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 RAM-2022»

Выполнил студент Романчук Алексей Михайлович

(Ф.И.О.)

Руководитель проекта ст. преп., Наркевич А. С.

(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. преп., Наркевич А. С.

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

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

Нормоконтролер ст. преп., Наркевич А. С.

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

Курсовой проект защищен с оценкой

Минск 2022

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

“ ” 202\_г.

**ЗАДАНИЕ**

**к курсовому проектированию по дисциплине**

«Конструирование программного обеспечения»

Специальность: ПОИТ Группа: 4 Студент: Романчук Алексей Михайлович

(фамилия, имя, отчество)

1. **Тема проекта**  Разработка компилятора RAM-2022 утверждена приказом по университету от « » 202\_ г. №
2. **Срок сдачи студентом законченного проекта:** 19 декабря 2022 г.
3. **Исходные данные к проекту:**

Разработка программы осуществляется на языке C++ (стандартизации International Standard ISO/IEC 14882:2020(E) Programming Language C++ 20) в среде разработки Visual Studio Community 2022 17.3.6. Операционная система, под которой происходит разработка: Windows 10 Pro 21H1 (64-bit). Типы данных: целочисленный, строковый. Представление литералов: целые – восьмеричное | десятичное, строковые | символьные – произвольно. Функции стандартной библиотеки: получение строки с текущей датой, получение строки с текущим временем. Арифметические операции: +, -, \*, /, %. Оператор вывода в стандартный поток: write. Инструкции: условный оператор, оператор цикла.

1. **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)
9. Заключение
10. Литература Приложения Литература
11. **Перечень графического материала (с точным указанием обязательных чертежей)**

1) Граф дерева разбора

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

|  |  |
| --- | --- |
| Раздел | Консультант |
| Разработка лексического анализатора | Наркевич А. С. |
| 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. |

1. **Календарный план**

|  |  |  |  |
| --- | --- | --- | --- |
| №  п/п | Наименование этапов курсового проекта | Срок выполнения  этапов проекта | Примечание |
| 1 | Спецификация специализированного  языка RAM-2022 | 27.09.2022 |  |
| 2 | Разработка лексического анализатора | 11.10.2022 |  |
| 3 | Разработка синтаксического анализатора | 25.10.2022 |  |
| 4 | Разработка семантического анализатора | 08.11.2022 |  |
| 5 | Генерация кода | 15.11.2022 |  |
| 6 | Тестирование компилятора | 22.11.2022 |  |
| 7 | Оформление пояснительной записки к  курсовому проект | 13.12.2022 |  |
| 8 | Сдача проекта | 19.12.2022 |  |

1. **Дата выдачи задания 15.09.2022**

Руководитель Наркевич А. С.

(подпись)

Задание принял к исполнению Романчук А. М.

(дата и подпись студента)

**Содержание**

[Введение 6](#_Toc122297063)

[1 Спецификация языка программирования 7](#_Toc122297064)

[1.1 Характеристика языка программирования 7](#_Toc122297065)

[1.2 Определение алфавита языка программирования 7](#_Toc122297066)

[1.3 Применяемые сепараторы 7](#_Toc122297067)

[1.4 Применяемые кодировки 7](#_Toc122297068)

[1.5 Типы данных 8](#_Toc122297069)

[1.6 Преобразование типов данных 9](#_Toc122297070)

[1.7 Идентификаторы 9](#_Toc122297071)

[1.8 Литералы 9](#_Toc122297072)

[1.9 Объявление данных 10](#_Toc122297073)

[1.10 Инициализация данных 10](#_Toc122297074)

[1.11 Инструкции языка 10](#_Toc122297075)

[1.12 Операции языка 11](#_Toc122297076)

[1.13 Выражения и их вычисление 12](#_Toc122297077)

[1.14 Конструкции языка 12](#_Toc122297078)

[1.15 Область видимости идентификаторов 13](#_Toc122297079)

[1.16 Семантические проверки 13](#_Toc122297080)

[1.17 Распределение оперативной памяти на этапе выполнения 14](#_Toc122297081)

[1.18 Стандартная библиотека и её состав 14](#_Toc122297082)

[1.19 Ввод и вывод данных 15](#_Toc122297083)

[1.20 Точка входа 15](#_Toc122297084)

[1.21 Препроцессор 15](#_Toc122297085)

[1.22 Соглашения о вызовах 15](#_Toc122297086)

[1.23 Объектный код 15](#_Toc122297087)

[1.24 Классификация сообщений транслятора 15](#_Toc122297088)

[1.25 Контрольный пример 16](#_Toc122297089)

[2 Структура транслятора 17](#_Toc122297090)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 17](#_Toc122297091)

[2.2 Перечень входных параметров транслятора 17](#_Toc122297092)

[2.3 Протоколы, формируемые транслятором 18](#_Toc122297093)

[3. Разработка лексического анализатора 19](#_Toc122297094)

[3.1 Структура лексического анализатора 19](#_Toc122297095)

[3.2 Контроль входных символов 19](#_Toc122297096)

[3.3 Удаление избыточных символов 19](#_Toc122297097)

[3.4 Перечень ключевых слов 20](#_Toc122297098)

[3.5 Основные структуры данных 21](#_Toc122297099)

[3.6 Структура и перечень сообщений лексического анализатора. 24](#_Toc122297100)

[3.7 Принцип обработки ошибок 25](#_Toc122297101)

[3.8 Параметры лексического анализатора 26](#_Toc122297102)

[3.9 Алгоритм лексического анализа 26](#_Toc122297103)

[3.10 Контрольный пример 26](#_Toc122297104)

[4. Разработка синтаксического анализатора 27](#_Toc122297105)

[4.1 Структура синтаксического анализатора 27](#_Toc122297106)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 27](#_Toc122297107)

[4.3 Построение конечного магазинного автомата 29](#_Toc122297108)

[4.4 Основные структуры данных 30](#_Toc122297109)

[4.5 Описание алгоритма синтаксического разбора 31](#_Toc122297110)

[4.6 Структура и перечень сообщений синтаксического анализа 31](#_Toc122297111)

[4.7 Параметры синтаксического анализатора и режимы его работы 31](#_Toc122297112)

[4.8 Принцип обработки ошибок 31](#_Toc122297113)

[4.9 Контрольный пример 32](#_Toc122297114)

[5 Разработка семантического анализатора 33](#_Toc122297115)

[5.1 Структура семантического анализатора 33](#_Toc122297116)

[5.2 Функции семантического анализатора 33](#_Toc122297117)

[5.3 Структура и перечеь сообщений семантического анализатора 34](#_Toc122297118)

[5.4 Принцип обработки ошибок 35](#_Toc122297119)

[5.5 Контрольный пример 35](#_Toc122297120)

[6 Вычисление выражений 36](#_Toc122297121)

[6.1 Выражения, допускаемые языком 36](#_Toc122297122)

[6.2 Польская запись и принцип её построения 36](#_Toc122297123)

[6.3 Программная реализация обработки выражений 37](#_Toc122297124)

[6.4 Контрольный пример 37](#_Toc122297125)

[7 Генерация кода 38](#_Toc122297126)

[7.1 Структура генератора кода 38](#_Toc122297127)

[7.2 Представление типов данных в оперативной памяти 38](#_Toc122297128)

[7.3 Статическая библиотека 38](#_Toc122297129)

[7.4 Особенности алгоритма генерации кода 39](#_Toc122297130)

[7.5 Входные параметры, управляющие генерацией кода 40](#_Toc122297131)

[7.6 Контрольный пример 40](#_Toc122297132)

[8 Тестирование транслятора 41](#_Toc122297133)

[8.1 Общие положения 41](#_Toc122297134)

[8.2 Результаты тестирования 41](#_Toc122297135)

[Заключение 44](#_Toc122297136)

[Список использованных источников 45](#_Toc122297137)

[Приложение А 46](#_Toc122297138)

[Приложение Б 47](#_Toc122297139)

[Приложение В 48](#_Toc122297140)

[Приложение Г 49](#_Toc122297141)

[Приложение Д 52](#_Toc122297142)

[Приложение Е 56](#_Toc122297143)

[Приложение Ж 57](#_Toc122297144)

[Приложение И 59](#_Toc122297145)

[Приложение К 62](#_Toc122297146)

[Приложение Л 63](#_Toc122297147)

# Введение

Целью выполнения данного курсового проекта является приобретение навыков разработки системы программирования (транслятора) для собственного языка программирования. Название языка программирования – “RAM-2022”. Трансляция осуществляется в язык MASM.

Для достижения цели курсового проекта необходимо выполнить следующие задачи:

* разработать спецификацию языка программирования;
* разработать программную реализацию лексического анализатора;
* разработать программную реализацию синтаксического анализатора;
* разработать программную реализацию семантического анализатора;
* разработать программную реализацию генератора кода;
* выполнить тестирование транслятора.

В разделе 1 описана спецификация языка; в разделе 2 – общая структура транслятора; в разделе 3 – разработка лексического анализатора, в разделе 4 – разработка синтаксического анализатора; в разделе 5 – разработка семантического анализатора; в разделе 6 – выражения, их форма, вычисление; в разделе 7 – генерация кода; в разделе 8 – тестирование транслятора.

Минимальные требования к языку RAM-2022 включают:

* не менее двух типов данных, поддерживаемых языком;
* наличие нескольких видов программных блоков (функций,  
  процедур и пр.);
* стандартная библиотека, содержащая не менее двух функций;
* наличие одной управляющей конструкции языка;
* поддержка выражений с вызовом функций;
* наличие оператора вывода данных.

Контрольный пример должен отражать все конструкции языка. Тесты должны обеспечивать проверку вывода всех диагностических сообщений, генерируемых транслятором.

# 1 Спецификация языка программирования

## 1.1 Характеристика языка программирования

Язык программирования RAM-2022 компилируемый, высокоуровневый, императивный (поддерживается процедурное, структурное программирование), имеет статическую явную сильную типизацию.

## 1.2 Определение алфавита языка программирования

Алфавит языка RAM-2022 основан на кодировке Windows-1251. Набор символов времени трансляции, как и набор символов времени выполнения, включает в себя символы пробела, табуляции, перевода строки, а также все символы с кодами от 0x20 до 0x7E, от 0x80 до 0xFF кодировки Windows-1251. Остальные символы кодировки игнорируются.

## 1.3 Применяемые сепараторы

Для определения начала и конца лексем в исходном коде используются различные символы-сепараторы, которые представлены в таблице 1.1.

|  |  |
| --- | --- |
| Сепаратор | Назначение, правила применения |
| «пробел», «табуляция», «перевод строки» | Разделение лексем. Не допускается в имени идентификатора. |
| , | Разделение параметров функции. |
| ; | Разделение инструкций. Обязателен в конце инструкций. |
| { } | Начало и конец блока инструкций. Может быть пустым. |
| ( ) | Начало и конец условия цикла или условной конструкции; блока параметров функции или процедуры (может быть пустым); изменение приоритета выполнения операций. |
| +, -, \*, /, % | Арифметические операторы (бинарные, + и – могут быть также унарными). Для строк + обозначает конкатенацию. |
| >, <, ~, ! | Операторы сравнения: больше, меньше, равенство, неравенство. |
| = | Оператор присваивания. |
| " " | Начало и конец строкового литерала |

Таблица 1.1 Сепараторы и их назначение

## 1.4 Применяемые кодировки

Для написания исходного кода на языке RAM-2022 используется кодировка Windows-1251, представленная на рисунке 1.1.

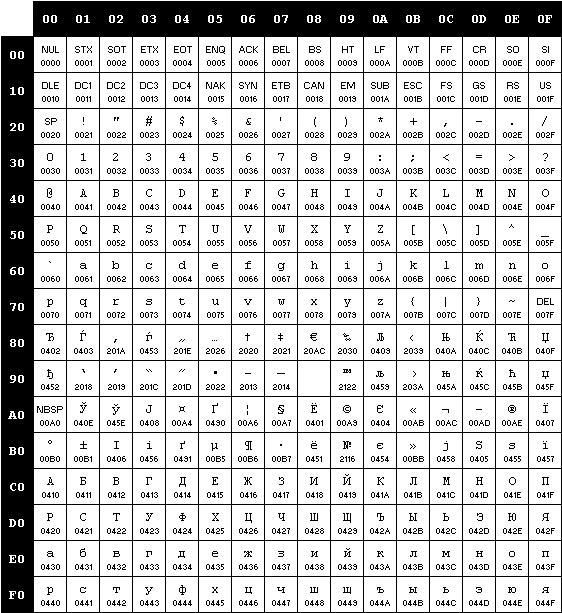


Рисунок 1.1 Кодировка Windows-1251.

## 1.5 Типы данных

В языке RAM-2022 поддерживаются следующие типы данных: знаковый целочисленный, строковый. Их описание представлено в таблице 1.2.

|  |  |
| --- | --- |
| Тип данных | Описание |
| int (знаковый целочисленный) | Размер: 4 байта. Диапазон значений: [-232, 232-1]  Значение по умолчанию: 0  Размещение в памяти: старший бит – знаковый; для отрицательных чисел используется дополнительный код.  Поддерживаемые операции: арифметические, сравнения, присваивание. |

Таблица 1.2 Типы данных и их описание

Продолжение таблицы 1.2

|  |  |
| --- | --- |
| Тип данных | Описание |
| str (строковый) | Размер: от 1 до 256 байт (по 1 байту на каждый символ строки и нуль-символ)  Максимальное количество символов: 255  Значение по умолчанию: пустая строка  Размещение в памяти: 1 байт на каждый символ, последний байт – нулевой (признак конца строки). Поддерживаемые операции: сравнения (посимвольное сравнение в соответствии с Windows-1251), конкатенация, присваивание. |

## 1.6 Преобразование типов данных

В языке RAM-2022 не поддерживаются явные или неявные преобразования типов данных. Для этой цели можно использовать функции стандартной библиотеки.

## 1.7 Идентификаторы

Идентификатор (в языке RAM-2022) – это последовательность символов, уникально идентифицирующая некоторую переменную, функцию (процедуру), параметр. Может включать в себя буквы английского алфавита и цифры, начинается с буквы. Максимальная допустимая длина: 50 символов. Если длина идентификатора больше, он усекается. Общее количество идентификаторов в программе не должно превышать 10000. Идентификатор не должен совпадать с ключевыми словами и другими идентификаторами в той же области видимости.

Описание идентификатора в РБНФ:

<идентификатор> ::= (A-Z | a-z){ (A-Z | a-z | 0-9) }

Примеры правильных идентификаторов: result, iVal2, lineX

Примеры неправильных идентификаторов: va\_lue, int

## 1.8 Литералы

Литерал – это последовательность символов в исходном коде, непосредственно представляющая собой некоторое фиксированное значение. Язык RAM-2022 поддерживает целочисленные и строковые литералы. Их описание представлено в таблице 1.3.

