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 YVA-2024»

Выполнил студент Яскевич Валерия Александровна

(Ф.И.О.)

Руководитель проекта преп.-стажер Волчек Дарья Ивановна

(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.)

Консультанты преп.-стажер Волчек Дарья Ивановна

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

Нормоконтролер ассистент Волчек Дарья Ивановна

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

Курсовой проект защищен с оценкой

Минск 2024

Содержание

[Введение 4](#_Toc185266885)

[1.1 Характеристика языка программирования 5](#_Toc185266886)

[1.2 Определение алфавита языка программирования 5](#_Toc185266887)

[1.3 Применяемые сепараторы 5](#_Toc185266888)

[1.4 Применяемые кодировки 5](#_Toc185266889)

[1.5 Типы данных 6](#_Toc185266890)

[1.6 Преобразование типов данных 7](#_Toc185266891)

[1.7 Идентификаторы 7](#_Toc185266892)

[1.8 Литералы 8](#_Toc185266893)

[1.9 Объявление данных 8](#_Toc185266894)

[1.10 Инициализация данных 9](#_Toc185266895)

[1.11 Инструкции языка 9](#_Toc185266896)

[1.12 Операции языка 9](#_Toc185266897)

[1.13 Выражения и их вычисления 10](#_Toc185266898)

[1.14 Конструкции языка 10](#_Toc185266899)

[1.15 Область видимости идентификаторов 11](#_Toc185266900)

[1.16 Семантические проверки 11](#_Toc185266901)

[1.17 Распределение оперативной памяти на этапе выполнения 12](#_Toc185266902)

[1.18 Стандартная библиотека и ее состав 12](#_Toc185266903)

[1.19 Вывод и ввод данных 12](#_Toc185266904)

[1.20 Точка входа 12](#_Toc185266905)

[1.21 Препроцессор 13](#_Toc185266906)

[1.22 Соглашения о вызове 13](#_Toc185266907)

[1.23 Объектный код 13](#_Toc185266908)

[1.24 Классификация сообщений транслятора 13](#_Toc185266909)

[1.25 Контрольный пример 13](#_Toc185266910)

[2. Структура транслятора 15](#_Toc185266911)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 15](#_Toc185266912)

[2.2 Перечень входных параметров транслятора 16](#_Toc185266913)

[2.3 Перечень протоколов, формируемых транслятором и их содержимое 16](#_Toc185266914)

[3. Разработка лексического анализатора 18](#_Toc185266915)

[3.1 Структура лексического анализатора 18](#_Toc185266916)

[3.2. Контроль входных символов 19](#_Toc185266917)

[3.3 Удаление избыточных символов 19](#_Toc185266918)

[3.4 Перечень ключевых слов 19](#_Toc185266919)

[3.5 Основные структуры данных 20](#_Toc185266920)

[3.6 Принцип обработки ошибок 20](#_Toc185266921)

[3.7 Структура и перечень сообщений лексического анализа 21](#_Toc185266922)

[3.8 Параметры лексического анализатора 21](#_Toc185266923)

[3.9 Алгоритм лексического анализа 21](#_Toc185266924)

[3.10 Контрольный пример 21](#_Toc185266925)

[4. Разработка синтаксического анализатора 22](#_Toc185266926)

[4.1 Структура синтаксического анализатора 22](#_Toc185266927)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис 22](#_Toc185266928)

[4.3 Построение конечного магазинного автомата 24](#_Toc185266929)

[4.4 Основные структуры данных 25](#_Toc185266930)

[4.5 Описание алгоритма синтаксического разбора 25](#_Toc185266931)

[4.6 Структура и перечень сообщений синтаксического анализатора 25](#_Toc185266932)

[4.7. Параметры синтаксического анализатора и режимы его работы 25](#_Toc185266933)

[4.8. Принцип обработки ошибок 25](#_Toc185266934)

[4.9. Контрольный пример 26](#_Toc185266935)

[5. Разработка семантического анализатора 27](#_Toc185266936)

[5.1 Структура семантического анализатора 27](#_Toc185266937)

[5.2 Функции семантического анализатора 27](#_Toc185266938)

[5.3 Структура и перечень семантических ошибок 27](#_Toc185266939)

[5.4 Принцип обработки ошибок 27](#_Toc185266940)

[5.5 Контрольный пример 28](#_Toc185266941)

[6. Вычисление выражений 29](#_Toc185266942)

[6.1 Выражения, допускаемые языком 29](#_Toc185266943)

[6.2 Польская запись и принцип ее построения 29](#_Toc185266944)

[6.3 Программная реализация обработки выражений 30](#_Toc185266945)

[6.4 Контрольный пример 30](#_Toc185266946)

[7. Генерация кода 31](#_Toc185266947)

[7.1 Структура генератора кода 31](#_Toc185266948)

[7.2 Представление типов данных в оперативной памяти 31](#_Toc185266949)

[7.3 Статическая библиотека 32](#_Toc185266950)

[7.4 Особенности алгоритма генерации кода 32](#_Toc185266951)

[7.5 Входные параметры генератора кода 32](#_Toc185266952)

[7.6 Контрольный пример 32](#_Toc185266953)

[8. Тестирование транслятора 33](#_Toc185266954)

[8.1 Общие положения 33](#_Toc185266955)

[8.2 Результаты тестирования 33](#_Toc185266956)

[Заключение 34](#_Toc185266957)

[Список использованных источников 35](#_Toc185266958)

[Приложение А 36](#_Toc185266959)

[Приложение Б 38](#_Toc185266960)

[Приложение В 49](#_Toc185266961)

[Приложение Г 54](#_Toc185266962)

[Приложение Д 58](#_Toc185266963)

# **Введение**

Задачей данного курсового проекта была поставлена разработка транслятора своего языка программирования YVA-2024. Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над числами.

Главная задача транслятора заключается в том, чтобы сделать исходный код на данном языке программирования понятной компьютеру. Для решения этой задачи был выбран способ трансляции исходного кода моего языка программирования в исходный код на языке ассемблера.

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 обратной польской нотации;

– генерация кода на язык ассемблера;

– тестирование транслятора;

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

**1. Спецификация языка программирования**

**1.1 Характеристика языка программирования**

Язык программирования YVA-2024 является языком программирования высокого уровня. Он является компилируемым. В языке отсутствует преобразование типов. В языке поддерживается 4 типа данных: целочисленный (numb), строковый (stroke), символьный (symbol), логический (boolean). В стандартной библиотеке имеются функции для работы с целочисленным и строковыми типами данных, например, преобразование строки в число.

**1.2 Определение алфавита языка программирования**

Символы, используемые на этапе выполнения: [a…z], [A…Z], [0…9], [a…я], [A…Я], символы пробела, перевода строки, спецсимволы: [] (), ; : + - / \* % > < ! {}|&.

**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

|  |  |
| --- | --- |
| Символ(ы) | Назначение |
| Пробел | Разделитель цепочек. Допускается везде, кроме имен идентификаторов и ключевых слов |
| **[** … **]** | Блок цикла |
| { … } | Блок функций |
| **,** | Разделитель параметров функций |
| **+ - \* / %** | Арифметические операции |
| **> < ! & | ^** | Логические операторы (Операции сравнения: больше, меньше, логическое не, логическое и, логическое или, логическое равенство) |
| **;** | Разделитель программных конструкций |
| **=** | Оператор присваивания |

Символы сепараторы включают в себя символы пробела, блоки функций или циклов, блоки параметров функций и тд.

