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 ААА-2018»

Выполнил студент Астахова Анастасия Алексеевна

(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 Н.В.

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

Консультанты ст.пр. Наркевич Аделина Сергеевна

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

(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 2017

Содержание

[Введение 5](#_Toc532773882)

[Глава 1. Спецификация языка программирования AAA-2018 6](#_Toc532773883)

[1.1 Характеристика языка программирования 6](#_Toc532773884)

[1.2 Алфавит языка 6](#_Toc532773885)

[1.3 Применяемые сепараторы 6](#_Toc532773886)

[1.4 Применяемые кодировки 7](#_Toc532773887)

[1.5 Идентификаторы 7](#_Toc532773888)

[1.6 Литералы 7](#_Toc532773889)

[1.7 Типы данных 8](#_Toc532773890)

[1.8 Преобразование типов данных 8](#_Toc532773891)

[1.9 Объявление данных и область видимости 8](#_Toc532773892)

[1.10 Инициализация данных 9](#_Toc532773893)

[1.11 Инструкции языка 9](#_Toc532773894)

[1.12 Операции языка 9](#_Toc532773895)

[1.13 Выражения и их вычисления 10](#_Toc532773896)

[1.14 Программные конструкции языка 10](#_Toc532773897)

[1.15 Область видимости идентификаторов 10](#_Toc532773898)

[1.16 Точка входа 10](#_Toc532773899)

[1.17 Препроцессор 10](#_Toc532773900)

[1.18 Ввод и вывод данных 10](#_Toc532773901)

[1.19 Семантические проверки 11](#_Toc532773902)

[1.20 Распределение оперативной памяти на этапе выполнения 11](#_Toc532773903)

[1.21 Стандартная библиотека и её состав 11](#_Toc532773904)

[1.22 Соглашения о вызовах 11](#_Toc532773905)

[1.23 Объектный код 11](#_Toc532773906)

[1.24 Классификация сообщений транслятора 12](#_Toc532773907)

[1.25 Контрольный пример 12](#_Toc532773908)

[Глава 2. Структура транслятора 14](#_Toc532773909)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 14](#_Toc532773910)

[2.2 Перечень входных параметров транслятора 14](#_Toc532773911)

[2.3 Перечень файлов, формируемых транслятором и их содержимое 15](#_Toc532773912)

[Глава 3. Разработка лексического анализатора 16](#_Toc532773913)

[3.1 Структура лексического анализатора 16](#_Toc532773914)

[3.2 Контроль входных символов 16](#_Toc532773915)

[3.3 Удаление избыточных символов 16](#_Toc532773916)

[3.4 Перечень ключевых слов, сепараторов, символов операций соответствующим им лексемам 16](#_Toc532773917)

[3.5 Основные структуры данных 16](#_Toc532773918)

[3.6 Структура и перечень сообщений лексического анализатора 17](#_Toc532773919)

[3.7 Принцип обработки ошибок 17](#_Toc532773920)

[3.8 Параметры лексического анализатора и режимы его работы 17](#_Toc532773921)

[3.9 Алгоритм лексического анализа 18](#_Toc532773922)

[3.10 Контрольный пример 18](#_Toc532773923)

[Глава 4. Разработка синтаксического анализатора 19](#_Toc532773924)

[4.1 Структура Синтаксического анализатора 19](#_Toc532773925)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 19](#_Toc532773926)

[4.3 Построение конечного магазинного автомата 20](#_Toc532773927)

[4.4 Основные структуры данных 20](#_Toc532773928)

[4.5 Описание алгоритма синтаксического разбора 21](#_Toc532773929)

[4.6 Структура и перечень сообщений синтаксического анализатора 21](#_Toc532773930)

[4.7 Параметры синтаксического анализатора и режимы его работы 21](#_Toc532773931)

[4.8 Принцип обработки ошибок 21](#_Toc532773932)

[4.9 Контрольный пример 21](#_Toc532773933)

[Глава 5. Разработка семантического анализатор 22](#_Toc532773934)

[5.1 Структура семантического анализатора 22](#_Toc532773935)

[5.2 Функции семантического анализа 22](#_Toc532773936)

[5.3 Структура и перечень сообщений семантического анализатора 22](#_Toc532773937)

[5.4 Принцип обработки ошибок 23](#_Toc532773938)

[5.5 Контрольный пример 23](#_Toc532773939)

[Глава 6. Преобразование выражений 24](#_Toc532773940)

[6.1 Выражения, допускаемые языком 24](#_Toc532773941)

[6.2 Польская запись и принцип ее построения 24](#_Toc532773942)

[6.3 Программная реализация 24](#_Toc532773943)

[6.4 Контрольный пример 25](#_Toc532773944)

[Глава 7. Генерация кода 26](#_Toc532773945)

[7.1 Структура генератора кода 26](#_Toc532773946)

[7.2 Представление типов данных в памяти 26](#_Toc532773947)

[7.3 Статическая библиотека 26](#_Toc532773948)

[7.4 Особенности алгоритма генерации кода 27](#_Toc532773949)

[7.5 Контрольный пример 27](#_Toc532773950)

[Глава 8. Тестирование транслятора 28](#_Toc532773951)

[8.1 Тестирование проверки на допустимость символов 28](#_Toc532773952)

[8.2 Тестирование лексического анализатора 28](#_Toc532773953)

[8.3 Тестирование синтаксического анализатора 28](#_Toc532773954)

[8.4 Тестирование семантического анализатора 29](#_Toc532773955)

[Заключение 30](#_Toc532773956)

[Приложение А 31](#_Toc532773957)

[Приложение Б 35](#_Toc532773958)

[Приложение В 37](#_Toc532773959)

[Приложение Г 52](#_Toc532773960)

[Литература 56](#_Toc532773961)

# Введение

Основной целью данного курсового проекта является разработка компилятора для языка программирования ААА-2018. Главная задача компилятора заключается в том, чтобы сделать программу, написанную языке программирования ААА-2018, понятной компьютеру. Этого можно добиться одним из двух способов: компиляцией или интерпретацией. В данном курсовом проекте трансляция будет осуществляться в язык CIL (Common Intermediate Language).

Компиляция состоит из двух частей: анализа и синтеза. Анализ – это разбиение исходной программы на составные части и создание ее промежуточного представления, а синтез, в свою очередь, – конструирование требуемой целевой программы из промежуточного представления

Исходя из цели курсового проекта, были определены следующие задачи:

* разбработка спецификации языка программирования;
* разбратка структуры транслятора;
* разработка лексического анализатора;
* разработка синтаксического анализатора;
* разработка семантического анализатора;
* обработка выражений;
* генерация кода;
* тестирование компилятора.

Язык прогарммирования ААА-2018 предназначен для выполнения простейших арифметический действий и операций над строками.

# Глава 1. Спецификация языка программирования AAA-2018

## 1.1 Характеристика языка программирования

Язык программирования ААА– это процедурный, универсальный, строго типизированный (преобразование типов не допускается), компилируемый язык. Не является объектно-ориентированным.

## 1.2 Алфавит языка

Алфавит языка ААА-2018 основан на кодировке Windows-1251. Исходный код ААА-2018 может содержать символы латинского алфавита, цифры десятичной системы счисления от 0 до 9, русские символы, которые разрешены только в строковых литералах, символы пробела, табуляции и перевода строки.

## 1.3 Применяемые сепараторы

Символы-разделители, разрешенные к использованию в языке AAA-2018, приведены в таблице 1.1.

Таблица 1.1 - Применяемые сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| ;  :  ‘ ‘ (space)  ,  =  +  -  \*  <  >  ! | разделение инструкций |
| {  } | программный блок |
| (  ) | параметры/приоритетность операций (в выражениях) |
| {  } | разделение инструкций |
| (  ) | параметры/приоритетность операций (в выражениях) |
| [  ] | Блок условий в конструкциях ветвления и цикла |

## 1.4 Применяемые кодировки

Для написания исходного кода на языке программирования ААА-2018 используется кодировка Windows-1251.

Таблица 1.2 – Коды символов кодировки Windows-1251



## 1.5 Идентификаторы

В имени идентификатора допускаются символы латинского алфавита нижнего регистра. Максимальная длина имени - 15 символов.

Идентификаторы не должны совпадать с ключевыми словами. Данные правила справедливы как для переменных, так и для функций.

## 1.6 Литералы

Литералы осуществляют инициализацию переменных. Литералы, используемые для написания программы на языке AAA-2018 описаны в таблице 1.3.

Таблица 1.3 – Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Литералы целого типа | Целочисленные неотрицательные литералы, инициализируются 0. Литералы только rvalue. |
| Строковые литералы | Символы, заключённые в ‘ ’ (одинарные кавычки), инициализируются пустой строкой, строковые переменные. Только rvalue. |

## 1.7 Типы данных

В языке AAA-2018 есть 2 типа данных: целочисленный и строковый. Описание типов данных, предусмотренных в данным языке представлено в таблице 1.4.

