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 UVV-2024»

Выполнил студент Угоренко Виктория Владимировна

(Ф.И.О.)

Руководитель проекта преп.-стаж. Волчек Д.И.

(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

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

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

Нормоконтролер

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

Курсовой проект защищен с оценкой

Минск 2024

Содержание

[Введение 7](#_Toc185018346)

[1 Спецификация языка программирования 8](#_Toc185018347)

[1.1 Характеристика языка программирования 8](#_Toc185018348)

[1.2 Определение алфавита языка программирования 8](#_Toc185018349)

[1.3 Применяемые сепараторы 8](#_Toc185018350)

[1.4 Применяемые кодировки 9](#_Toc185018351)

[1.5 Типы данных 9](#_Toc185018352)

[1.6 Преобразование типов данных 10](#_Toc185018353)

[1.7 Идентификаторы 11](#_Toc185018354)

[1.8 Литералы 11](#_Toc185018355)

[1.9 Объявление данных 11](#_Toc185018356)

[1.10 Инициализация данных 12](#_Toc185018357)

[1.11 Инструкции языка 12](#_Toc185018358)

[1.12 Операции языка 13](#_Toc185018359)

[1.13 Выражения и их вычисление 13](#_Toc185018360)

[1.14 Конструкции языка 14](#_Toc185018361)

[1.15 Области видимости идентификаторов 14](#_Toc185018362)

[1.16 Семантические проверки 15](#_Toc185018363)

[1.17 Распределение оперативной памяти на этапе выполнения 15](#_Toc185018364)

[1.18 Стандартная библиотека и её состав 15](#_Toc185018365)

[1.19 Ввод и вывод данных 16](#_Toc185018366)

[1.20 Точка входа 17](#_Toc185018367)

[1.21 Препроцессор 17](#_Toc185018368)

[1.22 Соглашения о вызовах 17](#_Toc185018369)

[1.23 Объектный код 17](#_Toc185018370)

[1.24 Классификация сообщений транслятора 17](#_Toc185018371)

[1.25 Контрольный пример 18](#_Toc185018372)

[2 Структура транслятора 19](#_Toc185018373)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 19](#_Toc185018374)

[2.2 Перечень параметров транслятора 20](#_Toc185018375)

[2.3 Протоколы, формируемые транслятором 21](#_Toc185018376)

[3 Разработка лексического анализатора 22](#_Toc185018377)

[3.1 Структура лексического анализатора 22](#_Toc185018378)

[3.2 Контроль входных символов 22](#_Toc185018379)

[3.3 Удаление избыточных символов 23](#_Toc185018380)

[3.4 Перечень ключевых слов 23](#_Toc185018381)

[3.5 Основные структуры данных 24](#_Toc185018382)

[3.6 Структура и перечень сообщений лексического анализатора 24](#_Toc185018383)

[3.7 Принцип обработки ошибок 25](#_Toc185018384)

[3.8 Параметры лексического анализатора и режимы его работы 25](#_Toc185018385)

[3.9 Алгоритм лексического анализа 25](#_Toc185018386)

[3.10 Контрольный пример 26](#_Toc185018387)

[4. Разработка синтаксического анализатора 27](#_Toc185018388)

[4.1 Структура синтаксического анализатора 27](#_Toc185018389)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 27](#_Toc185018390)

[4.3 Построение конечного магазинного автомата 28](#_Toc185018391)

[4.4 Основные структуры данных 29](#_Toc185018392)

[4.5 Описание алгоритма синтаксического разбора 29](#_Toc185018393)

[4.6 Структура и перечень сообщений синтаксического анализатора 29](#_Toc185018394)

[4.7 Параметры синтаксического анализатора и режимы его работы 29](#_Toc185018395)

[4.8. Принцип обработки ошибок 30](#_Toc185018396)

[4.9 Контрольный пример 30](#_Toc185018397)

[5 Разработка семантического анализатора 31](#_Toc185018398)

[5.1 Структура семантического анализатора 31](#_Toc185018399)

[5.2 Функции семантического анализатора 31](#_Toc185018400)

[5.3 Структура и перечень сообщений семантического анализатора 31](#_Toc185018401)

[5.4 Принцип обработки ошибок 32](#_Toc185018402)

[5.5 Контрольный пример 32](#_Toc185018403)

[6. Преобразование выражений 33](#_Toc185018404)

[6.1 Выражения, допускаемые языком 33](#_Toc185018405)

[6.2 Польская запись и принцип её построения 33](#_Toc185018406)

[6.3 Программная реализация обработки выражений 34](#_Toc185018407)

[6.4 Контрольный пример 34](#_Toc185018408)

[7. Генерация кода 35](#_Toc185018409)

[7.1 Структура генератора кода 35](#_Toc185018410)

[7.2 Представление типов данных в оперативной памяти 35](#_Toc185018411)

[7.3 Статическая библиотека 36](#_Toc185018412)

[7.4 Особенности алгоритма генерации кода 36](#_Toc185018413)

[7.5 Параметры, управляющие генерацией кода 36](#_Toc185018414)

[7.6 Контрольный пример 36](#_Toc185018415)

[8. Тестирование транслятора 38](#_Toc185018416)

[8.1 Общие положения 38](#_Toc185018417)

[8.2 Результаты тестирования 38](#_Toc185018418)

[Заключение 39](#_Toc185018419)

[Список использованных источников 40](#_Toc185018420)

[Приложение А 41](#_Toc185018421)

[Приложение Б 43](#_Toc185018422)

[Приложение В 45](#_Toc185018423)

[Приложение Г 54](#_Toc185018424)

[Приложение Д 58](#_Toc185018425)

[Приложение Е 61](#_Toc185018426)

[Приложение Ж 62](#_Toc185018427)

[Приложение И 63](#_Toc185018428)

[Приложение К 70](#_Toc185018429)

# **Введение**

Задачей данного курсового проекта была поставлена разработка транслятора своего языка программирования UVV-2024. Он предназначен для выполнения арифметических и логических действий над числами.

Главная идея транслятора заключается в том, чтобы сделать исходный код на данном языке программирования понятным компьютеру. Для решения этой задачи был выбран способ трансляции исходного в исходный код на языке ассемблера. Язык ассемблера – это машинно-ориентированный язык, представляющий формат записи машинных команд, которые понятны для восприятия человеком.

Исходя из цели курсового проекта, были определены следующие задачи:

* разработка спецификации языка программирования;
* разработка структуры транслятора;
* разработка лексического анализатора;
* разработка синтаксического анализатора;
* разработка семантического анализатора;
* обработка выражений с помощью польской инверсии;
* генерация кода на язык ассемблера;
* тестирование транслятора.

Способы решения каждой задачи будут описаны в соответствующих главах курсового проекта.

В первой главе работы определена спецификация языка программирования.

Во второй главе представлена структура транслятора. В ней перечислены компоненты транслятора, их назначения и принципы взаимодействия.

В третьей главе описана разработка лексического анализатора, который создаёт таблицы лексем и идентификаторов.

В четвертой главе описана разработка синтаксического анализатора, который выполняет разбор исходного кода в соответствии с правилами языка программирования.

В пятой главе описан семантический анализатор, которые проверяет исходный код программы на наличие семантических ошибок.

В шестой главе описан способ преобразования выражений в польский формат.

В седьмой главе представлена генерация кода в язык ассемблера с помощью таблиц лексем и идентификаторов.

В восьмой главе описывается тестирование транслятора.

**1 Спецификация языка программирования**

**1.1 Характеристика языка программирования**

Язык программирования UVV-2024 является процедурным, строго типизированным, не объектно-ориентированным, компилируемым.

**1.2 Определение алфавита языка программирования**

Алфавит языка программирования – набор символов, которые могут использоваться при написании исходного кода.

Буква –> {a | b | … | z | A | B | … | Z}

Цифра –> {0 | 1 | … | 9}

Специальный символ -> {[ | ] | ( | ) | { | } | , | ” | ; | $ | ? | пробел | < | > | ~ | # | = | + | – | \* | / | %}

Символ языка –> {Буква | Цифра | Специальный символ}

**1.3 Применяемые сепараторы**

Символы-сепараторы служат в качестве разделителей цепочек языка во время обработки исходного текста программы с целью разделения на токены. Они представлены в таблице 1.1.

Таблица 1.1 – Символы-сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| **=** | Оператор присваивания |
| ‘пробел’ | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |
| [ … ] | Блок условной конструкции/цикла |
| ( … ) | Блок фактических или формальных параметров функции, а также приоритет арифметических операций |
| { … } | Блок определения функций и процедур |
| , | Разделитель параметров функций |
| + - \*/% | Арифметические операции |
| > < & ! <= >= | Логические операции (операции сравнения: больше, меньше, проверка на равенство, на неравенство), используемые в условии цикла/условной конструкции. |
| ; | Разделитель программных конструкций |

**1.4 Применяемые кодировки**

Для написания программ язык UVV-2024 использует кодировку Windows-1251, представленную на рисунке 1.1.

Изображение выглядит как текст, снимок экрана, число, Прямоугольник

Автоматически созданное описание

Рисунок 1.1 – Алфавит входных символов

**1.5 Типы данных**

В языке UVV-2024 реализованы четыре типа данных: целочисленный, символьный, строковый и булевый. Описание типов приведено в таблице 1.2.

Таблица 1.2 – Типы данных языка UVV-2024

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Целочисленный тип данных **int** | Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 2 байта. Максимальное значение: 32767.  Минимальное значение: -32768.  Инициализация по умолчанию: значение 0.  Поддерживаемые операции:  **+** (бинарный) – оператор сложения;  - (бинарный) – оператор вычитания;  **\*** (бинарный) – оператор умножения; |

Окончание таблицы 1.2

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Целочисленный тип данных **int** | **/** (бинарный) – оператор деления;  **%** (бинарный) – оператор ;  **=** (бинарный) – оператор присваивания.  В качестве условия цикла/условного оператора поддерживаются следующие логические операции:  **>** (бинарный) – оператор «больше»;  < (бинарный) – оператор «меньше»;  >= (бинарный) – оператор «больше либо равно»;  <= (бинарный) – оператор «меньше либо равно»;  ! (бинарный) – оператор проверки на неравенство.  & (бинарный) – оператор проверки на равенство. |
| Строковый тип данных **string** | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 255.  Инициализация по умолчанию: строка нулевой длины “”. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций. |
| Символьный тип данных **char** | Фундаментальный тип данных. Используется для работы с одним символом, Инициализация по умолчанию: символ нулевой “”.  Операции над данными символьного типа: присваивание символьному идентификатору значения другого символьного идентификатора, символьного литерала или значения символьной функции. |
| Булевый тип данных **bool** | Фундаментальный тип данных. Предусмотрен для объявления логической переменной, которая имеет одно из двух значений: true, false. Автоматическая инициализация значением false. |

**1.6 Преобразование типов данных**

Язык программирования UVV-2024 не обладает средствами для преобразования типов данных

**1.7 Идентификаторы**

Для именования функций, параметров и переменных используются идентификаторы. Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Идентификаторы не должны совпадать с ключевыми словами. Не предусмотрены зарезервированные идентификаторы. Имя идентификатора составляется по следующим образом:

* состоит из символов латинского алфавита нижнего регистра [a..z].
* максимальная длина идентификатора равна 10 и не должна превышать это значение.

