Он используется для работы с последовательностями символов, заключенными в двойные кавычки, при этом каждый символ занимает 1 байт в памяти. Максимальная длина строки — 255 символов. Инициализация по умолчанию: строка нулевой длины. Операции с данными строкового типа включают присваивание строковому идентификатору строкового литерала или результата строковой функции, а также использование библиотечных функций. |

Фундаментальный тип данных — это базовый тип, вводимый самим языком программирования для определения конкретной формы данных. Он обладает уникальными характеристиками и фиксированным набором свойств, которые описаны в спецификации языка и определяют его поведение при выполнении операций.

**1.6 Преобразование типов данных**

В языке программирования DAA-2024 преобразование типов данных не поддерживается, так как данный язык является строго типизированным.

**1.7 Идентификаторы**

Для именования функций, параметров и переменных используются идентификаторы. Зарезервированных идентификаторов в языке не предусмотрено. Имя идентификатора не может совпадать с ключевыми словами языка или названиями функций стандартной библиотеки. Общее количество идентификаторов ограничивается размером таблицы идентификаторов. Идентификаторы должны начинаться с латинской буквы и могут включать цифры. Идентификаторы, объявленные внутри функционального блока, получают префикс, соответствующий имени функции, в которой они созданы. Максимальная длина идентификатора, включая префикс, составляет 12 символов.

**1.8 Литералы**

В языке программирования DAA-2024 предусмотрено три основных типа литералов: целые, символьные и строковые. Литералы представляют собой фиксированные значения, которые непосредственно используются в коде программы. Они служат для задания исходных данных и часто применяются при инициализации переменных, записи констант или передачи аргументов в функции. Краткое описание литералов рассматриваемого языка программирования представлено в таблице 1.3.

Таблица 1.3 — Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание литерала |
| Строковые | Состоят из последовательности символов (букв, цифр, знаков препинания и т. д.), заключенной в двойные кавычки, инициализируются пустой строкой. |
| Целые | Последовательность цифр 0…9. Могут быть представлены как в десятичном, так и в восьмеричном (первые символы — 0o) представлении. |

Эти литералы формируют базу для работы с данными в языке **DAA-2024** и обеспечивают удобство записи значений прямо в коде.

**1.9 Объявления данных**

В языке программирования DAA-2024 переменная должны быть объявлена до ее использования. Областью видимости переменной является блок функции, в которой она определена. Вне блока функции определение функции запрещено. Не допустимо объявление глобальных переменных. Конструкция для объявления переменных:

spawn <тип данных> <идентификатор>;

Примеры объявления переменных на языке DAA-2024:

Листинг 1.1

spawn boolean bo;

spawn arinteger ar;

spawn string str;

Листинг 1.1 — Объявления переменных

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

Пример объявления функции на языке DAA-2024:

Листинг 1.2

arinteger process go(arinteger a)

{

spawn arinteger sum;

sum : a + a;

out sum;

};

Листинг 1.2— Объявление функции

Каждая переменная получает префикс — название функции, в которой она объявлена.

**1.10 Инициализация данных**

После объявлении переменной допускается инициализация данных. Объектами-инициализаторами могут быть идентификаторы, литералы, выражения и вызов функции. Переменной можно присвоить значение с помощью оператора присваивания (:). Он является бинарным, так что тут должно быть два операнда. Слева должна быть переменная, а справа выражение, дающее какое-либо значение.

Примеры инициализации переменных на языке DAA-2024:

Листинг 1.3

ar : 10;

str1 : "Hi!";

x : (go(ar) + 10) - (15 ~ 16);

Листинг 1.3— Инициализация переменных

Для неинициализированных переменных предусмотрены значения по умолчанию: **0** для целочисленных типов данных, пустая строка (строка длиной 0) для строкового типа данных и **false** для логического типа данных.

**1.11 Инструкции языка**

В языке программирования DAA-2024 применяются инструкции, представленные в таблице 1.4.

Таблица 1.4 — Инструкции языка

|  |  |
| --- | --- |
| Инструкция языка | Синтаксис |
| Объявление переменной | spawn <тип данных> <идентификатор>; |
| Объявление функции | <тип\_данных> process <идентификатор> (<тип\_данных> <идентификатор>, …) {…}; |
| Вызов функции | <идентификатор\_функции>(<идентификатор>|<литерал>,…) |
| Присвоение значения | <идентификатор> : <литерал>|<идентификатор>|<выражение>; |
| Вывод данных | write(<идентификатор> | <литерал>); |
| Вывод данных с переходом на новую строку | writeline(<идентификатор>|<литерал>); |

Окончание таблицы 1.4

|  |  |
| --- | --- |
| Возврат из функции | out <идентификатор>|<литерал>; |
| Условный оператор | if(<условие>) < <тело\_оператора> >; |

Тело условного оператора не может быть пустым. Любая функция должна возвращать значение.

**1.12 Операции языка**

Язык программирования DAA-2024 может выполнять побитовые и арифметические операции, представленные в таблице 1.5.

Таблица 1.5 — Приоритетности операций языка программирования DAA-2024

|  |  |
| --- | --- |
| Операция | Приоритетность |
| () | 1 |
| ~ (побитовая) | 5 |
| | (побитовая) | 2 |
| & (побитовая) | 3 |
| +; - (арифметические) | 4 |
| \*; / (арифметические) | 5 |

Операции языка применимы исключительно к целочисленному типу данных. Для строкового и логического типа операции языка не предусмотрены.

**1.13 Выражения и их вычисления**

Вычисление выражений является одной из ключевых задач любого языка программирования. В языке DAA-2024 выражения составляются согласно следующим правилам:

Скобки могут применяться для изменения приоритета операций в выражении.

Все выражения записываются в одну строку без использования переносов.

Невозможно записать два оператора без операндов между ними.

В выражении допускается использование вызова функции, за исключением булевских выражений.

Для булевских выражений можно использовать либо булевские переменные, либо операторы сравнения двух целочисленных значений. Однако вызов функции в этих выражениях запрещён.

Не допускается использование двух подряд идущих побитовых операций в выражении.

Каждое выражение перед генерацией кода преобразуется в польскую запись для удобства последующего вычисления на языке ассемблера.

Кроме того, выражения могут быть использованы после операции присваивания или после оператора возврата значения out.

**1.14 Конструкции языка**

Программа на языке DAA-2024 оформляется в виде функций пользователя и главной функции.

Ключевые программные конструкции языка программирования представлены в таблице 1.6.

Таблица 1.6 — Программные конструкции языка

|  |  |
| --- | --- |
| Конструкция | Представление в языке |
| Главная функция | Program  {  …  };  Является точкой входа в программу. |
| Функция | <тип\_данных> process <идентификатор> (<идентификатор>, …)  {  …  out <идентификатор> | <литерал> | <выражение>;  }; |
| Условный оператор | if(<условие>)  <  <тело\_оператора>  >; |

Для обеспечения качественного и понятного кода рекомендуется следовать правилам структурированного программирования, что способствует улучшению его читаемости и поддерживаемости.

**1.15 Область видимости идентификаторов**

Область видимости в языке DAA-2024 организована по принципу сверху вниз. Переменные, объявленные в одной функции, не доступны в других. Все объявления и манипуляции с переменными осуществляются внутри блоков кода. Каждая переменная или параметр функции получают префикс, совпадающий с именем функции, в которой они были созданы.

Идентификаторы в языке являются исключительно локальными и должны быть объявлены внутри соответствующей функции. Отсутствуют глобальные переменные. Параметры функции доступны только в пределах её тела.

**1.16 Семантические проверки**

Перечень семантических проверок, предусмотренных языком, приведен в таблице 1.7.

Таблица 1.7 — Перечень семантических проверок

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Наличие входной точки в программу (функции Program) |
| 2 | Операнды в операторах ветвления и выхода из функции должны быть целочисленного типа |
| 3 | Тип данных передаваемых значений в функцию должен совпадать с типом параметров при её объявлении |
| 4 | Идентификатор должен быть объявлен до его использования. |
| 5 | Операнды в арифметическом выражении не могут быть различных типов |
| 6 | Тип возвращаемого функцией значения должен совпадать с типом функции |
| 7 | Идентификаторы функций не должны повторяться |

Если семантическая проверка не проходит, то в файл журнала записывается соответствующая ошибка и дальнейшее выполнение невозможно.

**1.17 Распределение оперативной памяти на этапе выполнения**

Транслированный код использует две области памяти: сегмент констант для хранения всех литералов и сегмент данных для переменных и параметров функций. Локальная область видимости в исходном коде определяется с помощью правил именования идентификаторов и префиксов, что обеспечивает их локальность на уровне исходного кода.

**1.18 Стандартная библиотека и её состав**

В языке DAA-2024 присутствует стандартная библиотека, которая реализована на языке C++. Содержимое библиотеки и описание функций представлено в таблице 1.8.

Таблица 1.8 — Стандартная библиотека языка DAA-2024

|  |  |
| --- | --- |
| Функция | Описание |
| string concat(string a, string b); | Строковая функция. Принимает в качестве параметров 2 строки. Копирует вторую строку к концу первой строки. |
| string copy(string a, string b); | Строковая функция. Принимает в качестве параметров 2 строки. Копирует содержимое второй строки в первую. |
| boolean comp(string a, string b); | Логическая функция. Принимает в качестве параметров 2 строки. Если строки равны, возвращает true, иначе false. |

Стандартная библиотека автоматически подключается при трансляции исходного кода.

**1.19 Ввод и вывод данных**

Вывод данных осуществляется с помощью ключевого слова write/writeline. На вывод можно передавать как непосредственно литералы, так и переменные, вместо которых будут выведены их значения.

Примеры вывода данных на языке DAA-2024:

Листинг 1.4

writeline("Hello!");

write(x);

Листинг 1.4— Вывод данных

Языком программирования DAA-2024 ввод данных не поддерживается.

**1.20 Точка входа**

В языке DAA-2024 каждая программа должна содержать главную функцию (точку входа) Program, с первой инструкции которой начнётся последовательное выполнение команд программы. Наличие в исходном коде более одной точки входа недопустимо.

**1.21 Препроцессор**

В языке DAA-2024 препроцессор, предназначенный для обработки данных перед их передачей транслятору, отсутствует.

**1.22 Соглашения о вызовах**

В языке вызов функций происходит по соглашению о вызовах \_\_stdcall. Особенности \_\_stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

**1.23 Объектный код**

Исходный код языка транслируется сначала в язык ассемблера, а затем - в объектный код.

**1.24 Классификация сообщений транслятора**

Сообщения транслятора приведены в таблице 1.9.

Таблица 1.9 — Классификация сообщений транслятора

|  |  |
| --- | --- |
| Интервал | Описание ошибок |
| 0-1 | Системные ошибки |

Окончание таблицы 1.9

|  |  |
| --- | --- |
| 100-112 | Ошибки параметров |
| 113-119 | Ошибки лексического анализа |
| 600-607 | Ошибки синтаксического анализа |
| 120-129 | Ошибки семантического анализа |

Сообщения, генерируемые транслятором, определяют уровень его информативности. Они должны предоставлять пользователю максимально полную и точную информацию о допущенных ошибках в процессе написания программы.