Таблица 1.3 Литералы и их описание

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Целочисленный | Последовательность цифр, которой может предшествовать знак, за которой может следовать символ ‘o’ – признак восьмеричной системы счисления. Для десятичной системы счисления признак не нужен. Допустимый диапазон значений: [-232, 232-1]  Примеры правильных литералов: 5, -89, +12o  Примеры неправильных литералов: 49h, 28o, -2.3 |

Продолжение таблицы 1.3

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Строковый | Последовательность символов алфавита длиной от 0 до 255, заключённая в двойные кавычки. Может содержать специальные символы: ‘\n’ (перевод строки), ‘\t’ (символ табуляции), ‘\"’ (двойная кавычка), ‘\\’ (обратная косая черта).  Примеры правильных литералов: "hi", "", "#5\n4$"  Примеры неправильных литералов: "h"i", 5"6, """ |

## 1.9 Объявление данных

При объявлении переменной указывается её тип, идентификатор, ставится сепаратор ‘;’ (конец инструкции). Перед сепаратором может быть операция присваивания и выражение.

Переменную можно объявить глобально либо в любом блоке инструкций, в том числе в блоке main, в теле цикла, условного оператора, функции (процедуры). Переменная, как и функция (процедура), доступна в той области, в которой объявлена, во всех вложенных (если не скрыта) и недоступна вне её.

## 1.10 Инициализация данных

По умолчанию переменные типа int имеют значение 0, типа str – "" (неявная инициализация).

Примеры инициализации: int a; (переменная инициализирована нулём),

str s = "abc"; (переменная инициализирована "", во время выполнения ей будет присвоено новое значение).

## 1.11 Инструкции языка

Инструкции языка RAM-2022 и их описание представлены в таблице 1.4.

Таблица 1.4 Инструкции языка

|  |  |
| --- | --- |
| Инструкция | Описание |
| Объявление переменной | <тип> <идентификатор>;  Пример: int b; |
| Объявление переменной с инициализацией времени выполнения | <тип> <идентификатор> = <выражение>;  Пример: int b = 5 + 3; |
| Объявление функции с определением | <тип> func <идентификатор> “(“ <список параметров> “)” “{“ <список инструкций> return (<переменная>|<литерал>) ; “}”  Список параметров содержит перечисление через запятую параметров в виде <тип параметра> <идентификатор>. Возвращаемое значение имеет тип, указанный перед “func”.  Пример: int func add3(int val){val = val + 3; return val; } |

Продолжение таблицы 1.4

|  |  |
| --- | --- |
| Инструкция | Описание |
| Объявление внешней функции | link <тип> func <идентификатор> “(“ <список параметров> “)”;  Пример: link int func add3(int val); |
| Объявление процедуры с определением | proc <идентификатор> “(“ <список параметров> “)” “{“ <список инструкций> “}”  Пример: proc printI(int val) { write val; } |
| Объявление внешней процедуры | link proc <идентификатор> “(“ <список параметров> “)”;  Пример: link proc printI(int val); |
| Вызов функции или процедуры | <идентификатор> “(“ <список фактических параметров> “)”  Пример: printI(4); |
| Присваивание значения переменной | <идентификатор переменной> = <выражение>;  Выражение может содержать литералы, переменные, вызовы функций.  Пример: iV = 1 \* 2; |
| Вывод значения | write (<идентификатор переменной>|<литерал>);  Пример: write "hi"; |
| Ввод значения | read <идентификатор>;  Пример: read count; |
| Подключение библиотеки .lib | link <путь к lib-файлу>;  Пример: link "std.lib"; |

## 1.12 Операции языка

Операции языка RAM-2022 и их описание представлены в таблице 1.5. Порядок выполнения операций можно изменить при помощи круглых скобок ().

Таблица 1.5 Операции языка

|  |  |
| --- | --- |
| Оператор | Описание |
| + (унарный) | Унарный плюс.  Приоритет: 4. |
| - (унарный) | Унарный минус.  Приоритет: 4. |
| + (бинарный) | Сложение двух целых чисел или конкатенация строк. Ассоциативный, коммутативный.  Приоритет: 2. |
| - (бинарный) | Разность двух целых чисел. Левоассоциативный.  Приоритет: 2. |
| \* | Произведение двух целых чисел. Ассоциативный, коммутативный, дистрибутивный относительно сложения.  Приоритет: 3. |

Продолжение таблицы 1.5

|  |  |
| --- | --- |
| Оператор | Описание |
| / | Целочисленное деление. Левоассоциативный.  Приоритет: 3. |
| % | Деление по модулю. Левоассоциативный.  Приоритет: 3. |
| > | Делает условие истинным, если целое число слева от оператора больше целого числа справа от него или строка слева “больше” строки справа (посимвольное сравнение), иначе ложным.  Приоритет: 1. |
| < | Делает условие истинным, если целое число слева от оператора меньше целого числа справа от него или строка слева “меньше” строки справа, иначе ложным.  Приоритет: 1. |
| = | Присваивание значения переменной. Правоассоциативный. |
| ~ | Делает условие истинным, если значение слева равно значению справа, иначе ложным.  Приоритет: 1. |
| ! | Делает условие истинным, если значение слева не равно значению справа, иначе ложным.  Приоритет: 1. |

## 1.13 Выражения и их вычисление

Выражение – вычисляемая последовательность идентификаторов, литералов, вызовов функций, арифметических операций.

Типы выражений: простые (идентификатор, литерал), сложные (содержат вызовы функций, унарные и (или) бинарные операторы).

Тип значения выражения и типы значения всех его составляющих (если функция, то тип возвращаемого значения) должны совпадать с типом значения переменной, которой оно присваивается. Для строковых выражений допустима только операция конкатенации. В выражениях вызов функции, возвращающей строковое значение, может быть только первым операндом. Выражения читаются слева направо. Используется инфиксная запись.

## 1.14 Конструкции языка

Конструкции языка RAM-2022 и их описание представлены в таблице 1.6.

Таблица 1.6 Конструкции языка

|  |  |
| --- | --- |
| Конструкция | Описание |
| Точка входа | main “{“ <список инструкций> “}”  Здесь начинается выполнение программы. Её наличие в глобальной области обязательно. |

|  |  |
| --- | --- |
| Конструкция | Описание |
| Функция | <тип> func <идентификатор> “(“ <список параметров> “)” “{“ <список инструкций> return (<переменная>|<литерал>); “}”  Список параметров содержит перечисление через запятую параметров в виде <тип параметра> <идентификатор>. Возвращаемое значение имеет тип, указанный перед “func”.  Список параметров содержит перечисление через запятую параметров в виде <тип параметра> <идентификатор>.  Параметры типа int передаются по значению, типа str – по ссылке. Функция обязана возвращать значение указанного в объявлении типа. |
| Процедура | proc <идентификатор> “(“ <список параметров> “)” “{“ <список инструкций> “}”  Параметры типа int передаются по значению, типа str – по ссылке. Не возвращает значение. |
| Конструкция цикла | while “(” <логическое выражение> “)” “{”<список инструкций>“}”  Пока истинно логическое выражение, выполняется список инструкций. Логическое выражение имеет вид  (<переменная>|<литерал>) <оператор сравнения> (<переменная>|<литерал>). |
| Условная конструкция | if “(” <логическое выражение> “)” “{” <список инструкций> “}” { elif “(” <логическое выражение> “)” “{” <список инструкций> “}” }[else “{” <список инструкций> “}” ]  Если истинно логическое выражение, то выполняется следующий за ним список инструкций, иначе аналогично выполняется проверка следующего условия (если есть блоки elif) или выполняются инструкции блока else (если он есть и не прошли проверку все elif). |

Продолжение таблицы 1.6

## 1.15 Область видимости идентификаторов

Функции и процедуры могут быть как глобальными, так и локальными. Объявление глобальных переменных запрещено. Идентификаторы видны в пределах той области, в которой объявлены (блок main, тело функции (процедуры), блоки условной конструкции, блок конструкции цикла). Параметры функции (процедуры) видны в пределах функции (процедуры). Не допускается объявление одинаковых идентификаторов в одной области. Идентификатор, объявленный во внутренней области, скрывает в ней такие же идентификаторы во внешних областях.

## 1.16 Семантические проверки

Семантические проверки:

* наличие и единственность блока main;
* использование идентификаторов только после их объявления;
* уникальность идентификаторов в пределах области их объявления;
* допустимость длины строкового литерала;
* допустимость значения численного литерала;
* допустимость длины идентификатора;
* соответствие передаваемых функции (процедуре) параметров указанным в объявлении;
* соответствие типа возвращаемого функцией значения объявленному;
* соответствие типов операндов в выражениях, условиях;
* порядковый номер среди операндов выражения вызова функции, возвращающей строковое значение;
* допустимость использования оператора в выражении;
* деление на ноль;
* отсутствие попыток вызова переменных;
* отсутствие попыток использования идентификаторов функций (процедур) в качестве переменных;
* вызов объявленных функций.

## 1.17 Распределение оперативной памяти на этапе выполнения

Используется плоская модель памяти. Под сегмент стека выделяется 4МБ. В сегмент констант помещаются литералы. В сегмент данных помещаются переменные (уникальность идентификаторов обеспечивается использованием префиксов областей видимости). В сегмент кода помещаются инструкции.

## 1.18 Стандартная библиотека и её состав

Для использования функций (процедур) стандартной библиотеки необходимо их предварительно объявить как внешние (link) в текущей области. Прототипы и описания функций (процедур) стандартной библиотеки представлены в таблице 1.7. Функции стандартной библиотеки написаны на C/C++.

Таблица 1.7 Состав стандартной библиотеки

|  |  |
| --- | --- |
| Прототип стандартной функции (процедуры) | Описание |
| char\* stdGetDate(char\* s) | Записывает в строку s текущую дату в формате ДД.ММ.ГГГГ. |
| char\* stdGetTime(char\* s) | Записывает в строку s текущее время в формате ЧЧ.ММ.СС. |
| int stdGetTimeCount() | Возвращает количество секунд, прошедших с 1 января 1970 года 00:00:00 UTC. |
| int stdStrToInt(const char\* s) | Конвертирует строку s в число и возвращает его; 0 в случае неудачи. |
| char\* stdIntToStr(char\*s,int val) | Записывает число val в строку s. |
| int stdPow(int x, int y) | Возвращает число x, введённое в степень y. |

Продолжение таблицы 1.7

|  |  |
| --- | --- |
| Прототип стандартной функции (процедуры) | Описание |
| int stdSqrt(int x) | Возвращает квадратный корень числа x; 0 в случае неудачи. |
| char\*stdSubstr(char\*dest, const char\* from, const int POS, const int COUNT) | Записывает в строку dest подстроку строки from: начиная с позиции POS COUNT символов; пустую строку в случае неудачи. |
| int stdStrLength(const char\* s) | Возвращает длину строки s. |

## 1.19 Ввод и вывод данных

Для ввода данных в переменную используется оператор read, после которого указывается её идентификатор. В случае некорректного ввода переменной присваивается значение по умолчанию.

read <идентификатор переменной>;

Для вывода данных используется оператор write, после которого идёт переменная или литерал, значение которого нужно вывести.

write (<переменная>|<литерал>);

## 1.20 Точка входа

Точкой входа программы является блок main. Выполнение программы начинается с первой инструкции блока и заканчивается выполнением его последней инструкции.

## 1.21 Препроцессор

Препроцессор в языке RAM-2022 не предусмотрен.

## 1.22 Соглашения о вызовах

В языке RAM-2022 применяется соглашение о вызовах cdecl, согласно которому параметры функций (процедур) передаются через стек справа налево, значение возвращается в регистр EAX, стек восстанавливает вызывающий код.

## 1.23 Объектный код

Целевой язык трансляции – MASM.

## 1.24 Классификация сообщений транслятора

Сообщения транслятора могут быть разделены на предупреждения и ошибки: предупреждения не останавливают трансляцию, ошибки – останавливают. Некоторые ошибки могут прервать трансляцию в месте возникновения, остальные прерывают по завершении текущего этапа либо, если их количество на одном из этапов превысило 15, тоже прерывают в месте возникновения последней ошибки.

Сообщения транслятора также могут быть классифицированы следующим образом:

* ошибки системы (диапазон кодов: 0 – 99);
* ошибки параметров компилятора (100 – 109);
* ошибки работы с файлами (110 – 119);
* лексические ошибки (200 – 299);
* синтаксические ошибки (300 – 399);
* семантические ошибки (400 – 499).

## 1.25 Контрольный пример

Исходный код программы на языке RAM-2022, демонстрирующей основные возможности языка RAM-2022, представлен в приложении А.

# 2 Структура транслятора

## 2.1 Компоненты транслятора, их назначение и принципы взаимодействия

Транслятор – программа, преобразующая исходный код на одном языке программирования в исходный код на другом языке программирования. Схема транслятора представлена на рисунке 2.1.

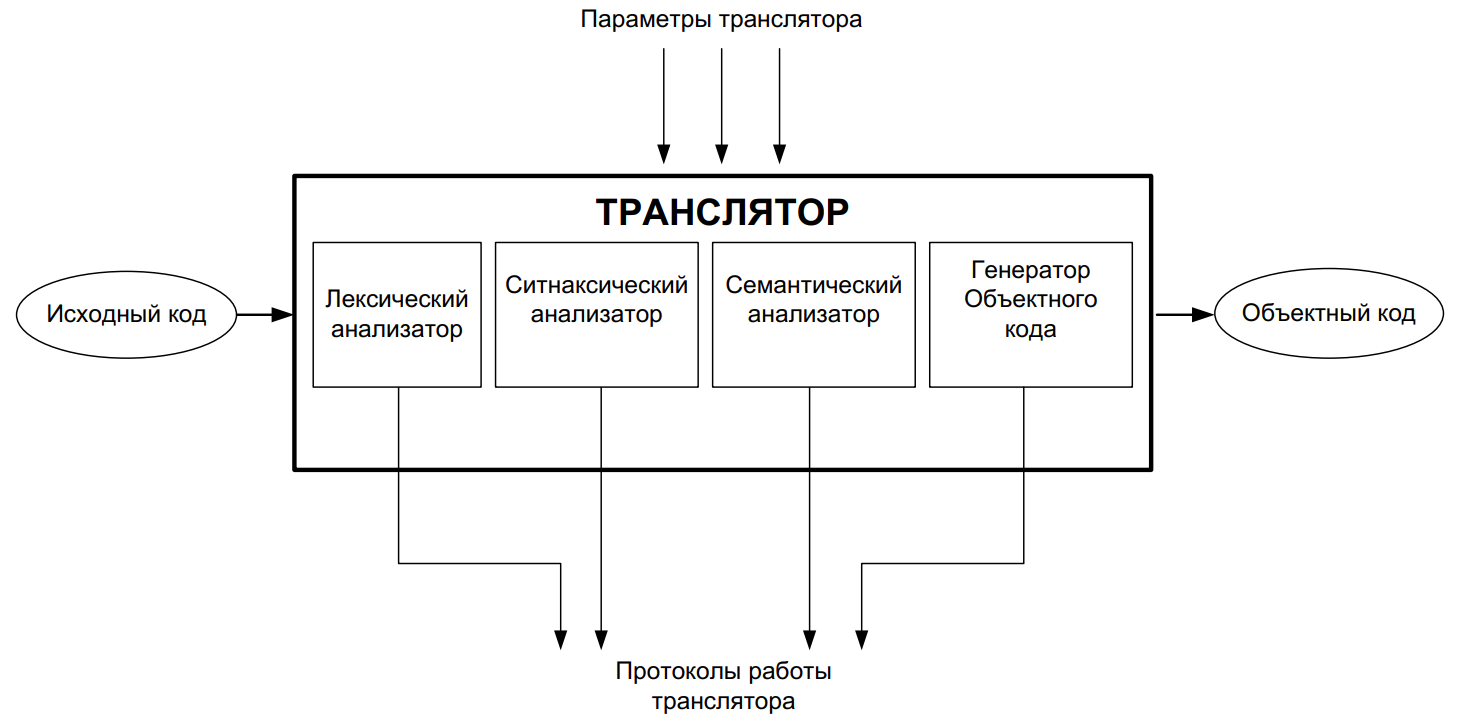


Рисунок 2.1 Схема транслятора

Транслятор последовательно осуществляет лексический, синтаксический, семантический анализ кода и генерирует объектный код.