**1.8 Литералы**

В языке существует 3 типа литералов: целого, символьного и строкового типов. С их помощью осуществляется инициализация переменных. Краткое описание литералов языка UVV-2024 представлено в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Литералы целого типа | Целочисленные литералы десятичного, двоичного и восьмеричного представления, автоматически инициализируются 0. |
| Символьные литералы | Символ, заключённые в “” (двойные кавычки), автоматически инициализируются NULL. |
| Строковые литералы | Символы, заключённые в “” (двойные кавычки), автоматически инициализируются пустой строкой, строковые переменные. Максимальное число символов в литерале . |

**1.9 Объявление данных**

Для объявления переменной используется ключевое слово **declare**, после которого указывается тип данных и имя идентификатора. Допускается инициализация при объявлении.

Пример объявления числового типа с инициализацией:

* declare int int1 = 12;
* declare int int2 = 0x1F;

Пример объявления переменной строкового типа с инициализацией:

* declare string str= “Hello”;

Пример объявления переменной символьного типа с инициализацией:

* declare char s= “S”;

Для объявления функций используется ключевое слово **function**, перед которым указывается тип функции (если функция возвращает значение), или ключевое слово **void**, если функция ничего не возвращает, а после – имя функции либо процедуры. Далее обязателен список параметров и тело функции. Точкой входа в программу является функция **main**, которая объявляется без указания типа возвращаемого значения и ключевого слова **function**.

Область видимости идентификаторов «сверху вниз» (по принципу С++). В языке UVV-2024 требуется обязательное объявление переменной перед её использованием. Все переменные должны находиться внутри программного блока языка. Имеется возможность объявления одинаковых переменных в разных блоках.

**1.10 Инициализация данных**

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. Объектами-инициализаторами могут быть только идентификаторы или литералы. При объявлении без инициализации предусмотрены значения по умолчанию: значение 0 для типа **int**, строка длины 0 (“”) для типа **string** и пустой символ (‘’) для типа **char**.

**1.11 Инструкции языка**

Инструкции языка UVV-2024 представлены в таблице 1.4.

Таблица 1.4 – ИнструкцииязыкаUVV-2024

|  |  |
| --- | --- |
| Инструкция | Реализация |
| Объявление переменной | declare<тип данных> <идентификатор>; |
| Объявление переменной с явной инициализацией | declare <тип данных> <идентификатор> =  <идентификатор/литерал>; |
| Возврат из функции или процедуры | Для функций, возвращающих значение:  return <идентификатор/литерал>;  Для процедур:  return; |
| Вывод данных | write <идентификатор/литерал>;  writeline <идентификатор/литерал>; |
| Вызов функции или процедуры | <идентификатор функции> (<список параметров>);  Список параметров может быть пустым. |
| Условный оператор | if <условное выражение>  […]  else[…] |
| Оператор цикла | while <условное выражение>  […] |

Окончание таблицы 1.4

|  |  |
| --- | --- |
| Присваивание | <идентификатор> = <выражение>;  Выражением может быть идентификатор, литерал, или вызов функции соответствующего типа. Для целочисленного типа выражение может быть дополнено арифметическими операциями с любым количеством операндов с использованием скобок. |

**1.12 Операции языка**

В языке UVV-2024 предусмотрены следующие операции с данными. Приоритетность операции умножения выше приоритета операций сложения и вычитания и операции инкремента и декремента. Для установки наивысшего приоритета используются круглые скобки. Условные операции представлены шестью операторами языка. Операции языка представлены в таблице 1.5

Таблица 1.5 – Операции языка UVV-2024

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические | 1. **+** –сложение 2. **-** – вычитание 3. \* – умножение 4. **/** – деление 5. = – присваивание |
| Строковые | 1. **=** – присваивание |
| Логические | 1. > – больше  2. < – меньше  3. >= – больше либо равно  4. <= – меньше либо равно  5. ! – проверка на неравенство  6. & – проверка на равенство |
| Символьные | 1**.** = – присваивание |

**1.13 Выражения и их вычисление**

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

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

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

**1.14 Конструкции языка**

Программа на языке UVV-2024 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода.

Ключевые программные конструкции языка программирования UVV-2024 представлены в таблице 1.6.

Таблица 1.6 – Программные конструкции языка UVV-2024

|  |  |
| --- | --- |
| Конструкция | Реализация |
| Главная функция | main{… } |
| Функция | function <тип данных> <идентификатор> [<тип данных> <идентификатор>, …]  { …  return <выражение>;  } |
| Процедура | procedure <идентификатор> [<тип данных> <идентификатор>, …]  { … } |
| Условная конструкция | if [<условие>]  { … }  else  { … }  <условное выражение> - может представлять собой два идентификатора или один идентификатор и один литерал, а также условный оператор между ними, или один идентификатор или литерал (в таком случае идентификатор или литерал используется в качестве булевого значения, то есть условие истинно, если значение не равно 0) |
| Цикл | while [<условие>]  { … }  <условное выражение> - может представлять собой два идентификатора или один идентификатор и один литерал, а также условный оператор между ними, или один идентификатор или литерал (в таком случае идентификатор или литерал используется в качестве булевого значения, то есть условие истинно, если значение не равно 0) |

**1.15 Области видимости идентификаторов**

Область видимости: сверху вниз (как и в С++). Переменные, объявленные в одной функции, недоступны в другой.

Все объявления и операции с переменными происходят внутри какого-либо блока. Каждая переменная или параметр функции получают префикс – название функции, внутри которой они находятся.

Все идентификаторы являются локальными и обязаны быть объявленными внутри какой-либо функции. Глобальных переменных нет. Параметры видны только внутри функции, в которой объявлены.

**1.16 Семантические проверки**

В языке программирования UVV-2024 выполняются следующие семантические проверки:

* наличие точки входа в программу;
* наличие типа идентификатора;
* совпадение типов данных в выражении;
* деление на ноль;
* проверка типа возвращаемого значения;
* проверка передаваемых параметров;
* единственность точки входа;
* переопределение идентификаторов;
* использование идентификаторов без их объявления;
* правильность строковых и символьных выражений;
* превышение размера символьных, строковых и числовых литералов;
* правильность составленного условия цикла/условного оператора.

**1.17 Распределение оперативной памяти на этапе выполнения**

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

**1.18 Стандартная библиотека и её состав**

В языке UVV-2024 присутствует стандартная библиотека, которая подключается автоматически при трансляции исходного кода в язык ассемблера. Содержимое библиотеки и описание функций представлено в таблице 1.7.

Таблица 1.7 – Стандартная библиотека языка UVV-2024

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| combine | string | string\_a, string\_b - строки | Функция производит конкатенацию строк a и b, возвращает строку |

Окончание таблицы 1.7

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| measure | int | string\_a | Функция вычисляет длину строки a |
| exponent | int | int\_a, int\_n | Функция возводит число a в степень n, возвращает результат |
| randomize | int | int\_min, int­­­\_max | Функция возвращает случайно сгенерированное число в промежутке [min, max] |
| OutputStr | - | string\_a | Функция выводит в консоль строку a |
| OutputInt | - | int\_a | Функция выводит в консоль число a |
| lexcomp | int | string\_a, string\_b - строки | Функция производит сравнение числовых кодов символов в строках. Возвращает значение равное 0, если числовые коды строк равны, больше нуля, если числовой код a больше b, меньше нуля в противном случае. |

Стандартная библиотека написана на языке С++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций.

**1.19 Ввод и вывод данных**

В языке программирования UVV-2024 не предусмотрена возможность ввода данных. Для вывода информации в стандартный поток используется команда: write [<идентификатор или литерал>].

В зависимости от типа переданного параметра будет вызвана соответствующая функция: OutputStr для строковых значений или OutputInt для целых чисел. Эти функции являются частью стандартной библиотеки и описаны в таблице 1.7.

**1.20 Точка входа**

В языке UVV-2024 каждая программа должна содержать главную функцию (точку входа) **main**, с первой инструкции которой начнётся последовательное выполнение команд программы. Главная функция **main** не имеет типа возвращаемого значения, принимаемых параметров.

**1.21 Препроцессор**

Препроцессор, принимающий и выдающий некоторые данные на вход транслятору, в языке UVV-2024 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**

Язык UVV-2024 транслируется в язык ассемблера, а затем - в объектный код.

**1.24 Классификация сообщений транслятора**

Генерируемые транслятором сообщения определяют степень его информативности, то есть сообщения транслятора должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Сообщения транслятора приведены в таблице 1.8, а также в приложении А.

Таблица 1.8 – Классификация ошибок

|  |  |
| --- | --- |
| Номера ошибок | Характеристика |
| 0-110 | Системные ошибки, ошибки параметров |
| 200-220 | Ошибки лексического анализа |
| 600-620 | Ошибки синтаксического анализа |
| 300-400 | Ошибки семантического анализа |

**1.25 Контрольный пример**

Контрольный пример демонстрирует главные особенности языка UVV-2024: его фундаментальные типы, основные структуры, функции, процедуры, использование функций статической библиотеки. Исходный код контрольного примера представлен в приложении А.

# **2 Структура транслятора**

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

В языке UVV-2024 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые им функции. Для того чтобы получить ассемблерный код, используется выходные данные работы лексического анализатора, а именно таблица лексем и таблица идентификаторов. Для указания выходных файлов используются входные параметры транслятора. Структура транслятора языка UVV-2024 приведена на рисунке 2.1.



Рисунок 2.1 – Структура транслятора языка программирования UVV-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, парсера).

Цели лексического анализатора:

* убрать все лишние пробелы и комментарии;
* 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 анализатор – часть транслятора, выполняющая семантический анализ, то есть проверку исходного кода на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Генератор кода – часть транслятора, выполняющая генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

**2.2 Перечень параметров транслятора**

Для формирования файлов с результатами работы лексического, синтаксического и семантического анализаторов используются входные параметры транслятора, которые приведены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка UVV-2024

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке UVV-2024 , имеющий расширение .txt | Не предусмотрено |
| -log:<путь к log-файлу> | Файл журнала для вывода протоколов работы программы. | <имя in-файла>.log |
| -out:<путь к out-файлу> | Выходной файл с кодом на языке ассемблера, имеющий расширение .asm | <имя in-файла>.asm |
| -tokens | Ключ для вывода промежуточного представления кода | По умолчанию отсутствует |
| -lex | Ключ для вывода таблицы лексем в консоль | По умолчанию отсутствует |
| -id | Ключ для вывода трассировки синтаксического анализа в файл | По умолчанию отсутствует |

**2.3 Протоколы, формируемые транслятором**

В ходе работы программы формируются протоколы работы лексического, синтаксического и семантического анализаторов, которые содержат в себе перечень протоколов работы. В таблице 2.2 приведены протоколы, формируемые транслятором и их содержимое.