**1.25 Контрольный пример**

Контрольный пример программы на языке DAA-2024 можно просмотреть в приложении А.

**2 Структура транслятора**

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

В языке DAA-2024 исходный код транслируется в язык Assembler. Транслятор разделён на несколько модулей, которые взаимодействуют друг с другом, выполняя определённые функции, описанные в пункте 2.1. Для генерации ассемблерного кода транслятор использует выходные данные лексического анализатора, включая таблицу лексем и таблицу идентификаторов. Указание выходных файлов осуществляется с помощью входных параметров транслятора, перечисленных в таблице 2.1. Общая структура транслятора языка DAA-2024 представлена на рисунке 2.1

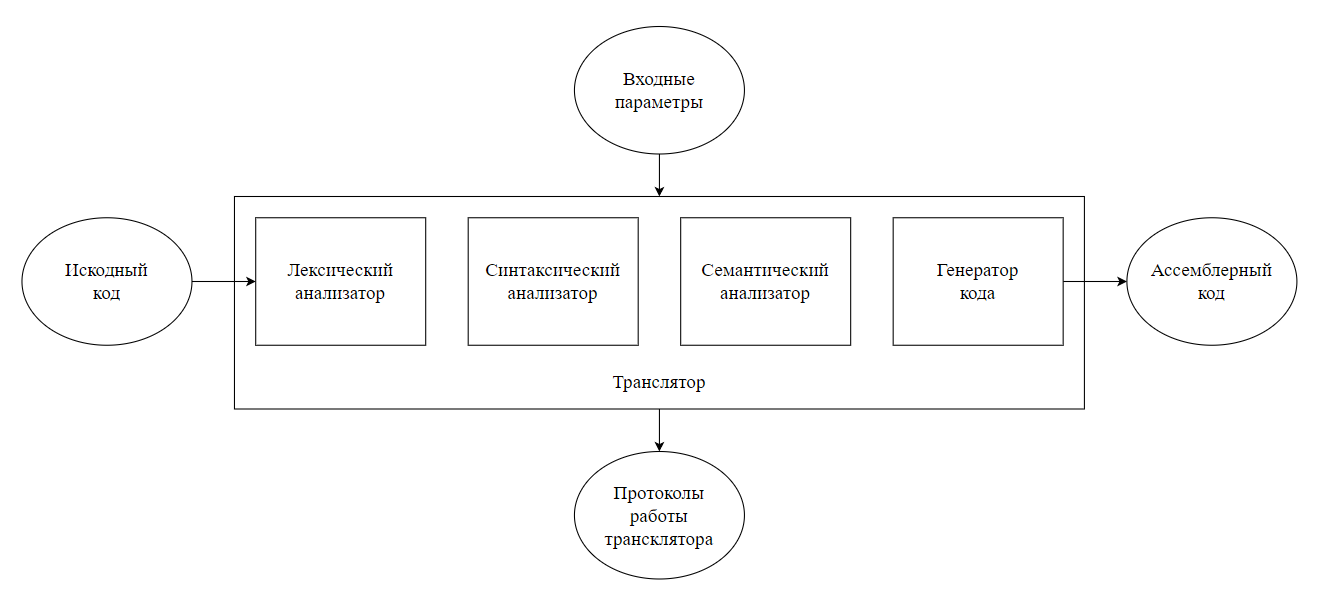


Рисунок 2.1 — Структура транслятора языка программирования DAA-2024

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, — лексическим анализатором (сканером). На вход лексического анализатора подаётся последовательность символов входного языка. Он производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив отдельных слов (в теории компиляции вместо термина «слово» часто используют термин «токен»). 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.

Таблица лексем (ТЛ) и таблица идентификаторов (ТИ) являются входом для следующей фазы компилятора — синтаксического анализа (разбора, или же парсера).

Цели лексического анализатора:

− убрать все лишние пробелы;

− выполнить распознавание лексем;

− построить таблицу лексем и таблицу идентификаторов;

− при неуспешном распознавании или обнаружении некоторых ошибок во входном тексте выдать сообщение об ошибке.

Синтаксический анализатор — часть компилятора, выполняющая синтаксический анализ, то есть проверку исходного кода на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией является дерево разбора

Семантический анализ в свою очередь является проверкой исходной программы DAA-2024 на семантическую согласованность с определением языка, т.е. проверяет правильность текста исходной программы с точки зрения семантики.

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 DAA-2024, прошедший все предыдущие этапы, в код на языке Ассемблер.

**2.2 Перечень входных параметров транслятора**

Входные параметры для запуска консольного приложения представлены в таблице 2.1.

Таблица 2.1 — Входные параметры транслятора языка DAA-2024

|  |  |  |
| --- | --- | --- |
| Параметр | Назначение | Тип |
| -in: | Указывает на файл с исходным кодом программы на языке DAA-2024. Исходный код содержится в файле с расширением «.txt» | Обязательный |
| -out: | Указывает имена выходных файлов. Если не указан явно, то имена файлов со сгенерированным кодом формируются конкатенацией имени файла исходного кода и постфиксом «.asm». | Необязательный |
| -log: | Указывает имя файла протокола. Если не указан явно, то имя протокола формируется конкатенацией имени файла исходного кода и постфикса «.log» | Необязательный |

Входные параметры транслятора представляют собой опции, передаваемые при компиляции программы. Они позволяют настроить различные аспекты процесса компиляции, предоставляя программисту возможность управлять её выполнением, оптимизировать производительность и использовать удобный механизм для отслеживания работы транслятора.

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

Перечень протоколов, формируемых транслятором языка программирования DAA-2024, и их назначение представлены в таблице 2.2.

Таблица 2.2 — Протоколы, формируемые транслятором языка DAA-2024

|  |  |
| --- | --- |
| Формируемый протокол | Описание протокола |
| ".log" | В файле находится информация о входных параметрах, количестве символов исходного кода. При наличии ошибки выводится ее краткое описание и соответствующая ей позиция в исходном коде. |
| ".asm" | Результат работы программы — файл, содержащий исходный код на языке ассемблера. |
| ".idtable" | В файле находится таблица идентификаторов |
| ".lextable" | В файле находится таблица лексем |
| ".polish" | В файле находится таблица лексем в польской записи |

Протокол работы нужен для отображения хода выполнения трансляции языка DAA-2024. Благодаря им пользователь может дополнительно обнаружить некорректно введенные данные или ошибки в исходном коде программы.

**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 последовательность символов входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка, которые называют лексическими единицами. 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. Каждому идентификатору и литералу в таблице лексем сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация. Структура лексического анализатора представлена на рисунке 3.1.

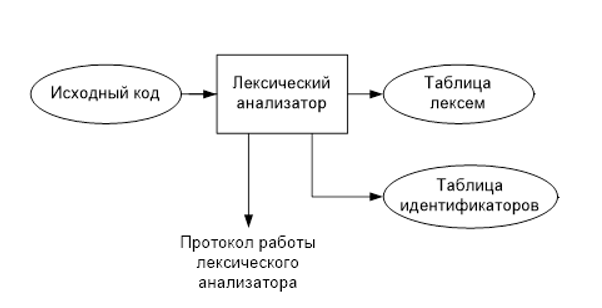


Рисунок 3.1 — Структура лексического анализатора DAA-2024

В результате формируется таблица лексем и таблица идентификаторов.

**3.2 Контроль входных символов**

Контроль входных символов в языках программирования представляет собой процесс проверки и валидации символов, вводимых пользователем или получаемых из внешних источников. Это важная задача, которая позволяет программе обрабатывать только допустимые символы и избегать ошибок или непредвиденного поведения.

Контроль входных символов может включать следующие аспекты: проверка типа символа, проверка наличия или отсутствия символов, проверка длины ввода, фильтрация символов, обработка ошибок.

Таблица для контроля входных символов представлена в листинге 3.1.

#define IN\_CODE\_TABLE {\

IN::P, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::P, IN::P, 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::P, IN::T, IN::T, IN::T, IN::T, IN::S, IN::S, IN::Q, IN::S, IN::S, IN::S, IN::S, IN::S, IN::S, IN::F, IN::S, \

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

Листинг 3.1

Принцип работы таблицы заключается в соответствии значения каждого элемента в шестнадцатеричной системе счисления значению в таблице кодировки Windows-1251.

Описание значения символов: T — разрешённый символ, F — запрещённый символ, S — сепаратор, P — пробелы, табуляция и переход на новую строку, Q — одинарная кавычка.

**3.3 Удаление избыточных символов**

Избыточными символами являются символы табуляции, символы конца строки, пробелы и символы, идущие после символа начала комментария (#), если он не включен в строковый литерал. Избыточные символы удаляются на этапе разбиения исходного кода на токены.

**3.4 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

|  |  |  |  |
| --- | --- | --- | --- |
| Тип цепочки | Цепочка | | Лексема |
| Ключевые слова | spawn | | m |
| arinteger | | t |
| string | | t |
| boolean | | t |
| process | | f |
| out | | o |
| write | | w |
| writeline | | w |
| Program | | p |
| if | | c |
| false | | t |
| true | | t |
| Иное | Идентификатор | | i |
| Литерал | | l |
| Сепараторы | ; | ; | |
| , | , | |
| Сепараторы | { | { | |
| } | } | |
| ( | ( | |
| ) | ) | |
| < | [ | |
| > | ] | |
| Операторы | + | x | |
| - | x | |
| / | x | |
| \* | x | |
| & | x | |
| | | x | |
| ~ | x | |
| = | x | |
| : | = | |

Пример реализации таблицы лексем представлен в приложении А. Также в приложении А находится таблица идентификаторов для контрольного примера.

**3.5 Основные структуры данных**

Основные структуры таблиц лексем и идентификаторов для языка DAA-2024 описаны в приложении А. Таблица лексем хранит лексему, её номер, присвоенный при разборе, номер строки в исходном коде, а также значение для арифметических операторов. В таблице идентификаторов содержится имя идентификатора, его номер в таблице лексем, тип данных, смысловой тип идентификатора и его значение.

**3.6 Принцип обработки ошибок**

Если в результате работы лексического анализатора найдена ошибка, анализатор добавляет информацию в таблицу ошибок, после чего главная функция записывает ошибки в файл протокола. Они включают следующую информацию: код ошибки, номер строки в коде, номер позиции в строке или только код ошибки.

При возникновении ошибки работа транслятора прекращается.

**3.7 Структура и перечень сообщений лексического анализатора**

Структура сообщений содержит информацию о номере сообщения, номер строки, где было вызвано сообщение в исходном коде, а также информацию об ошибке.

Перечень сообщений об ошибках работы лексического анализатора представлен в таблице 3.2.

Таблица 3.2 — Перечень сообщений лексического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 113 | Элемент не распознан |
| 114 | Ошибка при создании файла IT |
| 115 | Ошибка при создании файла LT |
| 116 | Ошибка при создании лексической таблицы (превышен максимальный размер) |
| 117 | Ошибка при создании таблицы идентификаторов (превышен максимальный размер) |
| 118 | Ошибка при получении строки лексической таблицы (нет элемента) |
| 119 | Ошибка при получении строки таблицы идентификаторов (нет элемента) |

Все ошибки являются критическими и приводят к прекращению работы транслятора и выводу диагностического сообщения в log-файл.

