МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ
ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий
Кафедра программной инженерии
Специальность 6-05-0612-01 Программная инженерия

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора IPP-2024»

Выполнил студент Иовчик Павел Петрович
(Ф.И.О.)

Руководитель проекта Волчек Дарья Ивановна

Заведующий кафедрой к.т.н., доц. Смелов В.В.

Консультант Волчек Дарья Ивановна

Нормоконтролер Волчек Дарья Ивановна

Курсовой проект защищен с оценкой

Минск 2024

Содержание

Введение

Целью выполнения курсового проекта по дисциплине «Конструирование программного обеспечения» является написание спецификации и разработка собственного языка программирования.

Название языка программирования, для которого разрабатывается компилятор, – IPP-2024. Компиляция будет производиться в язык ассемблера.

Исходя из ранее определённой цели курсового проекта, были определены следующие задачи:

– написание спецификации языка программирования;
– разработка структуры транслятора;
– разработка лексического анализатора;
– разработка синтаксического анализатора;
– разработка семантического анализатора;
– преобразование выражений;
– генерация кода;
– тестирование транслятора.

Информация о каждом этапе разработки компилятора приведена в соответствующих разделах пояснительной записки.

В первом разделе приведена спецификация языка, приведено точное формализованное описание набора правил, определяющих лексику, синтаксис и семантику языка.

Во втором разделе описана структура компилятора.

В третьем разделе описаны принцип работы и этапы разработки лексического анализатора, определены разрешенные символы и ключевые слова языка программирования.

В четвёртом разделе описан принцип работы синтаксического анализатора, определена формальная грамматика и приведена в нормальную форму Грейбах для выполнения синтаксического разбора.

В пятом разделе описаны выражения, допускаемые языком, форма, принципы построения и вычисления выражений.

В седьмом разделе описан процесс генерации кода.

В восьмом разделе приведены примеры тестирования транслятора.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования IPP-2024 – компилируемый, высокоуровневый, строго и статически типизированный, поддерживающий парадигмы процедурного, структурного и императивного программирования.

1.2 Определение алфавита языка программирования

Для языка программирования IPP-2024 разрешено использовать латинские буквы [A-Z, a-z], арабские цифры [0-9], а также специальные символы, такие как сепараторы и непечатные символы. Символы русского языка могут использоваться только в строковых литералах. Во время выполнения программы используются только те символы, которые заранее определены в исходном коде, включая символы, присутствующие в строковых литералах и выводимые в процессе работы программы.

1.3 Применяемые сепараторы

Сепараторы, применяемые в языке программирования IPP-2024, приведены в таблице 1.1.

Таблица 1.1 Применяемые сепараторы в языке программирования IPP-2024

Символ	Назначение
Пробел, табуляция, переход на новую строку	Используются для разделения лексем в исходном коде, обеспечивая корректное разделение и интерпретацию токенов компилятором.
( )	Ограничивают список параметров функций, задают приоритет выполнения операций в выражениях и используются для обозначения условий в конструкциях циклов и условных операторов.
,	Служит для разделения параметров функций или элементов массива при инициализации.
{ }	Определяют границы блоков кода, включая тело функций, циклов или условных операторов, а также используются для инициализации массивов.
[ ]	Определяют размерность массива или используются для доступа к элементу массива по индексу.
;	Обозначает завершение инструкции или выражения в коде.

1.4 Применяемые кодировки

Для написания исходного кода на языке программирования IPP-2024 используется кодировка Windows-1251, обеспечивающая поддержку символов кириллического алфавита и других символов, необходимых для корректного отображения текста.

1.5 Типы данных

В языке программирования IPP-2024 поддерживаются четыре типа данных: целочисленный, беззнаковый целочисленный, строковый и логический. Дополнительно реализована поддержка массивов для хранения элементов одного типа. Подробное описание типов данных приведено в таблице 1.2.

Таблица 1.2 Типы данных языка программирования IPP-2024

Тип данных	Описание
Целочисленный (int)	В памяти занимает 4 байта. Максимальное значение: 0x7FFFFFFF. Минимальное значение: 0x80000000. Принцип размещения в памяти: старший бит числа отведен под знак, оставшиеся 31 бит предназначены для хранения значения числа. Значение по умолчанию: 0. Возможные операции: - арифметические операции (+, -, *, /, %); - операции сравнения (>, <, >=, <=, ==, !=).
Беззнаковый целочисленный (uint)	В памяти занимает 4 байта. Максимальное значение: 0xFFFFFFFF. Минимальное значение: 0. Принцип размещения в памяти: все 32 бита предназначены для хранения значения числа. Значение по умолчанию: 0. Возможные операции: - арифметические операции (+, -, *, /, %); - операции сравнения (>, <, >=, <=, ==, !=).
Строковый (string)	В памяти занимает n + 1 байт, где n – количество символов в строке. Максимальное количество символов в строке: 254. Принцип размещения в памяти: каждый символ строки занимает 1 байт, в конце строки располагается нулевой символ (признак конца строки). Значение по умолчанию: пустая строка. Возможные операции: - не применяются

Окончание таблицы 1.2

Логический (bool)	В памяти занимает 4 байта. Может принимать одно из двух значений: true или false. Принцип размещения в памяти: значение младшего бита числа интерпретируется как true (если 1) или false (если 0). Возможные операции: - не применяются
Массив	В памяти занимает n * 4 байта, где n — количество элементов. Принцип размещения в памяти: элементы массива размещаются в памяти последовательно, каждый элемент занимает 4 байта. Значение по умолчанию: если массив не инициализируется явно при объявлении, его элементы заполняются значениями: − 0 – для массивов целочисленных и беззнаковых типов; − false – для массивов логического типа; − указатель на пустую строку - для массивов строк. Возможные операции (для массивов целочисленных и беззнаковых типов): - арифметические операции (+, -, *, /, %).

1.6 Преобразование типов данных

В языке программирования IPP-2024 предусмотрены неявные преобразования между знаковыми и беззнаковыми целочисленными типами. Преобразование происходит автоматически, если значение знакового типа неотрицательное и находится в диапазоне представления беззнакового типа.

1.7 Идентификаторы

Идентификаторы в языке программирования IPP-2024 представляют собой имена, используемые для обозначения переменных, функций, параметров функций и других объектов в программе, обеспечивая их уникальную идентификацию в пределах одной области видимости, при этом дублирование идентификаторов запрещено. Идентификаторы не могут совпадать с ключевыми словами языка, могут состоять только из латинских букв [A-Z, a-z], цифр [0-9] и знака нижнего подчеркивания (_), причём первый символ должен быть латинской буквой (цифры и знак подчеркивания в начале имени запрещены), а максимальная длина идентификатора ограничена 15 символами.

Правило записи идентификатора можно задать с помощью регулярного выражения: [a-zA-Z][a-zA-Z0-9_]*

Примеры корректных идентификаторов: idenf, idenf_123 и т. п.

Примеры некорректных идентификаторов: 1idenf, _stroka и т. п.

1.8 Литералы

Литерал в языке программирования – это константное значение, которое напрямую вписано в исходный код программы. В языке программирования IPP-2024 предусмотрены следующие типы литералов: целочисленный и строковый. Целочисленные литералы могут быть представлены в различных системах счисления: двоичной, восьмеричной, десятичной и шестнадцатеричной. Описание литералов приведено в таблице 1.3.

Таблица 1.3 Литералы языка программирования IPP-2024

Тип литерала	Характеристика
Целочисленный	Двоичный литерал: $[0\text{-}1]^{+}$b Восьмеричный литерал: $[0\text{-}7]^{+}$o Десятичный литерал: $[0\text{-}9]^{+}$ Шестнадцатеричный литерал: $[0\text{-}9A\text{-}F]^{+}$h Допустимый диапазон значений: От 0x80000000 до 0xFFFFFFFF (с учётом знака).
Строковый	Набор символов, состоящий из символов русского и латинского алфавитов, десятичных цифр и специальных символов, заключённых в двойные кавычки. Допустимый диапазон значений: От 0 до 254 символов.

Примеры правильных литералов: 1010b, 157o, 12345, 1A3Fh, «Hello, World!» и т. п.

Примеры неправильных литералов: 102b, 89o, 12a45, 1G3Fh, Example и т. п.

1.9 Объявление данных

Для объявления переменной используется ключевое слово new, после которого указывается тип данных переменной и имя идентификатора. Так же при объявлении допускается инициализация переменной. Правила объявления переменной:

- new <тип_данных> <имя_идентификатора>;
- new <тип_данных> <имя_идентификатора> = <значение>;
- new<тип_данных>array[<размер_массива>]<имя_идентификатора> = {<литерал>, …};

Переменные могут быть локальными или глобальными. Локальные переменные объявляются внутри функций, циклов или условных блоков, и их область видимости ограничивается блоком кода, заключенным в фигурные скобки.

Глобальные переменные объявляются вне всех функций и доступны из любого блока кода, начиная с момента их объявления и до конца программы.

1.10 Инициализация данных

В языке программирования IPP-2024 присутствует два вида инициализации для переменных и один для массивов:

− инициализация переменной в месте объявления:
new <тип_данных> <имя_идентификатора> = <значение>;
− инициализация переменной после объявления:
<имя_идентификатора> = <значение>;
− инициализация массива в месте объявления:
new <тип_данных><имя_массива> = {<литерал>, …};
Примеры:
− new int num = 5;
− new string text;
text = «Hello World!»;
− new int array[3] nums = {1, 2, 3};

Так же в языке программирования IPP-2024 присутствует инициализация по умолчанию. Если массив или переменная объявлены без явной инициализации, они автоматически инициализируются значениями по умолчанию: целочисленные и беззнаковые целочисленные типы – 0, строковый – пустой строкой, а логический – значением false.

1.11 Инструкции языка

Инструкции языка программирования IPP-2024 приведены в таблице 1.4.

Таблица 1.4 Инструкции языка программирования IPP-2024

Инструкция языка	Синтаксис
Объявление переменной	new <тип_данных> <идентификатор>;
Объявление переменной с явной инициализацией	new <тип_данных> <идентификатор> = <значение>; Значение – литерал, идентификатор, вызов функции соответствующего типа или выражение.
Объявление массива	new <тип_данных> array[<размер_массива>] <идентификатор>;
Объявление массива с явной инициализацией	new <тип_данных> array[<размер_массива>] <идентификатор> = {<литерал>, …};

Окончание таблицы 1.4

Инструкция языка	Синтаксис

Объявление функции	<тип_данных>function<идентификатор>(<тип_данных> <идентификатор>, …)
	{
	/*тело функции*/
	return <литерал>|<идентификатор>;
	}

Вызов функции	<идентификатор>(<литерал>|<идентификатор>, …);
Присвоение значения переменной	<идентификатор> = <значение>; Значение – литерал, идентификатор, вызов функции соответствующего типа или выражение.
Присвоение значения элементу массива	<идентификатор>[<индекс_элемента>] = <значение>; Значение – литерал, идентификатор, вызов функции соответствующего типа или выражение.
Вывод данных без переноса на новую строку	write <идентификатор>|<литерал>|<идентификатор> [индекс_элемента>];
Вывод данных с переносом на новую строку	writeline <идентификатор>|<литерал>|<идентификатор> [индекс_элемента>];
Возврат из функции	return <литерал>|<идентификатор>;

1.12 Операции языка

В языке программирования IPP-2024 доступны два типа операций: арифметические и логические. Эти операции применимы исключительно к целочисленным и беззнаковым целочисленным типам данных. Основные характеристики операций языка:

− операции с одинаковым приоритетом выполняются слева направо. Приоритетность можно изменять с помощью круглых скобок;

− выполнение операций над данными разных типов не допускается;

− все логические операции имеют одинаковый приоритет.

Подробное описание операций языка программирования IPP-2024 приведено в таблице 1.5.