Лексический анализатор разбирает исходный код на предлексемы, на основе которых формирует таблицы лексем и идентификаторов. Синтаксический анализатор на основе этих таблиц формирует дерево разбора. Семантический анализатор на основе таблиц лексем и идентификаторов проверяет смысловую правильность программных конструкций. Генератор объектного кода на основе этих таблиц выполняет генерацию кода на языке MASM. На каждом этапе трансляции дополняется протокол работы транслятора.

## 2.2 Перечень входных параметров транслятора

Описание входных параметров транслятора представлено в таблице 2.1.

Таблица 2.1 Входные параметры транслятора и их описание

|  |  |
| --- | --- |
| Параметр | Описание |
| -in | Путь к файлу с исходным кодом. Обязателен для указания. |
| -out | Путь к файлу, куда будет сгенерирован объектный кодке MASM.  Значение по умолчанию: <путь к входному файлу>.asm |
| -log | Путь к файлу, куда будет записан протокол работы транслятора.  Значение по умолчанию: <путь к входному файлу>.log |

## 2.3 Протоколы, формируемые транслятором

Описание протоколов, формируемых транслятором, представлено в таблице 2.2.

Таблица 2.2 Протоколы, формируемые транслятором, и их описание

|  |  |
| --- | --- |
| Протокол | Описание |
| Значение параметра -log | Содержит время создания протокола, общую характеристику исходного кода (количество символов, сколько из них проигнорировано, количество строк), таблицу предлексем, лексем и идентификаторов, ход работы синтаксического анализатора, дерево вывода, предупреждения и ошибки трансляции (если есть).  Если не удалось открыть файл протокола, то он будет выведен на консоль. |

# 3. Разработка лексического анализатора

## 3.1 Структура лексического анализатора

Лексический анализатор – программа, реализующая лексический анализ, то есть преобразующая исходный код, заменяя лексические единицы (предлексемы) лексемами, и создающая промежуточное представление исходного кода. Структурная схема лексического анализатора представлена на рисунке 3.1.

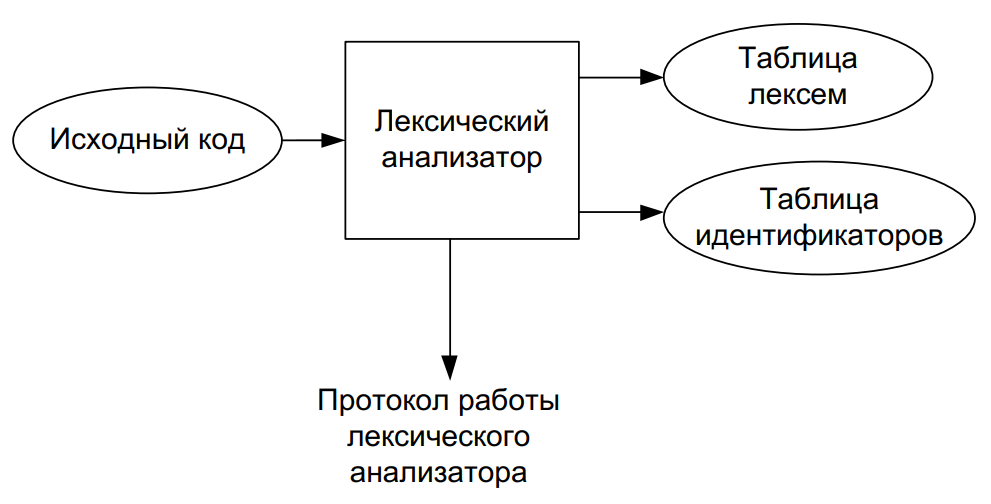


Рисунок 3.1 Структурная схема лексического анализатора.

Входные данные лексического анализатора: исходный код. Результаты работы лексического анализатора – таблицы лексем и идентификаторов с перекрёстными ссылками.

## 3.2 Контроль входных символов

Таблица разрешённых символов, используемая для контроля правильности написания исходного кода, представлена в приложении Б.

Типы символов определяются по обозначениям:

* T – разрешённые;
* I – игнорируемые;
* F – запрещённые;
* N – перевод строки;
* S – сепараторы;
* P – пробельные;
* Q – кавычки;
* C – комментарий.

В противном случае символ заменяется на указанный.

## 3.3 Удаление избыточных символов

Избыточными символами являются пробельные и перевод строки. Они служат разделителями лексем, но сами игнорируются на этапе разбиения кода на предлексемы.

Алгоритм удаления избыточных символов:

а) если достигнут конец исходного кода, завершить алгоритм;

б) прочесть следующий символ;

в) если символ избыточный и формируется предлексема, завершить её формирование;

д) перейти к следующему символу;

е) перейти к пункту а).

## 3.4 Перечень ключевых слов

Все ключевые слова языка RAM-2022, сепараторы, символы операций, соответствующие им лексемы и описание представлены в таблице 3.1.

Таблица 3.1 Ключевые слова

|  |  |  |
| --- | --- | --- |
| Ключевое слово | Лексема | Описание |
| int | t | Целочисленный тип. |
| str | t | Строковый тип. |
| link | x | Объявление внешней функции (процедуры); подключение библиотеки. |
| proc | p | Объявление процедуры. |
| func | f | Объявление функции. |
| return | r | Возврат функцией значения. |
| main | m | Главная конструкция. |
| write | w | Вывод данных на консоль. |
| read | g | Ввод данных в переменную. |
| if | ? | Начало условной конструкции. |
| else | : | Начало альтернативного блока условной конструкции. |
| elif | & | Начало условного альтернативного блока условной конструкции. |
| while | c | Начало конструкции цикла. |
| ; | ; | Конец инструкции. |
| , | , | Разделитель параметров. |
| = | = | Оператор присваивания. |
| { | { | Начало блока инструкций. |
| } | } | Конец блока инструкций. |
| ( | ( | Начало списка параметров; изменение приоритета арифметических операций; начало условия. |
| ) | ) | Конец списка параметров; изменение приоритета арифметических операций; конец условия. |
| + | a | Сложение двух чисел; конкатенация строк. |
| - | a | Разность двух чисел. |
| \* | a | Произведение двух чисел. |
| / | a | Деление одного числа на другое. |
| % | a | Остаток от деления одного числа на другое. |

Продолжение таблицы 3.1

|  |  |  |
| --- | --- | --- |
| Ключевое слово | Лексема | Описание |
| #+ | u | Унарный плюс. |
| #- | u | Унарный минус. |
| > | b | Оператор “больше” в условии. |
| < | b | Оператор “меньше” в условии. |
| ~ | b | Оператор равенства в условии. |
| ! | b | Оператор неравенства в условии. |
| = | = | Оператор присваивания. |

Ключевым словам, идентификаторам и литералам соответствуют регулярные выражения, при помощи которых лексический анализатор определяет тип лексемы для предлексем. Большинство регулярных выражений совпадают с ключевыми словами, которые они определяют (однозначное соответствие). Проверка соответствия предлексемы регулярному выражению реализована при помощи конечных автоматов. Пример графа переходов для предлексемы “if” и соответствующий фрагмент кода представлены рисунке 3.2.

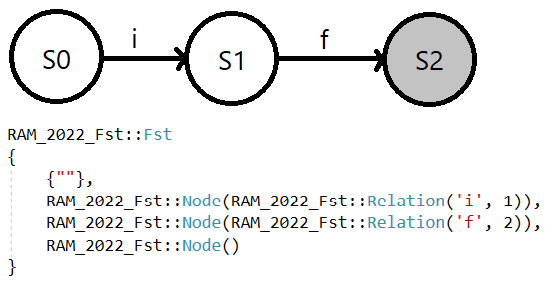


Рисунок 3.2 Граф переходов конечного автомата для ключевого слова “if” и его программная реализация.

Программная реализация структуры конечного автомата представлена в приложении В.

## 3.5 Основные структуры данных

Лексический анализатор представлен структурой Lexer, в которой хранятся таблицы предлексем (wordTable), лексем (lexTable), идентификаторов (idTable) и список функций с типами их параметров (funcsParamTypes). Реализация структуры показана на рисунке 3.3.

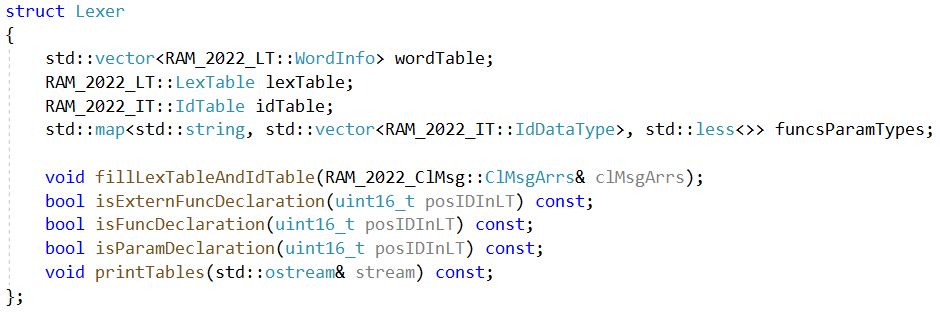


Рисунок 3.3 Структура лексического анализатора.

Таблица предлексем представлена вектором структур WordInfo, которые хранят предлексему (word) и её позицию (pos) в исходном коде. Реализация показана на рисунке 3.4.

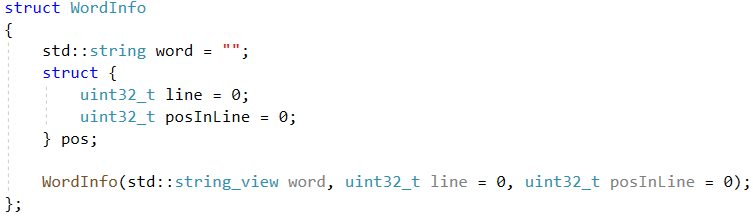


Рисунок 3.4 Структура записи о предлексеме.

Таблица лексем представлена структурой LexTable, у которой есть поля:

* size – текущий размер таблицы;
* maxSize – максимальный допустимый размер;
* table – вектор записей о лексемах (LexemeInfo).

Структура LexemeInfo представляет одну лексему, имеет поля:

* type – тип лексемы;
* pos – позиция в коде;
* pWORD – указатель на запись о предлексеме (WordInfo);
* pIdI – указатель на запись об идентификаторе (IdInfo), ненулевой у идентификаторов и литералов.

Реализация структур LexemeInfo и LexTable показана на рисунке 3.5.

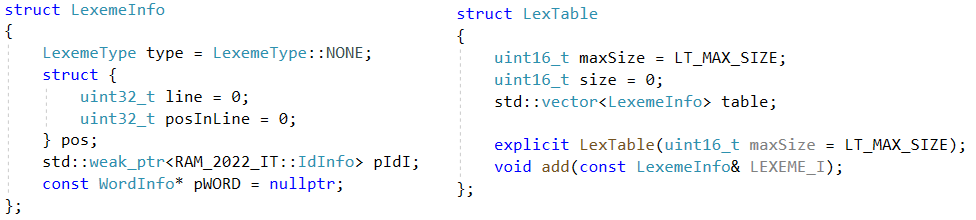


Рисунок 3.5 Структуры записи о лексеме и таблицы лексем.

Таблица идентификаторов представлена структурой IdTable, которая имеет следующие поля:

* size – размер таблицы, то есть количество идентификаторов в таблице;
* maxSize – максимальный допустимый размер;
* scope – указатель на глобальную область видимости (IdScope, описана далее);
* literals – вектор указателей на записи литералов (IdInfo);
* externs – вектор указателей на записи внешних функций и процедур (IdInfo).

Программная реализация структуры IdTable представлена на рисунке 3.6.

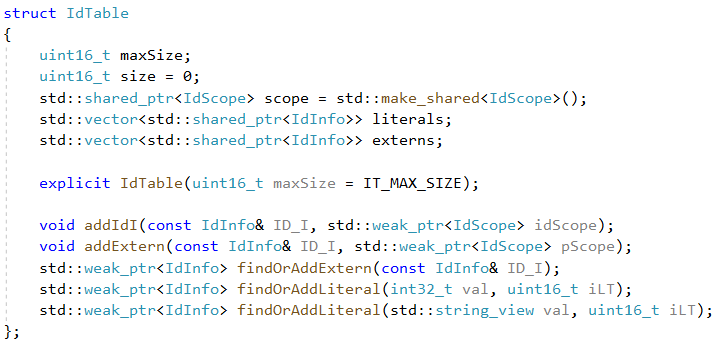


Рисунок 3.6 Структура таблицы идентификаторов.

Область видимости представлена структурой IdScope, она имеет следующие поля:

* scopeId – собственный номер-идентификатор;
* ids – вектор указателей на записи идентификаторов, объявленных в этой области;
* innerScopes – вектор указателей на вложенные области видимости;
* pOuterScope – указатель на внешнюю область видимости.

Реализация структуры IdScope представлена на рисунке 3.7.

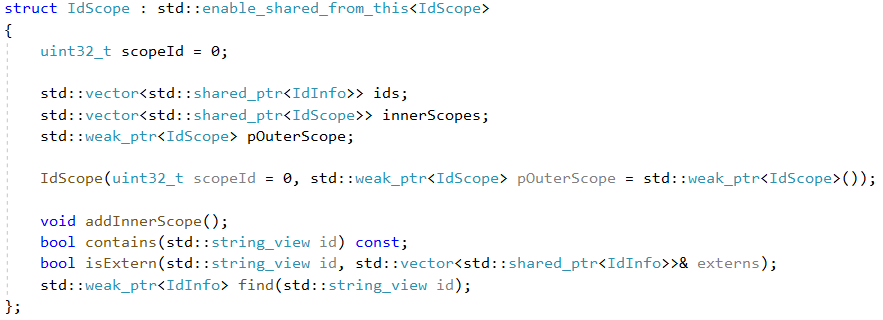


Рисунок 3.7 Структура области видимости.

Запись об идентификаторе (переменной, функции (процедуре), литерале) представлена структурой IdInfo, которая имеет следующие поля:

* iFirstLexLT – индекс первого появления идентификатора в таблице лексем;
* id – идентификатор;
* type – тип идентификатора (переменная, параметр, литерал или функция (процедура));
* dataType – тип значения идентификатора (int, str или пустое значение у процедур);
* value – значение идентификатора (есть у переменных (параметров) и литералов);
* pScope – указатель на область видимости, где объявлен идентификатор.