Таблица 2.2 – Протоколы, формируемые транслятором языка UVV-2024

|  |  |
| --- | --- |
| Файл журнала, заданный параметром "-log:" | Содержит информацию о времени выполнения приложения; входных параметрах в приложение; код на языке UVV-2024 с сепараторами и без избыточных пробелов, табуляций и переходов на новую строку; таблицу идентификаторов; таблицу лексем; промежуточное представление кода; трассировку синтаксического анализа; дерево разбора, время выполнения разбора; промежуточное представление кода после приведения его к польской нотации. |
| Выходной файл, c расширением ".asm" | Результат работы программы – сгенерированный код на языке Ассемблера. |

# **3 Разработка лексического анализатора**

**3.1 Структура лексического анализатора**

Лексический анализатор – часть транслятора, выполняющая лексический анализ. Лексический анализатор принимает обработанный и разбитый на отдельные компоненты исходный код на языке UVV-2024. На выходе формируется таблица лексем и таблица идентификаторов. Структура лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1 – Структура лексического анализатора UVV-2024

**3.2 Контроль входных символов**

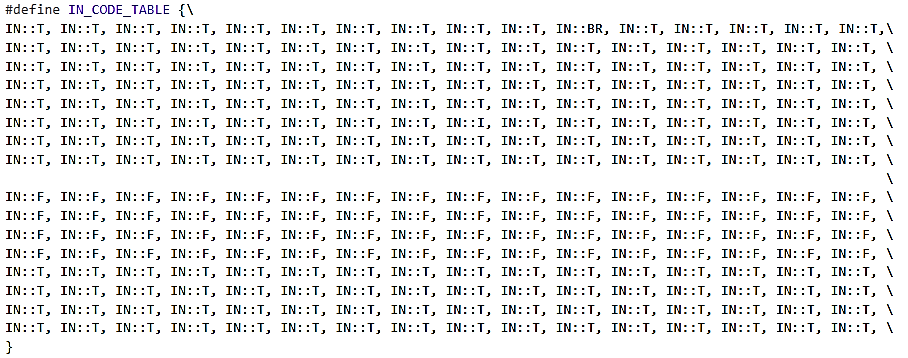
Таблица для контроля входных символов представлена на рисунке 3.2

Рисунок 3.2. – Таблица контроля входных символов

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

Описание значения символов: T – разрешённый символ, F – запрещённый символ, I – игнорируемый символ, BR – символ перехода на новую строку (\n).

**3.3 Удаление избыточных символов**

Избыточными символами являются символы табуляции, пробелы и комментарии. Избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

* посимвольно считываем файл с исходным кодом программы;
* встреча пробела или знака табуляции является своего рода встречей символа-сепаратора;
* в отличие от других символов-сепараторов не записываем в очередь лексем эти символы, т.е. игнорируем.

## **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 | declare | c |
| int, char, string, bool | t |
| main | m |
| function | f |
| procedure | p |
| return | r |
| write | s |
| while | u |
| when | w |
| or | ! |
| Иное | Идентификатор | i |
| Литерал | l |
| Функции стандартной библиотеки | combine | k |
| lexcomp | s |
| measure | d |
| exponent | e |
| break | b |
| randomize | z |

Окончание таблицы 3.1

|  |  |  |
| --- | --- | --- |
| Сепараторы | ; | ; |
| , | , |
| { | { |
| } | } |
| ( | ( |
| ) | ) |
| [ | [ |
| ] | ] |
| Операторы | + | + |
| - | - |
| \* | \* |
| / | / |
| % | % |
| Логические (~ # > <) | g |
| = | = |

Пример реализации таблицы лексем представлен в Приложении Б.

Также в Приложении B находятся конечные автоматы, соответствующие лексемам языка UVV-2024.

## **3.5 Основные структуры данных**

Описание основных структур данных, используемых для хранения таблиц идентификаторов, представлено в Приложении К.

Описание основных структур данных, используемых для хранения таблиц лексем, представлено на рис. 3.3.

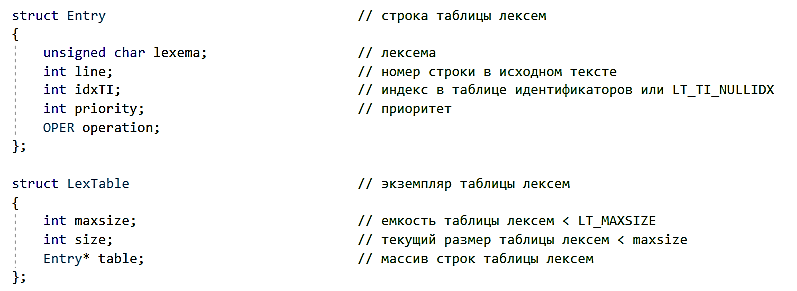


Рисунок 3.3 — Структуры таблиц лексем UVV-2024

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

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке.

При возникновении сообщения, лексический анализатор игнорирует найденную ошибку и продолжает работу с исходным кодом. Если в процессе лексического анализа будет выявлена ошибка, работа компилятора будет закончена. Перечень сообщений лексического анализатора представлен на рисунке 3.4.

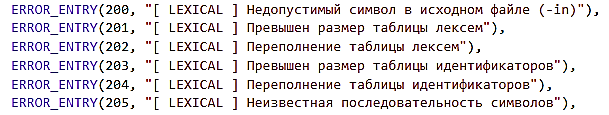


Рисунок 3.4 – Перечень ошибок лексического анализатора

## **3.7 Принцип обработки ошибок**

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением.

Когда возникает критическая ошибка – работа транслятора прекращается.

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

Входным параметром лексического анализатора является исходный текст программы, написанный на языке UVV-2024, а также файл протокола.

## **3.9 Алгоритм лексического анализа**

* проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для Вычисления номера строки для каждой лексемы;
* для выделенной части входного потока выполняется функция распознавания лексемы;
* при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
* формирует протокол работы;
* при неуспешном распознавании выдается сообщение об ошибке.

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример. Регулярное выражение для ключевого слова main.

Граф конечного автомата для этой лексемы представлен на рисунке 3.5. S0 – начальное состояние, S4 – конечное состояние автомата. В виде кода представлен на рисунке 3.6.

|  |
| --- |
| n  m  a  i |

Рисунок 3.5 — Граф переходов для цепочки “main”

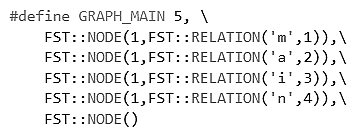


Рисунок 3.6 — Граф переходов для цепочки “main”

**3.10 Контрольный пример**

Результат работы лексического анализатора в виде таблиц лексем и идентификаторов, соответствующих контрольному примеру, представлен в приложении Б.

**4. Разработка синтаксического анализатора**

**4.1 Структура синтаксического анализатора**

Синтаксический анализатор: часть компилятора, выполняющая синтаксический анализ, то есть исходный код проверяется на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией– дерево разбора

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.



Рисунок 4.1 – Структура синтаксического анализатора.

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

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

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

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

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

S – начальный символ грамматики, являющийся нетерминалом.

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

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

Правила языка UVV-2024 представлена в Приложении Г.

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

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

Перечень правил, составляющих грамматику языка и описание нетерминальных символов UVV-2024 показаны в приложении Г.

## **4.3 Построение конечного магазинного автомата**

Конечный автомат с магазинной памятью представляет собой семерку, описание которой представлено в таблице 4.1. Структура данного автомата показана в Приложении Д.

Таблица 4.1 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит 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 |

**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 UVV-2024. Данные структуры представлены в приложении Д.

**4.5 Описание алгоритма синтаксического разбора**

Принцип работы синтаксического анализатора, 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 на рисунке 4.1.

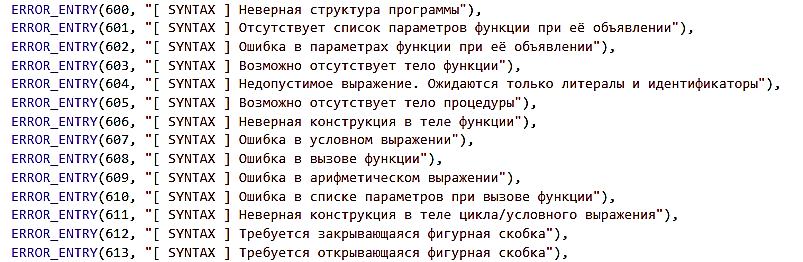


Рисунок 4.1 – Перечень сообщений синтаксического анализатора

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

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

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

**4.8. Принцип обработки ошибок**

Синтаксический анализатор выполняет разбор исходной последовательности лексем до тех пор, пока не дойдёт до конца цепочки лексем. При ошибке анализ останавливается и выводится сообщение.

## **4.9 Контрольный пример**

Пример разбора синтаксическим анализатором исходного кода предоставлен в Приложении Е в виде фрагмента трассировки и дерева разбора исходного кода.

**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

## **5.2 Функции семантического анализатора**

За семантический анализ отвечает функция Analyze. Ее входными параметрами является таблица лексем и поток вывода в протокол.

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

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

Сообщения, формируемые семантическим анализатором, представлены на рисунке 5.2.

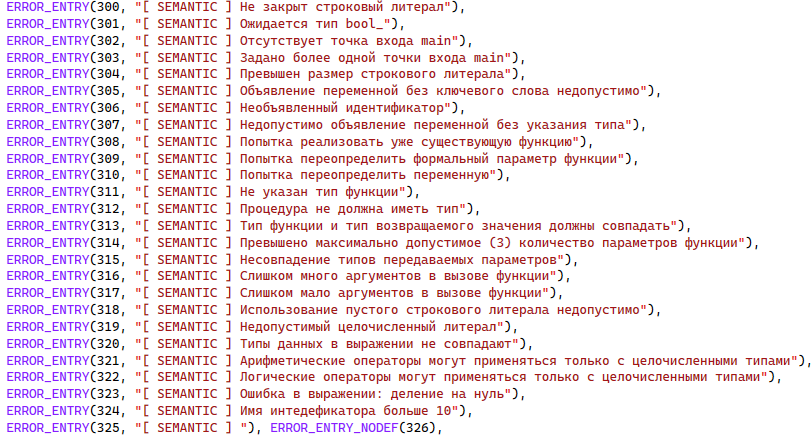


Рисунок 5.2 – Перечень сообщений семантического анализатора

**5.4 Принцип обработки ошибок**

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами.

В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Анализ останавливается после того, как будут найдены все ошибки.

## **5.5 Контрольный пример**

Результат работы контрольного примера расположен в Приложении Б, где показан результат лексического анализатора, т.к. представленные таблицы лексем и идентификаторов проходят лексическую и семантическую проверки одновременно.

Таблица 5.1 – Тестирование функций

|  |  |
| --- | --- |
| Исходный код с ошибкой | Генерируемое сообщение об ошибке |
| {  declare int a = 5;  write [a]; break;  } | ERROR CODE 302: [ SEMANTIC ] Отсутствует точка входа main  Строка -1 позиция -1 |
| main  {  declare int a = 5;  write [a]; break;  write ["Bye!];  } | ERROR CODE 300: [ SEMANTIC ] Не закрыт строковый литерал  Строка -1 позиция -1 |
| main  {  declare a = 5;  write [a]; break;  write ["Bye!"];  } | ERROR CODE 307: [ SEMANTIC ] Недопустимо объявление переменной без указания типа  Строка 2 позиция 0 |

# **6. Преобразование выражений**

## **6.1 Выражения, допускаемые языком**

В языке UVV-2024 допускаются выражения, применимые к целочисленным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, \*, /, %(остаток от деления) и (), и вызовы функций как операнды арифметических выражений.

Приоритет операций представлен в таблице 6.1.

