**Содержание**

[Введение 5](#_Toc153772913)

[Глава 1 Спецификация языка программирования 6](#_Toc153772914)

[1.1 Характеристика языка программирования 6](#_Toc153772915)

[1.2 Определение алфавита языка программирования 6](#_Toc153772916)

[1.3 Применяемые сепараторы 6](#_Toc153772917)

[1.4 Применяемые кодировки 7](#_Toc153772918)

[1.5 Типы данных 7](#_Toc153772919)

[1.6 Преобразование типов данных 8](#_Toc153772920)

[1.7 Идентификаторы 8](#_Toc153772921)

[1.8 Литералы 9](#_Toc153772922)

[1.9 Объявления данных 9](#_Toc153772923)

[1.10 Инициализация данных 10](#_Toc153772924)

[1.11 Инструкции языка 10](#_Toc153772925)

[1.12 Операции языка 11](#_Toc153772926)

[1.13 Выражения и их вычисления 12](#_Toc153772927)

[1.14 Конструкции языка 12](#_Toc153772928)

[1.15 Область видимости идентификаторов 13](#_Toc153772929)

[1.16 Семантические проверки 13](#_Toc153772930)

[1.17 Распределение оперативной памяти на этапе выполнения 14](#_Toc153772931)

[1.18 Стандартная библиотека и её состав 14](#_Toc153772932)

[1.19 Ввод и вывод данных 14](#_Toc153772933)

[1.20 Точка входа 15](#_Toc153772934)

[1.21 Препроцессор 15](#_Toc153772935)

[1.22 Соглашения о вызовах 15](#_Toc153772936)

[1.23 Объектный код 15](#_Toc153772937)

[1.24 Классификация сообщений транслятора 15](#_Toc153772938)

[1.25 Контрольный пример 15](#_Toc153772939)

[Глава 2 Структура транслятора 16](#_Toc153772940)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 16](#_Toc153772941)

[2.2 Перечень входных параметров транслятора 17](#_Toc153772942)

[2.3 Протоколы, формируемые транслятором 17](#_Toc153772943)

[Глава 3 Разработка лексического анализатора 18](#_Toc153772944)

[3.1 Структура лексического анализатора 18](#_Toc153772945)

[3.2 Входные и выходные данные лексического анализатора 18](#_Toc153772946)

[3.3 Параметры лексического анализатора 19](#_Toc153772947)

[3.4 Алгоритм лексического анализа 19](#_Toc153772948)

[3.5 Контроль входных символов 19](#_Toc153772949)

[3.6 Удаление избыточных символов 20](#_Toc153772950)

[3.7 Перечень ключевых слов 20](#_Toc153772951)

[3.8 Основные структуры данных 21](#_Toc153772952)

[3.9 Структура и перечень сообщений лексического анализатора 21](#_Toc153772953)

[3.10 Принцип обработки ошибок 21](#_Toc153772954)

[3.11 Контрольный пример 21](#_Toc153772955)

[Глава 4 Разработка синтаксического анализатора 22](#_Toc153772956)

[4.1 Структура синтаксического анализатора 22](#_Toc153772957)

[4.2 Контекстно свободная грамматика, описывающая синтаксис языка 22](#_Toc153772958)

[4.3 Построение конечного магазинного автомата 23](#_Toc153772959)

[4.4 Основные структуры данных 24](#_Toc153772960)

[4.5 Описание алгоритма синтаксического разбора 24](#_Toc153772961)

[4.6 Параметры синтаксического анализатора 24](#_Toc153772962)

[4.7 Структура и перечень сообщений синтаксического анализатора 24](#_Toc153772963)

[4.8 Принцип обработки ошибок 25](#_Toc153772964)

[4.9 Контрольный пример 25](#_Toc153772965)

[Глава 5 Разработка семантического анализатора 26](#_Toc153772966)

[5.1 Структура семантического анализатора 26](#_Toc153772967)

[5.2 Функции семантического анализатора 26](#_Toc153772968)

[5.3 Структура и перечень сообщений семантического анализатора 27](#_Toc153772969)

[5.4 Принцип обработки ошибок 28](#_Toc153772970)

[5.5 Контрольный пример 28](#_Toc153772971)

[Глава 6 Вычисление выражений 29](#_Toc153772972)

[6.1 Выражения, допускаемые языком 29](#_Toc153772973)

[6.2 Польская запись и принцип ее построения 29](#_Toc153772974)

[6.3 Программная реализация обработки выражений 30](#_Toc153772975)

[6.4 Контрольный пример 30](#_Toc153772976)

[Глава 7 Генерация кода 31](#_Toc153772977)

[7.1 Структура генератора кода 31](#_Toc153772978)

[7.2 Представление типов данных в оперативной памяти 31](#_Toc153772979)

[7.3 Статическая библиотека 32](#_Toc153772980)

[7.4 Особенности алгоритма генерации кода 32](#_Toc153772981)

[7.5 Параметры, управляющие генерацией кода 33](#_Toc153772982)

[7.6 Контрольный пример 33](#_Toc153772983)

[Глава 8 Тестирование транслятора 34](#_Toc153772984)

[8.1 Общие положения 34](#_Toc153772985)

[8.2 Результаты тестирования 34](#_Toc153772986)

[Заключение 36](#_Toc153772987)

[Список использованных источников 37](#_Toc153772988)

[Приложение А 38](#_Toc153772989)

[Приложение Б 40](#_Toc153772990)

[Приложение В 42](#_Toc153772991)

[Приложение Г 44](#_Toc153772992)

[Приложение Д 47](#_Toc153772993)

[Приложение Е 49](#_Toc153772994)

[Приложение Ж 51](#_Toc153772995)

[Приложение З 53](#_Toc153772996)

[Приложение И 56](#_Toc153772997)

Введение

Целью курсового проекта является разработка компилятора для нового языка программирования с названием KTI-2023. Этот язык предназначен для выполнения базовых операций и арифметических вычислений с числами.

Компилятор представляет собой программу, которая выполняет процесс преобразования программы, написанной на одном языке программирования, в эквивалентную программу, написанную на языке ассемблера.

Язык ассемблера — машинно-ориентированный язык программирования низкого уровня. Представляет собой систему обозначений, используемую для представления в удобно читаемой форме программ, записанных в машинном коде. Его команды прямо соответствуют отдельным командам машины или их последовательностям.

Сам процесс компиляции разделяется на две основные части: анализ и генерацию. Анализ – это разбиение исходной программы на составные части и создание ее промежуточного представления. Генерация – конструирование требуемой целевой программы из промежуточного представления. В данном курсовом проекте исходный код транслируется на язык ассемблера.

Компилятор KTI-2023 включает в себя четыре основных компонента:

* лексический анализатор;
* синтаксический анализатор;
* семантический анализатор;
* генератор исходного кода на языке ассемблера.

С учетом основной цели курсового проекта, были определены следующие этапы и задачи:

* разработка спецификации языка программирования;
* разработка структуры транслятора;
* разработка лексического анализатора;
* разработка синтаксического анализатора;
* выполнение преобразования выражений;
* генерация кода на язык ассемблер;
* проведение тестирование транслятора.

Глава 1 Спецификация языка программирования

1.1 Характеристика языка программирования

KTI-2023 является компилируемым процедурным языком высокого уровня. Поддерживает 2 типа данных: строковый (string) и целочисленный (int). В стандартной библиотеке языка содержится 2 функции для работы со строками: copystr (копирование строк) и joinstr (конкатенация строк). Преобразование типов в языке KTI-2023 не поддерживается.

1.2 Определение алфавита языка программирования

Алфавит языка программирования – это набор символов, которые используются для написания компьютерных программ. Как и в любом другом языке, алфавит определяет диапазон доступных символов и правила их комбинирования.

Алфавит языка KTI-2023 состоит из следующих множеств символов:

* для записи инструкций языка используются символы: [a…z], [A…Z];
* для записи литералов (строковых и целочисленных) используют символы: [a…я], [a…z], [0…9];
* В качестве сепараторов и специальных символов используются: () [] ; + - \* / % = , #; «пробел».

1.3 Применяемые сепараторы

Символы сепараторы служат в качестве разделителей операций языка. Сепараторы, используемые в языке программирования KTI-2023, приведены в таблице 1.1.

Таблица 1.1 - Применяемые сепараторы

|  |  |
| --- | --- |
| Символы | Назначение |
| (…) | Повышение приоритета арифметических операций, ограничение блока параметров функции, условие окончания цикла. |
| […] | Ограничение тела функции или цикла. |
| ; | Разделитель программных конструкций. |
| + - \* / % | Арифметические операции. |
| = | Оператор присваивания. |
| , | Разделитель параметров функции. |
| # | Начало строки комментария. |

Окончание таблицы 1.1

|  |  |
| --- | --- |
| Символы | Назначение |
| ; | Конец строки комментария. |
| «пробел» | Разделитель цепочек языка, не допустим и именах идентификаторов и ключевых словах языка. |

Сепараторы играют ключевую роль в структурировании кода

1.4 Применяемые кодировки

При трансляции исходного кода применяется кодировка Windows-1251. Содержимое таблицы символов представлено на рисунке 1.1.



Рисунок 1.1 – Таблица кодировки Windows-1251

Исходный код KTI-2023 может содержать символы кириллицы, латинского алфавита верхнего и малого регистров, цифры десятичной системы счисления от 0 до 9, символы табуляции и перевода строки, спецсимволы: [] () , ; + - / \* % = #. Описание кодировки представлено в пункте 1.2

1.5 Типы данных

Допускается использование фундаментальных типов данных. В языке KTI- 2023 есть 2 типа данных: целочисленный и строковый. Описание типов данных, предусмотренных в данным языке представлено в таблице 1.2.