Таблица 1.5 Операции языка программирования IPP-2024

Тип операции	Операция	Приоритетность операций	Описание
Арифметические	Сложение (+)	2	Бинарная, ассоциативная, коммутативная
	Вычитание (-)	2	Бинарная, некоммутативная, ассоциативная
	Умножение (*)	4	Бинарная, коммутативная, ассоциативная
	Деление (/)	4	Бинарная, некоммутативная, ассоциативная
	Остаток от деления (%)	4	Бинарная, некоммутативная, неассоциативная
Логические	Больше (>)	3	Бинарная, некоммутативная
	Меньше (<)	3	Бинарная, некоммутативная
	Проверка на равенство (==)	3	Бинарная, коммутативная
	Проверка на неравенство (!=)	3	Бинарная, коммутативная
	Больше или равно (>=)	3	Бинарная, некоммутативная
	Меньше или равно (<=)	3	Бинарная, некоммутативная

1.13 Выражения и их вычисление

Выражением называется совокупность переменных, констант, знаков операций, имен функций, скобок, которая может быть вычислена в соответствии с синтаксисом языка программирования.

Вычисление выражений в языке программирования IPP-2024 осуществляется по следующим правилам:

- операции выполняются в соответствии с их приоритетом, операции с одинаковым приоритетом выполняются слева направо;
- выражения записываются в одну строку;
- допускается использовать круглые скобки для смены приоритета;
- выражение может содержать вызов функции;
- использование двух операторов подряд не допускается;

− в одном выражении могут участвовать только операнды одного и того же типа данных;

− составные выражения с логическими операциями не допускаются.

В арифметических выражениях и выражениях сравнения допускаются только операнды целочисленного и целочисленного беззнакового типов.

Перед генерацией кода выражения приводятся к ПОЛИЗ для более удобного вычисления на языке ассемблера.

1.14 Конструкции языка

Конструкции языка программирования IPP-2024 приведены в таблице 1.6.

Таблица 1.6 Конструкции языка программирования IPP-2024

Конструкция	Описание

Главная функция	main
	{
	…
	}

Пользовательская функция	<тип_данных> function <идентификатор>(<тип_данных> <идентификатор>, …)
	{
	/*тело функции*/
	return <литерал>|<идентификатор>;
	}

Цикл	while(<условие>)
	{
	…
	}

Условная конструкция	if(<условие>)
	{
	…
	}
	else
	{
	…
	}

1.15 Область видимости идентификатора

В языке программирования IPP-2024 идентификаторы могут иметь локальную или глобальную область видимости:

- локальная область видимости: идентификаторы видимы только внутри конструкции, в которой они объявлены, с последовательным доступом сверху вниз.
- глобальная область видимости: идентификаторы, объявленные на уровне программы, доступны из любой точки кода.

Параметры функции имеют локальную область видимости и доступны только внутри этой функции. Создание пользовательских областей видимости не поддерживается.

1.16 Семантические проверки

Семантическим анализатором языка программирования IPP-2024 предусмотрены следующие проверки:

– наличие блока main, точки входа в программу;

– единственная точка входа в программу;

– использование идентификаторов до их объявления;

– переопределение идентификаторов;

– соответствие параметров, передаваемых в функцию, с параметрами в объявлении функции;

– соответствие типа возвращаемого значения с типом функции;

– соответствие типов в выражениях;

– превышение размера целочисленных и строковых литералов;

– превышение длины лексемы;

– соответствие операторов типам данных, для работы с которыми они предназначены;

– корректность типов в выражениях условных конструкций (if, while);

– отсутствие параметров у функций DATE и TIME, возвращающих строковый тип;

– проверка индексов массивов на выход за пределы допустимого диапазона;

– совпадение количества и типов элементов инициализатора массива с размером массива;

– недопустимость присваивания значений, не соответствующих типу переменной или массива;

– корректность количества и типов аргументов при вызове функций;

– наличие оператора return в функциях, которые должны возвращать значение;

– соответствие типов данных в операторе return типу возвращаемого значения функции;

– деление на ноль;

– проверка на переполнение при вычислении выражений;

– проверка размера и значения целочисленных литералов на соответствие допустимому диапазону;

– недопустимость повторного объявления функций, параметров, переменных и массивов в одной области видимости.

1.17 Распределение оперативной памяти на этапе выполнения

В языке программирования IPP-2024 для хранения промежуточных результатов в вычислении выражения используется стек. В сегмент констант

записываются все литералы языка. В сегмент данных записываются все имена переменных.

1.18 Стандартная библиотека и ее состав

В стандартной библиотеке языка программирования IPP-2024 содержатся функции, представленные в таблице 1.7.

Таблица 1.7 Стандартная библиотека языка программирования IPP-2024

Функция	Описание	Количество параметров
string DATE()	Возвращает строку с текущей датой в формате ДД.ММ.ГГ.	0
string TIME()	Возвращает строку с текущим временем в формате ЧЧ:ММ:СС.	0

Стандартная библиотека написана на языке C++, подключается автоматически на этапе компоновки. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций.

1.19 Ввод и вывод данных

В языке программирования IPP-2024 ввод данных не поддерживается.

Вывод данных на консоль осуществляется за счет операторов write и writeline. Использование данных операторов допускается только с идентификаторами или литералами. Функции, управляющие выводом данных на консоль, реализованы на языке ассемблера.

1.20 Точка входа

Точкой входа в программе на языке программирования IPP-2024 является функция main. Точка входа не может отсутствовать или быть переопределена.

1.21 Препроцессор

Препроцессор – программа для обработки текста. Может быть отдельной программой, или интегрированной в компилятор.

В языке программирования IPP-2024 препроцессор не предусмотрен.

1.22 Соглашение о вызовах

В языке программирования IPP-2024 используется соглашение о вызовах stdcall. Особенности stdcall:

– все параметры функции передаются через стек;
– освобождением памяти занимается вызываемый код;
– параметры заносятся в стек справа налево.

1.23 Объектный код

Язык программирования IPP-2024 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке программирования IPP-2024 и выявления её транслятором в текущий файл протокола выводится сообщение. Их классификация сообщений приведена в таблице 1.8.

Таблица 1.8 Описание ошибок транслятора языка программирования IPP-2024

Диапазон ошибок	Описание
0-99	Системные ошибки
100-109	Ошибки задания входных параметров
110-119	Ошибки чтения и открытия файлов
120-199	Ошибки лексического анализа
200-299	Ошибки синтаксического анализа
300-399	Ошибки семантического анализа
400-999	Зарезервированные ошибки

1.25 Контрольный пример

Контрольный пример, написанный на языке IPP-2024, представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора их назначение и принципы взаимодействия

Транслятор – это программа, преобразующая исходный код на одном языке программирования в исходный код на другом языке программирования.

Схема, поясняющая принцип работы транслятора, изображена на рисунке 2.1.

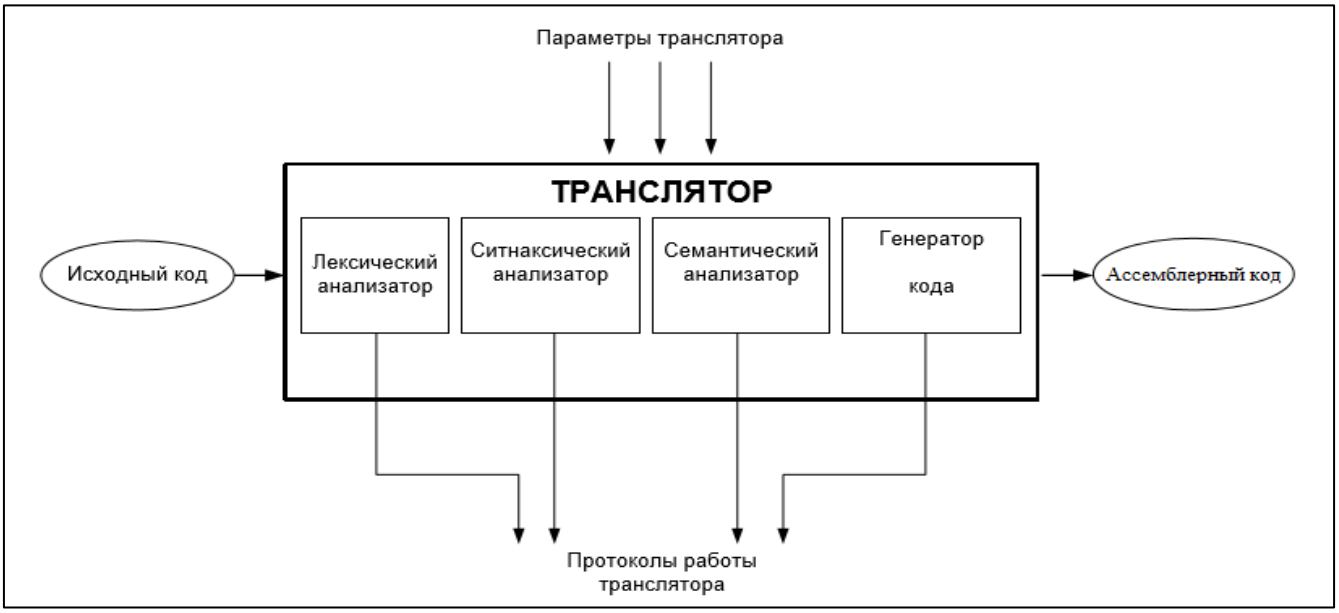

Рисунок 2.1 – Структура транслятора языка программирования IPP-2024

Трансляция исходного кода в язык ассемблера разделена на четыре этапа:

– лексический анализ;

– синтаксический анализ;

– семантический анализ;

– генерация кода.

Этапы выполняются последовательно. У каждого этапа есть входные и выходные данные, которые последовательно передаются следующему компоненту транслятора.

Первой частью трансляции является лексический анализ. На вход лексического анализатора подается исходный код программы. В свою очередь лексический анализатор производит деление исходного кода программы на токены, которые затем идентифицируются и заменяются на лексемы. На выходе лексического анализатора мы имеем две таблицы: таблицу лексем и таблицу идентификаторов.

Синтаксический анализ является второй частью работы транслятора. Синтаксический анализатор выполняет синтаксический анализ. Входными данными для синтаксического анализатора являются таблица лексем и таблица идентификаторов. Выходные данные – дерево разбора.

Затем выполняется семантический анализ. Задача семантического анализатора: проверка соблюдения в исходной программе семантических правил

входного языка программирования. На входе семантический анализатор получает таблицу идентификаторов, таблицу лексем и дерево разбора.

Последним этапом трансляции является генерация кода. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с исходным кодом на языке ассемблера.

2.2 Перечень входных параметров транслятора

Входные параметры необходимы для формирования файлов с результатами работы транслятора. Входные параметры, которые можно передать транслятору языка программирования IPP-2024, представлены в таблице 2.1.

Таблица 2.1 Входные параметры транслятора языка программирования IPP-2024

Ключ и входной параметр	Описание параметра	Значение по умолчанию
-in:<путь к файлу>	Текстовый файл с исходным кодом на языке программирования IPP- 2024	in.txt
-log:<путь к файлу>	Файл с протоколом работы транслятора	log.log
-out:<путь к файлу>	Выходной файл – результат работы транслятора. Содержит исходный код на языке ассемблера.	out.asm

2.3 Протоколы, формируемые транслятором

В ходе работы программы формируются протоколы работы лексического, синтаксического и семантического анализаторов, содержащие информацию о ходе их работы. Все файлы создаются в корневом каталоге. В таблице 2.2 приведены протоколы, формируемые транслятором и их содержимое.

Таблица 2.2 Протоколы, формируемые транслятором языка программирования IPP-2024

Формируемый протокол	Описание выходного протокола
Файл, заданный параметром «-log:»	Содержит общую информацию о ходе выполнения трансляции: перечисление входных параметров, количество символов и строк, успех или ошибку по каждому этапу трансляции. В случае возникновения ошибки, в файл будет выведена информация об ошибке.

