МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Программной инженерии

Специальность Программное обеспечение информационных технологий

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора NVA-2020»

Выполнил студент Новик Виктория Александровна

(Ф.И.О.)

Руководитель проекта преп.-стаж. Котович Дмитрий Витальевич

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

Заведующий кафедрой к.т.н., доц. Пацей Наталья Владимировна

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

Консультант преп.-стаж. Котович Дмитрий Витальевич

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

Курсовой проект защищен с оценкой

Минск 2020

Содержание

[Введение 5](#_Toc58708507)

[1. Спецификация языка программирования 6](#_Toc58708508)

[1.1. Характеристика языка программирования 6](#_Toc58708509)

[1.2. Определение алфавита языка программирования 6](#_Toc58708510)

[1.3. Применяемые сепараторы 7](#_Toc58708511)

[1.4. Применяемые кодировки 7](#_Toc58708512)

[1.5. Типы данных 7](#_Toc58708513)

[1.6. Преобразование типов данных 8](#_Toc58708514)

[1.7. Идентификаторы 9](#_Toc58708515)

[1.8. Литералы 9](#_Toc58708516)

[1.9. Объявление данных 9](#_Toc58708517)

[1.10. Инициализация данных 10](#_Toc58708518)

[1.11. Инструкции языка 10](#_Toc58708519)

[1.12. Операции языка 10](#_Toc58708520)

[1.13. Выражение и их вычисление 11](#_Toc58708521)

[1.14. Конструкции языка 11](#_Toc58708522)

[1.15. Область видимости переменных 12](#_Toc58708523)

[1.16. Семантические проверки 12](#_Toc58708524)

[1.17. Распределение оперативной памяти на этапе выполнения 13](#_Toc58708525)

[1.18. Стандартная библиотека и её состав 13](#_Toc58708526)

[1.19. Ввод и вывод данных 14](#_Toc58708527)

[1.20. Точка входа 14](#_Toc58708528)

[1.21. Препроцессор 14](#_Toc58708529)

[1.22. Соглашения о вызовах 14](#_Toc58708530)

[1.23. Объектный код 15](#_Toc58708531)

[1.24. Классификация сообщений транслятора 15](#_Toc58708532)

[1.25. Контрольный пример 15](#_Toc58708533)

[2. Структура транслятора 16](#_Toc58708534)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 16](#_Toc58708535)

[2.2 Перечень входных параметров транслятора 17](#_Toc58708536)

[2.3 Протоколы, формируемые транслятором 17](#_Toc58708537)

[3 Разработка лексического анализатора 18](#_Toc58708538)

[3.1 Структура лексического анализатора 18](#_Toc58708539)

[3.2 Контроль входных символов 19](#_Toc58708540)

[3.3 Удаление избыточных символов 19](#_Toc58708541)

[3.4 Перечень ключевых слов 20](#_Toc58708542)

[3.5 Основные структуры данных 22](#_Toc58708543)

[3.6 Принцип обработки ошибок 23](#_Toc58708544)

[3.7 Структура и перечень сообщений лексического анализатора 23](#_Toc58708545)

[3.8 Параметры лексического анализатора 24](#_Toc58708546)

[3.9 Алгоритм лексического анализа 24](#_Toc58708547)

[3.10 Контрольный пример 24](#_Toc58708549)

[4. Разработка синтаксического анализатора 25](#_Toc58708550)

[4.1 Структура синтаксического анализатора 25](#_Toc58708551)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 25](#_Toc58708552)

[4.3 Построение конечного магазинного автомата 26](#_Toc58708553)

[4.4 Основные структуры данных 27](#_Toc58708554)

[4.5 Описание алгоритма синтаксического разбора 27](#_Toc58708555)

[4.6 Структура и перечень сообщений синтаксического анализатора 27](#_Toc58708556)

[4.7. Параметры синтаксического анализатора и режимы его работы 27](#_Toc58708557)

[4.8. Принцип обработки ошибок 28](#_Toc58708558)

[4.9. Контрольный пример 28](#_Toc58708559)

[5 Разработка семантического анализатора 29](#_Toc58708560)

[5.1 Структура семантического анализатора 29](#_Toc58708561)

[5.2 Функции семантического анализатора 29](#_Toc58708562)

[5.3 Структура и перечень сообщений семантического анализатора 29](#_Toc58708563)

[5.4 Принцип обработки ошибок 30](#_Toc58708564)

[5.5 Контрольный пример 30](#_Toc58708565)

[6 Вычисление выражений 31](#_Toc58708566)

[6.1 Выражения, допускаемые языком 31](#_Toc58708567)

[6.2 Польская запись и принцип её построения 31](#_Toc58708568)

[6.3 Программная реализация обработки выражений 31](#_Toc58708569)

[6.4 Контрольный пример 32](#_Toc58708570)

[7. Генерация кода 33](#_Toc58708571)

[7.1 Структура генератора кода 33](#_Toc58708572)

[7.2 Представление типов данных в оперативной памяти 33](#_Toc58708573)

[7.3 Статическая библиотека 34](#_Toc58708574)

[7.4 Особенности алгоритма генерации кода 34](#_Toc58708575)

[7.5 Входные параметры генератора кода 34](#_Toc58708576)

[7.6 Контрольный пример 35](#_Toc58708577)

[8. Тестирование транслятора 36](#_Toc58708578)

[8.1 Общие положения 36](#_Toc58708579)

[8.2 Результаты тестирования 36](#_Toc58708580)

[Заключение 37](#_Toc58708581)

[Список использованных источников 38](#_Toc58708582)

[Приложение А 39](#_Toc58708583)

[Приложение Б 41](#_Toc58708585)

[Приложение В 42](#_Toc58708586)

[Приложение Г 46](#_Toc58708587)

[Приложение Д 48](#_Toc58708588)

[Приложение Е 52](#_Toc58708589)

[Приложение Ж 55](#_Toc58708590)

**Введение**

Целью курсового проекта является разработка компилятора для своего языка программирования: NVA-2020. Этот язык программирования предназначен для выполнения простейших операций и арифметических действий над числами.

Компилятор – это программа, задачей которого является перевод программы, написанной на одном из языков программирования (в моём случае на NVA-2020) в программу на язык ассемблера.

Компиляция состоит из двух частей: анализа и синтеза. Анализ – это разбиение исходной программы на составные части и создание ее промежуточного представления. Синтез – конструирование требуемой целевой программы из промежуточного представления. В данном курсовом проекте мой исходный код транслируется на язык ассемблера. Мой компилятор состоит из следующих составных частей:

– лексический и семантический анализаторы;

– синтаксический анализатор;

– генератор исходного кода на языке ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

– разбработка спецификации языка программирования;

– разбратка структуры транслятора;

– разработка лексического и семантического анализаторов;

– разработка синтаксического анализатора;

– преобразование выражений;

– генерация кода на язык ассемблера;

– тестирование транслятора.

Решения каждой из поставленных задач будут приведены в соответствующих главах курсового проекта, а именно:

В первой главе работы определена спецификация языка программирования, т.е. описан синтаксис и семантика языка.

Во второй главе работы представлена структура транслятора, т.е. перечислены компоненты транслятора, их назначение и принципы взаимодействия, перечень входных параметров, перечень протоколов, формируемых транслятором и их содержимое.

В третьей главе работы показана разработка лексического анализатора, порождающего таблицы лексем и идентификаторов.

В четвертой главе работы рассказывается о синтаксическом анализаторе, который выполняет синтаксический разбор текста с распечаткой протокола разбора и дерева разбора на основе таблицы лексем.

В пятой главе описан семантический анализатор, показана его работа (распечатка выданных сообщений в трёх примерах на разных этапах).

В шестой главе решены вопросы преобразования выражений, допускаемых языком и приведена часть протокола для контрольного примера, отображающая результаты преобразования выражений в польский формат.

В седьмой главе представлена генерация кода, где из промежуточного представления порождается код на целевом языке.

В восьмой главе описывается тестирование транслятора

# **1. Спецификация языка программирования**

## **Характеристика языка программирования**

Язык программирования NVA-2020 является процедурным, строго типизированным, не объектно-ориентированным, компилируемым.

Процедурный язык программирования — язык высокого уровня, в котором используется метод разбиения программ на отдельные связанные между собой модули — подпрограммы (процедуры и функции).

Строго типизированный язык программирования — язык, в котором переменные привязаны к конкретным типам данных. Язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования.

Объектно-ориентированный язык программирования — язык, построенный на принципах объектно-ориентированного программирования. В основе концепции объектно-ориентированного программирования лежит понятие объекта — некой сущности, которая объединяет в себе поля (данные) и методы (выполняемые объектом действия).

Компилируемый язык программирования — язык программирования, исходный код которого преобразуется компилятором в исходный код на другом языке программирования.

* 1. **Определение алфавита языка программирования**

В алфавите языка NVA-2020 используется кодировка ASCII, таблица которой

представлена на рисунке 1.1.



Рис. 1.1 – алфавит входных символов

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

* 1. **Применяемые сепараторы**

Применяемые сепараторы в языке NVA-2020, приведены в таблице 1.1.

Таблица 1.1 — Применяемые сепараторы

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

## **Применяемые кодировки**

Для написания программ язык NVA-2020 использует кодировку ASCII, содержащую английский алфавит, а также некоторые специальные символы, такие как { } [ ] ( ) , ; ~ + - / \* > < & !.

