МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПЕУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий   
Кафедра программной инженерии

Утверждаю

Заведующая кафедрой

\_\_\_\_\_\_\_\_\_\_\_\_ Н.В. Пацей

подпись инициалы и фамилия

“\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_2020г.

**ЗАДАНИЕ**

**к курсовому проектированию**

**по дисциплине** "Языки программирования"

Специальность: ПОИТ Группа: 5

Студент: Демьянов Владислав Русланович

(фамилия, имя, отчество)

**1. Тема проекта** Разработка компилятора DVR-2020

утверждена приказом по университету от «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г. №

**2. Срок сдачи студентом законченного проекта:** \_\_\_ декабря 2020 г.

**3. Исходные данные к проекту:**

Разработка программы осуществляется на языке C++ (стандартизации International Standard ISO/IEC 14882:2014(E) Programming Language C++ 14) в среде разработки Visual Studio 2019 (v142). Операционная система под которой происходит разработка Windows 7 (64-bit). Типы данных: posint, symbol, string. Целочисленные операции: сравнения; равенство, меньше, больше, меньше или равно, больше или равно. Литералы: целые (десятичное), строковые и символьные; Функции стандартной библиотеки: округление до ближайшего целого: RoundToNearest, генератор псевдослучайного числа от 0 до 1: Random. Инструкции: оператор цикла, операторы ветвления. Оператор вывода.

**4. Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов):**

Введение

1) Спецификация языка программирования

2) Структура транслятора

3) Разработка лексического анализатора

4) Разработка синтаксического анализатора

5) Разработка семантического анализатора

6) Вычисление выражений

7) Генерация кода

8) Тестирование транслятора (Разработка и тестирование интерпретатора)

Приложения

Литература

**5. Перечень графического материала (с точным указанием обязательных чертежей)**

1) Граф

**6. Консультанты по проекту с указанием относящихся к ним разделов проекта**

|  |  |
| --- | --- |
| Раздел | Консультант |
| Разработка синтаксического и семантического анализатора. | Наркевич А. С. |
| Генерация кода. Разработка тестовых примеров. | Наркевич А. С. |
| Оформление пояснительной записки к курсовому проект. | Наркевич А. С. |
|  |  |

**7. Календарный план**

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Наименование этапов курсового проекта | Срок выполнения этапов проекта | Примечание |
| 1 | Спецификация специализированного языка DVR-2020 | 22.10.2020 |  |
| 2 | Разработка лексического анализатора | 29.10.2020 |  |
| 3 | Разработка синтаксического анализатора | 12.11.2020 |  |
| 4 | Разработка семантического анализатора | 26.11.2020 |  |
| 5 | Генерация кода | 3.12.2020 |  |
| 6 | Тестирование компилятора | 10.12.2020 |  |
| 7 | Оформление пояснительной записки к курсовому проект | 12.12.2020 |  |
| 8 | Сдача проекта | 17.12.2020 |  |

**8. Дата выдачи задания 15.09.2018**

Руководитель Фамилия И.О.

(подпись)

Задание принял к исполнению Фамилия И.О.

**Содержание**

[ВВЕДЕНИЕ 6](#_Toc59056290)

[ОСНОВНАЯ ЧАСТЬ ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ 7](#_Toc59056291)

[1. Спецификация языка программирования DVR-2020 7](#_Toc59056292)

[1.1 Характеристики языка программирования DVR-2020 7](#_Toc59056293)

[1.2 Определение алфавита языка программирования 7](#_Toc59056294)

[1.3 Применяемы сепараторы 7](#_Toc59056295)

[1.4 Применяемые кодировки 8](#_Toc59056296)

[1.5 Типы данных 8](#_Toc59056297)

[1.6 Преобразование типов данных 8](#_Toc59056298)

[1.7 Идентификаторы 8](#_Toc59056299)

[1.8 Литералы 9](#_Toc59056300)

[1.9 Объявление данных 9](#_Toc59056301)

[1.10 Инициализация данных 10](#_Toc59056302)

[1.11 Инструкции языка 10](#_Toc59056303)

[1.12 Операции языка 11](#_Toc59056304)

[1.13 Выражения и их вычисление 12](#_Toc59056305)

[1.14 Конструкции языка 13](#_Toc59056306)

[1.15 Область видимости идентификаторов 13](#_Toc59056307)

[1.16 Семантические проверки 13](#_Toc59056308)

[1.17 Распределение оперативной памяти на этапе выполнения 14](#_Toc59056309)

[1.18 Стандартная библиотека и её состав 14](#_Toc59056310)

[1.19 Ввод и вывод данных 14](#_Toc59056311)

[1.20 Точка входа 14](#_Toc59056312)

[1.21 Препроцессор 14](#_Toc59056313)

[1.22 Соглашение о вызовах 14](#_Toc59056314)

[1.23 Объектный код 14](#_Toc59056315)

[1.24 Классификация сообщений транслятора 15](#_Toc59056316)

[1.25 Контрольный пример 16](#_Toc59056317)

[2. Структура транслятора 17](#_Toc59056318)

[2.1 Компоненты транслятора их назначение и принципы взаимодействия 17](#_Toc59056319)

[2.2 Перечень входных параметров транслятора 18](#_Toc59056320)

[2.3 Протоколы, формируемые транслятором 18](#_Toc59056321)

[3. Разработка лексического анализатора 18](#_Toc59056322)

[3.1 Структура лексического анализатора 19](#_Toc59056323)

[3.2 Контроль входных символов 20](#_Toc59056324)

[3.3 Удаление избыточных символов 21](#_Toc59056325)

[3.4 Перечень ключевых слов 22](#_Toc59056326)

[3.5 Основные структуры данных 23](#_Toc59056327)

[3.6 Структура и перечень сообщений лексического анализатора 24](#_Toc59056328)

[3.7 Принцип обработки ошибок 24](#_Toc59056329)

[3.8 Параметры лексического анализатора 24](#_Toc59056330)

[3.9 Алгоритм лексического анализа 24](#_Toc59056331)

[3.10 Контрольный пример 24](#_Toc59056332)

[4. Разработка синтаксического анализатора 25](#_Toc59056333)

[4.1 Структура синтаксического анализатора 25](#_Toc59056334)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 25](#_Toc59056335)

[4.3 Построение конечного магазинного автомата 27](#_Toc59056336)

[4.4 Основные структуры данных 28](#_Toc59056337)

[4.5 Контрольный пример 30](#_Toc59056338)

[5. Разработка семантического анализатора 32](#_Toc59056339)

[5.1 Структура семантического анализатора 32](#_Toc59056340)

[5.2 Контрольный пример 33](#_Toc59056341)

[6. Вычисление выражений 35](#_Toc59056342)

[6.1 Выражения, допускаемые языком 35](#_Toc59056343)

[6.2 Польская запись и принцип её построения 35](#_Toc59056344)

[6.3 Программная реализация обработки выражений 36](#_Toc59056345)

[6.4 Контрольный пример 37](#_Toc59056346)

[7. Генерация кода 38](#_Toc59056347)

[7.1 Структура генератора кода 38](#_Toc59056348)

[7.2 Статическая библиотека 38](#_Toc59056349)

[7.3 Контрольный пример 39](#_Toc59056350)

[8. Тестирование транслятора 40](#_Toc59056351)

[8.1 Результаты тестирования 40](#_Toc59056352)

[ЗАКЛЮЧЕНИЕ 41](#_Toc59056353)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 42](#_Toc59056354)

# **Введение**

В пояснительной записке представлена спецификация языка программирования DVR-2020.

Целью выполнения курсового проекта по дисциплине «Языки программирования» является приобретение навыков разработки системы программирования (трансляторов, интерпретаторов).

В процессе выполнения курсового проекта:

− получены навыки проектирования систем программирования;

− изучены основы теории формальных грамматик и основы общей теории компиляторов;

− приобретены навыки разработки программного обеспечения систем программирования.