Таблица 6.1 – Приоритет операций в языке UVV-2024

|  |  |
| --- | --- |
| Приоритет | Операция |
| 0 | ( ) |
| 1 | , |
| 2 | + - |
| 3 | \* / % |
| 4 | [ ] – скобки параметров функции |

**6.2 Польская запись и принцип её построения**

Все выражения языка UVV-2024 преобразовываются к обратной польской записи. Польская запись — это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок. Существует два типа польской записи: прямая и обратная, также известные как префиксная и постфиксная. Отличие их от классического, инфиксного способа заключается в том, что знаки операций пишутся не между, а, соответственно, до или после аргументов.

Алгоритм построения:

– исходная строка: выражение;

– результирующая строка: польская запись;

– стек: пустой;

– результирующая строка: польская запись;

– исходная строка просматривается слева направо;

– операнды переносятся в результирующую строку в порядке их следования;

– операция записывается в стек, если стек пуст или в вершине стека лежит отрывающая скобка;

– операция выталкивает все операции с большим или равным приоритетом в результирующую строку;

– запятая не помещается в стек, если в стеке операции, то все выбираются в строку;

– отрывающая скобка помещается в стек;

– закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются;

– закрывающая скобка с приоритетом, равным 4, выталкивает все до открывающей с таким же приоритетом и генерирует @ – специальный символ, в которого записывается информация о вызываемой функции, а в поле приоритета для данной лексемы записывается число параметров вызываемой функции;

– по концу разбора исходной строки все операции, оставшиеся в стеке, выталкиваются в результирующую строку.

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Исходная строка | Результирующая строка | Стек |
| (l+l)\*l/l |  |  |
| l+l)\*l/l |  |  |
| +l)\*l/l | l |  |
| l)\*l/l | l | + |
| )\*l/l | ll | + |
| \*l/l | ll+ |  |
| l/l | ll+ | \* |
| /l | ll+l | \* |
| l | ll+l\* | / |
|  | ll+l\*l | / |
|  | ll+l\*l/ |  |

## **6.3 Программная реализация обработки выражений**

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

## **6.4 Контрольный пример**

Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления.

В Приложении Ж приведен измененное представление промежуточного кода, отображающее результаты преобразования выражений в польский формат.

**7. Генерация кода**

**7.1 Структура генератора кода**

В языке UVV-2024 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода UVV-2024 представлена на рисунке 7.1.



Рисунок 7.1 – Структура генератора кода

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

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

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

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке UVV-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| int | SDWORD | Хранит целочисленный тип данных со знаком. |
| char | DWORD | Хранит целочисленный тип данных без знака, размером 1 байт. |
| string | DWORD | Каждый символ строки хранится размером в 1 байт. |
| bool | SDWORD | Хранит целочисленный тип данных без знака. |

## **7.3 Статическая библиотека**

Функции стандартной библиотеки находятся в проекте LIB, который настроен как «статическая библиотека» в свойствах конфигурации. Подключение библиотеки осуществляется через команду includelib на этапе генерации кода, с выводом информации в поток out. Объявление имен функций из библиотеки выполняется с помощью оператора EXTRN.

Оператор EXTRN выполняет две задачи. Во-первых, он указывает ассемблеру, что заданное символическое имя является внешним по отношению к текущему процессу ассемблирования. Во-вторых, EXTRN определяет тип указанного символического имени. Поскольку процесс ассемблирования строго формализован, ассемблеру необходимо точно знать, к какому типу относится каждый символ.

**7.4 Особенности алгоритма генерации кода**

В языке UVV-2024 генерация кода строится на основе таблиц лексем и идентификаторов. Схема работы генератора кода представлена на рисунке 7.2.

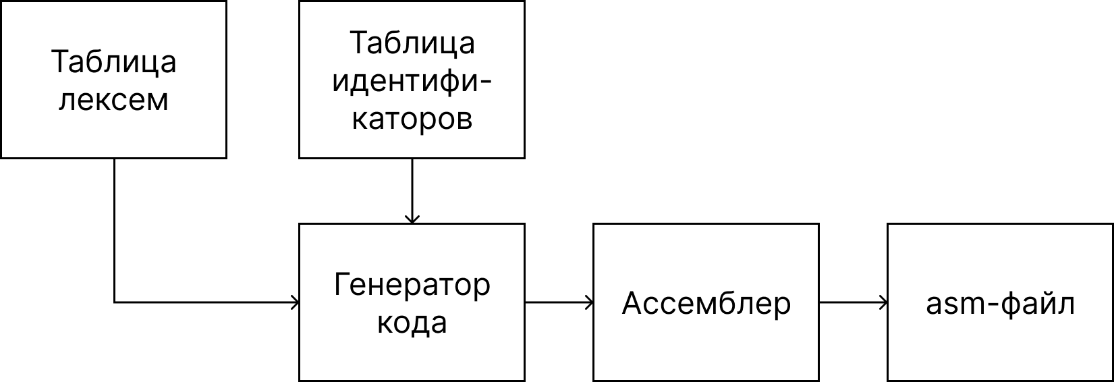


Рисунок 7.2 – Структура генератора кода

**7.5 Параметры, управляющие генерацией кода**

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

**7.6 Контрольный пример**

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении И. Результат работы контрольного примера приведён на рисунке 7.3.

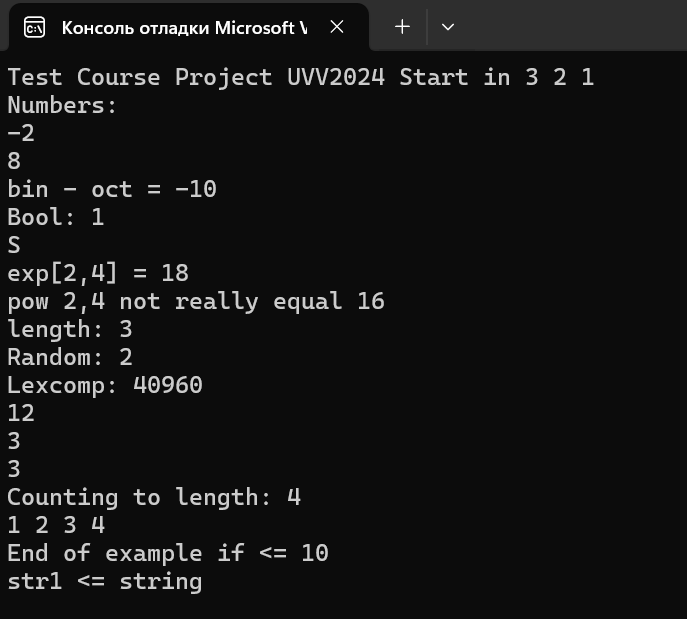


Рисунок 7.3 Результат работы программы на языке UVV-2024

# **8. Тестирование транслятора**

**8.1 Общие положения**

Тесты применяются для оценки функциональности транслятора, выявления ошибок и недочетов, а также для их последующего исправления.

## **8.2 Результаты тестирования**

В языке UVV-2024 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| Fu№аtion int aver [int n, int m] | ERROR CODE 200: [ LEXICAL ] Недопустимый символ в исходном файле (-in)  Строка 1 позиция 4 |

На этапе лексического анализа могут возникнуть ошибки, описанные в пункте 3.7. Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 – Тестирование лексического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| function intfhgff\_ aver [int n, int m] | ERROR CODE 205: [ LEXICAL ] Неизвестная последовательность символов  Строка 0 позиция 9 |

На этапе синтаксического анализа могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 – Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main  {  declare int hey = 5;  write [hey]; break; | ERROR CODE 612: [ SYNTAX ] Требуется закрывающаяся фигурная скобка  Строка -1 позиция -1 |

Итоги тестирования семантического анализатора приведены в пункте 5.5.

**Заключение**

В рамках курсовой работы был создан транслятор и генератор кода для языка программирования UVV-2024, включающий все необходимые компоненты. Таким образом, были выполнены основные задачи данной курсовой работы:

* сформулирована спецификация языка UVV-2024;
* разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
* осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
* разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
* осуществлена программная реализация синтаксического анализатора;
* разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
* разработан транслятор кода на язык ассемблера;
* проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка UVV-2024 включает:

* возможность вызова функций стандартной библиотеки;
* наличие 4 типов данных;
* наличие 5 арифметических операторов для вычисления выражений;
* наличие 6 логических операторов для вычисления выражений;
* поддержка функций, процедур, операторов цикла и условия;
* структурированная и классифицированная система для обработки ошибок пользователя;
* поддержка операторов вывода.

Проведённая работа дала возможность ознакомиться с структурами и процессами, используемыми при создании трансляторов, а также понять ключевые различия и преимущества различных методов трансляции.

# **Список использованных источников**

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

2 Ирвин К. Р. Язык ассемблера для процессоров Intel / К. Р. Ирвин. – M.: Вильямс, 2005. – 912с.

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

4 Курс лекций по КПО / Наркевич А.С.

