МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора KDA-2022»

Выполнил студент Козляковский Данила Александрович

(Ф.И.О.)

Руководитель проекта ст. пр. Наркевич Аделина Сергеевна

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

Заведующий кафедрой к. т. н., доц. Пацей Н.В.

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

Консультанты ст. пр. Наркевич Аделина Сергеевна

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

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

Нормоконтролер ст. пр. Наркевич Аделина Сергеевна

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

Курсовой проект защищен с оценкой

Содержание

[Введение 5](#_Toc90932662)

[1. Спецификация языка программирования 6](#_Toc90932663)

[1.1. Характеристика языка программирования 6](#_Toc90932664)

[1.2. Алфавит языка 6](#_Toc90932665)

[1.3. Символы сепараторы 6](#_Toc90932666)

[1.4. Применяемые кодировки 6](#_Toc90932667)

[1.5. Типы данных 7](#_Toc90932668)

[1.6. Преобразование типов данных 7](#_Toc90932669)

[1.7. Идентификаторы 7](#_Toc90932670)

[1.8. Литералы 7](#_Toc90932671)

[1.9. Область видимости идентификаторов 7](#_Toc90932672)

[1.10. Инициализация данных 8](#_Toc90932673)

[1.11. Инструкции языка 8](#_Toc90932674)

[1.12. Операции языка 8](#_Toc90932675)

[1.13. Выражения и их вычисления 8](#_Toc90932676)

[1.14. Программные конструкции языка 8](#_Toc90932677)

[1.15. Область видимости 9](#_Toc90932678)

[1.16. Семантические проверки 9](#_Toc90932679)

[1.17. Распределение оперативной памяти на этапе выполнения 9](#_Toc90932680)

[1.18. Стандартная библиотека и её состав 9](#_Toc90932681)

[1.19. Ввод и вывод данных 9](#_Toc90932682)

[1.20. Точка входа 9](#_Toc90932683)

[1.21. Препроцессор 10](#_Toc90932684)

[1.22. Соглашения о вызовах 10](#_Toc90932685)

[1.23. Объектный код 10](#_Toc90932686)

[1.24. Классификация сообщений транслятора 10](#_Toc90932687)

[1.25. Контрольный пример 10](#_Toc90932688)

[2. Структура транслятора 11](#_Toc90932689)

[2.1. Компоненты транслятора их назначение и принципы взаимодействия 11](#_Toc90932690)

[2.2. Перечень входных параметров транслятора 12](#_Toc90932691)

[2.3. Протоколы, формируемые транслятором 12](#_Toc90932692)

[3. Разработка лексического анализатора 13](#_Toc90932693)

[3.1. Структура лексического анализатора 13](#_Toc90932694)

[3.2. Контроль входных символов 13](#_Toc90932695)

[3.3. Удаление избыточных символов 14](#_Toc90932696)

[3.4. Перечень ключевых слов 14](#_Toc90932697)

[3.5. Основные структуры данных 15](#_Toc90932698)

[3.6. Структура и перечень сообщений лексического анализатора 16](#_Toc90932699)

[3.7. Принцип обработки ошибок 16](#_Toc90932700)

[3.8. Параметры лексического анализатора 16](#_Toc90932701)

[3.9. Алгоритм лексического анализа 17](#_Toc90932702)

[3.10. Контрольный пример 17](#_Toc90932703)

[4. Разработка синтаксического анализатора 18](#_Toc90932704)

[4.1. Структура синтаксического анализатора 18](#_Toc90932705)

[4.2. Контекстно-свободная грамматика, описывающая синтаксис языка 18](#_Toc90932706)

[4.3. Построение конечного магазинного автомата 19](#_Toc90932707)

[4.4. Основные структуры данных 19](#_Toc90932708)

[4.5. Описание алгоритма синтаксического разбора 20](#_Toc90932709)

[4.6. Структура и перечень сообщений синтаксического анализатора 20](#_Toc90932710)

[4.7. Параметры синтаксического анализатора и режимы его работы 20](#_Toc90932711)

[4.8. Принцип обработки ошибок 20](#_Toc90932712)

[4.9. Контрольный пример 21](#_Toc90932713)

[5. Разработка семантического анализатора 22](#_Toc90932714)

[5.1. Структура семантического анализатора 22](#_Toc90932715)

[5.2. Функции семантического анализатора 22](#_Toc90932716)

[5.3. Структура и перечень сообщений семантического анализатора 22](#_Toc90932717)

[5.4. Принцип обработки ошибок 22](#_Toc90932718)

[5.5. Контрольный пример 23](#_Toc90932719)

[6. Преобразование выражений 24](#_Toc90932720)

[6.1. Выражений, допускаемые языком 24](#_Toc90932721)

[6.2. Польская запись 24](#_Toc90932722)

[6.3. Программная реализация обработки выражений 24](#_Toc90932723)

[6.4. Контрольный пример 25](#_Toc90932724)

[7. Генерация кода 26](#_Toc90932725)

[7.1. Структура генерации кода 26](#_Toc90932726)

[7.2. Представление типов данных в оперативной памяти 26](#_Toc90932727)

[7.3. Статическая библиотека 26](#_Toc90932728)

[7.4. Особенности алгоритма генерации кода 27](#_Toc90932729)

[7.5. Входные параметры генерации кода 27](#_Toc90932730)

[7.6. Контрольный пример 27](#_Toc90932731)

[8. Тестирование транслятора 28](#_Toc90932732)

[8.1. Общие положения 28](#_Toc90932733)

[8.2. Результаты тестирования 28](#_Toc90932734)

[Приложения 29](#_Toc90932735)

[Приложение А 29](#_Toc90932736)

[Приложение Б 30](#_Toc90932737)

[Приложение В 32](#_Toc90932738)

[Приложение Г 34](#_Toc90932739)

[Приложение Д 36](#_Toc90932740)

[Заключение 38](#_Toc90932741)

[Список использованных источников 39](#_Toc90932742)

## Введение

Задачей данного курсового проекта была поставлена разработка транслятора своего языка программирования KDA-2022. Он предназначен для выполнения арифметических и сдвиговых действий над символами, числами и строками.

Главной задачей транслятора является трансляция кода на языке KDA-2022 в язык ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

– разработка спецификации языка программирования;

– разработка структуры транслятора;

– разработка лексического анализатора;

– разработка синтаксического анализатора;

– разработка семантического анализатора;

– обработка выражений с помощью обратной польской нотации;

– генерация кода в язык ассемблера;

– тестирование транслятора;

Способы решения каждой задачи будут описаны в соответствующих главах курсового проекта.

В первой главе работы определена спецификация языка программирования.

Во второй главе представлена структура транслятора. В ней перечислены компоненты транслятора, их назначения и принципы взаимодействия.

В третьей главе описана разработка лексического анализатора, который создаёт таблицы лексем и идентификаторов.

В четвертой главе описана разработка синтаксического анализатора, который выполняет разбор исходного кода в соответствии с правилами языка программирования.

В пятой главе описан семантический анализатор, показана его работа.

В шестой главе решены вопросы преобразования выражений, допускаемых языком и приведена часть протокола для контрольного примера, отображающая результаты преобразования выражений в польский формат.

В седьмой главе представлена генерация кода, где из промежуточного представления порождается код на целевом языке.

В восьмой главе описывается тестирование транслятора

## Спецификация языка программирования

## Характеристика языка программирования

Язык программирования KDA-2022 – это процедурный язык, который транслируется в язык ассемблера. Он строго типизируемый.

## Алфавит языка

Символы, используемые на этапе выполнения: [a…z], [A…Z], [0…9], символы пробела, перевода строки, спецсимволы: { }[ ] ( ) , ; + - \*.

## Символы сепараторы

Сепараторы необходимы для разделения операция языка. Сепараторы, используемые в языке программирования KDA-2022, приведены в таблице 1.1.

Таблица 1.1 – Сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| пробел | Разделитель цепочек, в именах идентификаторов и ключевых слов использовать нельзя |
| {} | Блок функции или цикла |
| () | Блок параметров функции, а так же приоритет арифметических операций |
| **+ - \*** | Арифметические операции |
| [] | Сдвиговые операции |
| ; | Разделитель программных конструкций |
| = | Оператор присваивания |
| “” | Запись строкового литерала |
| ‘’ | Запись символьного литерала |

## Применяемые кодировки

Для написания исходного кода на языке программирования KDA-2022 используется кодировка Windows-1251.

Исходный код KDA-2022 может содержать символы латинского алфавита верхнего и малого регистров, цифры десятичной системы счисления от 0 до 9, спецсимволы: [] () , ; : # + - \* !. Содержимое таблицы Windows-1251 представлено на рисунке 1.1.

