МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора SRA-2019»

Выполнил студент Савин Роман Алексеевич

(Ф.И.О.)

Руководитель проекта ст.пр. Наркевич Аделина Сергеевна

(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 ст.пр. Наркевич Аделина Сергеевна

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

Нормоконтролер ст.пр. Наркевич Аделина Сергеевна

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

Курсовой проект защищен с оценкой

Минск 2019

[Глава 1. Спецификация языка программирования 5](#_Toc27412796)

[1.1 Характеристика языка программирования 5](#_Toc27412797)

[1.2 Алфавит языка 5](#_Toc27412798)

[1.3 Применяемые сепараторы 5](#_Toc27412799)

[1.4 Применяемые кодировки 6](#_Toc27412800)

[1.5 Типы данных 6](#_Toc27412801)

[1.6 Преобразование типов данных 6](#_Toc27412802)

[1.7 Идентификаторы 6](#_Toc27412803)

[1.8 Литералы 7](#_Toc27412804)

[1.9 Объявления данных и область видимости 7](#_Toc27412805)

[1.10 Инициализация данных 7](#_Toc27412806)

[1.11 Инструкции языка 8](#_Toc27412807)

[1.12 Операции языка 8](#_Toc27412808)

[1.13 Выражения и их вычисления 9](#_Toc27412809)

[1.14 Программные конструкции языка 9](#_Toc27412810)

[1.15 Область видимости идентификаторов 9](#_Toc27412811)

[1.16 Семантические проверки 9](#_Toc27412812)

[1.17 Распределение оперативной памяти на этапе выполнения 10](#_Toc27412813)

[1.18 Стандартная библиотека и её состав 10](#_Toc27412814)

[1.19 Ввод и вывод данных 11](#_Toc27412815)

[1.20 Точка входа 11](#_Toc27412816)

[1.21 Соглашения о вызовах 11](#_Toc27412817)

[1.22 Объектный код 11](#_Toc27412818)

[1.23 Классификация сообщений транслятора 11](#_Toc27412819)

[1.24 Контрольный пример 11](#_Toc27412820)

[Глава 2. Структура транслятора 12](#_Toc27412821)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 12](#_Toc27412822)

[2.2 Перечень входных параметров транслятора 13](#_Toc27412823)

[2.3 Перечень протоколов, формируемых транслятором и их содержимое 13](#_Toc27412824)

[Глава 3. Разработка лексического анализатора 14](#_Toc27412825)

[3.1 Структура лексического анализатора 14](#_Toc27412826)

[3.2 Контроль входных символов 14](#_Toc27412827)

[3.3 Удаление избыточных символов 15](#_Toc27412828)

[3.4 Перечень ключевых слов, сепараторов, символов операций соответствующих им лексем 15](#_Toc27412829)

[3.6 Принцип обработки ошибок 17](#_Toc27412830)

[3.7 Структура и перечень сообщений лексического анализатора 17](#_Toc27412831)

[Глава 4. Разработка синтаксического анализатора 18](#_Toc27412832)

[4.1 Структура синтаксического анализатора. 18](#_Toc27412833)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 18](#_Toc27412834)

[4.3 Построение конченого магазинного автомата 19](#_Toc27412835)

[4.4 Основные структуры данных 20](#_Toc27412836)

[4.5 Описание алгоритма синтаксического разбора 20](#_Toc27412837)

[4.6 Структура и перечень сообщений синтаксического анализатора 20](#_Toc27412838)

[4.7 Параметры синтаксического анализатора и режимы его работы 21](#_Toc27412839)

[4.8 Принцип обработки ошибок 21](#_Toc27412840)

[4.9 Контрольный пример 21](#_Toc27412841)

[Глава 5. Разработка семантического анализатора 22](#_Toc27412842)

[5.1 Структура семантического анализатора 22](#_Toc27412844)

[5.2 Функции семантического анализатора 22](#_Toc27412845)

[5.3 Структура и перечень сообщений семантического анализатора 22](#_Toc27412846)

[5.4 Принцип обработки ошибок 23](#_Toc27412847)

[5.5 Контрольный пример 23](#_Toc27412848)

[Глава 6. Преобразование выражений 24](#_Toc27412849)

[6.1 Выражения, допускаемые языком 24](#_Toc27412850)

[6.2 Польская запись и принцип ее построения 24](#_Toc27412851)

[6.3 Программная реализация обработки выражений 25](#_Toc27412852)

[6.4 Контрольный пример 25](#_Toc27412853)

[Глава 7. Генерация кода 26](#_Toc27412854)

[7.1 Структура генератора кода 26](#_Toc27412855)

[7.2 Представление типов данных в памяти 26](#_Toc27412856)

[7.3 Статическая библиотека 27](#_Toc27412857)

[7.4 Особенности алгоритма генерации кода 27](#_Toc27412858)

[7.5 Контрольный пример 27](#_Toc27412859)

[Глава 8. Тестирование транслятора 28](#_Toc27412860)

[Заключение 31](#_Toc27412861)

[Приложение А 33](#_Toc27412862)

[Приложение Б 34](#_Toc27412863)

[Приложение Г 36](#_Toc27412864)

[Приложение Д 39](#_Toc27412865)

[Приложение Е 40](#_Toc27412866)

[Литература 44](#_Toc27412867)

**Введение**

Транслятор – это комплекс отдельных программ, позволяющих преобразовывать исходный код на одном языке программирования в исходный код на другом языке программирования.

Классический транслятор состоит из следующих частей:

* лексический анализатор;
* синтаксический анализатор;
* семантический анализатор;
* генератор кода, или интерпретатор.

Все части транслятора, взаимодействуя между собой, обрабатывают входной текст и строят для него эквивалентный текст на понятном компьютеру языке программирования.

# Глава 1. Спецификация языка программирования

* 1. **Характеристика языка программирования**

Язык программирования SRA-2019 является универсальным языком высокого уровня. Он является транслируемым, не объектно-ориентированным. Язык строго типизируемый, что говорит о невозможности преобразования типов, транслируемым языком программирования. В SRA-2019 используется два типа данных: целочисленные (number) и строковые (string) с функцией конкатенации и копирования строк.

* 1. **Алфавит языка**

В основе алфавита SRA-2019 лежит таблица символов ACSII. Недопустимо использование кириллицы, разрешены только латинские символы размером один байт, представленные на рисунке 1.1.