Окончание таблицы 2.2

Формируемый протокол	Описание выходного протокола
	Также файл содержит дерево разбора, сформированное во время синтаксического анализа.
Выходной файл с названием «LT.txt»	Файл содержит таблицу лексем, сформированную во время лексического анализа.
Выходной файл с названием «IT.txt»	Файл содержит таблицу идентификаторов, сформированную во время лексического анализа.
Файл, заданный параметром «-out:»	Результат работы программы – файл, содержащий исходный код на языке ассемблера.

3. Разработка лексического анализатора

3.1 Структура лексического анализатора

Лексический анализатор – это программа, преобразующая исходный текст программы, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Структурная схема лексического анализатора изображена на рисунке 3.1.

Рисунок 3.1 Структурная схема лексического анализатора

Лексический анализ в языке программирования IPP-2024 происходит в два этапа:

– разбиение исходного кода программы на токены;

– идентификация токенов и последующая их замена на лексемы. Заполнение таблиц лексем и идентификаторов.

Входные данные: исходный код на языке программирования IPP-2024.

Результат работы: таблица лексем и таблица идентификаторов.

3.2 Контроль входных символов

Исходный код на языке программирования IPP-2024 прежде, чем транслироваться проверяется на допустимость символов. То есть изначально из входного файла считывается по одному символу и проверяется, является ли он разрешённым. Таблица для проверки входных символов представлена на рисунке 3.2.

```
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, '|', IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T, IN::T,\
```

Рисунок 3.2 Таблица контроля входных символов

Индекс элемента соответствует его коду в таблице символов Window-1251. T – символ разрешён, F – символ запрещён, I – символ игнорируется. Если вместо символа указан другой символ, то при обработке будет происходить его замена на указанный.

3.3 Удаление избыточных символов

Избыточный символ – это символ, отсутствие которого никоем образом не влияет на исходный текст программы. В языке программирования IPP-2024 символы пробела и табуляции являются избыточными. Они игнорируются при считывании из файла исходного кода на языке программирования IPP-2024.

Алгоритм удаления избыточных символов:

Пока есть символ для чтения:

– читаем очередной символ;

– если символ является пробелом или табуляцией:

– если пробел или табуляция находятся между буквами или цифрами, сохраняем их как разделитель между словами;

– если символ не окружён значащими символами (буквами или цифрами), пропускаем его.

3.4 Перечень ключевых слов

Ключевые слова языка программирования IPP-2024, сепараторы, символы операций и соответствующие им лексемы приведены в таблице 3.1.

Таблица 3.1 Ключевые слова, сепараторы, символы операций и соответствующие им лексемы языка программирования IPP-2024

Токен	Лексема	Описание
int	t	Целочисленный тип
uint	t	Беззнаковый целочисленный тип
string	t	Строковый тип
bool	t	Логический тип
function	f	Объявления функции
main	m	Главная функция
while	h	Конструкция цикла
if	z	Условная конструкция (истинная ветвь)
else	e	Условная конструкция (ложная ветвь)
new	n	Объявление переменной
return	r	Выход из функции и возврат значения
true	u	Истинное значение
false	d	Ложное значение
write	w	Оператор вывода (без перевода на новую строку)
writeline	x	Оператор вывода (с переводом на новую строку)
{	{	Начало блока функции

Окончание таблицы 3.1

Токен	Лексема	Описание
}	}	Конец блока функции
(	(	Начало перечислений параметров у функций, приоритет операций в выражениях
)	)	Конец перечислений параметров у функций, приоритет операций в выражениях.
,	,	Разделитель параметров функции
;	;	Конец инструкции
+	+	Арифметический оператор (сложение)
-	-	Арифметический оператор (вычитание)
*	*	Арифметический оператор (умножение)
/	/	Арифметический оператор (деление)
%	%	Арифметический оператор (остаток от деления)
>	>	Логический оператор (больше)
<	<	Логический оператор (меньше)
==	&	Логический оператор (равенство)
>=	p	Логический оператор (больше или равно)
<=	k	Логический оператор (меньше или равно)
!=	j	Логический оператор (не равно)
=	=	Оператор присваивания
array	a	Объявление массива
[]	@	Индекс массива

Для распознавания вышеперечисленных цепочек используется механизм конечного автомата, в которых цепочки записаны в виде регулярных выражений. Пример записи регулярного выражения для идентификатора: [a-zA-Z][a-zA-Z0-9_]*.

Два графа переходов конечных автоматов для цепочки строкового типа и объявления переменной представлены на рисунках 3.3 и 3.4.

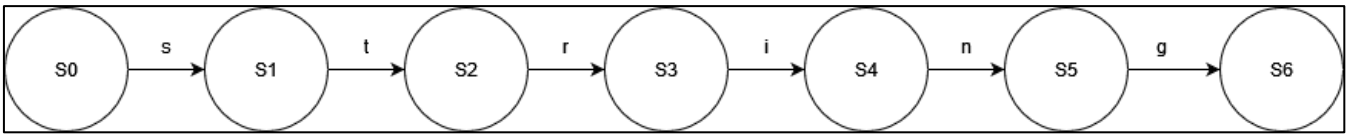

Рисунок 3.3 Граф для строкового типа string

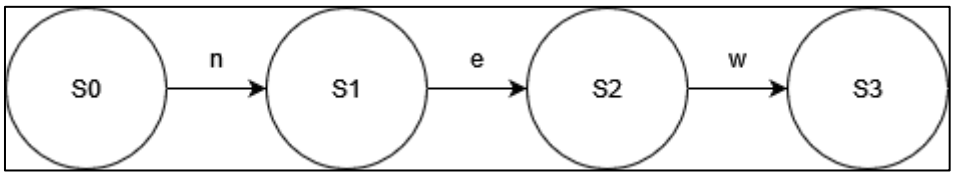

Рисунок 3.4 Граф для объявления переменной

Код на языке C++, который реализует данные цепочки разбора представлен в листингах 3.1 и 3.2.

```
#define FST_STRING FST::FST _string(                \
          str,                                      \
          7,                                        \
          FST::NODE(1, FST::RELATION('s', 1)),      \
          FST::NODE(1, FST::RELATION('t', 2)),      \
          FST::NODE(1, FST::RELATION('r', 3)),      \
          FST::NODE(1, FST::RELATION('i', 4)),      \
          FST::NODE(1, FST::RELATION('n', 5)),      \
          FST::NODE(1, FST::RELATION('g', 6)),      \
          FST::NODE()                               \
)
```

Листинг 3.1 Цепочка разбора для строкового типа

```
#define FST_NEW FST::FST _new(                      \
          str,                                      \
          4,                                        \
          FST::NODE(1, FST::RELATION('n', 1)),      \
          FST::NODE(1, FST::RELATION('e', 2)),      \
          FST::NODE(1, FST::RELATION('w', 3)),      \
          FST::NODE()                               \

)
```

Листинг 3.2 Цепочка разбора для объявления переменной

3.5 Основные структуры данных

Основными структурами данных лексического анализатора языка программирования IPP-2024 являются таблица лексем и таблица идентификаторов. Таблица лексем содержит максимальный и текущий размер, а также список записей, каждая из которых включает саму лексему, её номер строки в исходном тексте и индекс в таблице идентификаторов, если лексема является литералом или идентификатором. Таблица идентификаторов хранит информацию о своём максимальном и текущем размере, а также массив записей. Каждая запись содержит имя идентификатора, его область видимости, тип данных, категорию идентификатора, индекс первой связанной лексемы и значение, представленное в формате, соответствующем типу данных. Реализация таблицы лексем и таблицы идентификаторов на языке C++ представлена в приложении Б.

3.6 Структура и перечень сообщений лексического анализатора

В языке программирования IPP-2024 для обработки ошибок лексический анализатор использует таблицу с сообщениями, которые содержат номер ошибки, вид ошибки, её сообщение, а также номер строки и позиции возникшей ошибки.

Индексы ошибок, обнаруживаемых лексическим анализатором, находятся в диапазоне 120–199. Текст ошибки содержит в себе префикс «Лексический

анализатор». Перечень сообщений лексического анализатора представлен в таблице 3.2.

Таблица 3.2 Перечень ошибок при лексическом анализе языка программирования IPP-2024

Код ошибки	Сообщение
120	Лексический анализатор: недопустимый размер таблицы при её создании
121	Лексический анализатор: превышен допустимый размер таблицы при добавлении элемента
122	Лексический анализатор: недопустимый индекс при получении элемента таблицы
123	Лексический анализатор: недопустимый размер таблицы при её создании
124	Лексический анализатор: превышен допустимый размер таблицы при добавлении элемента
125	Лексический анализатор: недопустимый индекс при получении элемента таблицы
126	Лексический анализатор: превышен допустимый размер лексемы
127	Лексический анализатор: нераспознанная лексема
128	Лексический анализатор: ошибка при считывании строкового литерала

3.7 Принцип обработки ошибок

В языке программирования IPP-2024 при обнаружении ошибки в исходном коде программы лексический анализатор формирует сообщение об ошибке и выводит его в файл с протоколом работы, заданный параметром -log, а также в консоль.

3.8 Параметры лексического анализатора

Входным параметром лексического анализа является структура, полученная после чтения входного файла на этапе проверки исходного кода на допустимость символов.

3.9 Алгоритм лексического анализатора

Алгоритм работы лексического анализатора:

1) Исходный код читается посимвольно. Каждый символ записывается в строковый буфер, пока не будет достигнут символ-сепаратор**;**

2) Строка передаётся различным конечным автоматам. Если какой–либо из автоматов разберет строку успешно, то он вернёт лексическому анализатору одну символьную лексему, которая будет записана в таблицу лексем. Если лексема

является идентификатором или литералом, то далее идет пункт 3. В противном случае алгоритм начинается сначала**;**

3) В зависимости от прошлых лексем, которые означают тип идентификатора, идентификатору присваивается тип данных**;**

4) Если идентификатор только объявляется с явным указанием типа, то в частично заполненной таблице идентификаторов происходит поиск идентификатора с таким именем, предварительно будет записан тип и область видимости из стека. Если поиск успешен, то лексический анализ останавливается и выводится ошибка, в противном случае идентификатор добавляется в таблицу идентификаторов, а также в таблицу лексем добавляется лексема идентификатора с номер индекса таблицы идентификаторов, где он записан. Переходим в пункт 1**;**

5) Если уже объявленный идентификатор просто используется по ходу программы, то происходит его поиск в частично заполненной таблице. Если идентификатор не найден, то будет выведена соответствующая ошибка и лексический анализ будет завершён. В противном случае лексеме присваивается соответствующий индекс таблицы, и она записывается в таблицу лексем. Переходим в пункт 1**;**

6) Если лексема является литералом, то сначала будет выявлен тип литерала и значение. Если такой же литерал записан уже в таблицу идентификаторов, запись в неё производится не будет, в противном случае наоборот. В таблицу лексем записывается лексема со ссылкой на таблицу идентификаторов. Переходим в пункт 1;

7) Если идентификатор является функцией, то она будет записан в таблицу идентификаторов с соответствующими типом возвращаемого значений. Последующие идентификаторы в круглых скобках будут записаны как параметры. В стек помещается функция для отметки об области видимости последующих идентификаторов. Функция там останется пока не буде закончено объявление этой же функции. Переходим в пункт 1

8) Если не пройден весь исходный код, то переходим в пункт 1.

3.10 Контрольный пример

Результат работы лексического анализатора, полученный при выполнении контрольного примера, представлен в приложении Б.

4 Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализатор – часть транслятора, выполняющая синтаксический анализ. Входными данными для синтаксического анализа являются таблица лексем и таблица идентификаторов. Выходные данные – дерево разбора. Синтаксический анализатор использует файл, заданный параметром -log, для записи дерева разбора.

В языке программирования IPP-2024 синтаксический анализ выполняется после завершения работы лексического анализатора.

Структура синтаксического анализатора представлена на рисунке 4.1.