ASCII (American standard code for information interchange) — название таблицы (кодировки, набора), в которой некоторым распространённым печатным и непечатным символам сопоставлены числовые коды. Таблица была разработана и стандартизована в [США](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B5%D0%B4%D0%B8%D0%BD%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%A8%D1%82%D0%B0%D1%82%D1%8B_%D0%90%D0%BC%D0%B5%D1%80%D0%B8%D0%BA%D0%B8), в 1963 году.

## **Типы данных**

В языке NVA-2020 есть 3 типа данных: знаковый целочисленный, строковый и символьный. Описание типов данных, предусмотренных в данным языке представлено в таблице 1.2.

Таблица 1.2 – Типы данных языка NVA-2020

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| integer | Фундаментальный тип данных. Используется для работы с числовыми значениями. В памяти занимает 4 байта. Максимальное значение: 2147483647.  Минимальное значение: -2147483647.  Инициализация по умолчанию: значение 0.  Поддерживаемые операции:  + (бинарный) – оператор сложения;  - (бинарный) – оператор вычитания;  \* (бинарный) – оператор умножения;  / (бинарный) – оператор деления;  %(бинарный) – оператор остаток от деления;  = (бинарный) – оператор присваивания.  В качестве условия цикла/условного оператора поддерживаются следующие логические операции:  > (бинарный) – оператор «больше»;  <(бинарный) – оператор «меньше»;  & (бинарный) – оператор проверки на равенство;  ! (бинарный) – оператор проверки на неравенство. |
| string | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 255.  Инициализация по умолчанию: строка нулевой длины “”. Операции над данными строкового типа: присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, а также использование библиотечных функций. |
| symbol | Фундаментальный тип данных. Используется для работы с символами, каждый из которых занимает 1 байт. Максимальное количество символов – 1.  Инициализация по умолчанию: символ нулевой длины “”. Операции над данными строкового типа: присваивание символьному идентификатору значения другого символьного идентификатора, символьного литерала или значения символьной функции, а также использование библиотечных функций. |

## **Преобразование типов данных**

В языке программирования NVA-2020 преобразование типов данных не поддерживается, т.е. язык является строго типизированным.

## **Идентификаторы**

Общее количество идентификаторов ограничено максимальным размером таблицы идентификаторов. Идентификаторы должны начинаться только с символов латинского алфавита, могут содержать цифры. Максимальная длина идентификатора равна восьми символам. Идентификаторы, объявленные внутри функционального блока, получают префикс, идентичный имени функции, внутри которой они объявлены. Префикс занимает 8 дополнительных символов. В случае превышения заданной длины, идентификаторы усекаются до длины, равной 16 символов (8 символов на имя идентификатора, 8 символов на префикс). Данные правила действуют для всех типов идентификаторов. Идентификаторы не должны совпадать с ключевыми словами.

## **Литералы**

С помощью литералов осуществляется инициализация переменных. Типы литералов языка NVA-2020 представлены в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Литералы целого типа | Последовательность цифр 0…9 с предшествующим знаком минус или без него (знак минус не должен отделяться пробелом) |
| Строковые литералы | Набор символов (от 1 до 255), заключённых в “ ” |
| Символьные литералы | Символ , заключенный в ' ' |

## **Объявление данных**

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

Пример объявления числового типа с инициализацией:

**type integer** num1 = -1

Пример объявления переменной строкового типа с инициализацией:

**type string** str1= “hello world”

Пример объявления переменной символьного типа с инициализацией:

**type symbol** ch1= 'c'

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

## **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 предусмотрены значения по умолчанию: значение 0 для типа **integer** и строка длины 0 (“”) для типа **string** и **symbol**.

## **Инструкции языка**

Инструкции языка NVA-2020 представлена в таблице 1.4.

Таблица 1.4 – Инструкции языка

|  |  |
| --- | --- |
| Инструкция | Форма записи |
| Объявление переменной | **type** <тип данных> <идентификатор>. |
| Объявление переменной с явной инициализацией | **type** <тип данных> <идентификатор> = <значение>|<выражение>;  Значение – литерал, идентификатор, вызов функции соответствующего типа данных |
| Объявление функции | <тип данных> **function** <идентификатор> (<тип данных> <идентификатор>, …)  {  / программный блок /  **return** <идентификатор/литерал>.  } |
| Вызов функции | <идентификатор> (<идентификатор>, ...) |
| Присвоение значения | <идентификатор> = <значение>; |
| Вывод данных | **write** <идентификатор/литерал>; |
| Вывод данных с переходом на новую строку | **writeline** <идентификатор/литерал>; |
| Возвращаемое значение | **return** <литерал/идентификатор>. |

## **Операции языка**

В языке NVA-2020 предусмотрены следующие операции с данными. Приоритетность операции умножения выше приоритета операций сложения и вычитания. Для установки наивысшего приоритета используются круглые скобки. Операции языка представлены в таблице 1.5.

Таблица 1.5 — Операции языка

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические операции языка | + – сложение  - – вычитание  \* – умножение  / – деление  % – остаток от деления  = – присваивание |
| Логические | > – больше  < – меньше  & – проверка на равенство  ! – проверка на неравенство |

## **Выражение и их вычисление**

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

* Допускается использовать скобки для смены приоритета операций;
* Выражение записывается в строку без переносов;
* Использование двух подряд идущих операторов не допускается. (за исключением оператор “-”, идущего после любого другого оператора и предшествующему литералу);
* 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

## **Конструкции языка**

Программа на языке NVA-2020 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода.

Программные конструкции языка представлены в таблице 1.6.

Таблица 1.6 — Основные конструкции языка

|  |  |
| --- | --- |
| Конструкция | Реализация |
| 1 | 2 |
| Главная функция | main  {  …} |

Продолжение таблицы 1.6

|  |  |  |
| --- | --- | --- |
| 1 | | 2 |
| Внешняя функция | | <тип данных> function <идентификатор> (<тип> <идентификатор>, ...)  {…  return <идентификатор/литерал>;  } |
| Условная конструкция | is <идентификатор1> <оператор> <идентификатор2>  true [ … ]  false [ … ]  <идентификатор1>, <идентификатор2> - идентификаторы или литералы. <оператор> - один из операторов сравнения ( > < & ! ), устанавливающий отношение между двумя операндами и организующий условие данной конструкции. Причем с идентификаторами и литералами строкового и символьного типа, могут использоваться только операторы сравнения (& !) При истинности условия выполняется код внутри блока true, иначе – код внутри блока false. Любой из блоков true, false может отсутствовать, но не оба блока одновременно. При отсутствии одного из блоков, в зависимости от истинности или ложности условия программа может как выполнить один из заявленных блоков, так и передать управление инструкции, следующей в коде. | |
| Цикл | while <идентификатор1> <оператор> <идентификатор2>  do [ … ]  Цикл (операторы внутри блока do) выполняется, пока истинно условие “<идентификатор1> <оператор> <идентификатор2>”, имеющее тот же смысл, что и в примере выше. | |

## **Область видимости переменных**

Область видимости: сверху вниз (как и в С++). Переменные, объявленные в одной функции, недоступны в другой. Все объявления и операции с переменными происходят внутри какого-либо блока. Каждая переменная или параметр функции получают префикс – название функции, внутри которой они находятся.

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

## **Семантические проверки**

* В языке программирования NVA-2020 выполняются следующие семантические проверки:
* Наличие функции main – точки входа в программу;
* Единственность точки входа;
* Переопределение идентификаторов;
* Использование идентификаторов без их объявления;
* Проверка соответствия типа функции и возвращаемого параметра;
* Правильность передаваемых в функцию параметров: количество, типы;
* Правильность строковых выражений;
* Превышение размера строковых и числовых литералов;
* Правильность составленного условия цикла/условного оператора.

## **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.

## **Стандартная библиотека и её состав**

В языке NVA-2020 предусмотрена стандартная библиотека. Функции, входящие в состав библиотеки, описаны в табл. 1.7. Стандартная библиотека подключается автоматически на этапе генерации кода.

Таблица 1.7 - Функции стандартной библиотеки языка NVA-2020

|  |  |
| --- | --- |
| Функция | Описание |
| random (integer a) | Данная функция целочисленного типа принимает 1 параметр. Функция генерирует случайное число в диапазоне от -a до а и возвращает это число. |
| pow (integer a, integer b) | Данная функция целочисленного типа принимает два целочисленных параметра. Функция возводит число a в степень b и возвращает результат. |
| length (string a) | Данная функция целочисленного типа принимает 1 параметр.  Функция возвращает длину строки a. |

Стандартная библиотека написана на языке С++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю. Для вывода предусмотрен оператор **write** и **writeline**. Эти функции представлены в таблице 1.8.

Таблица 1.8 Дополнительные функции стандартной библиотеки

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void outnum(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| void outstr(char\* line) | Функция для вывода в стандартный поток значения строкового, символьного идентификатора/литерала. |
| void outstrline(char\* line) | Функция для вывода в стандартный поток значения строкового, символьного идентификатора/литерала и перехода на новую строку. |
| void outnumline(int value) | Функция для вывода в стандартный поток значения целочисленного идентификатора/литерала и перехода на новую строку. |
| void system\_pause() | Функция которая запускается по окончанию программы и требует нажать любой символ для завершения программы |

## **Ввод и вывод данных**

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

Функции, управляющие выводом данных, реализованы на языке С++ и вызываются из транслированного кода, конечному пользователю недоступны. Пользовательские команды **write** и **writeline** в транслированном коде будут заменены вызовом нужных библиотечных функций. Библиотека, содержащая нужные процедуры, подключается на этапе генерации кода.