В процесс выполнения курсового проекта входит:

− разработка спецификации языка программирования;

− разработка программной реализации лексического анализатора;

− разработка программной реализации синтаксического анализатора;

− разработка программной реализации семантического анализатора;

− разработка программной реализации генератора кода;

− выполнение тестирования, разработанного программного обеспечения;

− подготовка пояснительной записки к курсовому проекту.

# **Основная часть пояснительной записки**

## **1. Спецификация языка программирования DVR-2020**

### **1.1 Характеристики языка программирования DVR-2020**

Описание области применения языка программирования и его свойств: компилируемый или интерпретируемый; уровень языка программирования, поддерживаемые парадигмы программирования; контроль типов (статическая, динамическая типизация, строго или нестрого типизированный язык) и т.п.

Область применения. Язык программирования DVR-2020 может использоваться для выполнения простых математических операций, вывода информации на экран, выполнения цикличной работы.

Язык является компилируемым.

### **1.2 Определение алфавита языка программирования**

Описание множества символов, используемых для записи конструкций языка программирования, а также символы, применяемые на этапе выполнения.

Для записи конструкций языка программирования допускается использование символов:

* Буквы латинского алфавита
* Буквы кириллицы (допускаются в литералах)
* Цифры

### **1.3 Применяемы сепараторы**

Символы или зарезервированные слова, применяемые для ограничения программных конструкций:

|  |  |
| --- | --- |
| **Сепаратор** | **Описание** |
| Пробел | Может использоваться везде кроме идентификаторов и ключевых слов. |
| ; | Разделитель инструкций. |
| () | Параметры. |
| { | Начало программного блока. |
| } | Конец программного блока. |
| , | Перечисление программных лексем. |
|  |  |
| *Сепараторы. Таблица №1* | |

### **1.4 Применяемые кодировки**

Кодировка символов, используемая для написания текста на разрабатываемом языке программирования.

В разрабатываемом языке программирования используется кодировка символов Window-1251.

### **1.5 Типы данных**

Типы данных, их размер в байтах и допустимые диапазоны значений, принцип размещения в памяти, инициализация по умолчанию, контроль типов, применяемые операции:

Таблица №2 Типы данных.

|  |  |
| --- | --- |
| **Тип данных** | **Описание** |
| posint | Беззнаковые целочисленные данные  (четыре байта, диапазон от 0 до 231-1), автоматическая инициализация 0, LE. |
| string | Строка, любые символы (максимально 255 символов, первый байт – длина строки), автоматическая инициализация строкой длины 0. |
| symbol | Любой символ, автоматическая инициализация пустым символом. |

### **1.6 Преобразование типов данных**

Описание допустимости преобразования типов данных: явные и неявные преобразования, операторы преобразования типов (если они допускаются языком).

В языке программирования DVR-2020 поддерживаются только неявные преобразования.

### **1.7 Идентификаторы**

Идентификатор – переменная, которая является именованным хранилищем данных.

Правила, определяющие понятие «идентификатор»:

1. <строчная буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
2. <прописная буква> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
3. <цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
4. <идентификатор> ::= <строчная буква> {(<прописная буква> | <цифра> | <строчная буква>)}
5. размер ограничен 20-ю символами
6. идентификатор не может совпадать с ключевыми словами

Правильные идентификаторы: idN1, id;

Неправильные идентификаторы: Id, 1id.

### **1.8 Литералы**

Литералом является постоянное значение, у которого нет имени.

Правила, определяющие понятие «литерал»:

1. Правила для литерала, который является целым беззнаковым числом:
2. <цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
3. <целое без знака> ::= <цифра>
4. <целое без знака литерал> := <целое без знака> {<целое без знака>}
5. Правила для литерала, который является строкой:
6. <цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
7. <строчная буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | й | ц | у | к | е | н | г | ш | щ | з | х | ъ | ф | ы | в | а | п | р | о | л | д | ж | э | я | ч | с | м | и | т | ь | б | ю
8. <прописная буква> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | Й | Ц | У | К | Е | Н | Г | Ш | Щ | З | Х | Ъ | Ф | Ы | В | А | П | Р | О | Л | Д | Ж | Э | Я | Ч | С | М | И | Т | Ь | Б | Ю
9. <октоторп> ::= #
10. <литерал строка> ::= <октоторп> {(<строчная буква> | <прописная буква> | <цифра>)} <октоторп>

Правильный литерал: #RrйЙ12#;

Неправильный литерал: RrйЙ12.

### **1.9 Объявление данных**

Правила, способы и место объявления и/или инициализации переменных в тексте программы:

|  |  |
| --- | --- |
| **Обозначение** | **Описание** |
| $ | Объявить переменную. |
| <тип данных> | Тип данных, переменной.  см. в пункте *1.5 Типы данных*  posint | string | symbol |
| <идентификатор> | см. в пункте *1.7 Идентификаторы* |
| = | Присвоить значение. |
| <литерал> | см. в пункте *1.8 Литералы* |
| <выражение> | (<литерал> | <идентификатор>) {{(<идентификатор> | <литерал>)} (+ | - | / | \*) {(<идентификатор> | <литерал>)}} |
| ; | Конец инициализации. |
| *Объявление данных. Таблица №3* | |

1. Объявление переменной:

$ <тип данных> <идентификатор>;

1. Инициализация переменной:
2. при объявлении:

$ <тип данных> <идентификатор> = <выражение>;

1. после объявления:

<идентификатор> = <выражение>;

### **1.10 Инициализация данных**

Правила инициализации данных, вид инициализации (явный, неявный), применяемая инициализация по умолчанию, примеры. Обозначения [*см. в Таблица №3*](#Таблица3).

1. правила инициализации:
2. при объявлении:

$ <тип данных> <идентификатор> = <выражение>;

1. после объявления:

<идентификатор> = <выражение>;

1. тип выражения должен совпадать с типом идентификатора.
2. вид инициализации:
3. инициализация по умолчанию [*см. в Таблица №2 Типы данных*](#Таблица2).
4. примеры инициализации переменной:

* $ posint a = 10;
* $ string b = #abcd#;
* $ letter d;

d = #L#;

### **1.11 Инструкции языка**

Инструкции языка, их синтаксис и принцип применения. Примеры написания.

1. Инструкции языка, их синтаксис.
2. $ <тип данных> идентификатор; – объявление переменных;
3. $ <тип данных> func идентификатор (<тип данных> идентификатор1, … , <тип данных> идентификаторN); – объявление внешних функций;
4. = присвоение значения;
5. puts <идентификатор или литерал>; – вывод в стандартный поток вывода
6. цикл со предусловием:

until (<условие>)

…тело цикла

loop;

1. оператор ветвления:

if (<условие>)

…тело оператора ветвления

end;

или

if (<условие>)

…тело оператора ветвления

else

…тело оператора ветвления

end;

1. Примеры написания.
2. объявление переменной беззнакового целочисленного типа (posint):

$ posint a;

1. объявление внешней функции и присваивание; Объявление переменной строкового типа. Присваивание в неё внешней функции (возвращаемый тип строковый (string), два параметра строкового и беззнакового целочисленного типа):

$ string result = $ string func fn(str a, posint b);

1. вывод в стандартный поток вывода:

puts #HelloWorld!#; – где #HelloWorld!# строковый литерал.

1. цикл со счётчиком:

$ posint i = 0;

until (i < 5)

puts (i);

i = i + 1;

loop;

В результате цикл отработает 5 раз. На каждой итерации цикла будет выводится в стандартный поток вывода значение переменной i.

Вывод: 01234;

### **1.12 Операции языка**

Операции, поддерживаемые языком, их приоритетность и основные свойства (ассоциативность, коммутативность и дистрибутивность), количество и допустимые типы используемых операндов и результата, порядок выполнения операций с одинаковым приоритетом, способ явного указания очерёдности выполнения операций.

1. Операции отношения.