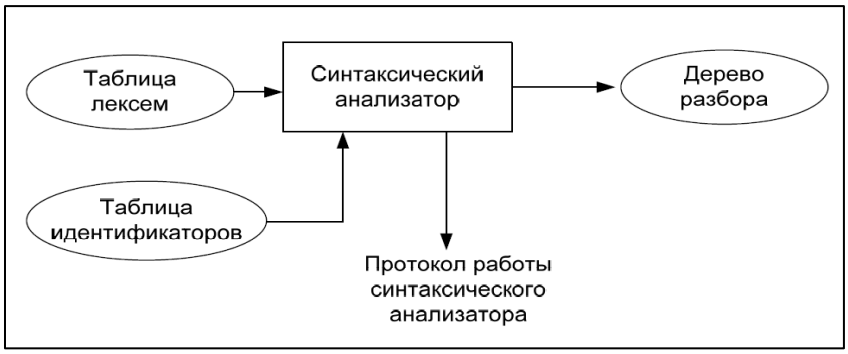

Рисунок 4.1 Структура синтаксического анализатора

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

Синтаксис языка программирования IPP-2024 описывается при помощи грамматики типа 2 иерархии Хомского (контекстно-свободной грамматики).

Контекстно-свободная грамматика – грамматика типа 2 по иерархии Хомского. Данная грамматика имеет вид G = {T, N, P, S}, где:

– T – множество терминальных символов;

– N – множество нетерминальных символов;

– P – множество правил переходов;

– S – стартовый символ;

В контекстно-свободной грамматике правила имеют вид:

– $A \rightarrow \alpha$, где $A \in N$, $\alpha \in V^*$, $V = N \cup T$– словарь грамматики G.

Контекстно-свободная грамматика G имеет нормальную форму Грейбах, если она не является леворекурсивной и правила P имеют вид:

– $A \rightarrow a\alpha$, где $a \in T$, $\alpha \in N^*$;

– $S \rightarrow \lambda$, где $S \in N$ – начальный символ, если есть такое правило, то S не должен встречаться в правой части правил.

Алгоритм преобразования грамматик в нормальную форму Грейбах:

– исключить недостижимые символы из грамматики;

– исключить лямбда-правила из грамматики;

– исключить цепные правила.

Грамматика языка программирования IPP-2024 имеет нормальную форму Грейбах, т. к. она не леворекурсивная (не содержит леворекурсивных правил). Перечень и описание терминальных, нетерминальных символов и правил языка программирования IPP-2024 приведен в таблице 4.1.

Таблица 4.1 Правила грамматики языка программирования IPP-2024

Нетерминал	Цепочка	Описание
S	m{N}|tfi(F){rE;}|tfi(){rE;}|tfi(F){NrE;}|tfi(){NrE;}|tfi(F){rE;}S|tfi(){rE;}S|tfi(F){NrE;}S|tfi(){NrE;}S|nti;|nti=E;|nti;S|nti=E;S|nta@li;|nta@li={A};|nta@li;S|nta@li={A};S|i=E;|i@l=E;|i@i=E;|i=E;S|i@l=E;S|i@i=E;S|wE;|xE;|wE;S|xE;S|h(E){N};|h(E){}|h(E){N};S|h(E){}S|z(E){N}e{N}|z(E){}e{N}|z(E){N}e{}|z(E){}e{}|z(E){N}e{N}S|z(E){}e{N}S|z(E){N}e{}S|z(E){}e{}S	Стартовый символ, порождает всю структуру исходного кода.
N	nti;|nti=E;|nti;N|nti=E;N|nta@li;|nta@li={A};|nta@li;N|nta@li={A};N|i=E;|i@l=E;|i@i=E;|i=E;N|i@l=E;N|i@i=E;N|wE;|xE;|wE;N|xE;N|h(E){N}|h(E){}|h(E){N}N|h(E){}N|z(E){N}e{N}|z(E){}e{N}|z(E){N}e{}|z(E){}e{}|z(E){N}e{N}N|z(E){}e{N}N|z(E){N}e{}N|z(E){}e{}N	Символ порождает правила, описывающие корректную запись операторов
E	i|l|i@l|i@i|(E)|i()|i(W)|iM|lM|i@lM|i@iM|(E)M|i()M|i(W)M	Символ порождает правила, описывающие корректную запись выражений
M	+E|-E|*E|/E|%E|>E|<E|&E|jE|pE|kE|+EM|-EM|*EM|/EM|%EM	Символ порождает правила, описывающие корректную запись подвыражений
F	ti|ti,F	Символ порождает правила, описывающие корректную запись параметров при объявлении функции

Окончание таблицы 4.1

Нетерминал	Цепочка	Описание
W	i|l|i,W|l,W	Символ порождает правила, описывающие корректную запись параметров при вызове функции
A	l|l,A	Символ порождает правила, описывающие корректную запись параметров при инициализации массива

4.3 Построение конечного автомата магазинного типа

В языке программирования IPP-2024 конечный автомат с магазинной памятью представляет собой семерку вида M = {Q, V, Z, δ, q_0, z_0, F}, где:

– Q – множество состояний автомата;
– V – алфавит входных символов;
– Z – алфавит специальных магазинных символов;
– δ - функция переходов автомата;
– q_0 – начальное состояние автомата;
– z_0 – начальное состояние магазина;
– F – множество конечных состояний.

Схема работы конечного автомата с магазинной памятью представлена на рисунке 4.2.

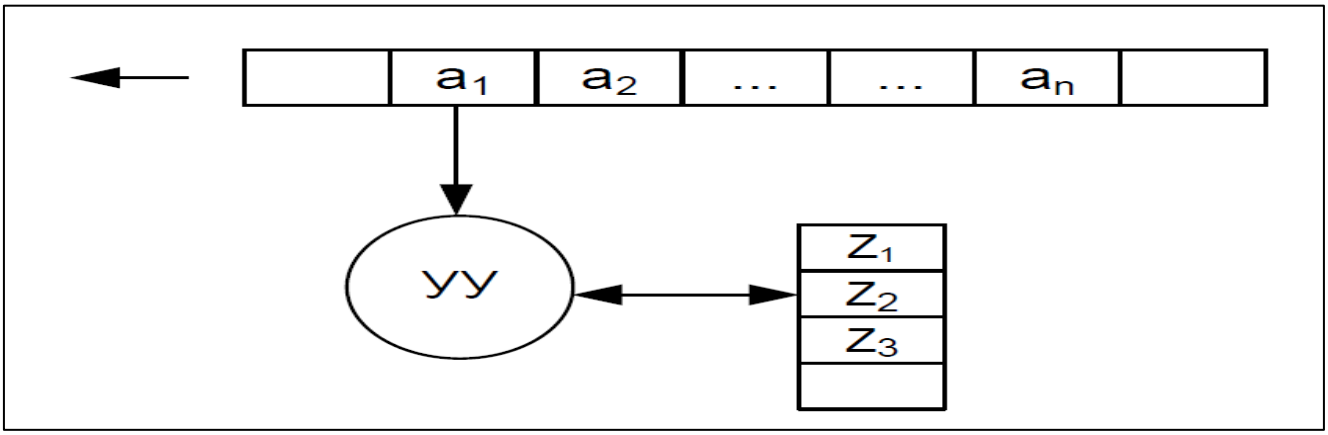

Рисунок 4.2 Схема работы конечного автомата с магазинной памятью

Алгоритм работы конечного автомата с магазинной памятью:

1) текущее состояние автомата – (q, aα, zβ);
2) возможны два случая:

– читает символ a, находящийся под головкой (сдвигает ленту);

– не читает ничего (читает λ, не сдвигает ленту);

3) по функции переходов δ определяет новое состояние q', если (q', γ) ∈ δ (q, a, z) или (q', γ) ∈ δ(q, λ, z);

4) читает верхний символ z (в магазине) и записывает цепочку γ т. к. (q', γ) ∈ δ (q, a, z), при этом, если γ = λ, то верхний символ магазина просто удаляется;

5) работа автомата заканчивается, когда (q, λ, λ).

Результат, демонстрирующий успешный разбор цепочки из контрольного примера, приведен в приложении В.

4.4 Основные структуры данных

Основными структуры данных синтаксического анализатора языка программирования IPP-2024 являются автомат с магазинной памятью и структура грамматики Грейбах, описывающей правила языка IPP-2024. Данные структуры, реализованные на языке C++, представлены в приложении В.

4.5 Описание алгоритма синтаксического разбора

Алгоритм функционирования синтаксического анализатора для языка программирования IPP-2024:

1) В стек автомата помещаются маркер дна и стартовый символ грамматики.

2) На основании предварительно построенной таблицы лексем формируется входная лента.

3) Запускается работа автомата.

4) Выбирается правило грамматики, соответствующее первому символу в ленте и текущему нетерминалу в стеке. Правило разворачивается, а его символы заносятся в стек в обратном порядке.

5) Если текущие терминалы в стеке и ленте совпадают, терминал удаляется из стека, а указатель входной ленты сдвигается на одну позицию вправо. Если они не совпадают, производится возврат к последнему сохранённому состоянию, и для текущего нетерминала выбирается альтернативное правило.

6) При появлении нового нетерминала в правиле анализатор возвращается к шагу 4.

7) Если на вершине стека остаётся только маркер дна, а входная лента полностью обработана, синтаксический анализ завершён успешно. В противном случае фиксируется ошибка синтаксического анализа.

Блок-схема, иллюстрирующая алгоритм, представлена на рисунке 4.3.

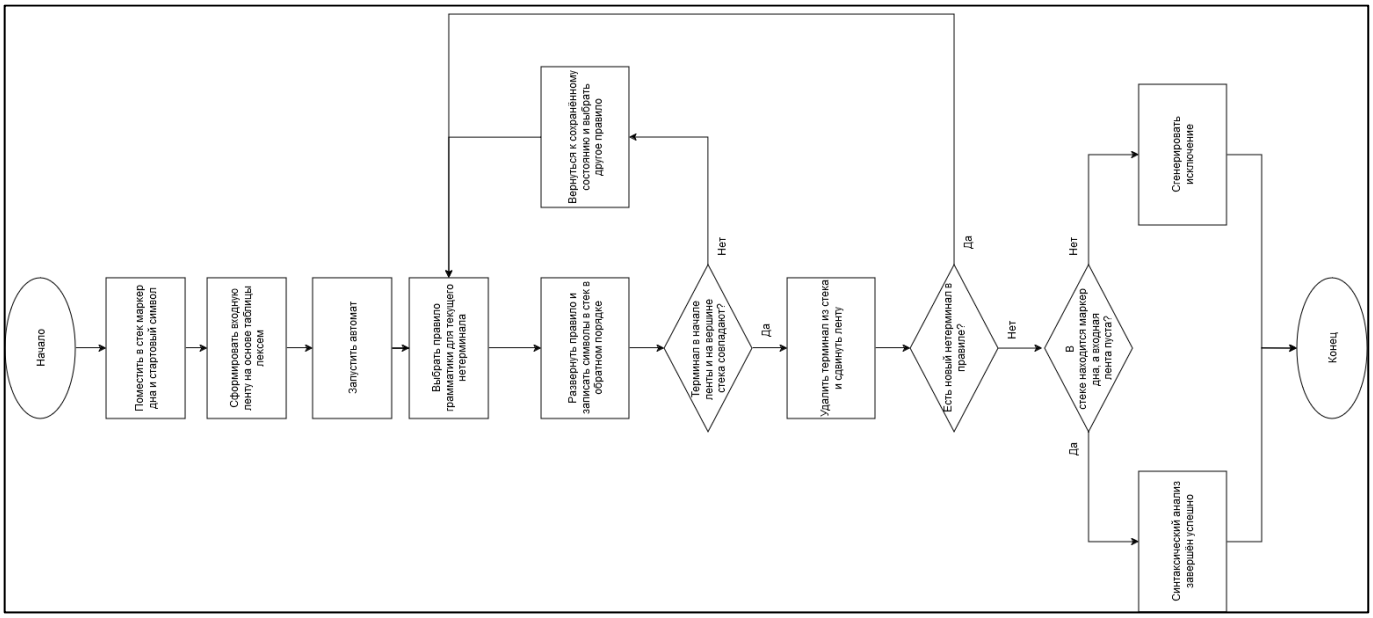

Рисунок 4.3 Блок-схема алгоритма синтаксического анализа

Данный алгоритм представляет обобщённую модель работы магазинного автомата.