Таблица 1.2 - Фундаментальные типы данных

|  |  |
| --- | --- |
| Тип данных | Описание |
| int | По умолчанию инициализируется 0.  Размер в байтах: 4.  Допустимый диапазоны значений: от −2147483648 до 2147483647.  Применяемые операции:  + (бинарный) – сложение;  - (бинарный) – вычитание;  - (унарный) – отрицательное число;  \* (бинарный) – умножение;  / (бинарный) – деление;  % (бинарный) – остаток от деления; |
| string | По умолчанию инициализируется пустой строкой: ‘’.  Размер в байтах (для одного символа): 1 байт.  Допустимый диапазоны значений: от 0 до 255.  Применяемые операции:  + (бинарный) – конкатенация. |

Тип данных в программировании определяет характеристики и возможности переменной или значения. Он указывает, какую информацию можно хранить в переменной, как оперировать этой информацией и какую память занимает переменная.

1.6 Преобразование типов данных

В языке программирования KTI-2023 преобразование типов данных не предусмотрено.

1.7 Идентификаторы

Для именования функций, параметров и переменных используются идентификаторы. Общее число идентификаторов не должно превышать максимальный размер таблицы идентификаторов (4096). Длина идентификатора не должна превышать 7 символов. В именах идентификаторов допустимо использование символов верхнего и нижнего регистров. Идентификаторы не должны совпадать с ключевыми словами языка. К типам идентификаторов относятся: имя переменной, имя функции, имя параметра.

Имя идентификатора составляется по следующим образом:

* состоит из символов латинского алфавита [a..z];
* не допускается использование цифр.

1.8 Литералы

Литерал — это элемент программы, который непосредственно представляет значение [1]. В программировании литерал является фиксированным значением, записанным непосредственно в коде программы.

В языке KTI-2023 существует 2 типа литералов: целочисленный и строковый. Язык поддерживает представление целочисленных литералов в десятичной, восьмеричной и двоичной системе счисления.

Краткое описание литералов представлено в таблице 1.3.

Таблица 1.3 - Описание литералов

|  |  |
| --- | --- |
| Литералы | Характеристика |
| Строковые | Набор символов алфавита языка, заключенный в одинарные кавычки (‘’).  Допустимо использование символов только нижнего регистра. |
| Целочисленные в двоичной форме | Последовательность цифр 0 и 1 с обязательным префиксом “Bx”: Bx101. |
| Целочисленные в восьмеричной форме | Последовательность цифр 0…7 с обязательным префиксом “Ox”: Ox172. |
| Целочисленные в десятичной форме | Последовательность цифр 0…9 без префиксов: 79. |
| Отрицательные целочисленные | Для того, чтобы получить отрицательное значение литерала, необходимо использовать унарный оператор “-“: -Ox172, -8. |

В языке KTI-2023 литералы используются для представления конкретных данных, таких как числа и строки.

1.9 Объявления данных

Для объявления переменной используется ключевое слово **var**, после которого необходимо указать тип данных и имя идентификатора.

Объявление без инициализации:

**var int** x;

Объявление с инициализацией:

**var string** m=’hello’;

Для объявления функции необходимо использовать ключевое слово **fn**, предварительно указав тип функции. Далее необходимо указать имя функции, список параметров и ее тело. Для объявления параметра необходимо перед именем идентификатора указать тип данных:

**int fn** Func(**int** a, **int** b)

[

…

];

1.10 Инициализация данных

По умолчанию переменные типа int инициализируются нулём, а переменные типа string – пустой строкой (‘’). В языке KTI-2023 также допустима инициализация при объявлении. При этом переменной присваивается значение литерала или идентификатора, находящееся справа от знака равенства.

Таблица 1.4 – Способы инициализации переменных

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

Инициализация данных в программировании означает присвоение начальных значений переменным или структурам данных в момент их создания. Это позволяет установить изначальное состояние для использования данных в программе.

1.11 Инструкции языка

В языке программирования KTI-2023 применяются инструкции, представленные в таблице 1.5.

Таблица 1.5 - Инструкции языка

|  |  |
| --- | --- |
| Инструкция | Описание |
| Объявление переменной | var <тип данных><идентификатор>;  Пример: var string q; |

Окончание таблицы 1.5

|  |  |
| --- | --- |
| Инструкция | Описание |
| Возврат значения функции | reout <идентификатор/литерал>;  Пример: reout m; reout 4; |
| Вывод данных | write<идентификатор/литерал>;  Пример: write'hi'; write Func(4,3); |
| Вызов функции | <идентификатор>(<список параметров>);  Пример: fi(z); fi(1); |
| Операция присваивания | <идентификатор>=<литерал/идентификатор>;  Пример: g=joinst(a,b); |

Таким образом язык KTI-2023 использует следующий синтаксис, приведённый выше.

1.12 Операции языка

Арифметические операции, которые можно использовать в языке KTI-2023, представлены в таблице 1.6.

Таблица 1.6 — Операции языка и их приоритеты

|  |  |  |
| --- | --- | --- |
| Оператор | Характеристика | Приоритет |
| + (для данных типа int) | сложение | 2 |
| + (для данных типа string) | конкатенация | 2 |
| - (бинарный) | вычитание | 2 |
| - (унарный) | отрицательное значение целочисленного литерала | - |
| \* | умножение | 3 |
| / | деление | 3 |
| % | остаток от деления | 3 |
| () | повышение приоритета операций | 4 |

Приоритет подчиняется правилам:

* если в выражение без скобок входят только сложение и вычитание или только умножение, деление и остаток от деления, то действия выполняют в том порядке, в каком они написаны;
* если в выражение без скобок входят не только действия сложения и вычитания, но и умножения, деления и остатка от деления, или оба этих действия, то сначала выполняется умножениея, деление и остаток от деления, а затем сложение и вычитание;
* если в выражении имеются скобки, то сначала вычисляют значение выражений в скобках.

1.13 Выражения и их вычисления

В языке KTI-2023 допустимы выражения как с целочисленными, так и со строковыми литералами. При этом в выражениях со строковыми операндами разрешается использовать только оператор “+”, помимо этого такое выражение не должно содержать более двух операндов.

Выражения языка KTI-2023 подчиняются следующим правилам:

* выражение, как и любая инструкция языка, должна заканчиваться сепаратором “;”;
* использование в одном выражении операндов разных типов недопустимо;
* присваивать идентификатору результат выражения разрешено только если их типы данных совпадают;
* разрешено изменять приоритет операций, используя оператор “()”;
* недопустимо использование двух арифметических операторов подряд;
* разрешено использовать в качестве операнда вызов функции.

1.14 Конструкции языка

Ключевые конструкции языка в программировании представляют собой основные строительные блоки для создания программ [2].

Ключевые программные конструкции языка программирования KTI-2023 представлены в таблице 1.7.

Таблица 1.7 - Программные конструкции языка KTI-2023

|  |  |
| --- | --- |
| Конструкция | Представление в языке |
| Главная функция | major  [  …  reout 0;  ]; |
| Функция | <тип данных> fn <идентификатор> (<тип данных> <идентификатор>, …)  [  …  reout <идентификатор/литерал>;  ]; |

Окончание таблицы 1.7

|  |  |
| --- | --- |
| Конструкция | Представление в языке |
| Функция | Функции пользователя должны быть определены вне тела главной функции. |
| Цикл | until(<идентификатор>=<литерал>;)  [  …  ];  Инструкции в теле цикла повторяются пока идентификатор не окажется равен указанному значению. |
| Строка комментария | #<литерал>; |

Таким образом, ключевые программные конструкции определяют структуру и логику выполнения программы.

1.15 Область видимости идентификаторов

Все переменные и параметры локальны и могут объявляться только в теле функции. Идентификаторы, объявленные в одной функции, недоступны любой другой.

1.16 Семантические проверки

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

Таблица 1.8 - Перечень семантических проверок

|  |  |
| --- | --- |
| Номер | Проверка |
| 1 | Наличие главной функции major – точки входа в программу |
| 2 | Количество точек входа в программу (не должно превышать 1) |
| 3 | Наличие повторного объявления идентификатора |
| 4 | Соответствие формальных и фактических параметров функции по типу и количеству |
| 5 | Совпадение типа функции и типа возвращаемого ею значения |
| 6 | Использование идентификатора до его объявления |

Окончание таблицы 1.8

|  |  |
| --- | --- |
| Номер | Проверка |
| 7 | Соответствие типа идентификатора и типа присваиваемого ему значения |
| 8 | Использование однотипных операндов в выражениях |
| 9 | Значения целочисленных литералов на допустимость (входит ли в разрешенный диапазон) |
| 10 | Допустимость операций со строками |

Следовательно, при несоблюдении хотя бы одного из правил, семантический анализатор будет сообщать об ошибке.

1.17 Распределение оперативной памяти на этапе выполнения

При трансляции кода используется две области памяти: сегмент констант и сегмент данных. Они заполняются с помощью таблицы лексем и таблицы идентификаторов, сформированных на этапе лексического анализа. В сегмент констант заносятся строковые и целочисленные литералы. В сегмент данных – переменные и параметры функций.

1.18 Стандартная библиотека и её состав

В языке KTI-2023 присутствует стандартная библиотека, которая подключается автоматически при трансляции исходного кода в язык ассемблера. Содержимое библиотеки и описание функций представлено в таблице 1.18.

Таблица 1.9 - Стандартная библиотека языка KTI-2023

|  |  |
| --- | --- |
| Функция | Характеристика |
| string joinst(string a, string b) | Строковая функция, возвращающая результат конкатенации строк а и b, записанный в строку a |
| copystr(string a, string b) | Строковая функция, возвращающая строку b, скопированную в строку a. |

Таким образом, язык предусматривает 2 вида функций: конкатенацию и копирование строк.

1.19 Ввод и вывод данных

Вывод данных осуществляется с помощью ключевого слова write. В качестве аргумента принимаются литералы, идентификаторы и выражения:

write 4;

write z;

write m+1;

1.20 Точка входа

Точкой входа является функция major. Точка входа в приложение не может отсутствовать, а также не может быть переопределена.