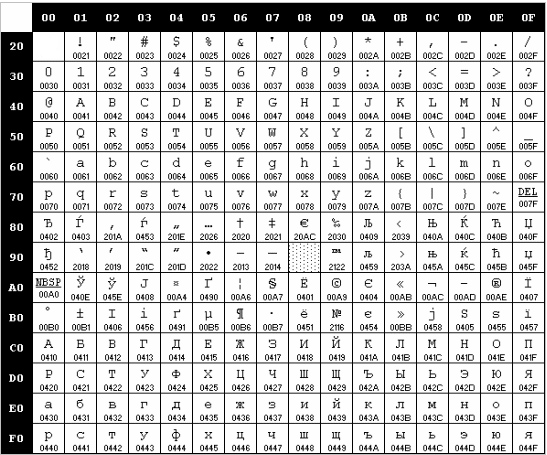


Рисунок 1.1 - Алфавит входных символов

* 1. **Применяемые сепараторы**

Сепараторы необходимы для разделения операций языка. Сепараторы, используемые в языке программирования SRA-2019, приведены в таблице 1.1.

1. Таблица 1.1 - Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение сепаратора |
| ; | Разделитель инструкций |
| { } | Программный блок |
| ( ) | Параметры;  Приоритетность операций |
| ˽ (Пробел) | Служит для их разделения. Допускается везде, кроме идентификаторов и ключевых слов |
| , | Разделитель параметров в функции |

* 1. **Применяемые кодировки**

При трансляции применяется стандартная кодировка ACSII. Описание кодировки представлено в пункте 1.1.

* 1. **Типы данных**

Пользовательские типы данных не поддерживаются. Допускается использование фундаментальных типов данных определенных в таблице 1.2.

Таблица 1.2 - Фундаментальные типы данных

|  |  |
| --- | --- |
| Тип данных | Описание |
| Number | Является целочисленным типом данных. Этот тип данных занимает 4 байта. Предназначен для арифметических операций над числами. Инициализация по умолчанию: 0.  Предусмотрены следующие арифметические операции:  + - бинарная операция суммирования (number + number);  - - бинарная операция вычитания (number - number);  \* - бинарная операция умножения (number \* number);  \_modint – функция вычисления остатка от деления (\_modint(number1, number2));  / - бинарная операция деления (number / number). |
| String | Является строковым типом данных. Предназначен для работы с символами, каждый символ занимает 1 байт. Максимальное количество символов – 255, включая символ окончания строки.  Инициализация по умолчанию: нулевой символ.  Операции над данными строкового типа: возможно присваивание строковому идентификатору значения другого строкового идентификатора, строкового литерала или значения строковой функции, конкатенация строк, копирование строк. |

* 1. **Преобразование типов данных**

Преобразование не поддерживается, все типы данных определены однозначно и не могут быть преобразованы в другие, так как язык SRA-2019 является строго типизируемым.

* 1. **Идентификаторы**

Для именования функций, параметров и переменных используются Идентификаторы. Не предусмотрены зарезервированные идентификаторы. Имя идентификатора составляется по следующим образом:

* состоит из символов латинского алфавита [a..z].
* максимальная длина идентификатора равна 10 и не должна превышать это значение. При превышении максимально значения длина идентификатора усекается до 10.
  1. **Литералы**

Предусмотрены числовые (number) и строковые (string) литералы. Правила записи приведены в таблице 1.3.

Таблица 1.3 - Правила записи литералов

|  |  |
| --- | --- |
| Тип литерала | Описание литерала |
| Числовые | Максимально допустимое значение равно . Минимально допустимое значение равно . Могут состоять только из цифр [0..9] без дробной части. При выходе за пределы допустимости будет выведена соответствующая ошибка. |
| Строковые | Состоит из из символов, заключенных в "…" (двойные кавычки). Максимальное число которых не может превышать . В случае превышения длины литерала работа транслятора прекращается. |

* 1. **Объявления данных и область видимости**

В языке программирования SRA-2019 переменная должны быть объявлена до ее использования. Областью видимости переменной является блок функции, в которой она определена. Вне блока функции определение переменной запрещено. Недопустимо объявление глобальных переменных. Область видимости схожа с областью видимости C++, то есть сверху вниз. Конструкция для объявления переменных:

var˽<типданных>˽<идентификатор>[=<литерал>|<идентификатор>| <выражение>];

* 1. **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 идентификаторы, литералы, выражения и вызовы функций. Предусмотрены значения по умолчанию, если переменные не инициализированы: значение 0 для целочисленного и типа данных и строка длины 0 ("") для строкового типа данных.

* 1. **Инструкции языка**

В языке программирования SRA-2019 применяются инструкции, представ-ленные в таблице 1.4.

Таблица 1.4 - Инструкции языка

|  |  |
| --- | --- |
| Инструкция языка | Синтаксис |
| Главная функция | main  {  <инструкции языка>  } |
| Вызов функций | <идентификатор функции>(<идентификатор / литерал>,…) |
| Возврат из функции | return˽<идентификатор / литерал> |
| Объявление функции | <тип данных>˽function˽<идентификатор>  {  <инструкции языка>  } |
| Объявление переменной | var˽<тип данных>˽<идентификатор> |
| Присваивание | string и number:  <идентификатор> = <литерал>;  <идентификатор> = <выражение>;  <идентификатор 1> = <идентификатор 2>; |
| Вывод данных | print˽<идентификатор / литерал> |

* 1. **Операции языка**

Операция сложения применима к строковым и целочисленным типам данных. При строковых типах данных происходит конкатенация строк. Остальные операции в языке программирования применимы исключительно к целочисленным типам данных.

Наибольшую приоритетность арифметических операций имеют операции сложения и деления, а сложение и вычитание меньшую. При одинаковом приоритете первой выполнится операция расположенная левее. Изменить приоритетность можно с помощью круглых скобок.

Операции в языке программирования SRA-2019 применимые к целочисленным и строковым типам данных приведены в таблице 1.5.

Таблица 1.5 - Операции языка

|  |  |
| --- | --- |
| Тип | Операторы |
| Арифметические | () – приоритетность операций  + ̶ сложение  -  ̶ вычитание  \* ̶ умножение  / ̶ деление |
| Строковые | + – конкатенация |

* 1. **Выражения и их вычисления**

Предусмотрены следующие правила составления выражений:

* + выражения читаются слева направо и записываются в одну строку;
  + реализация выражений происходит с помощью обратной польской записи;
  + для изменения приоритета операция используются круглые скобки.
  1. **Программные конструкции языка**

Программные конструкции представлены в таблице 1.7.

Таблица 1.7 - Программные конструкции языка

|  |  |
| --- | --- |
| Конструкция | Представление в языке |
| Главная функция | main  {  <инструкции языка>  } |
| Функция | <тип данных>˽function˽<идентификатор>(<тип данных>˽<идентификатор>, …)  {  <инструкции языка>  } |
| Точка входа | main |

* 1. **Область видимости идентификаторов**

Все идентификаторы обязаны быть объявленными внутри функции. Вне функции объявление идентификаторов недопустимы. Глобальных переменных нет, только локальные. Параметры видны только внутри функции, в которой объявлены.

* 1. **Семантические проверки**

Перечень семантических проверок, предусмотренных языком, приведен в таблице 1.8.

