МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора VYR-2023»

Выполнил студент Вовна Ярослава Руслановна

(Ф.И.О.)

Руководитель проекта acc. Мущук Артур Николаевич

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

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

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

Консультант acc. Мущук Артур Николаевич

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

Нормоконтролер асс. Мущук Артур Николаевич

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

Курсовой проект защищен с оценкой

Минск 2023

Содержание

[Введение 2](#_Toc152770236)

[Глава 1 Спецификация языка программирования 3](#_Toc152770237)

[1.1 Характеристика языка программирования 3](#_Toc152770238)

[1.2 Определение алфавита языка программирования 3](#_Toc152770239)

[1.3 Применяемые сепараторы 3](#_Toc152770240)

[1.4 Применяемые кодировки 4](#_Toc152770241)

[1.5 Типы данных 4](#_Toc152770242)

[1.6 Преобразование типов данных 5](#_Toc152770243)

[1.7 Идентификаторы 6](#_Toc152770244)

[1.8 Литералы 6](#_Toc152770245)

[1.9 Объявление данных 7](#_Toc152770246)

[1.10 Инициализация данных 7](#_Toc152770247)

[1.11 Инструкции языка 7](#_Toc152770248)

[1.12 Операции языка 8](#_Toc152770249)

[1.13 Выражения и их вычисления 8](#_Toc152770250)

[1.14 Конструкции языка 9](#_Toc152770251)

[1.15 Область видимости идентификаторов 10](#_Toc152770252)

[1.16 Семантические проверки 10](#_Toc152770253)

[1.17 Распределение оперативной памяти на этапе выполнения 10](#_Toc152770254)

[1.18 Стандартная библиотека и ее состав 11](#_Toc152770255)

[1.19 Ввод и вывод данных 11](#_Toc152770256)

[1.20 Точка входа 11](#_Toc152770257)

[1.21 Препроцессор 11](#_Toc152770258)

[1.22 Соглашения о вызове 12](#_Toc152770259)

[1.23 Объектный код 12](#_Toc152770260)

[1.24 Классификация сообщений транслятора 12](#_Toc152770261)

[1.25 Контрольный пример 12](#_Toc152770262)

[2. Структура транслятора 13](#_Toc152770263)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 13](#_Toc152770264)

[2.2 Перечень входных параметров транслятора 14](#_Toc152770265)

[2.3 Протоколы, формируемые транслятором и их содержимое 14](#_Toc152770266)

[3 Разработка лексического анализатора 16](#_Toc152770268)

[3.1 Структура лексического анализатора 16](#_Toc152770269)

[3.2 Входные и выходные данные лексического анализатор 17](#_Toc152770270)

[3.3 Параметры лексического анализатора 17](#_Toc152770271)

[3.4 Алгоритм лексического анализа 17](#_Toc152770272)

[3.5 Контроль входных символов 17](#_Toc152770273)

[3.6 Удаление избыточных символов 18](#_Toc152770274)

[3.7 Перечень ключевых слов 19](#_Toc152770275)

[3.8 Основные структуры данных 20](#_Toc152770276)

[3.9 Структура и перечень сообщений лексического анализатора 21](#_Toc152770277)

[3.10 Принцип обработки ошибок 22](#_Toc152770278)

[3.11 Контрольный пример 22](#_Toc152770279)

[4. Разработка синтаксического анализатора 23](#_Toc152770280)

[4.1 Структура синтаксического анализатора 23](#_Toc152770281)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 23](#_Toc152770282)

[4.3 Построение конечного магазинного автомата 25](#_Toc152770283)

[4.4 Основные структуры данных 26](#_Toc152770284)

[4.5 Описание алгоритма синтаксического разбора 26](#_Toc152770285)

[4.6 Параметры синтаксического анализатора и режимы его работы 27](#_Toc152770286)

[4.7 Структура и перечень сообщений синтаксического анализатора 27](#_Toc152770287)

[4.8 Принцип обработки ошибок 27](#_Toc152770288)

[4.9 Контрольный пример 28](#_Toc152770289)

[5 Разработка семантического анализатора 29](#_Toc152770290)

[5.1 Структура семантического анализатора 29](#_Toc152770291)

[5.2 Функции семантического анализатора 29](#_Toc152770292)

[5.3 Структура и перечень сообщений семантического анализатора 29](#_Toc152770293)

[5.4 Принцип обработки ошибок 30](#_Toc152770294)

[5.5 Контрольный пример 30](#_Toc152770295)

[6. Вычисление выражений 31](#_Toc152770296)

[6.1 Выражения, допускаемые языком 31](#_Toc152770297)

[6.2 Польская запись и принцип её построения 31](#_Toc152770298)

[6.3 Программная реализация обработки выражений 31](#_Toc152770299)

[6.4 Контрольный пример 31](#_Toc152770300)

[7. Генерация кода 33](#_Toc152770301)

[7.1 Структура генератора кода 33](#_Toc152770302)

[7.2 Представление типов данных в оперативной памяти 33](#_Toc152770303)

[7.3 Статическая библиотека 34](#_Toc152770304)

[7.4 Особенности алгоритма генерации кода 34](#_Toc152770305)

[7.5 Параметры, управляющие генерацией кода 35](#_Toc152770306)

[7.6 Контрольный пример 35](#_Toc152770307)

[8. Тестирование транслятора 36](#_Toc152770308)

[8.1 Общие положения 36](#_Toc152770309)

[8.2 Результаты тестирования 36](#_Toc152770310)

[9. Разработка и тестирование интерпретатора 39](#_Toc152770311)

[Заключение 40](#_Toc152770312)

[Список использованных источников 41](#_Toc152770313)

**Введение**

Целью курсового проекта является создание своего языка программирования VYR-2023 и реализация компилятора. Написание транслятора будет осуществляться на языке C++, при этом код на языке VYR-2023 будет транслироваться в язык ассемблера.

Компилятор VYR-2023 состоит из следующих частей:

* лексический анализатор;
* синтаксический анализатор;
* семантический анализатор;
* генератор исходного кода на языке ассемблера.

На основе поставленной цели курсового проекта, были выявлены следующие задачи:

1. Разработка спецификации языка VYR-2023;
2. Разработка лексического анализатора;
3. Разработка синтаксического анализатора;
4. Разработка семантического анализатора;
5. Разбор арифметических выражений;
6. Разработка генератора кода;
7. Тестирование транслятора.

# Глава 1 Спецификация языка программирования

## **1.1 Характеристика языка программирования**

Язык программирования VYR-2023 является процедурным, универсальным строго типизированным, не объектно-ориентированным, компилируемым.

Поддерживает 2 типа данных: целочисленный (integer) и строковый (string). В стандартной библиотеке языка программирования доступны две функции: одна для лексикографического сравнения строк comp(string x, string y), а вторая для определения длины строки len(string str).

**1.2 Определение алфавита языка программирования**

При написании программы на языке VYR-2023 используется таблица символов Windows-1251, представленная в рисунке 1.1.

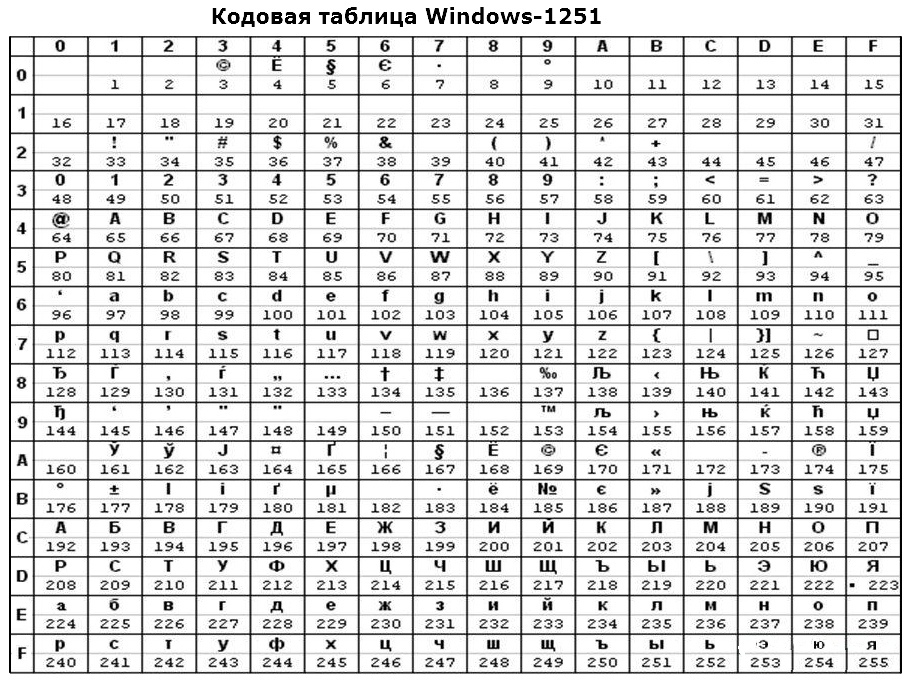


Рисунок 1.1 – Таблица кодировки Windows-1251

Исходный код VYR-2023 может содержать символы-сепараторы, символы операций, символы латинского алфавита малого регистра, цифры десятичной системы счисления от 0 до 9, символы кириллицы [а…я], [А…Я] разрешены только в строковых литералах.