Таблица 1.4 – Типы данных

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| integer | Фундаментальный тип данных. Предусмотрен для объявления целочисленных данных (4 байта).  Автоматически инициализируется нулевым значением.  Возможные арифметические операции:  + – бинарный, суммирование;  - – бинарный, вычитание;  \* – бинарный, умножение;  = – присваивание значения; |
| string | Фундаментальный тип данных. Предусмотрен для объявления строк. (1 символ – 1 байт). Максимальное количество символов в строке – 255. Автоматическая инициализация строкой нулевой длины. |

## 1.8 Преобразование типов данных

Преобразование типов данных не допускается.

## 1.9 Объявление данных и область видимости

Для объявления переменной следует указать имя переменной и её тип.

Пример объявления переменной числового типа:

var a:integer;

Пример объявления переменной строкового типа:

var a:string;

Для объявления функций используется ключевое слово function, после которого следует указать имя функции. Тип данных, возвращаемых функцией, указывается после секции параметров и знака двоеточие.

Пример объявления библиотечной функции:

use function fact (a:integer):integer;

Все переменные имеют область видимости, в которой они находятся, что разрешает использование в различных функциях переменных с одинаковым именем. Параметры функции видны только внутри функции. Переменные, объявленные в одной функции, недоступны в другой.

## 1.10 Инициализация данных

Таблица 1.5 – Способы инициализации переменных

|  |  |
| --- | --- |
| Вид инициализации | Примечание |
| var <идентификатор>: <тип данных>; | Автоматическая инициализация: переменные типа integer инициализируются нулём, переменные типа string – пустой строкой. |
| var <идентификатор> = <значение>; | Присваивание переменной значения. |

## 1.11 Инструкции языка

Все возможные инструкции языка программирования AAA-2018 представлены в общем виде в таблице 1.6.

Таблица 1.6 – Инструкции языка программирования AAA-2018

|  |  |
| --- | --- |
| Инструкция | Запись на языке AAA-2018 |
| Объявление переменной | var <идентификатор>: <тип данных>; |
| Присваивание | <идентификатор> = <значение>/<идентификатор>; |
| Объявление внешней функции | use function <идентификатор> (<идентификатор>: <тип данных>,…): <типданных>; |
| Блок инструкций | main  {  …  } |
| Возврат из подпрограммы | return <идентификатор> / <литерал>; |
| Вывод данных | out <идентификатор> / <литерал>; |

## 1.12 Операции языка

Язык программирования ААА-2018 может выполнять арифметические операции, представленные в таблице 1.7.

Таблица 1.7 – Приоритетности операций языка программирования AAA‑2018

|  |  |
| --- | --- |
| Операция | Приоритетность операции |
| (  ) | 0 или 4 |
| , | 1 |
| \*  / | 2 |
| +  - | 3 |

Максимальным значением приоритетности является “0”, минимальным соответственно “4”.

Также языком поддерживается операция сравнения операндов. Данная операция не имеет приоритета.

## 1.13 Выражения и их вычисления

Круглые скобки в выражении используются для изменения приоритета операций. Также не допускается запись двух подряд идущих арифметических операций. Выражение может содержать вызов функции, а параметр вызова функции может быть представлен в виде выражения.

## 1.14 Программные конструкции языка

Ключевые программные конструкции языка программирования AAA-2018 представлены в таблице 1.8.

Таблица 1.8 – Программные конструкции языка AAA-2018

|  |  |
| --- | --- |
| Главная функция (точка входа в приложение) | main  {  } |
| Функция | function <идентификатор> (<идентификатор> : <тип>, …) : <тип>  {  …  return <выражение>;  } |

## 1.15 Область видимости идентификаторов

В языке AAA-2018 переменные обязаны находится внутри программного блока функций (по принципу С++). Объявление глобальных переменных не предусмотрено. Объявление пользовательских областей видимости не предусмотрено.

## 1.16 Точка входа

Точкой входа в языке AAA-2018является функция main(). Если она присутствует выполнение продолжается, иначе фатальная ошибка и прекращение работы.

## 1.17 Препроцессор

В данном языке программирования AAA-2018 препроцессор не предусмотрен.

## 1.18 Ввод и вывод данных

Для вывода данных в языке AAA-2018 используется функция out(идентификатор).

## 1.19 Семантические проверки

Таблица с перечнем семантических проверок, предусмотренных языком, приведена в таблице 1.9.

Таблица 1.9 – Семантические проверки

|  |  |
| --- | --- |
| Номер | Правило |
| 1 | Идентификаторы функций не должны повторяться |
| 2 | Операнды в операторах ветвления и выхода из функции должны быть целочисленного типа |
| 3 | Тип данных передаваемых значений в функцию должен совпадать с типом параметров при её объявлении |
| 4 | Тип данных передаваемых значений в функцию стандартной библиотеки должен соответствовать заявленному. |
| 5 | Идентификатор должен быть объявлен до его использования. |
| 6 | Операнды в арифметическом выражении не могут быть разных типов |

## 1.20 Распределение оперативной памяти на этапе выполнения

Все переменные размещаются в стеке.

## 1.21 Стандартная библиотека и её состав

Функции стандартной библиотеки с описанием представлены в таблице 1.10. Стандартная библиотека написана на языке программирования CIL.

Таблица 1.10 – Состав стандартной библиотеки

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| max | integer | integer x-число  integer y-число | Функция вычисляет максимальное значение |
| min | integer | integer x-число  integer y-число | Функция вычисляет минимальное значение |
| fact | integer | integer x-число | Функция вычисляет факториал x |
| strlen | integer | string x-число | Функция вычисляет длину строки |
| out |  | string x – строка  integer x-число | Функция выводит на консоль x |

## 1.22 Соглашения о вызовах

В языке AAA-2018 используются соглашение о вызовах stdcall, то есть все параметры передаются в стек справа налево, память освобождает вызываемый код.

## 1.23 Объектный код

Исходный код, написанный на языке AAA-2018 , транслируется в язык CIL (Common Intermediate Language)

## 1.24 Классификация сообщений транслятора

Транслятор, в ходе своей работы, генерирует сообщения, которые информируют пользователя о допущенных ошибка. Все сообщения транслятора разделены на интервалы, в зависимости от того на каком этапе была обнаружена ошибка. Все интервалы представлены в таблице 1.11.

Таблица 1.11 - Сообщения транслятора

|  |  |
| --- | --- |
| Интервал | Описание ошибок |
| 000-099 | Системные ошибки |
| 100-109 | Ошибки параметров |
| 110-119 | Ошибки открытия и чтения файлов |
| 200-299 | Ошибки лексического анализа |
| 600-699 | Ошибки синтаксического анализа |
| 700-799 | Ошибки семантического анализа |

## 1.25 Контрольный пример

Контрольный пример демонстрирует главные особенности языка AAA-2018. Исходный код :

function fi(x:integer, y:integer):integer

{

var z:integer;

z = x \* (x + y);

if [z < 100] {

z = z + 1000;

} else {

z = z + 2000;

};

return z;

};

function fs(a:string, b:integer):integer

{

var c:integer;

use function strlen(p:string):integer;

c = strlen(a) + b;

return c;

};

main

{

var x:integer;

var y:integer;

var z:integer;

var sa:string = '1234567890';

var sb:string;

use function strlen(p:string):integer;

use function fact(x:integer):integer;

use function min(x:integer, y:integer):integer;

use function max(x:integer, y:integer):integer;

x = 1;

y = 5;

sa = '1234567890';

sb = '1234567890';

var w:integer;

w = fi(x,y);

y = fs(sa,x);

out 'контрольный пример';

out w;

z = fact(y);

out z;

z = max(x+1, y+1);

out z;

z = min(x, y);

out z;

out strlen(sb);

x=x\*(y+fi(strlen(sa)+y,z))-z/fi(x,y);

out x;

while [x < 10] {

if [x <= 6] {

out x;

} else {

out x \* x;

};

x = x + 1;

};

};

# Глава 2. Структура транслятора

## 2.1 Компоненты транслятора, их назначение и принципы взаимодействия

Основными компонентами транслятора являются лексический анализатор, синтаксический анализатор, семантический анализатор и генератор кода CIL, приведенные на рисунке 2.1.

Лексический анализатор – принимает на вход уже первично обработанный и разбитый на отдельные компоненты исходный код на языке AAA-2018 . Формирует таблицу идентификаторов и таблицу лексем, а также занимается обнаружением ошибок, связанных с лексикой языка.

Синтаксический анализатор – принимает на вход таблицу лексем, сформированную лексическим анализатором. Перебирая каждое правило языка (допустимую конструкцию) он выявляет синтаксические ошибки, допущенные в исходном коде. Формирует дерево разбора, а также выводит трассировку (разбор) цепочек.