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

Входным параметром лексического анализа являются редактированный код исходного файла, пустая таблица лексем, пустая таблица идентификаторов, таблица ошибок.

Лексический анализатор обрабатывает код исходного файла, проверяя символы и слова на соответствие возможным цепочкам, после чего заполняет таблицу лексем и таблицу идентификаторов.

**3.9 Алгоритм лексического анализа**

Алгоритм работы лексического анализа заключается в последовательном распознавании и разборе цепочек исходного кода и заполнение таблиц идентификаторов и лексем.  Это основывается на работе конечных автоматов, которую можно представить в виде графов. В случае, если подходящий автомат не был обнаружен, запоминается номер строки, в которой находился этот токен и выводится сообщение об ошибке. Если токен разобран, то дальнейшие действия, которые будут с ним производиться, будут зависеть от того, чем он является. 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.

В случае, если токен является идентификатором, перед его именем записывается название функции, в которой он объявлен и после этого он заносится в таблицу идентификаторов.

В случае, если токен является литералом, то он заносится в таблицу идентификаторов в виде ln, где b — “L”, n — текущий индекс литерала.

Когда встречаем токен, являющийся ключевым словом, которое отвечает за тип данных или вид идентификатора, заносим лексему, соответствующую ему, в таблицу лексем и запоминаем тип данных или вид идентификатора, которому он соответствует.

В последствии, когда встречаем идентификатор, заносим его в таблицу идентификаторов с соответствующим ему типом данных и видом идентификатора, и именем вида “i\_f”, где i — имя идентификатора, f — имя функции, где объявлен идентификатор.

Граф КА для ключевого слова spawn представлен на рисунке 3.2.

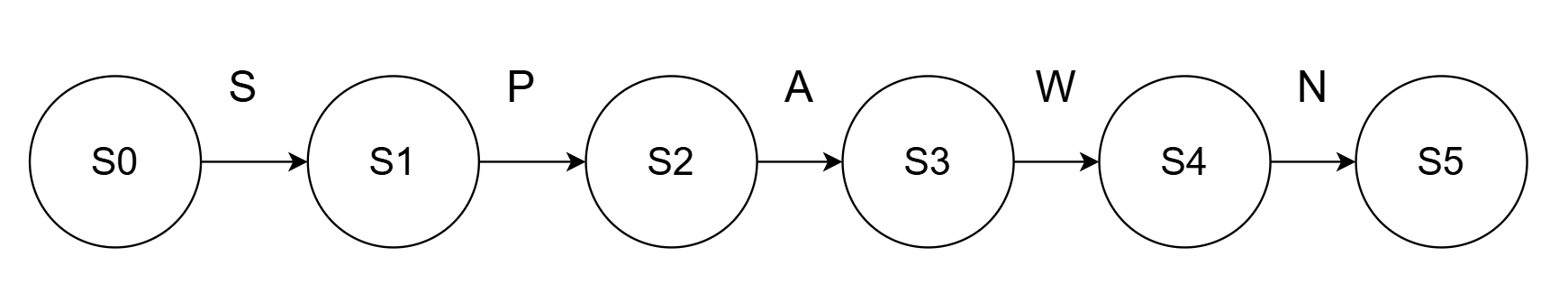


Рисунок 3.2 — Граф переходов для цепочки «spawn»

Здесь S0 — начальное состояние, S5 — конечное состояние автомата.

**3.10 Контрольный пример**

Результат работы лексического анализатора представлен в виде таблицы лексем и таблицы идентификаторов в приложении А.

Листинг 3.2

|  |
| --- |
| t | |1 |a  f | |1 |p  i |4 |1 |f  ( | |1 |(  t | |1 |a  i |5 |1 |s  ) | |1 |)  { | |2 |{  m | |3 |s  t | |3 |a  i |6 |3 |s  ; | |3 |;  i |6 |4 |s  = | |4 |:  i |5 |4 |s  x | |4 |+  i |5 |4 |s  ; | |4 |;  o | |5 |o  i |6 |5 |s  ; | |5 |;  } | |6 |}  ; | |6 |;  p | |7 |P  { | |8 |{  m | |9 |s |

Листинг 3.2 — Таблица лексем(фрагмент)

Структурированное представление всех лексем (элементарных единиц языка программирования), обнаруженных в исходном коде во время лексического анализа. Она служит промежуточным звеном между исходным кодом и дальнейшими этапами компиляции, такими как синтаксический и семантический анализ.

**4 Разработка синтаксического анализатора**

**4.1 Структура синтаксического анализатора**

Синтаксический анализатор является второй фазой работы транслятора. Назначением синтаксического анализаторы является распознавание синтаксических конструкций языка и формирование промежуточного кода. Исходными данными синтаксического анализатора являются таблицы лексем и идентификаторов. Свою работу начинает только при условии отсутствии ошибок в их общей структуре. Лексемы являются для синтаксического анализатора терминальными символами контекстно-свободной грамматики. Если в ходе его работы не было обнаружено ошибок, то формируется дерево разбора (промежуточное представление кода) и модифицируется протокол работы. Структура синтаксического анализатора представлена на рисунке 4.1.



Рисунок 4.1 — Структура синтаксического анализатора

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 DAA-2024 используется контекстно-свободная грамматика *G = <T, N, P, S>*, 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 P имеют вид:

1. , где ; (или , или );
2. , где — начальный символ, при этом если такое правило существует, то нетерминал  не встречается в правой части правил. TS — терминальные символы, которыми являются сепараторы, знаки арифметических операций и некоторые строчные буквы.

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

Таблица 4.1 — Перечень правил, составляющих грамматику языка и описание нетерминальных символов DAA-2024

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| S | p{N};  tfi(F){NoR;};S  tfi(F){oR;};S | Порождает правила, описывающие общую структуру программы |
| N | mti;  i=E;  mti;N  i=E;N  mtfi(F);N  w(R);  w(R);N  c(I)[N];N  c(I)[N];  fi(F);N};  i(W);N  oR; | Порождает правила, описывающие инструкции языка |
| E | i  l  (E)  i()  i(W)  iM  lM  (E)M  i()M  i(W)M | Порождает правила, описывающие выражения |
| M | xE  xEM | Порождает правила, описывающие побитовые и арифметические операции |
| F | ti  ti,F | Порождает правила, описывающие параметры шаблона функции |
| W | i  l  i,W  l,W | Порождает правила, описывающие параметры функции |
| R | l  i | Порождает правила, описывающие возвращаемый тип и операнды вывода в консоль |

Окончание таблицы 4.1

|  |  |  |
| --- | --- | --- |
| I | ixi  lxl  lxi  ixl  i  l | Порождает правила, описывающие условную конструкцию |

Правила языка DAA-2024 представлена в приложении Б.

**4.3 Построение конечного магазинного автомата**

Конечный автомат с магазинной памятью представляет собой семерку*M=<Q,V,Z,δ,q0,z0,F>*, описание которой представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
| *Q* | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
| *V* | Алфавит входных символов | Алфавит является множеством терминальных и нетерминальных символов, описание которых содержится в разделе 1.2 и в таблице 4.1. |
| *Z* | Алфавит специальных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека |
| *δ* | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
| *q0* | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики (нетерминальный символ S) |
| *z0* | Начальное состояние автомата | Символ маркера дна стека ($) |
| *F* | 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 |

Пример работы магазинного автомата для цепочки представлен на риc. 4.2.

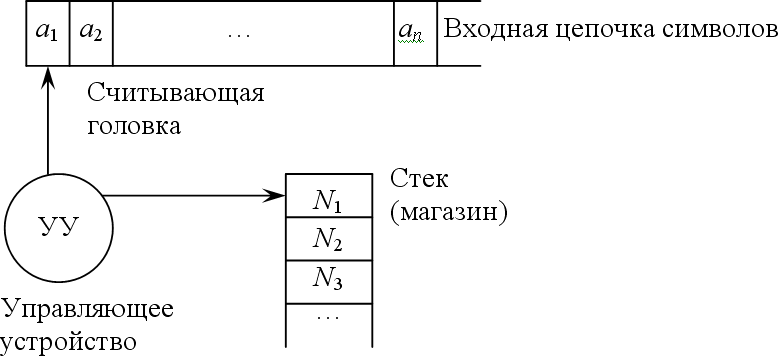


Рисунок 4.2 — Автомат с магазинной памятью

Структура магазинного автомата и функции, реализованные для его работы, прокомментированы показаны в приложении В.

**4.4 Основные структуры данных**

Основные структуры данных синтаксического анализатора представляются в виде структуры магазинного конечного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка DAA-2024. Данные структуры представлены в приложении В.

**4.5 Описание алгоритма синтаксического разбора**

Принцип работы автомата следующий:

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

**4.6 Структура и перечень сообщений синтаксического анализатора**

Перечень сообщений синтаксического анализатора представлен в таблице 4.3.

Таблица 4.3 — Перечень сообщений синтаксического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 600 | Неверная структура программы |
| 601 | Ошибочный оператор или функция не возвращает значение |
| 602 | Ошибка в выражениях |
| 603 | Ошибка в операторах выражения |

Окончание таблицы 4.3

|  |  |
| --- | --- |
| 604 | Ошибка в параметрах функции |
| 605 | Ошибка в параметрах функции |
| 606 | Значением данного оператора может быть только переменная или литерал |
| 607 | Ошибка в условии условного оператора |

Количество правил соответствует количеству ошибок, которые может обнаружить и обработать транслятор. Чем больше предусмотрено правил, тем выше вероятность выявления различных видов ошибок и предоставления программисту подробной информации для их исправления.

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

Входным параметром синтаксического анализатора является таблица лексем, полученная на этапе лексического анализа, а также правила контекстно-свободной грамматики в форме Грейбах.

Выходными параметрами являются трассировка прохода таблицы лексем (при наличии разрешающего ключа) и правила разбора, которые выводятся в консоль.

**4.8 Принцип обработки ошибок**

Синтаксический анализатор выполняет разбор исходной последовательности лексем до тех пор, пока не дойдёт до конца цепочки лексем или не найдёт ошибку. Тогда анализ останавливается и выводится сообщение об ошибке (если она найдена).

**4.9 Контрольный пример**

Пример разбора исходного кода на языке DAA-2024 синтаксическим анализатором представлен в приложении Г.

**5 Разработка семантического анализатора**

**5.1 Структура семантического анализатора**

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет необходимые ошибки. Некоторые проверки (такие как проверка на единственность точки входа, проверка на предварительное объявление переменной) осуществляются в процессе лексического анализа. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1 — Структура семантического анализатора

Семантический анализатор обеспечивает правильность и согласованность семантической структуры программы, написанной на языке DAA-2024.

**5.2 Функции семантического анализатора**

Семантический анализатор выполняет проверку на основе правил языка, описанных в пункте 1.16. Он и есть та самая подпрограмма, которая занимается автоматическим приведением типов.

