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 KII-2023»

Выполнил студент         Коваль Иван Иванович           

(Ф.И.О.)

Руководитель проекта      ст.преп. Наркевич Аделина Сергеевна

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

Заведующий кафедрой   доц.Смелов Владимир Владимирович

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

Консультант         ст.преп.  Наркевич Аделина Сергеевна

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

Нормоконтролер         ст.преп. Наркевич Аделина Сергеевна

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

Курсовой проект защищен с оценкой

**Оглавление**

[1.1 Характеристика языка программирования 6](#_Toc153726441)

[1.2 Алфавит языка 6](#_Toc153726442)

[1.3 Применяемые сепараторы 6](#_Toc153726443)

[1.4 Применяемые кодировки 6](#_Toc153726444)

[1.5 Типы данных 7](#_Toc153726445)

[Таблица 1.3 - Типы данных 7](#_Toc153726446)

[1.6 Преобразование типов данных 8](#_Toc153726447)

[1.7 Идентификаторы 8](#_Toc153726448)

[1.8 Литералы 9](#_Toc153726449)

[1.9 Объявление данных 9](#_Toc153726450)

[1.10 Инициализация данных 9](#_Toc153726451)

[1.11 Инструкции языка 10](#_Toc153726452)

[1.12 Операции языка 11](#_Toc153726453)

[Побитовые операции применяются к одному биту 11](#_Toc153726454)

[1.13 Выражения и их вычисление 11](#_Toc153726455)

[1.14 Конструкции языка 11](#_Toc153726456)

[1.16 Семантические проверки 12](#_Toc153726457)

[1.18 Стандартная библиотека и её состав 12](#_Toc153726458)

[1.19 Ввод и вывод данных 13](#_Toc153726459)

[1.22 Соглашения о вызовах 13](#_Toc153726460)

[1.23 Объектный код 13](#_Toc153726461)

[1.24 Классификация сообщений транслятора 13](#_Toc153726462)

[1.25 Контрольный пример 14](#_Toc153726463)

[**2 Структура транслятора 14**](#_Toc153726464)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 14](#_Toc153726465)

[2.2 Перечень входных параметров транслятора 16](#_Toc153726466)

[2.3 Перечень протоколов, формируемых транслятором и их содержимое 16](#_Toc153726467)

[**3 Разработка лексического анализатора 18**](#_Toc153726468)

[3.1 Структура лексического анализатора 18](#_Toc153726469)

[3.2. Контроль входных символов 18](#_Toc153726470)

[3.3 Удаление избыточных символов 19](#_Toc153726471)

[3.4 Перечень ключевых слов 19](#_Toc153726472)

[Листинг 3.4 - Пример графа перехода конечного автомата 23](#_Toc153726473)

[3.5 Основные структуры данных 23](#_Toc153726474)

[3.6 Принцип обработки ошибок 24](#_Toc153726475)

[3.7 Структура и перечень сообщений лексического анализатора 24](#_Toc153726476)

[3.8 Параметры лексического анализатора 25](#_Toc153726477)

[3.9 Алгоритм лексического анализа 25](#_Toc153726478)

[**4. Разработка синтаксического анализатора 27**](#_Toc153726479)

[4.1 Структура синтаксического анализатора 27](#_Toc153726480)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 27](#_Toc153726481)

[4.3 Построение конечного магазинного автомата 27](#_Toc153726482)

[4.4 Основные структуры данных 29](#_Toc153726483)

[4.6 Структура и перечень сообщений синтаксического анализатора 30](#_Toc153726484)

[4.7. Параметры синтаксического анализатора и режимы его работы 30](#_Toc153726485)

[4.8. Принцип обработки ошибок 30](#_Toc153726486)

[4.9. Контрольный пример 30](#_Toc153726487)

[**5 Разработка семантического анализатора 31**](#_Toc153726488)

[5.1 Структура семантического анализатора 31](#_Toc153726489)

[5.2 Функции семантического анализатора 31](#_Toc153726490)

[5.3 Структура и перечень сообщений семантического анализатора 31](#_Toc153726491)

[5.4 Принцип обработки ошибок 32](#_Toc153726492)

[5.5 Контрольный пример 32](#_Toc153726493)

[**6. Вычисление выражений 34**](#_Toc153726494)

[**6.1 Выражения, допускаемые языком 34**](#_Toc153726495)

[6.2 Польская запись и принцип её построения 34](#_Toc153726496)

[6.3 Программная реализация обработки выражений 34](#_Toc153726497)

[**7. Генерация кода 35**](#_Toc153726498)

[7.1 Структура генератора кода 35](#_Toc153726499)

[7.2 Представление типов данных в оперативной памяти 35](#_Toc153726500)

[7.3 Статическая библиотека 36](#_Toc153726501)

[7.4 Особенности алгоритма генерации кода 37](#_Toc153726502)