**1.4 Применяемые кодировки**

Для написания программ на языке YVA-2024 используется кодировка Windows – 1251, представленная на рис.1.1.

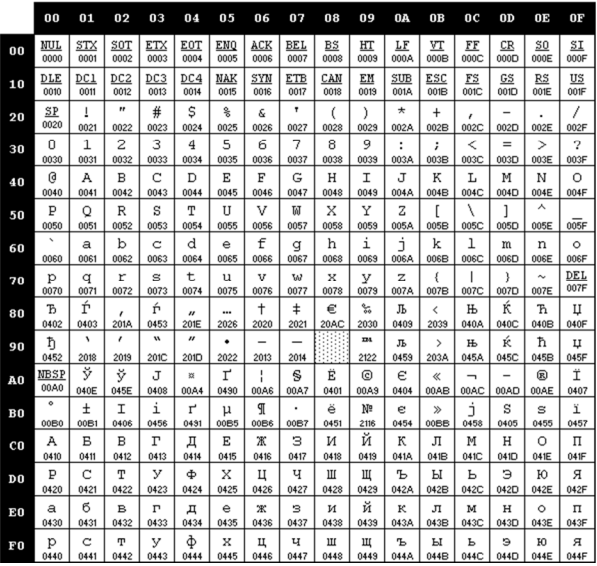


Рис. 1.1 - Алфавит вводных символов

Согласно кодировке Windows-1251 был разработан алфавит языка YVA-2024.

**1.5** **Типы данных**

В языке YVA-2024, помимо четырех фундаментальных типов данных — целочисленного, строкового, символьного и логического — существует возможность создания пользовательских типов данных, что позволяет повышать гибкость и мощность программирования. Каждый из фундаментальных типов имеет свои особенности: целочисленный тип поддерживает операции арифметики, строковый тип позволяет работать с текстовыми данными, символьный тип используется для представления одиночных символов, а логический тип предназначен для работы с булевыми значениями (истина или ложь). Эти базовые типы могут комбинироваться для создания более сложных структур данных

Таблица 1.2 – Типы данных языка YVA-2024

|  |  |
| --- | --- |
| Типы данных | Характеристика |
| Целочисленный тип данных numb | Является целочисленным типом данных. Этот тип данных занимает 4 байта. Предназначен для арифметических операций над числами. Инициализация по умолчанию: 0.  Поддерживаемые операции:  + (бинарный) – Оператор сложения;  - (бинарный) – Оператор вычитания;  \* (бинарный) – Оператор умножения;  / (бинарный) – Оператор деления;  = (бинарный) – Оператор присваивания  В качестве операторов условия или условия цикла можно использовать следующие операторы;  > (бинарный) – Оператор “больше”;  < (бинарный) – Оператор “меньше”  ^ (бинарный) – Оператор равенства |
| Cтроковый 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:  =(бинарный)оператор присваивания |
| Символьный тип данных char | Фундаментральный тип данных. Используется для работы с символом, занимающим 1 байт.  Инициализация по умолчанию: символ нулевой длины “”.  Операции над данными символьного типа:  =(бинарный)оператор присваивания. |
| Логический тип данный boolean | Является логическим типом данных. Переменные данного типа могу принимать 2 значения:true или false.  Операции над данными логического типа:  |(бинарный) – Логическое или  &(бинарный) – Логическое И |

Эти типы данных предоставляют разработчикам инструменты для выполнения разнообразных операций, от арифметических до операций сравнения и условий, что делает их ключевыми строительными блоками для создания разнообразных программных решений.

**1.6 Преобразование типов данных**

Преобразование типов данных в языке YVA-2024 не поддерживается, так как язык YVA-2024 является типизированным.

**1.7 Идентификаторы**

Общее количество идентификаторов ограниченно максимальным размером таблицы идентификаторов (4096). Идентификаторы могут содержать символы как нижнего регистра, так и верхнего. Максимальная длина идентификатора равна 10 символам. Идентификаторы, объявленные внутри функционального блока, получают область видимости , идентичную имени функции, внутри которой они объявлены. Данные правила действуют для всех идентификаторов. Зарезервированные идентификаторы не предусмотрены. Идентификаторы не должны совпадать с ключевыми словами. Типы идентификаторов: имя переменной, имя функции, параметр функции.

Правило составления идентификатора:

<буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | r | s | t | u | v | w | x | y |z

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<идентификатор> ::= <буква> {(<буква>| <цифра>)}

**1.8 Литералы**

С помощью литералов осуществляется инициализация переменных. Все литералы являются rvalue. Имеются литералы: целочисленные десятичного представления, строковые, логические, а также символьные. Подробное описание литералов языка YVA-2024 представлены в таблице  1.3.

Таблица 1.3 – Литералы

|  |  |
| --- | --- |
| Литералы | Определение |
| Целочисленные литералы в десятичном представлении | Последовательность цифр 0…9 с предшествующим знаком минус или без него (знак минус не отделяется пробелом) |
| Строковые литералы | Набор символов алфавита языка, заключенных в одинарные кавычки |
| Логические литералы | Может принимать 2 значения: true или false |
| Символьные литералы | Символ алфавита языка, заключенный в двойные кавычки |

Ограничения на целочисленные литералы: не могут начинаться с 0, если их значение не 0; если литерал отрицательный, после знака “-” не может идти 0.

**1.9** **Объявление данных**

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

Пример объявления числового типа данных с инициализации:

declare numb x = 14;

Пример объявления строкового типа данных с инициализацией:

declare string str = 'привет мир';

Пример объявления логического типа данные с инициализацией:

declare bool b = true;

declare bool c = false;

Пример объявления символьного типа данных с иницициализацией:

declare char s = “S”;

Для объявления функций используется ключевое слово **func**, 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. Объектами-инициализаторами могут быть только идентификаторы и литералы. При объявлении переменные инициализируются значением по умолчанию. Для numb значение 0, для string**,** charстрока нулевой длины (“”), для bool – false.

**1.11** **Инструкции языка**

Инструкции языка YVA-2024 представлены в таблице 1.4.

Таблица 1.4 – Инструкции языка

|  |  |
| --- | --- |
| Инструкция | Реализация |
| Объявление переменной | **declare** <тип данных><идентификатор>; |
| Возврат значения из функции | **return** <идентификатор> | <литерал>; |
| Вывод данных | **write** <идентификатор> | <литерал>;  **writeline** <идентификатор> | <литерал>; |
| Вызов функции | <идентификатор функции>(<список параметров>); |
| Присваивание | <идентификатор> = <выражение>; |

Программа обычно представляет собой последовательность инструкций.

**1.12 Операции языка**

Операции языка YVA-2024 (арифметические: сложение, разность, умножение, деление; логические: логическое равенство, больше, меньше, не равно, логическое и, логическое или) и их приоритет представлен в таблице 1.5.

Таблица 1.5 – Операции языка YVA-2024

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические | **+ -** сложение (приоритет 5)  - **-** разность (приоритет 5)  **\* -** умножение (приоритет 4)  **/ -** деление (приоритет 4) |
| Логические | ^ - логическое равенство (приоритет 7)  **>** - больше (приоритет 7)  **<** - меньше (приоритет 7)  **!** – не равно (приоритет 8)  &-логическое и (приоритет 9)  | - логическое или (приоритет 10) |

Для повышения приоритета выполнения операций 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. Преобразование выражений приведено в главе 5.