Таблица 1.8 - Перечень семантических проверок

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Наличие функции main, как точки входа в программу |
| 2 | Наличие только одной точки входа |
| 3 | Переопределение идентификаторов |
| 4 | Использование идентификаторов без их объявления |
| 5 | Проверка на количество параметров объявляемых функций и ограничения применяемые к ним |
| 6 | Правильность передаваемых в функцию параметров: количество, типы |
| 7 | Правильность строковых выражений |
| 8 | Превышение размера строковых и числовых литералов |

* 1. **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.

* 1. **Стандартная библиотека и её состав**

В языке SRA-2019 присутствует стандартная библиотека, которая автоматически подключается при трансляции исходного кода в язык ассемблера. У каждого типа данных есть свои функции реализующие различные команды. Функции стандартной библиотеки вызываются с наличием '\_' перед названием самой функции. Например: \_modint(a,b). Содержимое библиотеки и описание функций представлено в таблице 1.9.

Таблица 1.9 - Стандартная библиотека языка SRA-2019

|  |  |
| --- | --- |
| Функция | Описание |
| number function modint (number a, number b); | Целочисленная функция. Вычисляет и возвращает остаток от деления a на b. |
| number function strcpy(string a, string b) | Строковая функция. Копирует содержимое строки b в строку a |
| string function strcat(string a, string b); | Строковая функция. Возвращает результат конкатенации строк a, b, записанный в строку a. |

* 1. **Ввод и вывод данных**

Ввод данных не предусмотрен. Вывод данных осуществляется с помощью ключевого слова print. В качестве аргумента принимаются числовые и строковые идентификаторы, так же выражения:

print <идентификатор>;

print <выражение>;

* 1. **Точка входа**

В языке SRA-2019 точкой входа является ключевое слово “main”. Точка входа не может отсутствовать, также не может быть переопределена. В программе может быть только одна точка входа.

* 1. **Соглашения о вызовах**

При генерации кода используется соглашение \_cdecl, в котором все параметры передаются в стек справа налево. Освобождением памяти занимается вызываемая подпрограмма, которая очищает стек.

* 1. **Объектный код**

Исходный код языка транслируется в язык ассемблера.

* 1. **Классификация сообщений транслятора**

Транслятор генерирует сообщения о ошибках пользователю. В соответствии с префиксами будут различаться сообщения, представленные в таблице 1.10.

Таблица 1.10 - Классификация сообщений транслятора

|  |  |
| --- | --- |
| Префикс ошибки | Описание ошибки |
| LEX:### | ### - код ошибки. Сообщение, генерируемое на этапе лексического анализа. |
| SYN:### | ### - код ошибки. Сообщение, генерируемое на этапе синтаксического анализа. |
| SEM:### | ### - код ошибки. Сообщение, генерируемое на этапе семантического анализа. |
| [SYSTEM]### | ### - код ошибки. Сообщение, генерируемое при критической ошибке. |

* 1. **Контрольный пример**

Контрольный пример, написанный на языке SRA-2019, представлен в приложении А.

**Глава 2. Структура транслятора**

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

Исходный код, написанный на языке программирования SRA-2019, является для транслятора входными данными.

Как выходные данные используется объектный код и протоколы работы транслятора, описанные в пункте 2.3.

Компоненты транслятора приведены на рисунке 2.1.



Рис 2.1 - Структура транслятора SRA-2019

Первоначально на вход лексического анализатора передается исходный код. Анализатором проверяется исходный текст на недопустимые символы, выделяет литералы, идентификаторы и ключевые слова, а также формирует таблицы лексем и идентификаторов.

Далее наступает черед синтаксического анализатора, к нему на вход поступа-ет таблица лексем, полученная на этапе лексического анализа. Если программа по-строена синтаксически правильно, то осуществляется переход к этапу трансляции стоящему далее, при ином раскладе работа транслятора останавливается.

Наборы функций, проверяющие правила на разных этапах работы транслято-ра представлены в семантическом анализаторе. Продолжение или остановка работы транслятора всецело зависит от критичности возникающих ошибок.

Генерация кода реализуется посредством чистой интерпретации, без создания промежуточного представления кода. В финале происходит генерация кода, во время исполнения которого формируется объектный код.

**2.2 Перечень входных параметров транслятора**

В таблице 2.1 представлены входные параметры, которые могут использоваться для представления работы транслятора.

Таблица 2.1 - Входные параметры транслятора

|  |  |  |
| --- | --- | --- |
| Параметр | Назначение | Тип |
| -in: | Указывает на файл с исходным кодом. Исходный код содержится в файле с расширением \*.txt | Обязательный |
| -out: | Указывает имя протокола. Если не указан явно, то имя протокола не формируется. | Не обязательный |
| -log: | Указывает имя протокола. Если не указан явно, то имя протокола формируется конкатенацией имени файла исходного кода и постфикса «.log» | Не обязательный |

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

По итогам своей работы транслятор формирует протокол, согласно заданным входным параметрам. -log: <путь к файлу> - в этом файле находятся информация о входных параметрах, количестве символов исходного кода, таблицы лексем, идентификаторов, работы синтаксического анализатора, дерево разбора.

# Глава 3. Разработка лексического анализатора

**3.1 Структура лексического анализатора**

Структура лексического анализатора представлена на рисунке 3.1.



Рис 3.1 - Структура лексического анализатора

Исходный код на языке SRA-2019 является входными данными;

Таблицы лексем и идентификаторов являются выходными данными;

**3.2 Контроль входных символов**

Таблица допустимости представлена на рисунке 3.2.