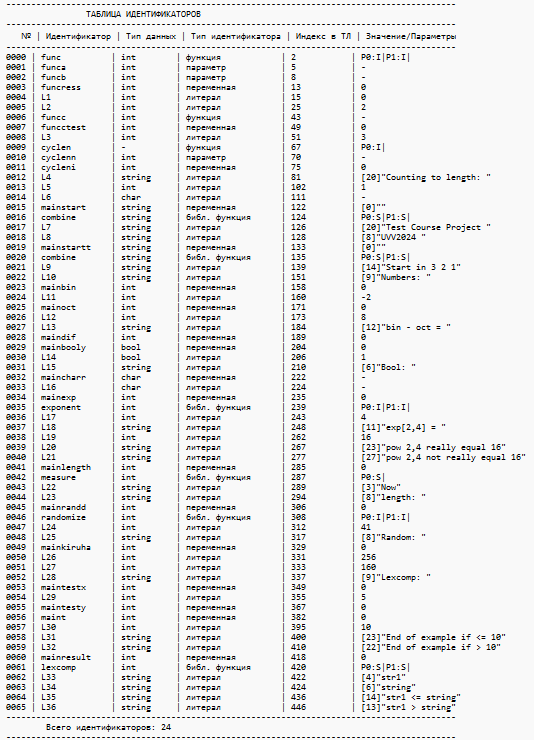
# **Приложение А**

Контрольный пример

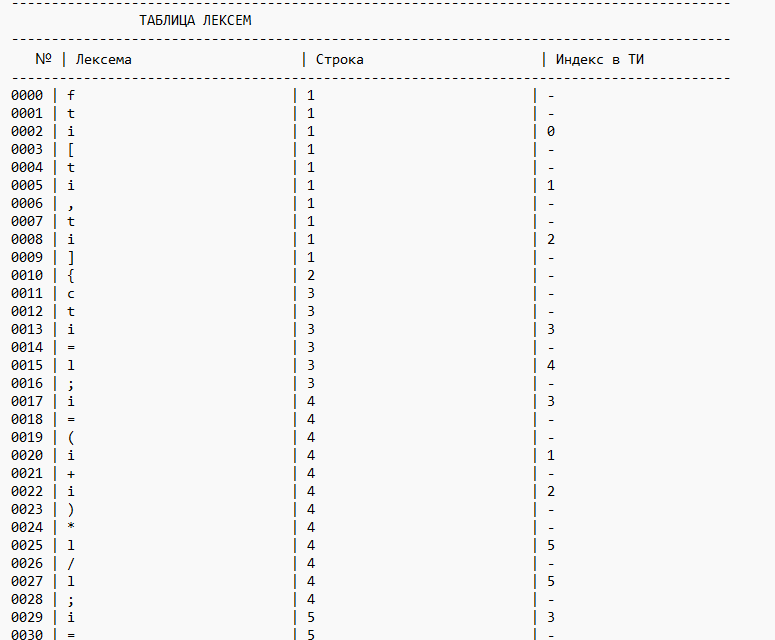
|  |
| --- |
| function int func [int a, int b]  {  declare int ress = 0;  ress = (a + b)\* 2 /2;  ress = ress \* 2;  return [ress];  }  function int funcc []  {  declare int test = 3;  write[test]; break;  return[test];  }  procedure cyclen [int n]  {  declare int i = 0;  write ["Counting to length: "];  write [n]; break;  while [i < n]  {  i = i + 1;  write [i]; write [" "];  }  break;  }  main  {  declare string start = combine ["Test Course Project ", "UVV2024 "];  declare string startt = combine [start, "Start in 3 2 1"];  write [startt]; break;    write ["Numbers: "]; break;  declare int bin = 1b10;  write [bin]; break;  declare int oct = 0o10;  write [oct]; break;  write ["bin - oct = "];  declare int dif = bin - oct;  write [dif]; break;  declare bool booly = true;  write["Bool: "];  write[booly]; break;  declare char charr = "S";  write [charr]; break;  declare int exp = 2 + exponent [2, 4];  write ["exp[2,4] = "];  write [exp]; break;  if[exp ~ 16]  {  write["pow 2,4 really equal 16"]; break;  }  else  {  write["pow 2,4 not really equal 16"]; break;  }  declare int length = measure ["Now"];  write ["length: "];  write [length]; break;  declare int randd = randomize [1, 41];  write ["Random: "];  write [randd]; break;    declare int kiruha = 256\*160;  write ["Lexcomp: "];  write [kiruha]; break;    declare int testx = func[1,5];  write [testx]; break;  declare int testy = funcc[];  write [testy]; break;  declare int t = 4;  cyclen[t];    if[t $ 10]  {  write ["End of example if <= 10"]; break;  }  else  {  write ["End of example if > 10"]; break;  }  declare int result = lexcomp["str1", "string"];  if [result $ 0]  {  write ["str1 <= string"]; break;  }  else  {  write ["str1 > string"]; break;  }  } |