1.21 Препроцессор

В языке KTI-2023 не предусмотрена реализация препроцессора.

1.22 Соглашения о вызовах

При генерации кода используется соглашение о вызове stdcall. Особенности stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

1.23 Объектный код

Исходный код языка KTI-2023 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

Определено 4 типа ошибок: ошибки лексического анализа, ошибки синтаксического анализа, ошибки семантического анализа, критические системные ошибки.

Таблица 1.10 – Префиксы сообщений об ошибках

|  |  |
| --- | --- |
| Префикс | Описание |
| Syn | Найдена синтаксическая ошибка. |
| Sem | Найдена семантическая ошибка. |
| Lex | Найдена лексическая ошибка. |

Отсутствие префикса говорит о том, что найдена критическая системная ошибка.

1.25 Контрольный пример

Контрольный пример на языке KTI-2023 в приложении А.

Глава 2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

Трансляторы играют ключевую роль в разработке программного обеспечения, обеспечивая перевод программ из одного языка программирования в другой [3]. В случае языка KTI-2023, транслятор выполняет преобразование программы, написанной на KTI-2023, в программу на языке ассемблера. Компоненты, составляющие транслятор, включают лексический, синтаксический и семантический анализаторы, а также генератор кода на языке ассемблера. Их взаимодействие основывается на определенных принципах и правилах, обеспечивающих корректность и точность процесса трансляции. Принцип их взаимодействия представлен на рисунке 2.1.

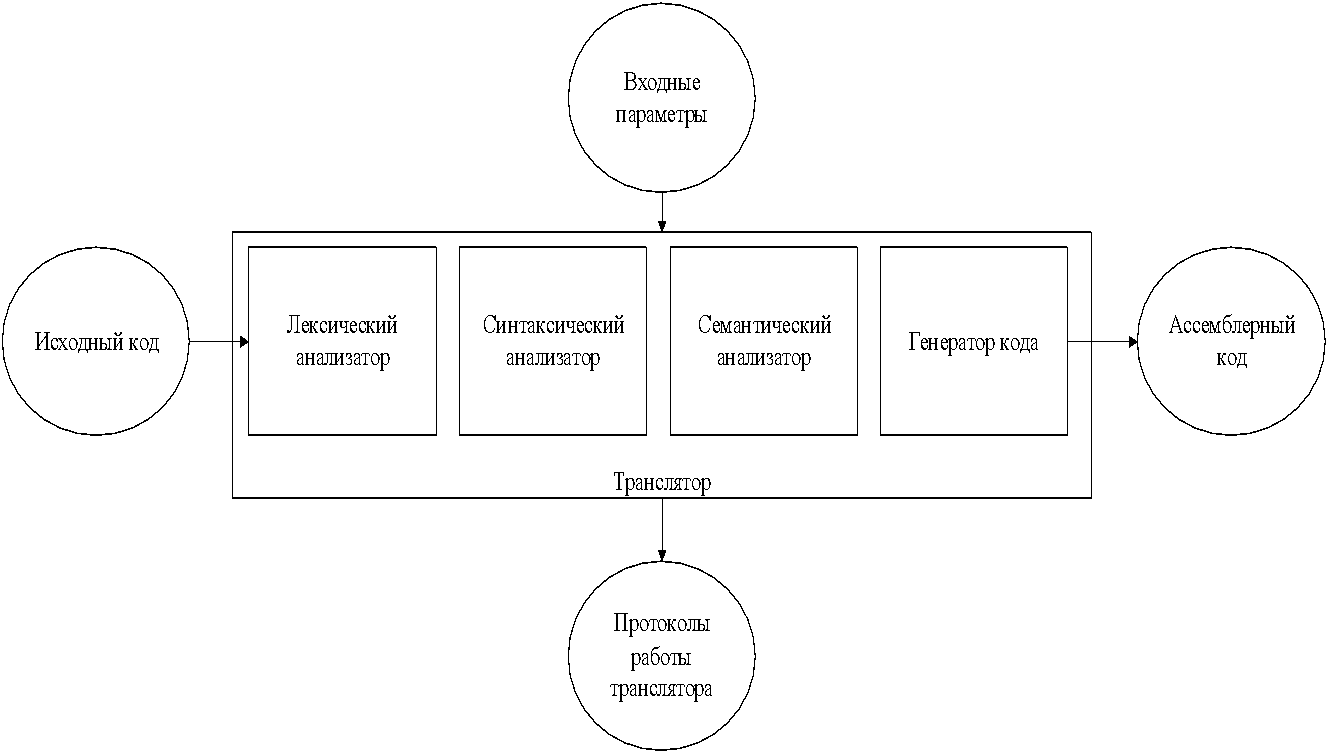


Рисунок 2.1 – Структура транслятора

Лексический анализ – первая фаза трансляции. Лексический анализатор выполняет важные функции в процессе трансляции программы. Он отвечает за обнаружение ошибок лексической структуры языка и создание таблиц лексем и идентификаторов. Подробнее описан в третьем разделе.

Синтаксический анализ является ключевой частью транслятора и выполняет задачи распознавания синтаксических конструкций и создания промежуточного представления программы. Входным параметром для синтаксического анализа является таблица лексем, полученная от лексического анализатора. Подробнее рассмотрен в четвертом разделе.

Семантический анализ выполняет важную функцию проверки исходной программы на согласованность семантики языка, то есть на соответствие правилам и определениям языка. Он осуществляет более глубокий анализ текста программы, сосредотачиваясь на его смысловом содержании. Подробное описание представлено в пятом разделе.

Генератор кода – этап транслятора, который выполняет преобразование полученных данных с предыдущих этапов трансляции в ассемблерный код. Генератор кода принимает на вход таблицы идентификаторов и лексем и, используя эти данные, генератор кода создает ассемблерные инструкции, которые соответствуют исходной программе на языке KTI-2023, пройдя через предыдущие этапы трансляции. Более полно описано в седьмом разделе.

2.2 Перечень входных параметров транслятора

В таблице 2.1 представлены входные параметры, которые могут быть использованы для управления работой транслятора.

Таблица 2.1 Входные параметры транслятора

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

Данные входные параметры транслятора используются для форматирования файлов с результатами работы лексического, синтаксического и семантического анализаторов.

2.3 Протоколы, формируемые транслятором

Таблица с перечнем протоколов, формируемых транслятором языка KTI-2023 и их назначением представлена в таблице 2.2.

Таблица 2.2 Протоколы, формируемые транслятором языка KTI-2023

|  |  |
| --- | --- |
| Протокол | Характеристика |
| Файл для вывода протокола работы программы (-log:) | Содержит таблицу лексем, таблицу идентификаторов, результат работы синтаксического анализатора, а также возможные сообщения об ошибках. |

Окончание таблицы 2.2

|  |  |
| --- | --- |
| Протокол | Характеристика |
| Выходной файл с расширением \*.asm | Содержит код на языке ассемблера, сгенерированный на основе исходного кода. |

Результатом работы транслятора языка KTI-2023 является исходный код на языке ассемблера и протокол работы транслятора, содержащий основную информацию о процессе обработки исходного кода.

Глава 3 Разработка лексического анализатора

3.1 Структура лексического анализатора

Лексический анализатор – часть транслятора, выполняющая лексический анализ. Структура лексического анализатора представлена на рисунке 3.1

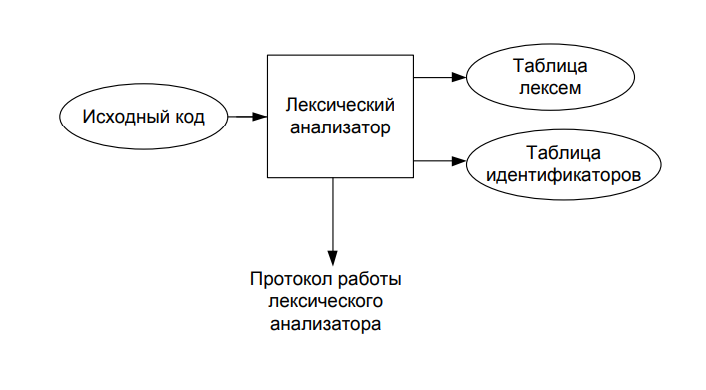


Рисунок 3.1 Структура лексического анализатора KTI-2023

Входными данными для лексического анализатора является исходный код на языке KTI-2023. Таблица лексем и таблица идентификаторов, сформированные в результате лексического анализа, являются результатом работы лексического анализатора.

3.2 Входные и выходные данные лексического анализатора

Лексический анализатор является компонентом компилятора или интерпретатора, который преобразует входной текст программы (обычно в виде последовательности символов) в последовательность лексем (токенов) для дальнейшей обработки [4]. Лексемы представляют собой минимальные значимые единицы программы, такие как идентификаторы, числа, операторы и ключевые слова.

Входные данные для лексического анализатора – это текст программы, который может быть представлен в виде строки или последовательности символов.

Результаты работы лексического анализатора – это последовательность лексем (токенов), которые образуют входную программу.

3.3 Параметры лексического анализатора

Текст кода на языке KTI-2023 подается на вход. Параметры не определяют режим работы лексического анализатора.

3.4 Алгоритм лексического анализа

Лексический анализ выполняется программой (входящей в состав транслятора), называемой лексическим анализатором. Цель лексического анализа — выделение и классификация лексем в тексте исходной программы.

Алгоритм лексического анализа можно описать в несколько шагов:

1. Разделение входного текста на отдельные лексемы, запись лексем в двумерный массив.
2. Распознавание каждой строки в двумерном массиве с помощью автоматов.
3. При удачном прохождении информация заносится в таблицу лексем и таблицу идентификаторов. Возврат к шагу 2.
4. При невозможности обработать лексему выводится сообщение об ошибке, работа транслятора останавливается.
5. Переход к следующему этапу трансляции.