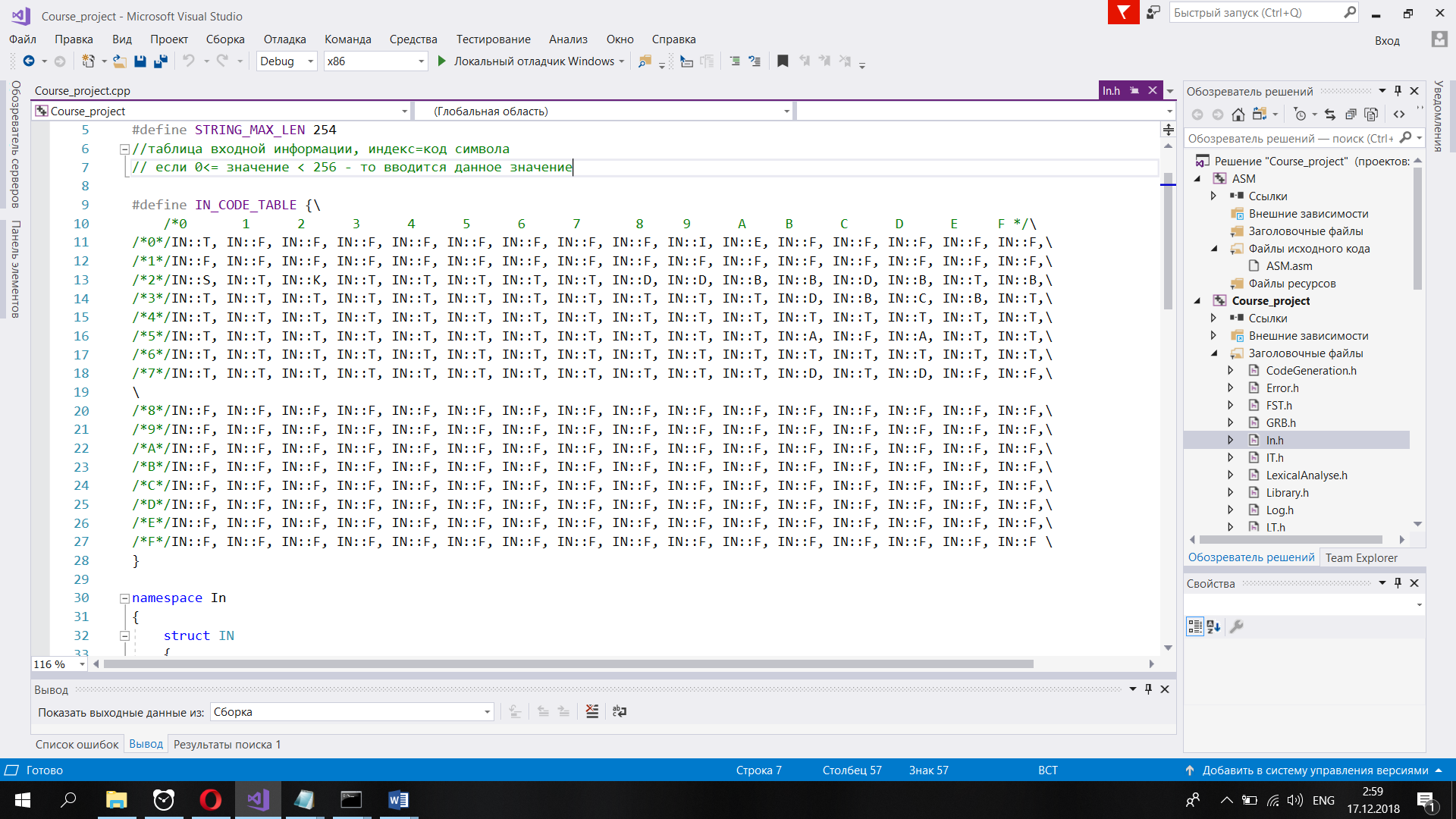


Рисунок 3.2 - Таблица допустимости входных символов

Таблица допустимости была сформирована на основе кодировки windows-1251. Таблица необходима для проверки входных символов на допустимость. Символы могут быть разрешенными, запрещенными, игнорируемыми и др.

Символы представлены в шестнадцатеричной системе счисления. В таблице записаны различные числовые значения соответствующие символам в данной таблице:

«T» - разрешенные алфавитом символы;

«F» - запрещенные алфавитом символы;

«I» - символы, которые игнорируются;

«E» - символ окончания строки;

«D» - символы, являющиеся сепараторами;

«S» - символ пробела;

«K» - символы кавычек;

«C» - символ знака равно;

«B» - символы бинарных операторов;

**3.3 Удаление избыточных символов**

Избыточными символами представлены пробелы, символы табуляции, символы перехода на новую строку.

Алгоритм удаления избыточных символов:

1. Просматриваем текущий символ
   1. Если «K», символы кавычек, то записываем слово, пока не встретим закрывающую кавычку и заносим это слово в массив из токенов.
   2. Если «T», разрешенные символы, то записываем символ в результирующее слово.

1.2.1) Просматриваем следующий символ

1.2.1.1) Если «D»,«B»,«C»,«S»,«E», то заносим слово в массив токенов.

1.3) Если «B», «C», «D», то заносим слова в массив токенов.

1.4) Если «E», то инкрементируем переменную, то есть увеличиваем на единицу, отвечающую за подсчет строк.

1.5) Если «\0», нулевой символ, то переход к пункту 3.

2) Перемещаем указатель на байт вправо и переходим к пункту 1.

3) Окончание алгоритма.

## 3.4 Перечень ключевых слов, сепараторов, символов операций соответствующих им лексем

Перечень ключевых слов, сепараторов, символов операций соответствующих им лексем представлен в таблице 3.1.

Таблица 3.1 - Перечень ключевых слов

|  |  |
| --- | --- |
| Цепочка | Лексема |
| number | n |
| string | s |
| function | f |
| print | p |
| return | r |
| main | m |
| var | d |
| + | + |
| - | - |
| \* | \* |
| / | / |
| = | = |
| ( | ( |
| ) | ) |
| { | { |
| } | } |
| , | , |
| ; | ; |
| идентификатор | i |
| числовой литерал | z |
| строковый литерал | l |

**3.5 Основные структуры данных**

В приложении Б представлены основные структуры данных на этапе синтаксического анализа.

**3.6 Принцип обработки ошибок**

В случае обнаружения критической ошибки, которая не позволяет работать анализаторам или генератору правильно функционировать, транслятор прекращает свою работу и в log-файл записывается ошибка. А при некритической ошибке происходит запись этой ошибки в log-файл без прекращения работы транслятора. Подсчет количества ошибок в обоих случаях не ведется.

## 3.7 Структура и перечень сообщений лексического анализатора

Префикс сообщений “LEX:”. Перечень сообщений, генерируемых на этапе лексического анализа, представлен в таблице 3.2.

Таблица 3.2 - Сообщения лексического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 114 | Превышение длины строкового литерала |
| 120 | Превышение максимальной длины идентификатора. Литерал будет усечен до 10 |
| 121 | Имя функции было продублировано |
| 122 | Дублирование имени идентификатора или отсутствие области видимости |
| 123 | Неизвестная лексема |
| 124 | Неизвестный идентификатор |

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

Текст кода на языке SRA-2019 подается на вход. Параметры не определяют режим работы лексического анализатора.

**3.9 Контрольный пример**

На вход лексического анализатора подается программа на языке SRA-2019, описанная в пункте 1.25. Результат работы лексического анализатора представлен в приложении В.

# Глава 4. Разработка синтаксического анализатора

## 4.1 Структура синтаксического анализатора.

Структура синтаксического анализатора представлена на рисунке 4.1.



Рисунок 4.1 - Структура синтаксического анализатора

Таблицы лексем и идентификаторов являются входными данными.

Дерево разбора является выходными данными.

## 4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

Грамматика, описывающая язык SRA-2019 представлена в таблице 4.1.