В качестве сепараторов и специальных символов используются: [ ] ( ) , ; : + - / \* > < ! ~ “пробел”.

Для записи инструкций языка используются символы: [a…z].

**1.3 Применяемые сепараторы**

Для того, чтобы разделить операции языка, необходимо использовать сепараторы. Используемые в языке программирования VYR-2023 сепараторы приведены в таблице 1.1.

Таблица 1.1. – Сепараторы

|  |  |
| --- | --- |
| Символ(ы) | Назначение |
| ‘пробел’ | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |
| **{** … **}** | Блок функции или условной конструкции |
| **(** … **)** | Блок фактических или формальных параметров функции, а также приоритет арифметических операций |
| **,** | Разделитель параметров функций |
| **+ - \* / %** | Арифметические операции |
| **< > ! ~** | Операции сравнения |
| **;** | Разделитель программных конструкций |
| **=** | Оператор присваивания |

Следует заметить, что эти символы имеют важную функцию и каждый выполняет определённую роль в языке программирования.

**1.4 Применяемые кодировки**

При трансляции исходного кода применяется кодировка Windows-1251. Более подробное описание кодировки представлено в пункте 1.2.

## **1.5 Типы данных**

Тип данных — это понятие, которое определяет набор значений, которые может принимать переменная или выражение, а также операции, которые могут быть выполнены над этими значениями. Тип данных определяет, как компьютер интерпретирует и обрабатывает информацию.

В языке VYR-2023 реализованы два фундаментальных типа данных: целочисленный и строковый. Описание типов приведено в таблице 1.2.

Таблица 1.2 – Типы данных языка VYR-2023

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Целочисленный тип данных **integer** | Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 4 байта.  Инициализация по умолчанию: значение 0.  Поддерживаемые операции:  **+** (бинарный) – оператор сложения; |

Продолжение таблицы 1.2

|  |  |
| --- | --- |
|  | - (бинарный) – оператор вычитания;  **\*** (бинарный) – оператор умножения;  **/** (бинарный) – оператор деления;  **%** (бинарный) – оператор определения остатка от деления;  **=** (бинарный) – оператор присваивания.  В качестве условия условного оператора поддерживаются следующие логические операции:  **>** (бинарный) – оператор «больше»;  **<** (бинарный) – оператор «меньше»; |
| Строковый тип данных **string** | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 255. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций. |

Строковый тип данных имеет размер в один байт на символ. Целочисленный тип имеет размер в четыре байта и поддерживает математические операции и операторы сравнения. Задание значения возможно как в десятеричной, так и в шестнадцатеричной системах счисления.

**1.6 Преобразование типов данных**

В языке программирования VYR-2023 преобразование типов данных не предусмотрено.

**1.7 Идентификаторы**

Общее количество идентификаторов ограничено максимальным размером таблицы идентификаторов. Идентификаторы должны содержать только символы нижнего регистра латинского алфавита. Максимальная длина идентификатора равна десяти символам. Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Префикс занимает 5 дополнительных символов. В случае превышения заданной длины, идентификаторы усекаются до длины, равной 10 символов (5 символов на имя идентификатора, 5 символов на префикс). Данные правила действуют для всех типов идентификаторов. Зарезервированные идентификаторы не предусмотрены. Идентификаторы не должны совпадать с ключевыми словами. Типы идентификаторов: имя переменной, имя функции, параметр функции. Имена идентификаторов-функций и имена идентификаторов-переменных не должны совпадать с именами команд ассемблера.

**1.8 Литералы**

С помощью литералов осуществляется инициализация переменных. Все литералы являются **rvalue**. Типы литералов языка VYR-2023 представлены в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Литералы | Пояснение |
| Целочисленные литералы в десятичном представлении | Может состоять из чисел [0-9]. Минимальное значение равно -2147483647, максимальное 2147483647. При выходе за пределы значение будет укорочено до максимального значения. |
| Строковые литералы | Набор символов (от 1 до 255), заключённых в двойные кавычки |
| Целочисленные литералы в шестнадцатеричном представлении | Может состоять из чисел [0-9], а также символов [A-F]. Начинается с суффикса 16x. Минимальное значение равно -16x7FFFFFFF, максимальное 16x7FFFFFFF. Не содержит дробную часть. При выходе за пределы значение будет укорочено до максимального значения. |

Ограничения на строковые литералы языка VYR-2023: внутри литерала не допускается использование одинарных и двойных кавычек.

## **1.9 Объявление данных**

Для объявления переменной указывается тип данных и имя идентификатора. Допускается инициализация при объявлении.

Пример объявления числового типа с инициализацией:

**integer** num1 = -1

**integer** num2 = 16xFFF

Пример объявления переменной символьного типа с инициализацией:

**stroka** str1= “hello world”

Для объявления функций и процедур используется ключевое слово **function**, перед которым указывается тип функции. Далее обязателен список параметров и тело функции.

## **1.10 Инициализация данных**

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства. При объявлении без инициализации предусмотрено значение 0 для типа **integer** по умолчанию.

Способы инициализации переменных языка программирования VYR-2023 представлены в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

|  |  |
| --- | --- |
| Вид инициализации | Примечание |
| <тип данных> <идентификатор>; | Автоматическая инициализация переменной integer нулем, string – пустой строкой. |
| <тип данных> <идентификатор> = <значение>; | Инициализация переменной с присваиванием значения. |

Объектами-инициализаторами могут быть только идентификаторы или литералы.

## **1.11 Инструкции языка**

Инструкции языка VYR-2023 представлены в таблице 1.5.

Таблица 1.5 – ИнструкцииязыкаVYR-2023

|  |  |
| --- | --- |
| Инструкция | Реализация |
| Объявление переменной | <тип данных> <идентификатор>; |
| Объявление переменной с явной инициализацией | <тип данных> <идентификатор> = <значение>;  Значение – инициализатор конкретного типа. Может быть только литералом или идентификатором |
| Возврат из функции или процедуры | Для функций, возвращающих значение:  **return** <идентификатор/литерал>;  Для процедур:  **return;** |
| Вывод данных | **print** <идентификатор/литерал>; |
| Вывод данных с переходом на новую строку | **println** <идентификатор/литерал>; |
| Вызов функции или процедуры | <идентификатор функции> (<список параметров>);  Список параметров может быть пустым. |
| Присваивание | <идентификатор> **=** <выражение>;  Выражением может быть идентификатор, литерал, или вызов функции соответствующего типа. Для |

Продолжение таблицы 1.5

|  |  |
| --- | --- |
|  | целочисленного типа выражение может быть дополнено арифметическими операциями с любым количеством операндов с использованием скобок. Для строкового типа выражение может быть только идентификатором, литералом или вызовом функции, возвращающей значение строкового типа. |
| Условный оператор | if: <условие>  {…}  ^{…} |

Все эти инструкции совместно используются для построения программ на языке VYR-2023 и обеспечивают выполнение требуемых действий в программе.

**1.12 Операции языка**

В языке VYR-2023 предусмотрены следующие операции с данными. Приоритетность операции умножения выше приоритета операций сложения и вычитания. Операции языка представлены в таблице 1.6.

Таблица 1.6 – Операции языка VYR-2023

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические | 1. **+ –** сложение 2. - – вычитание 3. \* – умножение 4. **/** – деление без остатка 5. **%** – остаток от деления 6. = – присваивание |
| Строковые | 1. **=**  – присваивание |
| Логические | 1. **>** – больше  2. **<** – меньше  3. ~ – равно  4. ! – неравно |

Для установки наивысшего приоритета используются круглые скобки.

**1.13 Выражения и их вычисления**

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

1. Допускается использовать скобки для смены приоритета операций;
2. Выражение записывается в строку без переносов;
3. Использование двух подряд идущих операторов не допускается;
4. 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 VYR-2023 оформляется в виде функций пользователя и главной функции. Программные конструкции языка представлены в таблице 1.7.