Реализация структуры IdInfo представлена на рисунке 3.8.

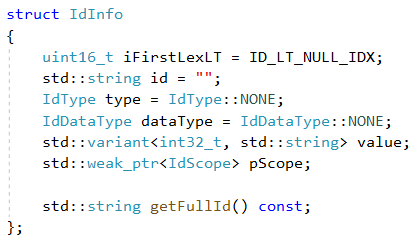


Рисунок 3.8 Структура записи об идентификаторе.

## 3.6 Структура и перечень сообщений лексического анализатора.

Лексический анализатор может выдавать сообщения типа ошибки или предупреждения. Сообщения об ошибках начинаются на “Ошибка:”, далее следует номер сообщения, текст сообщения, может следовать номер строки ошибки, может следовать номер позиции ошибки в строке. Сообщения о предупреждениях имеют такой же формат, за исключением того, что они начинаются не на “Ошибка:”, а на “Предупреждение:”. Если сообщение имеет номер строки, то эта строка выводится после сообщения.

Перечень сообщений лексического анализатора представлен в таблице 3.2.

Таблица 3.2 Сообщения лексического анализатора и предварительного лексического анализа

|  |  |  |
| --- | --- | --- |
| Номер сообщения | Тип | Сообщение |
| 2 | Ошибка | Ошибка выделения памяти для исходного кода |
| 2 | Ошибка | Ошибка выделения памяти для таблицы предлексем |
| 2 | Ошибка | Ошибка выделения памяти для таблицы лексем |

Продолжение таблицы 3.2

|  |  |  |
| --- | --- | --- |
| Номер сообщения | Тип | Сообщение |
| 3 | Ошибка | Превышен максимальный допустимый размер таблицы лексем |
| 4 | Ошибка | Превышен максимальный допустимый размер таблицы идентификаторов |
| 110 | Ошибка | Ошибка при открытии входного файла |
| 112 | Предупреждение | Превышен максимальный допустимый размер исходного кода |
| 200 | Ошибка | Нераспознанная лексема |
| 201 | Ошибка | Встречен недопустимый символ |
| 202 | Предупреждение | Встречен игнорируемый символ |
| 203 | Предупреждение | Строковый литерал содержит неизвестную управляющую последовательность |
| 204 | Ошибка | Строковый литерал не закрыт |
| 205 | Ошибка | Комментарий не закрыт |
| 401 | Ошибка | Повторное объявление идентификатора |
| 402 | Предупреждение | Целочисленное значение не входит в допустимый диапазон |
| 403 | Предупреждение | Превышена максимальная допустимая длина строкового литерала |
| 404 | Предупреждение | Превышена максимальная допустимая длина идентификатора |
| 405 | Ошибка | Неизвестный идентификатор |
| 406 | Ошибка | Не найдена конструкция main |
| 407 | Ошибка | Повторное объявление конструкции main |
| 408 | Ошибка | Конструкция main должна быть в глобальной области видимости |

## 3.7 Принцип обработки ошибок

Если не удалось открыть входной файл, или произошла ошибка выделения памяти, или был превышен максимальный допустимый размер таблицы лексем или идентификаторов, то работа лексического анализатора немедленно прекращается. В этом случае выбрасывается исключение, которое перехватывается в главной функции транслятора.

В остальных случаях ошибки и предупреждения накапливаются: на количество предупреждений нет ограничений, ошибок может храниться до 15 (при большем количестве ошибок лексический анализатор немедленно останавливает работу). После работы анализатора в протокол записываются ошибки, затем предупреждения. Сообщения могут сопровождаться соответствующими строками исходного кода. Если были ошибки, работа транслятора завершается.

## 3.8 Параметры лексического анализатора

Входными данными для лексического анализатора служит путь к файлу с исходным кодом, который задаётся в параметре -in транслятора.

## 3.9 Алгоритм лексического анализа

Общее словесное описание алгоритма работы лексического анализатора:

а) открыть файл с исходным кодом для чтения;

б) посимвольно считывая код, сформировать таблицу предлексем;

в) если рассмотрены все предлексемы, завершить алгоритм;

д) перейти к следующей нерассмотренной предлексеме;

е) определить тип лексемы для текущей предлексемы;

ж) если тип лексемы – идентификатор или литерал, то добавить запись в таблицу идентификаторов;

з) добавить запись в таблицу лексем;

и) перейти к пункту в).

## 3.10 Контрольный пример

Результаты работы лексического анализатора (таблицы лексем и идентификаторов) представлены в приложении Г.

# 4. Разработка синтаксического анализатора

## 4.1 Структура синтаксического анализатора

Синтаксический анализатор – программа, реализующая синтаксический анализ, то есть проверяющая последовательность лексем исходного кода на соответствие правилам заданной формальной грамматики. Структурная схема синтаксического анализатора представлена на рисунке 4.1.



Рисунок 4.1 Структурная схема лексического анализатора.

Входные данные синтаксического анализатора: таблицы лексем и идентификаторов. Результат работы – дерево разбора.

## 4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

Правила языка RAM-2022 описываются контекстно-свободной грамматикой типа 2 иерархии Хомского , где T – множество терминальных символов, N – множество нетерминальных символов, P – множество правил грамматики, S – начальный символ грамматики. Терминальные символы – символы лексем.

Грамматика приведена к нормальной форме Грейбах: она не леворекурсивная, правила P имеют вид A → aα, где a T, α (N ⋃ T)\*.

Алгоритм преобразования грамматики к нормальной форме Грейбах:

* удалить бесплодные символы;
* удалить недостижимые символы;
* удалить λ-правила;
* удалить цепные правила.

Перечень и описание нетерминальных символов и правил грамматики языка приведены в таблице 4.1.

Таблица 4.1 Грамматика языка RAM-2022

|  |  |  |
| --- | --- | --- |
| Нетерминал | Правила | Описание |
| S | xl;  xl;S  m{N}  m{N}S  tfiP{rI;}  tfiP{rI;}S  tfiP{NrI;}  tfiP{NrI;}S  piP{N}  piP{N}S  xtfiP;  xtfiP;S  xpiP;  xpiP;S | Общая структура программы: подключение библиотек, конструкция main, объявления глобальных функций (процедур). |
| I | i  l | Переменная или литерал. |
| P | ()  (F) | Параметры функции (процедуры) при определении. |
| F | ti  ti,F | Список объявлений параметров функции (процедуры). |
| C | ()  (W) | Параметры функции (процедуры) при вызове. |
| W | i  l  i,W  l,W | Список параметров функции (процедуры) при вызове. |
| E | i  iaE  l  laE  iC  iCaE  (E)  (E)aE  uE | Выражение. |
| A | &(IbI){N}  &(IbI){N}A | Условный альтернативный блок условной конструкции. |

Продолжение таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Нетерминал | Правила | Описание |
| N | ti;  ti;N  ti=E;  ti=E;N  i=E;  i=E;N  wI;  wI;N  gi;  gi;N  iC;  iC;N  tfiP{rI;}  tfiP{rI;}N  tfiP{NrI;}  tfiP{NrI;}N  piP{N}  piP{N}N  xtfiP;  xtfiP;N  xpiP;  xpiP;N  ?(IbI){N}  ?(IbI){N}N  ?(IbI){N}:{N}  ?(IbI){N}:{N}N  ?(IbI){N}A  ?(IbI){N}AN  ?(IbI){N}A:{N}  ?(IbI){N}A{N}N  c(IbI){N}  c(IbI){N}N | Инструкции и конструкции. |

## 4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку , где Q – множество состояний конечного автомата, V – алфавит входных символов, Z – специальный алфавит магазинных символов, δ – функция переходов автомата, q0 Q – начальное состояние автомата, z0 Z – начальное состояние магазина (маркер дна), F Q – множество конечных состояний.

Схема конечного автомата с магазинной памятью представлена на рисунке 4.2.

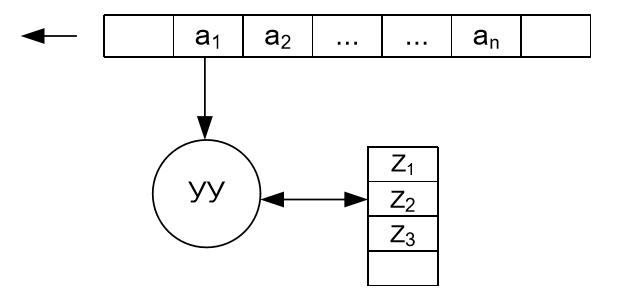


Рисунок 4.2 Схема конечного автомата с магазинной памятью

Алгоритм работы конечного автомата с магазинной памятью:

* состояние автомата (q, aα, zβ);
* прочесть символ а, находящийся под головкой (сдвиг ленты);
* не прочесть ничего (читается λ, не сдвигается лента);
* из δ определить новое состояние q', если (q', γ) δ(q, a, z) или (q', γ)  δ(q, λ, z);
* прочесть верхний символ стека z и записать цепочку γ, при этом, если γ = λ, то верхний символ магазина удалить;
* работа автомата заканчивается (q, λ, λ).

Далее приведена последовательность мгновенных стостояний автомата, демонстрирующая успешный разбор для цепочки ti=i()ai; , полученной из инструкции контрольного примера int diff = stdGetTimeCount() - start; :

* (q0, ti=i()ai; … , … N )
* (q0, ti=i()ai; … , … ;E=it )
* (q0, i=i()ai; … , … ;E=i )
* (q0, =i()ai; … , … ;E= )
* (q0, i()ai; … , … ;E )
* (q0, i()ai; … , … ;EaCi )
* (q0, ()ai; … , … ;EaC )
* (q0, ()ai; … , … ;Ea)( )
* (q0, )ai; … , … ;Ea) )
* (q0, ai; … , … ;Ea )
* (q0, i; … , … ;E )
* (q0, i; … , … ;i )
* (q0, ; … , … ; )
* (q0, … , … )

## 4.4 Основные структуры данных

Программный код на языке C++ основных структур данных, описывающих контекстно-свободную грамматику (цепочка (Chain), правило (Rule), грамматика Грейбах (Greibach)), представлен в приложении Д.

## 4.5 Описание алгоритма синтаксического разбора

Алгоритм синтаксического разбора:

а) в магазин автомата добавить стартовый символ;

б) на основе таблицы лексем сформировать входную ленту автомата;

в) найти цепочку, соответствующую нетерминальному символу, и добавить её в магазин в обратном порядке;

д) если терминалы на вершине стека и в начале ленты совпадают, то удалить их, иначе восстановить последнее сохранённое состояние автомата и найти следующую цепочку, соответствующую нетерминалу;

е) если в магазине встречается нетерминал, перейти к пункту д;

ж) если достигнуто дно стека и входная лента прочитана, то синтаксический анализ выполнен успешно, иначе записать в протокол ошибку.

Обобщённая блок-схема алгоритма синтаксического анализа представлена в приложении Е.

## 4.6 Структура и перечень сообщений синтаксического анализа

Перечень сообщений синтаксического анализатора представлен в таблице 4.2. Сообщения имеют такой же формат, как и сообщения лексического анализатора.

Таблица 4.2 Сообщения синтаксического анализатора

|  |  |  |
| --- | --- | --- |
| Номер сообщения | Тип | Сообщение |
| 300 | Ошибка | Неверная структура программы |
| 301 | Ошибка | Ошибочный оператор |
| 302 | Ошибка | Ошибка в выражении |
| 303 | Ошибка | Ошибка в параметрах функции (процедуры) |
| 304 | Ошибка | Ошибка в параметрах вызываемой функции (процедуры) |
| 305 | Ошибка | Ожидается идентификатор |
| 306 | Ошибка | Ошибка в альтернативном условии |

## 4.7 Параметры синтаксического анализатора и режимы его работы

Входными параметрами синтаксического анализатора служат таблицы лексем и идентификаторов. Протокол разбора и дерево разбора выводятся в файл протокола, определяющийся параметром -log программы.

## 4.8 Принцип обработки ошибок

Если не удалось разобрать входную ленту, то в протокол -log записываются три (или менее) самые глубокие ошибки разбора. Формат ошибок такой же, как и у сообщений лексического анализатора. В случае ошибки разбора работа транслятора на этом этапе завершается.

## 4.9 Контрольный пример

Фрагменты протокола синтаксического разбора конструкций языка RAM‑2022 из контрольного примера и полученное дерево разбора приведены в приложении Ж.

# 5 Разработка семантического анализатора

## 5.1 Структура семантического анализатора

Семантический анализатор проверяет смысловую правильность языковых конструкций. Его входными данными служат таблицы лексем и идентификаторов, дерево разбора. Его структурная схема представлена на рисунке 5.1. Некоторые семантические проверки выполняются лексическим анализатором на этапе лексического разбора, они приведены в перечне сообщений лексического анализатора.

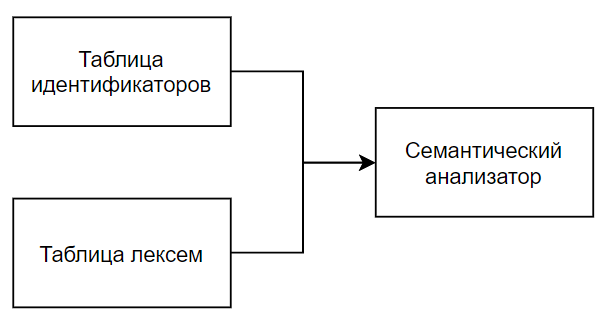


Рисунок 5.1 Структурная схема семантического анализатора.

## 5.2 Функции семантического анализатора

Семантические проверки и фазы их выполнения приведены в таблице 5.1.

Таблица 5.1 Семантические проверки

|  |  |
| --- | --- |
| Семантическая проверка | Фаза выполнения |
| Повторное объявление идентификатора | Лексический анализ |
| Значение целочисленного литерала вне допустимого диапазона | Лексический анализ |
| Превышена максимальная допустимая длина строки | Лексический анализ |
| Превышена максимальная допустимая длина идентификатора | Лексический анализ |
| Использование необъявленного идентификатора | Лексический анализ |
| Наличие конструкции main | Лексический анализ |
| Единственность конструкции main | Лексический анализ |
| Конструкция main в глобальной области видимости | Лексический анализ |
| Соответствие количества параметров при вызове функции (процедуры) объявленным | Семантический анализ |
| Соответствие типов параметров при вызове функции (процедуры) объявленным | Семантический анализ |
| Использование идентификатора функции как переменной | Семантический анализ |

Продолжение таблицы 5.1

|  |  |
| --- | --- |
| Семантическая проверка | Фаза выполнения |
| Попытка вызова переменной как функции | Семантический анализ |
| Использование объявленных функций | Семантический анализ |
| Соответствие типов операндов выражения типу переменной, которой оно присваивается. | Семантический анализ |
| Допустимость использования оператора в выражении. | Семантический анализ |
| Равенство типов левого и правого операнда в условии | Семантический анализ |
| Деление на ноль | Семантический анализ |
| Имя внешней библиотеки – строковый литерал | Семантический анализ |
| Соответствие типа возвращаемого функцией (процедурой) значения объявленному | Семантический анализ |
| Превышение максимально допустимого количества параметров функции в объявлении. | Семантический анализ |
| Функция, возвращающая строковое значение, является первым операндом выражения. | Семантический анализ |

Семантические проверки фазы лексического анализа выполняются в функции, заполняющей таблицу предлексем, и функции, заполняющей таблицы лексем и идентификаторов. Семантические проверки фазы семантического анализа выполняются функцией checkSemantics за два прохода по таблице лексем. В результате работы семантического анализатора также заполняется таблица параметров функций (поле funcsParamTypes структуры Lexer).