Таблица 4.1 - грамматика языка SRA-2019

|  |  |
| --- | --- |
| Нетерминалы | Описание |
|  | Правила, описывающие общую структуру программы |
|  | Порождает правила, описывающие инструкции языка |
|  | Порождает правила, описывающие выражения |
|  | Порождает правила, описывающие арифметические действия |
|  | Порождает правила, описывающие формальные параметры функции |
|  | Порождает правила, описывающие фактические параметры функции |

## 4.3 Построение конченого магазинного автомата

Принцип действия конечного магазинного автомата представлен на рисунке 4.2.



Рис. 4.2 - МП-автомат

Формальное описание МП-автомата:



 - множество состояний;

 - алфавит входных символов;

 - специальный алфавит магазинных символов;

-функция переходов автомата , где  - множество подмножеств ;

 - начальное состояние автомата;

- начальное состояние магазина (маркер дна);

- множество конечных состояний.

Конфигурация (текущее состояние автомата) описывается тройкой , где - текущее состояние автомата, - остаток цепочки,  - цепочка-содержимое магазина.

Начальное состояние , - начальное состояние автомата,  - входная цепочка, - маркер дна магазина.

Цепочка  является допустимой (распознается) автоматом , если  и .

Работа автомата 

1. состояние автомата 
2. читает символ  находящийся под головкой (сдвигает ленту);
3. не читает ничего (читает , не сдвигает ленту);
4. из  определяет новое состояние , если  или .
5. читает верхний (в стеке) символ  и записывает цепочку  т.к. , при этом, если , то верхний символ магазина просто удаляется.

работа автомата заканчивается 

## 4.4 Основные структуры данных

В приложении Г представлены основные структуры данных и правила пере-хода, используемые на этапе синтаксического анализа.

## 4.5 Описание алгоритма синтаксического разбора

Входные символы и лексемы в форме Грейбах находятся в ленте на входе конечного автомата.

1) Если лента не пустая, переходим далее следующему пункту, иначе переходим к пункту 5.

2) Если на верхушке магазина нетерминальный символ.

2.1) Если есть такое правило, то переходим к следующему пункту.

2.1.1) Если цепочка есть, возвращаем NS\_OK. Переходим к пункту 4.

2.1.2) Иначе восстанавливаем состояние. Переходим к пункту 4.

2.2) Иначе возвращаем ошибку. Переход к пункту 4.

3) Если на верхушке терминал и он совпадает с символом на ленте, то удаляем его из стека и продвигаем ленту. Переход к пункту 4.

4) Повторяем шаг, переходим к пункту 1.

5) Конец работы.

## 4.6 Структура и перечень сообщений синтаксического анализатора

Префикс сообщений “SYN::”. Перечень сообщений, генерируемых на этапе синтаксического анализа, представлен в таблице 4.3.

Таблица 4.3 - Перечень сообщений

|  |  |
| --- | --- |
| Код | Сообщение |
| 200 | Неправильная структура программы |
| 201 | Ошибочный оператор |
| 202 | Ошибка в выражении |
| 203 | Ошибка в расстановке знаков |
| 204 | Ошибка в формальных параметрах |
| 205 | Ошибка в параметрах вызываемой функции |

## 4.7 Параметры синтаксического анализатора и режимы его работы

Таблицы идентификаторов и лексем являются входными параметрами для синтаксического анализатора. Эти таблицы мы получаем в ходе лексического анализа. В конце после разбора формируется дерево разбора, которое выводится в протокол работы –log.

## 4.8 Принцип обработки ошибок

При обнаружении ошибки в цепочке какого-либо правила, синтаксический анализатор идет вверх по дереву разбора, пока не найдет верный вариант. Иначе запоминается самая глубокая ошибка, которая выводится в протокол работы.

## 4.9 Контрольный пример

Результатом работы синтаксического анализатора является дерево разбора. Результат работы синтаксического анализатора представлен в приложении Д.

# Глава 5. Разработка семантического анализатора

1. 1. **Структура семантического анализатора**

Семантический анализатор состоит из набора функций для проверки правильности исходной программы. Функции анализатора выполняются на различных этапах работы транслятора. Структура семантического анализатора представлена на рисунке 5.1.

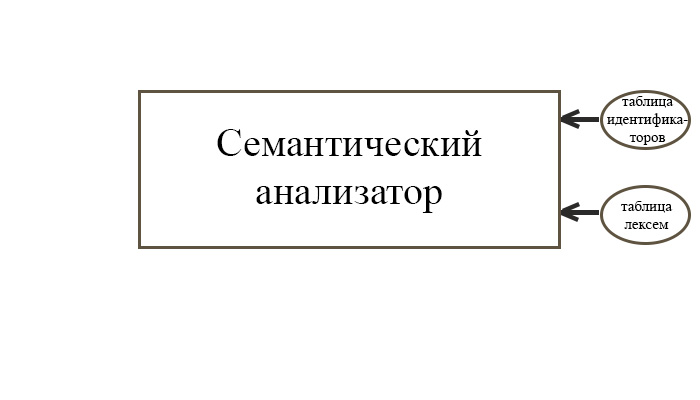


Рисунок 5.1 - Структура семантического анализатора

* 1. **Функции семантического анализатора**

Функции, представляющие проверку правил, представлены в таблице 5.1. Некоторые проверки встроены непосредственно в код этапов транслятора.

Таблица 5.1 - Семантические проверки

|  |  |
| --- | --- |
| Функция | Описание |
| Typecheck | Проверка, на соответствие типов формальных и фактических параметров |
| FuncRet | Проверка на возврат функции нужного типа данных |
| DoNotChangeParam | Проверка неизменности передаваемых параметров |
| ParamCheck | Проверка, на соответствие количества формальных и фактических параметров |

* 1. **Структура и перечень сообщений семантического анализатора**

Префикс сообщений “SEM:”. Сообщения, генерируемые при выполнении семантических проверок, представлены в таблице 5.2.

Таблица 5.2 - Перечень сообщений

|  |  |
| --- | --- |
| Код | Сообщение |
| 125 | Отсутствие точки входа main |
| 126 | Дублирование точки входа main |
| 300 | Несовпадение типов параметров функции |
| 301 | Несовпадение количества параметров функции |
| 302 | Присвоение аргументу не соответствующего типа |
| 303 | Недопустимые операции со строками |
| 304 | Операция над разными типами |
| 305 | Функция возвращает неверный тип |
| 306 | Вызов не существующей функции |
| 307 | Число выходит за границы допустимого |
| 308 | Нельзя изменять строковые параметры функции |

* 1. **Принцип обработки ошибок**

Все семантические ошибки являются критическими, из-за чего транслятор прекращает свою работу и в протокол работы транслятора выводится соответствующее сообщение об ошибке.

* 1. **Контрольный пример**

Результат работы семантических функций описан в главе 8.