## **Точка** **входа**

В языке NVA-2020 каждая программа должна содержать главную функцию (точку входа) **main**, с первой инструкции которой начнётся последовательное выполнение команд программы.

## **Препроцессор**

Препроцессор, принимающий и выдающий некоторые данные на вход транслятору, в языке NVA-2020 отсутствует.

## **Соглашения о вызовах**

В языке вызов функций происходит по соглашению о вызовах stdcall. Особенности stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

## **Объектный код**

NVA-2020 транслируется в язык ассемблера, а затем - в объектный код

## **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 1.9, а также в приложении А.

Таблица 1.9 Классификация ошибок

|  |  |
| --- | --- |
| Номера ошибок | Характеристика |
| 0 – 200 | Системные ошибки |
| 200 – 299 | Ошибки лексического анализа |
| 300 – 399 | Ошибки семантического анализа |
| 600 – 699 | Ошибки синтаксического анализа |
| 400 – 499, 700 – 999 | Зарезервированные коды ошибок |

## **Контрольный пример**

Контрольный пример демонстрирует главные особенности языка NVA-2020: 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**

Транслятор преобразует программу, написанную на языке NVA-2020 в программу на языке ассемблера. Компонентами транслятора являются лексический, синтаксический и семантический анализаторы, а также генератор кода на язык ассемблера. Принцип их взаимодействия представлен на рисунке 2.1.



Рис. 2.1 – структура транслятора языка программирования NVA-2020

Лексический анализ – первая фаза трансляции. Назначением лексического анализатора является нахождение ошибок лексики языка и формирование таблицы лексем и таблицы идентификаторов. Подробнее описан в 3 главе.

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

Семантический анализ в свою очередь является проверкой исходной программы на семантическую согласованность с определением языка, т.е. проверяет правильность текста исходной программы с точки зрения семантики. Подробное описание представлено в 5 главе.

Генератор кода – этап транслятора, выполняющий генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. Генератор кода принимает на вход таблицы идентификаторов и лексем и транслирует код на языке NVA-2020, прошедший все предыдущие этапы, в код на языке Ассемблера. Более полно описан в главе 7.

## **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 NVA-2020

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке NVA-2020, 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> | Выходной файл – результат работы транслятора. Содержит исходный код на языке асемблера. | Значение по умолчанию:  <имя in-файла>.asm |

## **2.3 Протоколы, формируемые транслятором**

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

Таблица 2.2 Протоколы, формируемые транслятором языка NVA-2020

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования NVA-2020 . 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, заданный параметром "-out:" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

# **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,. Лексический анализатор 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 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, структура лексического анализатора представлена на рисунке 3.1.



Рис. 3.1 – структура лексического анализатора

## **3.2 Контроль входных символов**

Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. Таблица входных символов представлена на рисунке 3.2, категории входных символов представлены в таблице 3.1.

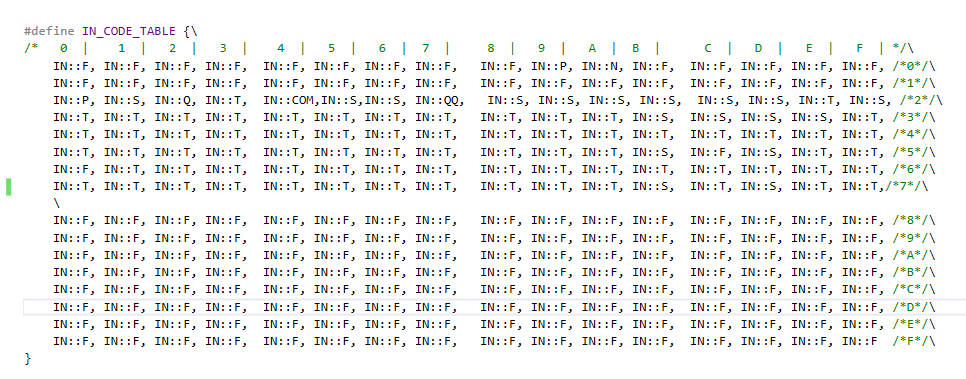


Рисунок 3.2. – таблица контроля входных символов

Таблица 3.1 Соответствие символов и их значений в таблице

|  |  |
| --- | --- |
| Значение в таблице входных символов | Символы |
| Разрешенный | T |
| Запрещенный | F |
| Игнорируемый | I |
| Строковый литерал | Q |
| Символьный литерал | QQ |
| Сепаратор | S |
| Перевод строки | N |
| Пробел, табуляция | P |
| Комментарии | COM |

## **3.3 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.

## **3.4 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 3.2.

Таблица 3.2 Соответствие токенов и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| 1 | 2 | 3 |
| integer, string,symbol | t | Названия типов данных языка. |
| Идентификатор | i | Длина идентификатора – 8 символов. |
| Литерал | l | Литерал любого доступного типа. |
| function | f | Объявление функции. |
| Функции стандартной библиотеки | p | Ключевое слово для стандартных функций. |
| return | e | Выход из функции/процедуры. |
| main | m | Главная функция. |
| type | n | Объявление переменной. |
| write | o | Вывод данных. |
| writeline | ^ | Вывод данных c переводом строки. |
| is | ? | Указывает начало условного оператора. |
| while | c | Указывает начало цикла. |
| true | r | Истинная ветвь условного оператора. |
| false | w | Ложная ветвь условного оператора. |
| do | d | Указывает на начало тела цикла. |
| ; | ; | Разделение выражений. |
| , | , | Разделение параметров функций. |
| [ | [ | Начало блока цикла/условного выражения. |
| ] | ] | Конец блока цикла/условного выражения. |
| { | { | Начало тела функции. |
| } | } | Конец тела функции |
| ( | ( | Передача параметров в функцию, приоритет операций. |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций. |
| = | = | Знак присваивания. |

Продолжение таблицы 3.2

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| +  -  \*  /  % | +  -  \*  /  % | 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. Структура конечного автомата и пример графа перехода конечного автомата изображены на рисунках 3.3 и 3.4 соответственно.

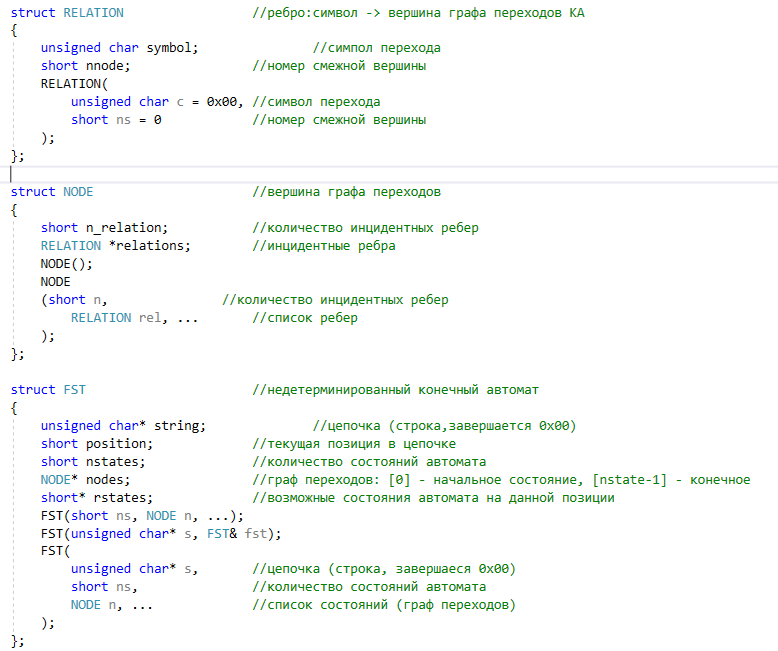


Рис. 3.3 – структура конечного автомата

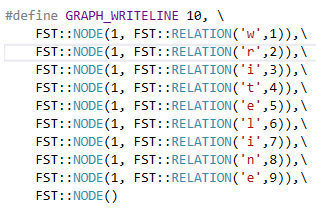


Рис. 3.4 – пример реализации графа конечного автомата для токена writeline

## **3.5 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 лексем представлен на рисунке 3.3. Код C++ со структурой таблицы идентификаторов представлен на рисунке 3.4.

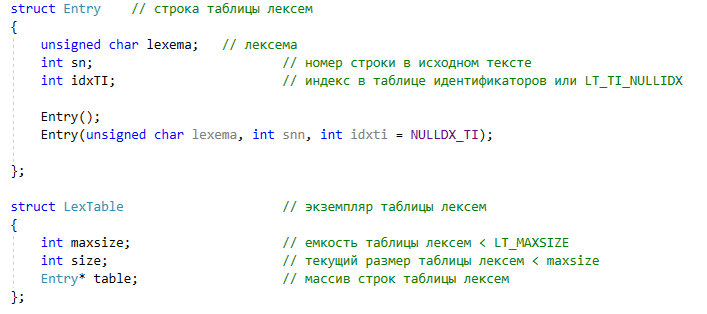


Рис. 3.3 – структура таблицы лексем

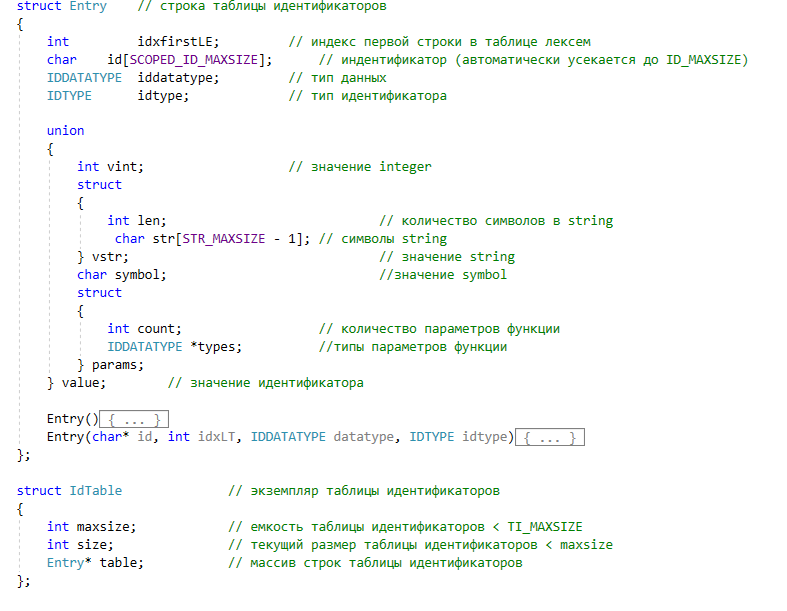


Рис. 3.4 – структура таблицы идентификаторов

## **3.6 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 на рисунке 3.5.



Рис. 3.5 - сообщения лексического анализатора

## **3.7 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.

## **3.8 Параметры лексического анализатора**

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся как в файл журнала, так и в командную строку.