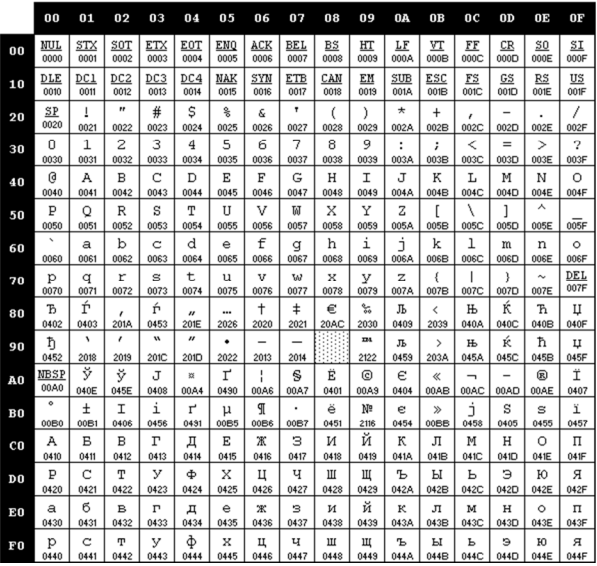


Рисунок 1.1 таблица кодировки, используемой в языке KDA-2022

## Типы данных

В языке программирования KDA-2022 используются четыре основных типа данных, которые описываются в таблице 1.2 .

Таблица 1.2 – Типы данных

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| uint | Тип данных, используемый для объявления беззнаковых целочисленных данных. Без явно указанной инициализации переменной, присваивается нулевое значение.  В шестнадцатеричном представлении числа записываются с постфиксом :h.  Поддерживает:  +(бинарный) – операция сложения;  - (бинарный) – оператор вычитания;  \* (бинарный) – оператор умножения; |
| char | Тип данных, используемый для объявления символов. Без явно указанной инициализации переменной, присваивается ноль терминатор. Данные этого типа заключаются в ‘’.  Поддерживает:  [(бинарный) – особый сдвиг влево;  ](бинарный) – особый сдвиг вправо; |
| void | Используется для указания того, что функция ничего не возвращает. Переменная не может быть типа void |
| string | Строковый тип. Предусмотрен для объявления строк. (1 символ – 1 байт).Максимальный размер строки может быть 254 символа. Данные этого типа заключаются в “”. Автоматическая инициализация: заполняются символами ”\0”.  Поддерживает:  +(бинарный) – конкатенация строк |

## Преобразование типов данных

В языке программирования KDA-2022 преобразование типов данных не поддерживается. Так как язык - строготипизированный

## Идентификаторы

Идентификаторы применяются для наименования переменных, функция и параметров. Идентификаторы, объявленные внутри функционального блока, получают область видимости, которая отображается в таблице идентификаторов. Зарезервированные идентификаторы не поддерживаются. Предусмотрены несколько правил составления идентификатора:

* состоит из символов латинского алфавита и нижнего подчеркивания;
* могут содержать цифры, но не могут начинаться с них;
* могут начинаться с символа нижнего подчеркивания;
* максимальная длина идентификатора равна 30;
* идентификатор не может совпадать с ключевыми словами языка программирования.

## Литералы

В языке программирования KDA-2022 существует 3 типа литералов: беззнаковые целые, строковые и символьные. Их краткое описание представлено в таблице 1.3.

Таблица 1.3 – Литералы

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Литералы беззнакового целого типа | Целочисленные литералы. Представляют числа в диапазоне 0 – 4294967295. При использовании числа большего максимального, число урежется до максимального(справедливо как для 16, так и для десятичной системы счисления). |
| Символьные литералы | Символ, заключенный в одинарные кавычки (‘’). |
| Строковый литерал | Символы, заключенные в двойные кавычки(“”). Максимальная длинна 251 символ. |

## Область видимости идентификаторов

Область видимости в языке программирования KDA-2022 работает по принципу («сверху вниз»). Перед использованием переменной необходимо её объявление. Допускается использование переменной только внутри её области видимости. Допускается объявление переменных с одинаковыми именами в разных программных блоках.

## Инициализация данных

Способы инициализации переменных языка программирования KDA-2022 представлены в таблице 1.4.

Таблица 1.4 – Способы инициализации переменных

|  |  |
| --- | --- |
| Вид инициализации | Примечание |
| var <тип данных> <идентификатор>; | Автоматическая инициализация переменной. uint – инициализируется нулем, char – нуль символом, string – заполняется нулями терминаторами. |
| <идентификатор> = <значение>; | Присваивание переменной значения. |

## Инструкции языка

Все возможные инструкции языка программирования KDA-2022 представлены в общем виде в таблице 1.5.

Таблица 1.5 – Инструкции языка программирования KDA-2022

|  |  |
| --- | --- |
| Инструкция | Запись |
| Объявление переменной | var <тип данных> <идентификатор>; |
| Присваивание | <идентификатор> = <литерал>|<идентификатор>; |
| Объявление функции | function <тип данных> <идентификатор> (<тип данных> <идентификатор>, …); |
| Блок инструкций | {  …  } |
| Возврат из подпрограммы | return <выражение>; |
| Вывод данных | print(<идентификатор>|<литерал>); |
| Оператор цикла | work (<количество итераций>)  {…} |

## Операции языка

Язык программирования KDA-2022 может выполнять арифметические и сдвиговые операции, представленные в таблице 1.6.

Таблица 1.6 – Операции и их приоритеты

|  |  |
| --- | --- |
| Операция | Приоритет операции/действие |
| +, - | 2/сложение, вычитание |
| \* | 3/умножение |
| [,] | 4/сдвиг влево, сдвиг вправо |

## Выражения и их вычисления

Не допускается запись двух подряд операций. Круглые скобки могут использоваться для передачи параметров функций, смены приоритета операций. Фигурные скобки содержат блоки кода функций и циклов, допускается в выражении вызов функции.

Перед генерацией кода каждое выражение приводится к записи в польской записи для дальнейшего вычисления выражения на языке ассемблера.

## Программные конструкции языка

Ключевые программные конструкции языка программирования KDA-2022 представлены в таблице 1.7

Таблица 1.7 – программные конструкции

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

## Область видимости

В языке программирования KDA-2022 переменные обязаны находиться внутри программного блока функций. Внутри разных областей видимости разрешено объявление переменных с одинаковыми именами. У литералов нет области видимости. Объявление глобальных переменных не предусмотрено. Объявление пользовательских областей видимости не предусмотрено.

## Семантические проверки

* Наличие функции main (точки входа в программу)
* Единственность точки входа
* Проверка соответствия типа функции и возвращаемого параметра
* Попытка реализовать существующую функцию
* Несоответствие типов данных
* Проверка void return
* Проверка количества параметров
* Переопределение переменных

## Распределение оперативной памяти на этапе выполнения

Все переменные размещаются в стеке. Таблицы лексем и идентификаторов размещены в структуры с выделенной под них оперативной памятью, которая очищается по окончанию работы транслятора.

## Стандартная библиотека и её состав

* Вернуть случайное uint число - casual
* Распечатать uint значение - printUint
* Распечатать string/char значение - printString
* Сделать сдвиги с char - shiftChar
* Копировать строку - stringCopy
* Конкатенация строк - strconcat
* Проверка на превышение длинны строки при конкатенации – checkstringconcat
* Перенос текста на новую строку в консоле – BREAKL
* Выдать ошибку отрицательного значения при вычислении – print\_error\_les\_zero
* Выдать ошибку на превышение длинны строки при конкатенации – print\_error\_str\_literal\_length\_exceeded

## Ввод и вывод данных

В языке программирования KDA-2022 ввод данных не поддерживается. Вывод данных происходит с помощью функции print(<идентификатор>|<литерал>);

## Точка входа

Точкой входа в языке программирования KDA-2022 является функция main.

## Препроцессор

В языке программирования KDA-2022 препроцессор не предусматривается.

## Соглашения о вызовах

В языке KDA-2022 используется соглашение о вызовах stdcall.

## Объектный код

Язык программирования KDA-2022 транслируется в язык ассемблера.

## Классификация сообщений транслятора

В случае возникновения ошибки в исходном коде программы на языке программирования KDA-2022 и выявлении её транслятором в файл протокола выводится сообщение. Классификация обрабатываемых ошибок приведена в таблице 1.10.

Таблица 1.10 – Классификация сообщений транслятора

|  |  |
| --- | --- |
| Интервал | Описание ошибок |
| 0-99 | Системные ошибки |
| 100-199 | Лексические ошибки |
| 200-299 | Синтаксические ошибки |
| 300-317 | Семантические ошибки |

## Контрольный пример

Пример программы на языке программирования KDA-2022 представлен в приложении А.

## Структура транслятора

## Компоненты транслятора их назначение и принципы взаимодействия

Транслятор, разработанный в ходе выполнения курсового проекта, представляет из себя программный продукт, который осуществляет перевод исходного кода языка KDA-2022 в язык ассемблера. Транслятор состоит из лексического, синтаксического, семантического анализаторов, преобразователя выражений в обратную польскую нотацию и генератора кода.