**1.14 Конструкции языка**

Программа на языке YVA-2024 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода. Кроме того, следует придерживаться принципа модульности, разбивая сложные задачи на более простые, что облегчает тестирование и отладку отдельных компонентов. Использование стандартных наименований и соглашений о кодировании способствует более понятному и организованному коду, что особенно важно при работе в команде. Программные конструкции языка YVA-2024 представлены в таблице 1.6.

Таблица 1.6 – Конструкции языка YVA-2024

|  |  |
| --- | --- |
| Конструкция | Реализация |
| Главная функция | **main**  **{**…  **return** <идентификатор/литерал>;**}** |
| Внешняя функция | <тип данных> **func** <идентификатор> (<тип> <идентификатор>, ...)  {  …    **return** <идентификатор/литерал>;  } |
| Условное выражение | **state:** <идентификатор1> <логический оператор><идентификатор2>**$**  **true:[**<идентификатор> = <литерал> | <идентификатор>**]**  **false:[**<идентификатор> = <литерал> | <идентификатор>**]**  **$** |
| Цикл | **state:**<идентификатор1><логический оператор><идентификатор2>**$**  **while[…]** |

За проверку реализации всех конструкций языка отвечает семантический анализатор.

**1.15 Область видимости идентификаторов**

Область видимости: сверху вниз. Переменные, объявленные в одной функции не доступны в другой. Все операции и объявления происходят внутри какого-либо блока или тела функции. Каждая переменная или параметр функции получают область видимости – название функции, в 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 YVA-2024 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, объявления;
7. Превышение размера строковых и числовых литералов;
8. Проверка совпадений типов в операциях;

**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 областью видимости, 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 YVA-2024 присутствует стандартная библиотека, которая подключается автоматически на этапе трансляции исходного кода в язык ассемблера.

Содержимое стандартной библиотеки представлено в таблице 1.8.

Таблица 1.8 – Стандартная библиотека языка YVA-2024

|  |  |
| --- | --- |
| Функция | Описание |
| numb Rand(numb b) | Целочисленная функция, возвращает псевдослучайное число в определенном диапазоне b. |
| numb Strlen(stroke str) | Целочисленная функция, возвращает размер строки |
| numbInput(numb a) | Целочисленная функция, возвращает число, введенное пользователем. |
| numbToNumb(char\* str) | Целочисленная функция, возвращает числовое значение, преобразованное из строки. |

Так же статическая библиотека содержит функции вывода в поток, которые используются при генерации кода в Assembler.

**1.19 Вывод и ввод данных**

Вывод данных осуществляется с помощью операторов **write** и **writeline**. Допускается использование оператора **write**  с литералами и идентификаторами.

**1.20 Точка входа**

В языке YVA-2024 каждая программа должна содержать главную функцию (точку входа) **main**, с первой инструкции которой начнётся последовательное выполнение команд программы.

Должна иметься только одна точка входа main.

**1.21 Препроцессор**

Команды препроцессора в языке YVA-2024 отсутствуют.

**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**

Язык YVA-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 об ошибках имеют специфический постфикс, зависящий от этапа, на котором обнаружена ошибка.

Список постфиксов приведен в таблице 1.9.

Таблица 1.9 – Список префиксов ошибок в языке YVA-2024

|  |  |
| --- | --- |
| Постфикс | Пояснение |
| !Синтаксическая! | Указывает, что ошибка была обнаружена на стадии синтаксического анализа. |
| !Лексического анализа! | Указывает, что ошибка была обнаружена на стадии лексического анализа. |
| !Семантическая! | Указывает, что ошибка была обнаружена на стадии семантического анализа. |

Перечень всех ошибок находится в приложениях Б, В и Г.

**1.25 Контрольный пример**

Контрольный пример демонстрирует главные особенности языка YVA-2024: его фундаментальные типы, основные структуры, функции, использование функция стандартной библиотеки. Исходный код контрольного примера представлен в приложении А.

**2. Структура транслятора**

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

В языке YVA-2024 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые им функции, которые представлены в пункте 2.1. 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, которые описаны в таблице 2.1. Структура транслятора языка YVA-2024 приведена на рис. 2.1



Рис.2.1 - Структура транслятора языка программирования YVA-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;
* 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 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 YVA-2024

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке YVA-2024 , имеющий расширение .txt | Не предусмотрено |
| -log:<путь к log-файлу> | Файл журнала для вывода протоколов работы программы. | Значение по умолчанию:  <имя файла>.log |

Данные параметры указываются как аргументы команды в свойствах отладки и при запуске проекта через консоль разработчика.

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

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 YVA-2024

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования YVA-2024 . |
| Выходной файл, c расширением ".asm" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

Файлы всех протоколов находятся в папке YVA-2024.

**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, производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов.

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.

Функции лексического анализатора:

* удаление «пустых» символов. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) 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, структура лексического анализатора представлена на рис. 3.1.



Рис.3.1 - Структура лексического анализатора

**3.2. Контроль входных символов**

Таблица контроля входных символов представлена в приложении Б

Принцип работы таблицы заключается в соответствии значения каждого элемента значению в таблице ASCII.

Описание значения символов: T – разрешённый символ, F – запрещённый символ, I – игнорируемый символ, S – символ-разделитель.

**3.3 Удаление избыточных символов**

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на лексемы.

Описание алгоритма удаления избыточных символов:

* посимвольно считываем файл с исходным кодом программы;
* в отличие от других символов-разделителей, не записываем пробелы и символы табуляции в таблицу лексем;
* продолжаем считывание файла с исходным кодом программы до встречи с лексемой, отличной от пробела или символа табуляции.

**3.4 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 3.1.

Таблица 3.1 – Соответствие токенов и лексем в языке YVA-2024

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| numb, char, bool, string | t | Названия типов данных языка. |
| Идентификатор | i | Содержит информацию о идентификаторе |
| Литерал | l | Литерал любого доступного типа. |
| func | f | Объявление функции. |
| return | r | Выход из функции/процедуры. |
| main | m | Главная функция. |
| declare | d | Объявление переменной. |
| write | w | Вывод данных. |
| while | v | Указывает на начало тела цикла. |

Окончание таблицы 3.1 – Соответствие токенов и лексем в языке YVA-2024

|  |  |  |
| --- | --- | --- |
| ; | ; | Разделение выражений. |
| Токен | Лексема | Пояснение |
| , | , | Разделение параметров функций. |
| { | { | Начало блока/тела функции. |
| } | } | Закрытие блока/тела функции. |
| ( | ( | Передача параметров в функцию, |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций. |
| = | = | Знак присваивания. |
| +  -  \*  /  ]  [  >  <  !  ^  & | v | Знаки операций. |

В приложении Б находится пример конечного автомата, используемый для разбора цепочки символов.

**3.5 Основные структуры данных**

Структуры таблиц лексем и идентификаторов данных языка YVA-2024, используемых для хранения, представлены в приложении Б.

В таблице лексем содержатся сами лексемы, строка для каждой лексемы, в которой она была замечена. Так же размер самой таблицы лексем. В таблице идентификаторов содержится имя идентификатора, его номер в таблице лексем, тип данных, смысловой тип идентификатора и его значение, а также имя родительской функции.

**3.6 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.

## **3.7 Структура и перечень сообщений лексического анализа**

Перечень сообщений представлен в приложении Б.

Сообщения об ошибках данной стадии имеют предупреждение «!Лексический анализ!» что с легкостью дает пользователю понять, на каком этапе возникла ошибка.