[7.5 Входные параметры генератора кода 37](#_Toc153726503)

[7.6 Контрольный пример 37](#_Toc153726504)

[**8. Тестирование транслятора 38**](#_Toc153726505)

[8.1 Тестирование проверки на допустимость символов 38](#_Toc153726506)

[8.2 Тестирование лексического анализатора 38](#_Toc153726507)

[8.3 Тестирование синтаксического анализатора 38](#_Toc153726508)

[8.4 Тестирование семантического анализатора 39](#_Toc153726509)

[**Заключение 40**](#_Toc153726510)

[**Список использованных источников 41**](#_Toc153726511)

[**Приложение А 42**](#_Toc153726512)

[**Приложение Б 43**](#_Toc153726513)

[**Приложение В 45**](#_Toc153726514)

[**Приложение Г 51**](#_Toc153726515)

[**Приложение Д 53**](#_Toc153726516)

**Введение**

Целью курсового проектирования является разработка собственного языка программирования и транслятора, разработка которого будет осуществляться на языке С++, при этом код на языке KII-2023 будет транслироваться в язык ассемблера

Задачи курсового проектирования:

1. Разработка спецификации языка;
2. Разработка лексического анализатора;
3. Разработка синтаксического анализатора;
4. Разработка семантического анализатора;
5. Разработка генерации код

**1 Спецификация языка программирования**

## **1.1 Характеристика языка программирования**

Язык программирования KII-2023 является процедурным, строго типизированным, компилируемым.

## **1.2 Алфавит языка**

Алфавит языка программирования – набор символов, которые могут быть использованы при написании программы.

Символы, используемые на этапе выполнения: [a…z], [A…Z], [0…9], символы пробела, табуляции и перевода строки, спецсимволы: {}|[ ] ( ) , ; : + - ^ & ~ % ''.

## **1.3 Применяемые сепараторы**

Символы-сепараторы служат в качестве разделителей цепочек языка во время обработки исходного текста программы с целью разделения на токены.

Таблица 1.2 – Символы-сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| **=** | Оператор присваивания |
| пробел | Разделитель цепочек. Допускается везде кроме названий идентификаторов и ключевых слов |
| **(** … **)** | Блок параметров функции, а также приоритет арифметических операций |
| **,** | Разделитель параметров функций |
| **~,** %, & | Побитовые операции |
| **+, -,\*,/** | Арифметические операции |
| **> <** | Логические операции |
| **;** | Разделитель программных конструкций |
| {} | Блок функций, блок условной конструкции |
| # | Разделитель условной конструкции |

## **1.4 Применяемые кодировки**

Для написания программ язык KII-2023 использует кодировку Windows-1251 содержащую английский алфавит, а также некоторые специальные символы, такие как {}, [ ], ( ), , , ;, +, -, /,\*, ~, &, %, ^.

При написании программы на языке KII-2023 используется таблица символов Windows-1251.

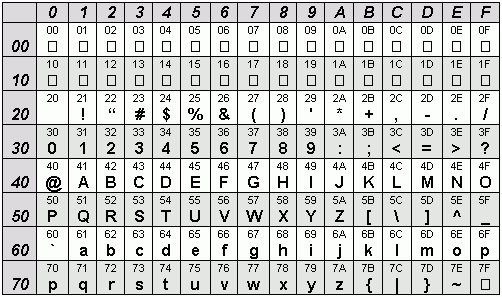


Рисунок 1.1 Алфавит языка KII-2023

## **1.5 Типы данных**

В языке KII-2023 реализованы такие типы данных как: беззнаковый целочисленный, символьный, логический, строковый.

## Таблица 1.3 - Типы данных

|  |  |
| --- | --- |
| Тип данных | Характеристика |
| Беззнаковый целочисленный тип данных uint | Основной тип данных. Используется для работы с числовыми значениями.  Поддерживаемые операции:  **+** – оператор сложения;  - – оператор вычитания;  **=** – оператор присваивания. |
| Символьный тип данных char | Основной тип данных. Используется для работы с символами.Операции над данными символьного типа: присваивание символьному идентификатору значения другого символьного идентификатора. |
| Логический тип данных bool | Основной тип данных, который может принимать только два возможных значения true и falsе.  С помощью данного типа данных можно определять какие части кода нужно выполнять. |

## **1.6 Преобразование типов данных**

В языке программирования KII-2023 присутствует преобразование строки в число с помощью функции стандартной библиотеки strton(string), которая возвращает значение uint.

## **1.7 Идентификаторы**

Идентификатор — это последовательность символов, используемая для обозначения одного из следующих элементов: имени переменной или функции. Ограничения по длине идентификатора не предусмотрена. Идентификатор должен начинаться с символа нижнего регистра, который разрешен таблицей кодировки, а также может содержать цифру.