## **3.9 Алгоритм лексического анализа**

* проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;
* для выделенной части входного потока выполняется функция распознавания лексемы;
* при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
* формирует протокол работы;
* при неуспешном распознавании выдается сообщение об ошибке.
* Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «string» представлен на рисунке 3.2, где S0 – начальное, а S6 – конечное состояние автомата.

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «string» представлен на рисунке 3.2, где S0 – начальное, а S6 – конечное состояние автомата.

s t r i n g

Рис. 3.2 – пример графа переходов для цепочки string

## **3.10 Контрольный пример**

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

# **4. Разработка синтаксического анализатора**

## **4.1 Структура синтаксического анализатора**

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

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.



Рис. 4.1 – структура синтаксического анализатора.

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

В синтаксическом анализаторе транслятора языка NVA-2020 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 правил переходов и описание нетерминальных символов представлена в приложении Е.

## **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 |

## **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 NVA-2020. Правила языка NVA-2020 в приложении В.

## **4.5 Описание алгоритма синтаксического разбора**

Принцип работы автомата следующий:

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

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

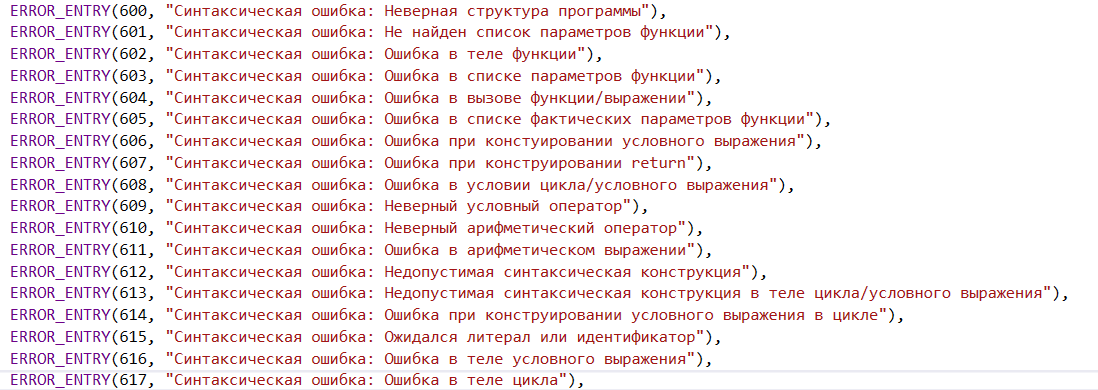
 Перечень сообщений синтаксического анализатора представлен на рисунке 4.3.

Рис. 4.3 – сообщения синтаксического анализатора

## **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 синтаксического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

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

## **4.9. Контрольный пример**

Результаты работы синтаксического 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 выполняет проверку на основные правила языка (семантики языка), которые описаны в разделе 1.16.

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

Сообщения, формируемые семантическим анализатором, представлены на рисунке 5.2.

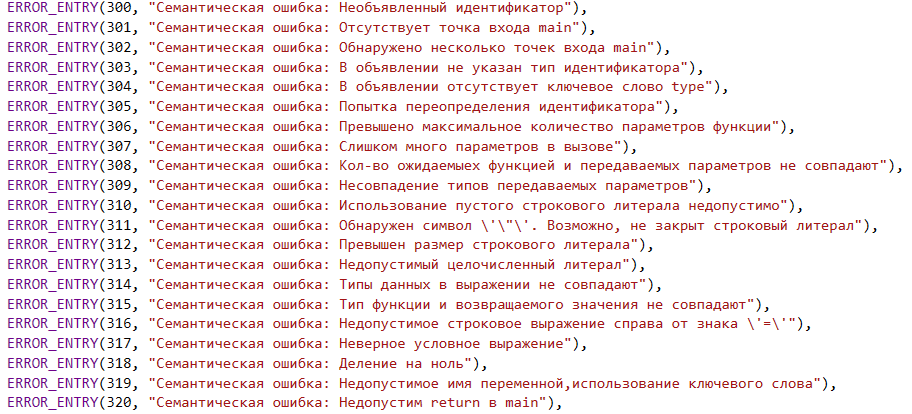


Рис. 5.2 – перечень сообщений семантического анализатора

## **5.4 Принцип обработки ошибок**

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Анализ останавливается после того, как будут найдены все ошибки.

## **5.5 Контрольный пример**

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

Таблица 5.1. Примеры диагностики ошибок

|  |  |
| --- | --- |
| Исходный код | Текст сообщения |
| main{  integer x = 9;  write x;  } | Ошибка N304: Семантическая ошибка: В объявлении отсутствует ключевое слово type Строка: 2 |
| main{  type integer x = 9;  type integer y = x;  } | Ошибка N314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 3 |
| main{  type integer x = 9;  }  main{  type string y = "qwerty";  } | Ошибка N302: Семантическая ошибка: Обнаружено несколько точек входа main |

# **6 Вычисление выражений**

## **6.1 Выражения, допускаемые языком**

В языке NVA-2020 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 |
| + | 1 |
| - | 1 |

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

Все выражения языка NVA-2020 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 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]\* |

# **7. Генерация кода**

## **7.1 Структура генератора кода**

В языке NVA-2020 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода NVA-2020 представлена на рисунке 7.1.



Рис. 7.1 – структура генератора кода

## **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 NVA-2020 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка NVA-2020 и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке NVA-2020 | Тип идентификатора на языке ассемблера | Пояснение |
| integer | sdword | Хранит целочисленный тип данных. |
| string, symbol | dword | Хранит указатель на начало строки. Строка должна завешаться нулевым символом. |

## **7.3 Статическая библиотека**

В языке NVA-2020 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 |
| void outstr(char\* ptr) | Вывод на консоль строки ptr |
| void outnum(int value) | Вывод на консоль целочисленной переменной value |
| void outstrline(char\* ptr) | Вывод на консоль строки ptr и перевод на новую строку |
| void outnumline(int value) | Вывод на консоль целочисленной переменной value и перевод на новую строку |
| int lenght(char\* str) | Вычисление длины строки |
| void system\_pause() | Ожидание нажатия клавиши пользователем |
| random(int a) | Генерация случайного числа в диапазоне от -a до a |
| power(int a, int b) | Возведение числа a в степень b |

## **7.4 Особенности алгоритма генерации кода**

В языке NVA-2020 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2

Рис. 7.2 – структура генератора кода

## **7.5 Входные параметры генератора кода**

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке NVA-2020. 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.

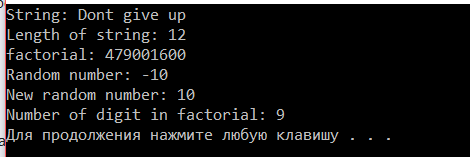


Рисунок 7.2 – Результат работы программы на языке NVA-2020

# **8. Тестирование транслятора**

## **8.1 Общие положения**

При возникновении ошибки на каком-либо этапе трансляции, она обрабатывается в главном файле программы: ошибка выводится на консоль и записывается в файл логирования.

## **8.2 Результаты тестирования**

Результаты тестирования приведены в приложении Ж.

# **Заключение**

В ходе выполнения курсовой работы был разработан транслятор для языка программирования NVA-2020. Таким образом, были выполнены основные задачи данной курсовой работы:

* Сформулирована спецификация языка NVA-2020;
* Разработан лексический анализатор;
* Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика;
* Разработан синтаксический анализатор;
* Разработан семантический анализатор;
* Разработан транслятор кода на язык ассемблера;
* Проведено тестирование транслятора.

Окончательная версия языка NVA-2020 включает:

* 3 типа данных;
* Поддержка операторов вывода;
* Возможность вызова функций стандартной библиотеки;
* Наличие 5 арифметических операторов для вычисления выражений;
* Поддержка функций, процедур, операторов цикла и условия;
* 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. Ахо, А. Теория синтаксического анализа, перевода и компиляции /А. Ахо, Дж. Ульман. – Москва : Мир, 1998. – Т. 2 : Компиляция. - 487 с.