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| == | эквивалентно — проверка на равенство; бинарный (posint, posint); |
| < | меньше; бинарный (posint, posint); |
| > | больше; бинарный (posint, posint); |
| <= | меньше или равно; бинарный (posint, posint); |
| >= | больше или равно; бинарный (posint, posint). |
| *Целочисленные операции сравнения. Таблица №4* | |

Операции отношения используются при организации условий и ветвлений. Результатом этих операций является 1 бит, значение которого равно 1, если результат выполнения операции — истина, и равно 0, если результат выполнения операции — ложь.

1. Арифметические операции. Расположены в порядке уменьшения приоритета.

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| \* | бинарный, умножение, (posint, posint); |
| / | бинарный, деление, (posint, posint); |
| + | бинарный, суммирование, (posint, posint);  Бинарный, конкатенация, (str, str); |
| - | бинарный, вычитание, (posint, posint); |
| *Арифметические операции. Таблица №5* | |

* результатом выполнения арифметических операций, приведённых в таблице *Арифметические операции. Таблица №5*, будут данные такого же типа, что используются в операндах, к которым применяется операция.
* арифметические операции выполняются слева направо.
* скобки позволяют переопределить порядок вычислений.

### **1.13 Выражения и их вычисление**

Определение выражений, допустимых в языке. Типы выражений, правила составления выражений, операции, используемые в выражениях, допустимые типы операндов, а также порядок вычисления подвыражений.

### **1.14 Конструкции языка**

Программные конструкции языка, используемые для управления процессом вычисления, их формальное описание и примеры, особенности работы с процедурами и/или функциями, программными блоками.

1. Главная функция (точка входа):

go

{

**}**;

1. Функции:

$ <тип данных> func <идентификатор> <<тип данных> идентификатор, …>

{

ret <идентификатор>;

};

1. Параметры:

Передаются по значению.

### **1.15 Область видимости идентификаторов**

Принципы видимости идентификаторов, используемые в языке: область видимости на уровне блоков кода, область видимости на уровне функций, при объявлении параметров функции, при объявлении идентификаторов вне функций.

1. сверху вниз;
2. параметры — только внутри функций;
3. объявление внутри функции — видно только внутри функции;
4. объявление идентификаторов вне функции — видно на уровне блока кода, в котором они объявлены, и во вложенных блоках кода (функциях);

### **1.16 Семантические проверки**

Перечисление и описание правил семантической проверки текста языка программирования.

* Каждый идентификатор должен быть объявлен только один раз (с учётом блочной структуры объявлений);
* Все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции;
* Типы переменных в выражениях должны быть согласованы между собой;
* При вызове процедур и функций количество и типы фактических параметров должны быть согласованы с количеством и типами формальных параметров;
* Каждая переменная должна быть определена до её первого использования при любом ходе выполнения программы;

### **1.17 Распределение оперативной памяти на этапе выполнения**

Описание принципов распределения памяти на этапе выполнения, виды областей памяти (область кода, статическая область, стек, динамическая область).

Статическая область. В область генерируются все литералы.

Динамическая область. В область генерируются переменные.

Область кода. В область генерируются инструкции языка. Также в ней используются данные из статической области и динамической.

### **1.18 Стандартная библиотека и её состав**

У языка DVR-2020 стандартная библиотека не реализована.

### **1.19 Ввод и вывод данных**

В языке программирования DVR-2020 предусмотрен оператор вывода данных (puts <идентификатор или литерал>). Он осуществляет работу с стандартным потоком stdout.

### **1.20 Точка входа**

Определение синтаксические правила, описывающего точку входа.

Точкой входа в программу является конструкция языка «go». Конструкция описана в [*1.14 Конструкции языка*](#Конструкции_языка).

Синтаксические правила, описывающие точку входа: g{NrE;}; | g{NrE;};S. Где N – правила для операторов, E – правила для выражений, S – правила для структуры программы.

### **1.21 Препроцессор**

В языке программирования DVR-2020 отсутствуют директивы препроцессора.

### **1.22 Соглашение о вызовах**

В языке программирования DVR-2020 применяется стандартное соглашение о вызовах stdcall.

Аргументы функций передаются через стек, справа налево. Аргументы, размер которых меньше 4-х байт, расширяются до 4-х байт. Очистку стека производит вызываемая подпрограмма.

### **1.23 Объектный код**

Целевым языком трансляции в языке программирования DVR-2020 является ассемблер.

### **1.24 Классификация сообщений транслятора**

Коды ошибок: 0-99 – системные ошибка, 100-109 – ошибки параметров, 110-119 – ошибки с файлами, 120-199 – ошибки лексического анализа, 200-599 – ошибки семантического анализа, 600-900 – ошибки синтаксического анализа.

|  |  |  |
| --- | --- | --- |
| **Стадия** | **Код** | **Сообщение** |
| Системные ошибки | 0 | Недопустимый код ошибки |
| 1 | Системный сбой |
| Ошибки с параметрами | 100 | Параметр -in должен быть задан |
| 104 | Превышена длина входного параметра |
| 105 | Параметр -asm должен быть задан |
| Ошибки с файлами | 110 | Ошибка при открытии файла с исходным кодом (-in) |
| 111 | Недопустимый символ в исходном файле (-in) |
| 112 | Ошибка при создании файла протокола (-log) |
| Лексический анализатор | 120 | Цепочка символов не распознана |
| 121 | Длина идентификатора превышает допустимую длину |
| 122 | Превышено макс кол-во строк в таблице идентификаторов |
| 123 | Превышено макс кол-во строк в таблице лексем |
| Семантический анализатор | 200 | Ошибка: переопределение идентификатора |
| 201 | Ошибка: идентификатор не определён, но используется |
| 202 | Ошибка: все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции |
| 203 | Ошибка: количество или типы фактических параметров не согласованы с количеством или типами формальных параметров |
| 204 | Ошибка: тип возвращаемого значения функции |
| Синтаксический анализатор | 600 | Неверная структура программы |
| 601 | Ошибочный оператор |
| 602 | Ошибка в выражении |
| 603 | Ошибка в арифметических операторах |
| 604 | Ошибка в параметрах функции |
| 605 | Ошибка в параметрах вызываемой функции |
| 606 | Ошибка в параметрах цикла или оператора ветвления |
| *Ошибки. Таблица №6* | | |

### **1.25 Контрольный пример**

Пример программы на языке программирования DVR-2020.

|  |
| --- |
| posint func print (string str, posint count) |
| { |
| $ posint i; |
| i = 0; |
| puts #InPrint#; |
| until (i < count) |
| puts str; |
| i = i + 1; |
| loop; |
| ret i; |
| }; |
|  |
| posint func makeCoef (posint x, posint y) |
| { |
| $ posint result; |
| result = x + y; |
| ret result; |
| }; |
|  |
| posint func makeCompute (posint x, posint y, posint z) |
| { |
| $ posint result; |
| result = 0; |
| result = ((x + y + z) / z) \* makeCoef(x, y); |
| ret result; |
| }; |
|  |
| go |
| { |
| $ string str; |
| $ posint result; |
| $ posint x; |
| $ posint y; |
| str = #cycle#; x = 3; y = 4; result = 0; |
| result = makeCompute(x, y, 2); |
| x = print(str, 5); |
| puts #Result:#; |
| puts result; |
| ret 0; |
| }; |

## **2. Структура транслятора**

В данном разделе описывается структура транслятора. Транслятор преобразует исходный текст программы в текст целевого языка. Получив на вход исходный текст, транслятор проверяет его принадлежность заданному языку и определяет набор грамматических правил языка. Процесс трансляции состоит из фаз: лексический анализ, синтаксический анализ, семантический анализ и генерация кода. На всех фазах трансляции применяется таблица идентификаторов, которая пополняется дополнительной информацией в ходе трансляции.

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

Определение транслятора, схема транслятора, поясняющая принцип его работы. Компоненты транслятора, их назначение, входные и выходные данные.

Транслятор – программа или техническое средство, выполняющее трансляцию программы.

Трансляция программы – преобразование программы, представленной на одном из языков программирования, в программу на другом языке.