Таблица 1.7 – Программные конструкции языка VYR-2023

|  |  |  |
| --- | --- | --- |
| Конструкция | | Реализация |
| Главная функция | | **main**  **{**  …  **}** |
| Внешняя функция | | <тип данных> **function** <идентификатор> **(**<тип> <идентификатор>, ...**)**  **{**…  **return** <идентификатор/литерал>;  **}** |
| Внешняя процедура | **void function** <идентификатор> **(** <тип> <идентификатор>, ...**)**  **{**…  **return**;  **}** | |
| Условная конструкция | **if:** <идентификатор1> <оператор> <идентификатор2>  **блок1{**…**}**  **^блок2{**…**}**  <идентификатор1>, <идентификатор2> - идентификаторы или литералы целочисленного типа (но не два литерала одновременно). <оператор> - один из операторов сравнения (**> <**), устанавливающий отношение между двумя операндами и организующий условие данной конструкции. При истинности условия выполняется код внутри блока1, иначе – код внутри блока **блок2**. | |

При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода.

## **1.15 Область видимости идентификаторов**

Область видимости: сверху вниз. 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 VYR-2023 выполняются следующие семантические проверки:

1. Наличие функции **main** – точки входа в программу;
2. Единственность точки входа;
3. Переопределение идентификаторов;
4. Использование идентификаторов без их объявления;
5. Проверка соответствия типа функции и возвращаемого параметра;
6. Правильность передаваемых в функцию параметров: количество, типы;
7. Правильность строковых выражений;
8. Превышение размера строковых и числовых литералов;
9. Правильность составленного условия условного оператора;
10. Запрет деления на ноль.

**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 ее состав**

В языке VYR-2023 присутствует стандартная библиотека, которая подключается автоматически при трансляции исходного кода в язык ассемблера. Стандартные библиотеки подключены по умолчанию в программу. Вызовы функций доступны в месте вызова пользовательских функций. Содержимое библиотеки и описание функций представлено в таблице 1.8.

Таблица 1.8 – Стандартная библиотека языка VYR-2023

|  |  |
| --- | --- |
| Функция | Описание |
| **integer** len(**string** str1); | Строковая функция, высчитывает длину строки. |
| **integer** comp(**string** str1, str2); | Сравнивает две строки и возвращает 1, если строки равны и 0, если строки не равны. |

Стандартная библиотека написана на языке С++, подключается к транслированному коду на этапе генерации кода.

## **1.19 Ввод и вывод данных**

Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю. Для вывода предусмотрены операторы **print** и **println**. Эти функции представлены в таблице 1.9.

Таблица 1.9 – Дополнительные функции стандартной библиотеки

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void outn(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| void outnf(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала с переходом на новую строку. |
| void outw(char\* value) | Функция для вывода в стандартный поток значения строкового идентификатора/литерала. |
| void outwt(char\* value) | Функция для вывода в стандартный поток значения строкового идентификатора/литерала с переходом на новую строку. |

Для реализации вывода с переходом на новую строку были созданы копии функций с некоторым изменением.

## **1.20 Точка входа**

В языке VYR-2023 каждая программа должна содержать главную функцию (точку входа) **main**, с первой инструкции которой начнётся последовательное выполнение команд программы.

**1.21 Препроцессор**

В языке VYR-2023 препроцессор не предусматривается.

## **1.22 Соглашения о вызове**

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 в язык ассемблера.

**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 1.10, а также в приложении А.

Таблица 1.10 – Классификация ошибок

|  |  |
| --- | --- |
| Номера ошибок | Характеристика |
| 0 – 200 | Системные ошибки |
| 200 – 299 | Ошибки лексического анализа |
| 300 – 399 | Ошибки семантического анализа |
| 600 – 699 | Ошибки синтаксического анализа |

В таблице 1.10 представлены ошибок, которые могут возникнуть при анализе исходного кода программы на языке программирования VYR-2023.

**1.25 Контрольный пример**

Контрольный пример демонстрирует главные особенности языка VYR-2023: 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 VYR-2023 исходный код проходит трансляцию в язык Assembler. Транслятор состоит из нескольких компонентов, которые взаимодействуют между собой и выполняют свои специфические функции, описанные в разделе 2.1. Чтобы получить ассемблерный код, требуется использовать выходные данные, сгенерированные лексическим анализатором. Эти данные включают таблицу лексем и таблицу идентификаторов. Для указания выходных файлов, которые будут сгенерированы, используются определенные входные параметры, описанные в таблице. 2.1.

Структура транслятора языка VYR-2023 приведена на рисунке 2.1.

|  |
| --- |
|  |

Рисунок 2.1 – Структура транслятора языка программирования VYR-2023

Первый этап компиляции называется лексическим анализом, который выполняется с помощью лексического анализатора или сканера. Лексический анализатор принимает последовательность символов исходного языка в качестве входных данных и осуществляет предварительный анализ текста. Он разбивает эту последовательность символов на отдельные слова или "токены". Примеры таких "токенов" включают идентификаторы, числа, операторы, ключевые слова и другие лексические единицы. Каждому токену присваивается тип, и он записывается в таблицу лексем. Кроме того, информация о каждом идентификаторе записывается в таблицу идентификаторов, где хранится дополнительная информация о каждом идентификаторе. Таблица лексем и таблица идентификаторов являются входными данными для следующего этапа компиляции - синтаксического анализа или "парсера".

Синтаксический анализатор - это часть компилятора, которая выполняет синтаксический анализ или проверку соответствия исходного кода грамматическим правилам. Входными данными для синтаксического анализатора являются таблица лексем и таблица идентификаторов. Результатом работы синтаксического анализатора является дерево разбора.

Семантический анализатор - это часть транслятора, которая выполняет семантический анализ или проверку исходного кода на наличие ошибок, которые невозможно обнаружить при помощи грамматики. Входными данными для семантического анализатора являются таблица лексем и таблица идентификаторов.

Генератор кода - это часть транслятора, которая создает ассемблерный код на основе данных, полученных на предыдущих этапах компиляции. Входными данными для генератора кода являются таблица лексем и таблица идентификаторов. Генератор кода использует эти данные для создания файла с ассемблерным кодом.

**2.2 Перечень входных 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 VYR-2023

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке VYR-2023, имеющий расширение .txt | Не предусмотрено |
| -log:<путь к log-файлу> | Файл журнала для вывода протоколов работы программы. | Значение по умолчанию:  <имя in-файла>.log |
| -out:<путь к out-файлу> | Файл с преобразованным кодом программы | Значение по умолчанию:  <имя in-файла>.out |

Таблица 2.1 содержит параметры, которые принимает транслятор языка VYR-2023 в качестве входных данных для создания файлов, содержащих результаты работы лексического, синтаксического и семантического анализаторов.

**2.3 Протоколы, формируемые транслятором и их содержимое**

В ходе работы программы формируются протоколы работы лексического, синтаксического и семантического анализаторов, которые содержат в себе перечень протоколов работы. С использованием указанных входных параметров, программист имеет возможность настроить транслятор VYR-2023 таким образом, чтобы создавать соответствующие файлы с результатами работы различных анализаторов и сгенерированным программным кодом. Это обеспечивает удобство отслеживания и анализа процесса трансляции и отладки программ. В таблице 2.2 приведены протоколы, формируемые транслятором и их содержимое.

Таблица 2.2 – Протоколы, формируемые транслятором языка VYR-2023

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования VYR-2023. Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа, а также результат работы алгоритма преобразования выражений к польской записи. |
| Выходной файл, c расширением ".asm" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

При возникновении ошибки, компилятор прекратит свою работу, а информация об ошибке будет записана в log-файл. Трансляция на ассемблер не будет выполнена в таком случае.

**3 Разработка лексического анализатора**

**3.1 Структура лексического анализатора**

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся исходный код входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка, согласно п. 1.1 стандарта [1]. Лексический анализатор производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов.

Структурная схема лексического анализатора языка VYR-2023 представлена на рисунке 3.1.



Рисунок 3.1 – Структурная схема лексического анализатора

Примеры лексических единиц включают идентификаторы, числа, символы операций и ключевые слова, и так далее. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением - лексемами. Это создает промежуточное представление исходной программы. Каждой лексеме присваивается тип, и она записывается в таблицу идентификаторов, где хранится дополнительная информация о каждой лексеме.

Функции лексического анализатора:

* удаление «пустых» символов и комментариев. Если «пустые» символы

(пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними (альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации);

* распознавание идентификаторов и ключевых слов;
* распознавание констант;
* распознавание разделителей и знаков операций.

Исходный код программы представлен в приложении А.

## **3.2 Входные и выходные данные лексического анализатор**

Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. Лексический анализатор получает исходный код программы или текстовый файл в качестве входных данных. Он сканирует этот код, выделяет лексические единицы, такие как ключевые слова, операторы, идентификаторы, числа и другие символы.