4.6. Структура и перечень сообщений синтаксического анализатора

Индексы ошибок, обнаруживаемых синтаксическим анализатором, находятся в диапазоне 200–299. Текст ошибки содержит в себе префикс «Синтаксический анализатор», также ошибка содержит номер строки и позицию возникшей ошибки. Перечень сообщений синтаксического анализатора языка программирования IPP-2024 представлен в таблице 4.2.

Таблица 4.2 Перечень сообщений синтаксического анализатора языка программирования IPP-2024

Код ошибки	Сообщение
200	Синтаксическая анализ: неверная структура программы
201	Синтаксический анализ: некорректное использование операторов языка
202	Синтаксическая анализ: ошибка в выражении
203	Синтаксическая анализ: ошибка в подвыражении
204	Синтаксическая анализ: ошибка в объявлении параметров функции
205	Синтаксическая анализ: ошибка в передаваемых параметрах функции
206	Синтаксическая анализ: ошибка в объявлении массива
207	Синтаксическая анализ: синтаксический анализ завершён досрочно
208	Синтаксический анализ: некорректное использование массива
209	Синтаксический анализ: некорректное закрытие индекса массива

4.7. Параметры синтаксического анализатора и режимы его работы

В языке программирования IPP-2024 специальные параметры для управления режимом работы синтаксического анализатора не предусмотрены. Результат работы лексического анализатора выводится в файл, указанный параметром -log.

4.8. Принцип обработки ошибок

Процесс обработки ошибок организован следующим образом:

1) Синтаксический анализатор проверяет все правила и их цепочки в грамматике, чтобы найти совпадение с конструкцией из таблицы лексем.
2) Если подходящая цепочка не обнаружена, создаётся ошибка соответствующего типа.
3) Все найденные ошибки сохраняются в общей структуре для их учета.
4) После завершения трассировки, при наличии ошибок, в протокол выводится диагностическое сообщение.

Синтаксический анализ завершается досрочно при достижении количества ошибок, равного трем.

4.9 Контрольный пример

Синтаксический анализатор в результате своей работы создаёт дерево разбора контрольного примера, которое приведено в приложении А. Протокол анализа и само дерево разбора исходного кода можно найти в приложении В, а его графическое отображение представлено в Графической работе №1. Протокол анализа демонстрирует пошаговый процесс работы конечного автомата с использованием магазинной памяти. В файл, указанный через параметр -log, записываются такие данные, как номер текущего шага, анализируемое правило, состояние входной ленты и содержимое стека.

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор проверяет смысловую структуру программы, обеспечивая её семантическую корректность. Он выполняет более глубокий анализ, чем лексический и синтаксический анализаторы, выявляя ошибки в типах данных, областях видимости, использовании операторов и функций, а также другие нарушения семантики. Структура семантического анализатора языка программирования IPP-2024 представлена на рисунке 5.1.

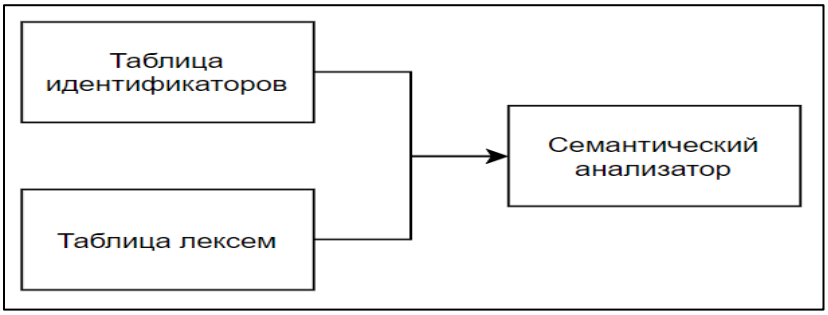

Рисунок 5.1 Структура семантического анализатора языка программирования IPP- 2024

5.2 Функции семантического анализатора

Семантические проверки языка программирования IPP-2024, включая фазы их выполнения представлены в таблице 5.1.

Таблица 5.1 Семантические проверки языка программирования IPP-2024

Семантическая проверка	Фаза выполнения
Дублирование функции main	Лексический анализ
Превышение целочисленным литералом максимального значения для хранения	Лексический анализ
Дублирование пользовательской функции	Лексический анализ
Дублирование параметров функции	Лексический анализ
Дублирование переменной	Лексический анализ
Дублирование массива	Лексический анализ
Значение размера массива	Лексический анализ
Тип размера массива	Лексический анализ
Наличие переменной в доступной области видимости	Лексический анализ
Индекс массива при обращении	Семантический анализ
Тип индекса массива при обращении	Семантический анализ
Типы элементов массива при инициализации	Семантический анализ

Окончание таблицы 5.1

Семантическая проверка	Фаза выполнения
Элементы массива при инициализации должны быть литералами	Семантический анализ
Пустой блок в условных, циклических конструкциях	Семантический анализ
Возможность присваивания	Семантический анализ
Совпадение типов при присваивании	Семантический анализ
Проверка типов библиотечных функций	Семантический анализ
Совпадение типов в логическом выражении	Семантический анализ
Возможность присвоения результата логического выражения	Семантический анализ
Бесконечный цикл в выражении while	Семантический анализ
Проверка аргументов библиотечных функций	Семантический анализ
Проверка типов аргументов при вызове функций	Семантический анализ
Наличие точки входа в программу	Семантический анализ

5.3 Структура и перечень сообщений семантического анализатора

Коды ошибок, обнаруживаемых семантическим анализатором, находятся в диапазоне 300–399, а текст ошибок включает префикс «Семантический анализатор». Полный перечень кодов и сообщений ошибок семантического анализатора приведён в таблице 5.2.

Таблица 5.2 Перечень сообщений семантического анализатора языка программирования IPP-2024

Код ошибки	Сообщение
300	Семантический анализатор: дублирование функции main
301	Семантический анализатор: повторное объявление функции
302	Семантический анализатор: дублирование параметра функции
303	Семантический анализатор: повторное объявление переменной
304	Семантический анализатор: повторное объявление массива
305	Семантический анализатор: недопустимое значение размера массива
306	Семантический анализатор: размер массива должен быть статическим значением
307	Семантический анализатор: идентификатор не найден в доступной области видимости
308	Семантический анализатор: отсутствует точка входа в программу
309	Семантический анализатор: присваивание невозможно
310	Семантический анализатор: функции DATE и TIME возвращают строковый тип

Окончание таблицы 5.2

Код ошибки	Сообщение
311	Семантический анализатор: невозможно присвоить результат логического выражения
312	Семантический анализатор: несоответствие типов данных в выражении
313	Семантический анализатор: недопустимые типы данных для арифметических операций
314	Семантический анализатор: функции DATE и TIME не должны содержать аргументов
315	Семантический анализатор: функция не должна содержать параметры
316	Семантический анализатор: превышено количество аргументов при вызове функции
317	Семантический анализатор: несовпадение типов аргументов функции
318	Семантический анализатор: недостаточное количество аргументов при вызове функции
319	Семантический анализатор: неверный тип данных в выражении if/while
320	Семантический анализатор: недопустимое сравнение в выражении if/while
321	Семантический анализатор: отсутствует оператор return в функции
322	Семантический анализатор: тип возвращаемого значения не соответствует типу функции
323	Семантический анализатор: неверное выражение в операторе return
324	Семантический анализатор: индекс выходит за границы массива
325	Семантический анализатор: недопустимое значение для индекса массива
326	Семантический анализатор: тип элемента не соответствует типу массива при присваивании
327	Семантический анализатор: обнаружен не литерал в инициализаторе массива
328	Семантический анализатор: количество элементов в инициализаторе не совпадает с размером массива
329	Семантический анализатор: несовпадение типов при присваивании
330	Семантический анализатор: несовпадение типов в логическом выражении
331	Семантический анализатор: бесконечный цикл в выражении while
332	Семантический анализатор: пустой блок в if/else/while
333	Семантический анализатор: переполнение в выражении
334	Семантический анализатор: деление на ноль
335	Семантический анализатор: недопустимый тип целочисленного литерала
336	Семантический анализатор: целочисленный литерал превысил максимальное значение для хранения

5.4 Принцип обработки ошибок

При обнаружении ошибки в исходном коде программы на языке программирования IPP-2024 семантический анализатор формирует сообщение об ошибке и выводит его в консоль, а также записывает в файл протокола, указанный параметром –log.

5.5 Контрольный пример

Таблица 5.3 содержит примеры кода с семантическими ошибками и соответствующими сообщениями об этих ошибках.

Таблица 5.3 Примеры диагностики ошибок семантического анализатора

Исходный код программы на языке программирования IPP-2024	Сообщение об ошибке

int function mul(int a, int b)
{
return 2;
}
	Ошибка 308: Семантический анализатор: отсутствует точка входа в программу

int function mul(int a, int b)
{
return 2;
}
 	
main
{
new string str = "param";
new int num = mul(str, 2);
}
	Ошибка 317: Семантический анализатор: несовпадение типов аргументов функции
	
	Строка:9 Позиция:27

main
{
new int num = 10 / 0;
}
	Ошибка 334: Семантический анализатор: деление на ноль

6 Вычисление выражений

6.1 Выражения, допускаемые языком

В языке программирования IPP-2024 разрешены вычисления выражений с использованием целочисленных и беззнаковых целочисленных типов данных, включая возможность вызова функций внутри таких выражений. Также поддерживаются простые логические выражения, состоящие из переменных или литералов целочисленного и беззнакового целочисленного типов. Однако комбинировать логические и арифметические операции в одном выражении нельзя. Приоритет операций указан в таблице 6.1.

Таблица 6.1 Приоритеты операций языка программирования IPP-2024

Операция	Значение приоритета
( )	0
,	1
+, -	2
>, <, >=, <=, ==, !=	3
*, /, %	4
@	5

Примеры выражений из контрольного примера: a * b, num1 > num2, 10 / 5, cnt + 1 и т.д.

6.2 Польская запись и принцип ее построения

Обратная польская запись – форма записи выражений, в которой операнды расположены перед знаками операций.

Язык программирования IPP-2024 транслируется в язык ассемблера, в котором все вычисления производятся через стек. Преобразование исходных выражений в обратную польскую запись, упрощает генерацию кода вычисления выражений в язык ассемблера.

Алгоритм преобразования выражений в обратную польскую запись:

− выражение просматривается слева направо;

− идентификаторы и литералы переносятся в результирующую строку в порядке их следования;

− если идентификатор является именем функции, то он помещается в стек и заменяется специальным символом «$», если после него следует открывающая скобка;

− операция записывается в стек, если стек пуст или в вершине стека находится открывающая скобка;

− операция выталкивает из стека в результирующую строку все операции с приоритетом больше или равным её собственному, после чего помещается в стек;

− открывающая скобка помещается в стек;

− закрывающая скобка выталкивает из стека в результирующую строку все операции до открывающей скобки, после чего обе скобки уничтожаются;

− запятая выталкивает из стека в результирующую строку все операции до открывающей скобки;

− по завершении разбора выражения все оставшиеся в стеке операции выталкиваются в результирующую строку;

− результирующая строка заменяет исходное выражение в таблице лексем.

Примеры построения обратной польской записи из контрольного примера: a b *, cnt 1 +, $ 11o 5o max $ 1011b 1111b max +, 10 5 / и т.д.

6.3 Программная реализация обработки выражения

Программная реализация алгоритма преобразования выражений к обратной польской записи на языке C++ представлена в приложении Г.

6.4 Контрольный пример

Контрольные примеры с выражениями до и после преобразования в обратную польскую нотацию приведены в таблице 6.2.