# **Приложение Б**

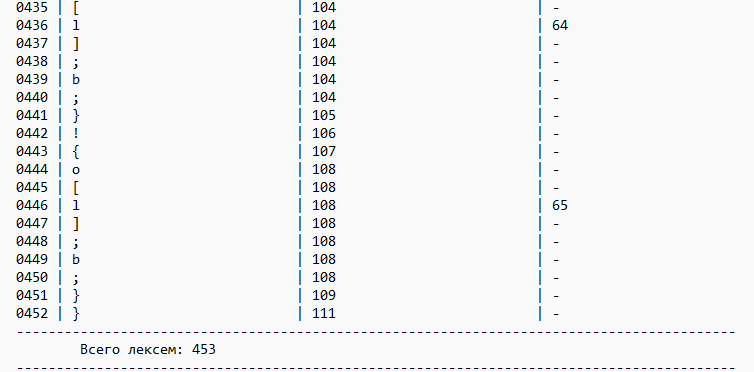
Таблица идентификаторов



Начало таблицы лексем



Конец таблицы лекесм



# **Приложение В**

Конечные автоматы, соответствующие лексемам языка

|  |
| --- |
| #define GRAPH\_Declare 8, \  FST::NODE(1, FST::RELATION('d', 1)), \  FST::NODE(1, FST::RELATION('e', 2)), \  FST::NODE(1, FST::RELATION('c', 3)), \  FST::NODE(1, FST::RELATION('l', 4)), \  FST::NODE(1, FST::RELATION('a', 5)), \  FST::NODE(1, FST::RELATION('r', 6)), \  FST::NODE(1, FST::RELATION('e', 7)), \  FST::NODE()  #define GRAPH\_INT 4, \  FST::NODE(1,FST::RELATION('i',1)),\  FST::NODE(1,FST::RELATION('n',2)),\  FST::NODE(1,FST::RELATION('t',3)),\  FST::NODE()  #define GRAPH\_CHR 5, \  FST::NODE(1,FST::RELATION('c',1)),\  FST::NODE(1,FST::RELATION('h',2)),\  FST::NODE(1,FST::RELATION('a',3)),\  FST::NODE(1,FST::RELATION('r',4)),\  FST::NODE()  #define GRAPH\_STR 7, \  FST::NODE(1,FST::RELATION('s',1)),\  FST::NODE(1,FST::RELATION('t',2)),\  FST::NODE(1,FST::RELATION('r',3)),\  FST::NODE(1,FST::RELATION('i',4)),\  FST::NODE(1,FST::RELATION('n',5)),\  FST::NODE(1,FST::RELATION('g',6)),\  FST::NODE()  #define GRAPH\_BOOL 5, \  FST::NODE(1,FST::RELATION('b',1)),\  FST::NODE(1,FST::RELATION('o',2)),\  FST::NODE(1,FST::RELATION('o',3)),\  FST::NODE(1,FST::RELATION('l',4)),\  FST::NODE()  #define GRAPH\_TRUE 5, \  FST::NODE(1,FST::RELATION('t',1)),\  FST::NODE(1,FST::RELATION('r',2)),\  FST::NODE(1,FST::RELATION('u',3)),\  FST::NODE(1,FST::RELATION('e',4)),\  FST::NODE()  #define GRAPH\_FALSE 6, \  FST::NODE(1,FST::RELATION('f',1)),\  FST::NODE(1,FST::RELATION('a',2)),\  FST::NODE(1,FST::RELATION('l',3)),\  FST::NODE(1,FST::RELATION('s',4)),\  FST::NODE(1,FST::RELATION('e',5)),\  FST::NODE()  #define GRAPH\_BREAKL 6,\  FST::NODE(1, FST::RELATION('b', 1)), \  FST::NODE(1, FST::RELATION('r', 2)), \  FST::NODE(1, FST::RELATION('e', 3)), \  FST::NODE(1, FST::RELATION('a', 4)), \  FST::NODE(1, FST::RELATION('k', 5)), \  FST::NODE()  #define GRAPH\_OPERATOR 2, \  FST::NODE(5, FST::RELATION('+', 1), \  FST::RELATION('-', 1), \  FST::RELATION('\*', 1), \  FST::RELATION('/', 1), \  FST::RELATION('%', 1)),\  FST::NODE()  #define GRAPH\_LOG\_OPERATOR 2, \  FST::NODE(6, FST::RELATION('>', 1), \  FST::RELATION('<', 1), \  FST::RELATION('~', 1), \  FST::RELATION('#', 1), \  FST::RELATION('$', 1), \  FST::RELATION('?', 1)), \  FST::NODE()  #define GRAPH\_SEMICOLON 2, \  FST::NODE(1, FST::RELATION(';', 1)),\  FST::NODE()  #define GRAPH\_COMMA 2, \  FST::NODE(1, FST::RELATION(',', 1)),\  FST::NODE()  #define GRAPH\_LEFTBRACE 2, \  FST::NODE(1, FST::RELATION('{', 1)),\  FST::NODE()  #define GRAPH\_BRACELET 2, \  FST::NODE(1, FST::RELATION('}', 1)),\  FST::NODE()  #define GRAPH\_LEFTTHESIS 2, \  FST::NODE(1, FST::RELATION('[', 1)),\  FST::NODE()  #define GRAPH\_RIGHTTHESIS 2, \  FST::NODE(1, FST::RELATION(']', 1)),\  FST::NODE()  #define GRAPH\_LEFTBRACKET 2, \  FST::NODE(1, FST::RELATION('(', 1)),\  FST::NODE()  #define GRAPH\_RIGHTBRACKET 2, \  FST::NODE(1, FST::RELATION(')', 1)),\  FST::NODE()  #define GRAPH\_EQUAL 2, \  FST::NODE(1, FST::RELATION('=', 1)),\  FST::NODE()  #define GRAPH\_ID 2, \  FST::NODE(52, \  FST::RELATION('a', 0), FST::RELATION('b', 0), FST::RELATION('c', 0), \  FST::RELATION('d', 0), FST::RELATION('e', 0), FST::RELATION('f', 0),\  FST::RELATION('g', 0), FST::RELATION('h', 0), FST::RELATION('i', 0), \  FST::RELATION('j', 0), FST::RELATION('k', 0), FST::RELATION('l', 0),\  FST::RELATION('m', 0), FST::RELATION('n', 0), FST::RELATION('o', 0), \  FST::RELATION('p', 0), FST::RELATION('q', 0), FST::RELATION('r', 0),\  FST::RELATION('s', 0), FST::RELATION('t', 0), FST::RELATION('u', 0), \  FST::RELATION('v', 0), FST::RELATION('w', 0), FST::RELATION('x', 0),\  FST::RELATION('y', 0), FST::RELATION('z', 0),\  \  FST::RELATION('a', 1), FST::RELATION('b', 1), FST::RELATION('c', 1), \  FST::RELATION('d', 1), FST::RELATION('e', 1), FST::RELATION('f', 1),\  FST::RELATION('g', 1), FST::RELATION('h', 1), FST::RELATION('i', 1), \  FST::RELATION('j', 1), FST::RELATION('k', 1), FST::RELATION('l', 1),\  FST::RELATION('m', 1), FST::RELATION('n', 1), FST::RELATION('o', 1), \  FST::RELATION('p', 1), FST::RELATION('q', 1), FST::RELATION('r', 1),\  FST::RELATION('s', 1), FST::RELATION('t', 1), FST::RELATION('u', 1), \  FST::RELATION('v', 1), FST::RELATION('w', 1), FST::RELATION('x', 1),\  FST::RELATION('y', 1), FST::RELATION('z', 1)),\  FST::NODE()  #define GRAPH\_STRING\_LITERAL 3,\  FST::NODE(1, FST::RELATION('\"', 1)),\  FST::NODE(87, \  FST::RELATION('a', 1), FST::RELATION('b', 1), FST::RELATION('c', 1), FST::RELATION('d', 1),\  FST::RELATION('e', 1), FST::RELATION('f', 1), FST::RELATION('g', 1), FST::RELATION('h', 1),\  FST::RELATION('i', 1), FST::RELATION('j', 1), FST::RELATION('k', 1), FST::RELATION('l', 1),\  FST::RELATION('m', 1), FST::RELATION('n', 1), FST::RELATION('o', 1), FST::RELATION('p', 1),\  FST::RELATION('q', 1), FST::RELATION('r', 1), FST::RELATION('s', 1), FST::RELATION('t', 1),\  FST::RELATION('u', 1), FST::RELATION('v', 1), FST::RELATION('w', 1), FST::RELATION('x', 1),\  FST::RELATION('y', 1), FST::RELATION('z', 1),\  FST::RELATION('A', 1), FST::RELATION('B', 1), FST::RELATION('C', 1), FST::RELATION('D', 1),\  FST::RELATION('E', 1), FST::RELATION('F', 1), FST::RELATION('G', 1), FST::RELATION('H', 1),\  FST::RELATION('I', 1), FST::RELATION('J', 1), FST::RELATION('K', 1), FST::RELATION('L', 1),\  FST::RELATION('M', 1), FST::RELATION('N', 1), FST::RELATION('O', 1), FST::RELATION('P', 1),\  FST::RELATION('Q', 1), FST::RELATION('R', 1), FST::RELATION('S', 1), FST::RELATION('T', 1),\  FST::RELATION('U', 1), FST::RELATION('V', 1), FST::RELATION('W', 1), FST::RELATION('X', 1),\  FST::RELATION('Y', 1), FST::RELATION('Z', 1),\  FST::RELATION('0', 1), FST::RELATION('1', 1), FST::RELATION('2', 1), FST::RELATION('3', 1),\  FST::RELATION('4', 1), FST::RELATION('5', 1), FST::RELATION('6', 1), FST::RELATION('7', 1),\  FST::RELATION('8', 1), FST::RELATION('9', 1),\  FST::RELATION(' ', 1), FST::RELATION(',', 1), FST::RELATION('.', 1), FST::RELATION(';', 1),\  FST::RELATION('-', 1), FST::RELATION('+', 1), FST::RELATION('\*', 1), FST::RELATION('/', 1),\  FST::RELATION('=', 1), FST::RELATION(':', 1), FST::RELATION(')', 1), FST::RELATION('(', 1),\  FST::RELATION('}', 1), FST::RELATION('{', 1), FST::RELATION(']', 1), FST::RELATION('[', 1),\  FST::RELATION('!', 1), FST::RELATION('?', 1), FST::RELATION('#', 1), FST::RELATION('&', 1),\  FST::RELATION('>', 1), FST::RELATION('<', 1), FST::RELATION('[', 1), FST::RELATION(']', 1),\  FST::RELATION('\"', 2)),\  FST::NODE()  #define GRAPH\_CHR\_LITERAL 4,\  FST::NODE(1, FST::RELATION('\"', 1)),\  FST::NODE(87, \  FST::RELATION('a', 2), FST::RELATION('b', 2), FST::RELATION('c', 2), FST::RELATION('d', 2),\  FST::RELATION('e', 2), FST::RELATION('f', 2), FST::RELATION('g', 2), FST::RELATION('h', 2),\  FST::RELATION('i', 2), FST::RELATION('j', 2), FST::RELATION('k', 2), FST::RELATION('l', 2),\  FST::RELATION('m', 2), FST::RELATION('n', 2), FST::RELATION('o', 2), FST::RELATION('p', 2),\  FST::RELATION('q', 2), FST::RELATION('r', 2), FST::RELATION('s', 2), FST::RELATION('t', 2),\  FST::RELATION('u', 2), FST::RELATION('v', 2), FST::RELATION('w', 2), FST::RELATION('x', 2),\  FST::RELATION('y', 2), FST::RELATION('z', 2),\  FST::RELATION('A', 2), FST::RELATION('B', 2), FST::RELATION('C', 2), FST::RELATION('D', 2),\  FST::RELATION('E', 2), FST::RELATION('F', 2), FST::RELATION('G', 2), FST::RELATION('H', 2),\  FST::RELATION('I', 2), FST::RELATION('J', 2), FST::RELATION('K', 2), FST::RELATION('L', 2),\  FST::RELATION('M', 2), FST::RELATION('N', 2), FST::RELATION('O', 2), FST::RELATION('P', 2),\  FST::RELATION('Q', 2), FST::RELATION('R', 2), FST::RELATION('S', 2), FST::RELATION('T', 2),\  FST::RELATION('U', 2), FST::RELATION('V', 2), FST::RELATION('W', 2), FST::RELATION('X', 2),\  FST::RELATION('Y', 2), FST::RELATION('Z', 2),\  FST::RELATION('0', 2), FST::RELATION('1', 2), FST::RELATION('2', 2), FST::RELATION('3', 2),\  FST::RELATION('4', 2), FST::RELATION('5', 2), FST::RELATION('6', 2), FST::RELATION('7', 2),\  FST::RELATION('8', 2), FST::RELATION('9', 2),\  FST::RELATION(' ', 2), FST::RELATION(',', 2), FST::RELATION('.', 2), FST::RELATION(';', 2),\  FST::RELATION('-', 2), FST::RELATION('+', 2), FST::RELATION('\*', 2), FST::RELATION('/', 2),\  FST::RELATION('=', 2), FST::RELATION(':', 2), FST::RELATION(')', 2), FST::RELATION('(', 2),\  FST::RELATION('}', 2), FST::RELATION('{', 2), FST::RELATION(']', 2), FST::RELATION('[', 2),\  FST::RELATION('!', 2), FST::RELATION('?', 2), FST::RELATION('#', 2), FST::RELATION('&', 2),\  FST::RELATION('>', 2), FST::RELATION('<', 2), FST::RELATION('[', 2), FST::RELATION(']', 2)),\  FST::NODE(1, FST::RELATION('\"', 3)), FST::NODE()  #define GRAPH\_INT\_LITERAL 3, \  FST::NODE(19, \  FST::RELATION('1',1),\  FST::RELATION('2',1),FST::RELATION('3',1),\  FST::RELATION('4',1),FST::RELATION('5',1),\  FST::RELATION('6',1),FST::RELATION('7',1),\  FST::RELATION('8',1),FST::RELATION('9',1),\  \  FST::RELATION('0',2),FST::RELATION('1',2),\  FST::RELATION('2',2),FST::RELATION('3',2),\  FST::RELATION('4',2),FST::RELATION('5',2),\  FST::RELATION('6',2),FST::RELATION('7',2),\  FST::RELATION('8',2),FST::RELATION('9',2)),\  FST::NODE(20,\  FST::RELATION('0',1), FST::RELATION('1',1),\  FST::RELATION('2',1),FST::RELATION('3',1),\  FST::RELATION('4',1),FST::RELATION('5',1),\  FST::RELATION('6',1),FST::RELATION('7',1),\  FST::RELATION('8',1),FST::RELATION('9',1),\  \  FST::RELATION('0',2),FST::RELATION('1',2),\  FST::RELATION('2',2),FST::RELATION('3',2),\  FST::RELATION('4',2),FST::RELATION('5',2),\  FST::RELATION('6',2),FST::RELATION('7',2),\  FST::RELATION('8',2),FST::RELATION('9',2)),\  FST::NODE()  #define GRAPH\_BIN\_INT\_LITERAL 4, \  FST::NODE(2,FST::RELATION('0', 1),FST::RELATION('1', 1)), \  FST::NODE(1,FST::RELATION('b', 2)), \  \  FST::NODE(4, \  FST::RELATION('1', 2), FST::RELATION('0', 2),\  \  FST::RELATION('1', 3), FST::RELATION('0', 3)),\  FST::NODE()  #define GRAPH\_OCT\_INT\_LITERAL 4, \  FST::NODE(2,FST::RELATION('0', 1),FST::RELATION('1', 1)), \  FST::NODE(1,FST::RELATION('o', 2)), \  \  FST::NODE(16, \  FST::RELATION('1', 3), FST::RELATION('2', 3), FST::RELATION('3', 3), FST::RELATION('4', 3), FST::RELATION('5', 3), FST::RELATION('6', 3),\  FST::RELATION('7', 3), FST::RELATION('0', 3),\  \  FST::RELATION('1', 2), FST::RELATION('2', 2), FST::RELATION('3', 2), FST::RELATION('4', 2), FST::RELATION('5', 2), FST::RELATION('6', 2),\  FST::RELATION('7', 2), FST::RELATION('0', 2)),\  FST::NODE()  #define GRAPH\_FUNCTION 9, \  FST::NODE(1, FST::RELATION('f', 1)),\  FST::NODE(1, FST::RELATION('u', 2)),\  FST::NODE(1, FST::RELATION('n', 3)),\  FST::NODE(1, FST::RELATION('c', 4)),\  FST::NODE(1, FST::RELATION('t', 5)),\  FST::NODE(1, FST::RELATION('i', 6)),\  FST::NODE(1, FST::RELATION('o', 7)),\  FST::NODE(1, FST::RELATION('n', 8)),\  FST::NODE()  #define GRAPH\_PROCEDURE 10, \  FST::NODE(1, FST::RELATION('p', 1)),\  FST::NODE(1, FST::RELATION('r', 2)),\  FST::NODE(1, FST::RELATION('o', 3)),\  FST::NODE(1, FST::RELATION('c', 4)),\  FST::NODE(1, FST::RELATION('e', 5)),\  FST::NODE(1, FST::RELATION('d', 6)),\  FST::NODE(1, FST::RELATION('u', 7)),\  FST::NODE(1, FST::RELATION('r', 8)),\  FST::NODE(1, FST::RELATION('e', 9)),\  FST::NODE()  #define GRAPH\_MAIN 5, \  FST::NODE(1,FST::RELATION('m',1)),\  FST::NODE(1,FST::RELATION('a',2)),\  FST::NODE(1,FST::RELATION('i',3)),\  FST::NODE(1,FST::RELATION('n',4)),\  FST::NODE()  #define GRAPH\_WHERE 3, \  FST::NODE(1,FST::RELATION('i',1)),\  FST::NODE(1,FST::RELATION('f',2)),\  FST::NODE()  #define GRAPH\_OTHERWISE 5, \  FST::NODE(1,FST::RELATION('e',1)),\  FST::NODE(1,FST::RELATION('l',2)),\  FST::NODE(1,FST::RELATION('s',3)),\  FST::NODE(1,FST::RELATION('e',4)),\  FST::NODE()  #define GRAPH\_CYCLE 6, \  FST::NODE(1,FST::RELATION('w',1)),\  FST::NODE(1,FST::RELATION('h',2)),\  FST::NODE(1,FST::RELATION('i',3)),\  FST::NODE(1,FST::RELATION('l',4)),\  FST::NODE(1,FST::RELATION('e',5)),\  FST::NODE()  #define GRAPH\_SPEAK 6, \  FST::NODE(1,FST::RELATION('w',1)),\  FST::NODE(1,FST::RELATION('r',2)),\  FST::NODE(1,FST::RELATION('i',3)),\  FST::NODE(1,FST::RELATION('t',4)),\  FST::NODE(1,FST::RELATION('e',5)),\  FST::NODE()  #define GRAPH\_GIVE 7, \  FST::NODE(1,FST::RELATION('r',1)),\  FST::NODE(1,FST::RELATION('e',2)),\  FST::NODE(1,FST::RELATION('t',3)),\  FST::NODE(1,FST::RELATION('u',4)),\  FST::NODE(1,FST::RELATION('r',5)),\  FST::NODE(1,FST::RELATION('n',6)),\  FST::NODE()  // библиотечные функции  #define GRAPH\_MEASURE 8, \  FST::NODE(1, FST::RELATION('m', 1)),\  FST::NODE(1, FST::RELATION('e', 2)),\  FST::NODE(1, FST::RELATION('a', 3)),\  FST::NODE(1, FST::RELATION('s', 4)),\  FST::NODE(1, FST::RELATION('u', 5)),\  FST::NODE(1, FST::RELATION('r', 6)),\  FST::NODE(1, FST::RELATION('e', 7)),\  FST::NODE()  #define GRAPH\_READ 5, \  FST::NODE(1, FST::RELATION('r', 1)),\  FST::NODE(1, FST::RELATION('e', 2)),\  FST::NODE(1, FST::RELATION('a', 3)),\  FST::NODE(1, FST::RELATION('d', 4)),\  FST::NODE()  #define GRAPH\_COMBINE 8, \  FST::NODE(1, FST::RELATION('c', 1)),\  FST::NODE(1, FST::RELATION('o', 2)),\  FST::NODE(1, FST::RELATION('m', 3)),\  FST::NODE(1, FST::RELATION('b', 4)),\  FST::NODE(1, FST::RELATION('i', 5)),\  FST::NODE(1, FST::RELATION('n', 6)),\  FST::NODE(1, FST::RELATION('e', 7)),\  FST::NODE()  #define GRAPH\_LEXCOMP 8, \  FST::NODE(1, FST::RELATION('l', 1)),\  FST::NODE(1, FST::RELATION('e', 2)),\  FST::NODE(1, FST::RELATION('x', 3)),\  FST::NODE(1, FST::RELATION('c', 4)),\  FST::NODE(1, FST::RELATION('o', 5)),\  FST::NODE(1, FST::RELATION('m', 6)),\  FST::NODE(1, FST::RELATION('p', 7)),\  FST::NODE()  #define GRAPH\_EXPONENT 9, \  FST::NODE(1, FST::RELATION('e', 1)),\  FST::NODE(1, FST::RELATION('x', 2)),\  FST::NODE(1, FST::RELATION('p', 3)),\  FST::NODE(1, FST::RELATION('o', 4)),\  FST::NODE(1, FST::RELATION('n', 5)),\  FST::NODE(1, FST::RELATION('e', 6)),\  FST::NODE(1, FST::RELATION('n', 7)),\  FST::NODE(1, FST::RELATION('t', 8)),\  FST::NODE()  #define GRAPH\_RANDOMIZE 10, \  FST::NODE(1, FST::RELATION('r', 1)),\  FST::NODE(1, FST::RELATION('a', 2)),\  FST::NODE(1, FST::RELATION('n', 3)),\  FST::NODE(1, FST::RELATION('d', 4)),\  FST::NODE(1, FST::RELATION('o', 5)),\  FST::NODE(1, FST::RELATION('m', 6)),\  FST::NODE(1, FST::RELATION('i', 7)),\  FST::NODE(1, FST::RELATION('z', 8)),\  FST::NODE(1, FST::RELATION('e', 9)),\  FST::NODE |