Затем анализатор формирует таблицы лексем и идентификаторов. В таблице лексем содержится информация о каждой лексеме, включая ее тип, позицию в исходном коде и другие подробности. Таблица идентификаторов содержит информацию о каждом обнаруженном идентификаторе, такую как его имя, тип и значение. Кроме того, может быть создана таблица сообщений, где записываются обнаруженные ошибки или предупреждения.

Полученные таблицы могут быть использованы для дальнейшего анализа и обработки исходного кода.

## **3.3 Параметры лексического анализатора**

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся в файл-протокол.

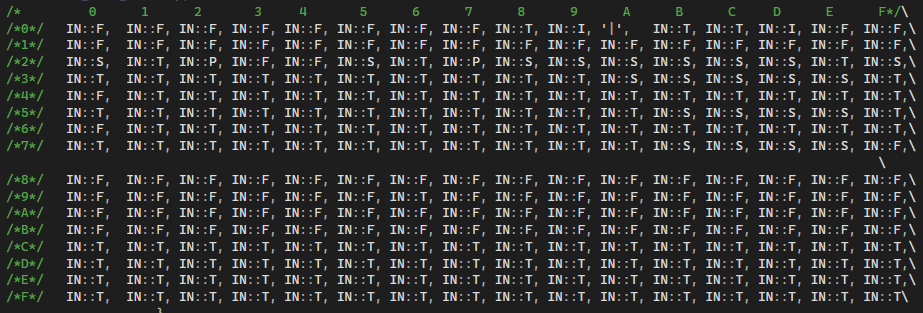
## **3.4 Алгоритм лексического анализа**

1. Проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы.
2. Для выделенной части входного потока выполняется функция распознавания лексемы.
3. При успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу.
4. Формирует протокол работы.
5. При неуспешном распознавании выдается сообщение об ошибке.

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов.

## **3.5. Контроль входных символов**

Таблица контроля входных символов представлена на рисунке 3.2.



Категории входных символов представлены в таблице 3.1.

## **3.6 Удаление избыточных символов**

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

1. Посимвольно считываем файл с исходным кодом программы;

2. Встреча пробела или знака табуляции является своего рода встречей символа-сепаратора;

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

## **3.7 Перечень ключевых слов**

Лексический анализатор преобразует исходный текст, заменяя лексические единицы лексемами для создания промежуточного представления исходной программы. Соответствие токенов и лексем приведено в таблице 3.2.

Таблица 3.2 – Соответствие токенов и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| integer, string | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 10 символов. |
| Литерал | l | Литерал любого доступного типа. |
| Шестнадцатеричный литерал | 16x | Литерал в шестнадцатеричном представлении. |
| function | f | Объявление функции. |
| return | r | Выход из функции/процедуры. |
| main | m | Главная функция. |
| if: | w | Указывает начало цикла/условного оператора. |
| ^ | ^ | Разделение конструкций в цикле/условном операторе |
| ; | ; | Разделение выражений |
| , | , | Разделение параметров функций |
| { | { | Начало блока/тела функции |
| } | } | Закрытие блока/тела функции |
| ( | ( | Передача параметров в функцию, приоритет операций |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций |
| = | = | Знак присваивания |

Продолжение таблицы 3.2

|  |  |  |
| --- | --- | --- |
| +  -  \*  /  % | +  -  \*  /  % | Знаки операций |
| >  <  !  ~ | >  <  !  ~ | Знаки логических операторов |

Пример реализации таблицы лексем представлен в приложении Б.

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

## **3.8 Основные структуры данных**

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде (sn), и номер в таблице идентификаторов, если лексема является идентификаторов (idxTI). Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value). Код C++ со структурой таблицы идентификаторов представлен в Листинге 3.1.

Листинг 3.1

struct Entry // строка таблицы идентификаторов

{

int idxfirstLE;

unsigned char id[ID\_MAXSIZE];

IDDATATYPE iddatatype = NUL;

IDTYPE idtype;

int parm = 0;

int nums = 0; //0 - 10 СС 1 - 16 СС

union

{

int vint;

struct

{} vstr;

} value;};

Каждый символ проходит по данному алгоритму магазинного автомата.

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

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. Если в процессе анализа находятся более трёх ошибок, то анализ останавливается.

## **3.10 Принцип обработки ошибок**

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. Перечень сообщений представлен в Листинге 3.2.

Листинг 3.2

ERROR\_ENTRY(200, "Лексический анализатор: Неинициализированная переменная"),

ERROR\_ENTRY(201, "Лексический анализатор: Неверное имя идентификатора"),

ERROR\_ENTRY(202, "Лексический анализатор: Превышен допустимый размер литерала"),

Благодаря такой обработке достаточно просто понять, с чем связана ошибка.

**3.11 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. Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

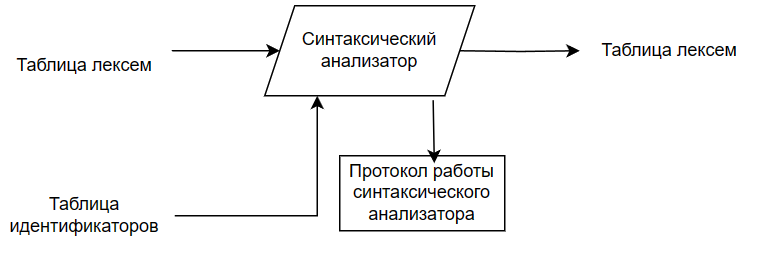


Рисунок 4.1 – Структура синтаксического анализатора

Результатом работы синтаксического анализатора является дерево разбора, которое представляет собой иерархическую структуру программы, отражающую их вложенность и последовательность операций.

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

В синтаксическом анализаторе транслятора языка VYR-2023 используется контекстно-свободная грамматика , где

T – множество терминальных символов,

N – множество нетерминальных символов,

P – множество правил языка,

S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, т.к. она не леворекурсивная (не содержит леворекурсивных правил). Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 – Таблица правил переходов нетерминальных символов

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| S | S->m{N}  S->tfi()TS  S->tfi(F)TS | Стартовые правила, описывающее общую структуру программы |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
|  | S->nfi()TS  S->nfi(F)TS  S->m{N} |  |
| T | T->{N}  T->{Nr;}  T->{NrE;}  T->{rE;} | Правила для тела функций |
| N | N->ti;N  N->i(W);N  N->i(W);  N->i();N  N->i();  N->ti=E;N  N->ti=E;  N->i=E;  N->i=E;N  N->pi;N  N->zi;N  N->pl;N  N->zl;N  N->pE;N  N->zE;N  N->pE;  N->zE;  N->ti;  N->pi;  N->zi;  N->pl;  N->zl;  N->wI|X  N->wI|XN | Правила набора операций |
| B | B->ti=E;  B->i=E;  B->pE;  B->zE;  B->ti;  B->pi;  B->zi; | Правила выражений |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
|  | B->pl;  B->wI|X  B->wI|XN |  |
| X | X->{N}^{N}  X->{N} | Правила для условного оператора |
| O | O->{N} | Правила для цикла |
| E | E->i;  E->l;  E->(E)  E->i(W)  E->i()  E->iM  E->lM  E->(E)M  E->i(W)M  E->i()M | Правила для выражений |
| F | F->ti  F->ti,F | Правила для передаваемых параметров |
| W | W->i  W->l  W->I,W  W->l,W | Правила для вызываемых параметров |
| M | M->vE  M->vEM | Правила для арифметических операторов |
| I | I->:EqE  I->:EqE&I | Правила для тела условного оператора |

В таблице 4.1 представлено описание нетерминальных символов и соответствующих правил переходов в контекстно-свободной грамматике языка VYR-2023.

**4.3 Построение конечного магазинного автомата**

Конечный автомат с магазинной памятью представляет собой семерку. Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $) |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
|  | Начальное состояние магазина автомата | Символ маркера дна стека $ |
|  | Множество конечных состояний | Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты |

Исходя из представленной таблицы, можно сделать вывод о том, что магазинный автомат применяется для анализа и обработки языка VYR-2023 с использованием контекстно-свободной грамматики.

**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 VYR-2023. Данные структуры 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 Параметры синтаксического анализатора и режимы его работы**

Входной информацией для синтаксического анализатора является таблица лексем и идентификаторов. Кроме того используется описание грамматики в форме Грейбах. Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

**4.7 Структура и перечень сообщений синтаксического анализатора**

Перечень сообщений синтаксического анализатора представлен в Листинге 4.1.