**5.3 Структура и перечень сообщений семантического анализатора**

Сообщения, формируемые семантическим анализатором, представлены в таблице 5.1.

Таблица 5.1 — Перечень сообщений семантического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 120 | Невозможно применить оператор к данному типу операнда |
| 121 | Тип возвращаемого значения не соответствует типу функции |
| 122 | Использована необъявленная переменная |
| 123 | Повторное объявление имени |
| 124 | Функция должна возвращать значение |
| 125 | Ошибка в условии условного оператора |
| 126 | Длина строкового литерала превышает допустимое значение |
| 127 | Ошибка в условии условного оператора |
| 128 | Несовпадение фактических и формальных параметров функции |
| 129 | Обнаружено больше 1 точки входа в программу |

Данный перечень сообщений представляет собой ценный инструмент для предоставления как визуальной, так и текстовой обратной связи. Он помогает эффективно выявлять, локализовать и устранять возможные проблемы, повышая общую точность и качество работы.

**5.4 Принцип обработки ошибок**

Семантический анализатор в случае возникновения ошибки заносит её в протокол. Следующий этап трансляции не будет запущен при возникновении ошибки. Семантический анализ начинает проверки уже на стадии лексического анализа, если на этой стадии обнаружены семантические ошибки — программа завершит свою работу.

**5.5 Контрольный пример**

Демонстрации ошибок, диагностируемых семантическим анализатором на разных этапах трансляции приведена в разделе 8.2.

# **6 Преобразование выражений**

**6.1 Выражения, допускаемые языком**

В языке DAA-2024 разрешено использовать выражения, работающие с целочисленными типами данных. В таких выражениях поддерживаются арифметические операции (сложение, вычитание, умножение, деление), побитовые операции (логическое И, ИЛИ, инверсия), а также вызовы функций, применяемые в качестве операндов арифметических выражений.

Приоритет операций представлен в таблице 6.1.

Таблица 6.1 — Приоритет операций в языке DAA-2024

|  |  |
| --- | --- |
| Операция | Приоритетность |
| () | 1 |
| ~ (побитовая) | 5 |
| | (побитовая) | 2 |
| & (побитовая) | 3 |
| +; - (арифметические) | 4 |
| \*; / (арифметические) | 3 |

Выражения и операции, допускаемые языком DAA-2024, подробно описаны в разделах 1.12 и 1.13.

**6.2 Польская запись и принцип ее построения**

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

Польская запись представляет собой альтернативный способ представления арифметических выражений, ключевым преимуществом которого является отсутствие необходимости в использовании скобок. Обратная польская запись, в свою очередь, — это форма записи математических и логических выражений, где операнды предшествуют операторам.

Алгоритм построения:

1. Исходная строка: выражение.
2. Результирующая строка: польская запись.
3. Стек: пустой.
4. Исходная строка просматривается слева направо.
5. Операнды переносятся в результирующую строку в порядке их следования.
6. Операция записывается в стек, если стек пуст или в вершине стека лежит отрывающая скобка.
7. Операция выталкивает все операции с большим или равным приоритетом в результирующую строку.
8. Запятая не помещается в стек, если в стеке операции, то все выбираются в строку.
9. Отрывающая скобка помещается в стек.
10. Закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются.
11. Закрывающая скобка, если поднят флаг функции, выталкивает все до открывающей скобки в обратном порядке и добавляет идентификатор функции в конце.
12. По концу разбора исходной строки все операции, оставшиеся в стеке, выталкиваются в результирующую строку.

Пример преобразования выражения в обратную польскую запись представлен в таблице 6.2.

Таблица 6.2 — Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Исходная строка | Результирующая строка | Стек |
| (a+b) - f(i) |  |  |
| +b) - f(i) | a | ( |
| b) - f(i) | a | (+ |
| ) - f(i) | ab | (+ |
| - f(i) | ab+ |  |
| f(i) | ab+c\* | - |
|  | ab+i@1- |  |

Подобная система приоритетов операций играет ключевую роль в языке программирования, гарантируя корректное выполнение вычислений и позволяя разработчикам управлять порядком выполнения операций. Это способствует созданию более понятного и предсказуемого кода, упрощая его чтение, сопровождение и поддержку.

**6.3 Программная реализация обработки выражений**

Программная реализация алгоритма преобразования выражений к польской записи с комментариями представлена в приложении Д.

**6.4 Контрольный пример**

Пример преобразования выражения к обратной польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления.

**7 Генерация кода**

**7.1 Структура генератора кода**

Заключительным этапом трансляции языка DAA-2024 является генерация кода. Генератор принимает на вход таблицу лексем, таблицу идентификаторов и дерево разбора. На выходе получается файл с исходным кодом на ассемблере, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода представлена на рисунке 7.1.

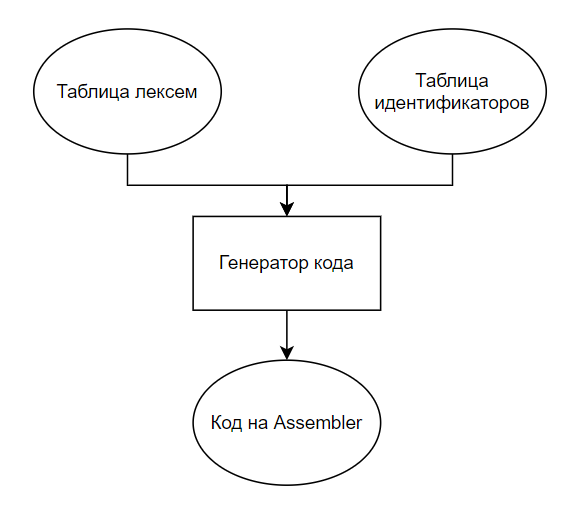


Рисунок 7.1 – Структура генератора кода

Генератор кода начинает свою работу только в том случае, если код прошёл предыдущие этапы без ошибок.

**7.2 Представление типов данных в оперативной памяти**

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

Таблица 7.1 – Соответствие типов идентификаторов языка DAA-2024 и языка Ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке DAA-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| arinteger  boolean | DWORD | Хранит целочисленный тип данных (для boolean: 0 или 1). |
| string | DWORD | Хранит указатель на начало строки. |
| Литерал | BYTE  DWORD | Литералы: строковые,  целочисленные |

Сгенерированный код на языке Assembler приведен в приложении Е.

**7.3 Статическая библиотека**

В языке DAA-2024 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера.

Стандартная библиотека находится в директории языка и при генерации кода подключается автоматически. Путь к библиотеке генерируется автоматически на стадии генерации кода.

**7.4 Особенности алгоритма генерации кода**

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

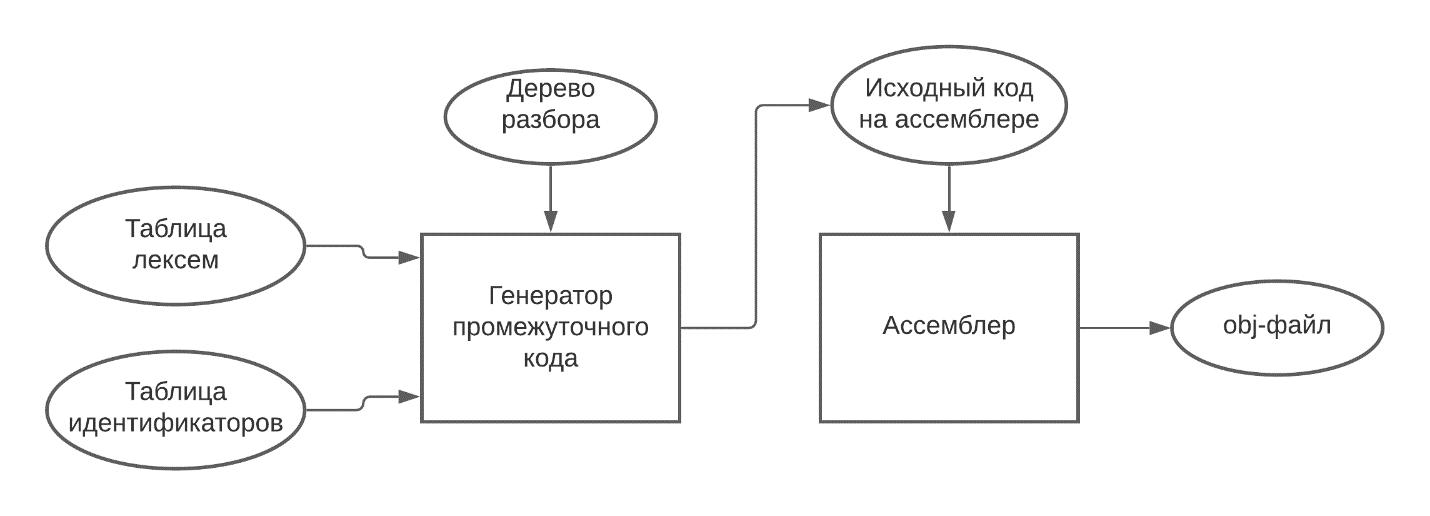


Рисунок 7.2 — Структура генератора кода

Алгоритм представляет собой механизм генерации кода на языке Assembler для языка программирования DAA-2024. Входными данными для генератора служат таблицы лексем и идентификаторов. Генератор осуществляет последовательный проход по таблице лексем, анализируя их тип и вызывая соответствующие функции для генерации необходимого машинного кода.

**7.5 Входные параметры, управляющие генерацией кода**

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

**7.6 Контрольный пример**

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

Листинг 6.1

|  |
| --- |
| .586P  .MODEL FLAT, STDCALL  includelib kernel32.lib  includelib libucrt.lib  includelib "..\Debug\Library.lib  EXTRN comp: PROC  EXTRN concat: PROC  EXTRN copy: PROC  EXTRN ConsNum: PROC  EXTRN ConsStr: PROC  EXTRN ConsBool: PROC  EXTRN consPause: proc  ExitProcess PROTO : DWORD  .STACK 4096  .CONST  L1 dd 1  L2 db "Arifmetika" , 0  L3 dd 3  L4 dd 4  L5 dd 5  L6 dd 2  L7 dd 16  L8 dd 52  L9 db "strike111" , 0  L10 db "stroke" , 0  L11 dd 7  L12 dd 0 |

Листинг 6.1– Результат генерации ассемблерного кода(фрагмент)

Результат генерации ассемблерного кода — это текстовая программа на языке ассемблера, которая является промежуточным представлением между высокоуровневым языком программирования и машинным кодом. Этот код создается генератором в процессе компиляции и предназначен для последующей трансляции в машинные инструкции процессора.

**8 Тестирование транслятора**

**8.1 Общие положения**

В результате обработки исходного кода программы, представленного в приложении А, транслятор языка DAA-2024 генерирует общий протокол работы, куда записываются все возникшие ошибки и предупреждения. Кроме того, все ошибки, возникшие на этапах лексического и семантического анализов, выводятся на консоль. Из ошибок, возникших на этапе синтаксического анализа, на консоль выводится только первая.