Алгоритм лексического анализа позволяет систематически обрабатывать и классифицировать лексемы в исходном тексте программы, обнаруживать ошибки и создавать структурированные данные для дальнейшей обработки

3.5 Контроль входных символов

Таблица для контроля входных символов представлена на рисунке 3.2

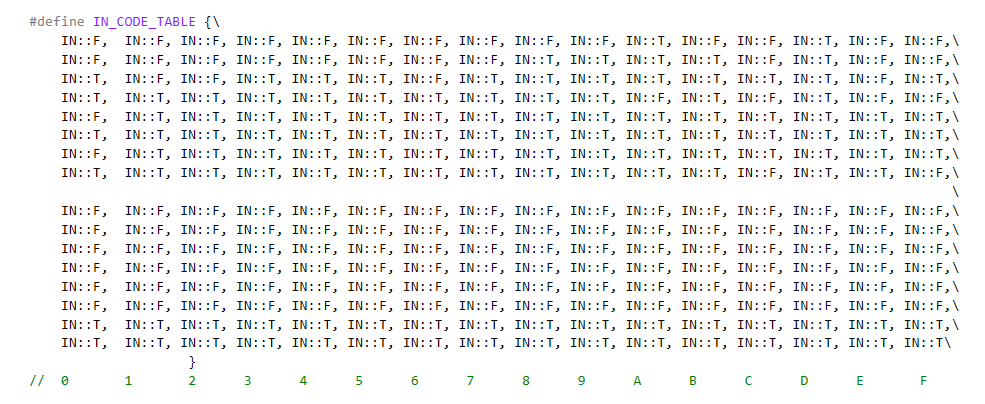
****



Рисунок 3.2 Таблица входных символов



Принцип работы таблицы заключается в соответствии значения каждого её элемента значению в таблице ASCII.

Значения символов: F – запрещенные символы, Т – разрешенные символы.

При считывании из файла символы сравниваются с символами данной таблицы. Если в таблице входной символ помечен как Т, то программа считывает его и записывает в буфер, если помечен как F – работа транслятора останавливается и выдаётся соответствующее сообщение об ошибке.

3.6 Удаление избыточных символов

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

Алгоритм удаления избыточных символов:

1. Посимвольно считываем файл с исходным кодом;
2. При нахождении избыточного символа пропускаем его, не записывая в результирующий массив;

Продолжаем посимвольное считывание файла до встречи с символом, отличным от избыточного.

3.7 Перечень ключевых слов

Таблица 3.1 Перечень ключевых слов, сепараторов, символов операций

|  |  |
| --- | --- |
| Токен | Лексема |
| int | t |
| string | t |
| fn | f |
| write | w |
| reout | r |
| until | u |
| major | m |
| var | v |
| +-\*/% | o |
| = | = |
| ( | ( |
| ) | ) |
| [ | [ |
| ] | ] |
| , | , |
| идентификатор | i |
| литерал | l |

Лексический анализатор преобразует исходный текст, заменяя лексические единицы лексемами для создания промежуточного представления исходной программы. Фрагмент графов переходов конечных автоматов, соответствующие регулярным выражениям приведены в приложении Б.

3.8 Основные структуры данных

Основные структуры таблиц лексем и идентификаторов данных языка KTI -2023, используемых для хранения, представлены в приложении В. В таблице лексем содержится исходная программа, записанная последовательностью лексем. В таблице идентификаторов содержится имя идентификатора, индекс в таблице лексем, тип данных, смысловой тип идентификатора и его значение.

3.9 Структура и перечень сообщений лексического анализатора

Перечень сообщений лексического анализатора представлен в таблице 3.2.

Таблица 3.2 Перечень сообщений лексического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 120 | Lex: превышен размер таблицы лексем. |
| 121 | Lex: таблица лексем переполнена. |
| 122 | Lex: превышен размер таблицы идентификаторов. |
| 123 | Lex: таблица идентификаторов переполнена. |
| 5 | Lex: лексема не распознана. |

Таким образом, при написании кода, достаточно просто понять, с чем конкретно связана та или иная ошибка.

3.10 Принцип обработки ошибок

В случае обнаружения ошибки, в log-файл выводится соответствующее сообщение и транслятор прекращает свою работу. Подсчет количества ошибок не ведется.

3.11 Контрольный пример

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

Глава 4 Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализ – это фаза трансляции, выполняемая после лексического анализа и предназначенная для распознавания синтаксических конструкций. Структура синтаксического анализатора представлена на рисунке 4.1.

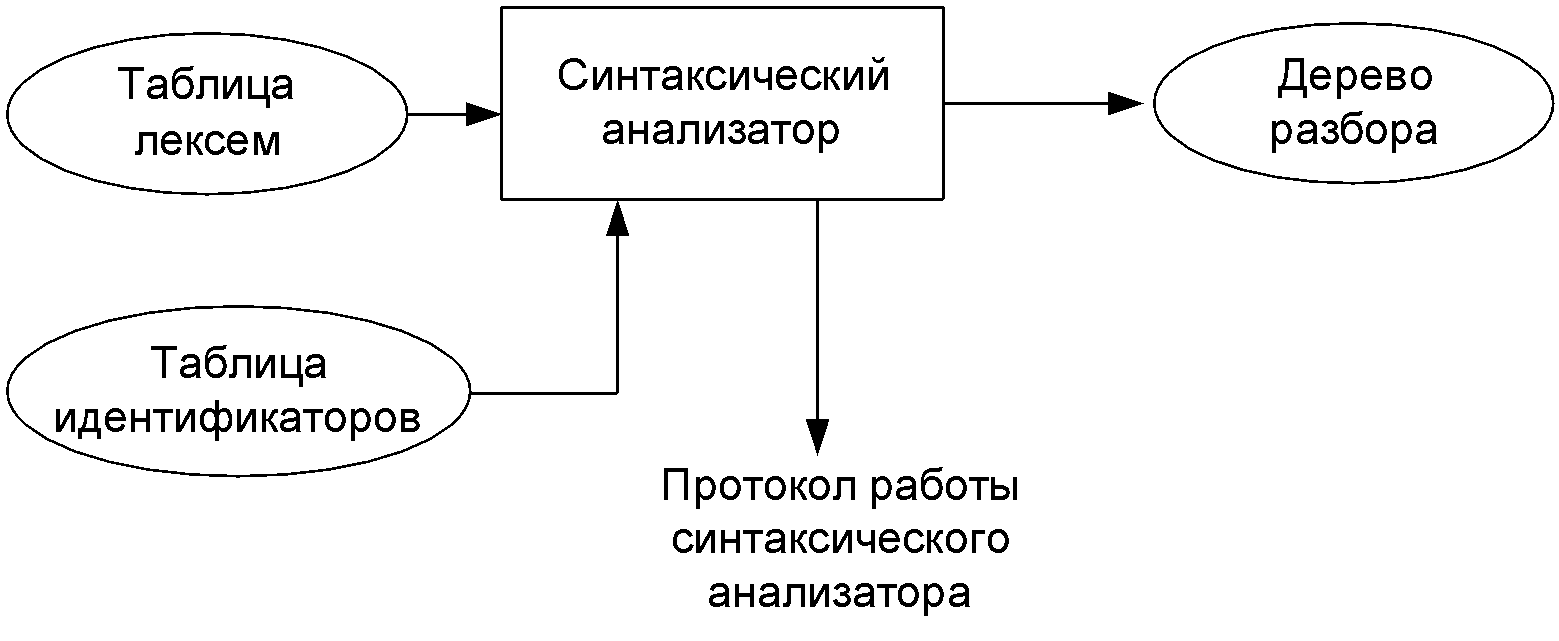


Рисунок 4.1 – Структура синтаксического анализатора

Исходными данными синтаксического анализатора являются таблицы лексем и идентификаторов, полученными на первой фазе трансляции. Результат работы синтаксического анализатора – дерево разбора (промежуточное представление кода).

4.2 Контекстно свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка KTI-2023 используется контекстно-свободная грамматика , где

T – множество терминальных символов,

N – множество нетерминальных символов,

P – множество правил языка,

S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, т.к. она не леворекурсивная (не содержит леворекурсивных правил) и правила  имеют вид:

1. , где ; (или , или );
2. , где — начальный символ, при этом если такое правило существует, то нетерминал  не встречается в правой части правил.

T – терминальные символы, которыми являются сепараторы, знаки арифметических операций и некоторые строчные буквы.

N – нетерминальные символы, представленные заглавными буквами латинского алфавита.

Таблица 4.1, описывающая правила грамматики языка KTI-2023 представлена в приложении Д.

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку, описание которой представлено в таблице 4.2. Структура данного автомата показана в приложении В.

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит является множеством терминальных и нетерминальных символов, описание которых содержится в разделе 1.2 и в таблице 4.1. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики (нетерминальный символ S) |
|  | Начальное состояние магазина автомата | Символ маркера дна стека ($) |
|  | Множество конечных состояний | Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты |

Из данной таблицы можно сделать вывод, что магазинный автомат используется для анализа и обработки языка KTI-2023 с использованием контекстно-свободной грамматики. Автомат состоит из состояний, алфавитов символов, функции переходов и имеет начальное и конечные состояния. С помощью этих компонентов автомат выполняет анализ и трансляцию программного кода на языке KTI-2023.

4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного автомата и структуру грамматики Грейбах, описывающей правила языка KTI-2023. Данные структуры представлены в приложении E.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:

1. В магазин записывается стартовый символ;
2. На основе полученных ранее таблиц формируется входная лента;
3. Запускается автомат;
4. Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке;
5. Если терминалы в стеке и в ленте совпадают, то данный терминал удаляется из ленты и стека. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другую цепочку нетерминала;
6. Если в магазине встретился нетерминал, переходим к пункту 4;
7. Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. После 3 исключений синтаксический анализатор завершает работу и генерирует последнее исключение.