Листинг 4.1

ERROR\_ENTRY(600, "Синтаксическая ошибка: Неверная структура программы"),

ERROR\_ENTRY(601, "Синтаксическая ошибка: Ошибка в теле функции"),

ERROR\_ENTRY(602, "Синтаксическая ошибка: Ошибка в выражении"),

ERROR\_ENTRY(603, "Синтаксическая ошибка: Ошибка в параметрах функции"),

ERROR\_ENTRY(604, "Синтаксическая ошибка: Ошибка в параметрах вызываемой функции"),

ERROR\_ENTRY\_NODEF(605),

ERROR\_ENTRY(606, "Синтаксическая ошибка: Неверная структура условия"),

ERROR\_ENTRY(607, "Синтаксическая ошибка: Неверная конструкция цикла")

Благодаря такой обработке достаточно просто понять, с чем связана ошибка.

**4.8 Принцип обработки ошибок**

Синтаксический анализатор осуществляет разбор исходной последовательности лексем до тех пор, пока не достигнет конца последовательности лексем или не обнаружит ошибку. В случае обнаружения ошибки процесс анализа прекращается, и выводится сообщение об ошибке. Если в процессе анализа обнаруживается ошибка, процесс также останавливается, и выводится соответствующее сообщение об ошибке.

**4.9 Контрольный пример**

Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью приведены в приложении В.

**5 Разработка семантического анализатора**

**5.1 Структура семантического анализатора**

Семантический анализатор принимает результаты работы лексического и синтаксического анализаторов в качестве входных данных: таблицы лексем, таблицы идентификаторов и дерево разбора, полученное в результате синтаксического анализа. Общая структура семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1 – Структура семантического анализатора

Семантический анализатор последовательно выполняет различные проверки для обнаружения возможных ошибок. Однако некоторые проверки, такие как проверка уникальности точки входа или предварительное объявление переменных, могут быть выполнены уже на этапе лексического анализа.

**5.2 Функции семантического анализатора**

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

**5.3 Структура и перечень сообщений семантического анализатора**

Перечень сообщений семантического анализатора представлены в Листинге 5.1.

Листинг 5.1

ERROR\_ENTRY(313, "Семантическая ошибка: Недопустимый размер целочисленного литерала"),

ERROR\_ENTRY(314, "Семантическая ошибка: Типы данных в выражении не совпадают"),

ERROR\_ENTRY(315, "Семантическая ошибка: Тип функции и возвращаемого значения не совпадают"),

ERROR\_ENTRY(316, "Семантическая ошибка: Недопустимое строковое выражение справа от знака \'=\'"),

ERROR\_ENTRY(317, "Семантическая ошибка: Неверное условное выражение"),

ERROR\_ENTRY(318, "Семантическая ошибка: Попытка деления на ноль"),

Благодаря такой обработке достаточно просто понять, с чем связана ошибка.

**5.4 Принцип обработки ошибок**

В процессе трансляции программы ошибки, которые возникают, регистрируются в протоколе, который определяется входными параметрами. Если возникают ошибки, они записываются в протокол с указанием номера ошибки и диагностического сообщения. Анализ прекращается после обнаружения всех ошибок.

## **5.5 Контрольный пример**

Соответствие примеров некоторых ошибок в исходном коде и диагностических сообщений об ошибках приведено в таблице 5.1.

Таблица 5.1. – Примеры диагностики ошибок

|  |  |
| --- | --- |
| main{  integer x = 9;  string y = x;  } | Ошибка №314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 3 |
| main{  integer x = 9;  }  main{  string y = "qwerty";  } | Ошибка №302: Семантическая ошибка: Обнаружено несколько точек входа main Строка: 0 |

Ошибки записываются в протокол с указанием номера ошибки и диагностического сообщения. Анализ программы продолжается до обнаружения всех ошибок, после чего процесс анализа останавливается.

**6. Вычисление выражений**

**6.1 Выражения, допускаемые языком**

В языке VYR-2023 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. – Приоритеты операций

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

Операторы в скобках имеют наивысший приоритет, за ними следуют умножение и деление, а затем сложение и вычитание.

**6.2 Польская запись и принцип её построения**

Все выражения языка VYR-2023 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 польской записи:

1. исходная строка: выражение;
2. результирующая строка: польская запись;
3. стек: пустой;
4. исходная строка просматривается слева направо;
5. операнды переносятся в результирующую строку;
6. операция записывается в стек, если стек пуст;
7. операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
8. отрывающая скобка помещается в стек;
9. закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются.

**6.3 Программная реализация обработки выражений**

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

**6.4 Контрольный пример**

Пример преобразования выражений из контрольных примеров к обратной польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления и преобразования к ассемблерному коду. В приложении Г приведены изменённые таблицы лексем и идентификаторов, отображающие результаты преобразования выражений в польский формат.

Таблица 6.2. – Преобразование выражений к ПОЛИЗ

|  |  |
| --- | --- |
| Выражение | Обратная польская запись для выражения |
| i[2]=(((l[3]+l[4])-i[0])\*l[5])/l[6]; | i[2]=l[3]l[4]+i[0]-l[5]\*l[6]/ |
| i[23]=(i[23]+l[26])\*l[26] | i[23]=i[23]l[26]+l[26]\* |
| i[3]=(((l[4]+l[5])-i[0])\*l[6]) | i[3]=l[4]l[5]+i[0]-l[6]\* |

Преобразование выражений в обратную польскую запись в языке VYR-2023 упрощает алгоритмы их вычисления и преобразования к ассемблерному коду.

**7. Генерация кода**

**7.1 Структура генератора кода**

В языке VYR-2023 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. Структура генератора кода VYR-2023 представлена на рисунке 7.1.



Рисунок 7.1 – Структура генератора кода

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.

**7.2 Представление типов данных в оперативной памяти**

Элементы таблицы идентификаторов расположены сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке VYR-2023 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка и ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке VYR-2023 | Тип идентификатора на языке ассемблера | Пояснение |
| integer | sdword | Хранит целочисленный тип данных. |
| string | byte | Каждый символ строки типа str хранится в поле размером 1 байт. |

Следовательно, таблица 7.1 показывает соответствия между типами идентификаторов на языке VYR-2023 и языке ассемблера. Это важно при переводе кода с языка VYR-2023 на язык ассемблера, чтобы правильно определить типы данных и использовать соответствующие инструкции и регистры для работы с идентификаторами.

**7.3 Статическая библиотека**

В языке VYR-2023 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически.

Таблица 7.3 – Функции статической библиотеки

|  |  |
| --- | --- |
| Функция | Назначение |
| int outw(char\* value)  int outwf(char\* value) | Вывод на консоль строки value  Вывод с переходом на новую строку |
| int outn(int value)  int outnf(int value) | Вывод на консоль целочисленной переменной value  Вывод с переходом на новую строку |
| int len(char\* source) | Вычисление длины строки |
| int comp(char\* source1, char\* source2) | сравнение строк source и str2 |

На этапе генерации кода в языке ассемблера библиотека подключается с использованием директивы includelib. Затем, с помощью оператора EXTRN, объявляются имена функций из этой библиотеки.

* 1. **Особенности алгоритма генерации кода**

В языке VYR-2023 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке

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 VYR-2023. 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 Д. Результат работы контрольного примера приведён на рисунке 7.2.

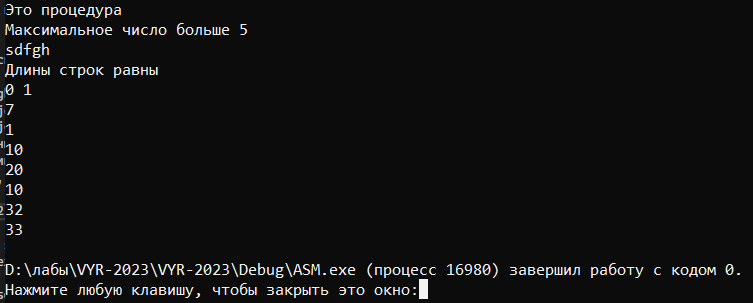


Рисунок 7.2. Результат работы программы на языке VYR-2023

Пример программы проверку прошёл корректно.

**8. Тестирование транслятора**

**8.1 Общие положения**

В основе тестов лежит проверка работоспособности всех анализаторов. При обнаружении компилятором ошибки она будет обрабатываться одним из анализаторов в зависимости от типа ошибки. Все сообщения об ошибках будут храниться в файле text.txt.log.