## 5.3 Структура и перечеь сообщений семантического анализатора

Перечень сообщений семантического анализатора представлен в таблице 5.2. Сообщения имеют такой же формат, как и сообщения лексического анализатора.

Таблица 5.2 Сообщения семантического анализатора.

|  |  |  |
| --- | --- | --- |
| Номер сообщения | Тип | Сообщение |
| 409 | Ошибка | Неверное количество параметров при вызове функции (процедуры) |
| 410 | Ошибка | Неверный тип фактического параметра при вызове функции (процедуры) |
| 411 | Ошибка | При вызове функции (процедуры) обязательны круглые скобки |
| 412 | Ошибка | Вызвать можно только функцию (процедуру) |

Продолжение таблицы 5.2

|  |  |  |
| --- | --- | --- |
| Номер сообщения | Тип | Сообщение |
| 413 | Предупреждение | Функция (процедура) не вызывается в коде |
| 414 | Ошибка | Типы операндов выражения должны совпадать с типом переменной, которой оно присваивается |
| 415 | Ошибка | Оператор недопустим в данном выражении |
| 416 | Ошибка | Обе части условия должны быть одного типа |
| 417 | Ошибка | Возвращаемый функцией тип не совпадает с указанным в объявлении |
| 418 | Ошибка | Деление на ноль |
| 419 | Ошибка | Превышено максимальное допустимое количество параметров функции (процедуры) |
| 420 | Ошибка | Путь к библиотеке должен быть строковым литералом |
| 421 | Ошибка | Вызов функции, возвращающей строковое значение, может быть только первым операндом выражения |

## 5.4 Принцип обработки ошибок

Принцип обработки ошибок и предупреждений семанического анализатора идентичен принципу обработки ошибок и предупреждений лексического анализатора, описанному в пункте 3.7.

## 5.5 Контрольный пример

Примеры семантических ошибок и их описание представлены в таблице 5.3.

Таблица 5.3 Примеры семантических ошибок

|  |  |
| --- | --- |
| Фрагмент кода и сообщение | Фаза разбора |
|  | Лексический анализ |
| Ошибка 407: Повторное объявление конструкции main, строка 2, позиция 1 |
|  | Семантический анализ |
| Ошибка 414: Типы операндов выражения должны совпадать с типом переменной, которой оно присваивается, строка 2, позиция 13 |
|  | Семантический анализ |
| Ошибка 415: Оператор недопустим в данном выражении, строка 2, позиция 17 |

# 6 Вычисление выражений

## 6.1 Выражения, допускаемые языком

В исходном коде в выражениях используется инфиксная запись, которая затем преобразуется в польскую инверсную. В выражениях могут использоваться целочисленный либо строковый типы данных в зависимости от того, куда помещается результат вычисления. Арифметические выражения могут содержать операторы + и – (бинарные и унарные), \*, /, %, (, ). Последовательности унарных операторов перед литералами вычисляются и применяются к литералам на этапе формирования предлексем. Строковые выражения могут содержать только оператор конкатенации +. В выражениях могут содержаться вызовы функций. Вызов функции, возвращающей строковое значение, может бытьь только первым операндом выражения. В условных выражениях допускается только однократное использование оператора сравнения (>, <, ~ или !), запрещены вызовы функций и другие операторы. Унарные операторы в численных литералах обрабатываются на этапе формирования таблицы предлексем. Приоритет и характеристики операторов были приведены ранее в пункте 1.12.

Примеры инструкций из контрольного примера, содержащих выражения:

* int diff = stdGetTimeCount() - start;
* str now = stdGetTime(now) + " ";
* while (choice ! 0)

## 6.2 Польская запись и принцип её построения

Выражения в операциях присваивания, условиях, вызовы функций (процедур) преобразовываются к инверсной польской записи, когда операторы располагаются непосредственно после своих операндов. Такие выражения проще для вычисления и генерации кода.

Алгоритм построения инверсной польской записи:

* исходная строка: выражение;
* результирующая строка: польская запись;
* стек (операторов): пустой;
* стек параметров: пустой;
* исходная строка просматривается слева направо;
* операнды переносятся в результирующую строку в порядке следования;
* операция записывается в стек, если стек пуст или на вершине стека лежит открывающая скобка;
* открывающая скобка помещается в стек;
* закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются;
* функция помещается в стек (максимальный приоритет), все последующие операнды помещаются в стек параметров, пока не встретится закрывающая скобка: тогда все операнды из стека параметров помещаются в результирующую строку;
* операция выталкивает все операции с большим или равным приоритетом в результирующую строку;
* в конце разбора все операции, оставшиеся в стеке, выталкиваются в результирующую строку.

Инверсия порядка параметров функции при помощи дополнительного стека упрощает последующую генерацию кода.

Преобразование выражения в ПОЛИЗ на примере инструкции now = stdGetDate(now) + "\n"; из контрольного примера: now = now stdGetDate "\n" +;

## 6.3 Программная реализация обработки выражений

Код на языке C++, реализующий преобразование выражений в ПОЛИЗ, приведён в приложении И. После выполнения преобразований обновлённая таблица предлексем, лексем и идентификаторов выводится в протокол -log.

## 6.4 Контрольный пример

Часть обновлённой таблицы предлексем, лексем и идентификаторов контрольного примера после преобразования выражений к ПОЛИЗ представлена в приложении К. Примеры этих преобразований представлены в таблице 6.1.

Таблица 6.1 Преобразование выражений к ПОЛИЗ.

|  |  |
| --- | --- |
| Выражение | ПОЛИЗ выражения |
| ti= i()ai ; | ti= iia ; |
| ti= i(i)al ; | ti= iila ; |
| c (ibl) | c ilb |

# 7 Генерация кода

## 7.1 Структура генератора кода

Целевой язык трансляции – MASM. Генератор кода принимает на вход таблицы предлексем, лексем, идентификаторов и таблицу сопоставления функциям списков типов их параметров. Генератор формирует код, последовательно проходя таблицу лексем, при необходимости получая информацию из других таблиц. Структура генератора кода представлена на рисунке 7.1.

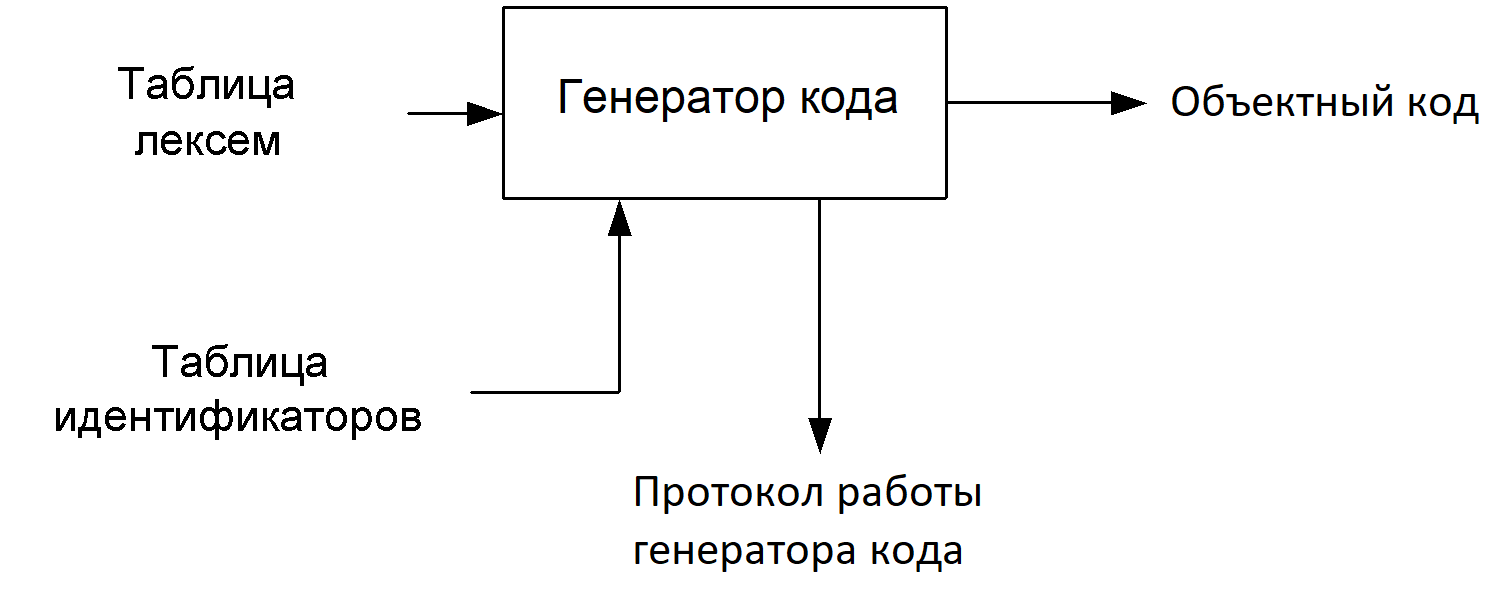


Рисунок 7.1 Структура генератора кода.

## 7.2 Представление типов данных в оперативной памяти

В объектном коде используется плоская модель памяти. Литералы заносятся в сегмент .CONST, переменные – в сегмент .DATA, инструкции – в сегмент .CODE. Соответствие типов языков RAM-2022 и MASM представлено в таблице 7.1.

Таблица 7.1 Соответствие типов языков RAM-2022 и MASM

|  |  |  |
| --- | --- | --- |
| Тип даных в RAM-2022 | Тип данных в MASM | Пояснение |
| int | SDWORD | Знаковый целочисленный тип, 4Б |
| str | DWORD | В MASM: указатель на начало строки, завершающейся нулевым символом. Указатель занимает 4Б, вся строка занимает 256Б. |

## 7.3 Статическая библиотека

Список и описание функций стандартной статической библиотеки указаны в пункте 1.18. Для использования функций (процедур) библиотеки необходимо в исходном коде её подключить при помощи инструкции link <путь к библиотеке>; , а используемые функции (процедуры) объявить как внешние. Если в их коде на C/C++ у параметра тип – char\*, то при объявлении нужно указать тип str, если int, то int. Если функция возвращает void, то она объявляется как процедура. Подключение статической библиотеки происходит на этапе генерации кода.

## 7.4 Особенности алгоритма генерации кода

Обобщённая блок-схема алгоритма генерации кода представлена на рисунке 7.2.

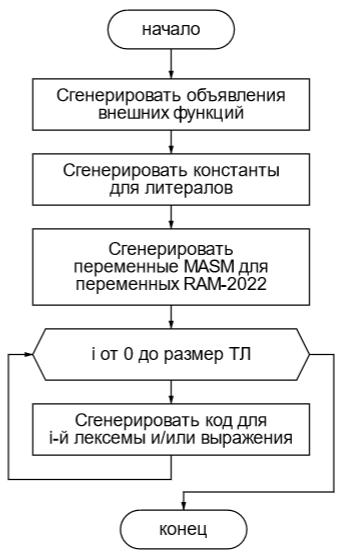


Рисунок 7.2 Обобщённая блок-схема алгоритма генерации кода.

Код генерируется для каждой лексемы, принимая во внимание её контекст среди других лексем. При генерации кода для функций генерируются их прототипы, которые указываются перед main, и определения, которые указываются после процедуры main.

В сгенерированном коде на MASM подключается статическая библиотека RAM\_2022\_ClLib.lib и могут вызываться её функции.

Целочисленные параметры функций принимаются и возвращаются по значению, строковые – по ссылке, однако при операциях присваивания и конкатенации происходит копирование строки с использованием буфера – имитируется возврат значения строки.

Для генерации кода для выражений реализована отдельная функция translateExpr, возвращающая значение через стек, которое затем обрабатывается.

Для вычисления смещения для восстановления стека после вызовов функций (процедур) при помощи call используется таблица, сопоставляющая функцию и список типов её параметров. Стек восстанавливает вызывающая функция.

Для некоторых операций со строками неявно используется буфер. Так как он один, некоторые действия со строками ограничены, например использование в выражениях функций, возвращающих строки (функция.

Если в ходе выполнения программы происходит попытка деления на ноль или переполнение целочисленного значения, на консоль выводится ошибка, позиция оператора, который её повлёк, и программа завершается.

## 7.5 Входные параметры, управляющие генерацией кода

Генератор кода использует таблицы лексем, идентификаторов, списки типов параметров функций, которые были сформированы на предыдущих этапах. Списки параметров типов позволяют восстанавливать код после вызовов функций.

## 7.6 Контрольный пример

Результат генерациии кода на языке MASM представлен в приложении Л.

# 8 Тестирование транслятора

## 8.1 Общие положения

Тесты транслятора призваны выявить ошибки его реализации. При обнаружении ошибок в исходном коде программы на RAM-2022 компилятор выводит их в протокол -log или на консоль, если не удалось открыть файл протокола, вместе с позицией в коде.

## 8.2 Результаты тестирования

Тестовые наборы (исходный код, соощение об ошибке, этап трансляции) представлены в таблице 8.1.