4.6 Параметры синтаксического анализатора

Входным параметром синтаксического анализатора является таблица лексем, полученная на этапе лексического анализа, а также правила контекстно-свободной грамматики в форме Грейбах.

Выходными параметрами являются трассировка прохода таблицы лексем и правила разбора, которые записываются в файл протокола данного этапа обработки.

4.7 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен в таблице 4.3.

Таблица 4.3 – Перечень сообщений синтаксического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 600 | Syn: неверная структура программы |
| 601 | Syn: неверная конструкция функции |

Окончание таблицы 4.3

|  |  |
| --- | --- |
| Код | Сообщение |
| 602 | Syn: ошибка в выражении |
| 603 | Syn: ошибка в параметрах функции. |
| 604 | Syn: ошибка в параметрах вызываемой функции |
| 666 | Ошибка синтаксического анализа |

Таким образом, при написании кода, достаточно просто понять, с чем конкретно связана та или иная ошибка.

4.8 Принцип обработки ошибок

Обработка ошибок происходит следующим образом:

1. Синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.
2. Если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка, которая записывается в протокол работы и работа транслятора останавливается.

4.9 Контрольный пример

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

Глава 5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализ – третья фаза работы транслятора. Он является одной из ключевых компонент транслятора и выполняет анализ семантики программы, т.е. ее смыслового содержания. Проверяет соответствие семантических правил и выявляет ошибки, связанные с неправильным использованием переменных, типов данных, операций и других аспектов программы, которые не могут быть обнаружены только на основе лексического и синтаксического анализа.

Структура семантического анализатора представлена на рисунке 5.1.

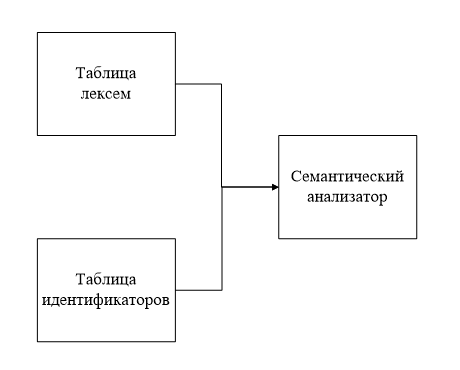


Рисунок 5.1 — структура семантического анализатора

Семантический анализатор состоит из набора функций для проверки правильности исходной программы. Часть ошибок семантического анализа обрабатываются на этапе лексического анализа. Но ошибки, требующие более сложной обработки (например, несоответствие типов операндов) вынесены в отдельный этап, следующий после синтаксического анализа.

5.2 Функции семантического анализатора

Функции семантического анализатора представлены на рисунке 5.1.

Таблица 5.1 Функции, выполняющие проверку на этапе семантического анализа

|  |  |
| --- | --- |
| Функция | Характеристика |
| CheckParm | Проверка параметров функции на совпадение по типу и количеству |
| CheckReturnType | Проверка совпадения типа функции и типа возвращаемого значения |

Окончание таблицы 5.1

|  |  |
| --- | --- |
| Функция | Характеристика |
| CheckAssignValue | Проверка соответствия типа переменной и присваиваемого значения |
| CheckAllowedValue | Проверка присваиваемых числовых значений на допустимость |
| CheckFunc | Проверка стандартных функций, на отсутствие объявления |
| CheckNativeParm | Проверка параметров функций стандартной библиотеки на совпадение по типу и количеству |

Функции семантического анализатора используются для анализа и понимания семантической структуры текста или данных.

5.3 Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены в таблице 4.3.

Таблица 4.3 – Перечень сообщений семантического анализатора

|  |  |
| --- | --- |
| Код | Сообщение |
| 2 | Sem: отсутствует точка входа major |
| 3 | Sem: допустима только 1 точка входа major |
| 4 | Sem: переменная или функция должны иметь определенный тип данных |
| 6 | Sem: повторное объявление переменной запрещено |
| 130 | Sem: количество параметров функции не совпадает |
| 131 | Sem: типы формальных и фактических параметров функции не совпадают |
| 132 | Sem: тип функции и тип возвращаемого значения не совпадают |
| 133 | Sem: недопустимый тип присваиваемого значения |
| 134 | Sem: операции над аргументами разных типов недопустимы |
| 135 | Sem: число выходит за границы допустимого диапазона. |
| 136 | Sem: недопустимые операции со строками |
| 137 | Sem: идентификатор не может быть использован до его объявления |
| 138 | Sem: стандартные функции не требуют объявления |
| 140 | Sem: много параметров. Разрешено только 9 |

Таким образом, при написании кода, достаточно просто понять, с чем конкретно связана та или иная ошибка.

5.4 Принцип обработки ошибок

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

5.5 Контрольный пример

Результат работы семантических функций описан в главе 8.

Глава 6 Вычисление выражений

6.1 Выражения, допускаемые языком

В языке KTI-2023 допустимы выражения с использованием как целочисленных, так и строковых литералов. К строковым литералам применима только операция сложения. В выражениях языка также допустимо использование вызова функции в качестве операнда. К допустимым операторам выражений относятся арифметические операторы, описанные в пункте 1.12

6.2 Польская запись и принцип ее построения

Польская запись – это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок, а также намного более простой обработки выражений впоследствии [5].

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

Алгоритм построения:

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

Таблица 6.1 – Пример преобразования выражения в обратную польскую запись

|  |  |  |
| --- | --- | --- |
| Выражение | Стек | Результат |
| (x + y)\*7 |  |  |
| x + y)\*7 | ( |  |
| + y)\*7 | ( | x |
| y)\*7 | ( + | x |
| )\*7 | ( + | x y |
| \*7 |  | x y + |
| 7 | \* | x y + |
|  | \* | x y + 7 |
|  |  | x y + 7 \* |

Таким образом, в результате преобразования все выражения языка KTI-2023 представляются в виде обратной польской записи, что позволяет выполнять их вычисление без использования скобок и с легкостью определять порядок операций.

6.3 Программная реализация обработки выражений

Программная реализация обработки выражений приведена в приложении З.

6.4 Контрольный пример

Пример преобразования выражения к польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления.

Пример преобразования выражений из контрольных примеров к обратной польской записи представлен в таблице 6.2. Преобразование выражений в формат польской записи необходимо для построения более простых алгоритмов их вычисления и преобразования к ассемблерному коду.

Таблица 6.2. Приведение выражений к ПОЛИЗ

|  |  |
| --- | --- |
| Выражение | Обратная польская запись выражения |
| i=i(l,l) | i=ill@2# |
| i=i\*(l+i)%l | i=li+i\*l% |

Таким образом, исходя из вышеприведённой таблицы, преобразование выражений в обратную польскую запись в языке KTI-2023 упрощает алгоритмы их вычисления и преобразования к ассемблерному коду.

Глава 7 Генерация кода

7.1 Структура генератора кода

Заключительным этапом трансляции языка KTI-2023 является генерация кода. Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка. На вход генератора подаются таблицы лексем и идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

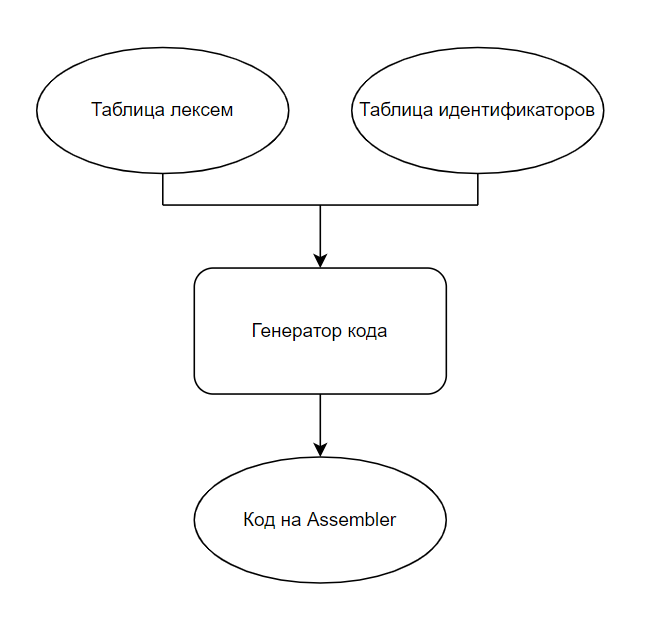


Рисунок 7.1 – Структура генератора кода

Таким образом, генератор кода выполняет не менее значимую часть компиляции.

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в разных сегментах– .data и.const. Идентификаторы размещены в сегменте данных(.data). Литералы – в сегменте констант (const). Соответствия между типами данных идентификаторов на языке KTI-2023 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка KTI-2023 и языка Ассемблера

|  |  |
| --- | --- |
| Тип идентификатора на языке KTI-2023 | Тип идентификатора на языке ассемблера |
| int | SDWORD |
| string | BYTE |

Следовательно, таблица 7.1 показывает соответствия между типами идентификаторов на языке KTI-2023 и языке ассемблера. Это важно при переводе кода с языка KTI-2023 на язык ассемблера, чтобы правильно определить типы данных и использовать соответствующие инструкции и регистры для работы с идентификаторами.

7.3 Статическая библиотека

В языке KTI-2023 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически.

Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной библиотеке реализованы функции для манипулирования выводом, недоступные пользователю. Эти функции представлены в таблице 7.2.

Таблица 7.2 - Дополнительные функции стандартной библиотеки

|  |  |
| --- | --- |
| Функция на языке С++ | Описание |
| void outint(int i) | Функции для вывода в стандартный поток значения целочисленного идентификатора/литерала. |
| void outstr(char\* s) | Функции для вывода в стандартный поток значения строкового идентификатора/литерала. |
| char\* copystr(char\* str1, char\* str2) | Функция, копирующая значение строки str2 в строку str1. |
| char\* joinst(char\* str1, char\* str2) | Строковая функция. Возвращает результат конкатенации строк str1, str2, записанный в строку str1. |

7.4 Особенности алгоритма генерации кода

В языке KTI-2023 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2.