## **3.8 Параметры лексического анализатора**

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся в файл с таблицей лексем, файл с таблицей идентификаторов, а также в командную строку.

**3.9 Алгоритм лексического анализа**

Последовательность выполнения алгоритма работы лексического анализатора представлен ниже.

1. Разделение текста на отдельные лексемы.
2. Распознавание каждой строки в двумерном массиве с помощью автоматов.
3. При удачном прохождении информация заносится в таблицу лексем и идентификаторов. Возврат к шагу 2).
4. Формирование протокола работы
5. При невозможности обработать строку двумерного массива выводится сообщение об ошибке.
6. Конец работы лексического анализатора

**3.10 Контрольный пример**

Результат работы лексического анализатора – таблицы лексем и идентификаторов – 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.



Рис.4.1 – Структура синтаксического анализатора

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

Контекстно-свободная грамматика (КС-грамматика) — это формальная грамматика, используемая для описания синтаксиса языков программирования и естественных языков. Она представляет собой набор правил, которые определяют, как строки символов (например, слова или предложения) могут быть сгенерированы из начального символа. Основные аспекты КС-грамматик включают их структуру, компоненты, характеристики и применение.

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

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

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

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

S – начальный символ грамматики, являющийся нетерминалом.

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

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

Таблица 4.1 – Таблица правил переходов нетерминальных символов

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| S | S–>dtfi(){NrE;};S  S–>m{NrE;} | Стартовые правила, описывающее общую структуру программы |
| F | F->ti | Правила для объявления функции |
| N | N->dtivE;N  N->wi;N  N->wl  N->atfi();N  N->wl;N  N->dtivE;N  N->u(B)[N];N  N->ivE;N  N->ivE;  N->o(B)[N]; | Правила объявления переменных |
| E | E –>i  E –>l  E –>i;  E –>l;  E –>iVE  E –>lVE  E –>iOE  E –>lOE  E–>SQ  E –>UQ  E –>RQ  Ew –>i(G) | Правила построения выражений |

Данные правила используются для построения дерева разбора контрольного примера, представленного в приложении А.

**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 |

Конечные автоматы являются мощным инструментом для распознавания формальных языков. Процесс их составления включает в себя определение алфавита, языка, состояний, переходов и начальных/конечных состояний. Конечные автоматы могут быть использованы для реализации различных задач, включая обработку текстов, компиляцию и верификацию программ.

**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. 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 Структура и перечень сообщений синтаксического анализатора**

Сообщения, генерируемые синтаксическим анализатором представлены в приложении В.

**4.7. 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 синтаксического анализатора.

**4.8. Принцип обработки ошибок**

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

**4.9. Контрольный пример**

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 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.

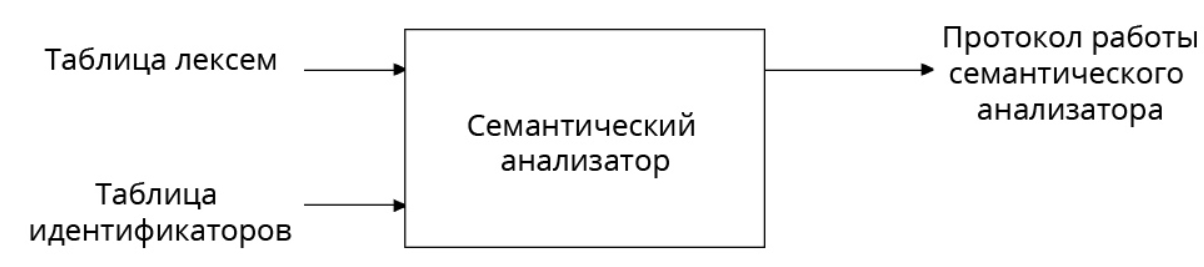


Рис.5.1. - Структура семантического анализатора

**5.2 Функции семантического анализатора**

Семантический анализатор проверяет правильность составления программных конструкций. При невозможности подобрать правило перехода будет выведен код ошибки, а также код этой ошибки. Информация об ошибках выводится в консоль, а также в протокол работы.

**5.3 Структура и перечень семантических ошибок**

Сообщения, формируемые семантическим анализатором, представлены в приложении Г.

## **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.

## **5.5 Контрольный пример**

Таблица 5.5 представляет собой контрольный пример исходного кода с возможными ошибками, которые могут возникнуть при компиляции. В ней демонстрируются сценарии неправильного использования ключевых слов, повторных объявлений идентификаторов и прочих недочетов.

Таблица 5.5 – Таблица возможных ошибок исходного кода

|  |  |
| --- | --- |
| Исходный код | Текст сообщения |
| declare string func sayIt() {  writeline "valar morghulis";  return 0;  }; | Ошибка 704: !Семантическая! Нарушены типы данных в выражении или условной конструкции, строка 50, столбец 5 |
| add numb func StrLen(string a);  add numb func ToNumb(numb a);  add numb func Rand(numb n);  add numb func Input(); | Ошибка 706: !Семантическая! Ошибка экспорта: неверные параметры, строка 9, столбец 28 |
| declare numb num = StrLen("0000000");  declare numb num = StrLen("0000000");  declare numb num2 = ToNumb("123"); | Ошибка 700: !Семантическая! Повторное объявление идентификатора, строка 39, столбец 14 |
| main  {  add numb func StrLen(string a);  add numb func ToNumb(string a);  add numb func Rand(numb n);  add numb func Input();  }  main  {  add numb func StrLen(string a);  add numb func ToNumb(string a);  add numb func Rand(numb n);  add numb func Input();….} | Ошибка 125: !Лексического анализа! Обнаружена вторая точка входа, строка 13, столбец 1 |

Анализ таких контрольных примеров помогает разработчикам лучше понимать возможные проблемы в своем коде

**6. Вычисление выражений**

**6.1 Выражения, допускаемые языком**

В языке YVA-2024 допускаются вычисления выражений целочисленного, а также логического типов данных с поддержкой вызова функций внутри целочисленных выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритет операций

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| ( ) | 0 |
| \* | 4 |
| / | 4 |
| + | 5 |
| - | 5 |
| % | 4 |
| | | 10 |
| & | 9 |

Чем выше приоритет имеет операция, тем левее она будет находиться после преобразования в польскую запись.

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

Все выражения языка YVA-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 польской записи:

* исходная строка: выражение;
* результирующая строка: польская запись;
* стек: пустой;
* исходная строка просматривается слева направо;
* операнды переносятся в результирующую строку;
* операция записывается в стек, если стек пуст;
* операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
* отрывающая скобка помещается в стек;
* закрывающая скобка выталкивает все операции;

## **6.3 Программная реализация обработки выражений**

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

## **6.4 Контрольный пример**

Пример преобразования выражения к польской записи представлен в таблице 6.2.

Преобразование выражений в формат польской записи в нашем случае необходимо для построения более простых алгоритмов при последующей обработки таблицы лексем.