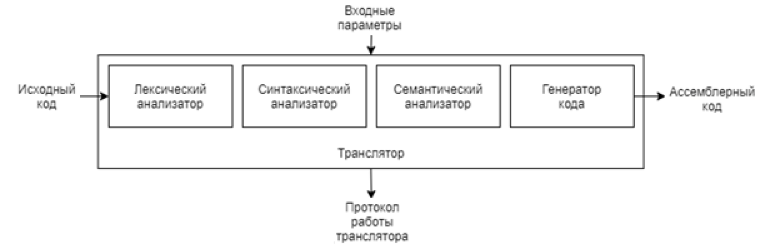


Рис. 2.1 Структура работы транслятора

Первым этапом трансляции является лексический анализ. В самом начале происходит сканирование входного файла с проверкой отдельных символов на соответствие с таблицей допустимости. Исключаются лишние пробелы и все лексемы отделяются сепараторми для дальнейшего удобного разбиения на отдельные лексемы. Затем происходит проверка на соответствие с зарезервированными конечными автоматами. После чего начинают заполнятся таблицы лексем и идентификаторов(выходные структуры данного этапа). На данном этапе присутствуют некоторые семантические проверки.

Вторая фаза ­— синтаксический анализ­. Синтаксический анализатор принимает таблицу лексем. Его задачей является распознавание синтаксических конструкций согласно правилам языка и формирование дерева разбора.

Третий этап — семантический анализ. Его задачей является проверка исходного кода на соответствие правилам семантики языка и нахождение семантических ошибок.

Четвертый этап — преобразование выражений в обратную польскую нотацию для удобной последующей записи в язык ассемблера.

Пятый этап — генерация кода. На вход принимаются таблицы лексем и идентификаторов. Происходит формирование кода на языке ассемблера, основанного на данных, полученных на предыдущих этапах.

## Перечень входных параметров транслятора

Входные параметры представлены в таблице 2.2.

Таблица 2.2 – Входные параметры транслятора языка KDA-2022

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание | Значение по умолчанию |
| -in:<имя\_файла> | Входной файл, в котором содержится исходный код на языке KDA-2022. Данный параметр должен быть указан обязательно. В случае если он не будет задан, то выполнение этапа трансляции не начнётся. | Не предусмотрено |
| -log:<имя\_файла> | Файл содержит в себе краткую информацию об исходном коде на языке KDA-2022. И в случае ошибки, показывает текст допустимой ошибки. | <имя\_файла>.log |
| -out:<имя\_файла> | В этот файл будет записан результат обработки исходного кода. Либо файл без лишних пробелов, либо файл с лексемами, разделенными сепараторми. | <имя\_файла>.out |

## Протоколы, формируемые транслятором

В процессе трансляции формируется файл протокола (по умолчанию <имя файла>.log). В файле протокола отображается время выполнения трансляции, количество проанализированных символов и строк. Так же в случае присутствия ошибок, они записываются туда.

Формируется файл asm.asm, который содержит оттранслировнный код на языке ассемблера.

## Разработка лексического анализатора

## Структура лексического анализатора

Лексический анализатор — часть транслятора, выполняющая лексический анализ. Лексический анализатор принимает разбитый на лексемы и сепараторы текст исходного кода языка KDA-2022. На выходе анализатора формируется таблица лексем и идентификаторов, а также промежуточное представление кода. Структура лексического анализатора представлен на рисунке 3.1.

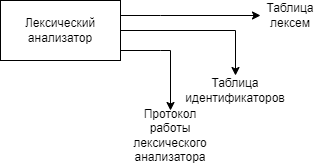


Рис. 3.1 – структура лексического анализатора

## Контроль входных символов

Контроль входных символов производится при помощи таблицы допустимых символов (рисунок 3.2).

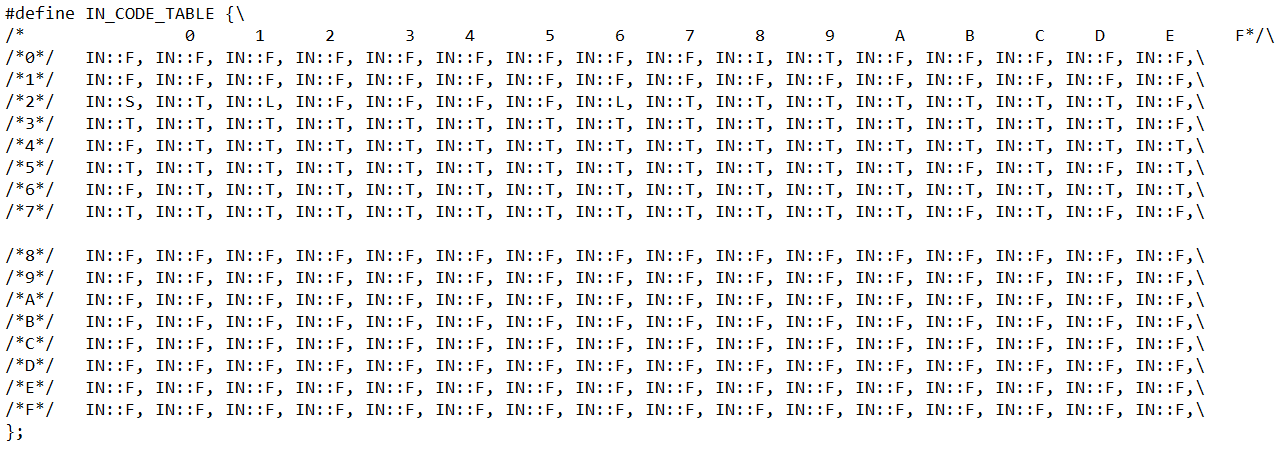


Рисунок 3.2. Таблица контроля входных символов

Принцип работы таблицы заключается в том, что каждому элементу таблицы соответствует символ таблицы ASCII. Таким образом каждому символу таблицы ставится в соответствие флаг, показывающий чем в языке является тот или иной символ.

Таблица 3.1 – Соответствие символов флагам

|  |  |
| --- | --- |
| Значение в таблице входных символов | Символы |
| Разрешенный | T |
| Запрещенный | F |
| Игнорируемый | I |
| Символьный/строковый литерал | L |
| Пробел | S |

## Удаление избыточных символов

Избыточными символами являются символы пробела и табуляции, помеченные ключами IN::S и IN::I в таблице символов.

Избыточные символы удаляются в процессе разбиения исходного кода на отдельные слова, после чего сами слова записываются в массив строк.

Встреча символа пробела или табуляции равносильны встрече символа-сепаратора однако он не заносится в таблицу лексем (исключением является встреча символьного литерала пробел).

## Перечень ключевых слов

Различные лексемы соответствуют таким составным частям языка как ключевые слова, символы операций и сепараторы. Наличие лексем нужно нам для упрощения дальнейшей обработки исходного кода программы. Соответствие представлено в таблице 3.1.

Таблица 3.1 – Соответствие лексем с ключевыми словами языка KDA-2022

|  |  |  |
| --- | --- | --- |
| Тип цепочки | Цепочка | Лексема |
| Ключевые слова | var | d |
| uint, char, string, void | t |
| main | m |
| function | f |
| return | r |
| print | p |
| BREAKL | b |
| work | w |
| Иное | Идентификатор | i |
| Литерал | l |
| Сепараторы | ; | ; |
| , | , |
| { | { |
| } | } |
| ( | ( |
| ) | ) |
| Операторы | [ | [ |
| ] | ] |
| \* | \* |
| + | + |
| - | - |

## Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и идентификаторов. Таблица лексем содержит номер лексемы, лексему, полученную при разборе, номер строки в исходном коде, приоритет(если это знак операции) и номер в таблице идентификаторов, если лексема является им.

Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype), значение (или параметры функций) (value), типы данных параметров и их количество, область видимости. Код C++ со структурой таблицы лексем представлен на листинге 3.1. Код C++ со структурой таблицы идентификаторов представлен на листинге 3.2.

|  |
| --- |
| char lexema;  int sn;  int csn;  int idxTI;  std::string tekst;  int priority; |

Листинг 3.1 – таблица лексем

|  |
| --- |
| int idxfirstLE;  std::string id = "NaN";  IDDATATYPE iddatatype;  IDTYPE idtype;  union  {  unsigned int vint;  struct  {  unsigned int len;  char str[TI\_STR\_MAXSIZE];  }vstr;  char vchar;  }value;  std::vector<IT::IDDATATYPE> funcParams;  std::string visibility; |

Листинг 3.2 – таблица идентификаторов

## Структура и перечень сообщений лексического анализатора

|  |
| --- |
| ERROR\_ENTRY(100, "{Lexis} Слишком большой размер таблицы лексем"),//LT.cpp  ERROR\_ENTRY(101, "{Lexis} Размер таблицы лексем привысел максимальный возможный"),//LT.cpp  ERROR\_ENTRY(102, "{Lexis} Слишком большой размер таблицы идентификаторов"),//IT.cpp  ERROR\_ENTRY(103, "{Lexis} Размер таблицы идентификаторов привысел максимальный возможный"),//IT.cpp  ERROR\_ENTRY(104, "{Lexis} Ошибка лексического анализа"),  ERROR\_ENTRY(105, "{Lexis} Неверный индекс таблицы идентификаторов"),//IT.cpp  ERROR\_ENTRY(106, "{Lexis} Неверный индекс таблицы лексем"),//LT.cpp  ERROR\_ENTRY(107, "{Lexis} Ошибка при создании файла вывода тиблицы идентификаторов"),//IT.cpp  ERROR\_ENTRY(108, "{Lexis} Ошибка при создании файла вывода тиблицы лексем"),//LT.cpp  ERROR\_ENTRY(109, "{Lexis} Незарезервированная лексема"),//LexicalAnalysis.cpp |

## Принцип обработки ошибок

Когда возникает ошибка – работа транслятора прекращается, а ошибка записывается в log журнал.Листинг 3.3 - сообщения лексического анализатора

## Параметры лексического анализатора

Входными параметрами лексического анализатора является структура IN, которая содержит исходный текст программы с сепарированными лексемами, а также размер выходных таблиц.