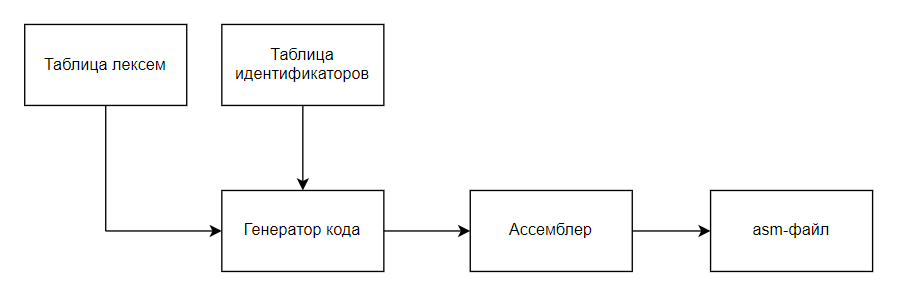


Рисунок 7.2 - Структура генератора кода

Алгоритмы генерации кода варьируются в зависимости от конкретной задачи и языка программирования.

7.5 Параметры, управляющие генерацией кода

На вход генератора кода поступают таблицы лексем и идентификаторов исходного кода программы на языке KTI-2023. Результаты работы генератора кода выводятся в файл с расширением \*.asm.

7.6 Контрольный пример

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

Глава 8 Тестирование транслятора

8.1 Общие положения

Тестирование должно покрывать как можно больше сценариев использования языка и его конструкций. Все тесты были представлены для типичных ошибок пользователей при использовании языка. Когда компилятор обнаруживает ошибку, он записывает информацию о ней в протокол, содержащий номер ошибки и диагностическое сообщение, помогающее разработчику понять причину ошибки компиляции. Результаты тестирования записываются в файл .log.

8.2 Результаты тестирования

В таблице 8.1 приведена ошибка, когда лексический анализатор не может распознать заданную лексему или токен в исходном коде программы.

Таблица 8.1 – Тестирование лексического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| major  [  #;  reout 0;  ] | "Lex: лексема не распознана. Строка 3, позиция 1." |

В таблице 8.2 приведены ошибки, когда обнаруживается нарушение синтаксических правил или ожидаемой структуры языка программирования или функции переданы неправильные параметры.

Таблица 8.2 – Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| major  [  write 1;  ]; | “Syn: неверная структура программы. Строка -1, позиция -1.” |
| int fn F(int )  [  var int k=5;  reout k;  ]; | "Syn: ошибка в параметрах функции. Строка 1, позиция -1." |

В таблице 8.3 также показаны некоторые ошибоки семантического анализатора.

Таблица 8.3 – Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| var int a;  var int a; | "Sem: повторное объявление переменной запрещено. Строка 2, позиция 3. " |

Окончание таблицы 8.3

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| string fi(string n)  [  n=’p’;  reout 7;  ]; | "Sem: тип функции и тип возвращаемого значения не совпадают. Строка 4 позиция -1." |

Таким образом данный раздел предоставляет набор тестов для проверки лексического, синтаксического и семантического анализаторов.

# Заключение

В ходе выполнения курсовой работы был разработан транслятор для языка программирования KTI-2023. Таким образом, были выполнены основные задачи данной курсовой работы:

* сформулирована спецификация языка KTI-2023;
* разработаны конечные автоматы и алгоритмы для реализация лексического анализатора;
* разработана контекстно-свободная, приведённая к ослабленной нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
* разработан семантический анализатор, осуществляющий проверку смысла используемых инструкций;
* разработан транслятор с языка программирования KTI-2023 на язык Assembler;
* проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка KTI-2023 включает:

* 2 типа данных;
* поддержку операции вывода;
* возможность вызова функций стандартной библиотеки;
* наличие 5 арифметических операторов для вычисления выражений;
* оператор цикла;

В результате работы были получены следующие результаты:

1. Было получено представление о структурах и процессах, используемых при построении трансляторов.
2. Были изучены основные различия и преимущества тех или иных средств трансляции.

Список использованных источников

1. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт.  - 3-е изд. – Москва: Вильямс, 2003. – 429 с.
2. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c.
3. Принципы работы транслятора [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/435520/>.  – Дата доступа:28.11.2023.
4. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768 с.
5. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с.

# Приложение А

int fn Func(int p, int c)

[

var int x;

var int prev=p;

var int cur=c;

var int temp;

until(x=5;)

[

temp=cur;

cur=temp+prev;

prev=temp;

x=x+1;

];

reout cur;

];

major

[

var int rez;

var int other;

var string m;

var string n;

var string q;

var string e;

var string s;

m=' chisla ';

n=' fibonachi ';

q=joinst(m,n);

write q;

rez=Func(1,1);

write rez;

n=' chislo ';

m=' other ';

q=m+n;

write q;

var int oct;

oct = Ox11;

other=oct+rez;

write other;

s=copystr(e,m);

write s;

var int bin;

bin = Bx11;

other=bin+rez;

write other;

reout 0;

];

Листинг 1 - Исходный код на языке KTI-2023

# Приложение Б

|  |
| --- |
| #define FST\_NEW LEX\_NEW, IT::NO, 4, \  ffst::NODE(1, ffst::RELATION('v', 1)),\  ffst::NODE(1, ffst::RELATION('a', 2)),\  ffst::NODE(1, ffst::RELATION('r', 3)),\  ffst::NODE()  #define FST\_UNTIL LEX\_UNTIL, IT::NO, 6, \  ffst::NODE(1, ffst::RELATION('u', 1)),\  ffst::NODE(1, ffst::RELATION('n', 2)),\  ffst::NODE(1, ffst::RELATION('t', 3)),\  ffst::NODE(1, ffst::RELATION('i', 4)),\  ffst::NODE(1, ffst::RELATION('l', 5)),\  ffst::NODE()  #define FST\_INTEGER LEX\_INTEGER, IT::INT, 4, \  ffst::NODE(1, ffst::RELATION('i', 1)),\  ffst::NODE(1, ffst::RELATION('n', 2)),\  ffst::NODE(1, ffst::RELATION('t', 3)),\  ffst::NODE()  #define FST\_STRING LEX\_STRING, IT::STR, 7, \  ffst::NODE(1, ffst::RELATION('s', 1)),\  ffst::NODE(1, ffst::RELATION('t', 2)),\  ffst::NODE(1, ffst::RELATION('r', 3)),\  ffst::NODE(1, ffst::RELATION('i', 4)),\  ffst::NODE(1, ffst::RELATION('n', 5)),\  ffst::NODE(1, ffst::RELATION('g', 6)),\  ffst::NODE()  #define FST\_FUNCTION LEX\_FUNCTION, IT::NO, 3, \  ffst::NODE(1, ffst::RELATION('f', 1)),\  ffst::NODE(1, ffst::RELATION('n', 2)),\  ffst::NODE()  #define FST\_RETURN LEX\_RETURN, IT::NO, 6, \  ffst::NODE(1, ffst::RELATION('r', 1)),\  ffst::NODE(1, ffst::RELATION('e', 2)),\  ffst::NODE(1, ffst::RELATION('o', 3)),\  ffst::NODE(1, ffst::RELATION('u', 4)),\  ffst::NODE(1, ffst::RELATION('t', 5)),\  ffst::NODE()  #define FST\_PRINT LEX\_PRINT, IT::NO, 6, \  ffst::NODE(1, ffst::RELATION('w', 1)),\  ffst::NODE(1, ffst::RELATION('r', 2)),\  ffst::NODE(1, ffst::RELATION('i', 3)),\  ffst::NODE(1, ffst::RELATION('t', 4)),\  ffst::NODE(1, ffst::RELATION('e', 5)),\  ffst::NODE()  #define FST\_HEAD LEX\_HEAD, IT::NO, 6, \  ffst::NODE(1, ffst::RELATION('m', 1)),\  ffst::NODE(1, ffst::RELATION('a', 2)),\  ffst::NODE(1, ffst::RELATION('j', 3)),\  ffst::NODE(1, ffst::RELATION('o', 4)),\  ffst::NODE(1, ffst::RELATION('r', 5)),\  ffst::NODE()  #define FST\_OCT LEX\_LITERAL, IT::INT, 4,\  ffst::NODE(1, ffst::RELATION('O', 1)),\  ffst::NODE(1, ffst::RELATION('x', 2)),\  ffst::NODE(16, ffst::RELATION('0', 2),ffst::RELATION('1', 2), ffst::RELATION('2', 2), ffst::RELATION('3', 2),\  ffst::RELATION('4', 2), ffst::RELATION('5', 2),ffst::RELATION('6', 2), ffst::RELATION('7', 2),\  ffst::RELATION('0', 3), ffst::RELATION('1', 3), ffst::RELATION('2', 3),ffst:: RELATION('3', 3),\  ffst::RELATION('4', 3), ffst::RELATION('5', 3), ffst::RELATION('6', 3), ffst::RELATION('7', 3)),\  ffst::NODE()  #define FST\_BIN LEX\_LITERAL, IT::INT, 4,\  ffst::NODE(1, ffst::RELATION('B', 1)),\  ffst::NODE(1, ffst::RELATION('x', 2)),\  ffst::NODE(4, ffst::RELATION('0', 2),ffst::RELATION('1', 2),\  ffst::RELATION('0', 3), ffst::RELATION('1', 3)),\  ffst::NODE() |