Семантический анализатор – состоит из нескольких функций, отвечающих за выявления тех или иных ошибок, а также некоторых проверок, выполняемых на этапе лексического анализатора. На вход подается таблица лексем и таблица идентификаторов.

Генератор кода в CIL (Common Intermediate Language) – принимает на вход таблицу идентификаторов и таблицу лексем а также дерево разбора. Задача этого компонента заключается в трансляции, уже пройденного все предыдущие этапы кода на языке AAA-2018 , в код в CIL (Common Intermediate Language)



Рисунок 2.1 - Структура транслятора AAA-2018

## 2.2 Перечень входных параметров транслятора

С помощью входных параметров транслятора осуществляется контроль за выводом протоколов работы разных компонент, а также указывается файл с исходным кодом программы. Перечень входных параметров представлена в таблице 2.1.

Таблица 2.1 - Входные параметры транслятора

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Тип параметра |
| -in: | Файл с исходным кодом на языке AAA-2018. | Имя файла |
| -log: | Файл с журналом работы транслятора. | Имя файла |
| -out: | Файл с результатом трансляции | Имя файла |
| -logToConsole | Добавить вывод сообщений в консоль |  |

## 2.3 Перечень файлов, формируемых транслятором и их содержимое

Транслятор AAA-2018 в ходе своей работы формирует файл протокола работы, в котором содержатся таблица лексем, таблица идентификаторов, а также другая служебная информация и дерево разбора. Благодаря ему пользователь может обнаружить некорректно введенные данные или ошибку в исходном коде программы. Перечень всех формируемых танслятором файлов представлен в таблице 2.2.

Таблица 2.2 - Протоколы транслятора AAA-2018

|  |  |
| --- | --- |
| Файл | Содержимое |
| “file\_name.log” | Файл с протоколом работы транслятора языка программирования AAA-2018. Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа. |
| “file\_name.il ” | Сгенерированный файл с кодом программы на языке CIL |

# Глава 3. Разработка лексического анализатора

## 3.1 Структура лексического анализатора

Входными данными для лексического анализатора является массив текст, предварительно сформированный на первичной обработке исходного кода программы. Выходными данными являются таблица лексем, таблица идентификаторов, а также протокол работы. Структура лексического анализатора AAA-2018 представлена на рисунке 3.1.

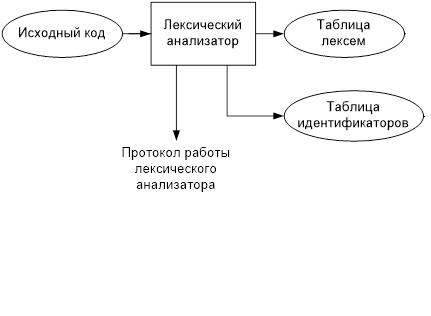


Рисунок 3.1 - Структура лексического анализатора AAA-2018

## 3.2 Контроль входных символов

Таблица для контроля входных символов представлена в приложении A. В таблице: F - запрещённый символ, T - разрешённый символ, I - игнорируемый символ.

## 3.3 Удаление избыточных символов

Описание алгоритма удаления избыточных символов, а также разбиения исходного кода на цепочки:

1. Посимвольно считываем файл с исходным кодом программы.

2. Если встречаем кавычку, то програма переходит в режим считывания значения литерала, в котором допустимы любые символы. Признаком окончания литерала является следующая кавычка.

3. При считывании символа сепаратора программа пытается распознать предыдущий фрагмент текста: какой лексеме он соотвествует.

## 3.4 Перечень ключевых слов, сепараторов, символов операций соответствующим им лексемам

Благодаря замене фрагментов текста на лексемы упрощается дальнейшая обработка исходного кода программы. Перечень цепочек символов и соответствующих им лексем, представлен в приложении А.

## 3.5 Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему, полученную при разборе, номер строки в исходном коде, является ли лексема арифметической операцией и номер в таблице идентификаторов, если лексема является идентификатором.

Таблица идентификаторов содержит имя идентификатора, номер в таблице лексем, тип данных, тип идентификатора и значение.

## 3.6 Структура и перечень сообщений лексического анализатора

Для обработки ошибок лексический анализатор использует таблицу с сообщениями, структура которой представлена в приложении А. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке.

Перечень сообщений, формируемых лексическим анализатором в ходе своей работы, представлен в таблице 3.1.

Таблица 3.1. Сообщения лексического анализатора

|  |  |
| --- | --- |
| Код ошибки | Сообщение |
| 220 | Недопустимый синтаксис |
| 221 | Превышен максимальный размер таблицы лексем |
| 222 | Превышен максимальный размер таблицы идентификаторов |
| 223 | Отсутствует закрывающаяся кавычка |
| 224 | Недопустимый номер возвращаемой строки таблицы лексем |
| 225 | Недопустимый номер возвращаемой строки таблицы идентификаторов |
| 226 | Открывающихся скобок '{' больше чем закрывающихся '}' |
| 227 | Для закрывающейся скобки '}' не найдена открывающаяся |
| 228 | Ключевое слово main встретилось в неположенном месте |
| 229 | Превышена максимальная длинна идентификатора |
| 230 | Не определен тип данных для индентификатора либо идентификатор не объявлен |
| 231 | Переопределение идентификатора |
| 232 | Функция main объявляется во второй раз |
| 233 | Объявление функции внутри функции недопустимо |
| 234 | Размер длины строки превышен (255) |
| 235 | Превышено максимальное значение целочисленного литерала |

## 3.7 Принцип обработки ошибок

В трансляторе AAA-2018 предусмотрена генерация ошибок - в случае ошибки работа прекращается

## 3.8 Параметры лексического анализатора и режимы его работы

Лексический анализатор принимает на вход считаный исходный файл и при прохождении его разбивает на фрагменты и по ним составляет таблицы лексем и идентификаторов. Если в коде встречается символ кавычки, то програма переходит в специальный режим считывания значения литерала, в котором допустимы любые символы. Признаком окончания литерала является следующая встреченная кавычка.

## 3.9 Алгоритм лексического анализа

Лексический анализ – первая и наиболее простая фаза трансляции. Алгоритм работы лексического анализатора заключается распознавании и разборе фрагментов исходного кода. Распознавание фрагментов основывается на работе конечных автоматов. Работу конечного автомата можно представить в виде графа. Пример графа представлен на рисунке 3.2. На рисунке осуществляется разбор цепочки “return”, где S0 – начальное состояние, а S6 – конечное.

https://yuml.me/cabbfc62.png

Рисунок 3.2 – Граф переходов для цепочки “return”

## 3.10 Контрольный пример

Результатом работы лексического анализатора являются таблица идентификаторов и таблица лексем. Содержимое таблиц лексем и идентификаторов для контрольного примера представлено в приложении А

# Глава 4. Разработка синтаксического анализатора

## 4.1 Структура Синтаксического анализатора

Синтаксический анализатор – следующая ступень транслятора. На вход анализатор принимает таблицу лексем, которую в ходе свой работы он перебирает с целью обнаружений в ней неверных синтаксических конструкций. Начинает свою работу только при условии отсутствия ошибок в их общей структуре. Если в ходе его работы не было обнаружено ошибок, то формируется дерево разбора. Структура синтаксического анализатора представлена на рисунке 4.1.



Рисунок 4.1 – Структура синтаксического анализатора

## 4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

Грамматика для синтаксического разбора языка AAA-2018 представляется четверкой G = <T, N, P, S>, где Т – множество терминальных символов, N – множество нетерминальных символов, P – множество правил языка, S – начальный символ грамматики, являющийся нетерминалом.

В грамматике языка AAA-2018 множество нетерминальных символов представлено в таблице 4.1.

Табл. 4.1 - Таблица правил переходов нетерминальных символов

|  |  |
| --- | --- |
| S→ | Порождает стартовые правила, описывающее общую конструкцию программы (функции) |
| N→ | Порождает правила, описывающие инструкции языка |
| E→ | Порождает правила, описывающие выражения |
| F→ | Порождает правила, описывающие формальные параметры функции |
| W→ | Порождает правила, описывающие принимаемые параметры функции |
| M→ | Порождает правила, описывающие арифметические действия |
| С→ | Порождает правила, описывающие условия для блока ветвления и цикла |

## 4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку,где Q – множество состояний автомата, V – алфавит входных символов, Z – алфавит специальных магазинных символов,  - функция переходов автомата,  - начальное состояние автомата,  - начальное состояние магазинного автомата, F – множество конечных состояний.

|  |  |  |
| --- | --- | --- |
| Таблица 4.2 - Описание компонентов магазинного автомата | | |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит представляет из себя множества терминальных и нетерминальных символов. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $) |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
|  | Начальное состояние магазина автомата | Представляет из себя символ маркера дна стека $ |
|  | Множество конечных состояний | Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты |

Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

## 4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного автомата и структуру грамматики Грейбах, описывающая правила языка AAA-2018. Данные структуры представлены в приложении Б.