Таблица 6.2 Преобразование выражений контрольного примера

Выражение	Результат
a * b	a b *
a > b	a b >
10 / 5	10 5 /
max(11o, 5o) + max(1011b, 1111b)	$ 11o 5o max $ 1011b 1111b max +

7 Генерация кода

7.1 Структура генератора кода

Для трансляции кода с языка программирования IPP-2024 был выбран язык ассемблера. Процесс генерации кода осуществляется следующим образом: генератор кода последовательно обрабатывает таблицу лексем, при необходимости обращаясь к таблице идентификаторов. На основе анализируемых лексем выполняется генерация соответствующего ассемблерного кода. Входными данными для генератора служат таблица лексем и таблица идентификаторов, а результатом работы является готовый ассемблерный код. Структура генератора кода представлена на рисунке 7.1.

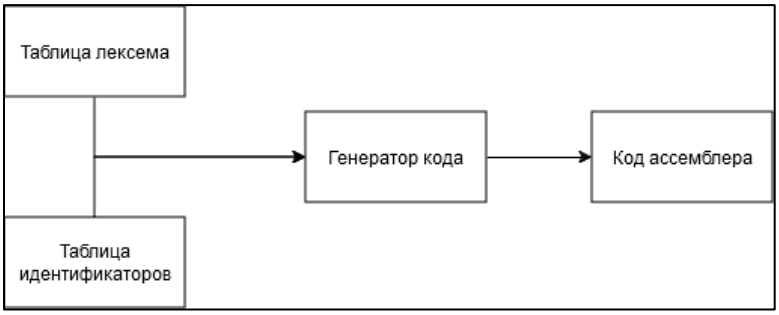

Рисунок 7.1 Структура генератора кода

7.2 Представление типов данных в оперативной памяти

Язык программирования IPP-2024 основан на плоской модели памяти (flat), при которой приложению выделяется единый непрерывный сегмент для размещения кода и данных. Этот сегмент разделён на следующие области:

- .stack – для стека;
- .const – для хранения констант;
- .data – для переменных;
- .code – для машинного кода.

Соответствие типов данных в исходном языке программирования IPP-2024 типам целевого языка приведены в таблице 7.1.

Таблица 7.1 Соответствие типов данных языка программирования IPP-2024 и языка ассемблера

Тип данных	Представление на языке ассемблера	Описание
int	sdword	Хранит целочисленное знаковое значение размером 4 байта.

Окончание таблицы 7.1

Тип данных	Представление на языке ассемблера	Описание
uint	dword	Хранит целочисленное беззнаковое значение размером 4 байта.
bool	dword	Хранит логическое значение true, false
string	dword	Хранит указатель на начало строки

7.3 Статическая библиотека

Функции, входящие в состав статической библиотеки IPP-2024, приведены в таблице 7.2.

Таблица 7.2 Функции стандартной библиотеки языка программирования IPP- 2024

Функция	Описание	Количество параметров
string DATE()	Возвращает строку с текущей датой в формате ДД.ММ.ГГ.	0
string TIME()	Возвращает строку с текущим временем в формате ЧЧ:ММ:СС.	0

Статическая библиотека написана на языке C++. Подключение статической библиотеки производится автоматически на этапе компоновки.

7.4 Особенности алгоритма генерации кода

Обобщенная блок-схема алгоритма генерации кода языка ассемблера изображена на рисунке 7.2.

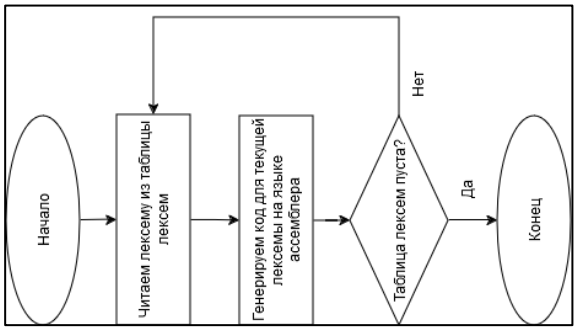

Рисунок 7.2 Обобщенная блок-схема алгоритма генерации кода

Алгоритм генерации исходного кода с языка программирования IPP-2024 на язык ассемблера:

− в файл, заданный параметром -out, записывается информация о модели памяти, используемом соглашении о вызовах и подключаются необходимые библиотеки;

− проходим по таблице идентификаторов и в сегмент констант записываем литералы;

− проходим по таблице лексем и ищем объявление переменных, заполняем сегмент данных этими переменными;

− снова проходим по таблице лексем, заполняя уже сегмент кода.

Для генерации используются шесть функций:

1) CreateNameWithScope(char*& scopedName, const IT::Entry entry) – предназначена для создания имён идентификаторов с учётом их области видимости.

2) Head (Out::OUT& out) – отвечает за запись информации о модели памяти, соглашении о вызовах, а также подключение необходимых библиотек.

3) Const (LA::LEX lex, Out::OUT& out) – используется для генерации сегмента констант.

4) Data (LA::LEX lex, Out::OUT& out) – служит для генерации сегмента переменных.

5) Code (LA::LEX lex, Out::OUT& out) – занимается генерацией сегмента кода.

6) Generation (LA::LEX lex, Out::OUT& out) – объединяет работу всех перечисленных функций и выполняет трансляцию исходного кода на язык ассемблера.

7.5 Входные параметры, управляющие генерацией кода

Входными параметрами генератора в языке программирования IPP-2024 являются таблица идентификаторов и таблица лексем, которые предназначены для генерации кода ассемблера. Результат работы генератора кода выводится в файл, указанный параметром -out.

7.6 Контрольный пример

Контрольный пример языка программирования IPP-2024, сгенерированный в язык ассемблера, представлен в приложении Д.

8 Тестирование транслятора

8.1 Общие положения

Тесты предназначены для выявления ошибок в работе компилятора и их последующего устранения. Ошибки обнаруживались как на ранних этапах разработки, так и на более поздних стадиях.

При возникновении ошибки работа транслятора прекращается, так как ошибка на одном этапе может привести к сбоям на последующих. Сообщение об ошибке с указанием её номера будет выведено в файл протокола и на консоль.

8.2 Результаты тестирования

Описание тестовых наборов, демонстрирующих проверки на разных этапах трансляции, приведено в таблице 8.1.

Таблица 8.1 Результаты тестирования транслятора

Исходный код	Ошибка	Этап

main
{
new int a#;
}
	Ошибка 127:Лексический анализатор: нераспознанная лексема
	
	Строка:3 Позиция:13
		Лексический анализ

main
{
new string str = "stroka;
}
	Ошибка 128:Лексический анализатор: ошибка при считывании строкового литерала
	
	Строка:3 Позиция:26
		Лексический анализ

main
{
new int a = 123
}
	202: строка 3,Синтаксическая анализ: ошибка в выражении
	Ошибка 207:Синтаксическая анализ: синтаксический анализ завершён досрочно
		Синтаксический анализ

int function mul(int a,)
{
return 2;
}
	204: строка 1,Синтаксическая анализ: ошибка в объявлении параметров функции
	Ошибка 207:Синтаксическая анализ: синтаксический анализ завершён досрочно
		Синтаксический анализ

main
{
new int array3] = {1, 2, 3};
}
	Ошибка 209:Синтаксический анализ: некорректное закрытие индекса массива
	
	Строка:3 Позиция:17
		Синтаксический анализ

Окончание таблицы 8.1

Исходный код	Ошибка	Этап

main
{
new int a = "stroka";
}
	Ошибка 329: Семантический анализатор: несовпадение типов при присваивании
	
	Строка:3 Позиция:6
		Семантический анализ

int function mul(int a, int b)
{
new int res = a * b;
return "stroka";
}
	Ошибка 322: Семантический анализатор: тип возвращаемого значения не соответствует типу функции
	
	Строка:4 Позиция:20
		Семантический анализ

main
{
new int array[3] arr = {1, 2, "stroka"};
}
	Ошибка 326: Семантический анализатор: тип элемента не соответствует типу массива при присваивании
	
	Строка:3 Позиция:14
		Семантический анализ

main
{
while(2 > 1)
{
 	
writeline "infinity";
}
}
	Ошибка 331: Семантический анализатор: бесконечный цикл в выражении while
	
	Строка:3 Позиция:3
		Семантический анализ

Заключение

В ходе выполнения курсовой работы был разработан транслятор с языка программирования IPP-2024 в язык ассемблера и написана пояснительная записка со спецификацией языка.

Таким образом, были выполнены основные задачи данной курсовой работы:

- сформулирована спецификация языка программирования IPP-2024;
- разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
- осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка программирования;
- разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
- осуществлена программная реализация синтаксического анализатора;
- разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
- разработан транслятор кода на язык ассемблера;
- проведено тестирование всех вышеперечисленных компонентов.

Количественные и качественные характеристики реализации транслятора:

- количество типов данных: 4;
- количество инструкция языка: 4;
- количество лексем: 33;
- правил грамматики: 7;
- наличие стандартной библиотеки даты и времени;
- наличие арифметических операторов и операторов сравнения;
- наличие пользовательской структуры данных – массива.

Работа по разработке компилятора позволила получить необходимое представление о структурах и процессах, использующихся при построении компиляторов, также были изучены основы теории формальных грамматик и основы общей теории компиляторов.

Список использованных источников

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

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

3 Прата, С. Язык программирования C++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 с.

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

Приложение А

```
uint function mul(uint a, uint b)
{
    new uint res;
    res = a * b;
    return res;
}
main
{
    new string str = "Hello world!";
    writeline str;
    new int num1 = 21;
    new int num2 = 43;
    if(num1 > num2)
    {
        str = DATE();
    }
    else
    {
        str = TIME();
    }
    writeline str;
    new uint num3;
    new uint num4;
    num3 = mul(11o,5o) + mul(1011b,1111b);
    num4 = 10 / 5;
    new int array[4] nums = {1, 2, 3, 4};
    new uint num5 = mul(num1,num2);
    writeline num5;
    new uint cnt = 0;
    while(cnt < 4)
    {
        write cnt;
        write " - ";
        writeline nums[cnt];
        cnt = cnt + 1;
    }
}
```

Листинг 1 Контрольный пример на языке программирования IPP-2024

Приложение Б

```
namespace LT
{
	struct Entry
	{
		char lexema;
		int sn;
		int idxTI = LT_TI_NULLXDX;

		Entry();
		Entry(char lexema, int sn, int idxTI);
	};

	struct LexTable
	{
		int maxsize;
		int size;
		Entry* table;
	};

	LexTable Create(int size);
	void Add(LexTable& lextable, Entry entry);
	Entry GetEntry(LexTable& lextable, int n);
	void WriteTable(LexTable& lextable);
	void Delete(LexTable& lextable);
};
```

Листинг 2 Структура таблицы лексем на языке C++

```
namespace IT
{
	enum IDDATATYPE { VOID = 0, UINT = 1, STRING = 2, BOOL = 3, INT
= 4 };
	enum IDTYPE { V = 1, F = 2, P = 3, L = 4, A = 5};
	struct Entry
	{
		int idxfirstLE;
		char id[TI_ID_MAXSIZE];
		char scope[TI_ID_MAXSIZE];
		IDDATATYPE iddatatype;
		IDTYPE idtype;
		union
		{
			bool vbool;
			unsigned int vuint;
			int vint;
			struct
			{
				int len;
				char str[TI_STR_MAXSIZE];
			} vstr;
		} value;
```

```
        Entry();
    };

    struct IdTable

    {
        int maxsize;
        int size;
        Entry* table;
    };

    IdTable Create(int size);
    void Add(IdTable& idtable, Entry entry);
    Entry GetEntry(IdTable& idtable, int n);
    int IsId(IdTable& idtable, char id[TI_ID_MAXSIZE]);
    int Search(IdTable& idtable, Entry entry);
    void WriteTable(IdTable& idtable);
    void Delete(IdTable& idtable);
}
```