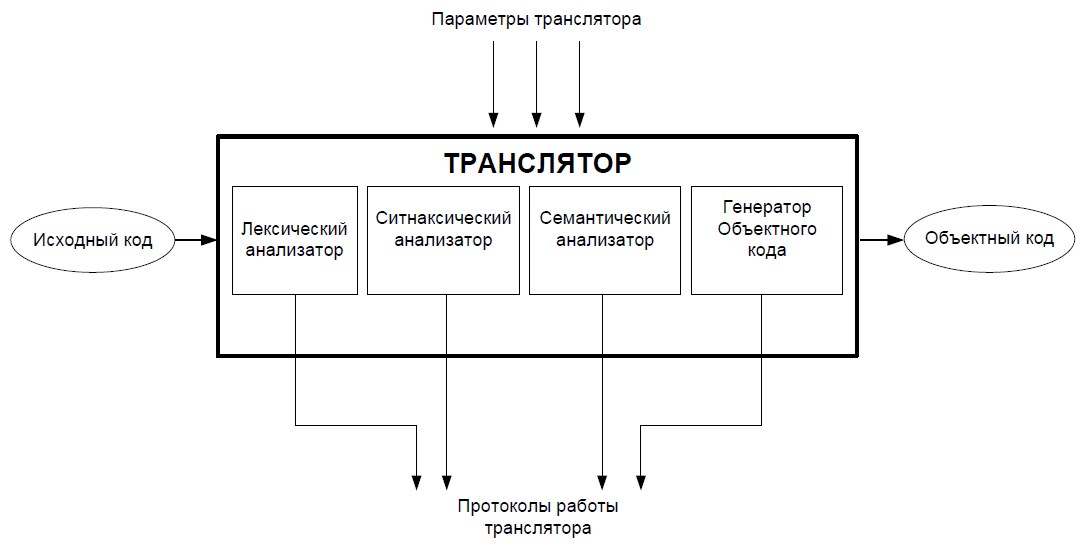


Рисунок 1 Схема транслятора

Входными данными является исходный язык (сама программа – исходный код). Выходными данными является целевой язык.

### **2.2 Перечень входных параметров транслятора**

|  |  |
| --- | --- |
| **Параметр** | **Описание** |
| -in:имя\_файла.txt | Создаёт файл с исходным кодом. Является обязательным параметром в языке программирования DVR-2020. |
| -log:имя\_файла.log | Создаёт файл с записями о событиях в хронологическом порядке. Параметр является необязательным. При его отсутствии создаётся автоматически. При автоматическом создании ему генерируется имя из полного названия файла с исходным кодом. |
| -asm:имя\_файла.asm | Создаёт файл с выходными данными. В него будет генерироваться целевой язык. Является обязательным параметром в языке программирования DVR-2020. |
| *Входные параметры транслятора. Таблица №7* | |

### **2.3 Протоколы, формируемые транслятором**

|  |  |
| --- | --- |
| **Файл** | **Описание** |
| file.txt | Файл с исходным кодом. |
| file.log | Файл с записями о событиях в хронологическом порядке. |
| file.asm | Файл с выходными данными. В него будет генерироваться целевой язык. |
| *Протоколы, формируемые транслятором. Таблица №8* | |

## **3. Разработка лексического анализатора**

Первая фаза работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором. Лексический анализатор преобразует исходный текст, заменяя лексические единицы языка их внутренним представлением – лексемами. Для описания лексики языка программирования применяются регулярные грамматики, относящиеся к типу 3 иерархии Хомского. Язык, заданный регулярной грамматикой, называется регулярным языком (типа 3 иерархии Хомского). Регулярный язык однозначно задается регулярным выражением, а распознавателями для регулярных языков являются конечные автоматы.

### **3.1 Структура лексического анализатора**

Определение лексического анализатора и его структурная схема. Описание входных данных, результатов работы анализатора, его параметры.

Лексический анализатор – первый из “слоёв” компилятора, отвечающий за выделение лексем для последующей обработки.

Лексема – минимальная единица некого словаря, представляющего язык. В роли лексемы могут служить служебные слова, операторы, идентификаторы и так далее.

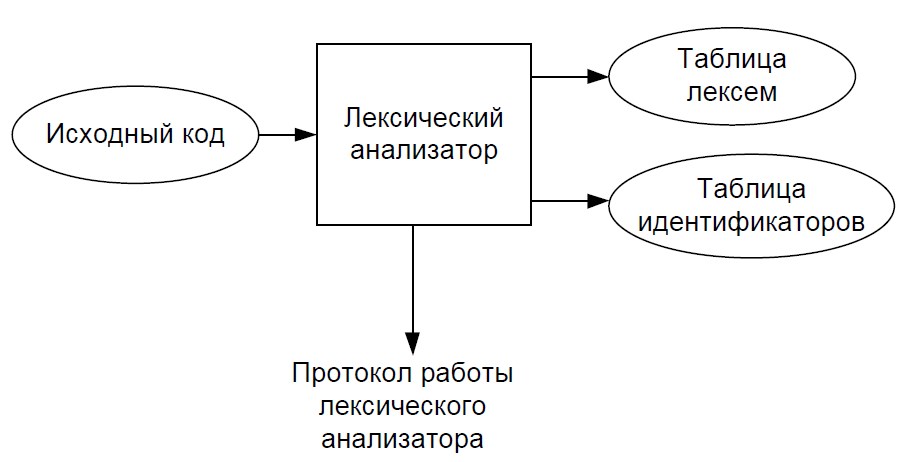


Рисунок 2 Структурная схема лексического анализатора.

Исходный код – входные данные лексического анализатора.

Таблица лексем и таблица идентификаторов – выходные данные лексического анализатора. Таблица лексем содержит токены, которые являются обозначениями соответствующих им лексем исходного кода. У токенов, которые являются идентификаторами или литералами, есть ссылки на соответствующие им элементы таблицы идентификаторов. Таблица идентификаторов содержит имена идентификаторов и лексем с соответствующими им значениями и ссылками на токены таблицы лексем.

### **3.2 Контроль входных символов**

Таблица разрешённых символов, используемая для контроля. Принцип её применения.

#define IN\_CODE\_TABLE {\

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::S,IN::S,IN::S,IN::S,IN::S,IN::S,IN::T,IN::S, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, IN::T,IN::S,IN::S,IN::S,IN::S,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::S,IN::T,IN::S,IN::T,IN::T, \

\

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T,IN::T, \

}

Приведённый выше код, реализует таблицу разрешённых символов. Она представляет собой расширенную таблицу ASCII.

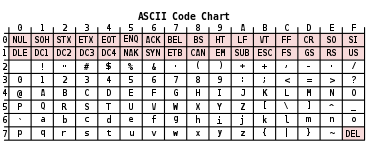


Рисунок 3 Таблица ASCII

В таблице (которая реализована кодом) есть запрещённые символы IN::F, разрешённые IN::T, символы сепараторы IN::S, игнорируемые символы IN::I.

В языке DVR-2020 один запрещённый символ – @.

### **3.3 Удаление избыточных символов**

Избыточный символ – символ, который не несёт в себе семантической и синтаксической нагрузки и не является ошибочным. Также удаление такого символа не должно нарушать синтаксической структуры.

Алгоритм удаления. Проходим по исходному коду. Признаки, по которым мы удаляем избыточные символы (пробел и табуляция):

1. Символ расположен до или после сепаратора.
2. Символ расположен вначале исходного кода.
3. Пустая строка

### **3.4 Перечень ключевых слов**

Таблица ключевых слов языка, сепараторы, символы операций, соответствующие им лексемы и регулярные выражения.

|  |  |  |
| --- | --- | --- |
| **Ключевые слова, сепараторы, символы операций** | **Лексемы** |  |
| == | w |  |
| <= | w |  |
| >= | w |  |
| = | = |  |
| > | w |  |
| < | w |  |
| / | v |  |
| \* | v |  |
| - | v |  |
| + | v |  |
| ( | ( |  |
| ) | ) |  |
| { | { |  |
| } | } |  |
| , | , |  |
| ; | ; |  |
| posint | t |  |
| symbol | t |  |
| string | t |  |
| [id] | i |  |
| [symbol\_lex] | c |  |
| [string\_lex] | s |  |
| [posint\_lex] | n |  |
| if | q |  |
| end | d |  |
| until | u |  |
| loop | l |  |
| else | e |  |
| func | f |  |
| $ | $ |  |
| ret | r |  |
| puts | p |  |
| go | g |  |
| *Ключевые слова, сепараторы и т.д. Таблица №9* | | |

### **3.5 Основные структуры данных**

Основные структуры данных лексического анализатора (таблицу лексем и таблицу идентификаторов), описать назначение всех полей структур, привести их реализацию на языке С++.