Листинг 2 – Фрагмент графов переходов конечных автоматов

# Приложение В

|  |
| --- |
| namespace LT {  struct Entry { //строка таблицы лексем  char lexema[LEXEMA\_FIXSIZE]; //лексема  int sn; //номер строки в исх. тексте  int idxTI; //индекс в таблице идентификаторов или LT\_TI\_NULLIDX  int priority; //приоритет оператора  };  struct LexTable //экземпляр таблицы лексем  {  int maxsize; //ёмкость таблицы лексем <LT\_MAXSIZE>  int size = 0; //текущий размер таблицы лексем <maxsize>  Entry\* table; //массив строк таблицы лексем  };  LexTable Create(int size);  //создать таблицу лексем (ёмкость таблицы лексем <LT\_MAXSIZE>)  void Add(LexTable& lextable, Entry entry);  //добавить строку в таблицу лексем  Entry GetEntry(LexTable& lextable, int n);  //получить строку таблицы лексем  void Delete(LexTable& lextable);  //удалить таблицу лексем  Entry WriteEntry(Entry& entry, char lexema, int indx, int line); //запролнение строки таблицы лексем |

Листинг 3 – Структура таблицы лексем

|  |
| --- |
| namespace IT  {  enum IDDATATYPE { INT = 1, STR = 2, NO = 0 };  enum IDTYPE { V = 1, F = 2, P = 3, L = 4, OP = 5 }; //типы идентификаторов: переменная, функция, параметр, литерал  struct Entry //строка таблицы идентификаторов  {  int idxfirstLE; //индекс первой строки в таблице лексем  char id[ID\_MAXSIZE]; //идентификатор  IDDATATYPE iddatatype; //тип данных  IDTYPE idtype; //тип идентификатора  union  {  char op; //значение оператора  int vint; //значение int  struct  {  int len; //количество символов в string  char str[TI\_STR\_MAXSIZE - 1]; //символы строки  } vstr[TI\_STR\_MAXSIZE]; //значение строки  }value; //значение идентификатора  };  struct IdTable //экземпляр таблицы идентификатора  {  int maxsize; //ёмкость таблицы идентификатора  int size; //текущий размер таблицы идентификаторов  Entry\* table; //строка таблицы идентификаторов  };  IdTable Create(int size);  void Add(IdTable& idtable, Entry entry);  Entry GetEntry(IdTable& idtable, int n);  int IsId(IdTable& idtable, char id[ID\_MAXSIZE]);  void Delete(IdTable& idtable); |

Листинг 4 – Структура таблицы идентификаторов

# Приложение Г

|  |
| --- |
| #pragma once  #include "Greibach.h"  #define GRB\_ERROR\_SERIES 600  #define NS(n) GRB::Rule::Chain::N(n)  #define TS(n) GRB::Rule::Chain::T(n)  #define ISNS(n) GRB::Rule::Chain::isN(n)  namespace GRB  {  Greibach greibach( //создание элемента структуры с помощью конструктора с параметрами  NS('S'), TS('$'), // стартовый символ S, дно стека $  7, // количество правил  Rule(  NS('S'), GRB\_ERROR\_SERIES + 0, // первое правило - неверная структура программы(мы проверяем если все таки неверная то потом выкидываем ошибку)  4, //  Rule::Chain(8, TS('m'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';')),  Rule::Chain(15, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(14, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';')),  Rule::Chain(9, TS('m'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), TS(';'), NS('S'))  ),  Rule(  NS('N'), GRB\_ERROR\_SERIES + 1, // конструкции в функциях  21, // количество правил  Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),  /\* Rule::Chain(4, TS('i'), TS('('), TS(')'), TS(';')),\*/  Rule::Chain(5, TS('c'), TS('i'), TS('('), TS(')'), TS(';')),  Rule::Chain(6, TS('c'), TS('i'), TS('('), TS(')'), TS(';'), NS('N')),  Rule::Chain(6, TS('c'), TS('i'), TS('('), NS('W'), TS(')'), TS(';')),  Rule::Chain(7, TS('c'), TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('N')),  /\*Rule::Chain(5, TS('i'), TS('('), TS(')'), TS(';'), NS('N')),\*/  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(9, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';'), NS('N')),  /\* Rule::Chain(8, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS(';'), NS('N')),\*/  Rule::Chain(4, TS('o'), TS('i'), TS(';'), NS('N')),  Rule::Chain(4, TS('o'), TS('l'), TS(';'), NS('N')),  Rule::Chain(4, TS('o'), NS('E'), TS(';'), NS('N')),  Rule::Chain(3, TS('o'), NS('E'), TS(';')),  Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),  Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(8, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS(';')),  Rule::Chain(7, TS('d'), TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS(';')),  Rule::Chain(3, TS('o'), TS('i'), TS(';')),  Rule::Chain(3, TS('o'), TS('l'), TS(';')),  Rule::Chain(10, TS('?'), TS('('), NS('E'), NS('I'), TS(';'), TS(')'), TS('{'), NS('N'), TS('}'), TS(';')), //оператор if  Rule::Chain(11, TS('?'), TS('('), NS('E'), NS('I'), TS(';'), TS(')'), TS('{'), NS('N'), TS('}'), TS(';'), NS('N')) //  ),  Rule(  NS('E'), GRB\_ERROR\_SERIES + 2, // ошибка в выражении  8, //  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('('), NS('E'), TS(')')),  Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),  /\*Rule::Chain(3, TS('i'), TS('('), TS(')')),\*/  Rule::Chain(2, TS('i'), NS('M')),  Rule::Chain(2, TS('l'), NS('M')),  Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M'))  /\* Rule::Chain(4, TS('i'), TS('('), TS(')'), NS('M'))\*/  ),  Rule(  NS('F'), GRB\_ERROR\_SERIES + 3, // ошибка в параметрах функции  2, //  Rule::Chain(2, TS('t'), TS('i')),  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F'))  ),  Rule(  NS('W'), GRB\_ERROR\_SERIES + 4, // ошибка в параметрах вызываемой функции  4, //  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(3, TS('i'), TS(','), NS('W')),  Rule::Chain(3, TS('l'), TS(','), NS('W'))  ),  Rule(  NS('M'), GRB\_ERROR\_SERIES + 5, // оператор  2, //  Rule::Chain(2, TS('v'), NS('E')),  Rule::Chain(3, TS('v'), NS('E'), NS('M'))  ),  Rule(  NS('I'), GRB\_ERROR\_SERIES + 6, // ошибка в условном операторе  3, //  Rule::Chain(2, TS('!'), NS('E')),  Rule::Chain(2, TS('<'), NS('E')),  Rule::Chain(2, TS('>'), NS('E'))  )  );  } |

Листинг 5 – Структура данных грамматики Грейбах

# Приложение Д

Таблица 4.1 Таблица правил переходов нетерминальных символов

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| S | S->h{NrE~}~  S->h{NrE~}~S  S-> tfi(F){NrE~}~  S->tfi(F){NrE~}~S | Стартовые правила, описывающее общую структуру программы |
| N | N->nti~  N->nti~N  N->nti=E~  N->nti=E~N  N->i=E~  N->i=E~N  N->ntfi(F)~  N->ntfi(F)~N  N->pi~  N->pi~N  N->pl~  N->pl~N  N->pE~  N->pE~N  N->rE~  N->rE~N  N->u(i=E~){N}~  N-> u(i=E~){N}~N | Правила для конструкций в функциях |
| E | E->i(F)  E->i(F)M  E->i(F)ME  E->i  E->l  E->(E)  E->i(W)  E->iM  E->lM  E->(E)M  E->i(W)M | Правила для выражений |
| F | F->i,F  F->i  F->l,F  F->l  F->ti  F->ti,F | Правила для параметров функций |

Окончание таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Символ | Правила | Какие правила порождает |
| W | W->i  W->l  W->i,W  W->l,W | Правила для параметров вызываемой функции |
| M | M->vE  M->vEM | Правила для использования операторов |

# Приложение Е

|  |
| --- |
| struct Rule //правило в гр. Грейбах  {  GRBALPHABET nn; //нетерминал (левый символ правила) <0  int iderror; //ид-р диагностического сообщения  short size; //кол-во цепочек - правых частей правила  struct Chain //цепочка - правая часть правила  {  short size; //длина цепочки  GRBALPHABET\* nt; //цепочка терминалов(>0) и нетерминалов(<0)  Chain() { size = 0; nt = 0; }  Chain(  short psize, //кол-во символов в цепочке  GRBALPHABET s, ...//символы(терминал или нетерминал)  );  char\* getCChain(char\* b); //получить правую сторону правила  static GRBALPHABET T(char t) { return GRBALPHABET(t); };//терминал  static GRBALPHABET N(char n) { return GRBALPHABET(n - 2 \* n); };//не терминал  static bool isT(GRBALPHABET s) { return s > 0; };//терминал?  static bool isN(GRBALPHABET s) { return !isT(s); };//нетерминал?  static char alphabet\_to\_char(GRBALPHABET s)  {  return isT(s) ? char(s) : char(-s);  };  }\*chains;  Rule() { nn = 0x00; size = 0; }  Rule(  GRBALPHABET pnn, //нетерминал(<0)  int iderror,  //идентификатор диагностического сообщения(Error)  short psize,  //кол-во цепочек - правых частей правила  Chain c, ...  //множество цепочек- правых частей правила  );  char\* getCRule(  //получить правило в виде N->цепочка(для распечатки)  char\* b, //буфер  short nchain  //номер цепочки(правой части) в правиле  );  short getNextChain( //получить следующую за j подходящую цепочку, вернуть ее номер или -1  GRBALPHABET t, //первый символ цепочки  Rule::Chain& pchain, //возвращаемая цепочка  short j //номер цепочки  );  };  struct Greibach  {  short size; //количество правил  GRBALPHABET startN; //стартовый символ  GRBALPHABET stbottomT; //дно стека  Rule\* rules; //множество правил  Greibach() { short size = 0; startN = 0; stbottomT = 0; rules = 0; };  Greibach(  GRBALPHABET pstartN, //стартовый символ  GRBALPHABET pstbottomT, //дно стека  short psize, //кол-во правил  Rule r, ... //правила  );  short getRule( //получить правило, возвращается номер правила или -1  GRBALPHABET pnn, //левый символ правила  Rule& prule //возвращаемое правило грамматики  );  Rule getRule(short n); //получить правило по номеру  };  Greibach getGreibach(); //получить грамматику |