**Глава 6. Преобразование выражений**

**6.1 Выражения, допускаемые языком**

В языке SRA-2019 допускаются выражения с использованием числовых идентификаторов и литералов. Также предусмотрены следующие арифметические операции:

* сложения: «+» ;
* вычитания: «-» ;
* умножения: «\*» ;
* деления: «/» ;

Также есть возможность изменять приоритет выполнения арифметических операций при помощи скобок:

* «(»;
* «)»;

За счет заключения операции в скобки происходит повышение его приоритета, а значит при вычислении всего выражения операция с более высоким приоритетом будет вычисляться раньше операции с меньшим приоритетом.

Возможна конкатенация строк посредством оператора сложения. Приоритетность операций представлена в таблице 6.1. Чем выше число, тем выше и приоритет.

Таблица 6.1 - Приоритетность операций

|  |  |
| --- | --- |
| Операция | Приоритет |
| «(», «)» | 1 |
| «+», «-» | 2 |
| «\*», «/» | 3 |

**6.2 Польская запись и принцип ее построения**

Польская запись -форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Приоритетность операций приведена в таблице 6.1. Известен следующий принцип 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**

После этапов лексического и синтаксического анализа происходит преобразование в польскую запись, во время этапа генерации кода на язык ассемблера.

**6.4 Контрольный пример**

Контрольный пример разбора выражения содержится в таблице 6.2.

Таблица 6.2 - Разбор выражения

|  |  |  |
| --- | --- | --- |
| Выражение | Стек | Результат |
| (x + x)\*k |  |  |
| x + x)\*k | ( |  |
| + x)\*k | ( | x |
| x)\*k | ( + | x |
| )\*k | ( + | x x |
| \*k |  | x x + |
| k | \* | x x + |
|  | \* | x x + k |
|  |  | x x + k \* |

# Глава 7. Генерация кода

## 7.1 Структура генератора кода

Заключительным этапом трансляции языка SRA-2019 является генерация кода. Таблицы лексем и идентификаторов подаются Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. Выходной файл на языке ассемблера, который будет являться результатом работы транслятора, в соответствие с таблицей лексем. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода SRA-2019 представлена на рисунке 7.1.



Рисунок 7.1 – Структура генератора кода

## 7.2 Представление типов данных в памяти

Элементы таблицы идентификаторов расположены сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке SRA-2019 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка SRA-2019 и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке SRA-2019 | Тип идентификатора на языке ассемблера | Пояснение |
| number | SDWORD | Хранит целочисленный тип данных. |
| string | BYTE | Хранит указатель на начало строки. Строка должна завешаться нулевым символом. |

## 7.3 Статическая библиотека

В языке SRA-2019 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически.

Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные конечному пользователю. Эти функции представлены в таблице 1.9.

Таблица 1.9 - Дополнительные функции стандартной библиотеки

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void outint(int i) | Функции для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| void outstr(char\* s) | Функции для вывода в стандартный поток значения строкового идентификатора/литерала. |

## 7.4 Особенности алгоритма генерации кода

В языке SRA-2019 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2

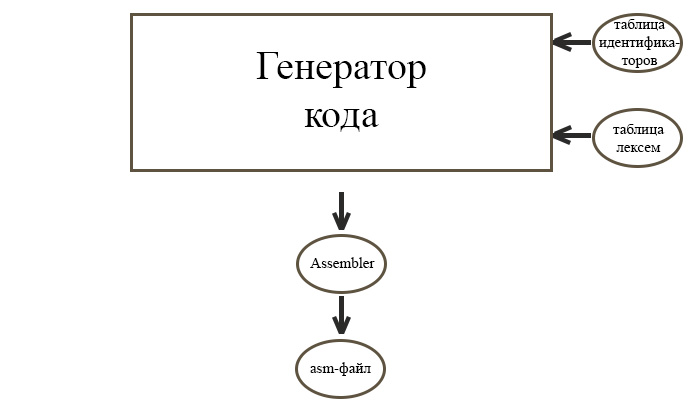


Рисунок 7.2 - Структура генератора кода

## 7.5 Контрольный пример

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

# Глава 8. Тестирование транслятора

В данной главе описаны возможные ошибки, возникающие на различных этапах работы транслятора. Результат тестирования представлен в таблице 8.1.

Таблица 8.1 - Тестирование транслятора

|  |  |
| --- | --- |
| Исходный код | Генерируемая ошибка |
| main  {  var string a = "string;  } | Ошибка 113: SYN:Отсутствуют закрывающие кавычки. Строка: 3 |
| main  {  var string abcdabcdabcd = "string";  } | Ошибка 120: LEX:Превышена максимальная длина идентификатора. Усечение до 10. Строка: 3. |
| number function а()  {  return 1;  };  number function а()  {  return 1;  };  main  {  var string a = "string";} | Ошибка 121: LEX:Дублирование имени функции. Строка: 5. |
| main  {  var number a;  var number a = "string";  } | Ошибка 122: LEX:Дублирование имени идентификатора или отсутствие области видимости. Строка: 4. |
| main  {  var number a#;  } | Ошибка 123: LEX:Неизвестная лексема. Строка: 3. |
| main  {  a = 1;  } | Ошибка 124: LEX:Неизвестный идентификатор. Строка: 3. |
| {  var number a = 1;  } | Ошибка 125: SEM:Отсутствует точка входа “main”. |

Продолжение Таблицы 8.1

|  |  |
| --- | --- |
| Исходный код | Генерируемая ошибка |
| main  {  var number a = 1;  }  main  {} | Ошибка 126: SEM:Дублирование точки входа “main”. |
| main  {  print 1; | Ошибка 200: SYN:Неверная структура программы. |
| main  {  print 1;;  } | 201: строка 3, SYN:Ошибочный оператор |
| main  {  print 1++;  } | 202: строка 3, SYN:Ошибка в выражении |
| main  {  print 1+(1+1;  } | 203: строка 3, SYN:Ошибка в расстановке знаков или скобок |
| number function a(number number a)  {  print 1;  };  main  {  print 1;  } | 204: строка 1, SYN:Ошибка в формальных параметрах |
| number function a()  {  return 1;  };  main  {  print a(var);  } | 205: строка 7, SYN:Ошибка в параметрах вызываемой функции |

Продолжение Таблицы 8.1

|  |  |
| --- | --- |
| Исходный код | Генерируемая ошибка |
| number function a(number a)  {  return 1;  };  main  {  print a("string");  } | Ошибка 300: SEM:Несовпадение типов параметров функции. Строка: 7. |
| number function a()  {  return 1;  };  main  {  print a(1);  } | Ошибка 301: SEM:Несовпадение количества параметров функции. Строка: 6. |
| main  {  var string a = 1;  } | Ошибка 302: SEM:Присвоение аргументу не соответствующий тип. Строка: 3. |
| main  {  string "string" / "string";  } | Ошибка 303: SEM:Недопустимые операции со строками. Строка: 3. |
| main  {  print "string" + 1;  } | Ошибка 304: SEM:Операция над разными типами. Строка: 3. |
| string function c()  {  return 1;  };  main  {  print "string";  } | Ошибка 305: SEM:Функция возвращает неверный тип. Строка: 3. |
| main  {  var string c;  print с(1);  } | Ошибка 306: SEM:Вызов не существующей функции. Строка: 4. |

Продолжение Таблицы 8.1

|  |  |
| --- | --- |
| Исходный код | Генерируемая ошибка |
| main  {  var number а;  a = 111111111111111;  } | Ошибка 307: SEM:Число выходит за границы допустимого. Строка: 4. |
| number function c(string a, string b)  {a = b;  print a;  };  main  {  var number a;  } | Ошибка 308: SEM:Нельзя изменять строковые параметры функции. Строка: 3. |

Таким образом, при встрече ошибок в процессе работы транслятора программа выводит сообщение об ошибке в файл и завершает работу.