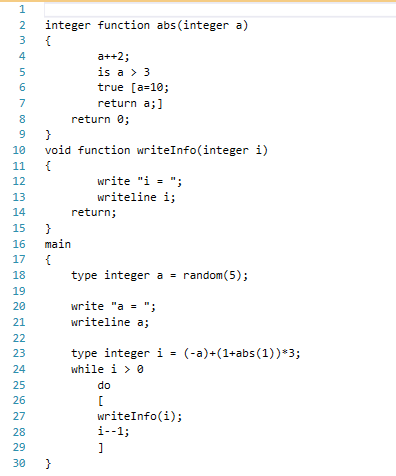
3. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.

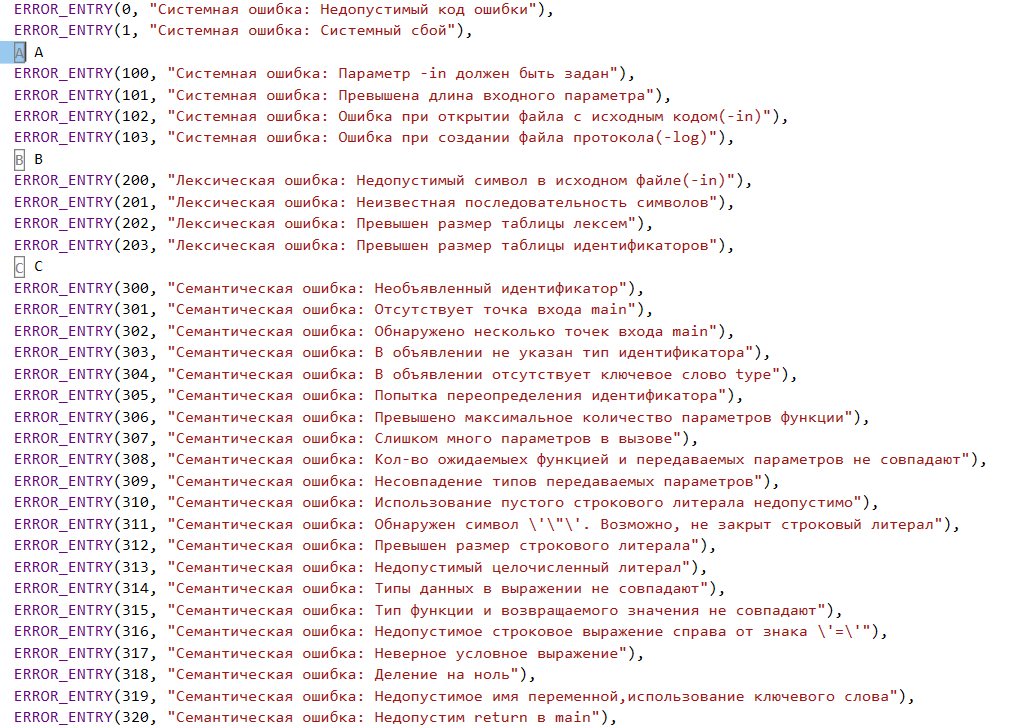
4. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c.

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

# **Приложение А**

Исходный код программы на языке NVA-2020

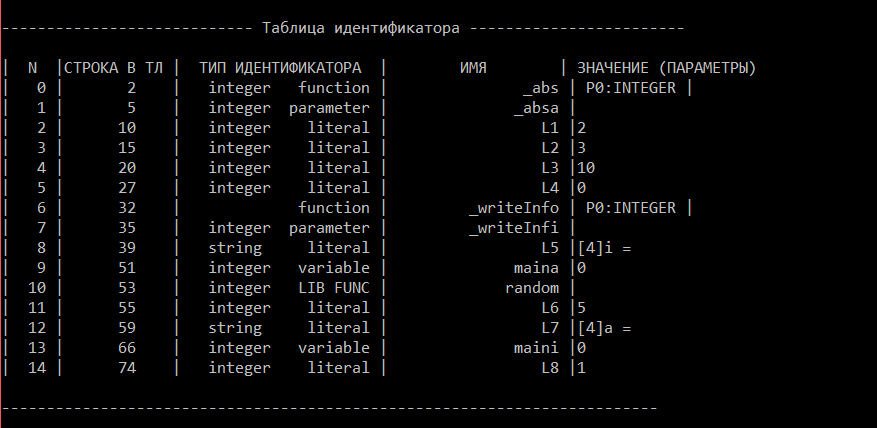


Таблица ошибок языка NVA-2020

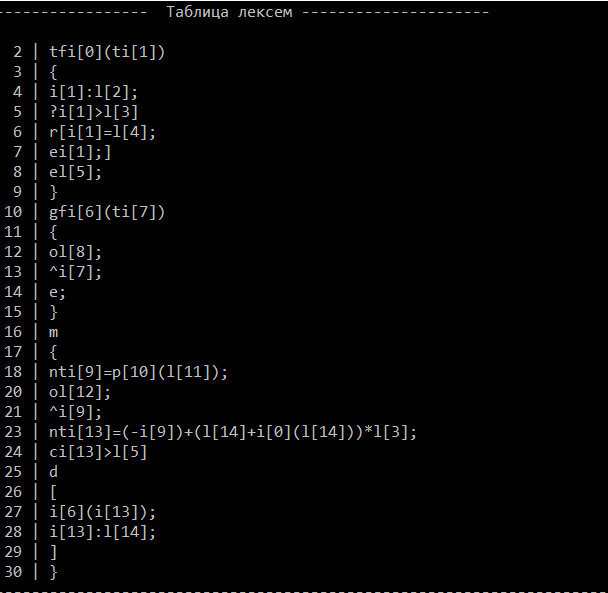
# 

# **Приложение Б**

Таблица идентификаторов контрольного примера

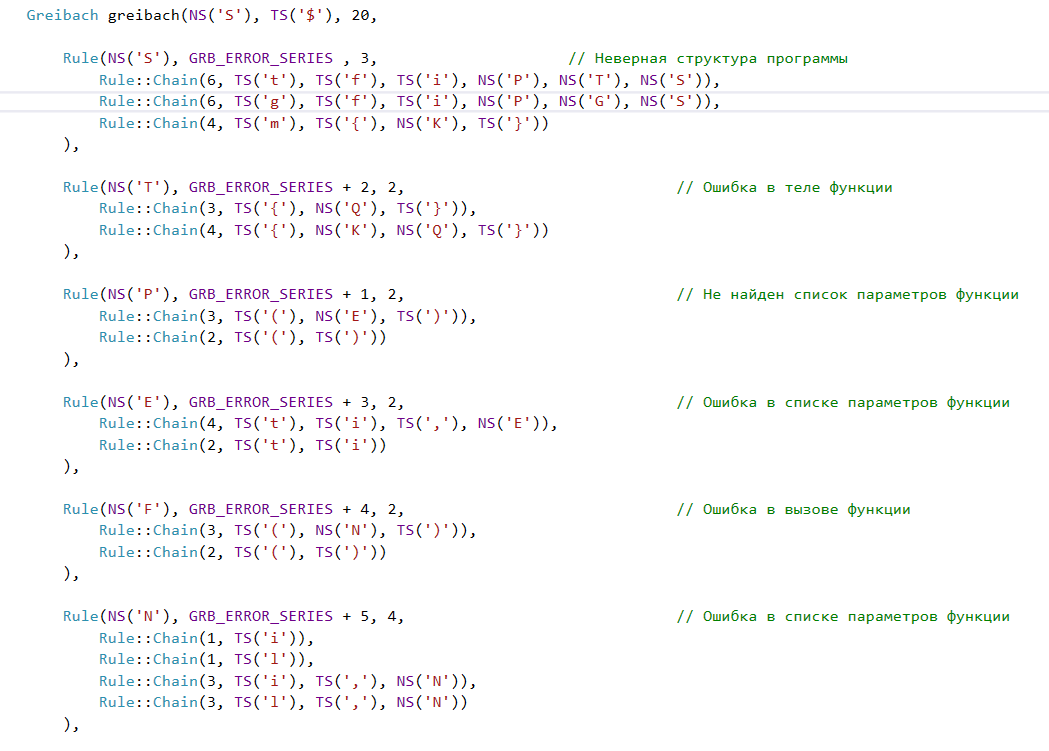


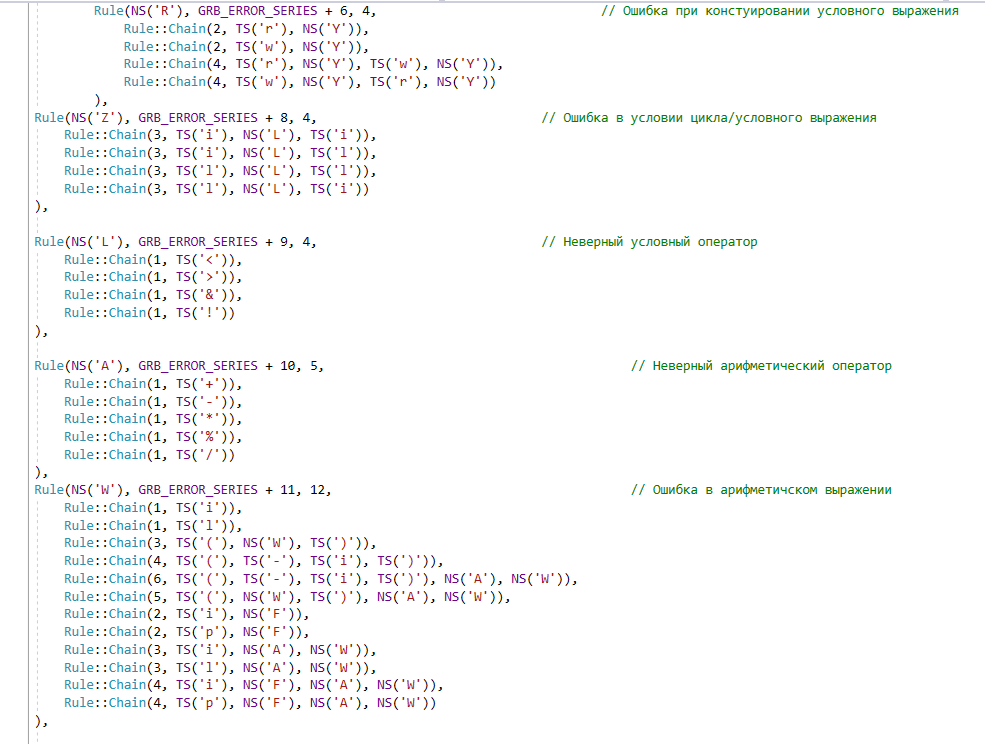
Лексемы соответствующие контрольному примеру