**8.2 Результаты тестирования**

В языке VYR-2023 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 – Тестирование проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| …  main  {  ё  integer а = 2;  …  } | Ошибка 111: Не доступный символ в исходном файле(-in) Строка: 13 Позиция в строке: 2 |

На этапе лексического анализа в языке VYR-2023 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 |
| …  main  {…  integer x1а1 = 1;  …  } | Ошибка 201: Лексический анализатор: Неверное имя идентификатора строка 14 позиция 4 |

На этапе синтаксического анализа в языке VYR-2023 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  {  integer x;}  } | Ошибка 601: Синтаксическая ошибка: Ошибка в теле функции.  601: строка 14, Синтаксическая ошибка: Ошибка в теле функции |
| string function fi({}  main{} | Ошибка 603: Синтаксическая ошибка: Ошибка в параметрах функции 603: строка 12, Синтаксическая ошибка: Ошибка в параметрах функции |
| string function fi()  {  output  } | Ошибка 602: Синтаксическая ошибка: Ошибка в выражении  602: строка 15, Синтаксическая ошибка: Ошибка в выражении |
| main  {  sum(4, 5;  } | Ошибка 600: Синтаксическая ошибка: Неверная структура программы 600: строка 12, Синтаксическая ошибка: Неверная структура программы |
| main {  integer x = 1;  integer y = 2;  if: x > y  } | Ошибка 606: Синтаксическая ошибка: Неверная структура условия  606: строка 19, Синтаксическая ошибка: Неверная структура условия |

Семантический анализ в языке VYR-2023 содержит множество проверок по семантическим правилам, описанным в пункте 1.16. Итоги тестирования семантического анализатора на корректное обнаружение семантических ошибок приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| string function fi(){} | Ошибка 301: Семантическая ошибка: Отсутствует точка входа main |
| main{}  main{} | Ошибка 302: Семантическая ошибка: Обнаружено несколько точек входа main |
| main  {  integer x = 1;  integer x = 2;  } | Ошибка 305: Семантическая ошибка: Попытка переопределения идентификатора cтрока 4 позиция 0 |

Продолжение таблицы 8.4

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| integer function fi(integer x){return x;}  main{fi();} | Ошибка 308: Семантическая ошибка: Кол-во ожидаемых и переданных функция и параметров не совпадают cтрока 3 позиция 0 |
| integer function fi(integer x){return x;}  main  {  string a = “1”;  fi(a);  } | Ошибка 309: Семантическая ошибка: Несовпадение типов передаваемых параметров cтрока 6 позиция 0 |
| main{  string x = "";  } | Ошибка 310: Семантическая ошибка: Использование пустого строкового литерала недопустимо cтрока 2 позиция 0 |
| main{  string x = ";  } | Ошибка 311: Семантическая ошибка: Не закрыт строковый литерал cтрока 2 позиция 0 |
| string function fi()  {  integer x = 5;  return x;  } | Ошибка 315: Семантическая ошибка: Тип функции и возвращаемого значения не совпадают cтрока 4 |
| main  {  string x;  x = "abc" + "d";  } | Ошибка 316: Семантическая ошибка: Недопустимое строковое выражение справа от знака '=' cтрока 4 |

# Заключение

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования VYR-2023 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

1. Сформулирована спецификация языка VYR-2023;
2. Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
3. Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
4. Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
5. Осуществлена программная реализация синтаксического анализатора;
6. Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
7. Разработан транслятор кода на язык ассемблера;
8. Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка VYR-2023 включает:

1. 2 типа данных;
2. Поддержка оператора вывода строки;
3. Возможность вызова функций стандартной библиотеки;
4. Наличие 5 арифметических операторов для вычисления выражений;
5. Поддержка функций, процедур, операторов цикла и условия;
6. Структурированная и классифицированная система для обработки ошибок пользователя.

Проделанная работа позволила получить необходимое представление о структурах и процессах, использующихся при построении трансляторов, а также основные различия и преимущества тех или иных средств трансляции.

# Список использованных источников

1. Habr – Об изучении компиляторов и создании языков программирования [Электронный ресурс]. – Режим доступа: https://habr.com/ru/search/.

2. Построение компиляторов / Никлаус Вирт 2010. – 194 с.

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

4. Язык программирования C++.Лекции и упражнения [6-е издание] / Стивен Прата 2019 – 1094 с.

Приложение А

Листинг 1 – Исходный код программы на язык VYR-2023

integer function find\_max(integer x, integer y){

integer max;

if: x>y

{max = x;}

^{max = y;}

return max;

}

string function foo(string xxx){

return xxx;

}

void function func(){

println "Это процедура";

return;

}

main

{

func();

integer max = find\_max(2, 9);

string text = "Максимальное число больше 5";

if:max>5

{println text;}

string res = foo("sdfgh");

println res;

string str1 = "Project";

string str2 = "Projec!";

integer strc1 = len(str1);

integer strc2 = len(str2);

if: strc1 ~ strc2

{text = "Длины строк равны";}

^{text = "Длины строк НЕ равны";}

println text;

integer answer1 = comp(str1, str2);

integer answer2 = comp(str1, str1);

print answer1;

print " ";

println answer2;

integer math1 = 5 - 2;

integer math2 = 5 + 2;

integer math3 = 5 % 2;

integer math4 = 5 \* 2;

integer math5 = 5 / 2;

println math2;

println math3;

println math4;

println math5;

integer check1 = 16x01;

integer check2 = 16x20;

integer res2 = check1 + check2;

println check1;

println check2;

println res2;

}

Приложение Б

Листинг 1 – Таблица идентификаторов контрольного примера

------------------Таблица индетификаторов------------------

Идентификатор | тип данных | первое вхождение | содержание

\_\_\_\_\_Литерал\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

L1 | string | 56 | [13]Это процедура

L2 | integer | 72 | 2

L3 | integer | 74 | 9

L4 | string | 80 | [27]Максимальное число больше 5

L5 | integer | 86 | 5

L6 | string | 98 | [5]sdfgh

L7 | string | 107 | [7]Project

L8 | string | 112 | [7]Projec!

L9 | string | 139 | [17]Длины строк равны

L10 | string | 146 | [20]Длины строк НЕ равны

L11 | string | 176 | [1]

L12 | integer | 231 | 1

L13 | integer | 236 | 32

\_\_\_\_\_Переменная\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

find\_maxmax | integer | 12 | -

max | integer | 68 | -

text | string | 78 | -

res | string | 94 | -

str1 | string | 105 | -

str2 | string | 110 | -

strc1 | integer | 115 | -

strc2 | integer | 123 | -

answer1 | integer | 153 | -

answer2 | integer | 163 | -

math1 | integer | 182 | -

math2 | integer | 189 | -

math3 | integer | 196 | -

math4 | integer | 203 | -

math5 | integer | 210 | -

check1 | integer | 229 | -

check2 | integer | 234 | -

res2 | integer | 239 | -

\_\_\_\_\_Параметр\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

find\_maxx | integer | 5 | -

find\_maxy | integer | 8 | -

fooxxx | string | 42 | -

source | string | -1 | -

source1 | string | -1 | -

source2 | string | -1 | -

\_\_\_\_\_Функция\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

find\_max | integer | 2 | -

foo | string | 39 | -

func | void | 51 | -

len | integer | 117 | -

comp | integer | 155 | -

\_\_\_\_\_Оператор\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

- | - | 185 | -

+ | - | 192 | -

% | - | 199 | -

\* | - | 206 | -

/ | - | 213 | -

+ | - | 242 | -

\_\_\_\_\_Логический оператор\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

> | - | 17 | -

> | - | 85 | -

~ | - | 133 | -

Приложение В

Листинг 1 – Грамматика языка VYR-2023

Greibach greibach(

NS('S'),

TS('$'),

11,

Rule(

NS('S'),

GRB\_ERROR\_SERIES + 0,

6,

Rule::Chain(4, TS('m'), TS('{'), NS('N'), TS('}')),

Rule::Chain(7, TS('t'), TS('f'), TS('i'), TS('('), TS(')'), NS('T'), NS('S')),

Rule::Chain(8, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), NS('T'), NS('S')),

Rule::Chain(7, TS('n'), TS('f'), TS('i'), TS('('), TS(')'), NS('T'), NS('S')),

Rule::Chain(8, TS('n'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), NS('T'), NS('S')),

Rule::Chain(5, TS('m'), TS('{'), NS('N'), TS('}'), NS('S'))

),

Rule(

NS('T'),

GRB\_ERROR\_SERIES + 0,

4,

Rule::Chain(3, TS('{'), NS('N'), TS('}')),

Rule::Chain(5, TS('{'), NS('N'), TS('r'), TS(';'), TS('}')),

Rule::Chain(6, TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}')),

Rule::Chain(5, TS('{'), TS('r'), NS('E'), TS(';'), TS('}'))