## Алгоритм лексического анализа

* проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для дальнейшего удобного разделения на лексемы;
* для выделенной части входного потока выполняется функция распознавания лексемы;
* при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;
* формирует протокол работы;
* при неуспешном распознавании выдается сообщение об ошибке.
* Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «uint» представлен на рисунке 3.2, где S0 – начальное, а S4 – конечное состояние автомата.
* Также на этом этапе присутствуют некоторые семантические проверки.



Рисунок 3.2 — Граф переходов для цепочки «uint»

## Контрольный пример

Таблицы лексем и идентификаторов представлены в приложении Б.

## Разработка синтаксического анализатора

## Структура синтаксического анализатора

Синтаксический анализатор: часть компилятора, выполняющая синтаксический анализ, то есть исходный код проверяется на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией– дерево разбора. Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.



Рис. 4.1 – структура синтаксического анализатора.

## Контекстно-свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка KDA-2022 используется контекстно-свободная грамматика , где

T – множество терминальных символов,

N – множество нетерминальных символов,

P – множество правил языка,

S – начальный символ грамматики, являющийся нетерминалом.

Грамматика языка KDA-2022 представлена в приложении B.

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

NS – нетерминальные символы, представленные несколькими заглавными буквами латинского алфавита.

Перечень правил, составляющих грамматику языка и описание нетерминальных символов KDA-2022

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочки правил | Описание |
| S | tfi(F){Nr;}S  tfi(F){Nr;}  tfi(F){NrE;}S  tfi(F){rE;}S  tfi(F){rE;}  tfi(F){NrE;} | Порождает правила, описывающее общую структуру программы |
| N | vti;N  vti;  w(i){A};N  w(i){A};  w(l){A};N  w(l){A};  i=E;N  i=E;  i(W);N  i(W);  pP;N  pP;  b;N  b; | Порождает правила, описывающие конструкции языка |
| E | iM  lM  i  l  (E)M  (E)  i(W)M  i(W)  cPM  cP | Порождает правила, описывающие выражения |
| M | +E  -E  \*E  [E  ]E  +EM  -EM  \*EM  [EM  ]EM | Порождает правила, описывающие выражения с операторами |
| F | ti  ti,F | Параметры функии |
| W | i,W  i  l | Подвыражения |
| A | i=E;A  i=E;  i(W);A  i(W);  pP;A  pP;  b;A  b; | Тело цикла |
| P | i  l | print |

## Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку. Подробное описание компонентов магазинного автомата представлено в таблице 4.2.

Таблица 4.2 – Описание компонентов магазинного автомата

|  |  |  |
| --- | --- | --- |
| Компонента | Определение | Описание |
|  | Множество состояний автомата | Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата |
|  | Алфавит входных символов | Алфавит представляет из себя множества терминальных и нетерминальных символов |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $) |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
|  | 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 |

## Основные структуры данных

Основные структуры данных синтаксического анализатора представляются в виде структуры магазинного конечного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка KDA-2022.

## 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. Иначе генерируется исключение.

## Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен на рисунке 4.3.

|  |
| --- |
| //Синтаксические  ERROR\_ENTRY(200, "{Syntax} Неверная структура программы"),  ERROR\_ENTRY(201, "{Syntax} Ошибочный оператор"),  ERROR\_ENTRY(202, "{Syntax} Ошибка в выражении"),  ERROR\_ENTRY(203, "{Syntax} Ошибка в выражении с операторами"),  ERROR\_ENTRY(204, "{Syntax} Ошибка в параметры функции"),  ERROR\_ENTRY(205, "{Syntax} Ошибка в подвыражении"),  ERROR\_ENTRY(206, "{Syntax} Ошибка в логике тела цикла"),  ERROR\_ENTRY(207, "{Syntax} Неверая структура print/casual"), |

Листинг 4.1 – сообщения синтаксического анализатора

## Параметры синтаксического анализатора и режимы его работы

Входными данными для синтаксического анализа являются таблицы лексем и идентификаторов, а также описание грамматики в форме Грейбах.

В результате успешного прохождения синтаксического анализа выводится дерево разбора, содержащее правила, по которым разобралось содержимое входных таблиц. В противном случае в консоль выводится ошибка синтаксического анализа. Также на протяжении прохождения анализа в консоль выводится трассировка анализатора.

## 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.
3. В случае нахождения ошибки, после всей процедуры трассировки в консоль будет выведено диагностическое сообщение.

## Контрольный пример

Трассировка синтаксического анализатора и дерево разбора представлены в приложении Г

## Разработка семантического анализатора

## Структура семантического анализатора

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет необходимые ошибки. Некоторые проверки (такие как переопределение идентификатора функции, идентификатор не определен, функция должна вызываться с параметром, переопределение идентификатора переменной, переменная не может быть типа void, превышена длина литерала string) осуществляются в процессе лексического анализа. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1

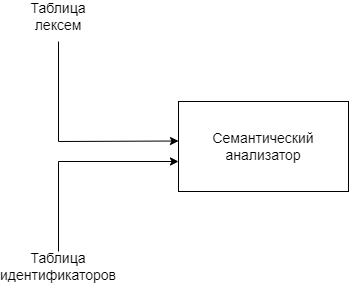


Рис. 5.1. – структура семантического анализатора

## Функции семантического анализатора

За семантический анализ отвечает функция SemAnalys. Ее входными параметрами является таблица лексем и таблица идентификаторов.

Семантический анализатор выполняет проверку на основные правила языка.

## Структура и перечень сообщений семантического анализатора

Сообщения, формируемые семантическим анализатором, представлены на листинге 5.1.

|  |
| --- |
| ERROR\_ENTRY(300, "{Semantics} Присутствует более одной точки вхождения в программу"),//SemanticsAnalysis.cpp  ERROR\_ENTRY(301, "{Semantics} Не найдено ниодной точки входа в программу"), //SemanticsAnalysis.cpp  ERROR\_ENTRY(302, "{Semantics} Переопределение идентификатора функции"), //LexicalAnalysis.cpp  ERROR\_ENTRY(303, "{Semantics} Идентификатор не определен"), //LexicalAnalysis.cpp  ERROR\_ENTRY(304, "{Semantics} Функция должна вызываться с параметрами"), //LexicalAnalysis.cpp  ERROR\_ENTRY(305, "{Semantics} Переопределение идентификатора переменной"), //LexicalAnalysis.cpp  ERROR\_ENTRY(306, "{Semantics} Переменная не может быть типа void"), //LexicalAnalysis.cpp  ERROR\_ENTRY(307, "{Semantics} Несовпадение типов"), //SemanticsAnalysis.cpp  ERROR\_ENTRY(308, "{Semantics} Недопустимый операнд в string rvalue, допустим только +"), //SemanticsAnalysis.cpp  ERROR\_ENTRY(309, "{Semantics} Недопустимая структура char rvalue"), //SemanticsAnalysis.cpp  ERROR\_ENTRY(310, "{Semantics} Несоответствие типа передаваемого параметра"), //SemanticsAnalysis.cpp  ERROR\_ENTRY(311, "{Semantics} Слишком мало аргументов в вызове функции"), //SemanticsAnalysis.cpp  ERROR\_ENTRY(312, "{Semantics} Слишком много аргументов в вызове функции"),  ERROR\_ENTRY(313, "{Semantics} Превышен максимальный лимит параметров функции(максимум допустимо 3)"),  ERROR\_ENTRY(314, "{Semantics} Неверное использование casual"),  ERROR\_ENTRY(315, "{Semantics} Превышена длинна строкового литерала(максимум 256)"),  ERROR\_ENTRY(316, "{Semantics} VOID функция не может возвращать значение"),  ERROR\_ENTRY(317, "{Semantics} Превышена максимальная длинна литерала. Максимальная длинна string литерала = 251"), |

Листинг 5.1 — перечень ошибок семантического анализатора

## Принцип обработки ошибок

Принцип обработки ошибок идентичен принципу обработки ошибок на этапе лексического анализа