**8.2 Результаты тестирования**

В таблице 8.1 приведены ошибки, генерируемые в процессе считывания входного файла, а также в процессе всех анализов.

Таблица 8.1 — Результаты тестирования транслятора

|  |  |
| --- | --- |
| Исходный код | Генерируемая ошибка |
| Program {  out 0;  } | Ошибка 600: Синтаксическая ошибка. Неверная структура программы |
| Program {  spawn arinteger go;  go = восемь;  out go;  }; | Ошибка 113: Лексическая ошибка. Элемент не распознан |
| arinteger process fn(arinteger sos) {  spawn arinteger sum;  sum : sos + sos;  out gym;  }; | Ошибка 122: Семантическая ошибка. Использована необъявленная переменная |
| Program {  spawn boolean t;  t : true;  spawn boolean t;  out t;  }; | Ошибка 123: Семантическая ошибка. Повторное объявление имени |
| arinteger process fn(arinteger sos) {  spawn arinteger sum;  sum : sos + sos;  out "str";  }; | Ошибка 121: Семантическая ошибка. Тип возвращаемого значения не соответствует типу функции |

Продолжение таблицы 8.1

|  |  |
| --- | --- |
| arinteger process fn(arinteger sos) {  spawn string sum;  sum : sps + sos;  out sum;  }; | Ошибка 127: Семантическая ошибка. Несоответствие типов в выражении |
| arinteger process fn(arinteger sos) {  out sos;  };  Program {  spawn arinteger x;  x : (fn("k") + 4) - (15 & 16);  writeline(x);  out 0;  }; | Ошибка 128: Семантическая ошибка. Несовпадение фактических и формальных параметров функции |
| if(y)  <  spawn arinteger s;  s : 125;  if(s = "hi")  <  writeline(s);  >;  writeline(str2);  >; | Ошибка 125: Семантическая ошибка. Ошибка в условии условного оператора |
| arinteger process fn(string boolean sos)  {  spawn arinteger sum;  sum : sos + sos;  out sum;  }; | Ошибка 604: Синтаксическая ошибка. Ошибка в параметрах функции |
| arinteger arinteger fn(arinteger sos) {  out sos;  };  Program {  spawn arinteger x;  x : 5;  out fn(x);  }; | Ошибка 606: Синтаксическая ошибка. Значением данного оператора может быть только переменная или литерал |

Окончание таблицы 8.1

|  |  |
| --- | --- |
| Program {  spawn string sum;  sum : sos + sos;  }; | Ошибка 124: Cемантическая ошибка. Функция должна возвращать значение |
| if(y, t)  <  spawn arinteger s;  s : 7;  if(s = 7)  <  writeline(s);  >;  writeline(“stop”);  >; | Ошибка 607: Синтаксическая ошибка. Ошибка в условии условного оператора |
| arinteger process fn(arinteger sos) {  out sos;  };  Program {  spawn arinteger x;  x : (fn(5) + 4), - (15 & 16);  writeline(x);  out 0;  }; | Ошибка 603: Синтаксическая ошибка. Ошибка в операторах выражений |
| arinteger process fn(arinteger sos) {  out sos;  };  Program {  spawn arinteger x;  x : (fn(5) + 4) - (spawn);  writeline(x);  out 0;  }; | Ошибка 602: Синтаксическая ошибка. Ошибка в выражении |

Ошибки позволяют обнаружить структурные и синтаксические проблемы в программе на этапе компиляции, обеспечивая оперативную и точную диагностику. Это значительно упрощает процесс исправления и повышает надежность кода.

**Заключение**

В ходе проделанной работы был разработан транслятор для языка программирования DAA-2024.

Таким образом, были выполнены основные задачи данной курсовой работы:

1. сформулирована спецификация языка DAA-2024;
2. разработаны конечные автоматы и алгоритмы для реализация лексического анализатора;
3. разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
4. разработан семантический анализатор, осуществляющий проверку смысла используемых инструкций;
5. разработан транслятор с языка программирования DAA-2024 на язык Ассемблер;
6. проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка DAA-2024 включает:

* 4 арифметических операторов и 3 побитовых оператора для вычисления выражений;
* 3 типа данных;
* оператор вывода;
* подключаемую стандартную библиотеку;
* функции лексического сравнения, конкатенации и копирования;
* условную конструкцию;
* арифметику целочисленных литералов;
* структурированную систему для обработки ошибок.

Таким образом была достигнута поставленная цель по разработке компилятора DAA-2024, были учтены все требования, все задачи курсового проекта выполнены.

# **Список использованных источников**

1. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c.
2. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с. – Дата доступа: 12.11.2024.
3. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с
4. Википедия: Обратная польская запись [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Reverse\_Polish\_notation. – Дата доступа: 14.11.2024.

# ПРИЛОЖЕНИЕ А

|  |
| --- |
| arinteger process fn(arinteger sos)  {  spawn arinteger sum;  sum : sos + sos;  out sum;  };  Program  {  spawn boolean t;  t : true;  if (t) < writeline("Arifmetika"); >;  spawn arinteger e;  e : (3 + 4) \* (5 - 2) / 3;  writeline(e);  spawn arinteger x;  x : (fn(e) + 4) - (16 ~ 16);  writeline(x);  x : (3 | 4) ~ (2 & 3);  spawn arinteger eig;  eig : 0o64;  writeline(eig);  spawn string str1;  spawn string str2;  str1 : "strike111";  str2 : "stroke";  spawn string str3;  str3 : concat(str1, str2);  writeline(str3);  str3 : copy(str1, str2);  writeline(str3);  spawn boolean f;  f : comp(str1, str2);  writeline(f);  spawn boolean y;  y : true;  if(y)  <  spawn arinteger s;  s : 7;  if(s = 7)  <  writeline(s);  >;  >;  out 0;  }; |