Таблица 8.1 Тестирование

|  |  |
| --- | --- |
| Этап лексического анализа | |
| Исходный код | Сообщение |
| main { 2a } | Ошибка 200: Нераспознанная лексема, строка 1, позиция 8 |
| main { str s = "\5"; } | Предупреждение 203: Строковый литерал содержит неизвестную управляющую последовательность, строка 1, позиция 17 |
| main { str s = "5; } | Ошибка 204: Строковый литерал не закрыт, строка 1, позиция 21 |
| main { str s = "5"; }@ | Ошибка 205: Комментарий не закрыт, строка 1, позиция 23 |
| main { int a; str a; } | Ошибка 401: Повторное объявление идентификатора, строка 1, позиция 19 |
| main { int a = 9999999999; } | Предупреждение 402: Целочисленное значение не входит в допустимый диапазон, строка 1, позиция 16 |
| main { str a = "1111111…  <256 единиц>  …1111111111"; } | Предупреждение 403: Превышена максимальная допустимая длина строкового литерала, строка 1, позиция 16 |
| main { str a0123456789…  <51 символ>  …0123456789; } | Предупреждение 404: Превышена максимальная допустимая длина идентификатора, строка 1, позиция 12 |
| @нет кода@ | Ошибка 406: Не найдена конструкция main |
| main{ int i; }  main{ int i; } | Ошибка 407: Повторное объявление конструкции main, строка 2, позиция 1 |
| proc f() { main{ int i; } } | Ошибка 408: Конструкция main должна быть в глобальной области видимости, строка 1, позиция 12 |
| Этап синтаксического анализа | |
| int a = 6;  main { str s = "5"; } | Ошибка 300: Неверная структура программы, строка 1, позиция 1 |

Продолжение таблицы 8.1

|  |  |
| --- | --- |
| Исходный код | Сообщение |
| main { str s < "5"; } | Ошибка 301: Ошибочный оператор, строка 1, позиция 8 |
| main { str s = "5"\*\*3; } | Ошибка 302: Ошибка в выражении, строка 1, позиция 20 |
| main { link proc s(int int i); } | Ошибка 303: Ошибка в параметрах функции (процедуры), строка 1, позиция 20 |
| main { link proc s(int i); s(2 3); } | Ошибка 304: Ошибка в параметрах вызываемой функции (процедуры), строка 1, позиция 30 |
| main { write ; } | Ошибка 305: Ожидается идентификатор, строка 1, позиция 14 |
| main { if (1~2) { write "!"; }  elif int(1~1) { int i; }  } | Ошибка 301: Ошибочный оператор, строка 2, позиция 8  Ошибка 306: Ошибка в альтернативном условии, строка 2, позиция 8 |
| Этап семантического анализа | |
| proc f() { int i; }  main { f(1); } | Ошибка 409: Неверное количество параметров при вызове функции (процедуры), строка 2, позиция 9 |
| proc f(str s) { write s; }  main { f(1); } | Ошибка 410: Неверный тип фактического параметра при вызове функции (процедуры), строка 2, позиция 9 |
| int func f(int i) { return i; }  main { f = 1; } | Ошибка 411: При вызове функции (процедуры) обязательны круглые скобки, строка 2, позиция 9 |
| main { int i; i(); } | Ошибка 412: Вызвать можно только функцию (процедуру), строка 1, позиция 16 |
| main { proc f() { int i; } } | Предупреждение 413: Функция (процедура) не вызывается в коде, строка 1, позиция 13 |
| main { int a = "1"; } | Ошибка 414: Типы операндов выражения должны совпадать с типом переменной, которой оно присваивается, строка 1, позиция 16 |
| main { str a = "1" - "2"; } | Ошибка 415: Оператор недопустим в данном выражении, строка 1, позиция 19 |
| main { if (1 ~ "1") { write "?"; } } | Ошибка 416: Обе части условия должны быть одного типа, строка 1, позиция 16 |
| main { int func f(){ return "1"; } f(); } | Ошибка 417: Возвращаемый функцией тип не совпадает с указанным в объявлении, строка 1, позиция 29 |
| main { int i = 1 / 0; } | Ошибка 418: Деление на ноль, строка 1, позиция 20 |
| proc f(int i1, int i2, <…>, int i10, int i11) { int i; }  main {f(1, 2, <…>, 10, 11);} | Ошибка 419: Превышено максимальное допустимое количество параметров функции (процедуры), строка 1, позиция 89 |
| link 1;  main { int i; } | Ошибка 420: Путь к библиотеке должен быть строковым литералом, строка 1, позиция 6 |

Продолжение таблицы 8.1

|  |  |
| --- | --- |
| Исходный код | Сообщение |
| str func f() { return "a"; }  main { str s = "b" + f(); } | Ошибка 421: Вызов функции, возвращающей строковое значение, может быть только первым операндом выражения, строка 2, позиция 22 |

# Заключение

В результате выполнения курсового проекта была составлена спецификация языка RAM-2022, разработан для него транслятор в язык MASM, проведено тестирование транслятора. Реализация транслятора поддерживает два типа данных (целочисленный и строковый), более 10 видов инструкций, условную конструкцию с возможным списком альтернативных условий, конструкцию цикла, функции, процедуры, 24 типа лексемы, 68 правил грамматики, 7 арифметических операторов (5 бинарных, 2 унарных), 4 оператора сравнения, есть стандартная библиотека с 10 функциями (в том числе тремя для работы с датой и временем). Код реализации содержит более 4000 строк. Время трансляции контрольного примера составило менее одной секунды. Контрольный пример отражает все конструкции языка. Было проведено тестирование транслятора, все тесты были успешно пройдены.

Таким образом, все задачи курсового проекта с учётом поставленных требований были выполнены, цель по приобретению навыков разработки системы программирования (транслятора) для собственного языка программирования была достигнута.

# Список использованных источников

1 Ахо, А. Компиляторы: принципы, технологии и инструментарий / А. Ахо, М. С. Лам, Р. Сети, Дж. Ульман. – M.: Вильямс, 2008. – 1184 с.

2 Вирт, Н. Построение компиляторов / Н. Вирт. – М.: ДМК Пресс, 2010. – 192 с.

3 Герберт, Ш. Полный справочник по C++ / Ш. Герберт. – 4‑е изд. – М.: Вильямс, 2006. – 800 с.

4 Ирвин, К. Р. Язык ассемблера для процессоров Intel / К. Р. Ирвин – М.: Вильямс, 2005. – 912 с.

5 Макконнелл, С. Совершенный код. Мастер-класс / С. Макконнелл. – М.: Русская редакция, 2010. – 896 с.

# Приложение А

Контрольный пример: демонстрация возможностей языка RAM-2022

Листинг А.1 Код контрольного примера

link "..\\Debug\\StdLib.lib";

link int func stdGetTimeCount();

int func getDiff(int start)

{

int diff = stdGetTimeCount() - start;

return diff;

}

proc printNow()

{

link str func stdGetDate(str s);

link str func stdGetTime(str s);

str now = stdGetTime(now) + " ";

write now;

now = stdGetDate(now) + "\n";

write now;

}

main

{

int start = stdGetTimeCount();

str now;

int choice = 1;

while (choice ! 0)

{

write "\nВыберите:\n";

write " 1 - вывести текущее время и дату\n";

write " 2 - вывести число прошедших со старта секунд\n";

write " другое - завершить выполнение\n";

read choice;

if (choice ~ 1)

{

printNow();

}

elif (choice ~ 2)

{

int diff = getDiff(start);

write "Прошло ";

write diff;

write " секунд\n";

}

else {

choice = 0;

}

}

}

@ Эта программа должна скомпилироваться... @

# Приложение Б

Таблица разрешённых символов

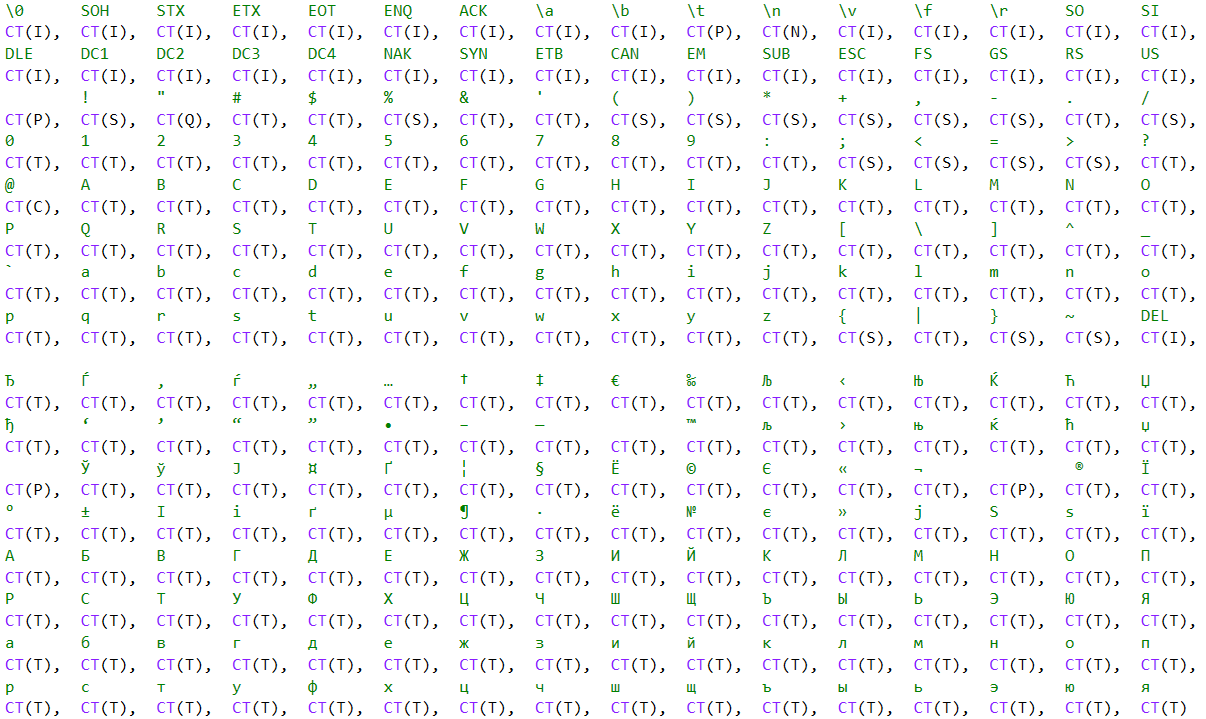


Рисунок Б.1 Таблица разрешённых символов

# Приложение В

Программная реализация структуры конечного автомата, разбирающего лексемы

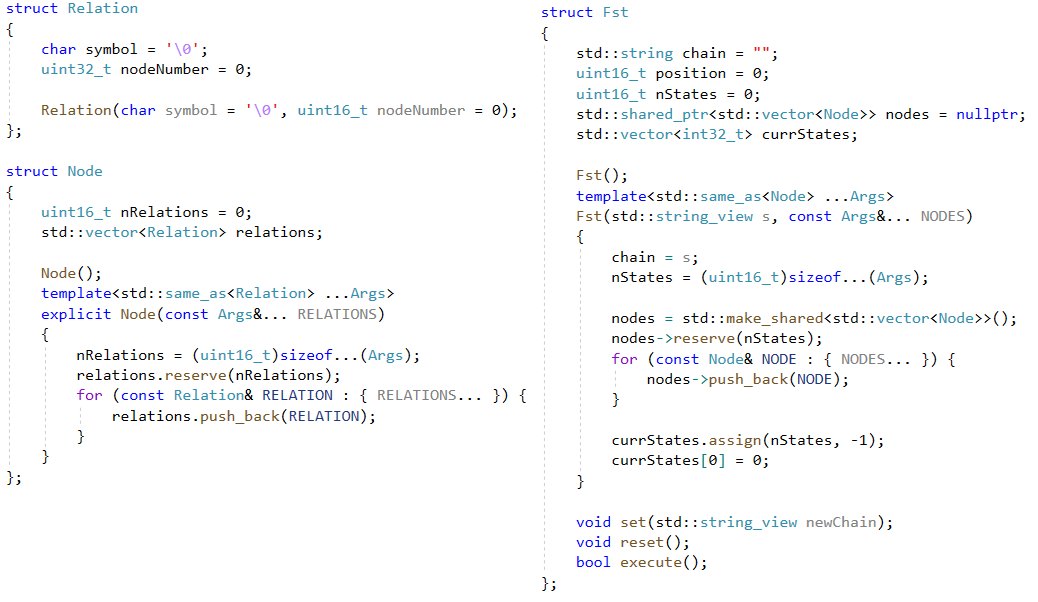


Рисунок В.1 Структура конечного автомата

# Приложение Г

Контрольный пример: таблица лексем и идентификаторов

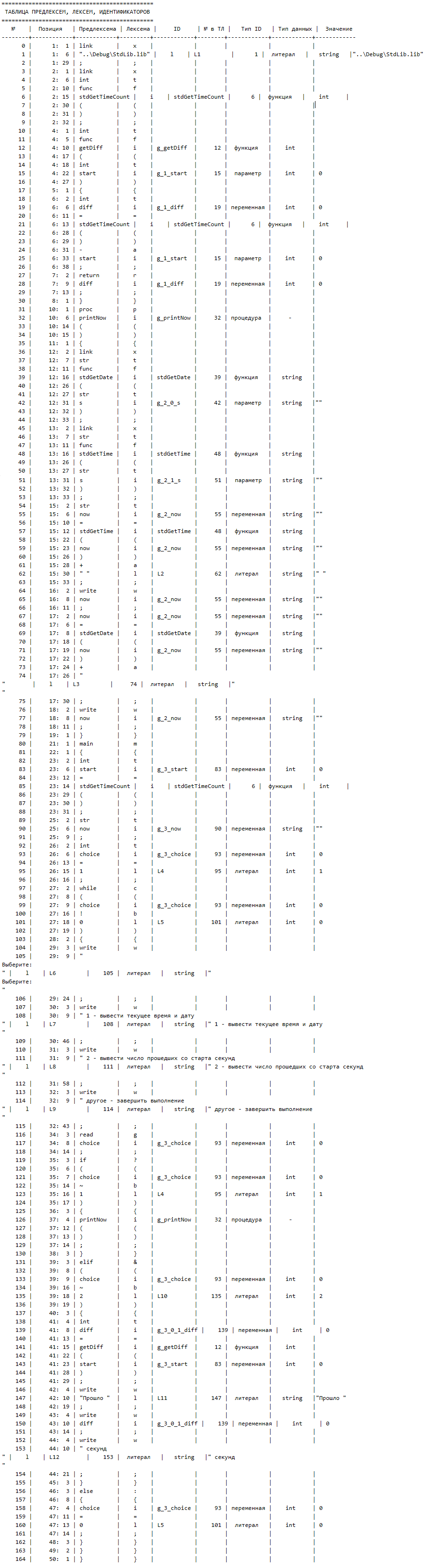


Рисунок Г.1 Таблица лексем и идентификаторов (лексемы с первой по шестидесятую)