Таблица 6.2 – Преобразование выражений к ПОЛИЗ

|  |  |  |
| --- | --- | --- |
| Исходное выражение | Стек | Результирующая строка |
| l\*((l-l)/l)+l |  |  |
| \*((l-l)/l)+l |  | l |
| ((l-l)/l)+l | \* | l |
| (l-l)/l)+l | \*,( | l |
| l-l)/l)+l | \*,(,( | l |
| -l)/l)+1 | \*,(,( | l,l |
| l)/l)+l | \*,(,(,- | l,l |
| )/l)+l | \*,(,(,- | l,l,l |
| /l)+l | \*,( | l,l,l,- |
| l)+l | \*,(,/ | l,l,l,- |
| )+l | \*,(,/ | l,l,l,-,l |
| +l | \* | l,l,l,-,l,/ |
| l | + | l,l,l,-,l,/,\* |
|  | + | l,l,l,-,l,/,\*,l |
|  |  | l,l,l,-,l,/,\*,l,+ |

В приложении Г приведена изменённая таблица лексем, отображающая результаты преобразования выражений в польский формат.

**7. Генерация кода**

**7.1 Структура генератора кода**

В языке YVA-2024 генерация кода является заключительным этапом трансляции. Структура генератора кода YVA-2024 представлена на рисунке 7.1.

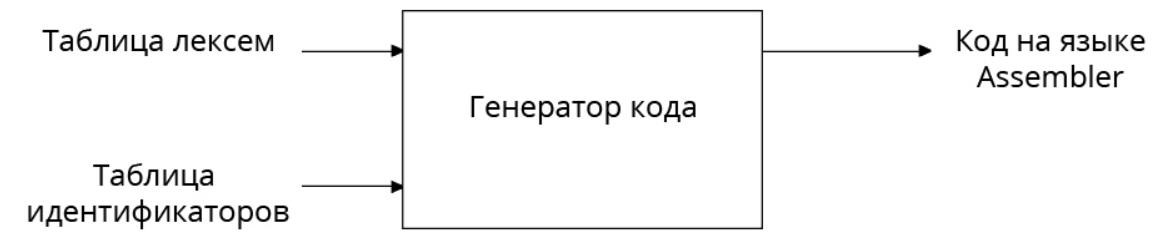


Рисунок 7.1 – Структура генератора кода

Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа в виде обратной польской записи. 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 YVA-2024 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке YVA-2024 | Тип идентификатора на языке ассемблера | Пояснение |
| numb | dword | Хранит целочисленный тип данных. |
| string | byte | Хранит указатель на начало строки. Строка должна завешаться нулевым символом. |
| bool | byte | Хранит логический тип данных |
| char | byte | Хранит указатель на символ, оканчивается нулевым символом |

При представлении переменной в языке Assembler, необходимый тип идентификатора пишется после нее через двоеточие.

**7.3 Статическая библиотека**

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

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

**7.4 Особенности алгоритма генерации кода**

В процессе генерации используются векторы и строки. Отдельные сегменты сначала записываются в строки, а затем отправляются в вектор. В конце работы весь вектор последовательно выводится в файл.

**7.5 Входные параметры генератора кода**

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке YVA-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 Д. Результат работы контрольного примера приведён в приложении Д.

**8. Тестирование транслятора**

**8.1 Общие положения**

В языке YVA-2024, при возникновении ошибки на одном из этапов, генерируется исключение, которое обрабатывается в главной функции. Затем код ошибки и сообщение выводится в консольное окно, а так же записывается в протокол работы.

**8.2 Результаты тестирования**

В таблице 8.1 приведены ошибки возникающие при считывании из файла, а так же на стадии лексического, синтаксического и семантического анализа.

Таблица 8.1 – Результаты тестирования транслятора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| declare string func sayIt() {  writeline "valar morghulis";  return 0;  }; | Ошибка 704: !Семантическая! Нарушены типы данных в выражении или условной конструкции, строка 50, столбец 5 |
| add numb func StrLen(string a);  add numb func ToNumb(numb a);  add numb func Rand(numb n);  add numb func Input(); | Ошибка 706: !Семантическая! Ошибка экспорта: неверные параметры, строка 9, столбец 28 |
| declare numb num = StrLen("0000000");  declare numb num = StrLen("0000000");  declare numb num2 = ToNumb("123"); | Ошибка 700: !Семантическая! Повторное объявление идентификатора, строка 39, столбец 14 |

# 

# **Заключение**

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования YVA-2024 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

* Сформулирована спецификация языка YVA-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 YVA-2024 включает:
* 4 типа данных;
* Поддержка операторов ввода и вывода строки;
* Наличие 4 арифметических операторов для вычисления выражений
* Наличие 7 логических операторов для использования в условиях цикла и условной конструкции
* Поддержка функций; Операторов цикла и условия;
* Наличие библиотеки стандартных функций языка
* 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 Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768с

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

3 Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c

4 Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с

# **Приложение А**

Листинг 1 – Исходный код программы на языке YVA-2024

|  |
| --- |
| declare numb func sayIt() {  writeline "valar morghulis";  return 0;  };  main  {  add numb func StrLen(string a);  add numb func ToNumb(string a);  add numb func Rand(numb n);  add numb func Input();  writeline "Угадайте число от 0 до 10";  writeline "Введите число(0-10)";  declare numb programNumb = Rand(10);  declare bool flag = true;  declare numb guess;  while (flag) [  guess = Input();  if (guess ^ programNumb) [  flag = false;  ];  if (guess > programNumb) [  writeline "Ваше число больше";  ];  if (guess < programNumb) [  writeline "Ваше число меньше";  ];  ];  writeline "Правильно";  writeline programNumb;  declare numb num = StrLen(" ");  declare numb num2 = ToNumb("123");  writeline num;  writeline num2;  declare numb a = 30 \* 2;  writeline a;  declare char sym = 'a';  writeline sym;  a = sayIt();  return 0;  } |

**Приложение Б**

Листинг 1 – Таблица контроля входных символов

|  |
| --- |
| IN::T,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::S,IN::N,IN::F,IN::F,IN::T,IN::F,IN::F,\  IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,\  IN::S,IN::L,IN::D,IN::F,IN::O,IN::L,IN::L,IN::C,IN::L,IN::L,IN::L,IN::L,IN::L,IN::L,IN::F,IN::L,\  IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::L,IN::L,IN::L,IN::L,IN::L,IN::F,\  IN::F,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,\  IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::L,IN::L,IN::L,IN::L,IN::T,\  IN::F,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,\  IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::L,IN::L,IN::L,IN::L,IN::F,\  IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,\  IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,\  IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,\  IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,\  IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,\  IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,\  IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,\  IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F,IN::F, \ |

Листинг 2 – Структура таблицы лексем

|  |
| --- |
| struct LEX  {  IT::IdTable idtable;  LT::LexTable lextable;  LEX(  LT::LexTable \_lextable, IT::IdTable \_idtable) {  idtable = \_idtable;  lextable = \_lextable;  }  LEX() {}  }; |

Листинг 3 – Пример конечного автомата

|  |
| --- |
| FST l\_true(  str,  5,  NODE(1, RELATION('t', 1)),  NODE(1, RELATION('r', 2)),  NODE(1, RELATION('u', 3)),  NODE(1, RELATION('e', 4)),  NODE()  ); |