Листинг A.1 — Контрольный пример

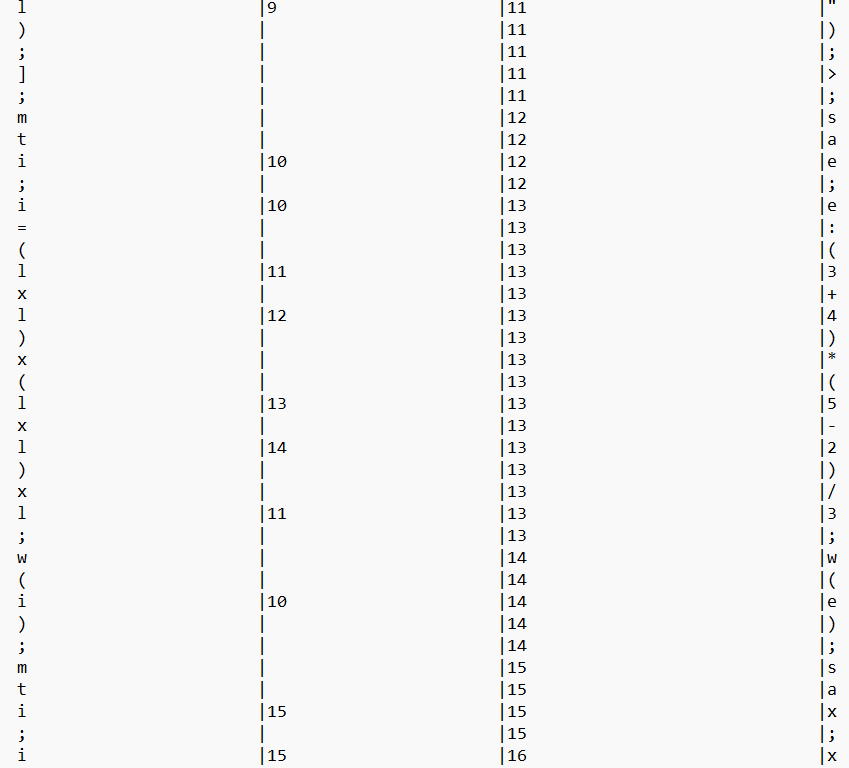
****

Рисунок A.1 — Часть таблицы лексем

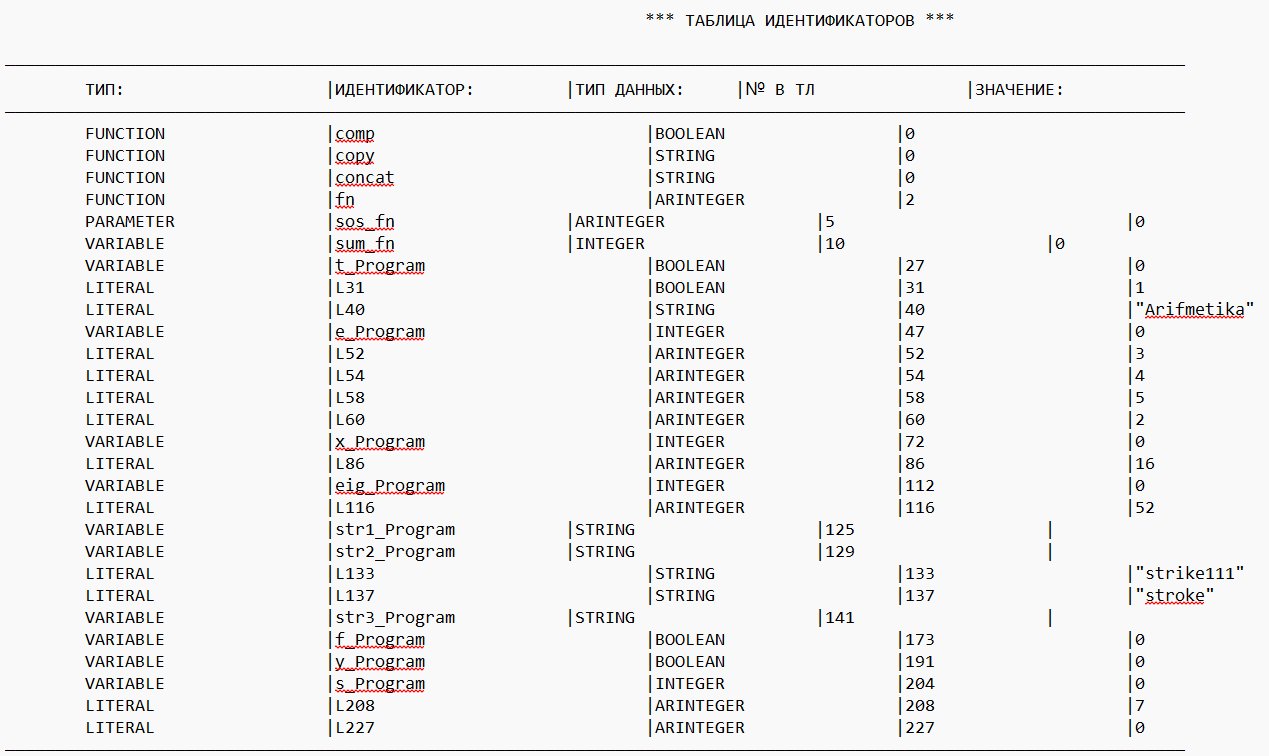


Рисунок A.2 — Таблица идентификаторов

# ПРИЛОЖЕНИЕ Б

|  |
| --- |
| Greibach greibach(NS('S'), TS('$'), 8,  Rule(NS('S'), GRB\_ERROR\_SERIES + 0, 3, // Неверная структура программы  Rule::Chain(5, TS('p'), TS('{'), NS('N'), TS('}'), TS(';')),  Rule::Chain(14, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('o'), NS('R'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('o'), NS('R'), TS(';'), TS('}'), TS(';'), NS('S'))  ),  Rule(NS('N'), GRB\_ERROR\_SERIES + 1, 12, // Ошибочный оператор  Rule::Chain(4, TS('m'), TS('t'), TS('i'), TS(';')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(5, TS('m'), TS('t'), TS('i'), TS(';'), NS('N')),  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(9, TS('m'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';'), NS('N')),  Rule::Chain(5, TS('w'), TS('('), NS('R'), TS(')'), TS(';')),  Rule::Chain(6, TS('w'), TS('('), NS('R'), TS(')'), TS(';'), NS('N')),  Rule::Chain(9, TS('c'), TS('('), NS('I'), TS(')'), TS('['), NS('N'), TS(']'), TS(';'), NS('N')),  Rule::Chain(8, TS('c'), TS('('), NS('I'), TS(')'), TS('['), NS('N'), TS(']'), TS(';')),  Rule::Chain(9, TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';'), NS('N'), TS('}'), TS(';')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('N')),  Rule::Chain(3, TS('o'), NS('R'), TS(';'))  ),  Rule(NS('E'), GRB\_ERROR\_SERIES + 2, 10, // Ошибка в выражении  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('('), NS('E'), TS(')')),  Rule::Chain(3, TS('i'), TS('('), TS(')')),  Rule::Chain(4, TS('i'), TS('('), NS('W'), 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(4, TS('i'), TS('('), TS(')'), NS('M')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M'))  ),  Rule(NS('M'), GRB\_ERROR\_SERIES + 3, 2,  Rule::Chain(2, TS('x'), NS('E')),  Rule::Chain(3, TS('x'), NS('E'), NS('M'))  ),  Rule(NS('F'), GRB\_ERROR\_SERIES + 4, 2, // Ошибка в параметрах  Rule::Chain(2, TS('t'), TS('i')),  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F'))  ),  Rule(NS('W'), GRB\_ERROR\_SERIES + 5, 4, // Ошибка в пар-ах вызыв. ф-ции  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('R'), GRB\_ERROR\_SERIES + 6, 2, // Только переменная или литерал  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l'))  ),  Rule(NS('I'), GRB\_ERROR\_SERIES + 7, 6, // Ошибка в условном операторе  Rule::Chain(1, TS('l')),  Rule::Chain(1, TS('i')),  Rule::Chain(3, TS('i'), TS('x'), TS('i')),  Rule::Chain(3, TS('i'), TS('x'), TS('l')),  Rule::Chain(3, TS('l'), TS('x'), TS('i')),  Rule::Chain(3, TS('l'), TS('x'), TS('l'))  )  ); |

Листинг Б.1 — Правила, описывающие грамматику языка

# ПРИЛОЖЕНИЕ В

|  |
| --- |
| struct MfstState { // состояние автомата (для сохранения)  short lenta\_position; // позиция на ленте  short nrule; // номер текущего правила  short nrulechain; // номер текущей цепочки, текущего правила  MFSTSTACK st; // стек автомата  MfstState();  MfstState(  short pposition, // позиция на ленте  MFSTSTACK pst, // стек автомата  short pnrulechain // номер текущей цепочки, текущего правила  );  MfstState(  short pposition, // позиция на ленте  MFSTSTACK pst, // стек автомата  short pnrule, // номер текущего правила  short pnrulechain // номер текущей цепочки, текущего правила  );  };  struct Mfst { // магазинный автомат  enum RC\_STEP { // код возврата функции step  NS\_OK, // нашли правило и цепочку, цепочка -> в стек  NS\_NORULE, // не найдено правило (ошибка в грамматике)  NS\_NORULECHAIN, // не найдена цепочка (ошибка в исходном коде)  NS\_ERROR, // неизвестный нетерм. символ грамматики  TS\_OK, // тек. символ = вершине стека, продв. лента  TS\_NOK, // тек. символ != вершине стека, восст. 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(  short plenta\_position, // позиция на ленте  RC\_STEP prt\_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  short lenta\_position; // текущая позиция на ленте  short nrule; // номер текущего правила  short nrulechain; // номер тек. цепочки, тек. правила  short lenta\_size; // размер ленты  GRB::Greibach grebach; // грамматика Грейбах  LA::Tables lex; // результат работы лекс. анализатора  MFSTSTACK st; // стек автомата  std::stack<MfstState> storestate; // стек для сохранения состояний  Mfst();  Mfst(  LA::Tables pt, // результат работы лекс. 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-ю строку диагностики или 0х00  bool savestate(); // сохранить состояние автомата  bool restate(); // восстановить состояние автомата  bool push\_chain( // поместить цепочку правила в стек  GRB::Rule::Chain chain // цепочка правила  );  RC\_STEP step(); // выполнить шаг автомата  bool start(); // запустить автомат  bool savediagnosis(  RC\_STEP pprc\_step // код завершения шага  );  void printrules(); // вывести последовательность правил  struct Deducation { // вывод  short size; // количество шагов в выводе  short\* nrules; // номера правил грамматики  short\* nrulechains; // номера цепочек правил грамматики  Deducation() { size = 0; nrules = 0; nrulechains = 0; };  } deducation;  bool savededucation(); // сохранить дерево вывода  }; |