Таблица лексем:

struct Entry { // строка таблицы лексем

char lexema; // лексема

int lineSource; // номер строки в исходном коде

int indexIdTable; // индекс в таблице идентификаторов или LT\_TI\_NULLIDX

char\* sign; // поле для хранения оператора (арифметического или логического)

int numberOfParams; // количество параметров функции

Entry();

Entry(char l, int n);

Entry(char l, int ls, int i, char\* s);

};

struct LexTable { // экзэмпляр таблицы лексем

int maxsize; // ёмкость таблицы лексем < LT\_MAXSIZE

int size; // текущий размер таблицы лексем < maxsize

std::vector<int> refsToAssigns; // индексы таблицы лексем, которые содержат =

Entry\* table; // массив строк таблицы лексем

};

Таблица идентификаторов:

enum IDDATATYPE {INT=1, STR=2, SYBM=3, NDT=4}; // типы данных идентификаторов: integer, string, symbol

enum IDTYPE {V = 1, F = 2, P = 3, L = 4, NT = 5}; // типы идентификаторов: переменная, функция, параметр, литерал

struct Entry { // строка таблицы идентификаторов

int idxfirstLE; // индекс первой строки в таблице лексем

char\* id; // идентификатор (автоматически усекается до ID\_MAXSIZE)

IDDATATYPE iddatatype; // тип данных

IDTYPE idtype; // тип идентификатора

Union {

int vint; // значение integer

struct {

int len; // количесвто символов в string

char\* str; // символы string

} vstr[TI\_STR\_MAXSIZE]; // значение string

} value; // значение идентификатора

};

struct IdTable { // экземпляр таблицы идентификаторов

int maxsize; // ёмкость таблицы идентификаторов < TI\_MAXSIZE

int size; // текущий размер таблицы идентификаторов < maxsize

Entry\* table; // массив строк таблицы идентификаторов

};

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

Перечень сообщений лексического анализатора указан в [*1.24 Классификация сообщений транслятора*](#Классификация_сообщений).

### **3.7 Принцип обработки ошибок**

Описать действие лексического анализатора при обнаружении ошибки в исходном коде программы. Задать действующий лимит на количество ошибок.

try

{

if (lexStr[0] != '\0') // условие при котором сработает ошибка

throw ERROR\_THROW\_IN(120, i, j); // вызов исключения

}

catch (Error::ERROR e)

{

Log::WriteError(log, e);

};

В блоке try выполняется какая-либо инструкция.

Блок catch срабатывает в случае ошибки.

Для вызова исключения используется ключевое слово throw.

В качестве параметров мы передаём i – номер строки, j – номер колонки, где произошла ошибка.

### **3.8 Параметры лексического анализатора**

В качестве фактического параметра в лексический анализатор мы передаём структуру данных в которой хранится обработанный исходный код (без избыточных символов).

### **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 вычисленной части входного потока выполняется функция распознавания лексемы;
* При успешном распознавании информации о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
* Формирует протокол работы;
* При неуспешном распознавании выдаётся сообщение об ошибки, а дальнейшие действия зависят от реализации сканера – либо его выполнение прекращается, либо делается попытка распознать следующую лексему.

### 

### **3.10 Контрольный пример**

Таблица лексем:

|  |  |
| --- | --- |
| **Номер строки** | **Строка** |
| 01 | tfi(ti,ti,ti) |
| 02 | { |
| 03 | pi; |
| 04 | $ti; |
| 05 | i=ivi; |
| 06 | ri; |
| … | |
| 29 | $ti; |
| 30 | i=((ivi)vivivivi)vi; |
| 31 | pi; |
| 32 | pi; |
| 33 | rn; |
| 34 | }; |
| *Таблица лексем. Таблица №10* | |

Таблица идентификаторов:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя идентификатора** | **Тип данных** | **Тип лексемы** | **Значение** | **Ссылка на первое вхождение в таблице лексем** |
| fu | posint | function | 0 | 2 |
| x | posint | parameter | 0 | 5 |
| y | posint | parameter | 0 | 8 |
| z | string | parameter | “ ” | 11 |
| result | posint | variable | 0 | 19 |
| … | | | | |
| k | posint | variable | 0 | 47 |
| a | posint | variable | 0 | 51 |
| b | posint | variable | 0 | 55 |
| … | | | | |
| *Таблица идентификаторов. Таблица №11* | | | | |

## **4. Разработка синтаксического анализатора**

Вторая фаза работы компилятора называется синтаксическим анализом, назначением которой является распознавание синтаксических конструкций языка и формирование промежуточного кода. Правила языка программирования описываются с помощью контекстно-свободных грамматик (тип 2 иерархии Хомского). Программа, выполняющая синтаксический анализ, называется синтаксическим анализатором. Исходными данными синтаксического анализатора являются таблицы лексем и идентификаторов. Лексемы являются для синтаксического анализатора терминальными символами контекстно-свободной грамматики. Результат работы синтаксического анализатора – дерево разбора (промежуточное представление кода). Распознавателями для контекстно-свободных языков являются односторонние недетерминированные распознаватели с ограниченной магазинной памятью (МП-автоматы).

### **4.1 Структура синтаксического анализатора**

Синтаксический анализатор: часть компилятора, выполняющая синтаксический анализ. Входом для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходом – дерево разбора.

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

Грамматики типа иерархии Хомского:

– контекстно-свободные грамматики.

Правила имеют вид:

Грамматика:

|  |  |
| --- | --- |
| **Терминалы** | **Назначение** |
| g | go – точка входа |
| f | func - функция |
| r | ret – возврат значения |
| p | puts – оператор вывода |
| q | if – оператор ветвления |
| u | until – оператор цикла |
| l | loop – конец цикла |
| e | else |
| d | end |
| t | тип |
| i | идентификатор |
| c | символьный литерал |
| s | строковый литерал |
| n | беззнаковый целый литерал |
| $ | объявить |
| ; | ; |
| , | , |
| { | { |
| } | } |
| ( | ( |
| ) | ) |
| v | +  -  \*  / |
| w | ==  >=  <=  >  < |
| *Таблица терминалов Таблица №12* | |

|  |  |
| --- | --- |
| **Нетерминалы** | **Назначение** |
| S | структура программы |
| N | оператор |
| E | выражение |
| M | вложенный оператор |
| F | параметры функции |
| W | параметры вызываемой функции |
| I | параметры цикла или оператора ветвления |
| *Таблица нетерминалов Таблица №13* | |

Правила грамматики:

### **4.3 Построение конечного магазинного автомата**

Формальное описание конечного автомата с магазинной памятью, алгоритм работы МП-автомата, схема и последовательность мгновенных состояний МП-автомата, демонстрирующая успешный разбор цепочки языка из контрольного примера.