Листинг 4 – Структура таблицы идентификаторов

|  |
| --- |
| struct Entry  {  int idxfirstLE;  char id[ID\_MAXSIZE];  bool isExternal;  IDDATATYPE iddatatype;  IDTYPE idtype;  union {  unsigned int vint;  char vchar;  bool vbool;  struct  {  char len;  char\* str;  } vstr;  } value;  Entry(int idxfirstLE, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, bool e = false) {  this->isExternal = e;  this->idxfirstLE = idxfirstLE;  this->iddatatype = iddatatype;  this->idtype = idtype;  if (iddatatype == CHR)  {  this->value.vchar = '\0';  }  if (iddatatype == INT)  {  this->value.vint = 0;  }  if (iddatatype == STR)  {  this->value.vstr.len = 0;  this->value.vstr.str = nullptr;  }  if (iddatatype == BOOL)  {  this->value.vbool = false;  }  int len = 0;  for (int i = 0; id[i] != '\0' && i < ID\_MAXSIZE; i++)  {  len++;  this->id[i] = id[i];  }  if (len >= ID\_MAXSIZE)  this->id[ID\_MAXSIZE - 1] = '\0';  else  {  this->id[len] = '\0';  } |

Листинг 5 – Сообщения об ошибках стадии лексического анализа

|  |
| --- |
| ERROR\_ENTRY(120, "!Лексического анализа! Ошибка при разборе токена"),  ERROR\_ENTRY(121, "!Лексического анализа! Используется необъявленный идентификатор"),  ERROR\_ENTRY(122, "!Лексического анализа! Идентификатор не имеет типа"),  ERROR\_ENTRY\_NODEF(123),  ERROR\_ENTRY(124, "!Лексического анализа! Отсутствует точка входа"),  ERROR\_ENTRY(125, "!Лексического анализа! Обнаружена вторая точка входа"), |

|  |
| --- |
|  |

Листинг 6 – Таблица идентификаторов контрольного примера

Листинг 7 – Таблица лексем контрольного примера

|  |
| --- |
| Позиция № строки № столбца Лексема  0 1 1 d -1  1 1 9 t -1  2 1 14 f -1  3 1 19 i 0  4 1 24 ( -1  5 1 25 ) -1  6 1 27 { -1  7 2 2 w -1  8 2 29 l 1  9 2 30 ; -1  10 3 2 r -1  11 3 9 l 2  12 3 10 ; -1  13 4 1 } -1  14 4 2 ; -1  15 6 1 m -1  16 7 1 { -1  17 8 1 a -1  18 8 5 t -1  19 8 10 f -1  20 8 15 i 3  21 8 21 ( -1  22 8 22 t -1  23 8 29 i 4  24 8 30 ) -1  25 8 31 ; -1  26 9 1 a -1  27 9 5 t -1  28 9 10 f -1  29 9 15 i 5  30 9 21 ( -1  31 9 22 t -1  32 9 29 i 6  33 9 30 ) -1  34 9 31 ; -1  35 10 1 a -1  36 10 5 t -1  37 10 10 f -1  38 10 15 i 7  39 10 19 ( -1  40 10 20 t -1  41 10 25 i 8  42 10 26 ) -1  43 10 27 ; -1  44 11 1 a -1  45 11 5 t -1  46 11 10 f -1  47 11 15 i 9  48 11 20 ( -1  49 11 21 ) -1  50 11 22 ; -1  51 13 1 w -1  52 13 38 l 10  53 13 39 ; -1  54 14 1 w -1  55 14 32 l 11  56 14 33 ; -1  57 16 1 d -1  58 16 9 t -1  59 16 14 i 12  60 16 26 v -1  61 16 28 i 7  62 16 32 ( -1  63 16 33 l 13  64 16 35 ) -1  65 16 36 ; -1  66 19 1 d -1  67 19 9 t -1  68 19 14 i 14  69 19 19 v -1  70 19 21 l 15  71 19 25 ; -1  72 20 1 d -1  73 20 9 t -1  74 20 14 i 16  75 20 19 ; -1  76 22 1 u -1  77 22 7 ( -1  78 22 8 i 14  79 22 12 ) -1  80 22 14 [ -1  81 23 2 i 16  82 23 8 v -1  83 23 10 i 9  84 23 15 ( -1  85 23 16 ) -1  86 23 17 ; -1  87 24 2 o -1  88 24 5 ( -1  89 24 6 i 16  90 24 12 b -1  91 24 14 i 12  92 24 25 ) -1  93 24 27 [ -1  94 25 3 i 14  95 25 8 v -1  96 25 10 l 17  97 25 15 ; -1  98 26 2 ] -1  99 26 3 ; -1  100 27 2 o -1  101 27 5 ( -1  102 27 6 i 16  103 27 12 b -1  104 27 14 i 12  105 27 25 ) -1  106 27 27 [ -1  107 28 3 w -1  108 28 32 l 18  109 28 33 ; -1  110 29 2 ] -1  111 29 3 ; -1  112 30 2 o -1  113 30 5 ( -1  114 30 6 i 16  115 30 12 b -1  116 30 14 i 12  117 30 25 ) -1  118 30 27 [ -1  119 31 3 w -1  120 31 32 l 19  121 31 33 ; -1  122 32 2 ] -1  123 32 3 ; -1  124 33 1 ] -1  125 33 2 ; -1  126 35 1 w -1  127 35 22 l 20  128 35 23 ; -1  129 36 1 w -1  130 36 11 i 12  131 36 22 ; -1  132 38 1 d -1  133 38 9 t -1  134 38 14 i 21  135 38 18 v -1  136 38 20 i 3  137 38 26 ( -1  138 38 36 l 22  139 38 37 ) -1  140 38 38 ; -1  141 39 1 d -1  142 39 9 t -1  143 39 14 i 23  144 39 19 v -1  145 39 21 i 5  146 39 27 ( -1  147 39 33 l 24  148 39 34 ) -1  149 39 35 ; -1  150 41 1 w -1  151 41 11 i 21  152 41 14 ; -1  153 42 1 w -1  154 42 11 i 23  155 42 15 ; -1  156 44 1 d -1  157 44 9 t -1  158 44 14 i 25  159 44 16 v -1  160 44 18 l 26  161 44 21 v -1  162 44 23 l 27  163 44 24 ; -1  164 45 1 w -1  165 45 11 i 25  166 45 12 ; -1  167 47 1 d -1  168 47 9 t -1  169 47 14 i 28  170 47 18 v -1  171 47 22 l 29  172 47 23 ; -1  173 48 1 w -1  174 48 11 i 28  175 48 14 ; -1  176 50 1 i 25  177 50 3 v -1  178 50 5 i 0  179 50 10 ( -1  180 50 11 ) -1  181 50 12 ; -1  182 52 1 r -1  183 52 8 l 30  184 52 9 ; -1  185 53 1 } -1 |