# **Приложение Г**

Перечень правил, составляющих грамматику языка и описание нетерминальных символов

|  |
| --- |
| Greibach greibach(  NS('S'), TS('$'), // стартовый символ, дно стека  13, // количество правил  Rule(  NS('S'), GRB\_ERROR\_SERIES + 0, // неверная структура программы  5,  Rule::Chain(6, TS('f'), TS('t'), TS('i'), NS('F'), NS('B'), NS('S')),  Rule::Chain(5, TS('p'), TS('i'), NS('F'), NS('U'), NS('S')),  Rule::Chain(4, TS('m'), TS('{'), NS('N'), TS('}')),  Rule::Chain(5, TS('f'), TS('t'), TS('i'), NS('F'), NS('B')),  Rule::Chain(4, TS('p'), TS('i'), NS('F'), NS('U'))  ),  Rule(  NS('F'), GRB\_ERROR\_SERIES + 1,  2,  Rule::Chain(3, TS('['), NS('P'), TS(']')),  Rule::Chain(2, TS('['), TS(']'))  ),  Rule(  NS('P'), GRB\_ERROR\_SERIES + 2, // Ошибка в параметрах функции при её объявлении  2,  Rule::Chain(2, TS('t'), TS('i')),  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('P'))  ),  Rule(  NS('B'), GRB\_ERROR\_SERIES + 3, // Отсутствует тело функции  2,  Rule::Chain(8, TS('{'), NS('N'), TS('r'), TS('['), NS('I'), TS(']'), TS(';'), TS('}')),  Rule::Chain(7, TS('{'), TS('r'), TS('['), NS('I'), TS(']'), TS(';'), TS('}'))  ),  Rule(  NS('I'), GRB\_ERROR\_SERIES + 4, // Недопустимое выражение. Ожидаются только литералы и идентификаторы  4,  Rule::Chain(1, TS('l')),  Rule::Chain(1, TS('l'), NS('I')),  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('i'), NS('I'))  ),  Rule(  NS('U'), GRB\_ERROR\_SERIES + 5, // Ошибка в теле процедуры  1,  Rule::Chain(3, TS('{'), NS('N'), TS('}'))  ),  Rule(  NS('N'), GRB\_ERROR\_SERIES + 6, // Неверная конструкция в теле функции  28,  Rule::Chain(5, TS('c'), TS('t'), TS('i'), TS(';'), NS('N')),  Rule::Chain(7, TS('c'), TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(8, TS('u'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}'), NS('N')),  Rule::Chain(8, TS('w'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}'), NS('N')),  Rule::Chain(12, TS('w'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}'), TS('!'), TS('{'), NS('X'), TS('}'), NS('N')),  Rule::Chain(6, TS('d'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('s'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('k'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('e'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('z'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('o'), TS('['), NS('I'), TS(']'), TS(';'), NS('N')),  Rule::Chain(3, TS('b'), TS(';'), NS('N')),  Rule::Chain(4, TS('i'), NS('K'), TS(';'), NS('N')),  Rule::Chain(4, TS('c'), TS('t'), TS('i'), TS(';')),  Rule::Chain(6, TS('c'), TS('t'), TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(7, TS('u'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}')),  Rule::Chain(7, TS('w'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}')),  Rule::Chain(11, TS('w'), TS('['), NS('R'), TS(']'), TS('{'), NS('X'), TS('}'), TS('!'), TS('{'), NS('X'), TS('}')),  Rule::Chain(3, TS('k'), NS('K'), TS(';')),  Rule::Chain(3, TS('d'), NS('K'), TS(';')),  Rule::Chain(3, TS('s'), NS('K'), TS(';')),  Rule::Chain(3, TS('e'), NS('K'), TS(';')),  Rule::Chain(3, TS('z'), NS('K'), TS(';')),  Rule::Chain(3, TS('o'), NS('K'), TS(';')),  Rule::Chain(2, TS('b'), TS(';')),  Rule::Chain(3, TS('i'), NS('K'), TS(';'))  ),  Rule(  NS('R'), GRB\_ERROR\_SERIES + 7, // Ошибка в условном выражении  4,  Rule::Chain(1, TS('i')),  Rule::Chain(3, TS('i'), TS('g'), TS('i')),  Rule::Chain(3, TS('i'), TS('g'), TS('l')),  Rule::Chain(3, TS('l'), TS('g'), TS('i'))  ),  Rule(  NS('K'), GRB\_ERROR\_SERIES + 8, // Ошибка в вызове функции  2,  Rule::Chain(3, TS('['), NS('W'), TS(']')),  Rule::Chain(2, TS('['), TS(']'))  ),  Rule(  NS('E'), GRB\_ERROR\_SERIES + 9, // ошибка в арифметическом выражении  18,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('('), NS('E'), TS(')')),  Rule::Chain(2, TS('i'), NS('K')),  Rule::Chain(2, TS('d'), NS('K')),  Rule::Chain(2, TS('s'), NS('K')),  Rule::Chain(2, TS('k'), NS('K')),  Rule::Chain(2, TS('e'), NS('K')),  Rule::Chain(2, TS('z'), NS('K')),  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(3, TS('i'), NS('K'), NS('M')),  Rule::Chain(3, TS('d'), NS('K'), NS('M')),  Rule::Chain(3, TS('s'), NS('K'), NS('M')),  Rule::Chain(3, TS('k'), NS('K'), NS('M')),  Rule::Chain(3, TS('e'), NS('K'), NS('M')),  Rule::Chain(3, TS('z'), NS('K'), NS('M'))  ),  Rule(  NS('W'), GRB\_ERROR\_SERIES + 10, // ошибка в параметрах вызываемой функции  4,  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('M'), GRB\_ERROR\_SERIES + 9, // ошибка в арифметическом выражении  10,  Rule::Chain(2, TS('+'), NS('E')),  Rule::Chain(2, TS('-'), NS('E')),  Rule::Chain(2, TS('/'), NS('E')),  Rule::Chain(2, TS('\*'), NS('E')),  Rule::Chain(2, TS('%'), NS('E')),  Rule::Chain(3, TS('+'), NS('E'), NS('M')),  Rule::Chain(3, TS('-'), NS('E'), NS('M')),  Rule::Chain(3, TS('/'), NS('E'), NS('M')),  Rule::Chain(3, TS('\*'), NS('E'), NS('M')),  Rule::Chain(3, TS('%'), NS('E'), NS('M'))  ),  Rule(  NS('X'), GRB\_ERROR\_SERIES + 11, // Неверная конструкция в теле цикла/условного выражения  14,  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(6, TS('d'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('s'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('k'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('e'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('z'), NS('K'), TS(';'), NS('N')),  Rule::Chain(6, TS('o'), TS('['), NS('I'), TS(']'), TS(';'), NS('N')),  Rule::Chain(3, TS('b'), TS(';'), NS('N')),  Rule::Chain(4, TS('i'), NS('K'), TS(';'), NS('N')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(3, TS('k'), NS('K'), TS(';')),  Rule::Chain(3, TS('e'), NS('K'), TS(';')),  Rule::Chain(3, TS('z'), NS('K'), TS(';')),  Rule::Chain(3, TS('d'), NS('K'), TS(';')),  Rule::Chain(3, TS('s'), NS('K'), TS(';')),  Rule::Chain(3, TS('o'), NS('K'), TS(';')),  Rule::Chain(2, TS('b'), TS(';')),  Rule::Chain(3, TS('i'), NS('K'), TS(';'))  )  ); |

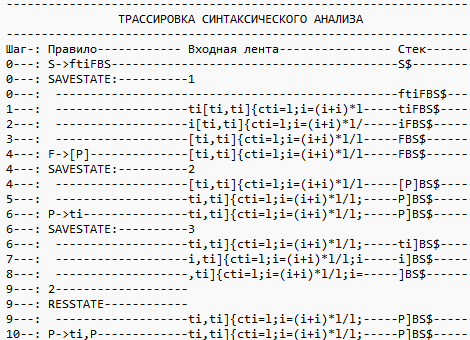
# **Приложение Д**

Основные структуры данных синтаксического анализатора, которые представляются в виде структуры магазинного конечного автомата