# 

# Заключение

* Реализованы 4 арифметических оператора для вычисления выражений;
* Реализованы два типа данных;
* реализована конкатенация строк через оператор “+”;
* поддерживается оператор вывода;
* Реализовано 3 функции для двух типов данных;
* присутствует подключаемая стандартная библиотека;
* обрабатывается 30 возможных исключительных ситуаций.

# Приложение А

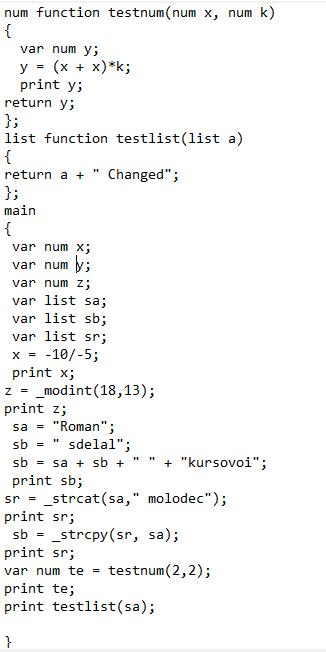


Рис. 8 - Исходный код

# Приложение Б

enumber IDDATATYPE {

NDEFD = 0, //неопределен

INT = 1, //ingeter

STR = 2 }; //string

enumber IDTYPE

{

NDEFT = 0, //неопределен

V = 1, //переменная

F = 2, //функция

L = 3, //литерал

P = 4 //параметр

};

struct Entry // строка таблицы идентификаторов

{

int idxfirstLE; // индекс первой строки в таблице лексем

char id[30] = "";// идентефикатор (автоматически усекается до ID\_MAXSIZE)

IDDATATYPE iddatatype; // тип данных

IDTYPE idtype;

char\* scope;

union

{

int vint;

struct

{

char len;

char str[TI\_STR\_MAXSIZE - 1];

} vstr[TI\_STR\_MAXSIZE];

}value;

void reset()

{

memset(id, NULL, ID\_MAXSIZE);

idxfirstLE = TI\_NULLIDX;

iddatatype = IT::NDEFD;

idtype = IT::NDEFT;

}

};

struct IdTable // экземпляр таблицы идентефикаторов

{

int maxsize;// емкость таблицы идентефикаторов < TI\_MAXSIZE

int size; // текущий размер таблицы идентефикаторов < maxsize

Entry\* table;// массив строк таблицы идентефикаторов

};

Рис. 9 - Структуры данных, используемые на фазе лексического анализа

**Приложение В**

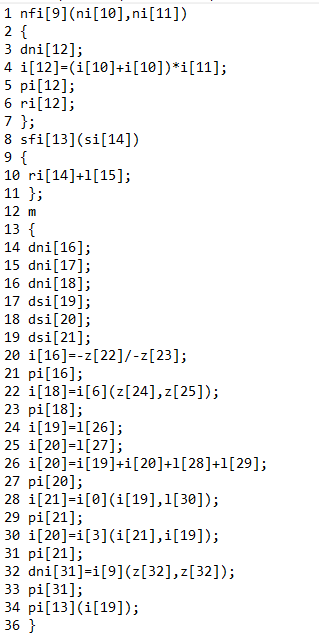


Рис. 10 - Таблица лексем

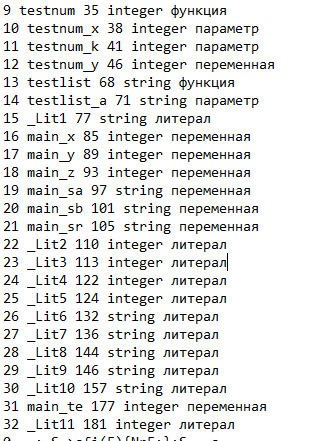


Рис. 11 - Таблица идентификаторов

# Приложение Г

Greibach greibach(NS('S'), TS('$'), // стартовый символ, дно стека

6, // количество правил

Rule(NS('S'), GRB\_ERROR\_SERIES + 0, // Неверная структура программы

9, // S->m{N}; | pi(F){N}S

Rule::Chain(14, TS('n'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),

Rule::Chain(14, TS('s'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),

Rule::Chain(13, TS('n'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),

Rule::Chain(13, TS('s'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),

Rule::Chain(12, TS('n'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),

Rule::Chain(12, TS('s'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),

Rule::Chain(13, TS('n'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),

Rule::Chain(13, TS('s'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),

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

),

Rule(NS('N'), GRB\_ERROR\_SERIES + 1, // Ошибочный оператор

25,//c(i>i)N c(i<i)N

Rule::Chain(5, TS('d'), TS('n'), TS('i'), TS(';'), NS('N')), //объявление integer

Rule::Chain(5, TS('d'), TS('s'), TS('i'), TS(';'), NS('N')), //объявление strin

Rule::Chain(5, TS('d'), TS('k'), TS('i'), TS(';'), NS('N')), //объявление bool

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

Rule::Chain(7, TS('d'), TS('n'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')), // объявл.+иниц. integer

Rule::Chain(7, TS('d'), TS('s'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')), // объявл.+иниц. string

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

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

Rule::Chain(8, TS('i'), TS('='), TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('N')), // присваение ид-ру рез-та вызова ф-ии

Rule::Chain(7, TS('p'), TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('N')), // вывод результата ф-ии

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

Rule::Chain(9, TS('i'), TS('='), TS('i'), TS('('), NS('W'), TS(')'), NS('M'), TS(';'), NS('N')),

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

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

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

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

Rule::Chain(6, TS('d'), TS('n'), TS('i'), TS('='), NS('E'), TS(';')),

Rule::Chain(6, TS('d'), TS('s'), TS('i'), TS('='), NS('E'), TS(';')),

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

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

Rule::Chain(7, TS('i'), TS('='), TS('i'), TS('('), NS('W'), TS(')'), TS(';')),

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

Rule::Chain(5, TS('p'), TS('i'), TS('('), TS(')'), TS(';')),

Rule::Chain(8, TS('i'), TS('='), TS('i'), TS('('), NS('W'), TS(')'), NS('M'), TS(';')),