**Приложение В**

Листнинг 1 – Грамматика языка YVA-2024

|  |
| --- |
| Greibach greibach(NS('S'), TS('$'),  9,  Rule(NS('S'), GRB\_ERROR\_SERIES + 0,  7,  Rule::Chain(7, TS('m'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}')),  Rule::Chain(6, TS('m'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}')),  Rule::Chain(15, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(14, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(14, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(6, TS('m'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'))  ),  Rule(NS('N'), GRB\_ERROR\_SERIES + 1,  18,  Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),  Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')),  Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('v'), NS('E'), TS(';')),  Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('v'), NS('E'), TS(';'), NS('N')),  Rule::Chain(4, TS('i'), TS('v'), NS('E'), TS(';')),  Rule::Chain(5, TS('i'), TS('v'), NS('E'), TS(';'), NS('N')),  Rule::Chain(9, TS('a'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';'), NS('N')),  Rule::Chain(8, TS('a'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';')),  Rule::Chain(8, TS('a'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS(';'), NS('N')),  Rule::Chain(7, TS('a'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS(';')),  Rule::Chain(4, TS('w'), TS('i'), TS(';'), NS('N')),  Rule::Chain(3, TS('w'), TS('i'), TS(';')),  Rule::Chain(4, TS('w'), TS('l'), TS(';'), NS('N')),  Rule::Chain(3, TS('w'), TS('l'), TS(';')),  Rule::Chain(8, TS('o'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), TS(';')),  Rule::Chain(9, TS('o'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), TS(';'), NS('N')),  Rule::Chain(8, TS('u'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), TS(';')),  Rule::Chain(9, TS('u'), TS('('), NS('B'), TS(')'), TS('['), NS('N'), TS(']'), TS(';'), NS('N'))  ),  Rule(NS('E'), GRB\_ERROR\_SERIES + 2,  16,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(2, TS('n'), TS('i')),  Rule::Chain(2, TS('n'), 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(3, TS('n'), TS('i'), NS('M')),  Rule::Chain(3, TS('n'), TS('l'), NS('M')),  Rule::Chain(4, TS('n'), TS('('), NS('E'), TS(')')),  Rule::Chain(5, TS('n'), TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M'))  ),  Rule(NS('M'), GRB\_ERROR\_SERIES + 3,  4,  Rule::Chain(2, TS('v'), NS('E')),  Rule::Chain(4, TS('v'), TS('('), NS('E'), TS(')')),  Rule::Chain(5, TS('v'), TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(3, TS('v'), NS('E'), NS('M'))  ),  Rule(NS('F'), GRB\_ERROR\_SERIES + 4,  5,  Rule::Chain(2, TS('t'), TS('i')),  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F')),  Rule::Chain(1, TS('i')),  Rule::Chain(3, TS('i'), TS(','), NS('F')),  Rule::Chain(3, TS('l'), TS(','), NS('F'))  ),  Rule(NS('W'), GRB\_ERROR\_SERIES + 5,  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('B'), GRB\_ERROR\_SERIES + 6,  6,  Rule::Chain(3, TS('i'), TS('b'), TS('i')),  Rule::Chain(3, TS('i'), TS('b'), TS('l')),  Rule::Chain(3, TS('l'), TS('b'), TS('i')),  Rule::Chain(3, TS('l'), TS('b'), TS('l')),  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l'))  )  ); |

Листнинг 2 – Структура магазинного автомата

|  |
| --- |
| 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];  GRBALPHABET\* lenta;  short lenta\_position;  short nrule;  short nrulechain;  short lenta\_size;  GRB::Greibach greibach;  LEX::LEX lex;  MFSTSTACK st;  bool shallWrite;  use\_container<std::stack<MfstState>> storestate;  Mfst();  Mfst(  LEX::LEX plex,  GRB::Greibach pgreibach,  bool shouldWrite  );  char\* getCSt(char\* buf);  char\* getCLenta(char\* buf, short pos, short n = 25);  char\* getDiagnosis(short n, char\* buf);  bool savestate(const Log::LOG& log);  bool reststate(const Log::LOG& log);  bool push\_chain(  GRB::Rule::Chain chain  );  RC\_STEP step(const Log::LOG& log);  bool start(const Log::LOG& log);  bool savediagnosis(  RC\_STEP pprc\_step  );  void printrules(const Log::LOG& lo);  struct Deducation  {  short size;  short\* nrules;  short\* nrulechains;  Deducation() { size = 0; nrules = 0; nrulechains = 0; };  } deducation;  bool savededucation();  }; |

|  |
| --- |
|  |

Листинг 3 – Разбор исходного кода синтаксическим анализатором

|  |
| --- |
|  |
| Листинг 4 – Структура грамматики Грейбах  struct Greibach  {  short size;  GRBALPHABET startN;  GRBALPHABET stbottomT;  Rule\* rules;  Greibach() { size = 0; startN = 0; stbottomT = 0; rules = 0; };  Greibach(GRBALPHABET pstartN, GRBALPHABET pstbootomT, short psize, Rule r, ...);  short getRule(GRBALPHABET pnn, Rule& prule);  Rule getRule(short n);  }; |

Листинг 5 – Сообщения об ошибках стадии синтаксического анализа

|  |
| --- |
| ERROR\_ENTRY(600, "!Синтаксическая! Неверная структура программы"),  ERROR\_ENTRY(601, "!Синтаксическая! Ошибочный оператор"),  ERROR\_ENTRY(602, "!Синтаксическая! Ошибка в выражении"),  ERROR\_ENTRY(603, "!Синтаксическая! Ошибка в параметрах функции"),  ERROR\_ENTRY(604, "!Синтаксическая! Ошибка в параметрах вызываемой функции"),  ERROR\_ENTRY(605, "!Синтаксическая! Ошибка знака в выражении"),  ERROR\_ENTRY(606, "!Синтаксическая! Ошибка синтаксического анализа"),  ERROR\_ENTRY(607, "!Синтаксическая! Ошибка условной конструкции"),  ERROR\_ENTRY\_NODEF(608),  ERROR\_ENTRY(609, "!Синтаксическая! Обнаружена синтаксическая ошибка (Log)"), |

|  |
| --- |
|  |

# **Приложение Г**

Листинг 1 – Сообщения об ошибках стадии семантического анализа

|  |
| --- |
| ERROR\_ENTRY(700, "!Семантическая! Повторное объявление идентификатора"),  ERROR\_ENTRY(701, "!Семантическая! Ошибка в возвращаемом значении"),  ERROR\_ENTRY(702, "!Семантическая! Ошибка в передаваемых значениях в функции: количество параметров не совпадает"),  ERROR\_ENTRY(703, "!Семантическая! Ошибка в передаваемых значениях в функции: типы параметров не совпадают"),  ERROR\_ENTRY(704, "!Семантическая! Нарушены типы данных в выражении или условной конструкции"),  ERROR\_ENTRY(705, "!Семантическая! Ошибка экспорта: в библиотеке нет такой функции"),  ERROR\_ENTRY(706, "!Семантическая! Ошибка экспорта: неверные параметры"),  ERROR\_ENTRY(707, "!Семантическая! Ошибка экспорта: ошибочный тип возвращаемого значения"),  ERROR\_ENTRY(708, "!Семантическая! Для строк операторы запрещены"),  ERROR\_ENTRY(709, "!Семантическая! Ошибочные параметры условной конструкции: строки не могут быть параметрами условной конструкции"),  ERROR\_ENTRY(710, "!Семантическая! Ошибочный оператор: для типа char разрешены только операции + и -"),  ERROR\_ENTRY(711, "!Семантическая! Для логических переменных использование арифметических и побитовых операторов запрещено"),  ERROR\_ENTRY(712, "!Семантическая! Количество параметров не может быть больше 10"), |