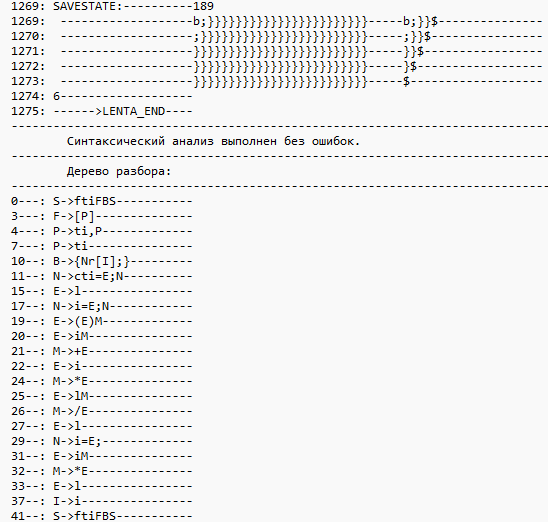
|  |
| --- |
| struct MfstState //состояние автомата для сохранения  {  short lenta\_position; //состояние автомата для сохранения  short nrule; //номер текущего правила  short nrulechain; //номер текущей цепчки, текущего правила  MFSTSTSTACK st; //стек автомата  MfstState();  MfstState(short pposition, MFSTSTSTACK pst, short pnrulechain);  // (позиция на ленте; стек автомата; номер текущей цепочки текущего правила)  MfstState(short pposition, MFSTSTSTACK pst, short pnrule, short pnrulechain);  // (позиция на ленте; стек автомата; номер текущего правила; номер текущей цепочки текущего правила)  };  struct Mfst //магазинный автомат  {  enum RC\_STEP { //код вовзрата функции step  NS\_OK, //найдено правило и цепочка, цепочка записана в стек  NS\_NORULE, //не найдено правило грамматики(ошибка в грамматике)  NS\_NORULECHAIN, //не найдена подходящая цепочка правила(ошибка в исходном коде)  NS\_ERROR, //неизвествный нетерминальный символ грамматики  TS\_OK, //тек. символ ленты == вершине стека, продвинулась лента, pop стека  TS\_NOK, //тек. символ ленты != вершине стека, продвинулась лента, pop стека  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  class my\_stack\_MfstState :public std::stack<MfstState> {  public:  using std::stack<MfstState>::c;  };  GRBALPHABET\* lenta; //перекодированныя (TN/NS) лента (из LEX)  short lenta\_position; //текущая позиция на ленте  short nrule; //номер текущего правила  short nrulechain; //номер текущей цепочки,текущего правила  short lenta\_size; //размер ленты  GRB::Greibach grebach; //грамматика Грейбах  LT::LexTable lex; //результат работы лексического анализатора  MFSTSTSTACK st; //стек автомата  my\_stack\_MfstState storestate; //стек для хранения состояний  Mfst();  Mfst(LT::LexTable& plex, GRB::Greibach pgrebach);  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(Log::LOG log); //сохранить состояние автомата  bool resetstate(Log::LOG log); //восстановить состояние автомата  bool push\_chain(GRB::Rule::Chain chain); //поместить цепочку правила в стек  RC\_STEP step(Log::LOG log); //выполнить шаг автомата  bool start(Log::LOG log); //запустить автомат  bool savediagnosis(RC\_STEP pprc\_step); //код завершения шага  void printrules(Log::LOG log); //вывести послдеовательность правил  struct Deducation  {  short size;  short\* nrules;  short\* nrulechains;  Deducation()  {  size = 0;  nrules = 0;  nrulechains = 0;  };  }deducation; |

# **Приложение Е**

Начало разбора

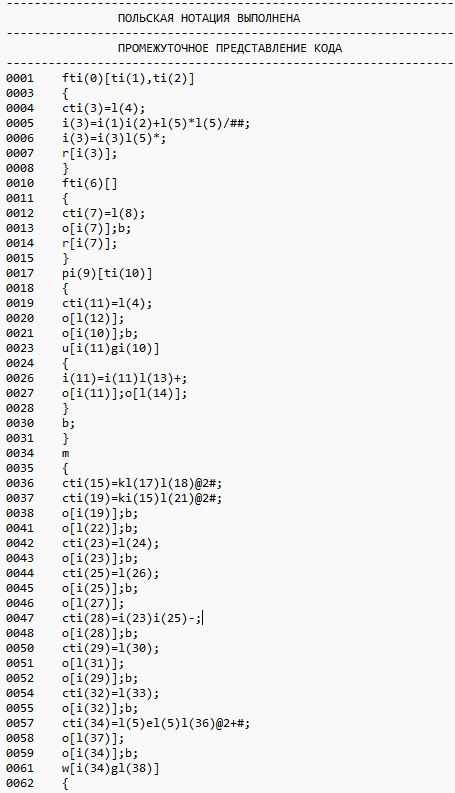


Конец разбора



# **Приложение Ж**

Представление промежуточного кода, отображающее результаты преобразования выражений в польский формат



# **Приложение И**

Сгенерированный код

|  |
| --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib ../Debug/Lib  ExitProcess PROTO :DWORD  EXTRN COMBINE: proc  EXTRN LEXCOMP: proc  EXTRN MEASURE: proc  EXTRN BREAKL: proc  EXTRN RANDOMIZE: proc  EXTRN EXPONENT: proc  EXTRN read: proc  EXTRN OutputInt: proc  EXTRN OutputStr: proc  .stack 4096  .const  L1 SDWORD 0  L2 SDWORD 2  L3 SDWORD 3  L4 BYTE "Counting to length: ", 0  L5 SDWORD 1  L6 BYTE " ", 0  L7 BYTE "Test Course Project ", 0  L8 BYTE "UVV2024 ", 0  L9 BYTE "Start in 3 2 1", 0  L10 BYTE "Numbers: ", 0  L11 SDWORD -2  L12 SDWORD 8  L13 BYTE "bin - oct = ", 0  L14 SDWORD 1  L15 BYTE "Bool: ", 0  L16 BYTE "S", 0  L17 SDWORD 4  L18 BYTE "exp[2,4] = ", 0  L19 SDWORD 16  L20 BYTE "pow 2,4 really equal 16", 0  L21 BYTE "pow 2,4 not really equal 16", 0  L22 BYTE "Now", 0  L23 BYTE "length: ", 0  L24 SDWORD 41  L25 BYTE "Random: ", 0  L26 SDWORD 256  L27 SDWORD 160  L28 BYTE "Lexcomp: ", 0  L29 SDWORD 5  L30 SDWORD 10  L31 BYTE "End of example if <= 10", 0  L32 BYTE "End of example if > 10", 0  L33 BYTE "str1", 0  L34 BYTE "string", 0  L35 BYTE "str1 <= string", 0  L36 BYTE "str1 > string", 0  .data  buffer BYTE 256 dup(0)  funcress SDWORD 0  funcctest SDWORD 0  cycleni SDWORD 0  mainstart DWORD ?  mainstartt DWORD ?  mainbin SDWORD 0  mainoct SDWORD 0  maindif SDWORD 0  mainbooly SDWORD 0  maincharr DWORD ?  mainexp SDWORD 0  mainlength SDWORD 0  mainrandd SDWORD 0  mainkiruha SDWORD 0  maintestx SDWORD 0  maintesty SDWORD 0  maint SDWORD 0  mainresult SDWORD 0  .code  func PROC funca : SDWORD, funcb : SDWORD  push L1  pop funcress  push funca  push funcb  pop eax  pop ebx  add eax, ebx  push eax  push L2  pop eax  pop ebx  mul ebx  push eax  push L2  pop ebx  pop eax  cdq  idiv ebx  push eax  pop funcress  push funcress  push L2  pop eax  pop ebx  mul ebx  push eax  pop funcress  push funcress  jmp local0  local0:  pop eax  ret  func ENDP  funcc PROC  push L3  pop funcctest  push funcctest  call OutputInt  call BREAKL  pop eax  push funcctest  jmp local1  local1:  pop eax  ret  funcc ENDP  cyclen PROC cyclenn : SDWORD  push L1  pop cycleni  push offset L4  call OutputStr  push cyclenn  call OutputInt  call BREAKL  mov eax, cycleni  cmp eax, cyclenn  jl cycle0  jmp cyclenext0  cycle0:  push cycleni  push L5  pop eax  pop ebx  add eax, ebx  push eax  pop cycleni  push cycleni  call OutputInt  push offset L6  call OutputStr  mov eax, cycleni  cmp eax, cyclenn  jl cycle0  cyclenext0:  call BREAKL  ret  cyclen ENDP  main PROC  push offset L7  push offset L8  pop edx  pop edx  push offset L8  push offset L7  push offset buffer  call COMBINE  push eax  pop mainstart  push mainstart  push offset L9  pop edx  pop edx  push offset L9  push mainstart  push offset buffer  call COMBINE  push eax  pop mainstartt  push mainstartt  call OutputStr  call BREAKL  push offset L10  call OutputStr  call BREAKL  push L11  pop mainbin  push mainbin  call OutputInt  call BREAKL  push L12  pop mainoct  push mainoct  call OutputInt  call BREAKL  push offset L13  call OutputStr  push mainbin  push mainoct  pop ebx  pop eax  sub eax, ebx  push eax  pop maindif  push maindif  call OutputInt  call BREAKL  push L14  pop mainbooly  push offset L15  call OutputStr  push mainbooly  call OutputInt  call BREAKL  push offset L16  pop maincharr  push maincharr  call OutputStr  call BREAKL  push L2  push L2  push L17  pop edx  pop edx  push L17  push L2  call EXPONENT;  push eax  pop eax  pop ebx  add eax, ebx  push eax  pop mainexp  push offset L18  call OutputStr  push mainexp  call OutputInt  call BREAKL  mov eax, mainexp  cmp eax, L19  jz m0  jnz m1  je m1  m0:  push offset L20  call OutputStr  call BREAKL  jmp e0  m1:  push offset L21  call OutputStr  call BREAKL  e0:  push offset L22  pop edx  push offset L22  call MEASURE;  push eax  pop mainlength  push offset L23  call OutputStr  push mainlength  call OutputInt  call BREAKL  push L5  push L24  pop edx  pop edx  push L24  push L5  call RANDOMIZE;  push eax  pop mainrandd  push offset L25  call OutputStr  push mainrandd  call OutputInt  call BREAKL  push L26  push L27  pop eax  pop ebx  mul ebx  push eax  pop mainkiruha  push offset L28  call OutputStr  push mainkiruha  call OutputInt  call BREAKL  push func  push L5  push L29  pop edx  pop edx  push L29  push L5  call func  push eax  pop maintestx  push maintestx  call OutputInt  call BREAKL  push funcc  call funcc  push eax  pop maintesty  push maintesty  call OutputInt  call BREAKL  push L17  pop maint  push maint  call cyclen  mov eax, maint  cmp eax, L30  jl m2  jz m2  jg m3  je m3  m2:  push offset L31  call OutputStr  call BREAKL  jmp e1  m3:  push offset L32  call OutputStr  call BREAKL  e1:  push offset L33  push offset L34  pop edx  pop edx  push offset L34  push offset L33  call LEXCOMP;  push eax  pop mainresult  mov eax, mainresult  cmp eax, L1  jl m4  jz m4  jg m5  je m5  m4:  push offset L35  call OutputStr  call BREAKL  jmp e2  m5:  push offset L36  call OutputStr  call BREAKL  e2:  push 0  call ExitProcess  main ENDP  end main |

# **Приложение К**

Описание основных структур данных, используемых для хранения таблиц идентификаторов

|  |
| --- |
| struct Entry // строка таблицы лексем  {  unsigned char lexema; // лексема  int line; // номер строки в исходном тексте  int idxTI; // индекс в таблице идентификаторов или LT\_TI\_NULLIDX  int priority; // приоритет  OPER operation;  };  struct LexTable // экземпляр таблицы лексем  {  int maxsize; // емкость таблицы лексем < LT\_MAXSIZE  int size; // текущий размер таблицы лексем < maxsize  Entry\* table; // массив строк таблицы лексем  };  LexTable Declare( // создать таблицу лексем  int size // емкость таблицы лексем < LT\_MAXSIZE  );  void Add( // добавить строку в таблицу лексем  LexTable& lextable, // экземпляр таблицы лексем  Entry entry // строка таблицы лексем  );  Entry GetEntry( // получить строку таблицы лексем  LexTable& lextable, // экземпляр таблицы лексем  int n // номер получаемой строки  );  void Delete(LexTable& lextable); // удалить таблицу лексем (освободить память)  Entry writeEntry( // заполнить строку таблицы лексем  Entry& entry,  unsigned char lexema,  int indx,  int line,  int priority = -1,  OPER operation = NOT  ); |