## 4.5 Описание алгоритма синтаксического разбора

Алгоритм синтаксического разбора:

1. Поиск и выделение синтаксических конструкций в исходном тексте (разбор).
2. Распознавание (проверка правильности) синтаксических конструкций.
3. Выявление ошибок и продолжение процесса распознавания после обработки ошибок.
4. Если нет ошибок, формирование дерева разбора.

## 4.6 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений, формируемых синтаксическим анализатором в ходе своей работы, представлен в таблице 4.3.

Таблица 4.3. Сообщения синтаксического анализатора

|  |  |
| --- | --- |
| Код ошибки | Сообщение |
| 600 | Неверная структура программы |
| 601 | Ошибочный оператор |
| 602 | Ошибка в выражении |
| 603 | Ошибка в параметрах функции |
| 604 | Ошибка в параметрах вызываемой функции |
| 605 | Ошибка в условии ветвления или цикла |

## 4.7 Параметры синтаксического анализатора и режимы его работы

Синтаксический анализатор принимает в качестве параметров таблицу лексем и таблицу идентификаторов, а также правила разбора и при работе формирует дерево разбора.

## 4.8 Принцип обработки ошибок

Синтаксический анализатор перебирает всевозможные правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем. Если не была найдена ни одна подходящая цепочка, то формируется соответствующая ошибка.

## 4.9 Контрольный пример

Результатом работы синтаксического анализатора является дерево разбора. Дерево разбора исходного кода для контрольного примера представлено в приложении Б.

# Глава 5. Разработка семантического анализатор

## 5.1 Структура семантического анализатора

Семантический анализатор принимает на свой вход таблицы лексем и идентификаторов, а также дерево разбора. Он включает в себя несколько обособленных функций, осуществляющих проверки семантического характера. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1. Структура семантического анализатора

## 5.2 Функции семантического анализа

Семантический анализатор осуществляет проверки правил языка AAA-2018, которые описаны в пункте 1.19.

## 5.3 Структура и перечень сообщений семантического анализатора

Сообщения семантического анализа приведены в таблице 5.1.

Таблица 5.1. Сообщения, формируемые семантическим анализатором

|  |  |
| --- | --- |
| Код ошибки | Сообщение |
| 700 | Не совпадают типы данных в выражении |
| 701 | Не совпадают типы сравниваемых значений |
| 702 | Переменной присваивается значение неверного типа |
| 703 | Тип передаваемых данных не совпадает с типом данных для вызываемой функции |
| 704 | Тип возвращаемых из функции данных не соответствует описанию функции |
| 705 | Библиотечная функция c указанными параметрами не найдена |
| 706 | Ошибка синтаксиса в выражении |

## 5.4 Принцип обработки ошибок

Принцип обработки ошибок семантическим анализатором заключается в том, что работа программы прекращается в случае ошибки.

## 5.5 Контрольный пример

Протокол проверок семантического анализатора для контрольного примера приведен в приложении В.

# Глава 6. Преобразование выражений

## 6.1 Выражения, допускаемые языком

Выражения, допускаемые языком AAA-2018, выполняются над целочисленными типами данных. В выражениях поддерживаются арифметические операции, такие как +, -, \*, / и (), а также вызовы.

Приоритетность арифметических операций представлена в таблице 6.1.

Таблица 6.1 – Приоритетность операций

|  |  |
| --- | --- |
| Операция | Приоритет |
| ( | 4 |
| ) | 4 |
| \* | 3 |
| + | 2 |
| - | 2 |
| ( | 4 |

## 6.2 Польская запись и принцип ее построения

Польская запись — это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок. Ее отличие заключается в том, что знаки операций пишутся не между аргументами, а до или после них. Существует два типа польской записи: прямая и обратная, также известные как префиксная и постфиксная. Отличие их от классического, инфиксного способа заключается в том, что знаки операций пишутся не между, а, соответственно, до или после аргументов.

Алгоритм построения польской записи:

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

## 6.3 Программная реализация

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

## 6.4 Контрольный пример

Протокол вывода построения польской записи для всех выражений контрольного примера представлен в приложении В.

Преобразование выражений в формат польской записи необходимо для простой генерации кода в язык CIL на следующем этапе.

# Глава 7. Генерация кода

## 7.1 Структура генератора кода

Исходный код программы, написанный на языке AAA-2018, необходимо транслировать в CIL (Common Intermediate Language) для последующей его компиляции в исполняемый файл.

Трансляция будет осуществляться функцией generation, принимающей таблицу лексем и таблицу идентификаторов, а также дерево разбора. Выходным будет являться файл, генерируемый во время компиляции, который будет содержать в себе транслированный исходный код.

## 7.2 Представление типов данных в памяти

Соответствия между типами данных идентификаторов на языке AAA-2018 и приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка AAA-2018 и CIL

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке AAA-2018 | Тип идентификатора на языке ассемблера | Пояснение |
| integer | int32 | Хранит целочисленный тип данных. |
| string | string | Хранит целочисленный тип данных без знака. |

## 7.3 Статическая библиотека

В языке AAA-2018 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке. Объявление функций статической библиотеки генерируется автоматически. Список функций представлен в таблице 7.2.

Таблица 7.2 – Функции статической библиотеки

|  |  |
| --- | --- |
| Функция | Назначение |
| call void [mscorlib]System.Console::WriteLine(int32) | Вывод на экран числа |
| call void [mscorlib]System.Console::WriteLine(string) | Вывод на экран строки |
| .method private static int32 min (int32 x, int32 y) cil managed | Определяет минимальное значенние |
| .method private static int32 max (int32 x, int32 y) cil managed | Определяет максимальное значенние |
| .method private static int32 fact (int32 x) cil managed | Вычисление факториала. |
| callvirt instance int32 string::get\_Length() | Длинна строки |

## 7.4 Особенности алгоритма генерации кода

Алгоритм генерирования кода состоит в последовательном обходе дерева разбора и добавлением в результирующий файл нужных команд на языке CIL. Для каждого типа вершины дерева разбора предусмотрен свой код преобразования.

Для реализации поддержки циклов и ветвления предусмотрен специальный счетчик, позволяющий генерировать новые метки переходов.

## 7.5 Контрольный пример

Результат генерации кода на основе контрольного примера приведен в приложении Г.

После создания файла с расширением \*.il он может быть преобразован в исполняемый файл с помощью утилиты ilasm:

D:\main\Debug>C:\Windows\Microsoft.NET\Framework64\v2.0.50727\ilasm /exe ./test.il

В результате получим исполняемый файл. Результат работы программы контрольного примера также представлен в приложении Г.

# Глава 8. Тестирование транслятора

## 8.1 Тестирование проверки на допустимость символов

В языке AAA-2018 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 - Тестирование проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| var Ђ:integer; | Ошибка 111: Недопустимый символ в исходном файле (-іn), строка 23, позиция 6 |

## 8.2 Тестирование лексического анализатора

На этапе лексического анализа в языке AAA-2018 могут возникнуть ошибки, описанные в пункте 3.6. Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 - Тестирование лексического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| var dй:integer; | Ошибка 220: Недопустимый синтаксис, строка 23, позиция 8 |
| if [z < 100] {  z = z + 1000;  }}} else { | Ошибка 227: Для закрывающейся скобки '}' не найдена открывающаяся, строка 7, позиция 4 |
| …  cc = 5; | Ошибка 230: Не определен тип данных для индентификатора либо идентификатор не объявлен, строка 15, позиция 4 |

## 8.3 Тестирование синтаксического анализатора

На этапе синтаксического анализа в языке AAA-2018 могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 - Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main{  var x:integer;  return x;  var y:integer; | Ошибка 600: Неверная структура программы |
| if [z < 100] {  z = z + 1000;  }  return z; | 601: строка 9, Ошибочный оператор |
| z = z + 1000\*(z)); | 602: строка 6, Ошибка в выражении |

## 8.4 Тестирование семантического анализатора

Семантический анализ в языке AAA-2018 содержит множество проверок по семантическим правилам, описанным в пункте 1.16. Итоги тестирования семантического анализатора приведены в таблице 8.4.