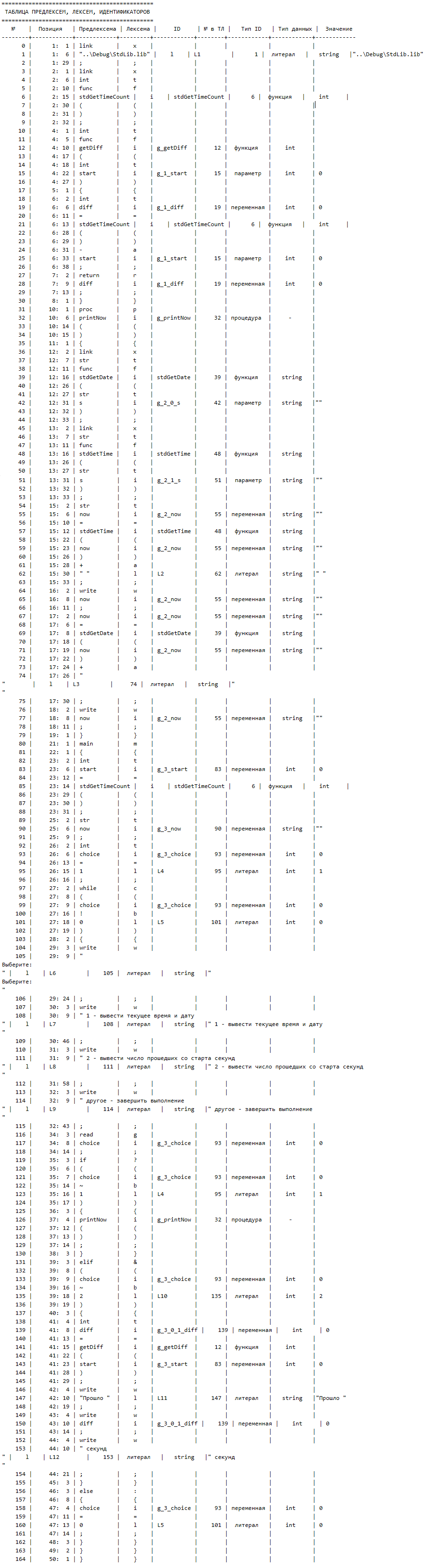


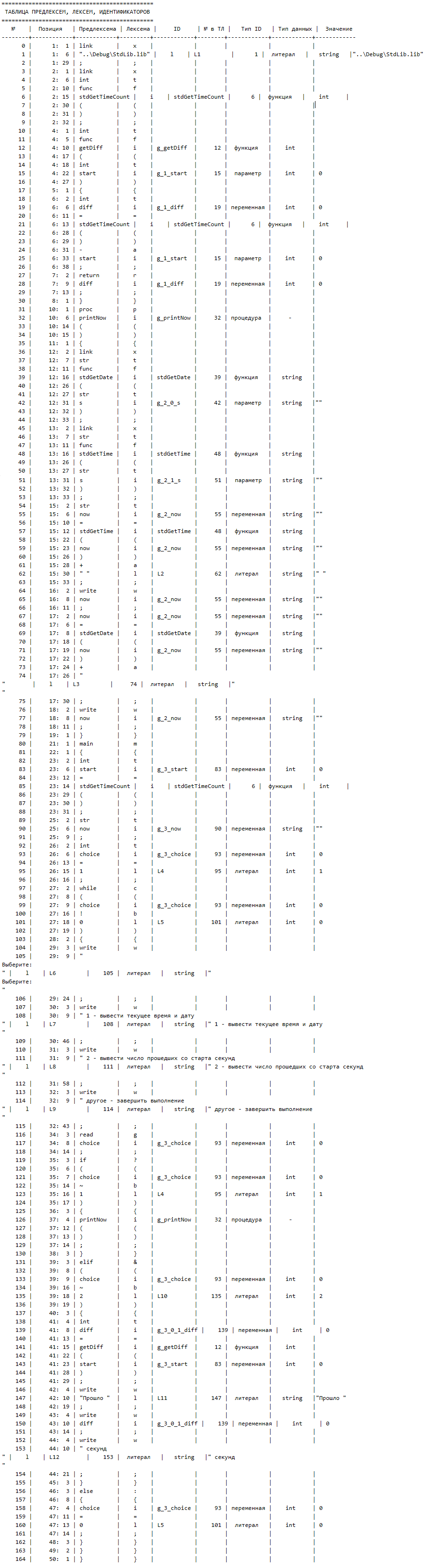
Рисунок Г.2 Таблица лексем и идентификаторов (лексемы с шестьдесят первой по сто десятую)

Рисунок Г.3 Таблица лексем и идентификаторов (лексемы со сто одиннадцатой по сто шестьдесят пятую)

# Приложение Д

Код основных структур данных, описывающих контекстно-свободную грамматику: цепочки (Chain), правила (Rule), грамматики Грейбах (Greibach))

Листинг Д.1 Код структур Chain, Rule, Greibach

using GrbAlphabet = int16\_t;

struct Rule

{

struct Chain

{

uint16\_t length = 0;

std::vector<GrbAlphabet> chain;

Chain();

template<std::same\_as<GrbAlphabet> ...Args>

explicit Chain(Args... symbols)

{

length = (uint16\_t)sizeof...(Args);

chain.reserve(length);

for (GrbAlphabet symbol : { symbols... }) {

chain.push\_back(symbol);

}

}

std::string getStrChain() const;

static constexpr GrbAlphabet T(char t) { return (GrbAlphabet)t; }

static constexpr GrbAlphabet N(char n) { return -(GrbAlphabet)n; }

static constexpr bool isT(GrbAlphabet s) { return s > 0; }

static constexpr bool isN(GrbAlphabet s) { return !isT(s); }

static constexpr char grbAlToChar(GrbAlphabet s) { return (char)(isT(s) ? s : -s); }

};

GrbAlphabet nonTerm = 0;

RAM\_2022\_ClMsg::ClMsgType errorType = RAM\_2022\_ClMsg::ClMsgType::INVALID\_ERROR\_ID;

uint16\_t nChains = 0;

std::vector<Chain> chains;

Rule();

template<std::same\_as<Chain> ...Args>

Rule(GrbAlphabet nonTerm, RAM\_2022\_ClMsg::ClMsgType errorType, const Args&... CHAINS)

{

this->nonTerm = nonTerm;

this->errorType = errorType;

nChains = (uint16\_t)sizeof...(Args);

chains.reserve(nChains);

for (const Chain& CHAIN : { CHAINS... }) {

chains.push\_back(CHAIN);

}

}

std::string getStrRule(uint16\_t chainNumber) const;

int32\_t getNextChain(GrbAlphabet t, Rule::Chain& chain, uint16\_t minChainNumber) const;

};

struct Greibach

{

GrbAlphabet startN = 0;

GrbAlphabet stackBottomT = 0;

uint16\_t nRules = 0;

std::vector<Rule> rules;

Greibach();

Продолжение листинга Д.1

template<std::same\_as<Rule> ...Args>

Greibach(GrbAlphabet startN, GrbAlphabet stackBottomT, const Args&... RULES)

{

this->startN = startN;

this->stackBottomT = stackBottomT;

nRules = (uint16\_t)sizeof...(Args);

rules.reserve(this->nRules);

for (const Rule& RULE : { RULES... }) {

rules.push\_back(RULE);

}

}

int32\_t getRule(GrbAlphabet nonTerm, Rule& rule) const;

Rule getRule(uint16\_t ruleNumber) const;

};

constexpr GrbAlphabet NS(char n) {

return Rule::Chain::N(n);

}

constexpr GrbAlphabet TS(char n) {

return Rule::Chain::T(n);

}

const Greibach GREIBACH(NS('S'), TS((char)RAM\_2022\_LT::LexemeType::END\_OF\_TAPE),

Rule(NS('S'), RAM\_2022\_ClMsg::ClMsgType::WRONG\_PROGRAM\_STRUCTURE,

Rule::Chain(TS('x'), TS('l'), TS(';')),

Rule::Chain(TS('x'), TS('l'), TS(';'), NS('S')),

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

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

Rule::Chain(TS('t'), TS('f'), TS('i'), NS('P'), TS('{'), TS('r'), NS('I'), TS(';'), TS('}')),

Rule::Chain(TS('t'), TS('f'), TS('i'), NS('P'), TS('{'), TS('r'), NS('I'), TS(';'), TS('}'), NS('S')),

Rule::Chain(TS('t'), TS('f'), TS('i'), NS('P'), TS('{'), NS('N'), TS('r'), NS('I'), TS(';'), TS('}')),

Rule::Chain(TS('t'), TS('f'), TS('i'), NS('P'), TS('{'), NS('N'), TS('r'), NS('I'), TS(';'), TS('}'), NS('S')),

Rule::Chain(TS('p'), TS('i'), NS('P'), TS('{'), NS('N'), TS('}')),

Rule::Chain(TS('p'), TS('i'), NS('P'), TS('{'), NS('N'), TS('}'), NS('S')),

Rule::Chain(TS('x'), TS('t'), TS('f'), TS('i'), NS('P'), TS(';')),

Rule::Chain(TS('x'), TS('t'), TS('f'), TS('i'), NS('P'), TS(';'), NS('S')),

Rule::Chain(TS('x'), TS('p'), TS('i'), NS('P'), TS(';')),

Rule::Chain(TS('x'), TS('p'), TS('i'), NS('P'), TS(';'), NS('S'))

),

Rule(NS('I'), RAM\_2022\_ClMsg::ClMsgType::WRONG\_ID,

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

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

),

Rule(NS('P'), RAM\_2022\_ClMsg::ClMsgType::WRONG\_PARAMETERS,

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

Rule::Chain(TS('('), NS('F'), TS(')'))

),

Rule(NS('F'), RAM\_2022\_ClMsg::ClMsgType::WRONG\_PARAMETERS,

Rule::Chain(TS('t'), TS('i')),

Rule::Chain(TS('t'), TS('i'), TS(','), NS('F'))

),

Rule(NS('C'), RAM\_2022\_ClMsg::ClMsgType::WRONG\_CALLED\_PARAMETERS,

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

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

),

Продолжение листинга Д.1

Rule(NS('W'), RAM\_2022\_ClMsg::ClMsgType::WRONG\_CALLED\_PARAMETERS,

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

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

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

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

),

Rule(NS('E'), RAM\_2022\_ClMsg::ClMsgType::WRONG\_EXPRESSION,

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

Rule::Chain(TS('i'), TS('a'), NS('E')),

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

Rule::Chain(TS('l'), TS('a'), NS('E')),

Rule::Chain(TS('i'), NS('C')),

Rule::Chain(TS('i'), NS('C'), TS('a'), NS('E')),

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

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

),

Rule(NS('A'), RAM\_2022\_ClMsg::ClMsgType::WRONG\_ELIF,

Rule::Chain(TS('&'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}')),

Rule::Chain(TS('&'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}'), NS('A'))

),

Rule(NS('N'), RAM\_2022\_ClMsg::ClMsgType::WRONG\_OPERATOR,

Rule::Chain(TS('t'), TS('i'), TS(';')),

Rule::Chain(TS('t'), TS('i'), TS(';'), NS('N')),

Rule::Chain(TS('t'), TS('i'), TS('='), NS('E'), TS(';')),

Rule::Chain(TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),

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

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

Rule::Chain(TS('w'), NS('I'), TS(';')),

Rule::Chain(TS('w'), NS('I'), TS(';'), NS('N')),

Rule::Chain(TS('g'), TS('i'), TS(';')),

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

Rule::Chain(TS('i'), NS('C'), TS(';')),

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

Rule::Chain(TS('t'), TS('f'), TS('i'), NS('P'), TS('{'), TS('r'), NS('I'), TS(';'), TS('}')),

Rule::Chain(TS('t'), TS('f'), TS('i'), NS('P'), TS('{'), TS('r'), NS('I'), TS(';'), TS('}'), NS('N')),

Rule::Chain(TS('t'), TS('f'), TS('i'), NS('P'), TS('{'), NS('N'), TS('r'), NS('I'), TS(';'), TS('}')),

Rule::Chain(TS('t'), TS('f'), TS('i'), NS('P'), TS('{'), NS('N'), TS('r'), NS('I'), TS(';'), TS('}'), NS('N')),

Rule::Chain(TS('p'), TS('i'), NS('P'), TS('{'), NS('N'), TS('}')),

Rule::Chain(TS('p'), TS('i'), NS('P'), TS('{'), NS('N'), TS('}'), NS('N')),

Rule::Chain(TS('?'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}')),

Rule::Chain(TS('?'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}'), NS('N')),

Rule::Chain(TS('?'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}'), TS(':'), TS('{'), NS('N'), TS('}')),

Rule::Chain(TS('?'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}'), TS(':'), TS('{'), NS('N'), TS('}'), NS('N')),

Rule::Chain(TS('?'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}'), NS('A')),

Rule::Chain(TS('?'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}'), NS('A'), NS('N')),

Rule::Chain(TS('?'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}'), NS('A'), TS(':'), TS('{'), NS('N'), TS('}')),

Rule::Chain(TS('?'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}'), NS('A'), TS(':'), TS('{'), NS('N'), TS('}'), NS('N')),

Rule::Chain(TS('c'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'),Продолжение листинга Д.1

TS('{'), NS('N'), TS('}')),

Rule::Chain(TS('c'), TS('('), NS('I'), TS('b'), NS('I'), TS(')'), TS('{'), NS('N'), TS('}'), NS('N')),

Rule::Chain(TS('x'), TS('t'), TS('f'), TS('i'), NS('P'), TS(';')),

Rule::Chain(TS('x'), TS('t'), TS('f'), TS('i'), NS('P'), TS(';'), NS('N')),

Rule::Chain(TS('x'), TS('p'), TS('i'), NS('P'), TS(';')),

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

));

# Приложение Е

Блок-схема алгоритма работы синтаксического анализатора

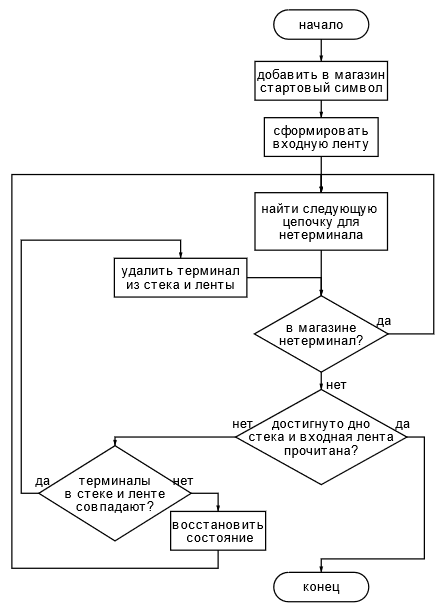


Рисунок Е.1 Блок-схема алгоритма работы синтаксического анализатора

# Приложение Ж

Фрагменты протокола синтаксического разбора контрольного примера и полученное дерево разбора

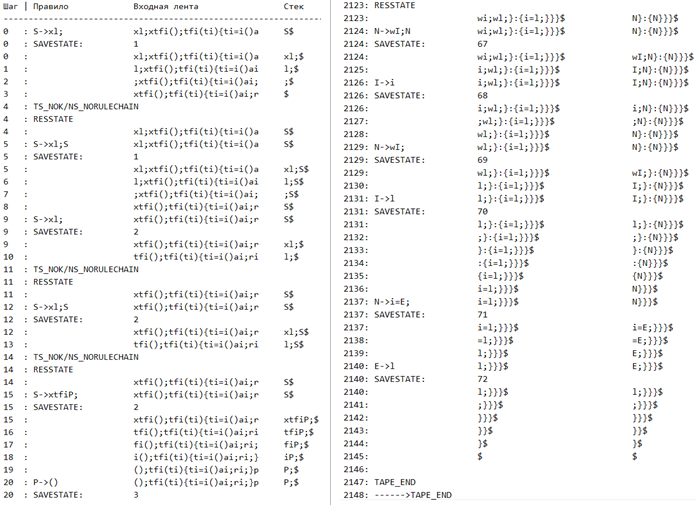


Рисунок Ж.1 Фрагменты протокола синтаксического рабора контрольного примера

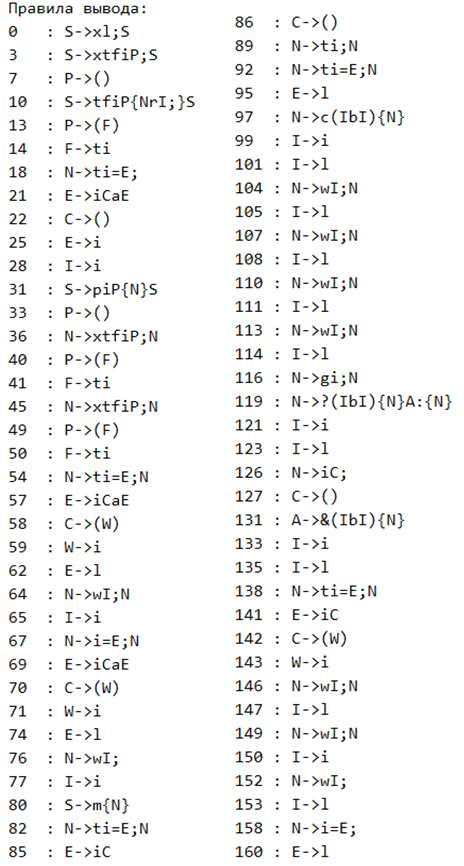


Рисунок Ж.2 Дерево разбора для контрольного примера в виде правил вывода

# Приложение И

Программная реализация преобразований выражений в ПОЛИЗ.