Листинг 3 Структура таблицы идентификаторов на языке C++

```
1    tfi(ti,ti)
2    {
3    nti;
4    i=i*i;
5    ri;
6    }
7    m
8    {
9    nti=l;
10   xi;
11   nti=l;
12   nti=l;
13   z(i>i)
14   {
15   i=i();
16   }
17   e
18   {
19   i=i();
20   }
21   xi;
22   nti;
23   nti;
24   i=i(l,l)+i(l,l);
25   i=l/l;
26   nta@li={l,l,l,l};
27   nti=i(i,i);
28   xi;
29   nti=l;
30   h(i<l)
31   {
32   wi;
```

```
33   wl;
34   xi@i;
35   i=i+l;
36   }
37   }
```

Листинг 4 Таблица лексем

```
№ Id|Identifier      |Data type |Identifier type  |Line in text
|Scope           |Value                          |
-------------------------------------------------------------------
-----------------------------------------------
0    |mul             |uint      |function         |1
|global          |null
1    |a               |uint      |parametr         |1
|mul             |null
2    |b               |uint      |parametr         |1
|mul             |null
3    |res             |uint      |variable         |3
|mul             |0
4    |main            |void      |function         |7
|global          |null
5    |str             |string    |variable         |9
|main            |""(0)
6    |L0              |string    |literal          |9
|main            |"Hello world!"(12)
7    |num1            |int       |variable         |11
|main            |0
8    |L1              |int       |literal          |11
|main            |21
9    |num2            |int       |variable         |12
|main            |0
10   |L2              |int       |literal          |12
|main            |43
11   |DATE            |string    |function         |15
|if0             |null
12   |TIME            |string    |function         |19
|else1           |null
13   |num3            |uint      |variable         |22
|main            |0
14   |num4            |uint      |variable         |23
|main            |0
15   |L3              |int       |literal          |24
|main            |9
16   |L4              |int       |literal          |24
|main            |5
17   |L5              |int       |literal          |24
|main            |11
18   |L6              |int       |literal          |24
|main            |15
19   |L7              |int       |literal          |25
|main            |10
20   |L8              |int       |literal          |25
|main            |5
```

```
21   |L9                |int       |literal          |26
|main            |4
22   |nums              |int       |array            |26
|main            |4
23   |L10               |int       |literal          |26
|main            |1
24   |L11               |int       |literal          |26
|main            |2
25   |L12               |int       |literal          |26
|main            |3
26   |L13               |int       |literal          |26
|main            |4
27   |num5              |uint      |variable         |27
|main            |0
28   |cnt               |uint      |variable         |29
|main            |0
29   |L14               |int       |literal          |29
|main            |0
30   |L15               |int       |literal          |30
|while2          |4
31   |L16               |string    |literal          |33
|while2          |" - "(3)
32   |L17               |int       |literal          |35
|while2          |1
```

Листинг 5 Таблица идентификаторов

Приложение В

```
namespace GRB
{
     struct Rule
     {
          GRBALPHABET nn;
          int iderror;
          short size;
          struct Chain
          {
               short size;
               GRBALPHABET* nt;
               Chain()
               {
                    (*this).size = 0;
                    (*this).nt = 0;
               };
               Chain(short psize, GRBALPHABET s, ...);
               char* GetCChain(char* b);
               static GRBALPHABET T(char t)
               {
                    return GRBALPHABET(t);
               };
               static GRBALPHABET N(char n)
               {
                    return -GRBALPHABET(n);
               };
               static bool IsT(GRBALPHABET s)
               {
                    return s > 0;
               };
               static bool isN(GRBALPHABET s)
               {
                    return !IsT(s);
               };
               static char AlphabetToChar(GRBALPHABET s)

               {
                    return IsT(s) ? char(s) : char(-s);
               };
          } *chains;
          Rule()
          {
               (*this).iderror = -1;
               (*this).nn = 0x00;
               (*this).size = 0;
               (*this).chains = nullptr;
          }
          Rule(GRBALPHABET pnn, int iderror, short psize, Chain c,
...);
          char* GetCRule(char* b, short nchain);
```

```
          short GetNextChain(GRBALPHABET t, Rule::Chain& pchain,
short j);
     };
     struct Greibach

     {
          short size;
          GRBALPHABET startN;
          GRBALPHABET stbottomT;
          Rule* rules;
          Greibach() { (*this).size = 0; (*this).startN = 0;
(*this).stbottomT = 0; (*this).rules = 0; };
          Greibach(GRBALPHABET pstartN, GRBALPHABET pstbottomT,
short psize, Rule r, ...);
          short GetRule(GRBALPHABET pnn, Rule& prule);
          Rule GetRule(short n);
     };
     Greibach GetGreibach();
}
```

Листинг 6 Структура грамматики Грейбах на языке C++

```
class my_stack_SHORT : public std::stack<short>
{
public:
     using std::stack<short>::c;
};

typedef my_stack_SHORT MFSTSTSTACK;


namespace MFST
{
     struct MfstState
     {
          short lenta_position;
          short nrule;
          short nrulechain;
          MFSTSTSTACK st;
          MfstState();
          MfstState(short pposition, MFSTSTSTACK pst, short
pnrulechain);
          MfstState(short pposition, MFSTSTSTACK pst, short pnrule,
short pnrulechain);

     };

     class my_stack_MfstState :public std::stack<MfstState> {
     public:
          using std::stack<MfstState>::c;
     };
     struct Mfst
     {
          enum RC_STEP {
```

```
                NS_OK,
                NS_NORULE,
                NS_NORULECHAIN,
                NS_ERROR,
                TS_OK,
                TS_NOK,
                LENTA_END,
                SURPRISE,
           };

           struct MfstDiagnosis
           {
                short lenta_position;
                RC_STEP rc_step;
                short nrule;
                short nrule_chain;
                MfstDiagnosis();
                MfstDiagnosis(short plenta_position, RC_STEP
prc_step, short pnrule, short pnrule_chain);

           } diagnosis[MFST_DIAGN_NUMBER];

           GRBALPHABET* lenta;
           short lenta_position;
           short nrule;
           short nrulechain;
           short lenta_size;
           GRB::Greibach grebach;
           LT::LexTable lex;
           MFSTSTSTACK st;
           my_stack_MfstState storestate;
           Mfst();
           Mfst(LT::LexTable& plex, GRB::Greibach pgrebach);
           char* GetCSt(char* buf);
           char* GetCLenta(char* buf, short pos, short n = 25);
           char* GetDiagnosis(short n, char* buf);

           bool SaveState(Log::LOG& log);
           bool ResetState(Log::LOG& log);
           bool PushChain(GRB::Rule::Chain chain);
           RC_STEP Step(Log::LOG& log);
           bool Start(Log::LOG& log);
           bool SaveDiagnosis(RC_STEP pprc_step);
           void PrintRules(Log::LOG& log);

           struct Deducation
           {
                short size;
                short* nrules;
                short* nrulechains;
                Deducation()
                {
                     size = 0;
                     nrules = 0;
```

```
                    nrulechains = 0;
                };
            } deducation;

            bool SaveDeducation();
        };
}
```

Листинг 7 Структура автомата с магазинной памятью на языке C++

```
Шаг : Правило                Входная лента                    Стек
0   : S->tfi(F){rE;}         tfi(ti,ti){nti;i=i*i;ri;}        S$
0   : SAVESTATE:             1
0   :                        tfi(ti,ti){nti;i=i*i;ri;}        tfi(F){rE;}$
1   :                        fi(ti,ti){nti;i=i*i;ri;}m        fi(F){rE;}$
2   :                        i(ti,ti){nti;i=i*i;ri;}m{        i(F){rE;}$
3   :                        (ti,ti){nti;i=i*i;ri;}m{n        (F){rE;}$
4   :                        ti,ti){nti;i=i*i;ri;}m{nt        F){rE;}$
5   : F->ti                  ti,ti){nti;i=i*i;ri;}m{nt        F){rE;}$
5   : SAVESTATE:             2
5   :                        ti,ti){nti;i=i*i;ri;}m{nt        ti){rE;}$
6   :                        i,ti){nti;i=i*i;ri;}m{nti        i){rE;}$
7   :                        ,ti){nti;i=i*i;ri;}m{nti=        ){rE;}$
8   : 2
8   : RESSTATE
8   :                        ti,ti){nti;i=i*i;ri;}m{nt        F){rE;}$
9   : F->ti,F                ti,ti){nti;i=i*i;ri;}m{nt        F){rE;}$
9   : SAVESTATE:             2
9   :                        ti,ti){nti;i=i*i;ri;}m{nt        ti,F){rE;}$
10  :                        i,ti){nti;i=i*i;ri;}m{nti        i,F){rE;}$
11  :                        ,ti){nti;i=i*i;ri;}m{nti=        ,F){rE;}$
12  :                        ti){nti;i=i*i;ri;}m{nti=l        F){rE;}$
13  : F->ti                  ti){nti;i=i*i;ri;}m{nti=l        F){rE;}$
13  : SAVESTATE:             3
13  :                        ti){nti;i=i*i;ri;}m{nti=l        ti){rE;}$
14  :                        i){nti;i=i*i;ri;}m{nti=l;        i){rE;}$
15  :                        ){nti;i=i*i;ri;}m{nti=l;x        ){rE;}$
16  :                        {nti;i=i*i;ri;}m{nti=l;xi        {rE;}$
17  :                        nti;i=i*i;ri;}m{nti=l;xi;        rE;}$
18  : 2
18  : RESSTATE
18  :                        ti){nti;i=i*i;ri;}m{nti=l        F){rE;}$
19  : F->ti,F                ti){nti;i=i*i;ri;}m{nti=l        F){rE;}$
19  : SAVESTATE:             3
19  :                        ti){nti;i=i*i;ri;}m{nti=l        ti,F){rE;}$
20  :                        i){nti;i=i*i;ri;}m{nti=l;        i,F){rE;}$
21  :                        ){nti;i=i*i;ri;}m{nti=l;x        ,F){rE;}$
22  : 2
22  : RESSTATE
22  :                        ti){nti;i=i*i;ri;}m{nti=l        F){rE;}$
23  : TNS_NORULECHAIN/NS_NORULE
23  : RESSTATE
23  :                        ti,ti){nti;i=i*i;ri;}m{nt        F){rE;}$
24  : TNS_NORULECHAIN/NS_NORULE
```

```
24  : RESSTATE
24  :                       tfi(ti,ti){nti;i=i*i;ri;}      S$
25  : S->tfi(){rE;}         tfi(ti,ti){nti;i=i*i;ri;}      S$
25  : SAVESTATE:            1
25  :                       tfi(ti,ti){nti;i=i*i;ri;}      tfi(){rE;}$
26  :                       fi(ti,ti){nti;i=i*i;ri;}m      fi(){rE;}$
27  :                       i(ti,ti){nti;i=i*i;ri;}m{      i(){rE;}$
28  :                       (ti,ti){nti;i=i*i;ri;}m{n      (){rE;}$
29  :                       ti,ti){nti;i=i*i;ri;}m{nt      ){rE;}$
30  : 2
30  : RESSTATE
30  :                       tfi(ti,ti){nti;i=i*i;ri;}      S$
31  : S->tfi(F){NrE;}       tfi(ti,ti){nti;i=i*i;ri;}      S$
31  : SAVESTATE:            1
31  :                       tfi(ti,ti){nti;i=i*i;ri;}
tfi(F){NrE;}$
32  :                       fi(ti,ti){nti;i=i*i;ri;}m      fi(F){NrE;}$
33  :                       i(ti,ti){nti;i=i*i;ri;}m{      i(F){NrE;}$
34  :                       (ti,ti){nti;i=i*i;ri;}m{n      (F){NrE;}$
35  :                       ti,ti){nti;i=i*i;ri;}m{nt      F){NrE;}$
36  : F->ti                 ti,ti){nti;i=i*i;ri;}m{nt      F){NrE;}$
36  : SAVESTATE:            2
36  :                       ti,ti){nti;i=i*i;ri;}m{nt      ti){NrE;}$
37  :                       i,ti){nti;i=i*i;ri;}m{nti      i){NrE;}$
38  :                       ,ti){nti;i=i*i;ri;}m{nti=      ){NrE;}$
39  : 2
39  : RESSTATE
39  :                       ti,ti){nti;i=i*i;ri;}m{nt      F){NrE;}$
40  : F->ti,F               ti,ti){nti;i=i*i;ri;}m{nt      F){NrE;}$
40  : SAVESTATE:            2
40  :                       ti,ti){nti;i=i*i;ri;}m{nt      ti,F){NrE;}$
41  :                       i,ti){nti;i=i*i;ri;}m{nti      i,F){NrE;}$
42  :                       ,ti){nti;i=i*i;ri;}m{nti=      ,F){NrE;}$
43  :                       ti){nti;i=i*i;ri;}m{nti=l      F){NrE;}$
44  : F->ti                 ti){nti;i=i*i;ri;}m{nti=l      F){NrE;}$
44  : SAVESTATE:            3
44  :                       ti){nti;i=i*i;ri;}m{nti=l      ti){NrE;}$
45  :                       i){nti;i=i*i;ri;}m{nti=l;      i){NrE;}$
46  :                       ){nti;i=i*i;ri;}m{nti=l;x      ){NrE;}$
47  :                       {nti;i=i*i;ri;}m{nti=l;xi      {NrE;}$
48  :                       nti;i=i*i;ri;}m{nti=l;xi;      NrE;}$
49  : N->nti;               nti;i=i*i;ri;}m{nti=l;xi;      NrE;}$
49  : SAVESTATE:            4
49  :                       nti;i=i*i;ri;}m{nti=l;xi;      nti;rE;}$
50  :                       ti;i=i*i;ri;}m{nti=l;xi;n      ti;rE;}$
...
2113: SAVESTATE:            67
2113:                       i+l;}}                         iM;}}$
2114:                       +l;}}                          M;}}$
2115: M->+E                 +l;}}                          M;}}$
2115: SAVESTATE:            68
2115:                       +l;}}                          +E;}}$
2116:                       l;}}                           E;}}$
2117: E->l                  l;}}                           E;}}$
```

```
2117: SAVESTATE:          69
2117:                     l;}}                              l;}}$
2118:                     ;}}                               ;}}$
2119:                     }}                                }}$
2120:                     }                                 }$
2121:                                                       $
2122: 6
2123: ------>LENTA_END
```