Таблица 8.4 - Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| var sa:string = '10';  w = max(sa, 9); | Ошибка 703: Тип передаваемых данных не совпадает с типом данных для вызываемой функции, строка 37, позиция 12 |
| var sa:string = 10; | Ошибка 702: Переменной присваивается значение неверного типа, строка 26, позиция 5 |
| use function fact(x:integer, y:integer):integer; | Ошибка 705: Библиотечная функция c указанными параметрами не найдена, строка 29, позиция 19 |

# Заключение

В ходе выполнения курсовой работы был разработан транслятор для языка программирования AAA-2018. Таким образом, были выполнены основные задачи данной курсовой работы:

* Сформулирована спецификация языка AAA-2018;
* Разработаны конечные автоматы и алгоритмы для реализация лексического анализатора;
* Разработана контекстно-свободная, приведённая к нормальной форме грамматика для описания синтаксически верных конструкций языка;
* Разработан семантический анализатор, осуществляющий проверку смысла используемых инструкций;
* Разработан транслятор с языка программирования AAA-2018 на язык CIL;
* Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия AAA-2018 включает:

1. 2 типа данных;
2. Поддержка операции вывода;
3. Возможность вызова функций стандартной библиотеки;
4. Наличие 4 арифметических операторов для вычисления выражений;
5. Поддержка ветвления программного кода и циклов
6. Структурированная система для обработки ошибок пользователя.

# Приложение А

Таблица A.1. Коды контроля допустимых символов

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| код | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 0 | F | F | F | F | F | F | F | F | F | F | T | F | F | I | F | F |
| 1 | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| 2 | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |
| 3 | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |
| 4 | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |
| 5 | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |
| 6 | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |
| 7 | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | F |
| 8 | F | F | F | F | T | T | F | F | T | T | F | T | F | F | F | F |
| 9 | F | T | T | T | T | T | T | T | T | T | F | T | F | F | F | F |
| A | F | T | T | F | F | F | F | T | T | T | T | T | F | F | T | F |
| B | T | T | T | T | F | F | T | T | T | T | T | T | T | T | T | F |
| C | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |
| D | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |
| E | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |
| F | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |

Таблица А.2. Список лексем

|  |  |  |
| --- | --- | --- |
| Символ лексемы | Название в программе | Описание |
| 't' | LEX\_DATATYPE | integer/string |
| 'i' | LEX\_ID | идентификатор |
| 'l' | LEX\_LITERAL | литерал |
| 'f' | LEX\_FUNCTION | function |
| 'u' | LEX\_USE | use |
| 'v' | LEX\_VAR | var |
| 'r' | LEX\_RETURN | return |
| 'o' | LEX\_OUT | out |
| ';' | LEX\_SEMICOLON | ; |
| ':' | LEX\_COLON | : |
| ',' | LEX\_COMMA | , |
| '{' | LEX\_LEFTBRACE | { |
| '}' | LEX\_RIGTHBRACE | } |
| '(' | LEX\_LEFTHESIS | ( |
| ')' | LEX\_RIGHTHESIS | ) |
| '+' | LEX\_PLUS | + |
| '-' | LEX\_MINUS | - |
| '\*' | LEX\_STAR | \* |
| '/' | LEX\_DIRSLASH | / |
| '=' | LEX\_COMPARE | = |
| 'm' | LEX\_MAIN | main |
| 'w' | LEX\_WHILE | while |
| 'e' | LEX\_ELSE | else |
| 'c' | LEX\_IF | if |
| '<' | LEX\_LESS | < |
| '>' | LEX\_MORE | > |
| '!' | LEX\_EXCLAMATION | ! |
| '[' | LEX\_LEFT\_SQ\_BR | [ |
| ']' | LEX\_RIGTH\_SQ\_BR | ] |

Таблица А.3. Список лексем

|  |  |
| --- | --- |
| Начало | Продолжение |
| 01: fi[0](i[1]:t,i[2]:t):t  02: {  03: vi[3]:t;  04: i[3]=i[1]\*(i[1]+i[2]);  05: c[i[3]<l]{  06: i[3]=i[3]+l;  07: }e{  08: i[3]=i[3]+l;  09: };  10: ri[3];  11: };  12:  13: fi[7](i[8]:t,i[9]:t):t  14: {  15: vi[10]:t;  16: ufi[11](i[12]:t):t;  17: i[10]=i[11](i[8])+i[9];  18: ri[10];  19: };  20:  21: m  22: {  23: vi[14]:t;  24: vi[15]:t;  25: vi[16]:t;  26: vi[17]:t=l;  27: vi[19]:t;  28: ufi[20](i[21]:t):t;  29: ufi[22](i[23]:t):t;  30: ufi[24](i[25]:t,i[26]:t):t; | 31: ufi[27](i[28]:t,i[29]:t):t;  32: i[14]=l;  33: i[15]=l;  34: i[17]=l;  35: i[19]=l;  36: vi[34]:t;  37: i[34]=i[0](i[14],i[15]);  38: i[15]=i[7](i[17],i[14]);  39: ol;  40: oi[34];  41: i[16]=i[22](i[15]);  42: oi[16];  43: i[16]=i[27](i[14]+l,i[15]+l);  44: oi[16];  45: i[16]=i[24](i[14],i[15]);  46: oi[16];  47: oi[20](i[19]);  48: i[14]=i[14]\*(i[15]+i[0](i[20](i[17])+i[15],i[16]))-i[16]/i[0](i[14],i[15]);  49: oi[14];  50:  51: w[i[14]<l]{  52: c[i[14]<=l]{  53: oi[14];  54: }e{  55: oi[14]\*i[14];  56: };  57: i[14]=i[14]+l;  58: };  59:  60: }; |

Таблица А.4. Список идентификаторов

|  |
| --- |
| 00: f int fi [1]  01: p int fi.x [3] = <0>  02: p int fi.y [7] = <0>  03: v int fi.z [15] = <0>  04: l int [33] = <100>  05: l int [40] = <1000>  06: l int [49] = <2000>  07: f int fs [59]  08: p str fs.a [61] = <> lenght = 0  09: p int fs.b [65] = <0>  10: v int fs.c [73] = <0>  11: f int fs.strlen [79]  12: p str fs.strlen.p [81] = <> lenght = 0  13: f int main [102]  14: v int x [105] = <0>  15: v int y [110] = <0>  16: v int z [115] = <0>  17: v str sa [120] = <> lenght = 0  18: l str [124] = <1234567890> lenght = 12  19: v str sb [127] = <> lenght = 0  20: f int strlen [133]  21: p str strlen.p [135] = <> lenght = 0  22: f int fact [144]  23: p int fact.x [146] = <0>  24: f int min [155]  25: p int min.x [157] = <0>  26: p int min.y [161] = <0>  27: f int max [170]  28: p int max.x [172] = <0>  29: p int max.y [176] = <0>  30: l int [185] = <1>  31: l int [189] = <5>  32: l str [193] = <1234567890> lenght = 12  33: l str [197] = <1234567890> lenght = 12  34: v int w [200] = <0>  35: l str [223] = <контрольный пример> lenght = 20  36: l int [244] = <1>  37: l int [248] = <1>  38: l int [308] = <10>  39: l int [316] = <6>  40: l int [336] = <1> |

# Приложение Б

Структура хранения конечного автомата:

struct RELATION {

char symbol; // символ перехода

short nnode; // номер смежной вершины - новое состояние

};

struct NODE {

short. n\_relation; // количество инцидентных ребер

RELATION\* relations; // инцидентные ребра

};

struct FST {

const char\* string; // цепочка, строка завершающася 0x00

short position; // текущая позиция в цепочке

short nstates; // количество состояний в автомате

NODE\* nodes; // граф переходов: [0] - начальное состояние, [nstate-1] - конечное

short\* rstates; // возможные состояния автомата для данной позиции

}

Структура хранения грамматики:

struct Chain {

vector<TN\_SYMBOL> symbols; // цепочка терминалов (>0) и нетерминалов (<0)

};

struct Rule {

TN\_SYMBOL ruleSymbol; // нетерминал, левый символ правила < 0

int  errorId; // идентификатор сообщения об ошибке

vector<Chain> chains; // множество цепочек - правых частей правила

};

struct Grammar {

TN\_SYMBOL startSymbol; // стартовый символ

TN\_SYMBOL bottomSymbol; // дно стека

vector<Rule> rules; // множестов правил

};