Листинг И.1 Реализация преобразований выражений в ПОЛИЗ

namespace

{

uint16\_t getPriority(const LexemeInfo& LI)

{

switch (LI.type)

{

using enum LexemeType;

case LEFT\_PARENTHESIS:

case RIGHT\_PARENTHESIS:

return 0;

case CMP\_BIN\_OP:

return 1;

case ARITHMETIC\_BIN\_OP:

{

if (LI.pWORD->word == "+" || LI.pWORD->word == "-") {

return 2;

}

else /\*if (word == "\*" || word == "/" || word == "%")\*/ {

return 3;

}

}

case ARITHMETIC\_UN\_OP:

return 4;

default:

return 5;

}

}

uint16\_t exprToRPN(uint16\_t exprPosInLT, Lexer& lexer)

{

queue<LexemeInfo> rpnExpr;

stack<LexemeInfo> operators;

stack<LexemeInfo> funcTemp;

uint16\_t nLexemes = 0;

bool exprEnded = false;

bool funcCall = false;

for (uint16\_t i = exprPosInLT; !exprEnded; i++)

{

if (funcCall) {

switch (lexer.lexTable.table[i].type)

{

using enum LexemeType;

case RIGHT\_PARENTHESIS:

{

while (!funcTemp.empty()) {

rpnExpr.push(funcTemp.top());

funcTemp.pop();

}

funcCall = false;

} break;

case ID:

case LITERAL:

{

funcTemp.push(lexer.lexTable.table[i]);

} break;

default:

break;

Продолжение листинга И.1

}

}

else

{

switch (lexer.lexTable.table[i].type)

{

using enum LexemeType;

case SEMICOLON:

case LEFT\_BRACE:

exprEnded = true;

continue;

case ID:

{

if (lexer.lexTable.table[i].pIdI.lock()->type == IdType::FUNC) {

operators.push(lexer.lexTable.table[i]);

funcCall = true;

}

else {

rpnExpr.push(lexer.lexTable.table[i]);

}

} break;

case LITERAL:

{

rpnExpr.push(lexer.lexTable.table[i]);

} break;

case LEFT\_PARENTHESIS:

{

operators.push(lexer.lexTable.table[i]);

} break;

case RIGHT\_PARENTHESIS:

{

while (!operators.empty() && operators.top().type != LEFT\_PARENTHESIS) {

rpnExpr.push(operators.top());

operators.pop();

}

if (operators.empty()) {

exprEnded = true;

break;

}

operators.pop();

} break;

case ARITHMETIC\_UN\_OP:

case ARITHMETIC\_BIN\_OP:

case CMP\_BIN\_OP:

{

uint16\_t thisPriority = getPriority(lexer.lexTable.table[i]);

while (!operators.empty() && thisPriority <= getPriority(operators.top())) {

rpnExpr.push(operators.top());

operators.pop();

}

operators.push(lexer.lexTable.table[i]);

} break;

default:

break;

}

}

Продолжение листинга И.1

nLexemes++;

}

while (!operators.empty()) {

rpnExpr.push(operators.top());

operators.pop();

}

for (uint16\_t i = 0; i < nLexemes; i++) {

if (rpnExpr.empty()) {

lexer.lexTable.table[exprPosInLT + i] = LexemeInfo();

continue;

}

switch (rpnExpr.front().type)

{

using enum LexemeType;

case ID:

case LITERAL:

{

if (exprPosInLT <= rpnExpr.front().pIdI.lock()->iFirstLexLT &&

rpnExpr.front().pIdI.lock()->iFirstLexLT < exprPosInLT + nLexemes) {

rpnExpr.front().pIdI.lock()->iFirstLexLT = exprPosInLT + i;

}

lexer.lexTable.table[exprPosInLT + i] = rpnExpr.front();

rpnExpr.pop();

} break;

default:

{

lexer.lexTable.table[exprPosInLT + i] = rpnExpr.front();

rpnExpr.pop();

} break;

}

}

return exprPosInLT + nLexemes;

}

}

void toRPN(Lexer& lexer)

{

for (uint16\_t i = 0; i < lexer.lexTable.size; ) {

if (lexer.lexTable.table[i].type == LexemeType::ASSIGN ||

lexer.lexTable.table[i].type == LexemeType::IF ||

lexer.lexTable.table[i].type == LexemeType::ELIF ||

lexer.lexTable.table[i].type == LexemeType::WHILE) {

i = exprToRPN(i + 1, lexer);

}

else if (lexer.lexTable.table[i].type == LexemeType::ID &&

lexer.lexTable.table[i].pIdI.lock()->type == IdType::FUNC &&

!lexer.isFuncDeclaration(i)) {

i = exprToRPN(i, lexer);

}

else {

i++;

}

}

}

# Приложение К

Фрагмент протокола преобразованной к ПОЛИЗ таблицы лексем и идентификаторов для контрольного примера

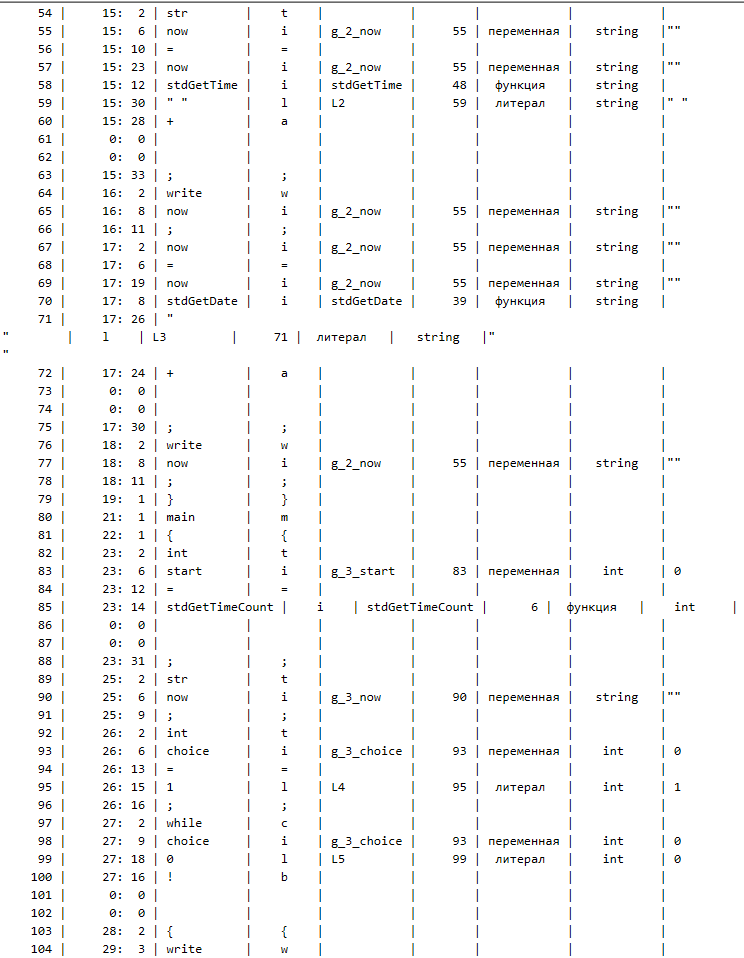


Рисунок К.1 Фрагмент протокола преобразованной к ПОЛИЗ таблицы лексем и идентификаторов для контрольного примера

# Приложение Л

Результат генерации кода контрольного примера

Листинг Л.1 Сгенерированный код на MASM для контрольного примера

.586P

.MODEL FLAT, C

INCLUDELIB kernel32.lib

INCLUDELIB libucrt.lib

ExitProcess PROTO STDCALL :DWORD

SetConsoleOutputCP PROTO STDCALL :DWORD

SetConsoleCP PROTO STDCALL :DWORD

atoi PROTO :DWORD

INCLUDELIB "..\Debug\RAM\_2022\_ClLib.lib"

\_ram2022\_strcat PROTO :DWORD, :DWORD, :DWORD

\_ram2022\_strcpy PROTO :DWORD, :DWORD

\_ram2022\_strcmp PROTO :DWORD, :DWORD

\_ram2022\_readStr PROTO :DWORD

\_ram2022\_writeInt PROTO :SDWORD

\_ram2022\_writeStr PROTO :DWORD

\_ram2022\_catchError PROTO :DWORD, :SDWORD, :SDWORD

INCLUDELIB "..\Debug\StdLib.lib"

EXTERN stdGetTimeCount :PROC

EXTERN stdGetDate :PROC

EXTERN stdGetTime :PROC

g\_getDiff PROTO :SDWORD

g\_printNow PROTO

.STACK 4096

.CONST

\_ram2022\_NEW\_LINE BYTE 10, 0

\_ram2022\_ERR\_LINE\_MSG BYTE ", строка ", 0

\_ram2022\_ERR\_POS\_MSG BYTE ", позиция ", 0

\_ram2022\_DIV\_BY\_NULL\_ERR\_MSG BYTE 10, "ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ: Деление на ноль", 0

\_ram2022\_INT\_OVERFLOW\_ERR\_MSG BYTE 10, "ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ: Переполнение значения типа int", 0

L2 BYTE " ", 0

L3 BYTE 10, 0

L4 SDWORD 1

L5 SDWORD 0

L6 BYTE 10, "Выберите:", 10, 0

L7 BYTE " 1 - вывести текущее время и дату", 10, 0

L8 BYTE " 2 - вывести число прошедших со старта секунд", 10, 0

L9 BYTE " другое - завершить выполнение", 10, 0

L10 SDWORD 2

L11 BYTE "Прошло ", 0

L12 BYTE " секунд", 10, 0

.DATA

\_ram2022\_buf BYTE 256 dup(0)

g\_1\_diff SDWORD 0

g\_2\_now BYTE 256 dup(0)

g\_3\_start SDWORD 0

g\_3\_now BYTE 256 dup(0)

g\_3\_choice SDWORD 0

g\_3\_0\_1\_diff SDWORD 0

.CODE

main PROC

Продолжение листинга Л.1

INVOKE SetConsoleOutputCP, 1251

INVOKE SetConsoleCP, 1251

call stdGetTimeCount

add ESP, 0

push EAX

pop EAX

mov g\_3\_start, EAX

mov g\_3\_now, 0

push OFFSET g\_3\_now

pop EAX

push L4

pop EAX

mov g\_3\_choice, EAX

.WHILE EAX == EAX

push g\_3\_choice

push L5

pop EBX

pop EAX

cmp EAX, EBX

jne cmp1\_t

push 0

jmp cmp1

cmp1\_t:

push 1

cmp1:

pop EAX

.IF EAX == 0

.BREAK

.ENDIF

INVOKE \_ram2022\_writeStr, OFFSET L6

INVOKE \_ram2022\_writeStr, OFFSET L7

INVOKE \_ram2022\_writeStr, OFFSET L8

INVOKE \_ram2022\_writeStr, OFFSET L9

INVOKE \_ram2022\_readStr, OFFSET \_ram2022\_buf

INVOKE atoi, OFFSET \_ram2022\_buf

mov g\_3\_choice, EAX

push g\_3\_choice

push L4

pop EBX

pop EAX

cmp EAX, EBX

je cmp2\_t

push 0

jmp cmp2

cmp2\_t:

push 1

cmp2:

pop EAX

.IF EAX == 1

call g\_printNow

add ESP, 0

push EAX

pop EAX

.ELSE

push g\_3\_choice

push L10

pop EBX

pop EAX

cmp EAX, EBX

je cmp3\_t

push 0

jmp cmp3

cmp3\_t:

Продолжение листинга Л.1

push 1

cmp3:

pop EAX

.IF EAX == 1

push g\_3\_start

call g\_getDiff

add ESP, 4

push EAX

pop EAX

mov g\_3\_0\_1\_diff, EAX

INVOKE \_ram2022\_writeStr, OFFSET L11

INVOKE \_ram2022\_writeInt, g\_3\_0\_1\_diff

INVOKE \_ram2022\_writeStr, OFFSET L12

.ELSE

push L5

pop EAX

mov g\_3\_choice, EAX

.ENDIF

.ENDIF

.ENDW

INVOKE ExitProcess, 0

main ENDP

\_ram2022\_catchError PROC,

\_ram2022\_str :DWORD,

\_ram2022\_line :SDWORD,

\_ram2022\_pos :SDWORD

INVOKE \_ram2022\_writeStr, \_ram2022\_str

INVOKE \_ram2022\_writeStr, OFFSET \_ram2022\_ERR\_LINE\_MSG

INVOKE \_ram2022\_writeInt, \_ram2022\_line

INVOKE \_ram2022\_writeStr, OFFSET \_ram2022\_ERR\_POS\_MSG

INVOKE \_ram2022\_writeInt, \_ram2022\_pos

INVOKE \_ram2022\_writeStr, OFFSET \_ram2022\_NEW\_LINE

INVOKE ExitProcess, -1

ret

\_ram2022\_catchError ENDP

g\_getDiff PROC,

g\_1\_start :SDWORD

call stdGetTimeCount

add ESP, 0

push EAX

push g\_1\_start

pop EBX

pop EAX

sub EAX, EBX

jno ov1

INVOKE \_ram2022\_catchError, OFFSET \_ram2022\_INT\_OVERFLOW\_ERR\_MSG, 6, 31

ov1:

push EAX

pop EAX

mov g\_1\_diff, EAX

mov EAX, g\_1\_diff

ret

g\_getDiff ENDP

g\_printNow PROC

mov \_ram2022\_buf[0], 0

push OFFSET g\_2\_now

call stdGetTime

add ESP, 4

push EAX

Продолжение листинга Л.1

push OFFSET L2

pop EBX

pop EAX

INVOKE \_ram2022\_strcat, OFFSET \_ram2022\_buf, EAX, EBX

push OFFSET \_ram2022\_buf

pop EAX

INVOKE \_ram2022\_strcpy, OFFSET g\_2\_now, EAX

mov \_ram2022\_buf[0], 0

INVOKE \_ram2022\_writeStr, OFFSET g\_2\_now

mov \_ram2022\_buf[0], 0

push OFFSET g\_2\_now

call stdGetDate

add ESP, 4

push EAX

push OFFSET L3

pop EBX

pop EAX

INVOKE \_ram2022\_strcat, OFFSET \_ram2022\_buf, EAX, EBX

push OFFSET \_ram2022\_buf

pop EAX

INVOKE \_ram2022\_strcpy, OFFSET g\_2\_now, EAX

mov \_ram2022\_buf[0], 0

INVOKE \_ram2022\_writeStr, OFFSET g\_2\_now

ret

g\_printNow ENDP

END main