## Контрольный пример

Таблица 5.1 — Ошибки семантического анализа

|  |  |
| --- | --- |
| Исходный код | Текст сообщения |
| main  {  var uint str;  var string str;  } | Ошибка 305: {Semantics} Переопределение идентификатора переменной |
| string function foo(string str)  {  return "nikita";  }  string function foo(string str1)  {  return "danila";  }  main  {  var uint str;  } | Ошибка 302: {Semantics} Переопределение идентификатора функции |
| string function foo(string str1)  {  return '1';  } | Ошибка 307: {Semantics} Несовпадение типов |
| main  {  var char symbol;  symbol = '1' ] 1;  } | Ошибка 309: {Semantics} Недопустимая структура char rvalue |
| main  {  var string str;  str = "nikita" - "danila";  } | Ошибка 308: {Semantics} Недопустимый операнд в string rvalue, допустим только + |
| void function foo(char ch)  {  var uint ui;  return ui;  }  main  {  var string str;  } | Ошибка 316: {Semantics} VOID функция не может возвращать значение |
| void function foo(char ch)  {  var uint ui;  return ui;  } | Ошибка 301: {Semantics} Не найдено ниодной точки входа в программу |
| void function foo(char ch, string str, uint number, string str2)  {  var uint ui;  return;  }  main  {  var string str;  } | Ошибка 313: {Semantics} Превышен максимальный лимит параметров функции(максимум допустимо 3) |
| uint function foo(char ch, string str, uint number)  {  var uint ui;  return 120;  }  main  {  var uint number;  number = foo('1', 1, "1");  } | Ошибка 310: {Semantics} Несоответствие типа передаваемого параметра |

## Преобразование выражений

## Выражений, допускаемые языком

В языке KDA-2022 допускаются выражения, применимые к целочисленным, строковым, символьным типам данных. В выражениях поддерживаются арифметические операции, такие как +, -, \* , [, ], (), и вызовы функций как операнды арифметических выражений.

## Польская запись

Выражения в языке KDA-2022 преобразовываются к обратной польской нотации.

Польская запись – это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок.

Обратная польская запись — это форма записи математических выражений, в которой операторы расположены после своих операндов. Выражение в обратной польской нотации читается слева направо: операция выполняется над двумя операндами, непосредственно стоящими перед знаком этой операции.

Алгоритм построения:

– исходная строка: выражение;

– результирующая строка: польская запись;

– стек: пустой;

– результирующая строка: польская запись;

– исходная строка просматривается слева направо;

– операнды переносятся в результирующую строку в порядке их следования;

– операция записывается в стек, если стек пуст или в вершине стека лежит отрывающая скобка;

– операция выталкивает все операции с большим или равным приоритетом в результирующую строку;

– запятая не помещается в стек, если в стеке операции, то все выбираются в строку;

– отрывающая скобка помещается в стек;

– закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются;

– закрывающая скобка, выталкивает все до открывающей и генерирует @ – специальный символ, в которого записывается информация о вызываемой функции, а в поле приоритета для данной лексемы записывается число параметров вызываемой функции;

– по концу разбора исходной строки все операции, оставшиеся в стеке, выталкиваются в результирующую строку.

## Программная реализация обработки выражений

Программная реализация алгоритма преобразования выражений в обратный польский формат основана функции convertToPN, которая принимает как параметр адрес таблицы лексем, таблицы идентификаторов и содержит цикл, в ходе которого перебираются все лексемы исходного кода.

## Контрольный пример

|  |
| --- |
| --------- Таблица лексем ---------  1. tf i\_0 (t i\_1 ,t i\_2 ,t i\_3 )  2. {  3. r i\_1 ;  4. }  6. tf i\_4 (t i\_5 ,t i\_6 ,t i\_7 )  7. {  8. i\_5 = l\_8 ;  10. r;  11. }  13. m  14. {  15. i\_4 ( l\_9 , l\_10 , l\_11 );  17. vt i\_12 ;  19. i\_12 = l\_13 + l\_14 \* l\_15 [ l\_16 ;  21. i\_12 = l\_17 + i\_12 [ i\_0 ( l\_18 , l\_19 , l\_20 );  23. } |

Таблица лексем до преобразования к польской нотации

|  |
| --- |
| --------- Таблица лексем ---------  1. tf i\_0 (t i\_1 ,t i\_2 ,t i\_3 )  2. {  3. r i\_1 i\_2 + i\_3 +;  4. }  6. tf i\_4 (t i\_5 )  7. {  8. vt i\_6 ;  9. vt i\_7 ;  10. i\_6 =c i\_5 ;  12. w( i\_6 )  13. {  15. i\_7 = i\_7 c i\_5 +;  16. };  18. vt i\_8 ;  19. i\_8 = l\_9 l\_10 l\_11  -1. @3  19. i\_0 ;  -1. ##  20. p i\_8 ;  22. p i\_7 ;  23. b;  25. r i\_7 ;  26. }  28. tf i\_12 (t i\_13 ,t i\_14 )  29. {  30. w( i\_14 )  31. {  32. i\_13 = i\_13 l\_15 [;  34. p i\_13 ;  35. b;  36. };  38. r;  39. }  41. m  42. {  43. vt i\_16 ;  44. i\_16 = l\_17 ;  46. i\_16 l\_18  -1. @2  46. i\_12  -1. #  46. ;  48. vt i\_19 ;  49. vt i\_20 ;  50. vt i\_21 ;  52. i\_19 = l\_22 ;  53. i\_20 = l\_23 ;  54. i\_21 = l\_24 ;  56. vt i\_25 ;  58. i\_25 = i\_19 i\_20 i\_21  -1. @3  58. i\_0 ;  -1. ##  59. p i\_25 ;  60. b;  62. vt i\_26 ;  63. i\_26 = l\_27  -1. @1  63. i\_4 ;  64. } |

Таблицы лексем после преобразования к польской нотации

## Генерация кода

## Структура генерации кода

Генерация объектного кода — это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка. На вход генератора подаются таблицы лексем и идентификаторов, а на выходе генерируется файл на языке ассемблера.



Рисунок 7.1 Структура генератора кода

## Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в разных сегментах языка ассемблера – .data и .const. Идентификаторы языка KDA-2022 размещены в сегменте данных(.data). Литералы – в сегменте констант (.const). Соответствия между типами данных идентификаторов на языке KDA-2022 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 — соответствия типов исходного языка и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке KDA-2022 | Тип идентификатора на языке ассемблера | Пояснение |
| uint | DWORD | Хранит целочисленный тип данных без знака. |
| char | BYTE | Хранит 252 байт. |
| string | BYTE | Хранит 252 байт. |

## Статическая библиотека

Функции из стандартной библиотеки содержатся в проекте StaticLibrary, в свойствах конфигурации которого указано, что это статическая библиотека.  
Подключение библиотеки происходит с помощью includelib на этапе генерации кода путем вывода специальной структуры позволяющей отслеживать при отладке пошаговый вывод в файл. С помощью оператора PROTO объявляются прототипы функций из библиотеки. Так как ассемблирование является очень формальной процедурой, то ассемблер должен знать, что представляет из себя каждый символ. Это позволяет ему генерировать правильные команды.

## Особенности алгоритма генерации кода

В языке KDA-2022 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на в приложении Д.

Генерация кода делится на 4 этапа, у меня это вызовы 4 функций:

Head, Const, Data, Code.

В Head создается стандартная шапка ассемиблеровского файла, выносятся прототипы функций из статической библиотеки, а также создаются прототипы пользовательских функций.

В Const выносятся все литералы из таблицы идентификаторов, для дальнейшего применения.

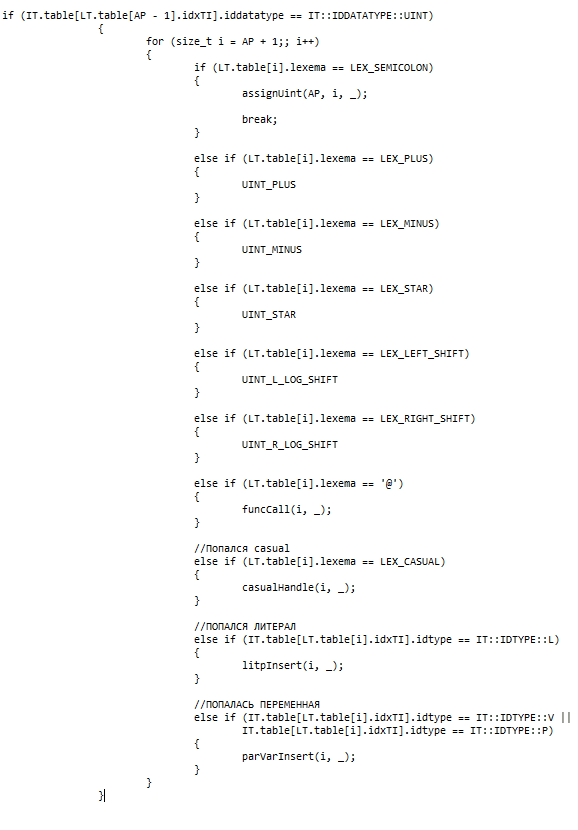
В Data выносятся все переменные и параметры функций.