Таблица Б.1. Вершины дерева разбора для контрольного примера

|  |  |  |  |
| --- | --- | --- | --- |
| Столбец 1 | Столбец 2 | Столбец 3 | Столбец 4 |
| |  | | --- | | S -> fi(F):t{NrE;};S | | F -> i:t,F | | F -> i:t | | N -> vi:t;N | | N -> i=E;N | | E -> iM | | M -> \*E | | E -> (E) | | E -> iM | | M -> +E | | E -> i | | N -> c[EC]{N}e{N}; | | E -> i | | C -> <E | | E -> l | | N -> i=E; | | E -> iM | | M -> +E | | E -> l | | N -> i=E; | | E -> iM | | M -> +E | | E -> l | | E -> i | | S -> fi(F):t{NrE;};S | | F -> i:t,F | | F -> i:t | | N -> vi:t;N | | N -> ufi(F):t;N | | F -> i:t | | N -> i=E; | | E -> i(W)M | | W -> i | | M -> +E | | |  | | --- | | E -> i | | E -> i | | S -> m{N}; | | N -> vi:t;N | | N -> vi:t;N | | N -> vi:t;N | | N -> vi:t=E;N | | E -> l | | N -> vi:t;N | | N -> ufi(F):t;N | | F -> i:t | | N -> ufi(F):t;N | | F -> i:t | | N -> ufi(F):t;N | | F -> i:t,F | | F -> i:t | | N -> ufi(F):t;N | | F -> i:t,F | | F -> i:t | | N -> i=E;N | | E -> l | | N -> i=E;N | | E -> l | | N -> i=E;N | | E -> l | | N -> i=E;N | | E -> l | | N -> vi:t;N | | N -> i=E;N | | E -> i(W) | | W -> i,W | | W -> i | | N -> i=E;N | | E -> i(W) | | |  | | --- | | W -> i,W | | W -> i | | N -> oE;N | | E -> l | | N -> oE;N | | E -> i | | N -> i=E;N | | E -> i(W) | | W -> i | | N -> oE;N | | E -> i | | N -> i=E;N | | E -> i(W) | | W -> iM,W | | M -> +E | | E -> l | | W -> iM | | M -> +E | | E -> l | | N -> oE;N | | E -> i | | N -> i=E;N | | E -> i(W) | | W -> i,W | | W -> i | | N -> oE;N | | E -> i | | N -> oE;N | | E -> i(W) | | W -> i | | N -> i=E;N | | E -> iM | | M -> \*E | | E -> (E)M | | |  | | --- | | E -> iM | | M -> +E | | E -> i(W) | | W -> i(W)M,W | | W -> i | | M -> +E | | E -> i | | W -> i | | M -> -E | | E -> iM | | M -> /E | | E -> i(W) | | W -> i,W | | W -> i | | N -> oE;N | | E -> i | | N -> w[EC]{N}; | | E -> i | | C -> <E | | E -> l | | N -> c[EC]{N}e{N};N | | E -> i | | C -> <=E | | E -> l | | N -> oE; | | E -> i | | N -> oE; | | E -> iM | | M -> \*E | | E -> i | | N -> i=E; | | E -> iM | | M -> +E | | E -> l | |

# Приложение В

Протокол работы семантического анализатора для контрольного примера. Было выполнено 60 проверок семантических правил:

Verify node: 7 :F -> i:t : int

Verify node: 3 :F -> i:t,F : int

Verify node: 26 :E -> i : int

Verify node: 25 :M -> +E : int

Verify node: 24 :E -> iM :

Checked (700) 24 == 25 : int

Verify node: 23 :E -> (E) : int

Verify node: 22 :M -> \*E : int

Verify node: 21 :E -> iM :

Checked (700) 21 == 22 : int

Verify node: 31 :E -> i : int

Verify node: 33 :E -> l : int

Verify node: 32 :C -> <E : int

Verify node: 40 :E -> l : int

Verify node: 39 :M -> +E : int

Verify node: 38 :E -> iM :

Checked (700) 38 == 39 : int

Verify node: 36 :N -> i=E; :

Checked (702) 36 == 38 : int

Verify node: 49 :E -> l : int

Verify node: 48 :M -> +E : int

Verify node: 47 :E -> iM :

Checked (700) 47 == 48 : int

Verify node: 45 :N -> i=E; :

Checked (702) 45 == 47 : int

Verify node: 29 :N -> c[EC]{N}e{N}; :

Checked (701) 31 == 33 : int

Verify node: 19 :N -> i=E;N :

Checked (702) 19 == 21 : int

Verify node: 14 :N -> vi:t;N : int

Verify node: 54 :E -> i : int

Verify node: 65 :F -> i:t : int

Verify node: 61 :F -> i:t,F : str

Verify node: 81 :F -> i:t : str

Verify node: 92 :W -> i :

Checked (703) 92 == 90 [1] : str

Verify node: 95 :E -> i : int

Verify node: 94 :M -> +E : int

Verify node: 90 :E -> i(W)M :

Checked (700) 90 == 94 : int

Verify node: 88 :N -> i=E; :

Checked (702) 88 == 90 : int

Verify node: 77 :N -> ufi(F):t;N :

Checked (705) 79 == -1 : int

Verify node: 72 :N -> vi:t;N : int

Verify node: 98 :E -> i : int

Verify node: 124 :E -> l : str

Verify node: 135 :F -> i:t : str

Verify node: 146 :F -> i:t : int

Verify node: 161 :F -> i:t : int

Verify node: 157 :F -> i:t,F : int

Verify node: 176 :F -> i:t : int

Verify node: 172 :F -> i:t,F : int

Verify node: 185 :E -> l : int

Verify node: 189 :E -> l : int

Verify node: 193 :E -> l : str

Verify node: 197 :E -> l : str

Verify node: 210 :W -> i :

Checked (703) 210 == 206 [2] : int

Verify node: 208 :W -> i,W :

Checked (703) 208 == 206 [1] : int

Verify node: 206 :E -> i(W) : int

Verify node: 219 :W -> i :

Checked (703) 219 == 215 [2] : int

Verify node: 217 :W -> i,W :

Checked (703) 217 == 215 [1] : str

Verify node: 215 :E -> i(W) : int

Verify node: 223 :E -> l : str

Verify node: 226 :E -> i : int

Verify node: 232 :W -> i :

Checked (703) 232 == 230 [1] : int

Verify node: 230 :E -> i(W) : int

Verify node: 236 :E -> i : int

Verify node: 244 :E -> l : int

Verify node: 243 :M -> +E : int

Verify node: 248 :E -> l : int

Verify node: 247 :M -> +E : int

Verify node: 246 :W -> iM :

Checked (700) 246 == 247 :

Checked (703) 246 == 240 [2] : int

Verify node: 242 :W -> iM,W :

Checked (700) 242 == 243 :

Checked (703) 242 == 240 [1] : int

Verify node: 240 :E -> i(W) : int

Verify node: 252 :E -> i : int

Verify node: 260 :W -> i :

Checked (703) 260 == 256 [2] : int

Verify node: 258 :W -> i,W :

Checked (703) 258 == 256 [1] : int

Verify node: 256 :E -> i(W) : int

Verify node: 264 :E -> i : int

Verify node: 269 :W -> i :

Checked (703) 269 == 267 [1] : str

Verify node: 267 :E -> i(W) : int

Verify node: 283 :W -> i :

Checked (703) 283 == 281 [1] : str

Verify node: 286 :E -> i : int

Verify node: 285 :M -> +E : int

Verify node: 288 :W -> i :

Checked (703) 288 == 279 [2] : int

Verify node: 281 :W -> i(W)M,W :

Checked (700) 281 == 285 :

Checked (703) 281 == 279 [1] : int

Verify node: 279 :E -> i(W) : int

Verify node: 278 :M -> +E : int

Verify node: 277 :E -> iM :

Checked (700) 277 == 278 : int

Verify node: 298 :W -> i :

Checked (703) 298 == 294 [2] : int

Verify node: 296 :W -> i,W :

Checked (703) 296 == 294 [1] : int

Verify node: 294 :E -> i(W) : int

Verify node: 293 :M -> /E : int

Verify node: 292 :E -> iM :

Checked (700) 292 == 293 : int

Verify node: 291 :M -> -E : int

Verify node: 276 :E -> (E)M :

Checked (700) 276 == 279 : int

Verify node: 275 :M -> \*E : int

Verify node: 274 :E -> iM :

Checked (700) 274 == 275 : int

Verify node: 302 :E -> i : int

Verify node: 306 :E -> i : int

Verify node: 308 :E -> l : int

Verify node: 307 :C -> <E : int

Verify node: 313 :E -> i : int

Verify node: 316 :E -> l : int

Verify node: 314 :C -> <=E : int

Verify node: 320 :E -> i : int

Verify node: 319 :N -> oE; : int

Verify node: 328 :E -> i : int

Verify node: 327 :M -> \*E : int

Verify node: 326 :E -> iM :

Checked (700) 326 == 327 : int

Verify node: 325 :N -> oE; : int

Verify node: 336 :E -> l : int

Verify node: 335 :M -> +E : int

Verify node: 334 :E -> iM :

Checked (700) 334 == 335 : int

Verify node: 332 :N -> i=E; :

Checked (702) 332 == 334 : int

Verify node: 311 :N -> c[EC]{N}e{N};N :

Checked (701) 313 == 316 : int

Verify node: 304 :N -> w[EC]{N}; :

Checked (701) 306 == 308 : int

Verify node: 301 :N -> oE;N : int

Verify node: 272 :N -> i=E;N :

Checked (702) 272 == 274 : int

Verify node: 266 :N -> oE;N : int

Verify node: 263 :N -> oE;N : int

Verify node: 254 :N -> i=E;N :