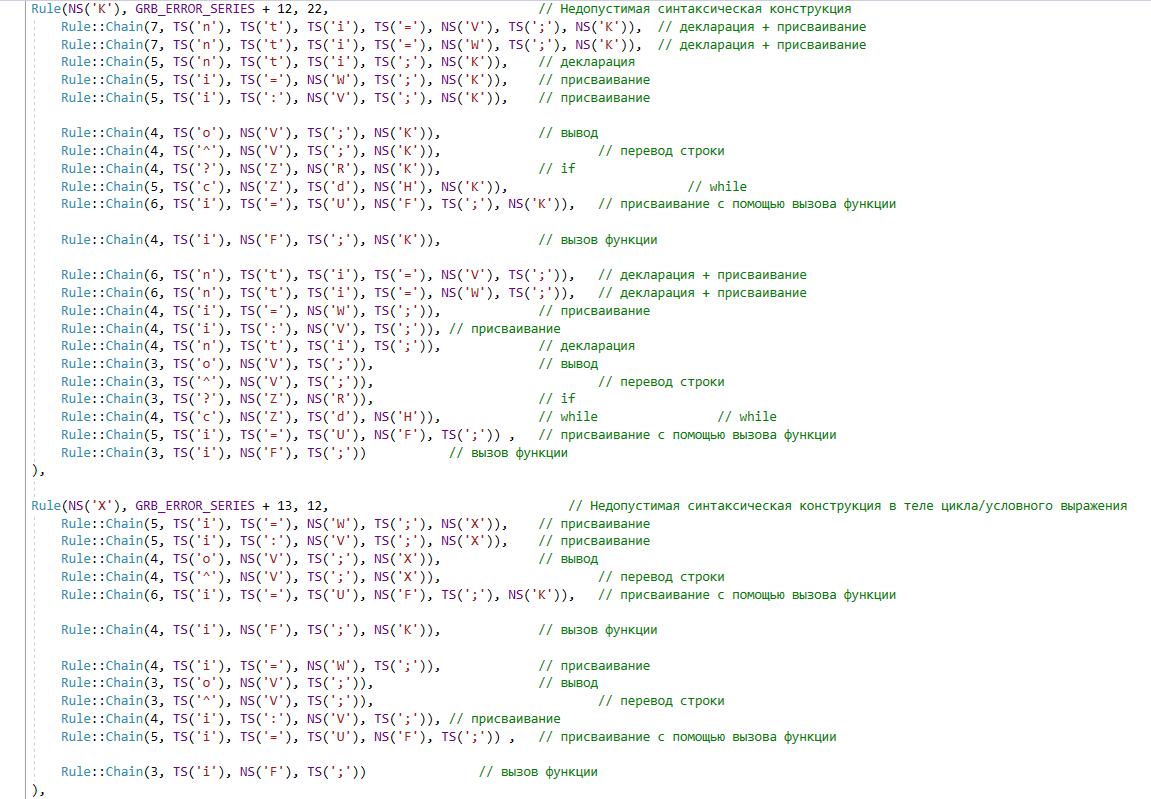


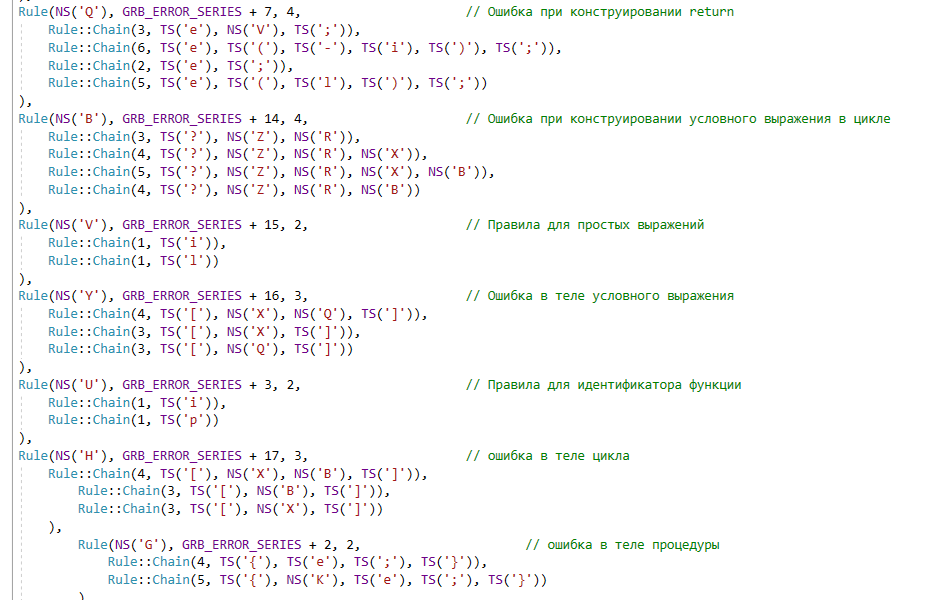
# **Приложение В**

Грамматика языка

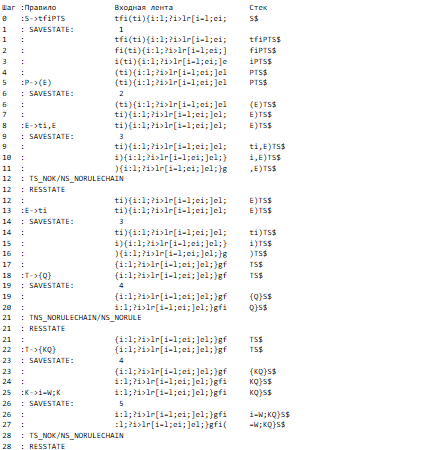


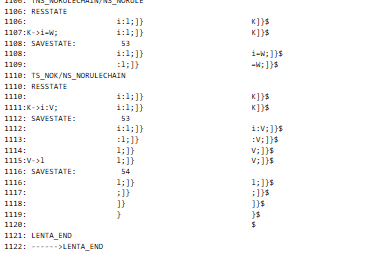




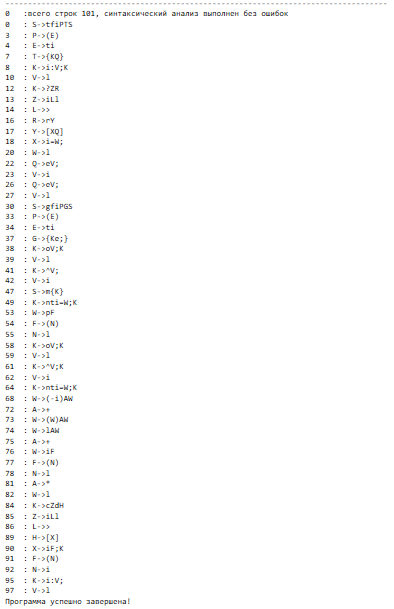


Разбор исходного кода синтаксическим анализатором



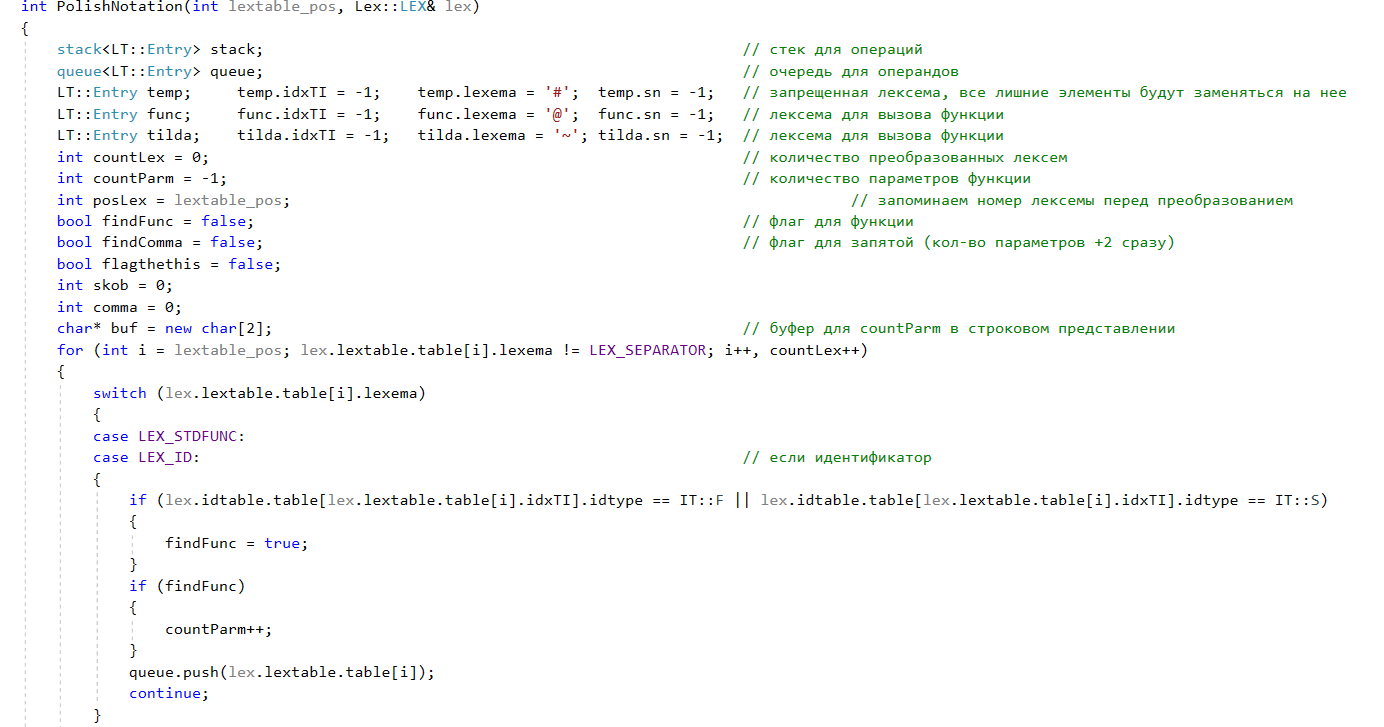


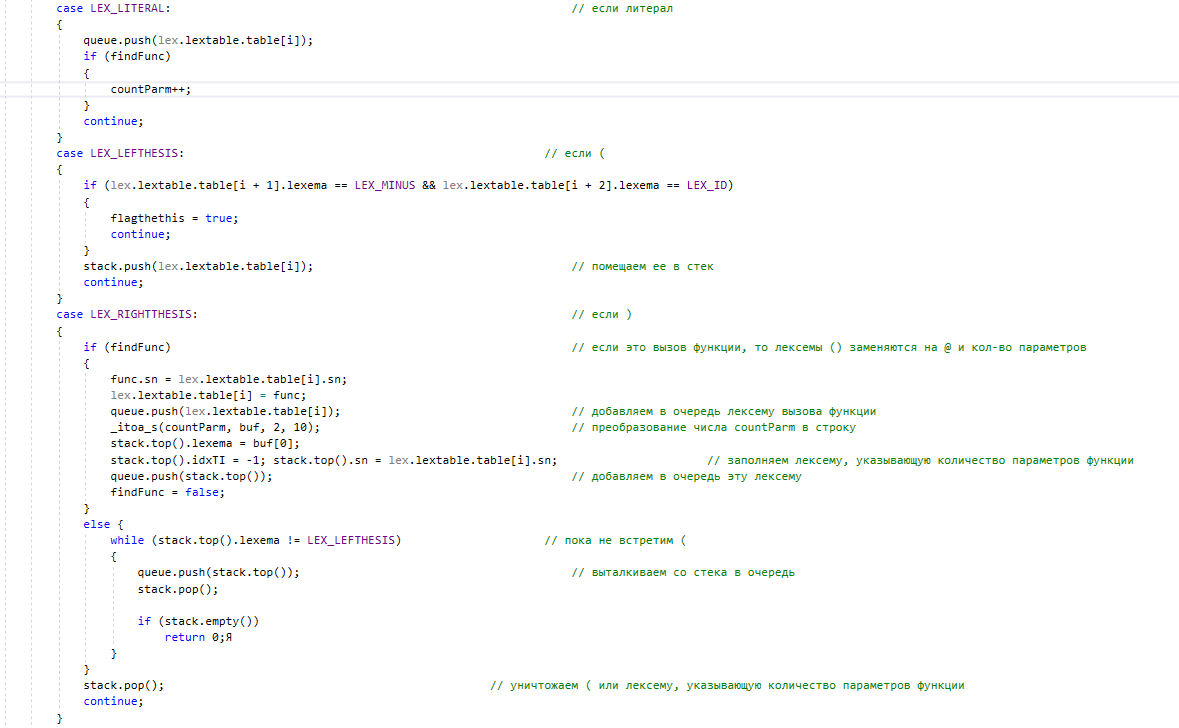
Дерево разбора

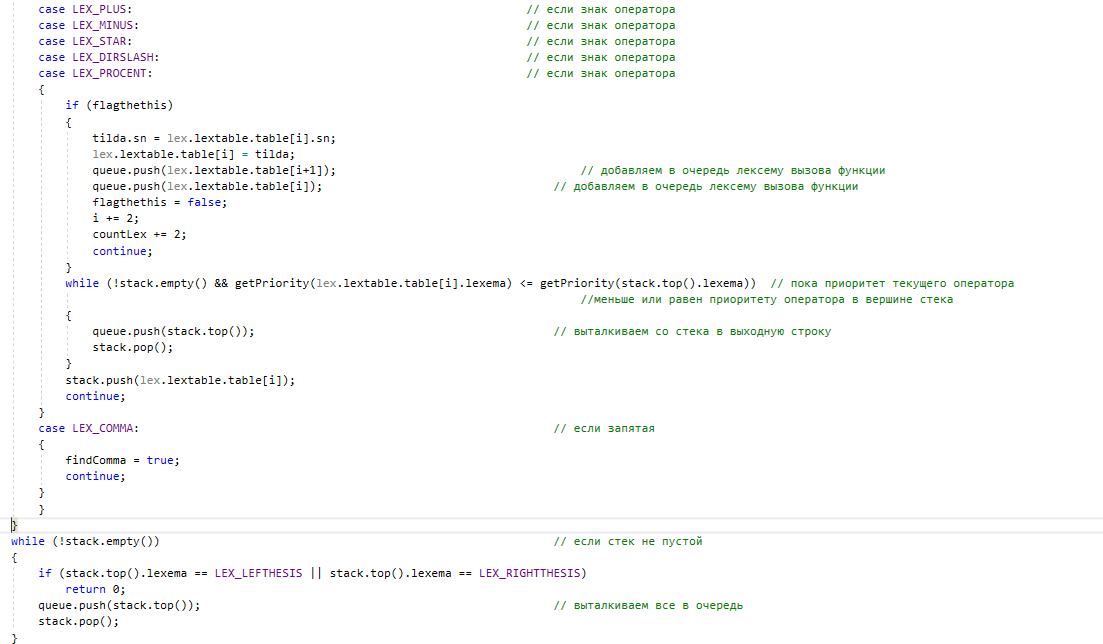


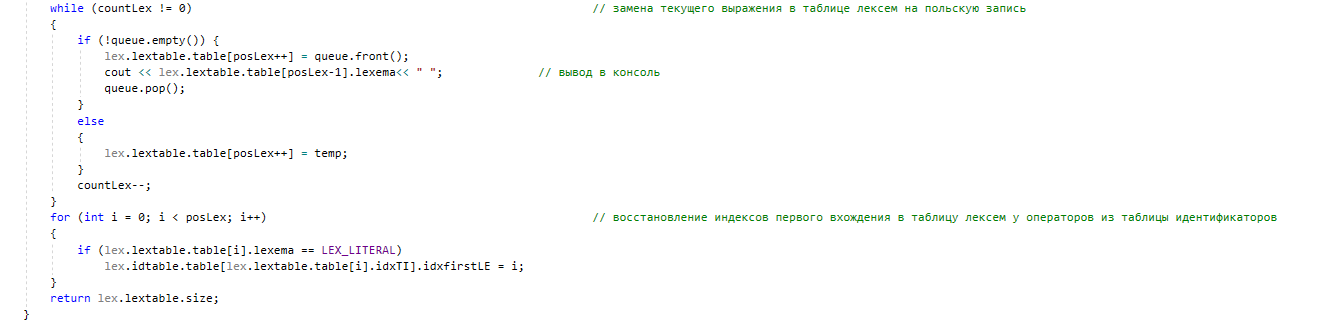
# **Приложение Г**

Программная реализация механизма преобразования в ПОЛИЗ





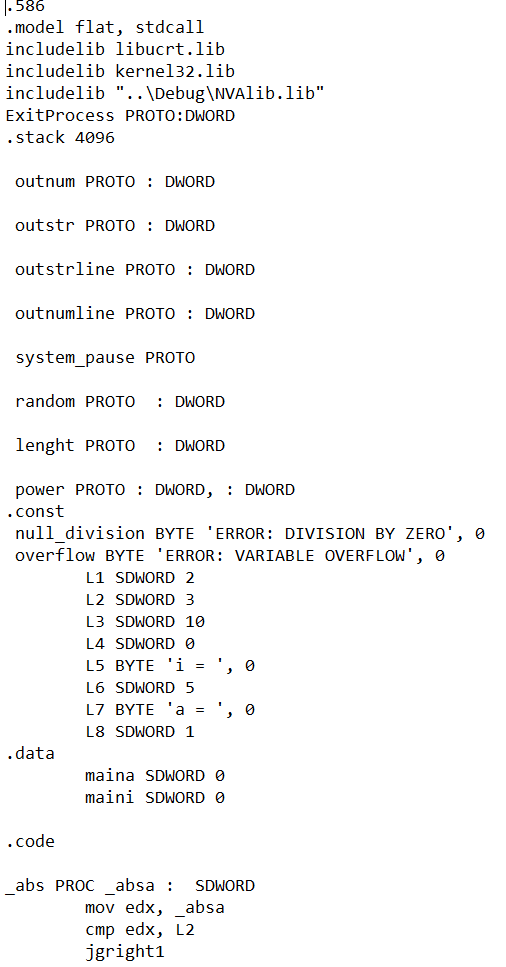


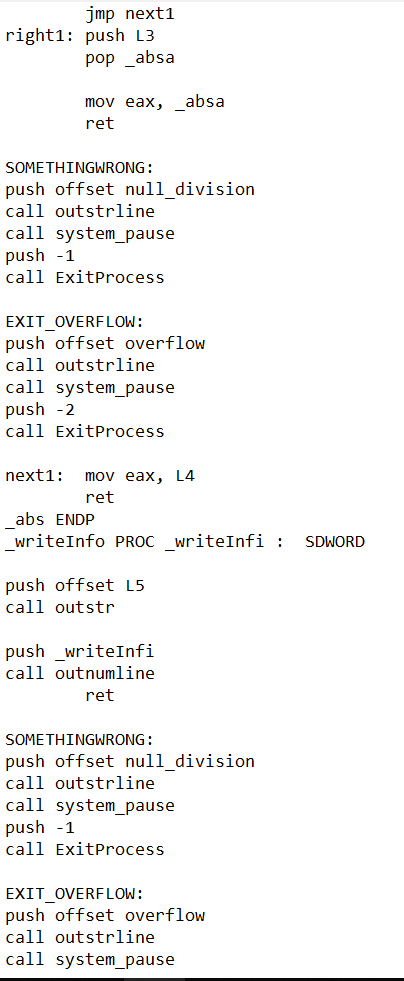


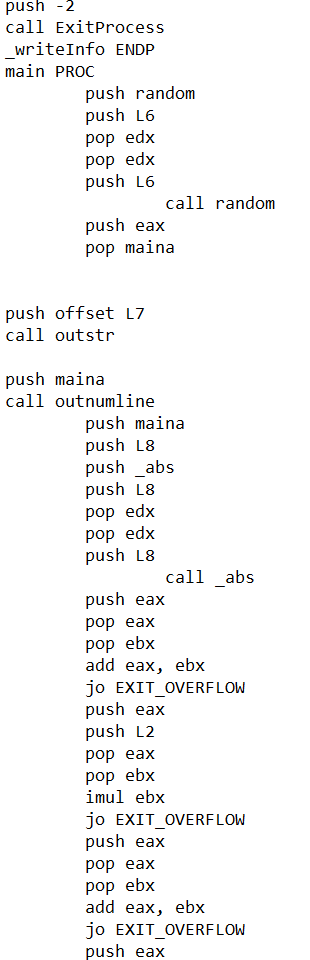
Лексемы соответствующие контрольному примеру после преобразования выражений в ПОЛИЗ

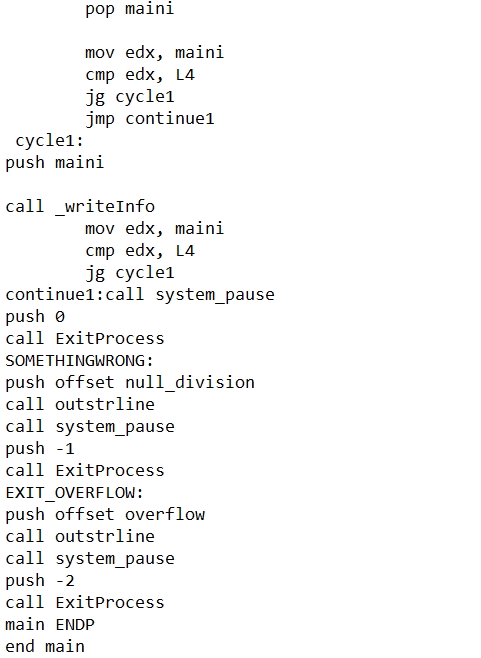
# **Приложение Д**

Результат генерации кода контрольного примера в Ассемблер









## **Приложение Е**

Таблица правил переходов нетерминальных символов

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| S | S->tfiPTS  S->m{K} | Стартовые правила, описывающее общую структуру программы |
| P | P->(E)  P->() | Правила для параметров объявляемых функций |
| T | T->{Q;}  T->[KQ;] | Правила для тела функций |
| Q | Q->eV;  Q->e(l);  Q->e(-i); | Правила для возврата значений |
| E | E->ti,E  E->ti | Правила для списка параметров функции |
| F | F->(N)  F->() | Правила для вывозов функций(в т.ч. и в выражениях) |
| N | N->i  N>l  N->l,N  N->i,N | Правила для параметров вызываемых функций |
| R | R->rY  R->wY  R->rYwY  R->wYrY | Правила составления условного оператора |
| Z | Z->iLi  Z->iLl  Z->lli | Правила для условия цикла/условного оператора |
| L | L-><  L->>  L->&  L->! | Правила для логических операторов |
| A | A->+  A->-  A->\*  A->/  A->% | Правила для арифметических операторов |