И наконец в Code происходит основная работа по генерации кода.

В начале мы ищем функцию main и преобразовываем все действия в ней в ассемиблер. Когда мы встретим литерал } и после него не будет следовать ; мы выходим из цикла, так как это признак завершения самой функции.

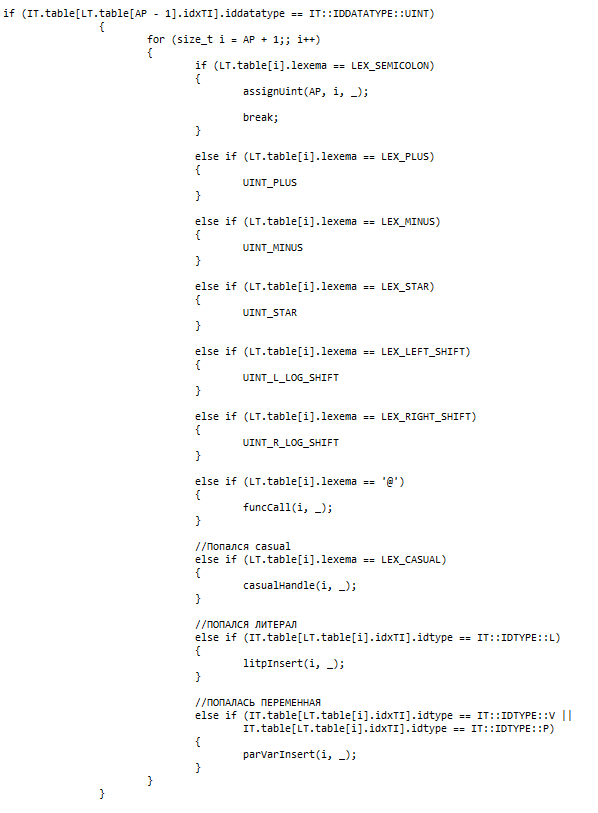


Если встретим литерал = мы понимаем, что будет какое-то выражение:



Мы должны выяснить какой в какой тип данных идет присвоение от этого зависит дальнейшая логика.

Если это uint, то:



В нем может использоваться все арифметические и сдвиговые операции, вызов пользовательских функций, вызов функции из статической библиотеке(casual). Литерал ; будет указывать, что выражение закончилось и пора его присвоить в переменную.

Вот пример команд для сложения двух операндов:

#define UINT\_PLUS \_.\_("POP"); \_.\_(" "); \_.\_("EDX\n");\

\_.\_("POP"); \_.\_(" "); \_.\_("EBX\n");\

\_.\_("ADD"); \_.\_(" "); \_.\_("EBX"); \_.\_(","); \_.\_("EDX\n");\

\_.\_("PUSH"); \_.\_(" "); \_.\_("EBX\n");

Помимо присваивания в функции могут быть также вызовы void функций, применение ключевого слова print и цикл.

Вот команды по трансляции цикла в язык ассемблера:

if (IT.table[LT.table[startCyclePosition + 2].idxTI].idtype == IT::IDTYPE::L)

{

\_.\_("MOV"); \_.\_(" "); \_.\_("ECX"); \_.\_(", "); \_.\_(IT.table[LT.table[startCyclePosition + 2].idxTI].id); \_.\_("\n");

}

else

{

\_.\_("MOV"); \_.\_(" "); \_.\_("ECX"); \_.\_(", "); \_.\_(IT.table[LT.table[startCyclePosition + 2].idxTI].visibility); \_.\_("@"); \_.\_(IT.table[LT.table[startCyclePosition + 2].idxTI].id); \_.\_("\n");

}

\_.\_("CMP"); \_.\_(" "); \_.\_("ECX"); \_.\_(","); \_.\_("0\n");

\_.\_("JE"); \_.\_(" "); \_.\_("skipcycle\n");

\_.\_("\t"); \_.\_("CYCLE:\n");

\_.\_("MOV"); \_.\_(" "); \_.\_("RemECX"); \_.\_(", "); \_.\_("ECX\n");

В логике цикла я вводили переменную для сохранения значения ECX, так как он перезаписывался в функции статической библиотеки, добавил проверку на то, что количество итераций цикла не может равняться 0, так как получился бесконечный цикл.

Есть 2 исключительных ситуации, которые обрабатываются во время выполнения кода ассемблера: при вычитании получается отрицательный результат, при конкатенации строка, результирующая строка превышает длину в 251 символ(1 байт расходуется под длину строки).

## Входные параметры генерации кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке KDA-2022. Результаты работы генератора кода выводятся в файл с расширением .asm.

## Контрольный пример

Сгенерированный файл на языке ассемблера представлен в приложении Д.

## Тестирование транслятора

## Общие положения

При возникновении ошибки на каком-либо этапе трансляции, она обрабатывается в главном файле программы: ошибка выводится на консоль или записывается в файл логирования.

В языке KDA-2022 не разрешается использовать запрещённые входным алфавитом символы. Допустимость входных символов проверяется при считывании данных из файла, если символ запрещён, то компилятор генерирует исключение.

При этапе лексического анализа, компилятор пытается распознать входящее слово. Если слово не распознано, то компилятор генерирует исключение.

Во время проверки входных данных синтаксическим анализатором, проверяется порядок использования лексем, то есть их структура. Если подходящее правило не найдено, то синтаксический анализатор диагностирует ошибку и генерирует исключение.

Так же в языке программирования KNS-2022 используется семантический анализатор. Он в свою очередь проверяет входящие данные на семантику. При неправильной семантике он также генерирует исключение.

## Результаты тестирования

Тестирование входных символов на допустимость, представлено в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main  {  var uфint a;  } | Ошибка 6: {System} Не доступный символ в исходном файле (-in)  Строка 3 позиция 15 |

Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 – Тестирование лексического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| m1ain  {  var uint ui;  print;  } | Ошибка 303: {Semantics} Идентификатор не определен  Строка 1 позиция 1 |

Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 – Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main  {  var uint ui;  print;  } | 207: строка 4, {Syntax} Неверая структура print/casual |
| main  {  var uint ui;  }  uint function foo(string str)  {  } | 200: строка 1, {Syntax} Неверная структура программы |

Итоги тестирования семантического анализатора приведены в таблице 8.4.

Таблица 8.4 – Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| uint function foo(string str, char ch)  {  return 1;  }  main  {  var uint ui;  ui = foo("aaaa");  }  … | Ошибка 311: {Semantics} Слишком мало аргументов в вызове функции  Строка 10 позиция 6 |