),

Rule(

NS('N'),

GRB\_ERROR\_SERIES + 1,

24,

Rule::Chain(4, TS('t'), TS('i'), TS(';'), NS('N')),

Rule::Chain(6, TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('N')),

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

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

Rule::Chain(4, TS('i'), TS('('), TS(')'), TS(';')),

Rule::Chain(6, TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),

Rule::Chain(5, TS('t'), TS('i'), TS('='), NS('E'), TS(';')),

Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),

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

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

Rule::Chain(4, TS('z'), TS('i'), TS(';'), NS('N')),

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

Rule::Chain(4, TS('z'), TS('l'), TS(';'), NS('N')),

Rule::Chain(4, TS('p'), NS('E'), TS(';'), NS('N')),

Rule::Chain(4, TS('z'), NS('E'), TS(';'), NS('N')),

Rule::Chain(3, TS('p'), NS('E'), TS(';')),

Rule::Chain(3, TS('z'), NS('E'), TS(';')),

Rule::Chain(3, TS('t'), TS('i'), TS(';')),

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

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

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

Rule::Chain(3, TS('z'), TS('l'), TS(';')),

Rule::Chain(4, TS('w'), NS('I'), TS('|'), NS('X')),

Rule::Chain(5, TS('w'), NS('I'), TS('|'), NS('X'), NS('N'))

),

Rule(

NS('B'),

GRB\_ERROR\_SERIES + 6,

11,

Rule::Chain(5, TS('t'), TS('i'), TS('='), NS('E'), TS(';')),

Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),

Rule::Chain(3, TS('p'), NS('E'), TS(';')),

Rule::Chain(3, TS('z'), NS('E'), TS(';')),

Rule::Chain(3, TS('t'), TS('i'), TS(';')),

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

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

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

Rule::Chain(3, TS('z'), TS('l'), TS(';')),

Rule::Chain(4, TS('w'), NS('I'), TS('|'), NS('X')),

Rule::Chain(5, TS('w'), NS('I'), TS('|'), NS('X'), NS('N'))

),

Rule(

NS('X'),

GRB\_ERROR\_SERIES + 6,

2,

Rule::Chain(7, TS('{'), NS('N'), TS('}'), TS('^'), TS('{'), NS('N'), TS('}')),

Rule::Chain(3, TS('{'), NS('N'), TS('}'))

),

Rule(

NS('O'),

GRB\_ERROR\_SERIES + 7,

1,

Rule::Chain(3, TS('{'), NS('N'), TS('}'))

),

Rule(

NS('E'),

GRB\_ERROR\_SERIES + 2,

10,

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

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

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

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

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

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

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

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

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

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

),

Rule(

NS('F'),

GRB\_ERROR\_SERIES + 3,

2,

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

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

),

Rule(

NS('W'),

GRB\_ERROR\_SERIES + 4,

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 + 2,

2,

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

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

),

Rule(

NS('I'),

GRB\_ERROR\_SERIES + 6,

2,

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

Rule::Chain(6, TS(':'), NS('E'), TS('q'), NS('E'), TS('&'), NS('I'))

)

);

Листинг 2 – Структура магазинного автомата

struct MfstState {

short lenta\_position;

short nrule;

short nrulechain;

MFSTSTSTACK st;

MfstState();

MfstState(short pposition, MFSTSTSTACK pst, short pnrulechain);

MfstState(short pposition, MFSTSTSTACK pst, short pnrule, short pnrulechain);

};

struct Mfst {

enum RC\_STEP {

NS\_OK,

NS\_NORULE,

NS\_NORULECHAIN,

NS\_ERROR,

TS\_OK,

TS\_NOK,

LENTA\_END,

SURPRISE,

};

struct MfstDiagnosis {

short lenta\_position;

RC\_STEP rc\_step;

short nrule;

short nrule\_chain;

MfstDiagnosis();

MfstDiagnosis(short plenta\_position, RC\_STEP prc\_step, short pnrule, short pnrule\_chain);

}diagnosis[MFST\_DIAGN\_NUMBER];

class my\_stack\_MfstState :public std::stack<MfstState> {

public:

using std::stack<MfstState>::c;

};

GRBALPHABET\* lenta;

short lenta\_position;

short nrule;

short nrulechain;

short lenta\_size;

GRB::Greibach greibach;

LT::LexTable lex;

bool more = false;

Log::LOG log;

MFSTSTSTACK st;

my\_stack\_MfstState storestate;

Mfst();

Mfst(LT::LexTable& plex, GRB::Greibach pgreibach);

char\* getCSt(char\* buf);

char\* getCLenta(char\* buf, short pos, short n = 25);

char\* getDiagnosis(short n, char\* buf);

bool savestate();

bool resetstate();

bool push\_chain(GRB::Rule::Chain chain);

RC\_STEP step();

bool start();

bool saveddiagnosis(RC\_STEP pprc\_step);

void printrules();

struct Deducation {

short size;

short\* nrules;

short\* nrulechains;

Deducation() {

size = 0;

nrules = 0;

nrulechains = 0;

}

}deducation;

bool savededucation();

};

Листинг 3 – Структура грамматики Грейбах

struct Greibach //грамматика Грейбах

{

short size; //количество правил

GRBALPHABET startN; //стартовый символ

GRBALPHABET stbottomT;//дно стека

Rule\* rules; //множество правил

Greibach() { short size = 0; startN = 0; stbottomT = 0; rules = 0; };

Greibach(

GRBALPHABET pstartN, //стартовый символ

GRBALPHABET pstbottom, //дно стека

short psize, //количество правил

Rule r, ... //правила

);

short getRule( //получить правило, возвращается номер правила или -1

GRBALPHABET pnn, //левый символ правила

Rule& prule //возвращаемое правило грамматики

);

Rule getRule(short n); //получить правило по номеру

};

Greibach getGreibach(); //получить грамматику

Листинг 4 – Разбор исходного кода синтаксическим анализатором

Шаг :Правило Входная лента Стек

0 : S->tfi()TS tfi(ti,ti){ti;w:iqi|{i=i; S$

0 : SAVESTATE: 1

0 : tfi(ti,ti){ti;w:iqi|{i=i; tfi()TS$

1 : fi(ti,ti){ti;w:iqi|{i=i;} fi()TS$

2 : i(ti,ti){ti;w:iqi|{i=i;}^ i()TS$

3 : (ti,ti){ti;w:iqi|{i=i;}^{ ()TS$

4 : ti,ti){ti;w:iqi|{i=i;}^{i )TS$

5 : TS\_NOK / NS\_NORULECHAIN

5 : RESSTATE

5 : tfi(ti,ti){ti;w:iqi|{i=i; S$

6 : S->tfi(F)TS tfi(ti,ti){ti;w:iqi|{i=i; S$

6 : SAVESTATE: 1

6 : tfi(ti,ti){ti;w:iqi|{i=i; tfi(F)TS$

7 : fi(ti,ti){ti;w:iqi|{i=i;} fi(F)TS$

8 : i(ti,ti){ti;w:iqi|{i=i;}^ i(F)TS$

9 : (ti,ti){ti;w:iqi|{i=i;}^{ (F)TS$

10 : ti,ti){ti;w:iqi|{i=i;}^{i F)TS$

11 : F->ti ti,ti){ti;w:iqi|{i=i;}^{i F)TS$

11 : SAVESTATE: 2

11 : ti,ti){ti;w:iqi|{i=i;}^{i ti)TS$

12 : i,ti){ti;w:iqi|{i=i;}^{i= i)TS$

13 : ,ti){ti;w:iqi|{i=i;}^{i=i )TS$

Листинг 4 (прод.) – Разбор исходного кода синтаксическим анализатором

242: E->i(W)M i;} E;N}$

2242: SAVESTATE: 109

2242: i;} i(W)M;N}$

2243: ;} (W)M;N}$

2244: TS\_NOK / NS\_NORULECHAIN

2244: RESSTATE

2244: i;} E;N}$

2245: E->i()M i;} E;N}$

2245: SAVESTATE: 109

2245: i;} i()M;N}$

2246: ;} ()M;N}$

2247: TS\_NOK / NS\_NORULECHAIN

2247: RESSTATE

2247: i;} E;N}$

2248: NS\_NRCHAIN/NS\_NR

2248: RESSTATE

2248: zi;} N}$

2249: N->zE; zi;} N}$

2249: SAVESTATE: 108

2249: zi;} zE;}$

2250: i;} E;}$

2251: E->i i;} E;}$

2251: SAVESTATE: 109

2251: i;} i;}$

2252: ;} ;}$

2253: } }$

2254: $

2255: 6

2256: ------>LENTA\_END

Приложение Г

Листинг 1 – Программная реализация механизма преобразования в ПОЛИЗ

bool PolishNotation(int i, Lex::LEX& lex)

{

std::stack<LT::Entry> stack; // В стек будем заносить операции

std::queue<LT::Entry> queue; // В очередь будем заносить операнды

LT::Entry placeholder\_symbol;

placeholder\_symbol.idxTI = -1;

placeholder\_symbol.lexema = ' ';

placeholder\_symbol.sn = lex.lextable.table[i].sn;