| V | V->l  V->i | Правила для простых выражений |

|  |  |  |
| --- | --- | --- |
| Y | Y->[XQ]  Y->[X]  Y->[Q] | Правила для тела условного выражения |
| W | W->l  W->i  W->(W)  W->(-i)  W->(-i)AW  W->(W)AW  W->iF  W->pF  W->iAW  W->lAW  W->iFAW  W->pFAW | Правила для сложных выражений |
| K | K->nti=V;K  K->nti=WK  K->nti;K  K->i=W;K  K->oV;K  K->^VZ;K  K->?ZdHK  K->cZdHK  K->i=UF;K  K->nti=V;  K->nti=W;  K->i=W;  K->nti;  K->oV;  K->^V;  K->?ZR  K->cZdH  K->i=UF; | Программные конструкции |
| X | X->i=W;X  X->oV;X  X->^V;X  X->i=UF;X  X->i=W;  X->oV;  X->^V;  X->i=UF; | Программные конструкции внутри цикла/условного оператора |
| U | U->i  U->p | Правила для идентификатора функции |

|  |  |  |
| --- | --- | --- |
| B | B->?ZR  B->?ZRX  B->?ZRXB  B->?ZRB | Правила для конструировании условного выражения в цикле |
| H | H->[XB]  H->[B]  H->[X] | Правила для тела цикла |

## **Приложение Ж**

Результаты тестирования транслятора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| мэйн | Ошибка 200: Лексическая ошибка: Недопустимый символ в исходном файле(-in) Строка: 1 |
| main{  type integer i = a;  } | Ошибка 300: Семантическая ошибка: Необъявленный идентификатор Строка: 2 |
| integer function a(){  return 1;  } | Ошибка 301: Семантическая ошибка: Отсутствует точка входа main |
| main{  integer i = 3;  } | Ошибка 304: Семантическая ошибка: В объявлении отсутствует ключевое слово type Строка: 2 |
| main{type integer i;  type integer i;} | Ошибка 305: Семантическая ошибка: Попытка переопределения идентификатора Строка: 3 |
| integer function func(integer a)  {return 1;}  main{  type integer i = func(1,2);} | Ошибка 308: Семантическая ошибка: Кол-во ожидаемых функцией и передаваемых параметров не совпадают Строка: 4 |
| integer function func(integer a)  {return 1;}  main{  type integer i = func("string");  } | Ошибка 309: Семантическая ошибка: Несовпадение типов передаваемых параметров Строка: 4 |