Формальное описание МП-автомата:

|  |
| --- |
|  |
| Q – множество состояний; |
| V – алфавит входных символов; |
| Z – специальный алфавит магазинных символов; |
| – функция переходов автомата |
| - множество подмножеств ; |
| Q – начальное состояние автомата; |
| – начальное состояние магазина (маркер дна); |
| – множество конечных состояний. |
| Работа МП-автомата : |
| 1) текущее состояние автомата – (q, aα, zβ) |
| 2) возможно два случая: |
| 1. Читает символ a, находящийся под головкой (сдвигает ленту); |
| 1. Не читает ничего (читает λ, не сдвигает ленту); |
| 3) по функции переходов определяет новое состояние , если |
| *.* |
| 4) читает верхний символ z (в магазине) и записывает цепочку λ т.к. |
| *при этом, если =* λ, то верхний символ магазина |
| просто удаляется. |
| 5) работа автомата заканчивается (q, λ, λ) |

### **4.4 Основные структуры данных**

Программный код основных структур данных на языке С++, описывающих контекстно-свободную грамматику.

struct Rule { // правило в грамматике Грейбах

GRBALPHABET nn; // нетерминал (левый символ правила) < 0

int iderror; // идентификатор диагностической ошибки

short size; // количество цепочек - правых частей правила

struct Chain { // цепочка (правая часть правила)

short size; // длина цепочки

GRBALPHABET\* nt; // цепочка терминалов (>0) и нетерминалов (<0)

Chain() { size = 0; nt = 0; };

Chain(

short psize, // количество символов в цепочке

GRBALPHABET s, ... // символы (терминал или нетерминал)

);

char\* getCChain(char\* b); // получить правую сторону правила

static GRBALPHABET T(char t) { return GRBALPHABET(t); }; // терминал

static GRBALPHABET N(char n) { return -GRBALPHABET(n); }; // не терминал

static bool isT(GRBALPHABET s) { return s > 0; }; // терминал?

static bool isN(GRBALPHABET s) { return !isT(s); }; // нетерминал?

static char alphabet\_to\_char(GRBALPHABET s) { return isT(s) ? char(s) : char(-s); }; // GRBALPHABET->char

}\* chains; // массив цепочек - правых частей правила

Rule() { nn = 0x00; size = 0; };

Rule(

GRBALPHABET pnn, // нетерминал (< 0)

int iderror, // идентификатор диагностического сообщения (Error)

short psize, // количество цепочек - правых частей правила

Chain c, ... // множество цепочек - правых частей правила

);

char\* getCRule( // получить правило в виде N-цепочки (для распечатки)

char\* b, // буфер

short nchain // номер цепочки (правой части) в правиле

);

short getNextChain( // получить следующую за j подходящую цепочку, вернуть её номер или -1

GRBALPHABET t, // первый символ цепочки

Rule::Chain& pchain, // возвращаемая цепочка

short j // номер цепочки

);

};

struct Greibach { // грамматика Грейбах

short size; // количество правил

GRBALPHABET startN; // стартовый символ

GRBALPHABET stbottomT; // дно стека

Rule\* rules; // множество правил

Greibach() {

short size = 0;

startN = 0;

stbottomT = 0;

rules = 0;

};

Greibach(

GRBALPHABET pstartN, // стартовый символ

GRBALPHABET pstbottomT, // дно стека

short psize, // количество правил

Rule r, ... // правила

);

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

GRBALPHABET pnn, // левый символ

Rule& prule // возвращаемое правило грамматики

);

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

};

Greibach getGreibach(); //Получить грамматику

* 1. **Контрольный пример**

Результат синтаксического разбора конструкций языка.

|  |
| --- |
| 0 : всего строк 183, синтаксический анализ выполнен без ошибок  0 : S->tfi(F){NrE;};S  4 : F->ti,F  7 : F->ti  11 : N->$ti;N  15 : N->i=E;N  17 : E->n  19 : N->ps;N  22 : N->u(I)Nl;  24 : I->iwi  28 : N->pi;N  31 : N->i=E;  33 : E->iM  34 : M->vE  35 : E->n  40 : E->i  44 : S->tfi(F){NrE;};S  48 : F->ti,F  51 : F->ti  55 : N->$ti;N  59 : N->i=E;  61 : E->iM  62 : M->vE  63 : E->i  66 : E->i  70 : S->tfi(F){NrE;};S  74 : F->ti,F  77 : F->ti,F  80 : F->ti  84 : N->$ti;N  88 : N->i=E;N  90 : E->n  92 : N->i=E;  94 : E->(E)M  95 : E->(E)M  96 : E->iM  97 : M->vE  98 : E->iM  99 : M->vE  100 : E->i  102 : M->vE  103 : E->i  105 : M->vE  106 : E->i(W)  108 : W->i,W  110 : W->i  114 : E->i  118 : S->g{NrE;};  120 : N->$ti;N  124 : N->$ti;N  128 : N->$ti;N  132 : N->$ti;N  136 : N->i=E;N  138 : E->s  140 : N->i=E;N  142 : E->n  144 : N->i=E;N  146 : E->n  148 : N->i=E;N  150 : E->n  152 : N->i=E;N  154 : E->i(W)  156 : W->i,W  158 : W->i,W  160 : W->n  163 : N->i=E;N  165 : E->i(W)  167 : W->i,W  169 : W->n  172 : N->ps;N  175 : N->pi;  179 : E->n |

## **5. Разработка семантического анализатора**

Семантический анализ – третья фаза работы транслятора. Семантический анализ может быть явно выделен в отдельную фазу или совмещаться с фазами лексического и синтаксического анализа. Семантический анализатор использует синтаксическое дерево и информацию из таблицы идентификаторов для проверки исходного текста на соответствие семантическим правилам языка. Основные действия семантического анализатора:

* проверка семантических правил исходного языка;
* дополнение внутреннего представления программы операторами и действиями, неявно предусмотренными семантикой исходного языка.

**5.1 Структура семантического анализатора**

Описание назначения семантического анализа, его входные и выходные данные. Схема семантического анализатора, поясняющая его взаимодействие с другими компонентами транслятора.

Назначение семантического анализа – проверка смысловой правильности

конструкций языка программирования.

Входные данные для семантического анализатора:

* таблица идентификаторов;
* дерево разбора – результат разбора синтаксических конструкций входного языка.

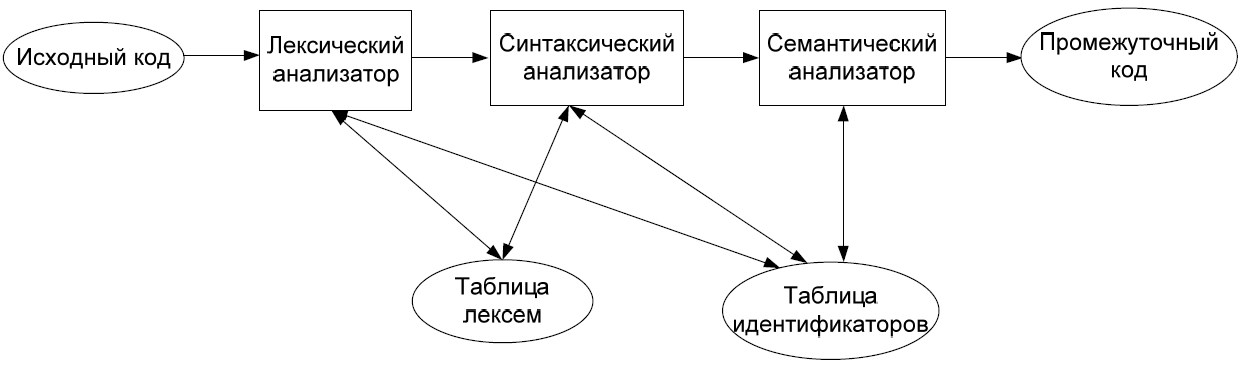
****

Рисунок 4 Схема взаимодействия семантического анализатора с другими компонентами транслятора.