Листинг В.1 — Структура магазинного конечного автомата

# ПРИЛОЖЕНИЕ Г

|  |
| --- |
| 0 : S->tfi(F){NoR;};S tfi(ti){mti;i=ixi;oi;};p{ S$  0 : SAVESTATE: 1  0 : tfi(ti){mti;i=ixi;oi;};p{ tfi(F){NoR;};S$  1 : fi(ti){mti;i=ixi;oi;};p{m fi(F){NoR;};S$  2 : i(ti){mti;i=ixi;oi;};p{mt i(F){NoR;};S$  3 : (ti){mti;i=ixi;oi;};p{mti (F){NoR;};S$  4 : ti){mti;i=ixi;oi;};p{mti; F){NoR;};S$  5 : F->ti ti){mti;i=ixi;oi;};p{mti; F){NoR;};S$  5 : SAVESTATE: 2  5 : ti){mti;i=ixi;oi;};p{mti; ti){NoR;};S$  6 : i){mti;i=ixi;oi;};p{mti;i i){NoR;};S$  7 : ){mti;i=ixi;oi;};p{mti;i= ){NoR;};S$  8 : {mti;i=ixi;oi;};p{mti;i=l {NoR;};S$  9 : mti;i=ixi;oi;};p{mti;i=l; NoR;};S$  10 : N->mti; mti;i=ixi;oi;};p{mti;i=l; NoR;};S$  10 : SAVESTATE: 3  10 : mti;i=ixi;oi;};p{mti;i=l; mti;oR;};S$  11 : ti;i=ixi;oi;};p{mti;i=l;c ti;oR;};S$  12 : i;i=ixi;oi;};p{mti;i=l;c( i;oR;};S$  13 : ;i=ixi;oi;};p{mti;i=l;c(i ;oR;};S$  14 : i=ixi;oi;};p{mti;i=l;c(i) oR;};S$  15 : TS\_NOK/NS\_NORULECHAIN  15 : RESTATE  15 : mti;i=ixi;oi;};p{mti;i=l; NoR;};S$  16 : N->mti;N mti;i=ixi;oi;};p{mti;i=l; NoR;};S$  16 : SAVESTATE: 3  16 : mti;i=ixi;oi;};p{mti;i=l; mti;NoR;};S$  17 : ti;i=ixi;oi;};p{mti;i=l;c ti;NoR;};S$  18 : i;i=ixi;oi;};p{mti;i=l;c( i;NoR;};S$  19 : ;i=ixi;oi;};p{mti;i=l;c(i ;NoR;};S$  20 : i=ixi;oi;};p{mti;i=l;c(i) NoR;};S$  21 : N->i=E; i=ixi;oi;};p{mti;i=l;c(i) NoR;};S$  21 : SAVESTATE: 4  21 : i=ixi;oi;};p{mti;i=l;c(i) i=E;oR;};S$  22 : =ixi;oi;};p{mti;i=l;c(i)[ =E;oR;};S$  23 : ixi;oi;};p{mti;i=l;c(i)[w E;oR;};S$  24 : E->i ixi;oi;};p{mti;i=l;c(i)[w E;oR;};S$  24 : SAVESTATE: 5  24 : ixi;oi;};p{mti;i=l;c(i)[w i;oR;};S$  25 : xi;oi;};p{mti;i=l;c(i)[w( ;oR;};S$  26 : TS\_NOK/NS\_NORULECHAIN  26 : RESTATE  26 : ixi;oi;};p{mti;i=l;c(i)[w E;oR;};S$  27 : E->i() ixi;oi;};p{mti;i=l;c(i)[w E;oR;};S$  27 : SAVESTATE: 5  27 : ixi;oi;};p{mti;i=l;c(i)[w i();oR;};S$  28 : xi;oi;};p{mti;i=l;c(i)[w( ();oR;};S$  29 : TS\_NOK/NS\_NORULECHAIN  29 : RESTATE  29 : ixi;oi;};p{mti;i=l;c(i)[w E;oR;};S$  30 : E->i(W) ixi;oi;};p{mti;i=l;c(i)[w E;oR;};S$  30 : SAVESTATE: 5  30 : ixi;oi;};p{mti;i=l;c(i)[w i(W);oR;};S$  31 : xi;oi;};p{mti;i=l;c(i)[w( (W);oR;};S$  32 : TS\_NOK/NS\_NORULECHAIN  32 : RESTATE  32 : ixi;oi;};p{mti;i=l;c(i)[w E;oR;};S$  33 : E->iM ixi;oi;};p{mti;i=l;c(i)[w E;oR;};S$  33 : SAVESTATE: 5  33 : ixi;oi;};p{mti;i=l;c(i)[w iM;oR;};S$  34 : xi;oi;};p{mti;i=l;c(i)[w( M;oR;};S$  35 : M->xE xi;oi;};p{mti;i=l;c(i)[w( M;oR;};S$  35 : SAVESTATE: 6  35 : xi;oi;};p{mti;i=l;c(i)[w( xE;oR;};S$  36 : i;oi;};p{mti;i=l;c(i)[w(l E;oR;};S$  37 : E->i i;oi;};p{mti;i=l;c(i)[w(l E;oR;};S$  37 : SAVESTATE: 7  37 : i;oi;};p{mti;i=l;c(i)[w(l i;oR;};S$  38 : ;oi;};p{mti;i=l;c(i)[w(l) ;oR;};S$  39 : oi;};p{mti;i=l;c(i)[w(l); oR;};S$  40 : i;};p{mti;i=l;c(i)[w(l);] R;};S$  41 : R->i i;};p{mti;i=l;c(i)[w(l);] R;};S$  41 : SAVESTATE: 8  41 : i;};p{mti;i=l;c(i)[w(l);] i;};S$  42 : ;};p{mti;i=l;c(i)[w(l);]; ;};S$  43 : };p{mti;i=l;c(i)[w(l);];m };S$  44 : ;p{mti;i=l;c(i)[w(l);];mt ;S$  45 : p{mti;i=l;c(i)[w(l);];mti S$  46 : S->p{N}; p{mti;i=l;c(i)[w(l);];mti S$  46 : SAVESTATE: 9  46 : p{mti;i=l;c(i)[w(l);];mti p{N};$  47 : {mti;i=l;c(i)[w(l);];mti; {N};$  48 : mti;i=l;c(i)[w(l);];mti;i N};$  49 : N->mti; mti;i=l;c(i)[w(l);];mti;i N};$  49 : SAVESTATE: 10  49 : mti;i=l;c(i)[w(l);];mti;i mti;};$  50 : ti;i=l;c(i)[w(l);];mti;i= ti;};$  51 : i;i=l;c(i)[w(l);];mti;i=( i;};$  52 : ;i=l;c(i)[w(l);];mti;i=(l ;};$  53 : i=l;c(i)[w(l);];mti;i=(lx };$  54 : TS\_NOK/NS\_NORULECHAIN  54 : RESTATE  54 : mti;i=l;c(i)[w(l);];mti;i N};$  55 : N->mti;N mti;i=l;c(i)[w(l);];mti;i N};$  55 : SAVESTATE: 10  55 : mti;i=l;c(i)[w(l);];mti;i mti;N};$  56 : ti;i=l;c(i)[w(l);];mti;i= ti;N};$  57 : i;i=l;c(i)[w(l);];mti;i=( i;N};$  58 : ;i=l;c(i)[w(l);];mti;i=(l ;N};$  59 : i=l;c(i)[w(l);];mti;i=(lx N};$  60 : N->i=E; i=l;c(i)[w(l);];mti;i=(lx N};$  60 : SAVESTATE: 11  60 : i=l;c(i)[w(l);];mti;i=(lx i=E;};$  61 : =l;c(i)[w(l);];mti;i=(lxl =E;};$  62 : l;c(i)[w(l);];mti;i=(lxl) E;};$  63 : E->l l;c(i)[w(l);];mti;i=(lxl) E;};$  63 : SAVESTATE: 12  63 : l;c(i)[w(l);];mti;i=(lxl) l;};$  64 : ;c(i)[w(l);];mti;i=(lxl)x ;};$  65 : c(i)[w(l);];mti;i=(lxl)x( };$  66 : TS\_NOK/NS\_NORULECHAIN  66 : RESTATE  66 : l;c(i)[w(l);];mti;i=(lxl) E;};$  67 : E->lM l;c(i)[w(l);];mti;i=(lxl) E;};$  67 : SAVESTATE: 12  67 : l;c(i)[w(l);];mti;i=(lxl) lM;};$  68 : ;c(i)[w(l);];mti;i=(lxl)x M;};$  69 : TNS\_NORULECHAIN/NS\_NORULE  69 : RESTATE  69 : l;c(i)[w(l);];mti;i=(lxl) E;};$  70 : TNS\_NORULECHAIN/NS\_NORULE  70 : RESTATE  70 : i=l;c(i)[w(l);];mti;i=(lx N};$  71 : N->i=E;N i=l;c(i)[w(l);];mti;i=(lx N};$  71 : SAVESTATE: 11  71 : i=l;c(i)[w(l);];mti;i=(lx i=E;N};$  72 : =l;c(i)[w(l);];mti;i=(lxl =E;N};$  73 : l;c(i)[w(l);];mti;i=(lxl) E;N};$  74 : E->l l;c(i)[w(l);];mti;i=(lxl) E;N};$  74 : SAVESTATE: 12  74 : l;c(i)[w(l);];mti;i=(lxl) l;N};$  75 : ;c(i)[w(l);];mti;i=(lxl)x ;N};$  76 : c(i)[w(l);];mti;i=(lxl)x( N};$  77 : N->c(I)[N];N c(i)[w(l);];mti;i=(lxl)x( N};$  77 : SAVESTATE: 13  77 : c(i)[w(l);];mti;i=(lxl)x( c(I)[N];N};$  78 : (i)[w(l);];mti;i=(lxl)x(l (I)[N];N};$  79 : i)[w(l);];mti;i=(lxl)x(lx I)[N];N};$  80 : I->i i)[w(l);];mti;i=(lxl)x(lx I)[N];N};$  80 : SAVESTATE: 14  80 : i)[w(l);];mti;i=(lxl)x(lx i)[N];N};$  81 : )[w(l);];mti;i=(lxl)x(lxl )[N];N};$  82 : [w(l);];mti;i=(lxl)x(lxl) [N];N};$  83 : w(l);];mti;i=(lxl)x(lxl)x N];N};$  84 : N->w(R); w(l);];mti;i=(lxl)x(lxl)x N];N};$  84 : SAVESTATE: 15  84 : w(l);];mti;i=(lxl)x(lxl)x w(R);];N};$  85 : (l);];mti;i=(lxl)x(lxl)xl (R);];N};$  86 : l);];mti;i=(lxl)x(lxl)xl; R);];N};$  87 : R->l l);];mti;i=(lxl)x(lxl)xl; R);];N};$  87 : SAVESTATE: 16  87 : l);];mti;i=(lxl)x(lxl)xl; l);];N};$  88 : );];mti;i=(lxl)x(lxl)xl;w );];N};$  89 : ;];mti;i=(lxl)x(lxl)xl;w( ;];N};$  90 : ];mti;i=(lxl)x(lxl)xl;w(i ];N};$  91 : ;mti;i=(lxl)x(lxl)xl;w(i) ;N};$  92 : mti;i=(lxl)x(lxl)xl;w(i); N};$  93 : N->mti; mti;i=(lxl)x(lxl)xl;w(i); N};$  93 : SAVESTATE: 17  93 : mti;i=(lxl)x(lxl)xl;w(i); mti;};$  94 : ti;i=(lxl)x(lxl)xl;w(i);m ti;};$  95 : i;i=(lxl)x(lxl)xl;w(i);mt i;};$  96 : ;i=(lxl)x(lxl)xl;w(i);mti ;};$  97 : i=(lxl)x(lxl)xl;w(i);mti; };$  98 : TS\_NOK/NS\_NORULECHAIN  98 : RESTATE  98 : mti;i=(lxl)x(lxl)xl;w(i); N};$  99 : N->mti;N mti;i=(lxl)x(lxl)xl;w(i); N};$  99 : SAVESTATE: 17  99 : mti;i=(lxl)x(lxl)xl;w(i); mti;N};$  100 : ti;i=(lxl)x(lxl)xl;w(i);m ti;N};$  101 : i;i=(lxl)x(lxl)xl;w(i);mt i;N};$  102 : ;i=(lxl)x(lxl)xl;w(i);mti ;N};$  103 : i=(lxl)x(lxl)xl;w(i);mti; N};$  104 : N->i=E; i=(lxl)x(lxl)xl;w(i);mti; N};$  104 : SAVESTATE: 18  104 : i=(lxl)x(lxl)xl;w(i);mti; i=E;};$  105 : =(lxl)x(lxl)xl;w(i);mti;i =E;};$  106 : (lxl)x(lxl)xl;w(i);mti;i= E;};$  107 : E->(E) (lxl)x(lxl)xl;w(i);mti;i= E;};$  107 : SAVESTATE: 19  107 : (lxl)x(lxl)xl;w(i);mti;i= (E);};$  108 : lxl)x(lxl)xl;w(i);mti;i=( E);};$  109 : E->l lxl)x(lxl)xl;w(i);mti;i=( E);};$  109 : SAVESTATE: 20  109 : lxl)x(lxl)xl;w(i);mti;i=( l);};$  110 : xl)x(lxl)xl;w(i);mti;i=(i );};$  111 : TS\_NOK/NS\_NORULECHAIN  111 : RESTATE  111 : lxl)x(lxl)xl;w(i);mti;i=( E);};$  112 : E->lM lxl)x(lxl)xl;w(i);mti;i=( E);};$  112 : SAVESTATE: 20  112 : lxl)x(lxl)xl;w(i);mti;i=( lM);};$  113 : xl)x(lxl)xl;w(i);mti;i=(i M);};$  114 : M->xE xl)x(lxl)xl;w(i);mti;i=(i M);};$  114 : SAVESTATE: 21  114 : xl)x(lxl)xl;w(i);mti;i=(i xE);};$  115 : l)x(lxl)xl;w(i);mti;i=(i( E);};$  116 : E->l l)x(lxl)xl;w(i);mti;i=(i( E);};$  116 : SAVESTATE: 22  116 : l)x(lxl)xl;w(i);mti;i=(i( l);};$  117 : )x(lxl)xl;w(i);mti;i=(i(i );};$  118 : x(lxl)xl;w(i);mti;i=(i(i) ;};$  119 : TS\_NOK/NS\_NORULECHAIN  119 : RESTATE  119 : l)x(lxl)xl;w(i);mti;i=(i( E);};$  120 : E->lM l)x(lxl)xl;w(i);mti;i=(i( E);};$  120 : SAVESTATE: 22  120 : l)x(lxl)xl;w(i);mti;i=(i( lM);};$  121 : )x(lxl)xl;w(i);mti;i=(i(i M);};$  122 : TNS\_NORULECHAIN/NS\_NORULE  122 : RESTATE  122 : l)x(lxl)xl;w(i);mti;i=(i( E);};$  123 : TNS\_NORULECHAIN/NS\_NORULE  123 : RESTATE  123 : xl)x(lxl)xl;w(i);mti;i=(i M);};$  124 : M->xEM xl)x(lxl)xl;w(i);mti;i=(i M);};$  124 : SAVESTATE: 21  124 : xl)x(lxl)xl;w(i);mti;i=(i xEM);};$  125 : l)x(lxl)xl;w(i);mti;i=(i( EM);};$  126 : E->l l)x(lxl)xl;w(i);mti;i=(i( EM);};$  126 : SAVESTATE: 22  126 : l)x(lxl)xl;w(i);mti;i=(i( lM);};$  127 : )x(lxl)xl;w(i);mti;i=(i(i M);};$  128 : TNS\_NORULECHAIN/NS\_NORULE  128 : RESTATE  128 : l)x(lxl)xl;w(i);mti;i=(i( EM);};$  129 : E->lM l)x(lxl)xl;w(i);mti;i=(i( EM);};$  129 : SAVESTATE: 22  129 : l)x(lxl)xl;w(i);mti;i=(i( lMM);};$  130 : )x(lxl)xl;w(i);mti;i=(i(i MM);};$  131 : TNS\_NORULECHAIN/NS\_NORULE  131 : RESTATE  131 : l)x(lxl)xl;w(i);mti;i=(i( EM);};$  132 : TNS\_NORULECHAIN/NS\_NORULE  132 : RESTATE  132 : xl)x(lxl)xl;w(i);mti;i=(i M);};$ |