| main{type string i = "sdj;  } | Ошибка 311: Семантическая ошибка: Обнаружен символ '"'. Возможно, не закрыт строковый литерал Строка: 2 |
| main{  type integer i = ~3~+3;} | Ошибка 314: Семантическая ошибка: Типы данных в выражении не совпадают Строка: 2 |
| integer function out(){  return; }  main{  out();} | Ошибка 315: Семантическая ошибка: Тип функции и возвращаемого значения не совпадают Строка: 2 |
| main{  is ~3~ < 4  true[write 3;]  } | Ошибка 317: Семантическая ошибка: Неверное условное выражение Строка: 1 |

|  |  |
| --- | --- |
| main{  type integer s = 1/0;  } | Ошибка 318: Семантическая ошибка: Деление на ноль Строка: 2 |
| main{  is 1 < 2  true[return 1;]  } | Ошибка 320: Семантическая ошибка: Недопустим return в main Строка: 3 |
| main type integer x; } | Ошибка 600: строка 1, Синтаксическая ошибка: Неверная структура программы |
| string function fi({}  main{  } | Ошибка 601: строка 1, Синтаксическая ошибка: Не найден список параметров функции |
| integer function func(){  return 1;  }  main{  type integer s = func(]);  } | Ошибка 605: строка 5, Синтаксическая ошибка: Ошибка в списке фактических параметров функции |
| main{ is  true [write ~a~;]  } | Ошибка 608: строка 2, Синтаксическая ошибка: Ошибка в условии цикла/условного выражения |
| main{  type integer s = 1 ++2;  } | Ошибка 610: строка 2, Синтаксическая ошибка: Неверный арифметический оператор |
| main{  type integer s = 1]2;  } | Ошибка 611: строка 2, Синтаксическая ошибка: Ошибка в арифметическом выражении |
| main{ type integer s  } | Ошибка 612: строка 2, Синтаксическая ошибка: Недопустимая синтаксическая конструкция |
| main{  while 1 < 2  do[return;]} | Ошибка 613: строка 3, Синтаксическая ошибка: Недопустимая синтаксическая конструкция в теле цикла/условного выражения |