### **5.2 Контрольный пример**

Примеры ошибок, выявляемых семантическим анализатором.

|  |  |
| --- | --- |
| **Исходный код с ошибкой** | **Протокол** |
| $ string str;  $ posint result;  **$ posint x;**  **$ symbol x;**  $ posint y;  str = #cycle#; | ---- Протокол ---- Дата: 16.12.2020 19:29:58 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 673  Проигнорировано: 141  Количество строк: 45  Ошибка 200: переопределение идентификатора, строка 29, колонка 10 |
| $ string str;  $ posint result;  $ posint x;  $ posint y;  str = #cycle#;  **t = 1;** | ---- Протокол ---- Дата: 16.12.2020 19:33:14 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 668  Проигнорировано: 143  Количество строк: 45  Ошибка 201: идентификатор не определён, но используется, строка 31, колонка 1 |
| $ posint result;  $ symbol t;  **t = #t#;**  result = 0;  result = ((x + y + z) / z) \* makeCoef(x, y) **+** t; | ---- Протокол ---- Дата: 16.12.2020 19:37:19 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 689  Проигнорировано: 148  Количество строк: 46  Ошибка 202: все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции, строка 23 |
| posint func testF (posint x, **posint y**, posint z)  {  ret 0;  };  go  {  $ string str;  $ posint result;  $ posint x;  $ posint y;  str = #cycle#;  x = 3;  y = 4;  y = testF(1, **#test#,** 2); | ---- Протокол ---- Дата: 16.12.2020 19:45:02 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 752  Проигнорировано: 152  Количество строк: 50  Ошибка 203: количество или типы фактических параметров не согласованы с количеством или типами формальных параметров, строка 37 |
| **posint** func makeCoef (posint x, posint y)  {  $ posint result;  result = x + y;  ret **#sdf#;**  }; | ---- Протокол ---- Дата: 16.12.2020 19:49:14 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 658  Проигнорировано: 138  Количество строк: 44  Ошибка 204: тип возвращаемого значения функции, строка 15, колонка 10 |

## **6. Вычисление выражений**

В разделе описываются выражения, допускаемые языком, форма, принципы построения и вычисление выражений.

Обычная форма выражений, в которой знак операции размещается между операндами, называется инфиксной. Обратная польская нотация (называют также польской инверсной записью, ПОЛИЗ) — это форма записи математических выражений, в которой операторы расположены после своих операндов. Выражения, представленные в обратной польской нотации, легко вычисляются, время вычисления — линейное. Выражение в обратной польской нотации читается слева направо: операция выполняется над двумя операндами, непосредственно стоящими перед знаком этой операции. Результат операции заменяет в выражении последовательность её операндов и символ операции. Результатом вычисления всего выражения является результат последней вычисленной операции. ПОЛИЗ удобна как для вычисления выражений, так и в качестве промежуточной формы представления выражений в трансляторе. Принцип обратной польской записи может быть применен не только к выражениям, но и операторам языков программирования.

### **6.1 Выражения, допускаемые языком**

Описание выражений, допускаемые языком (типы данных, используемые в выражениях, приоритетность операций, использование функций в выражениях), приведены типичные примеры выражений из контрольного примера.

|  |  |
| --- | --- |
| <выражение> | (<литерал> | <идентификатор>) {{(<идентификатор> | <литерал>)} (+ | - | / | \*) {(<идентификатор> | <литерал>)}} |

Приоритетность операторов можно найти в главе [*1.12 Операции языка*](#Операции_языка).

В качестве идентификатора может выступать функция.

В выражениях могут использовать только беззнаковый целый тип данных.

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

Назначение обратной польской записи, принципы ее построения, примеры преобразования выражений из контрольного примера в обратную польскую нотацию.

Обратная польская удобна:

* для вычисления выражений;
* как промежуточная форма представления выражений в трансляторе;
* как промежуточная форма представления операторов языков программирования.

Примеры преобразований выражений:

### **6.3 Программная реализация обработки выражений**

Фрагмент кода транслятора на языке С++, реализующего преобразование выражений в обратный польский формат.

int getPriority(std::string l) { // приоритет операции

std::map <std::string, int> mapping;

mapping["\*"] = 3;

mapping["/"] = 3;

mapping["-"] = 2;

mapping["+"] = 2;

mapping["("] = 1;

switch (mapping[l]) {

case 3: return 3; break;

case 2: return 2; break;

case 1: return 1; break;

}

return 0;

}

// создание новой таблице лексем с преобразованными выражениями

LT::LexTable PolishNotation(LT::LexTable& lextable, IT::IdTable& idtable) {

int ref = 0;

int semicolon;

std::list <LT::Entry> list\_of\_LT\_Entries;

LT::LexTable result = LT::Create(LT\_MAXSIZE);

for (int i = 0; i < lextable.size; i++) {

if (ref < lextable.refsToAssigns.size() && i == lextable.refsToAssigns[ref]) {

LT::Add(result, lextable.table[i]);

list\_of\_LT\_Entries = ConvertToPN(i + 1, lextable, idtable, semicolon);

LT::AddList(result, list\_of\_LT\_Entries);

i = semicolon;

ref++;

}

LT::Add(result, lextable.table[i]);

}

return result;

}

// трансформация выражения в форму обратной польской записи

std::list <LT::Entry> ConvertToPN(int lextable\_pos, LT::LexTable& lextable, IT::IdTable& idtable, int& semicolon) {

std::list <LT::Entry> list\_of\_LT\_Entries;

std::stack <LT::Entry> stack;

LT::Entry func;

int leftHesisCount = 0;

int paramCount = 0;

bool isFunction = false;

int i;

for (i = lextable\_pos; lextable.table[i].lexema != LEX\_SEMICOLON; i++) {

if (lextable.table[i].lexema == LEX\_ID || lextable.table[i].lexema == LEX\_NUMERICAL\_LITERAL || lextable.table[i].lexema == LEX\_STRING\_LITERAL) {

if (idtable.table[lextable.table[i].indexIdTable].idtype == IT::F) {

isFunction = true;

paramCount = 0;

lextable.table[i].lexema = LEX\_FUNC\_CALL;

func = lextable.table[i];

}

else

{

if (isFunction)

paramCount++;

list\_of\_LT\_Entries.push\_back(lextable.table[i]);

}

} else if (lextable.table[i].lexema == LEX\_LEFTHESIS) {

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

} else if (lextable.table[i].lexema == LEX\_RIGHTHESIS) {

while (!stack.empty() && stack.top().lexema != LEX\_LEFTHESIS) {

list\_of\_LT\_Entries.push\_back(stack.top());

stack.pop();

}

if (stack.top().lexema == LEX\_LEFTHESIS) {

stack.pop();

}

if (isFunction) {

func.numberOfParams = paramCount;

list\_of\_LT\_Entries.push\_back(func);

isFunction = false;

paramCount = 0;

}

} else if (lextable.table[i].lexema == LEX\_COMMA) {

continue;

} else {

while (!stack.empty() && getPriority(lextable.table[i].sign) <= getPriority(stack.top().sign)) {

list\_of\_LT\_Entries.push\_back(stack.top());

stack.pop();

}

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

}

}

while (stack.size() != 0) {

list\_of\_LT\_Entries.push\_back(stack.top());

stack.pop();

}

semicolon = i;

return list\_of\_LT\_Entries;

}

### **6.4 Контрольный пример**

Приведена часть протокола контрольного примера, отображающую результаты преобразования выражений в обратную польскую запись (формат: выражение – обратная польская запись для выражения).

|  |  |
| --- | --- |
| **Выражение в инфиксной форме** | **Выражение в обратной польской записи** |
| posint func makeCompute (posint x, posint y, posint z)  {  $ posint result;  result = 0;  result = **((x + y + z) / z) \* makeCoef(x, y)**;  ret result;  }; | tfi(ti,ti,ti)  {  $ti;  i=n;  i=**iivivivii@v**;  ri;  }; |

## **7. Генерация кода**

В разделе описывается процесс генерации кода. Генерация кода – четвертая последняя фаза работы транслятора. Исходными данными для генератора кода является промежуточное представление исходной программы. Одной из основных задач на этапе генерации кода является планирование памяти для переменных, литералов. Решение о распределении памяти принимается либо в процессе генерации промежуточного кода, либо при генерации целевого кода.

Самый простой подход к выполнению курсового проекта – это генерация кода в JavaScript. Интерпретатором JavaScript является js-движок браузера. Входными данными для генерации являются таблицы лексем и идентификаторов и дерево разбора.

Другой подход: можно выполнить генерацию в код на языке ассемблера. Затем вызвать транслятор ассемблера для получения объектного кода.