Листинг 2 – Программная реализация механизма преобразования в Обратную польскую запись

|  |
| --- |
| bool PolishNotation(int lextable\_pos, LT::LexTable& lextable, IT::IdTable& idtable)  {  container<std::stack<char>> stack;  std::string PolishString;  std::vector<int> ids;  int operators\_count = 0, operands\_count = 0, iterator = 0, right\_counter = 0, left\_counter = 0, params\_counter = 0;  for (int i = lextable\_pos; i < lextable.size; i++, iterator++) {  char lexem = lextable.table[i].lexema;  char data = lextable.table[i].data;  size\_t stack\_size = stack.size();  if (idtable.table[lextable.table[i].idxTI].idtype == IT::IDTYPE::F) {  stack.push('@');  operands\_count--;  }  switch (lexem) {  case LEX\_OPERATOR:  {  if (!stack.empty() && stack.top() != LEX\_LEFTHESIS) {  while (!stack.empty() && get\_priority(data) <= get\_priority(stack.top())) {  PolishString += stack.top();  stack.pop();  }  }  stack.push(data);  operators\_count++;  break;  }  case LEX\_INVERSION:  {  stack.push(data);  break;  }  case LEX\_COMMA:  {  while (!stack.empty()) {  if (stack.top() == LEX\_LEFTHESIS)  break;  PolishString += stack.top();  stack.pop();  }  operands\_count--;  break;  }  case LEX\_LEFTHESIS:  {  left\_counter++;  stack.push(lexem);  break;  }  case LEX\_RIGHTHESIS:  {  right\_counter++;  if (!find\_elem(stack, stack\_size, LEX\_LEFTHESIS))  return false;  while (stack.top() != LEX\_LEFTHESIS) {  PolishString += stack.top();  stack.pop();  }  stack.pop();  if (!stack.empty() && stack.top() == '@') {  PolishString += stack.top() + toString(params\_counter - 1);  params\_counter = 0;  stack.pop();  }  break;  }  case LEX\_SEMICOLON:  {  if (operators\_count != 0 && operands\_count != 0)  if ((!stack.empty() && (stack.top() == LEX\_RIGHTHESIS || stack.top() == LEX\_LEFTHESIS))  || right\_counter != left\_counter || operands\_count - operators\_count != 1)  return false;  while (!stack.empty()) {  PolishString += stack.top();  stack.pop();  }  fixIt(lextable, PolishString, iterator, lextable\_pos, ids);  return true;  break;  }  case LEX\_ID: {  if (std::find(stack.c.begin(), stack.c.begin(), '@') != stack.c.end())  params\_counter++;  PolishString += lexem;  if (lextable.table[i].idxTI != LT\_TI\_NULLIDX)  ids.push\_back(lextable.table[i].idxTI);  operands\_count++;  if (!stack.empty() && stack.top() == '~')  {  PolishString += stack.top();  stack.pop();  }  break;  }  case LEX\_LITERAL: {  if (std::find(stack.c.begin(), stack.c.begin(), '@') != stack.c.end())  params\_counter++;  PolishString += lexem;  if (lextable.table[i].idxTI != LT\_TI\_NULLIDX)  ids.push\_back(lextable.table[i].idxTI);  operands\_count++;  if (!stack.empty() && stack.top() == '~')  {  PolishString += stack.top();  stack.pop();  }  break;  }  }  }  return true;  } |

Листинг 3 – Результат выполнения польской записи

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **l**  **il@1**  **l**  **i@0**  **l**  **il@1**  **il@1**  **ll\***  **l**  **i@0** |  **Приложение Д** Листинг 1 – Результат генерации кода контрольного примера в Ассемблер   |  | | --- | | .586  .model flat, stdcall  includelib libucrt.lib  includelib kernel32.lib  includelib ../Debug/YVA-2024Lib.lib  ExitProcess PROTO :DWORD  StrLen PROTO :DWORD  ToNumb PROTO :DWORD  Rand PROTO :DWORD  Input PROTO  writeint PROTO :DWORD  writechar PROTO :BYTE  writestr PROTO :DWORD  writebool PROTO :BYTE  writelineint PROTO :DWORD  writelinechar PROTO :BYTE  writelinestr PROTO :DWORD  writelinebool PROTO :BYTE  StrLen PROTO :DWORD  ToNumb PROTO :DWORD  Rand PROTO :DWORD  .stack 4096  .const  divideOnZeroExeption BYTE "Попытка деления на ноль.", 0 ;STR, для вывода ошибки при делении на ноль  sayIt\_$LEX1 BYTE "valar morghulis", 0 ;STR  sayIt\_$LEX2 DWORD 0 ;int  main$LEX3 BYTE "Угадайте число от 0 до 10", 0 ;STR  main$LEX4 BYTE "Введите число(0-10)", 0 ;STR  main$LEX5 DWORD 10 ;int  main$LEX6 BYTE 1 ;BOOL  main$LEX7 BYTE 0 ;BOOL  main$LEX8 BYTE "Ваше число больше", 0 ;STR  main$LEX9 BYTE "Ваше число меньше", 0 ;STR  main$LEX10 BYTE "Правильно", 0 ;STR  main$LEX11 BYTE "0000000", 0 ;STR  main$LEX12 BYTE "123", 0 ;STR  main$LEX13 DWORD 30 ;int  main$LEX14 DWORD 2 ;int  main$LEX15 BYTE 'a' ;CHR  main$LEX16 DWORD 0 ;int  .data  mainprogramNum DWORD 0 ;int  mainflag BYTE 0 ;BOOL  mainguess DWORD 0 ;int  mainnum DWORD 0 ;int  mainnum2 DWORD 0 ;int  maina DWORD 0 ;int  mainsym BYTE 0 ;CHR  .code  $sayIt PROC uses ebx ecx edi esi  push offset sayIt\_$LEX1  CALL writelinestr  mov eax, sayIt\_$LEX2  ret  $sayIt ENDP  main PROC  push offset main$LEX3  CALL writelinestr  push offset main$LEX4  CALL writelinestr  ; String #16 :ivil@1  invoke Rand, main$LEX5  push eax ;результат функции  pop mainprogramNum  ; String #19 :ivl  movzx eax, main$LEX6  push eax  pop eax  mov mainflag, al  While76Start:  movzx eax, mainflag  mov ebx, 1  cmp eax, ebx  jne While76End  ; String #23 :ivi@0  invoke Input  push eax ;результат функции  pop mainguess  If88Start:  mov eax, mainguess  mov ebx, mainprogramNum  cmp eax, ebx  jne If88End  ; String #25 :ivl  movzx eax, main$LEX7  push eax  pop eax  mov mainflag, al  If88End:  If101Start:  mov eax, mainguess  mov ebx, mainprogramNum  cmp eax, ebx  jle If101End  push offset main$LEX8  CALL writelinestr  If101End:  If113Start:  mov eax, mainguess  mov ebx, mainprogramNum  cmp eax, ebx  jge If113End  push offset main$LEX9  CALL writelinestr  If113End:  jmp While76Start  While76End:  push offset main$LEX10  CALL writelinestr  push eax  mov eax, mainprogramNum  push eax  CALL writelineint  pop eax  ; String #38 :ivil@1  invoke StrLen, offset main$LEX11  push eax ;результат функции  pop mainnum  ; String #39 :ivil@1  invoke ToNumb, offset main$LEX12  push eax ;результат функции  pop mainnum2  push eax  mov eax, mainnum  push eax  CALL writelineint  pop eax  push eax  mov eax, mainnum2  push eax  CALL writelineint  pop eax  ; String #44 :ivllv  push main$LEX13  push main$LEX14  pop ebx  pop eax  mul ebx  push eax  pop maina  push eax  mov eax, maina  push eax  CALL writelineint  pop eax  ; String #47 :ivl  movzx eax, main$LEX15  push eax  pop eax  mov mainsym, al  push eax  movzx eax, mainsym  push eax  CALL writelinechar  pop eax  ; String #50 :ivi@0  invoke $sayIt  push eax ;результат функции  pop maina  mov eax, main$LEX16  jmp endPoint  div\_by\_0:  push offset divideOnZeroExeption  CALL writestr  endPoint:  invoke ExitProcess, eax  main ENDP  end main | |