Checked (702) 254 == 256 : int

Verify node: 251 :N -> oE;N : int

Verify node: 238 :N -> i=E;N :

Checked (702) 238 == 240 : int

Verify node: 235 :N -> oE;N : int

Verify node: 228 :N -> i=E;N :

Checked (702) 228 == 230 : int

Verify node: 225 :N -> oE;N : int

Verify node: 222 :N -> oE;N : str

Verify node: 213 :N -> i=E;N :

Checked (702) 213 == 215 : int

Verify node: 204 :N -> i=E;N :

Checked (702) 204 == 206 : int

Verify node: 199 :N -> vi:t;N : int

Verify node: 195 :N -> i=E;N :

Checked (702) 195 == 197 : str

Verify node: 191 :N -> i=E;N :

Checked (702) 191 == 193 : str

Verify node: 187 :N -> i=E;N :

Checked (702) 187 == 189 : int

Verify node: 183 :N -> i=E;N :

Checked (702) 183 == 185 : int

Verify node: 168 :N -> ufi(F):t;N :

Checked (705) 170 == -1 : int

Verify node: 153 :N -> ufi(F):t;N :

Checked (705) 155 == -1 : int

Verify node: 142 :N -> ufi(F):t;N :

Checked (705) 144 == -1 : int

Verify node: 131 :N -> ufi(F):t;N :

Checked (705) 133 == -1 : int

Verify node: 126 :N -> vi:t;N : str

Verify node: 119 :N -> vi:t=E;N :

Checked (702) 120 == 124 : str

Verify node: 114 :N -> vi:t;N : int

Verify node: 109 :N -> vi:t;N : int

Verify node: 104 :N -> vi:t;N : int

Verify node: 102 :S -> m{N}; : <unknown>

Verify node: 58 :S -> fi(F):t{NrE;};S :

Checked (704) 59 == 98 : int

Verify node: 0 :S -> fi(F):t{NrE;};S :

Checked (704) 1 == 54 : int

Программная реализация построения польской нотации:

int buildRPN(TranslationContext &ctx, const int start) {

LT::LexTable &lexTable = ctx.lexTable;

\*ctx.logger << "\nTry to build polish notation from " << start << " position:" << endl;

printExpression(\*ctx.logger, start, lexTable);

\*ctx.logger << "----------------------------------" << endl;

bool ok = true;

std::stack<char> stack;

std::stack<int>  functionStack;

int i = start;

int r = start; // result index

while (i < lexTable.table.size() && lexTable.table[i].lexema != LEX\_SEMICOLON && lexTable.table[i].lexema != LEX\_COMPARE &&

lexTable.table[i].lexema != LEX\_MORE && lexTable.table[i].lexema != LEX\_LESS && lexTable.table[i].lexema != LEX\_RIGTH\_SQ\_BR &&

ok) {

char c = lexTable.table[i].lexema;

switch (c) {

case LEX\_LEFTHESIS: {

if ((i - 1 < 0) || (lexTable.table[i - 1].lexema != LEX\_ID)) {

stack.push(LEX\_LEFTHESIS);

}

break;

}

case LEX\_PLUS:

case LEX\_MINUS:

case LEX\_STAR:

case LEX\_DIRSLASH: {

// extract all operations from stack with high or equal priority and move them to result

bool extractOperation = !stack.empty();

while (extractOperation) {

char o = stack.top();

if (priority(o) >= priority(c)) {

stack.pop();

lexTable.table[r].lexema     = o;

lexTable.table[r].lexemaType = LA::LT\_SIGN;

lexTable.table[r].idxTI  = LT\_TI\_NULLIDX;

r++;

if (stack.empty()) {

extractOperation = false;

}

} else {

extractOperation = false;

}

}

stack.push(c);

break;

}

case LEX\_ID:

case LEX\_LITERAL: {

if ((i + 1 < lexTable.table.size()) && (lexTable.table[i + 1].lexema == LEX\_LEFTHESIS)) {

// start of function. push "["

stack.push(LEX\_LEFT\_BRACKET);

functionStack.push(lexTable.table[i].idxTI);

} else {

// move operand to result position

lexTable.table[r] = lexTable.table[i];

r++;

}

break;

}

case LEX\_COMMA: {

stack.push(LEX\_COMMA);

break;

}

case LEX\_RIGHTHESIS: {

// extract all operations from stack while find open bracket or paranthesys and move them to result

int  paramsCount     = 1; // number of function parameters

bool extractOperation = !stack.empty();

char o       = LEX\_LEFTHESIS;

while (extractOperation) {

o = stack.top();

stack.pop();

if ((o != LEX\_LEFTHESIS) && (o != LEX\_LEFT\_BRACKET)) {

if (o == LEX\_COMMA) {

paramsCount++;

} else {

lexTable.table[r].lexema = o;

lexTable.table[r].lexemaType = LA::LT\_SIGN;

lexTable.table[r].idxTI  = LT\_TI\_NULLIDX;

r++;

}

if (stack.empty()) {

extractOperation = false;

}

} else {

extractOperation = false;

}

}

if ((paramsCount == 1) && (i > start) && (lexTable.table[i - 1].lexema == LEX\_LEFTHESIS)) {

paramsCount = 0;

}

// for close bracket generate @ sign (function placeholder)

if (o == LEX\_LEFT\_BRACKET) {

lexTable.table[r].lexema = LEX\_FUNCTION\_REF;

lexTable.table[r].lexemaType = LA::LT\_SIGN;

int functionIdx = functionStack.top();

functionStack.pop();

lexTable.table[r].idxTI = functionIdx; // нужен индекс ид объявления функции

r++;

} else {

if (paramsCount > 1) {

ok = false; // found comma inside paranthesys => stop parsing (syntax error)

}

}

break;

}

case LEX\_EMPTY: {

break;

}

default: {

ok = false; // found wrong lexema => stop parsing (syntax error)

}

}

i++;

printStep(\*ctx.logger, c, start, r, lexTable, stack);

}

while (!stack.empty()) {

char o = stack.top();

stack.pop();

lexTable.table[r].lexema     = o;

lexTable.table[r].lexemaType = LA::LT\_SIGN;

lexTable.table[r].idxTI  = LT\_TI\_NULLIDX;

r++;

}

while (r < i) {

lexTable.table[r].lexema     = LEX\_EMPTY;

lexTable.table[r].lexemaType = LA::LT\_SIGN;

lexTable.table[r].idxTI  = LT\_TI\_NULLIDX;

r++;

}

printStep(\*ctx.logger, ' ', start, r, lexTable, stack);

\*ctx.logger << "----------------------------------" << endl;

if (ok) {

\*ctx.logger << "польская запись построена:" << endl;

printExpression(\*ctx.logger, start, lexTable);

} else {

\*ctx.logger << "польская запиcь не построена" << endl;

}

\*ctx.logger << "=================================" << endl;

return ok ? (r - start) : (start - r);

}

Протокол вывода построения польской нотации для всех выражений в контрольном примере:

Try to build polish notation from 21 position:

21: i1

22: \*

23: (

24: i1

25: +

26: i2

27: )

----------------------------------

i: i1

\*: \* i1

(: (\* i1

i: (\* i1 i1

+: +(\* i1 i1

i: +(\* i1 i1 i2

): \* i1 i1 i2 +

: i1 i1 i2 + \*

----------------------------------

польская запись построена:

21: i1

22: i1

23: i2

24: +

25: \*

26:

27:

=================================

Try to build polish notation from 31 position:

31: i3

----------------------------------

i: i3

: i3

----------------------------------

польская запись построена:

31: i3

=================================

Try to build polish notation from 33 position:

33: l

----------------------------------

l: l

: l

----------------------------------

польская запись построена:

33: l

=================================

Try to build polish notation from 38 position:

38: i3

39: +

40: l

----------------------------------

i: i3

+: + i3

l: + i3 l

: i3 l +

----------------------------------

польская запись построена:

38: i3

39: l

40: +

=================================

Try to build polish notation from 47 position:

47: i3

48: +

49: l

----------------------------------

i: i3

+: + i3

l: + i3 l

: i3 l +

----------------------------------

польская запись построена:

47: i3

48: l

49: +

=================================

Try to build polish notation from 54 position:

54: i3

----------------------------------

i: i3

: i3

----------------------------------

польская запись построена:

54: i3

=================================

Try to build polish notation from 90 position:

90: i11

91: (

92: i8

93: )

94: +

95: i9

----------------------------------

i: [

(: [

i: [ i8

): i8 @11

+: + i8 @11

i: + i8 @11 i9

: i8 @11 i9 +

----------------------------------

польская запись построена:

90: i8

91: @11

92: i9

93: +

94:

95:

=================================

Try to build polish notation from 98 position:

98: i10

----------------------------------

i: i10

: i10

----------------------------------

польская запись построена:

98: i10

=================================

Try to build polish notation from 124 position:

124: l

----------------------------------

l: l

: l

----------------------------------

польская запись построена:

124: l

=================================

Try to build polish notation from 185 position:

185: l

----------------------------------

l: l

: l

----------------------------------

польская запись построена:

185: l

=================================

Try to build polish notation from 189 position:

189: l

----------------------------------

l: l

: l

----------------------------------

польская запись построена:

189: l

=================================

Try to build polish notation from 193 position:

193: l

----------------------------------

l: l

: l

----------------------------------

польская запись построена:

193: l

=================================

Try to build polish notation from 197 position:

197: l

----------------------------------

l: l

: l

----------------------------------

польская запись построена:

197: l

=================================

Try to build polish notation from 206 position:

206: i0

207: (

208: i14

209: ,

210: i15

211: )

----------------------------------

i: [

(: [

i: [ i14

,: ,[ i14

i: ,[ i14 i15

): i14 i15 @0

: i14 i15 @0

----------------------------------

польская запись построена:

206: i14

207: i15

208: @0

209:

210:

211:

=================================

Try to build polish notation from 215 position:

215: i7

216: (

217: i17

218: ,

219: i14

220: )

----------------------------------

i: [

(: [

i: [ i17

,: ,[ i17

i: ,[ i17 i14

): i17 i14 @7

: i17 i14 @7

----------------------------------

польская запись построена:

215: i17

216: i14

217: @7

218:

219:

220:

=================================

Try to build polish notation from 223 position:

223: l

----------------------------------

l: l

: l

----------------------------------

польская запись построена:

223: l

=================================

Try to build polish notation from 226 position:

226: i34

----------------------------------

i: i34

: i34

----------------------------------

польская запись построена:

226: i34

=================================

Try to build polish notation from 230 position:

230: i22

231: (

232: i15

233: )

----------------------------------

i: [

(: [

i: [ i15

): i15 @22

: i15 @22

----------------------------------

польская запись построена:

230: i15

231: @22

232:

233:

=================================

Try to build polish notation from 236 position:

236: i16

----------------------------------

i: i16

: i16

----------------------------------

польская запись построена:

236: i16

=================================

Try to build polish notation from 240 position:

240: i27

241: (

242: i14

243: +

244: l

245: ,

246: i15

247: +

248: l

249: )

----------------------------------

i: [

(: [

i: [ i14

+: +[ i14

l: +[ i14 l

,: ,+[ i14 l

i: ,+[ i14 l i15

+: +,+[ i14 l i15

l: +,+[ i14 l i15 l

): i14 l i15 l + + @27

: i14 l i15 l + + @27

----------------------------------

польская запись построена:

240: i14

241: l

242: i15

243: l

244: +

245: +

246: @27

247:

248:

249:

=================================

Try to build polish notation from 252 position:

252: i16

----------------------------------

i: i16

: i16

----------------------------------

польская запись построена:

252: i16

=================================

Try to build polish notation from 256 position:

256: i24

257: (

258: i14

259: ,

260: i15

261: )

----------------------------------

i: [

(: [

i: [ i14

,: ,[ i14

i: ,[ i14 i15

): i14 i15 @24

: i14 i15 @24

----------------------------------

польская запись построена:

256: i14

257: i15

258: @24

259:

260:

261:

=================================

Try to build polish notation from 264 position:

264: i16

----------------------------------

i: i16

: i16

----------------------------------

польская запись построена:

264: i16

=================================

Try to build polish notation from 267 position:

267: i20

268: (

269: i19

270: )

----------------------------------

i: [

(: [

i: [ i19

): i19 @20

: i19 @20

----------------------------------

польская запись построена:

267: i19

268: @20

269:

270:

=================================

# Приложение Г

Результат трансляции контрольного примера в язык CIL

.assembly extern mscorlib {}

.assembly Program {}

.module Program

.method private static int32 fact (int32 x) cil managed {

    ldarg.0

    ldc.i4 1

    cgt

    brfalse.s IL\_0018

    ldarg.0

    ldarg.0

    ldc.i4.1

    sub

    call int32 fact(int32)

    mul

    br.s IL\_001c

IL\_0018: nop

ldc.i4.1

IL\_001c: nop

    ret

}

.method private static int32 max (int32 x, int32 y) cil managed {

    ldarg.0

    ldarg.1

    cgt

brfalse.s IL\_000e

    ldarg.0

    br.s IL\_0013

IL\_000e: nop

    ldarg.1

IL\_0013: nop

ret

}

.method private static int32 min (int32 x, int32 y) cil managed {

    ldarg.0

    ldarg.1

    cgt

brfalse.s IL\_000e

    ldarg.1

    br.s IL\_0013

IL\_000e: nop

    ldarg.0

IL\_0013: nop

    ret

}

.method private static int32 fi(int32 x, int32 y) cil managed {

    .maxstack 100

    .locals init(int32 z)

    ldarg x

    ldarg x

    ldarg y

    add

    mul

    stloc z

    ldloc z

    ldc.i4 100

    blt.s LAB\_2

    ldloc z

    ldc.i4 2000

    add

    stloc z

    br.s LAB\_3

LAB\_2: nop

    ldloc z

    ldc.i4 1000

    add

    stloc z

LAB\_3: nop

    ldloc z

    ret

}

.method private static int32 fs(string a, int32 b) cil managed {

    .maxstack 100

    .locals init(int32 c)

    ldarg a

    callvirt instance int32 string::get\_Length()

    ldarg b

    add

    stloc c

    ldloc c

    ret

}

.method private static void main() cil managed {

    .entrypoint

    .maxstack 100

    .locals init(int32 x)

    .locals init(int32 y)

    .locals init(int32 z)

    .locals init(string sa)

    ldstr "1234567890"

    stloc sa

    .locals init(string sb)

    ldc.i4 1

    stloc x

    ldc.i4 5

    stloc y

    ldstr "1234567890"

    stloc sa

    ldstr "1234567890"

    stloc sb

    .locals init(int32 w)

    ldloc x

    ldloc y

    call int32 fi(int32, int32)

    stloc w

    ldloc sa

    ldloc x

    call int32 fs(string, int32)

    stloc y

    ldstr "контрольный пример"

    call void [mscorlib]System.Console::WriteLine(string)

    ldloc w

    call void [mscorlib]System.Console::WriteLine(int32)

    ldloc y

    call int32 fact(int32)

    stloc z

    ldloc z

    call void [mscorlib]System.Console::WriteLine(int32)

    ldloc x

    ldc.i4 1

    ldloc y

    ldc.i4 1

    add

    add

    call int32 max(int32, int32)

    stloc z

    ldloc z

    call void [mscorlib]System.Console::WriteLine(int32)

    ldloc x

    ldloc y

    call int32 min(int32, int32)

    stloc z

    ldloc z

    call void [mscorlib]System.Console::WriteLine(int32)

    ldloc sb

    callvirt instance int32 string::get\_Length()

    call void [mscorlib]System.Console::WriteLine(int32)

    ldloc x

    ldloc y

    ldloc sa

    callvirt instance int32 string::get\_Length()

    ldloc y

    ldloc z

    add

    call int32 fi(int32, int32)

    add

    mul

    ldloc z

    ldloc x

    ldloc y

    call int32 fi(int32, int32)

    div

    sub

    stloc x

    ldloc x

    call void [mscorlib]System.Console::WriteLine(int32)

LAB\_4: nop

    ldloc x

    ldc.i4 10

    blt.s LAB\_5

    br.s LAB\_6

LAB\_5: nop

    ldloc x

    ldc.i4 6

    ble.s LAB\_7

    ldloc x

    ldloc x

    mul

    call void [mscorlib]System.Console::WriteLine(int32)

    br.s LAB\_8

LAB\_7: nop

    ldloc x

    call void [mscorlib]System.Console::WriteLine(int32)

LAB\_8: nop

    ldloc x

    ldc.i4 1

    add

    stloc x

    br.s LAB\_4

LAB\_6: nop

    ret

}

Результат работы программы контрольного примера:

D:\main\Debug>test.exe

контрольный пример

1006

39916800

13

1

10

2231

# Литература

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

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

3. Орлов, С.А. Теория и практика языков программирования / С.А. Орлов – 2014. – 689 с.

4. Смелов, В. В. Введение в объектно-ориентированное программирование на С++ / В. В. Смелов.– Минск : БГТУ, 2009. - 94с.

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

6. \_\_stdcall [Электронный ресурс] – Режим доступа: https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx.

7. Common Intermediate Language [Электронный ресурс] – Режим доступа: https://en.wikipedia.org/wiki/Common\_Intermediate\_Language

8. Common Intermediate Language [Электронный ресурс] – Режим доступа: https://www.intuit.ru/studies/courses/89/89/lecture/28305?page=3