Листинг Г.1 — Работа синтаксического анализатора (фрагмент)

|  |
| --- |
| 0 : S->tfi(F){NoR;};S  4 : F->ti  8 : N->mti;N  12 : N->i=E;  14 : E->iM  15 : M->xE  16 : E->i  19 : R->i  23 : S->p{N};  25 : N->mti;N  29 : N->i=E;N  31 : E->l  33 : N->c(I)[N];N  35 : I->i  38 : N->w(R);  40 : R->l  45 : N->mti;N  49 : N->i=E;N  51 : E->(E)M  52 : E->lM  53 : M->xE  54 : E->l  56 : M->xE  57 : E->(E)M  58 : E->lM  59 : M->xE  60 : E->l  62 : M->xE  63 : E->l  65 : N->w(R);N  67 : R->i  70 : N->mti;N  74 : N->i=E;N  76 : E->(E)M  77 : E->i(W)M  79 : W->i  81 : M->xE  82 : E->l  84 : M->xE  85 : E->(E)  86 : E->lM  87 : M->xE  88 : E->l  91 : N->w(R);N  93 : R->i  96 : N->i=E;N  98 : E->(E)M  99 : E->lM  100 : M->xE  101 : E->l  103 : M->xE  104 : E->(E)  105 : E->lM  106 : M->xE  107 : E->l  110 : N->mti;N  114 : N->i=E;N  116 : E->l  118 : N->w(R);N  120 : R->i  123 : N->mti;N  127 : N->mti;N  131 : N->i=E;N  133 : E->l  135 : N->i=E;N  137 : E->l  139 : N->mti;N  143 : N->i=E;N  145 : E->i(W)  147 : W->i,W  149 : W->i  152 : N->w(R);N  154 : R->i  157 : N->i=E;N  159 : E->i(W)  161 : W->i,W  163 : W->i  166 : N->w(R);N  168 : R->i  171 : N->mti;N  175 : N->i=E;N  177 : E->i(W)  179 : W->i,W  181 : W->i  184 : N->w(R);N  186 : R->i  189 : N->mti;N  193 : N->i=E;N  195 : E->l  197 : N->c(I)[N];N  199 : I->i |

Листинг Г.2 — Результат работы синтаксического анализатора (фрагмент)

# ПРИЛОЖЕНИЕ Д

|  |
| --- |
| bool PN::PolishNotation(int n, LT::LexTable& lextable, IT::IdTable& idtable) {  std::queue<LT::Entry> current;  std::stack<LT::Entry> stack;  int i = n;  int priority;  int parCount = 0;  int indID = -1;  while (lextable.table[i].lexema != ';') {  priority = getP(lextable.table[i]);  if (lextable.table[i].lexema == 'i' &&  lextable.table[i + 1].lexema == '(') { // попали на вызов ф-ии  indID = lextable.table[i].indID;  i++;  priority = getP(lextable.table[i]);  while (priority < 2) {  if (priority == 0) {  current.push(lextable.table[i]);  parCount++;  }  else if (priority == 1)  stack.push(lextable.table[i]);  else if (priority == -2);  else if (priority == -1) {  while (getP(stack.top()) != 1) {  // выталкиваем стек пока не встретим открывашку  current.push(stack.top());  stack.pop();  }  stack.pop();  current.push({ '@', lextable.table[i].sn, lextable.table[i].idxLT, indID });  i++;  priority = getP(lextable.table[i]);  break;  }  i++;  priority = getP(lextable.table[i]);  }  }  if (priority == 0)  current.push(lextable.table[i]);  else if (priority == 1)  stack.push(lextable.table[i]);  else if (priority == 2 || priority == 3 ||  priority == 4 || priority == 5 || priority == 6) {  while (!stack.empty()) {  if (getP(stack.top()) >= priority) {  current.push(stack.top());  stack.pop();  }  else  break;  }  stack.push(lextable.table[i]);  }  else if (priority == -1) {  while (getP(stack.top()) != 1) {  // выталкиваем стек пока не встретим открывашку  current.push(stack.top());  stack.pop();  }  stack.pop();  }  else if (priority == -3)  break;  i++;  }  while (!stack.empty()) {  current.push(stack.top());  stack.pop();  }  current.push(lextable.table[i]);  //обратная польская запись построена  int posLast = i; // поз. последнего символа выр-я до польской записи  i = n;  for (i; i <= posLast; i++) {  lextable.table[i] = { '#' , -1 , i , -1 };  while (!current.empty()) {  lextable.table[i] = current.front();  lextable.table[i].idxLT = i;  current.pop();  i++;  }  }  return true;  }  int PN::getP(LT::Entry table) {  char token = table.lexema;  if (token == 'x')  token = table.operatorValue;  if (token == '~')  return 6;  else if (token == '\*' || token == '/')  return 5;  else if (token == '+' || token == '-')  return 4;  else if (token == '&')  return 3;  else if (token == '|')  return 2;  else if (token == '(')  return 1;  else if (token == ')')  return -1;  else if (token == ',')  return -2;  else if (token == ';')  return -3;  else return 0;  } |

Листинг Д.1 — Алгоритм преобразования выражений к польской записи

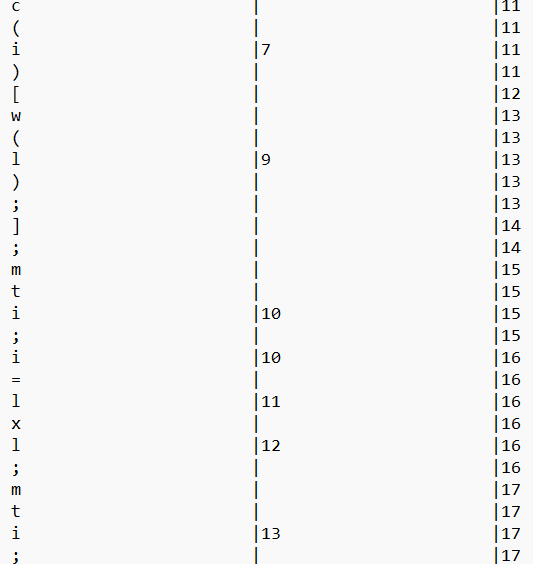
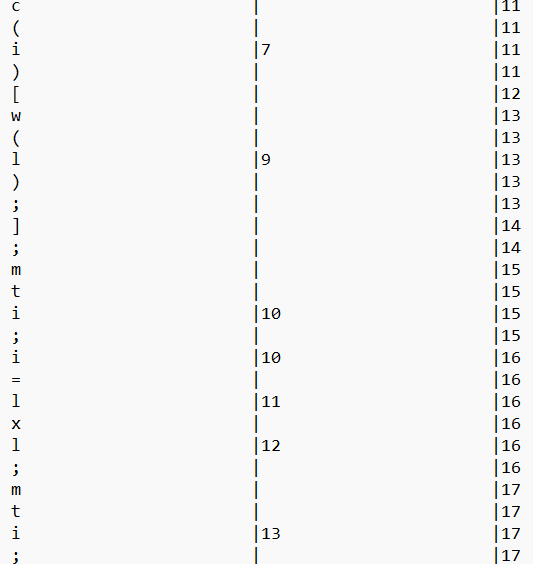
 

Рисунок Д.1 — Соответствие таблицы лексем и её аналога в польской нотации

# ПРИЛОЖЕНИЕ Е

|  |
| --- |
| .586P  .MODEL FLAT, STDCALL  includelib kernel32.lib  includelib libucrt.lib  includelib "..\Debug\Library.lib  EXTRN comp: PROC  EXTRN concat: PROC  EXTRN copy: PROC  EXTRN ConsNum: PROC  EXTRN ConsStr: PROC  EXTRN ConsBool: PROC  EXTRN consPause: proc  ExitProcess PROTO : DWORD  .STACK 4096  .CONST  L1 dd 1  L2 db "Arifmetika" , 0  L3 dd 3  L4 dd 4  L5 dd 5  L6 dd 2  L7 dd 16  L8 dd 52  L9 db "strike111" , 0  L10 db "stroke" , 0  L11 dd 7  L12 dd 0  .DATA  buffer BYTE 256 dup(0)  sum\_fn DD 0  t\_Program DD 0  e\_Program DD 0  x\_Program DD 0  eig\_Program DD 0  str1\_Program DD ?  str2\_Program DD ?  str3\_Program DD ?  f\_Program DD 0  y\_Program DD 0  s\_Program DD 0  .CODE  fn PROC sos\_fn : DWORD  push sos\_fn  push sos\_fn  pop EAX  pop EBX  add EAX, EBX  push EAX  pop sum\_fn  mov eax, sum\_fn  ret  fn ENDP  Program PROC  START:  push L1  pop t\_Program  mov EAX,t\_Program  mov EBX, 1  sub EAX, EBX  je true1  jmp exit1  true1:  mov EAX, offset L2  push EAX  call ConsStr  exit1:  push L3  push L4  pop EAX  pop EBX  add EAX, EBX  push EAX  push L5  push L6  pop EBX  pop EAX  sub EAX, EBX  push EAX  pop EAX  pop EBX  mul EBX  push EAX  push L3  pop EBX  mov EDX, 0  pop EAX  div EBX  push EAX  pop e\_Program  mov EAX, e\_Program  push EAX  call ConsNum  push e\_Program  call fn  push EAX  push L4  pop EAX  pop EBX  add EAX, EBX  push EAX  push L7  push L7  pop EAX  pop EBX  NOT EBX  add EAX, EBX  push EAX  pop EBX  pop EAX  sub EAX, EBX  push EAX  pop x\_Program  mov EAX, x\_Program  push EAX  call ConsNum  push L3  push L4  pop EAX  pop EBX  OR EAX, EBX  push EAX  push L6  push L3  pop EAX  pop EBX  AND EAX, EBX  push EAX  pop EAX  pop EBX  NOT EBX  add EAX, EBX  push EAX  pop x\_Program  push L8  pop eig\_Program  mov EAX, eig\_Program  push EAX  call ConsNum  push offset L9  pop str1\_Program  push offset L10  pop str2\_Program  push str1\_Program  push str2\_Program  push offset buffer  call concat  push EAX  pop str3\_Program  mov EAX, str3\_Program  push EAX  call ConsStr  push str1\_Program  push str2\_Program  push offset buffer  call copy  push EAX  pop str3\_Program  mov EAX, str3\_Program  push EAX  call ConsStr  push str1\_Program  push str2\_Program  push offset buffer  call comp  push EAX  pop f\_Program  mov EAX, f\_Program  push EAX  call ConsBool  push L1  pop y\_Program  mov EAX,y\_Program  mov EBX, 1  sub EAX, EBX  je true2  jmp exit2  true2:  push L11  pop s\_Program  mov EAX,s\_Program  mov EBX, L11  sub EAX, EBX  je true3  jmp exit3  true3:  mov EAX, s\_Program  push EAX  call ConsNum  exit3:  exit2:  mov eax, L12  push 0  call consPause  call ExitProcess  Program ENDP  end Program |

Листинг Е.1 — Результат генерации ассемблерного кода