Листинг 7 – Структура таблицы лексем

# Приложение Ж

0 : S->tfi(F)[NrE;];S

4 : F->ti,F

7 : F->ti

11 : N->vti;N

15 : N->vti=E;N

19 : E->i

21 : N->vti=E;N

25 : E->i

27 : N->vti;N

31 : N->u(i=E;)[N];

35 : E->l

39 : N->i=E;N

41 : E->i

43 : N->i=E;N

45 : E->iM

46 : M->oE

47 : E->i

49 : N->i=E;N

51 : E->i

53 : N->i=E;

55 : E->iM

56 : M->oE

57 : E->l

62 : E->i

66 : S->m[NrE;];

68 : N->vti;N

72 : N->vti;N

76 : N->vti;N

80 : N->vti;N

84 : N->vti;N

88 : N->vti;N

92 : N->vti;N

96 : N->i=E;N

98 : E->l

100 : N->i=E;N

102 : E->l

104 : N->i=E;N

106 : E->i(W)

108 : W->i,W

110 : W->i

113 : N->wi;N

116 : N->i=E;N

118 : E->i(W)

120 : W->l,W

122 : W->l

125 : N->wi;N

128 : N->i=E;N

130 : E->l

132 : N->i=E;N

134 : E->l

136 : N->i=E;N

138 : E->iM

139 : M->oE

140 : E->i

142 : N->wi;N

145 : N->vti;N

149 : N->i=E;N

151 : E->l

153 : N->i=E;N

155 : E->iM

156 : M->oE

157 : E->i

159 : N->wi;N

162 : N->i=E;N

164 : E->i(W)

166 : W->i,W

168 : W->i

171 : N->wi;N

174 : N->vti;N

178 : N->i=E;N

180 : E->l

182 : N->i=E;N

184 : E->iM

185 : M->oE

186 : E->i

188 : N->wE;

189 : E->i

192 : E->l

Листинг 8 – Дерево разбора

# Приложение З

|  |
| --- |
| bool PolishNotation(int i, LT::LexTable& lextable, IT::IdTable& idtable)  {  stack<LT::Entry> stack; // стек для операций  queue<LT::Entry> queue; // очередь для операндов  LT::Entry temp;  temp.idxTI = -1;  temp.lexema[0] = '#';  temp.sn = -1;  LT::Entry func;  func.lexema[0] = '@';  LT::Entry commas;  int countComma = 0; //подсчет количества запятых  char\* comma = new char[1]{ "" };  int countLex = 0; // количество преобразованных лексем  int posLex = i; // запоминаем номер лексемы перед преобразованием  bool findFunc = false;  for (i; lextable.table[i].lexema[0] != LEX\_SEPARATOR; i++, countLex++)  {  switch (lextable.table[i].lexema[0])  {  case LEX\_ID: // если идентификатор  {  queue.push(lextable.table[i]); // помещаем в очередь  continue;  }  case LEX\_LITERAL: // если литерал  {  queue.push(lextable.table[i]); // помещаем в очередь  continue;  }  case LEX\_LEFTTHESIS: // если (  {  if (idtable.table[lextable.table[i - 1].idxTI].idtype == IT::F)  findFunc = true;  stack.push(lextable.table[i]); // помещаем ее в стек  continue;  }  case LEX\_RIGHTTHESIS: // если )  {  if (findFunc)  {  itoa(++countComma, comma, 10);  strcpy(commas.lexema, comma);  stack.push(commas);  stack.push(func);  findFunc = false;  }  while (stack.top().lexema[0] != LEX\_LEFTTHESIS) // пока не встретим (  {  queue.push(stack.top()); // выталкиваем из стека в очередь  stack.pop();  if (stack.empty())  return false;  }  stack.pop(); // уничтожаем (  continue;  }  case LEX\_OPERATOR: // если знак оператора  {  while (!stack.empty() && lextable.table[i].priority <= stack.top().priority)  // пока приоритет текущего оператора  //меньше или равен приоритету оператора в вершине стека  {  queue.push(stack.top()); // выталкиваем со стека в выходную строку  stack.pop();  }  stack.push(lextable.table[i]);  continue;  }  case LEX\_COMMA:  {  countComma++;  continue;  }  }  }  while (!stack.empty()) // если стек не пустой  {  if (stack.top().lexema[0] == LEX\_LEFTTHESIS || stack.top().lexema[0] == LEX\_RIGHTTHESIS)  return false;  queue.push(stack.top()); // выталкиваем все в очередь  stack.pop();  }  while (countLex != 0) // замена текущего выражения в таблице лексем на выражение в ПОЛИЗ  {  if (!queue.empty()) {  lextable.table[posLex++] = queue.front();  //cout << lex.idtable.table[queue.front().idxTI].id; // вывод в консоль  queue.pop();  }  else  {  lextable.table[posLex++] = temp;  }  countLex--;  }  for (int i = 0; i < posLex; i++) // восстановление индексов первого вхождения  //в таблицу лексем у операторов из таблицы идентификаторов  {  if (lextable.table[i].lexema[0] == LEX\_OPERATOR || lextable.table[i].lexema[0] == LEX\_LITERAL)  idtable.table[lextable.table[i].idxTI].idxfirstLE = i;  }  return true;  }  void WritePolishNotationToFile(const char\* filename, LT::LexTable& lt) {  ofstream outfile(filename);  if (!outfile.is\_open()) {  cout << "Unable to open file for writing." << endl;  return;  }  int line = 1;  outfile << "\n";  outfile << "Таблица лексем:\n\n";  outfile << line << " ";  for (int i = 0; i < lt.size; i++)  {  if (lt.table[i].lexema[0] == ';' || lt.table[i].lexema[0] == '[')  {  outfile << lt.table[i].lexema[0];  line++;  outfile << '\n';  outfile << line << " ";  continue;  }  outfile << lt.table[i].lexema[0];  }  outfile.close();  } |

Листинг 9 – Структура таблицы лексем

# Приложение И

|  |
| --- |
| .586  .model flat, stdcall  includelib ..\Debug\StandartLib.lib  includelib kernel32.lib  includelib libucrt.lib  ExitProcess PROTO : DWORD  copystr PROTO : DWORD, : DWORD  joinst PROTO : DWORD, : DWORD  outstr PROTO : DWORD  outint PROTO : SDWORD  .stack 4096  .const  overflow db 'ERROR: VARIABLE OVERFLOW', 0  null\_division db 'ERROR: DIVISION BY ZERO', 0  L1 SDWORD 5  L2 SDWORD 1  L3 BYTE " chisla ", 0  L4 BYTE " fibonachi ", 0  L7 BYTE " chislo ", 0  L8 BYTE " other ", 0  L9 SDWORD 9  L10 SDWORD 3  L11 SDWORD 0  .data  Funcx SDWORD 0  Funcprev SDWORD 0  Funccur SDWORD 0  Functemp SDWORD 0  majorrez SDWORD 0  majorother SDWORD 0  majorm BYTE 255 DUP(0)  majorn BYTE 255 DUP(0)  majorq BYTE 255 DUP(0)  majore BYTE 255 DUP(0)  majors BYTE 255 DUP(0)  majoroct SDWORD 0  majorbin SDWORD 0  .code  Func\_proc PROC, Funcp : SDWORD, Funcc : SDWORD  push Funcp  pop Funcprev  push Funcc  pop Funccur  CYCLE:  push Funccur  pop Functemp  push Functemp  push Funcprev  pop eax  pop ebx  add eax, ebx  jo EXIT\_OVERFLOW  push eax  pop Funccur  push Functemp  pop Funcprev  push Funcx  push L2  pop eax  pop ebx  add eax, ebx  jo EXIT\_OVERFLOW  push eax  pop Funcx  cmp eax,L1  jg NEXTL  je NEXT  loop CYCLE  NEXTL:  push L1  NEXT:  push Funccur  pop eax  ret 8  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call outstr  push - 1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstr  push - 2  call ExitProcess  EXIT:  pop eax  ret 0  Func\_proc ENDP  main PROC  push offset L3  push offset majorm  call copystr  push offset L4  push offset majorn  call copystr  push offset majorm  push offset majorn  call joinst  push eax  push offset majorq  call copystr  push offset majorq  call outstr  push L2  push L2  call Func\_proc  push eax  pop majorrez  push majorrez  call outint  push offset L7  push offset majorn  call copystr  push offset L8  push offset majorm  call copystr  push offset majorm  push offset majorn  call joinst  jo EXIT\_OVERFLOW  push eax  push offset majorq  call copystr  push offset majorq  call outstr  push L9  pop majoroct  push majoroct  push majorrez  pop eax  pop ebx  add eax, ebx  jo EXIT\_OVERFLOW  push eax  pop majorother  push majorother  call outint  push offset majorm  push offset majore  call copystr  push eax  push offset majors  call copystr  push offset majors  call outstr  push L10  pop majorbin  push majorbin  push majorrez  pop eax  pop ebx  add eax, ebx  jo EXIT\_OVERFLOW  push eax  pop majorother  push majorother  call outint  push L11  push 0  call ExitProcess  jmp EXIT  EXIT\_DIV\_ON\_NULL:  push offset null\_division  call outstr  push - 1  call ExitProcess  EXIT\_OVERFLOW:  push offset overflow  call outstr  push - 2  call ExitProcess  EXIT:  push 0  call ExitProcess  main ENDP  end main |

Листинг 10 – Результат генерации кода