Листинг 8 Протокол работы синтаксического анализатора

```
0   : S->tfi(F){NrE;}S
4   : F->ti,F
7   : F->ti
11  : N->nti;N
15  : N->i=E;
17  : E->iM
18  : M->*E
19  : E->i
22  : E->i
25  : S->m{N}
27  : N->nti=E;N
31  : E->l
33  : N->xE;N
34  : E->i
36  : N->nti=E;N
40  : E->l
42  : N->nti=E;N
46  : E->l
48  : N->z(E){N}e{N}N
50  : E->iM
51  : M->>E
52  : E->i
55  : N->i=E;
57  : E->i()
64  : N->i=E;
66  : E->i()
71  : N->xE;N
72  : E->i
74  : N->nti;N
78  : N->nti;N
82  : N->i=E;N
84  : E->i(W)M
86  : W->l,W
88  : W->l
90  : M->+E
91  : E->i(W)
93  : W->l,W
95  : W->l
98  : N->i=E;N
100 : E->lM
101 : M->/E
102 : E->l
104 : N->nta@li={A};N
```

```
112 : A->l,A
114 : A->l,A
116 : A->l,A
118 : A->l
121 : N->nti=E;N
125 : E->i(W)
127 : W->i,W
129 : W->i
132 : N->xE;N
133 : E->i
135 : N->nti=E;N
139 : E->l
141 : N->h(E){N}
143 : E->iM
144 : M-><E
145 : E->l
148 : N->wE;N
149 : E->i
151 : N->wE;N
152 : E->l
154 : N->xE;N
155 : E->i@i
159 : N->i=E;
161 : E->iM
162 : M->+E
163 : E->l
```

Листинг 9 Дерево разбора контрольного примера

Приложение Г

```
#include "pch.h"

namespace PN
{
     void PN(LT::LexTable& lextable, IT::IdTable& idtable)
     {
          for (int i = 0; i < lextable.size; i++)
          {
               if (lextable.table[i].lexema == LEX_EQUAL_SIGN)
               {
                    if (lextable.table[i + 2].lexema ==
LEX_SEMICOLON)
                    {
                         continue;
                    }
                    else
                    {
                         PolishNotation(++i, lextable, idtable,
LEX_SEMICOLON);
                    }
               }
               if (lextable.table[i].lexema == LEX_IF ||
lextable.table[i].lexema == LEX_WHILE)
               {
                i += 2;
                PolishNotation(i, lextable, idtable,
LEX_RIGHTTHESIS);
               }
          }
     }
    void PolishNotation(int pos, LT::LexTable& lextable,
IT::IdTable& idtable, char endLexem)
    {
        std::stack<LT::Entry> operatorsStack;
        std::queue<LT::Entry> output;
        int countOfLex = 0;
        int expressionPosition = pos;

        for (expressionPosition;
lextable.table[expressionPosition].lexema != endLexem;
expressionPosition++, countOfLex++)
        {
            switch (lextable.table[expressionPosition].lexema)
            {
            case LEX_ID:
                if
(idtable.table[lextable.table[expressionPosition].idxTI].idtype ==
IT::F)
                {

operatorsStack.push(lextable.table[expressionPosition]);
```

```
                }
                else
                {
                    output.push(lextable.table[expressionPosition]);
                }
                break;
            case LEX_LITERAL:
                output.push(lextable.table[expressionPosition]);
                break;
            case LEX_LEFTTHESIS:
                if (lextable.table[expressionPosition - 1].lexema ==
LEX_ID && lextable.table[expressionPosition - 1].idxTI != TI_NULLIDX
&&
                    idtable.table[lextable.table[expressionPosition
- 1].idxTI].idtype == IT::F)
                {
                    output.push(LT::Entry{ '$',
lextable.table[expressionPosition].sn, -1 });
                }

operatorsStack.push(lextable.table[expressionPosition]);
                break;
            case LEX_RIGHTTHESIS:
                while (!operatorsStack.empty() &&
operatorsStack.top().lexema != LEX_LEFTTHESIS)
                {
                    output.push(operatorsStack.top());
                    operatorsStack.pop();
                }
                if (!operatorsStack.empty() &&
operatorsStack.top().lexema == LEX_LEFTTHESIS)
                {
                    operatorsStack.pop();
                }
                if (!operatorsStack.empty() &&
operatorsStack.top().idxTI != TI_NULLIDX &&
idtable.table[operatorsStack.top().idxTI].idtype == IT::F)
                {
                    output.push(operatorsStack.top());
                    operatorsStack.pop();
                }
                break;
            case LEX_COMMA:
                while (!operatorsStack.empty() &&
operatorsStack.top().lexema != LEX_LEFTTHESIS)
                {
                    output.push(operatorsStack.top());
                    operatorsStack.pop();
                }
                break;
            case LEX_PLUS:
            case LEX_MINUS:
            case LEX_STAR:
            case LEX_DIRSLASH:
```

```
            case LEX_PERCENT:
            case LEX_LESS:
            case LEX_MORE:
            case LEX_MORE_OR_EQUAL:
            case LEX_LESS_OR_EQUAL:
            case LEX_EQUAL:
            case LEX_NOT_EQUAL:
                if (lextable.table[expressionPosition].lexema ==
LEX_MINUS && lextable.table[expressionPosition + 1].lexema == LEX_ID
&&
                    lextable.table[expressionPosition + 1].idxTI !=
TI_NULLIDX && idtable.table[lextable.table[expressionPosition +
1].idxTI].idtype == IT::F)
                {
                    output.push(lextable.table[expressionPosition]);
                    continue;
                }
                while (!operatorsStack.empty() &&
GetPriority(lextable.table[expressionPosition], idtable) <=
GetPriority(operatorsStack.top(), idtable))
                {
                    output.push(operatorsStack.top());
                    operatorsStack.pop();
                }

operatorsStack.push(lextable.table[expressionPosition]);
                break;
            default:
                break;
            }
        }
        while (!operatorsStack.empty())
        {
            output.push(operatorsStack.top());
            operatorsStack.pop();
        }
        for (int i = 0; i < countOfLex; i++)
        {
            if (!output.empty())
            {
                lextable.table[pos + i] = output.front();
                output.pop();
            }
            else
            {
                lextable.table[pos + i] = LT::Entry{ PN_FILLER,
lextable.table[pos].sn, -1 };
            }
        }
    }
     int GetPriority(LT::Entry entry, IT::IdTable& idtable)
     {
          switch (entry.lexema)
          {
```

```
            case LEX_LEFTTHESIS:
            case LEX_RIGHTTHESIS:
                 return 0;
            case LEX_COMMA:
                 return 1;
            case LEX_PLUS:
            case LEX_MINUS:
                 return 2;
            case LEX_MORE:
            case LEX_LESS:
            case LEX_MORE_OR_EQUAL:
            case LEX_LESS_OR_EQUAL:
            case LEX_EQUAL:
            case LEX_NOT_EQUAL:
                 return 3;
            case LEX_STAR:
            case LEX_DIRSLASH:
          case LEX_PERCENT:
                 return 4;
            case LEX_IND:
                 return 5;
            }
            return -1;
      }
}
```

Листинг 10 Программная реализация обработки выражений на языке C++

Приложение Д

```
.586
.model flat, stdcall
includelib kernel32.lib
includelib libucrt.lib
includelib ../Debug/IPP-2024L.lib
includelib ../Debug/IPP-2024ASML.lib

ExitProcess proto : dword
SetConsoleTitleA proto: dword
GetStdHandle proto: dword
WriteConsoleA proto: dword, : dword, : dword, : dword, : dword
SetConsoleOutputCP proto : dword
SetConsoleCP proto : dword
overflow_error_message proto
division_by_zero_error_message proto
index_error_message proto
GetStringArrayElementAndOffset proto
GetIntArrayElementAndOffset proto
PrintUnsignedInt proto : dword
PrintUnsignedIntNewLine proto : dword
PrintInt proto : dword
PrintIntNewLine proto : dword
PrintBoolean proto : dword
PrintBooleanNewLine proto : dword
PrintConsole proto : dword
PrintConsoleNewLine proto : dword
extrn TIME : proc
extrn DATE : proc

.stack 4096

.const
consoleTitle byte "IPP-2024", 0
strPause byte "pause", 0
emptyString byte " ", 0
L0 sdword 3
L1 sdword 1
L2 sdword 2
L3 sdword 3
L4 sdword 1
L5 sdword 4
L6 sdword 0
L7 sdword 3
L8 sdword 1

.data
nums_main sdword  1, 2, 3
cnt_main sdword 0

.code
main proc
```

```
     push 1251d
     call SetConsoleOutputCP
     push 1251d
     call SetConsoleCP

     push offset consoleTitle
     call SetConsoleTitleA
     push -11
     call GetStdHandle

     xor eax, eax
     mov eax, L5
     push eax
     push lengthof nums_main
     push offset nums_main
     push L4
     call GetIntArrayElementAndOffset
     pop ebx
     mov [eax], ebx
     xor eax, eax
     mov eax, L6
     mov cnt_main, eax
WHILE_0:
     mov eax, cnt_main
     cmp eax, L7
     jge END_WHILE_0
     push lengthof nums_main
     push offset nums_main
     push cnt_main
     call GetIntArrayElementAndOffset
     push [eax]
     call PrintIntNewLine
     xor eax, eax
     mov eax, cnt_main
     push eax
     mov eax, L8
     pop ebx
     add eax, ebx
     jo OVERFLOW_ERROR
     mov cnt_main, eax
     jmp WHILE_0
END_WHILE_0:
     jmp NO_ERRORS


OVERFLOW_ERROR:
     call overflow_error_message

DIVISION_BY_ZERO_ERROR:
     call division_by_zero_error_message

NO_ERRORS:
     push 0
     call ExitProcess
```

```
main endp

end main
```

Листинг 11 Результат генерации кода на основе контрольного примера