LT::Entry function\_symbol; // Лексема, обозначающая конец функции

function\_symbol.idxTI = LT\_TI\_NULLIDX; // Признак начала и окончания функции

function\_symbol.lexema = '@';

function\_symbol.sn = lex.lextable.table[i].sn;

int idx;

int lexem\_counter = 0;

int parm\_counter = 0; // Количество параметров в функции

int lexem\_position = i; // Запоминаем номер лексемы перед преобразованием

char\* buf = new char[i];

bool findFunc = false; // флаг на нахождение функции

for (i; lex.lextable.table[i].lexema != LEX\_SEMICOLON; i++, lexem\_counter++)

{

switch (lex.lextable.table[i].lexema)

{

case LEX\_ID:

case LEX\_LITERAL:

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

{

findFunc = true;

idx = lex.lextable.table[i].idxTI;

}

else

{

if (findFunc)

parm\_counter++;

queue.push(lex.lextable.table[i]);

}

continue;

case LEX\_LEFTTHESIS:

stack.push(lex.lextable.table[i]);

continue;

case LEX\_RIGHTTHESIS:

while (stack.top().lexema != LEX\_LEFTTHESIS)

{

queue.push(stack.top());

stack.pop();

if (stack.empty())

return false;

}

if (!findFunc)

stack.pop();

else {

function\_symbol.idxTI = idx;

idx = LT\_TI\_NULLIDX;

lex.lextable.table[i] = function\_symbol;

queue.push(lex.lextable.table[i]);

\_itoa\_s(parm\_counter, buf, 2, 10);

stack.top().lexema = buf[0];

stack.top().idxTI = LT\_TI\_NULLIDX;

stack.top().sn = function\_symbol.sn;

queue.push(stack.top());

stack.pop();

parm\_counter = 0;

findFunc = false;

}

continue;

case LEX\_OPERATOR:

while (!stack.empty() && lex.lextable.table[i].priority <= stack.top().priority)

{

queue.push(stack.top());

stack.pop();

}

stack.push(lex.lextable.table[i]);

continue;

}

}

while (!stack.empty())

{

if (stack.top().lexema == LEX\_LEFTTHESIS || stack.top().lexema == LEX\_RIGHTTHESIS)

return false;

queue.push(stack.top());

stack.pop();

}

while (lexem\_counter != 0)

{

if (!queue.empty())

{

lex.lextable.table[lexem\_position++] = queue.front();

queue.pop();

}

else

lex.lextable.table[lexem\_position++] = placeholder\_symbol;

lexem\_counter--;

}

for (int i = 0; i < lexem\_position; i++)

{

if (lex.lextable.table[i].lexema == LEX\_OPERATOR || lex.lextable.table[i].lexema == LEX\_LITERAL)

lex.idtable.table[lex.lextable.table[i].idxTI].idxfirstLE = i;

}

return true;

}

Приложение Д

Листинг 1 – Результат генерации кода контрольного примера в Ассемблере

|  |  |
| --- | --- |
| .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib "../Debug/lib.lib"  ExitProcess PROTO:DWORD  .stack 4096  ExitProcess PROTO :DWORD  outn PROTO : SDWORD  outw PROTO : DWORD  outnf PROTO : SDWORD  outwf PROTO : DWORD  len PROTO : DWORD  comp PROTO : DWORD, : DWORD  stcmp PROTO : DWORD, : DWORD  .const  newline byte 13, 10, 0  L1 byte 'Это процедура', 0  L2 sdword 2  L3 sdword 9  L4 byte 'Максимальное число больше 5', 0  L5 sdword 5  L6 byte 'sdfgh', 0  L7 byte 'Project', 0  L8 byte 'Projec!', 0  L9 byte 'Длины строк равны', 0  L10 byte 'Длины строк НЕ равны', 0  L11 byte ' ', 0  L12 sdword 1  L13 sdword 32  .data  temp sdword ?  buffer byte 256 dup(0)  find\_maxmax sdword 0  max sdword 0  text dword ?  res dword ?  str1 dword ?  str2 dword ?  strc1 sdword 0  strc2 sdword 0  answer1 sdword 0  answer2 sdword 0  int\_to\_char ENDP  ;----------- find\_max ------------  find\_max PROC,  find\_maxx : sdword, find\_maxy : sdword  ; --- save registers ---  push ebx  push edx  ; ----------------------  push find\_maxx  pop ebx  mov left, ebx  push find\_maxy  pop ebx  mov rig, ebx  mov edx, left  cmp edx, rig  jg right1  jl wrong1  right1:  push find\_maxx  pop ebx  mov find\_maxmax, ebx  jmp next1  wrong1:  push find\_maxy  pop ebx  mov find\_maxmax, ebx  next1:  ; --- restore registers ---  pop edx  pop ebx  ; -------------------------  mov eax, find\_maxmax  ret  find\_max ENDP  ;------------------------------  ;----------- foo ------------  foo PROC,  fooxxx : dword  ; --- save registers ---  push ebx  pop ebx  mov rig, ebx  mov edx, left  cmp edx, rig  jg right2  jmp next2  right2:  INVOKE outwf, text  next2:  mov res, offset L6  INVOKE outwf, res  mov str1, offset L7  mov str2, offset L8  push str1  call len  push eax  pop ebx  mov strc1, ebx  push str2  call len  push eax  pop ebx  mov strc2, ebx  push strc1  pop ebx  mov left, ebx  push strc2  pop ebx  mov rig, ebx  mov edx, left  cmp edx, rig  jz right3  jnz wrong3  right3:  mov text, offset L9  jmp next3  wrong3:  mov math2, ebx  push L5  push L2  pop ebx  pop eax  test ebx, ebx  jz EXIT\_DIV\_ON\_NULL  cdq  mov edx, 0  idiv ebx  push edx  pop ebx  mov math3, ebx  push L5  push L2  pop ebx  pop eax  imul eax, ebx  push eax  pop ebx  mov math4, ebx  push L5  push L2  pop ebx  pop eax  test ebx, ebx  jz EXIT\_DIV\_ON\_NULL  cdq  mov edx, 0  idiv ebx  push eax  pop ebx  mov math5, ebx  mov eax, math2  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, math3  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, math4 | math1 sdword 0  math2 sdword 0  math3 sdword 0  math4 sdword 0  math5 sdword 0  check1 sdword 0  check2 sdword 0  res2 sdword 0  left dword ?  rig dword ?  result sdword ?  result\_str byte 4 dup(0)  .code  int\_to\_char PROC uses eax ebx ecx edi esi,  pstr: dword,  intfield : sdword  mov edi, pstr  mov esi, 0  mov eax, intfield  cdq  mov ebx, 10  idiv ebx  test eax, 80000000h  jz plus  neg eax  neg edx  mov cl, '-'  mov[edi], cl  inc edi  plus :  push dx  inc esi  test eax, eax  jz fin  cdq  idiv ebx  jmp plus  fin :  mov ecx, esi  write :  pop bx  add bl, '0'  mov[edi], bl  inc edi  loop write  ret  push edx  ; ----------------------  ; --- restore registers ---  pop edx  pop ebx  ; -------------------------  mov eax, fooxxx  ret  foo ENDP  ;------------------------------  ;----------- func ------------  func PROC    ; --- save registers ---  push ebx  push edx  ; ----------------------  INVOKE outwf, offset L1  ; --- restore registers ---  pop edx  pop ebx  ; -------------------------  ret  func ENDP  ;------------------------------  ;----------- MAIN ------------  main PROC  call func  push L2  push L3  call find\_max  push eax  pop ebx  mov max, ebx  mov text, offset L4  push max  pop ebx  mov left, ebx  push L5  mov text, offset L10  next3:  INVOKE outwf, text  push str1  push str2  call comp  push eax  pop ebx  mov answer1, ebx  push str1  push str1  call comp  push eax  pop ebx  mov answer2, ebx  mov eax, answer1  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outw, offset result\_str  INVOKE outw, offset L11  mov eax, answer2  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  push L5  push L2  pop ebx  pop eax  sub eax, ebx  push eax  pop ebx  mov math1, ebx  push L5  push L2  pop ebx  pop eax  add eax, ebx  push eax  pop ebx  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, math5  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  push L12  pop ebx  mov check1, ebx  push L13  pop ebx  mov check2, ebx  push check1  push check2  pop ebx  pop eax  add eax, ebx  push eax  pop ebx  mov res2, ebx  mov eax, check1  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, check2  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  mov eax, res2  mov result, eax  INVOKE int\_to\_char, offset result\_str, result  INVOKE outwf, offset result\_str  INVOKE ExitProcess,0  EXIT\_DIV\_ON\_NULL:  INVOKE ExitProcess,-1  main ENDP  end main |