Еще один подход – разработка транслятора-интерпретатора (исходный код транслируется в байт-код – последовательность команд для некоторой виртуальной машины, затем он сразу интерпретируется).

### **7.1 Структура генератора кода**

Целевой язык трансляции – ассемблер.

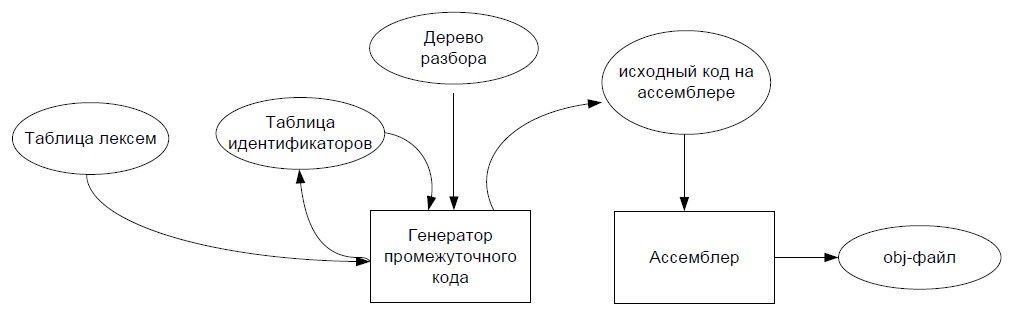


Рисунок 5 Генерация исходного ассемблерного кода

### **7.2 Статическая библиотека**

У языка DVR-2020 статическа библиотека не реализована.

### **7.3 Контрольный пример**

Результат генерации кода на основе контрольного примера.

|  |  |
| --- | --- |
| **Исходный код** | **Сгенерированный код** |
| posint func print(string str,posint count)  {  $ posint i;  i=0;  puts #InPrint#;  until(i<count)  puts str;  i=i+1;  loop;  ret i;  }; | print PROC printstr : DWORD, printcount : DWORD  push num0  pop printi  mov ESI, offset str0  invoke WriteConsoleA, stdout, ESI, 9, ADDR cWritten, 0  CYCLE:  mov EAX, printi  cmp EAX, printcount  jb cycle\_body  cycle\_end:  jmp CYCLEend  cycle\_body:  mov ESI, printstr  invoke WriteConsoleA, stdout, ESI, 9, ADDR cWritten, 0  push printi  push num1  pop EAX  pop EBX  add EAX, EBX  push EAX  pop printi  jmp CYCLE  CYCLEend:  push printi  pop EAX  ret  print ENDP |

## **8. Тестирование транслятора**

Контрольный пример, демонстрирующий правильную работу компилятора. Выполнение подбора тестов, которые должны быть включены в тестовый комплект.

### **8.1 Результаты тестирования**

В разделе сгруппировано описание тестовых наборов, демонстрирующих проверки на разных этапах трансляции. Представлены результаты в виде таблицы, содержащей фрагмент исходного кода с ошибкой и соответствующее диагностическое сообщение (код ошибки, этап, текст сообщения, место ошибки с указанием строки и позиции в исходном коде).

|  |  |
| --- | --- |
| **Исходный код с ошибкой** | **Протокол** |
| *Ошибки семантического анализа* | |
| $ string str;  $ posint result;  **$ posint x;**  **$ symbol x;**  $ posint y;  str = #cycle#; | ---- Протокол ---- Дата: 16.12.2020 19:29:58 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 673  Проигнорировано: 141  Количество строк: 45  Ошибка 200: переопределение идентификатора, строка 29, колонка 10 |
| $ string str;  $ posint result;  $ posint x;  $ posint y;  str = #cycle#;  **t = 1;** | ---- Протокол ---- Дата: 16.12.2020 19:33:14 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 668  Проигнорировано: 143  Количество строк: 45  Ошибка 201: идентификатор не определён, но используется, строка 31, колонка 1 |
| $ posint result;  $ symbol t;  **t = #t#;**  result = 0;  result = ((x + y + z) / z) \* makeCoef(x, y) **+** t; | ---- Протокол ---- Дата: 16.12.2020 19:37:19 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 689  Проигнорировано: 148  Количество строк: 46  Ошибка 202: все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции, строка 23 |
| posint func testF (posint x, **posint y**, posint z)  {  ret 0;  };  go  {  $ string str;  $ posint result;  $ posint x;  $ posint y;  str = #cycle#;  x = 3;  y = 4;  y = testF(1, **#test#,** 2); | ---- Протокол ---- Дата: 16.12.2020 19:45:02 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 752  Проигнорировано: 152  Количество строк: 50  Ошибка 203: количество или типы фактических параметров не согласованы с количеством или типами формальных параметров, строка 37 |
| **posint** func makeCoef (posint x, posint y)  {  $ posint result;  result = x + y;  ret **#sdf#;**  }; | ---- Протокол ---- Дата: 16.12.2020 19:49:14 ----  ---- Параметры ----  -log: In.txt.log  -out: In.txt.out  -in: In.txt  -asm: generation.asm  ---- Исходные данные ----  Количество символов: 658  Проигнорировано: 138  Количество строк: 44  Ошибка 204: тип возвращаемого значения функции, строка 15, колонка 10 |
| *Ошибки синтаксического анализатора* | |
| **posint test** (posint x, posint y, posint z)  {  ret 0;  }; | 600: строка 24, Неверная структура программы  600: строка 24, Неверная структура программы  601: строка 23, Ошибочный оператор  (отсутствует ключевое слово func) |
| puts #InPrint#;  until (i < count)  puts str;  i = i + 1;  ret i; | 601: строка 9, Ошибочный оператор  601: строка 9, Ошибочный оператор  601: строка 9, Ошибочный оператор  (у цикла отсутствует ключевое слово loop) |
| result = ((x + y + z) / z) **\*\*** makeCoef(x, y); | 602: строка 21, Ошибка в выражении  602: строка 21, Ошибка в выражении  602: строка 21, Ошибка в выражении |
| posint func makeCompute (posint x, posint y, posint z, **posint**) | 604: строка 17, Ошибка в параметрах функции  604: строка 17, Ошибка в параметрах функции  604: строка 17, Ошибка в параметрах функции |
| result = makeCompute(x, y, 2, , ); | 605: строка 34, Ошибка в параметрах вызываемой функции  605: строка 34, Ошибка в параметрах вызываемой функции  605: строка 34, Ошибка в параметрах вызываемой функции |
| until (i count) | 606: строка 5, Ошибка в параметрах цикла или оператора ветвления  606: строка 5, Ошибка в параметрах цикла или оператора ветвления  606: строка 5, Ошибка в параметрах цикла или оператора ветвления |

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения курсовой работы был разработан язык программирования DVR-2020. Реализованы его основные компоненты: лексический анализатор, синтаксический анализатор, семантический анализатор, генератор кода. Ниже приведены количественные и качественные характеристики языка.

Язык программирования DVR-2020 включает:

* три типа данных: string (строковый), symbol (символьный), posint (беззнаковый целый);
* инструкции: puts (оператор вывода), until (цикл), if и else (операторы ветвления);
* математические операторы: +, -, \*, /;
* операторы отношения: ==, <=, >=, <, >;
* 33 лексемы;
* 56 правил грамматики;
* у языка DVR-2020 стандартная библиотека не реализована;
* примерное количество строк кода: 3125;
* время трансляции контрольного примера:146мс.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Обратная польская запись / Интернет-портал habr.com [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/100869/>
2. Руководство по ассемблеру x86 для начинающих / Автор оригинала: Derek Maciel / Интернет-портал habr.com [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/423077/>
3. Уроки по Ассемблеру / Интернет-портал ravesli.com [Электронный ресурс]. – Режим доступа: <https://ravesli.com/uroki-assemblera/>
4. Рубрика: Assembler / Интернет-портал codetown.ru [Электронный ресурс]. – Режим доступа: <https://codetown.ru/category/assembler/>
5. Наркевич А. С. / Лекции по предмету “Языки программирования”