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

),

Rule(NS('E'), GRB\_ERROR\_SERIES + 2, // Ошибка в выражении

17, //E -> i | iM | (E) | (E)M | i(W) | i(W)M | nM | n

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

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

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

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

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

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

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

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

Rule::Chain(5, TS('-'), TS('('), NS('E'), TS(')'), NS('M')),

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

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

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

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

Rule::Chain(3, TS('-'), TS('i'), NS('M')),

Rule::Chain(3, TS('-'), TS('z'), NS('M')),

Rule::Chain(2, TS('-'), TS('z')),

Rule::Chain(2, TS('-'), TS('i'))

),

Rule(NS('M'), GRB\_ERROR\_SERIES + 3, // Ошибка в выражении

4, //M -> vE

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

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

Rule::Chain(2, TS('\*'), NS('E')),

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

),

Rule(NS('F'), GRB\_ERROR\_SERIES + 4, // Ошибка в параметрах функции +

4, //F -> ti | ti,F

Rule::Chain(2, TS('n'), TS('i')),

Rule::Chain(4, TS('n'), TS('i'), TS(','), NS('F')),

Rule::Chain(2, TS('s'), TS('i')),

Rule::Chain(4, TS('s'), TS('i'), TS(','), NS('F'))

),

Rule(NS('W'), GRB\_ERROR\_SERIES + 5, // Ошибки в параметрах вызываемой +

6, //W -> i | l | i,W | l,W

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

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

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

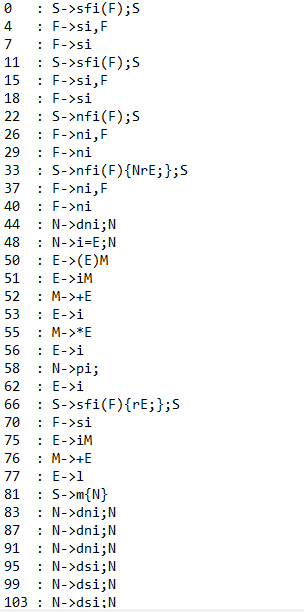
Rule::Chain(3, TS('i'), TS(','), NS('W')),

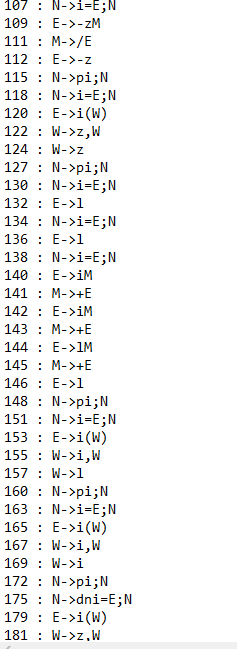
Rule::Chain(3, TS('l'), TS(','), NS('W')),

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

Рис.11 - Структура данных грамматики Грейбах

# Приложение Д





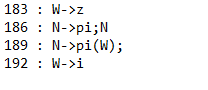


Рис. 12 - Дерево разбора

## Приложение Е

Таблица 1 – Результат генерации кода

|  |
| --- |
| .586  .model flat, stdcall  includelib userlib.lib  includelib kernel32.lib  includelib libucrt.lib  ExitProcess PROTO : DWORD  \_strcpy PROTO : DWORD, : DWORD  \_strcat PROTO : DWORD, : DWORD  outstr PROTO : DWORD  outint PROTO : SDWORD  \_modint PROTO : SDWORD, : SDWORD  .stack 4096  .const  overflow db 'ERROR: VARIABLE OVERFLOW', 0  null\_division db 'ERROR: DIVISION BY ZERO', 0  \_Lit1 BYTE " Changed", 0  \_Lit2 SDWORD 10  \_Lit3 SDWORD 5  \_Lit4 SDWORD 18  \_Lit5 SDWORD 13  \_Lit6 BYTE "Roman", 0  \_Lit7 BYTE " sdelal", 0  \_Lit8 BYTE " ", 0  \_Lit9 BYTE "kursovoi", 0  \_Lit10 BYTE " molodec", 0  \_Lit11 SDWORD 2  .data  testnumber\_y SDWORD 0  main\_x SDWORD 0  main\_y SDWORD 0  main\_z SDWORD 0  main\_sa BYTE 255 DUP(0)  main\_sb BYTE 255 DUP(0)  main\_sr BYTE 255 DUP(0)  main\_te SDWORD 0  .code  testnumber\_proc PROC, testnumber\_x : SDWORD, testnumber\_k : SDWORD  push testnumber\_x  push testnumber\_x  pop eax  pop ebx  add eax, ebx  jo EXIT\_OVERFLOW  push eax  push testnumber\_k  pop eax  pop ebx  imul eax, ebx  jo EXIT\_OVERFLOW  push eax  pop testnumber\_y  push testnumber\_y  call outint  push testnumber\_y  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call outstr  push - 1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstr  push - 2  call ExitProcess  EXIT:  pop eax  ret 8  testnumber\_proc ENDP  teststring\_proc PROC, teststring\_a : DWORD  push teststring\_a  push offset \_Lit1  call \_strcat  jo EXIT\_OVERFLOW  push eax  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call outstr  push - 1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstr  push - 2  call ExitProcess  EXIT:  pop eax  ret 4  teststring\_proc ENDP  main PROC  push \_Lit2  pop eax  neg eax  push eax  push \_Lit3  pop eax  neg eax  push eax  pop ebx  pop eax  test ebx,ebx  jz EXIT\_DIV\_ON\_NULL  cdq  idiv ebx  push eax  pop main\_x  push main\_x  call outint  push \_Lit5  push \_Lit4  call \_modint  push eax  pop main\_z  push main\_z  call outint  push offset \_Lit6  push offset main\_sa  call \_strcpy  push offset \_Lit7  push offset main\_sb  call \_strcpy  push offset main\_sa  push offset main\_sb  call \_strcat  jo EXIT\_OVERFLOW  push eax  push offset \_Lit8  call \_strcat  jo EXIT\_OVERFLOW  push eax  push offset \_Lit9  call \_strcat  jo EXIT\_OVERFLOW  push eax  push offset main\_sb  call \_strcpy  push offset main\_sb  call outstr  push offset main\_sa  push offset \_Lit10  call \_strcat  push eax  push offset main\_sr  call \_strcpy  push offset main\_sr  call outstr  push offset main\_sa  push offset main\_sr  call \_strcpy  push eax  push offset main\_sb  call \_strcpy  push offset main\_sr  call outstr  push \_Lit11  push \_Lit11  call testnumber\_proc  push eax  pop main\_te  push main\_te  call outint  push offset main\_sa  call teststring\_proc  push eax  call outstr  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call outstr  push - 1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstr  push - 2  call ExitProcess  EXIT:  push 0  call ExitProcess  main ENDP  end main |

# Литература

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

2. Смелов, В.В. Курс лекций по предмету языки программирования – 2016