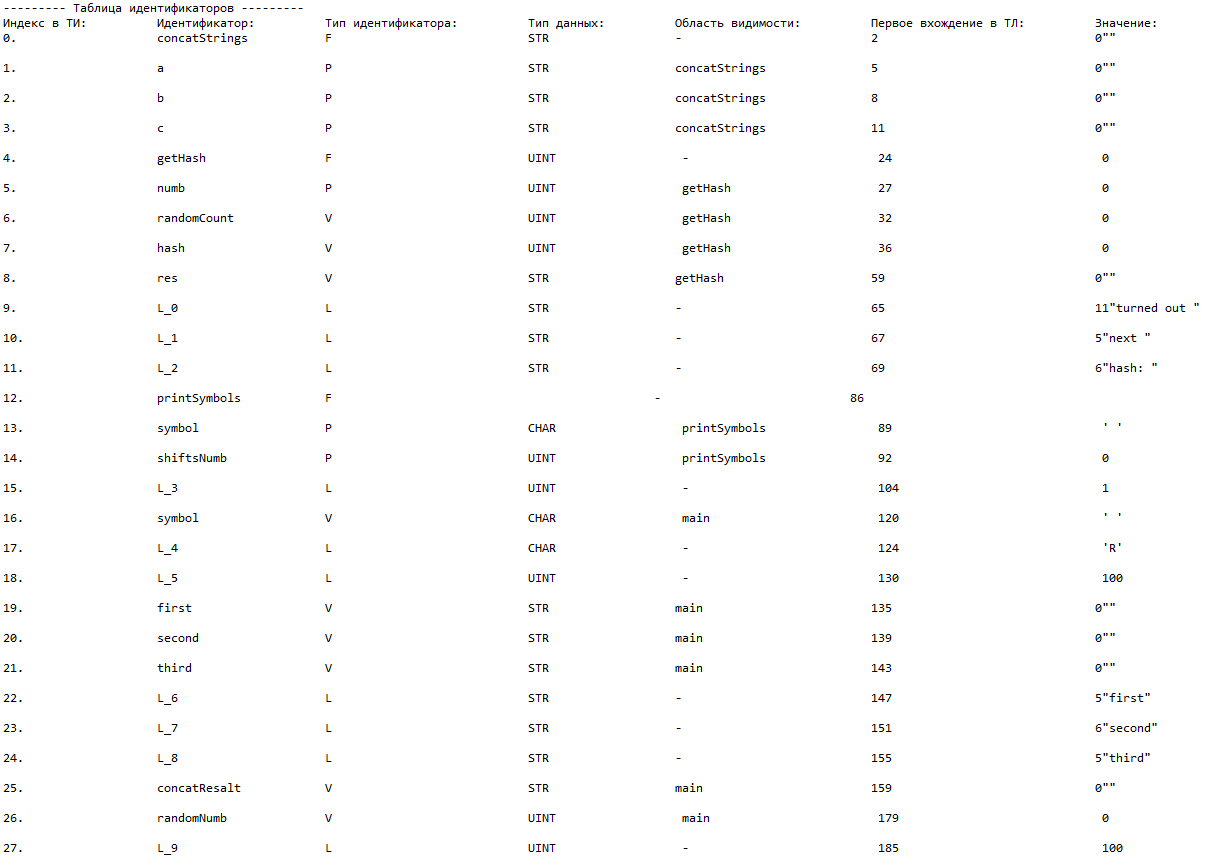
## Приложения

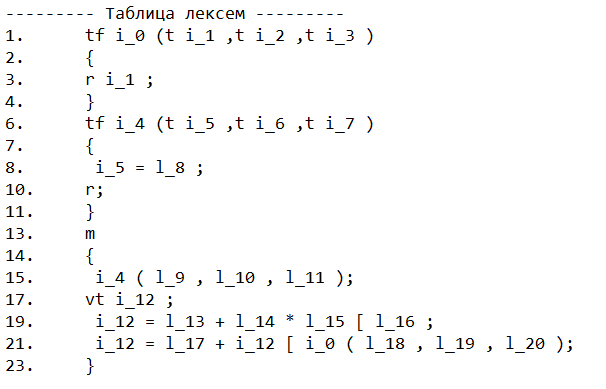
## Приложение А

|  |
| --- |
| string function concatStrings(string a, string b, string c)  {  return a + b + c;  }  uint function getHash(uint numb)  {  var uint randomCount;  var uint hash;  randomCount = casual numb;    work(randomCount)  {    hash = hash + casual numb;  };  var string res;  res = concatStrings("turned out ", "next ", "hash: ");  print res;  print hash;  BREAKL;  return hash;  }  void function printSymbols(char symbol, uint shiftsNumb)  {  work(shiftsNumb)  {  symbol = symbol [ 1;    print symbol;  BREAKL;  };    return;  }  main  {  var char symbol;  symbol = 'R';  printSymbols(symbol, 100);  var string first;  var string second;  var string third;  first = "first";  second = "second";  third = "third";  var string concatResalt;  concatResalt = concatStrings(first, second, third);  print concatResalt;  BREAKL;  var uint randomNumb;  randomNumb = getHash(100);  } |

Программа на языке KDA-2022

## Приложение Б

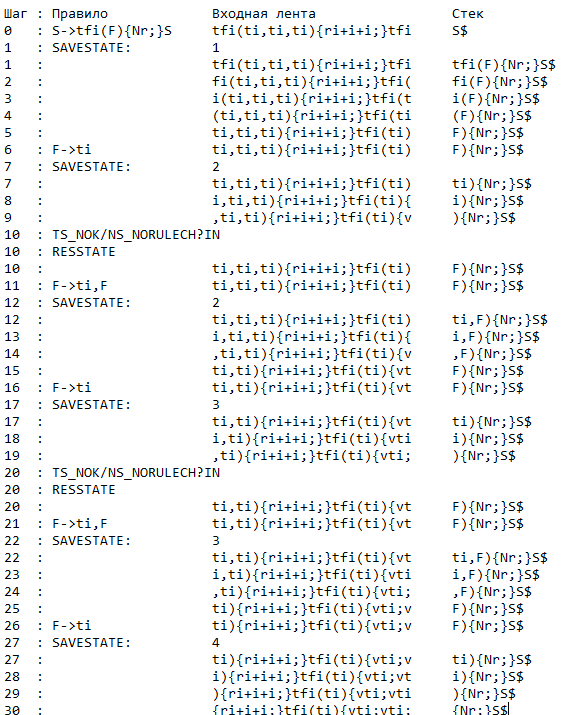
  
Таблица идентификаторов

   
Таблица лексем

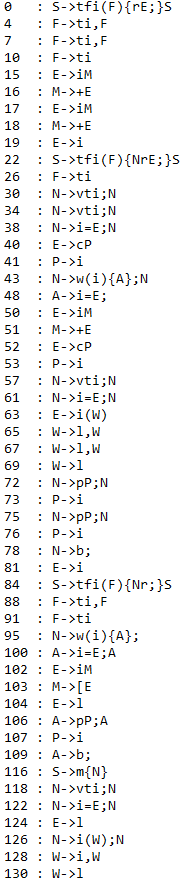
## Приложение В

|  |
| --- |
| Greibach greibach(  NS('S'),  TS('$'),  8,  Rule(NS('S'), GRB\_ERROR\_SERIES + 0, //неверная структура программы (функции и их содежимое)/////////////////////////////////////////////////////////////////////////////////////////////////////////////////  7,  //специально для void  Rule::Chain(12, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), TS(';'), TS('}'), NS('S')),  //специально для void  Rule::Chain(11, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), TS(';'), TS('}')),  Rule::Chain(13, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S')),  Rule::Chain(12, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S')),  Rule::Chain(11, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}')),    Rule::Chain(12, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}')),  Rule::Chain(4, TS('m'), TS('{'), NS('N'), TS('}'))  ),  Rule(NS('N'), GRB\_ERROR\_SERIES + 1, //Операторы программы/////////////////////////////////////////////////////////////////////////////////////////////////////////////////  14,  Rule::Chain(5, TS('v'), TS('t'), TS('i'), TS(';'), NS('N')),  Rule::Chain(4, TS('v'), TS('t'), TS('i'), TS(';')),    //цикл  Rule::Chain(9, TS('w'), TS('('), TS('i'), TS(')'), TS('{'), NS('A'), TS('}'), TS(';'), NS('N')),  Rule::Chain(8, TS('w'), TS('('), TS('i'), TS(')'), TS('{'), NS('A'), TS('}'), TS(';')),  Rule::Chain(9, TS('w'), TS('('), TS('l'), TS(')'), TS('{'), NS('A'), TS('}'), TS(';'), NS('N')),  Rule::Chain(8, TS('w'), TS('('), TS('l'), TS(')'), TS('{'), NS('A'), TS('}'), TS(';')),  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),    //вызывать просто функции  Rule::Chain(6, TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('N')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), TS(';')),  //static function  Rule::Chain(4, TS('p'), NS('P'), TS(';'), NS('N')),  Rule::Chain(3, TS('p'), NS('P'), TS(';')),  Rule::Chain(3, TS('b'), TS(';'), NS('N')),  Rule::Chain(2, TS('b'), TS(';'))  ),  Rule(NS('E'), GRB\_ERROR\_SERIES + 2, //выражение/////////////////////////////////////////////////////////////////////////////////////////////////////////////////  10,  Rule::Chain(2, TS('i'), NS('M')),  Rule::Chain(2, TS('l'), NS('M')),  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  //Траблы со сложными выражениями, где много скобок  Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(3, TS('('), NS('E'), TS(')')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M')),  Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),    Rule::Chain(3, TS('c'), NS('P'), NS('M')), //uint  Rule::Chain(2, TS('c'), NS('P'))  ),  Rule(NS('M'), GRB\_ERROR\_SERIES + 3,//выражение с операторами/////////////////////////////////////////////////////////////////////////////////////////////////////////////////  10,  Rule::Chain(2, TS('+'), NS('E')), //string, uint,  Rule::Chain(2, TS('-'), NS('E')), //uint  Rule::Chain(2, TS('\*'), NS('E')), //uint  Rule::Chain(2, TS('['), NS('E')), //uint, char  Rule::Chain(2, TS(']'), NS('E')), //uint, char  Rule::Chain(3, TS('+'), NS('E'), NS('M')),  Rule::Chain(3, TS('-'), NS('E'), NS('M')),  Rule::Chain(3, TS('\*'), NS('E'), NS('M')),  Rule::Chain(3, TS('['), NS('E'), NS('M')),  Rule::Chain(3, TS(']'), NS('E'), NS('M'))  ),  Rule(NS('F'), GRB\_ERROR\_SERIES + 4, //параметры 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 + 5, //подвыражение/////////////////////////////////////////////////////////////////////////////////////////////////////////////////  4,  Rule::Chain(3, TS('i'), TS(','), NS('W')),  Rule::Chain(3, TS('l'), TS(','), NS('W')),  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l'))  ),  Rule(NS('A'), GRB\_ERROR\_SERIES + 6, /////////////////////////////////////////////////////////////////////////////////////////////////////////////////  8,  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('A')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),  //мб еще вызывать просто функции(Это только VOID)  Rule::Chain(6, TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('A')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), TS(';')),  //static function  Rule::Chain(4, TS('p'), NS('P'), TS(';'), NS('A')),  Rule::Chain(3, TS('p'), NS('P'), TS(';')),  Rule::Chain(3, TS('b'), TS(';'), NS('A')),  Rule::Chain(2, TS('b'), TS(';'))  ),  //В print и casual могут быть только i/l  Rule(NS('P'), GRB\_ERROR\_SERIES + 7,  2,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l'))  )  ); |

## Приложение Г



Трассировка синтаксического анализатора



Дерево разбора

## Приложение Д

|  |
| --- |
| .586  .MODEL FLAT, STDCALL  includelib libucrt.lib  includelib kernel32.lib  includelib "C:\2курс, 1 сем\КПО\20\Lexer\_2\Lexer\Debug\StaticLibrary.lib"  ExitProcess PROTO :DWORD  casual PROTO : DWORD  printUint PROTO : DWORD  printString PROTO : DWORD  shiftChar PROTO : DWORD, : DWORD, : DWORD  stringCopy PROTO : DWORD, : DWORD  strconcat PROTO : DWORD, :DWORD  STRLEN PROTO : DWORD  BREAKL PROTO  print\_error\_les\_zero PROTO  checkstringconcat PROTO : DWORD, :DWORD  print\_error\_str\_literal\_length\_exceeded PROTO  concatStrings PROTO : DWORD, : DWORD, : DWORD  getHash PROTO : DWORD  printSymbols PROTO : DWORD, : DWORD  .STACK 4096  .CONST  L\_0 BYTE 11, "turned out "  L\_1 BYTE 5, "next "  L\_2 BYTE 6, "hash: "  L\_3 DWORD 1  L\_4 BYTE 1,'R'  L\_5 DWORD 100  L\_6 BYTE 5, "first"  L\_7 BYTE 6, "second"  L\_8 BYTE 5, "third"  L\_9 DWORD 100  ;Константы для предачи в shiftChar  charLeftShift BYTE 1,'['  charRightShift BYTE 1,']'  .data  concatStrings@a BYTE 256 DUP(0)  concatStrings@b BYTE 256 DUP(0)  concatStrings@c BYTE 256 DUP(0)  getHash@numb DWORD 0  getHash@randomCount DWORD 0  getHash@hash DWORD 0  getHash@res BYTE 256 DUP(0)  printSymbols@symbol BYTE 256 DUP(0)  printSymbols@shiftsNumb DWORD 0  main@symbol BYTE 256 DUP(0)  main@first BYTE 256 DUP(0)  main@second BYTE 256 DUP(0)  main@third BYTE 256 DUP(0)  main@concatResalt BYTE 256 DUP(0)  main@randomNumb DWORD 0  ;Чтобы не сбивался счетчик, когда мы заходим в функцию SL  RemECX DWORD 0  .code  main PROC  ;---обработка---выражения---после---=  PUSH OFFSET L\_4  PUSH OFFSET main@symbol  CALL stringCopy  ;---вызов---void---функции---  PUSH OFFSET main@symbol  PUSH L\_5  CALL printSymbols  ;---обработка---выражения---после---=  PUSH OFFSET L\_6  PUSH OFFSET main@first  CALL stringCopy  ;---обработка---выражения---после---=  PUSH OFFSET L\_7  PUSH OFFSET main@second  CALL stringCopy  ;---обработка---выражения---после---=  PUSH OFFSET L\_8  PUSH OFFSET main@third  CALL stringCopy  ;---обработка---выражения---после---=  PUSH OFFSET main@first  PUSH OFFSET main@second  PUSH OFFSET main@third  CALL concatStrings  PUSH EAX  PUSH OFFSET main@concatResalt  CALL stringCopy  ;---Распечатка---  PUSH OFFSET main@concatResalt  CALL printString  CALL BREAKL  ;---обработка---выражения---после---=  PUSH L\_9  CALL getHash  PUSH EAX  POP EAX  MOV main@randomNumb, EAX  INVOKE ExitProcess, 0  error\_les\_zero:  CALL print\_error\_les\_zero  INVOKE ExitProcess, 1  error\_str\_literal\_length\_exceeded:  CALL print\_error\_str\_literal\_length\_exceeded  INVOKE ExitProcess, 2  main ENDP  ;-----------FUNCTION concatStrings-----------  concatStrings PROC uses EBX ECX EDX, concatStrings@c\_STRING : DWORD, concatStrings@b\_STRING : DWORD, concatStrings@a\_STRING : DWORD  PUSH concatStrings@a\_STRING  PUSH OFFSET concatStrings@a  CALL stringCopy  PUSH concatStrings@b\_STRING  PUSH OFFSET concatStrings@b  CALL stringCopy  PUSH concatStrings@c\_STRING  PUSH OFFSET concatStrings@c  CALL stringCopy  PUSH OFFSET concatStrings@a  PUSH OFFSET concatStrings@b  POP EDX  POP EBX  INVOKE strconcat,EBX, EDX  PUSH EAX  PUSH OFFSET concatStrings@c  POP EDX  POP EBX  INVOKE strconcat,EBX, EDX  PUSH EAX  POP EAX  RET  concatStrings ENDP  ;-----------FUNCTION getHash-----------  getHash PROC uses EBX ECX EDX, getHash@numb\_UINT : DWORD  MOV EAX, getHash@numb\_UINT  MOV getHash@numb, EAX  ;---обработка---выражения---после---=  PUSH getHash@numb  CALL casual  PUSH EAX  POP EAX  MOV getHash@randomCount, EAX  MOV ECX, getHash@randomCount  CMP ECX,0  JE skipcycle  CYCLE:  MOV RemECX, ECX  ;---обработка---выражения---после---=  PUSH getHash@hash  PUSH getHash@numb  CALL casual  PUSH EAX  POP EDX  POP EBX  ADD EBX,EDX  PUSH EBX  POP EAX  MOV getHash@hash, EAX  MOV ECX, RemECX  loop CYCLE  skipcycle:  ;---обработка---выражения---после---=  PUSH OFFSET L\_0  PUSH OFFSET L\_1  PUSH OFFSET L\_2  CALL concatStrings  PUSH EAX  PUSH OFFSET getHash@res  CALL stringCopy  ;---Распечатка---  PUSH OFFSET getHash@res  CALL printString  ;---Распечатка---  PUSH getHash@hash  CALL printUint  CALL BREAKL  PUSH getHash@hash  POP EAX  RET  getHash ENDP  ;-----------FUNCTION printSymbols-----------  printSymbols PROC uses EBX ECX EDX, printSymbols@shiftsNumb\_UINT : DWORD, printSymbols@symbol\_CHAR : DWORD  PUSH printSymbols@symbol\_CHAR  PUSH OFFSET printSymbols@symbol  CALL stringCopy  MOV EAX, printSymbols@shiftsNumb\_UINT  MOV printSymbols@shiftsNumb, EAX  MOV ECX, printSymbols@shiftsNumb  CMP ECX,0  JE skipcycle  CYCLE:  MOV RemECX, ECX  ;---обработка---выражения---после---=  PUSH OFFSET printSymbols@symbol  PUSH L\_3  POP EDX  POP EBX  INVOKE shiftChar,OFFSET charLeftShift, EBX, EDX  PUSH EAX  PUSH OFFSET printSymbols@symbol  CALL stringCopy  ;---Распечатка---  PUSH OFFSET printSymbols@symbol  CALL printString  CALL BREAKL  MOV ECX, RemECX  loop CYCLE  skipcycle:  RET  printSymbols ENDP  END MAIN |

Сгенерированный код на языке ассемблера

## Заключение

В ходе выполнения курсовой работы был разработан транслятор для языка программирования KDA-2022. Таким образом, были выполнены основные задачи данной курсовой работы:

* Сформулирована спецификация языка KDA-2022;
* Разработан лексический анализатор;
* Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика;
* Разработан синтаксический анализатор;
* Разработан семантический анализатор;
* Разработан транслятор кода на язык ассемблера;
* Проведено тестирование транслятора.

Окончательная версия языка KDA-2022 включает:

* 4 типа данных;
* Поддержка операторов вывода;
* Возможность вызова функций стандартной библиотеки;
* Наличие 2 сдвиговых операций и 3 арифметических для вычисления выражений;
* Поддержка функций и оператора цикла;
* Структурированная и классифицированная система для обработки ошибок пользователя.

Проделанная работа позволила получить необходимое представление о структурах и процессах, использующихся при построении трансляторов, а также основные различия и преимущества тех или иных средств трансляции. Кроме того были получены знания при работе с многофайловыми проектами.

# **Список использованных источников**

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

2. Ахо, А. Теория синтаксического анализа, перевода и компиляции /А. Ахо, Дж. Ульман. – Москва : Мир, 1998. – Т. 2 : Компиляция. - 487 с.

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

4. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М., 2006 — 1104 c.

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