Таблица 1.4 - Идентификаторы

|  |  |
| --- | --- |
| Верный идентификатор | Неверный идентификатор |
| let uint a = 10;  let char str1 = 't'; | new uint A = 10;  let char Str = 't'; |

## **1.8 Литералы**

Литерал - запись в исходном коде компьютерной программы, представляющая собой фиксированное значение.

Таблица 1.5 - Литералы

|  |  |
| --- | --- |
| Литералы | Пояснение |
| Логические литералы | true/false |
| Строковые литералы | Набор символов [a..z][A...Z][0...9], заключённых в двойные кавычки |

Ограничения на строковые литералы языка KII-2023: внутри литерала не допускается использование символов кириллицы.

## **1.9 Объявление данных**

Для объявления переменной используется ключевое слово new, после которого указывается тип данных и имя идентификатора.

Пример объявления числового типа с инициализацией:

new uint num = 10**;**

Пример объявления переменной символьного типа с инициализацией:

newstr **c**;

Для объявления функций используется ключевое слово **function**, перед которым указывается тип функции, а после – имя функции. Далее обязателен список параметров и тело функции.

## **1.10 Инициализация данных**

При объявлении переменной допускается инициализация данных. При этом переменной будет присвоено значение литерала или идентификатора, стоящего справа от знака равенства.

Таблица 1.6 - Пример инициализации переменных

|  |
| --- |
| new uint a = 10;  new bool b = true;  new str s = "str"; |

## **1.11 Инструкции языка**

Инструкции языка KII-2023:

Таблица 1.7 – ИнструкцииязыкаKII-2023

|  |  |
| --- | --- |
| Инструкция | Реализация |
| Объявление переменной | let <тип данных> <идентификатор>; |
| Возврат из функции | Для функций, возвращающих значение:  **result** <идентификатор/литерал>; |
| Вывод данных | print<идентификатор/литерал>; |
| Вызов функции | <идентификатор функции> <список параметров>; |
| Перевод строки | endl; |
| Присваивание | <идентификатор> **=** <выражение>;  Для целочисленного типа выражение может быть дополнено арифметическими операциями с использованием скобок. Для строкового типа выражение может быть литералом. |

## **1.12 Операции языка**

В языке KII-2023 предусмотрены операции с данными:

Таблица 1.7 – Операции языка KII-2023

|  |  |
| --- | --- |
| Тип оператора | Оператор |
| Арифметические | **+ -** сложение  - - вычитание |
| Строковые | **=** - присваивание |
| Побитовые | & - логическое и  % - логическое или  ~ - логическое не |
| Символьные | = - присваивание |

## Побитовые операции применяются к одному биту

## **1.13 Выражения и их вычисление**

Вычисление выражений – одна из важнейших задач языков программирования. Всякое выражение составляется согласно следующим правилам:

1. Допускается использовать скобки для смены приоритета операций;
2. Использование двух подряд идущих операторов не допускается;
3. Допускается использовать вызов функции, которая вычисляет и возвращает значения целочисленного типа.

## **1.14 Конструкции языка**

Программа на языке KII-2023 оформляется в виде функций пользователя и главной функции. При составлении функций рекомендуется выделять блоки и фрагменты и применять отступы для лучшей читаемости кода.

Таблица 1.8 – Программные конструкции языка KII-2023

|  |  |
| --- | --- |
| Конструкция | Реализация |
| Главная функция | **main**  **{**  …  **}** |
| Продолжение таблицы 1.8 - Программные конструкции языка KII-2023 | |  |
| Конструкция | Реализация |  |
| Внешняя функция | <тип данных> **function** <идентификатор> **(**<тип> <идентификатор>, ...**)**  {…  **result** <идентификатор/литерал>;  } |
| Условная конструкция | **if(**<идентификатор1>)  [  …  ]  else[  ...  ]  При истинности условия выполняется код внутри блока if, иначе – код внутри блока else. |

**1.15 Области видимости идентификаторов**

Переменные, объявленные в одной функции, недоступны в другой. Все объявления и операции с переменными происходят внутри какого-либо блока.

Все идентификаторы являются локальными и должны быть объявленными внутри какой-либо функции. Параметры видны только внутри функции, в которой они объявлены.

## **1.16 Семантические проверки**

В языке программирования KII-2023 выполняются следующие семантические проверки:

1. Наличие функции **main** – точки входа в программу;
2. Проверка соответствия типа функции и возвращаемого параметра;
3. Правильность передаваемых в функцию параметров: количество, типы;
4. Правильность составленного условия условного оператора.

## **1.18 Стандартная библиотека и её состав**

В языке KII-2023 присутствует стандартная библиотека, которая подключается автоматически.

Таблица 1.9 Стандартная библиотека языка KII-2023

|  |  |
| --- | --- |
| Функция | Описание |
| uint touint (<идентификатор>); | Беззнаковая целочисленная функция. Преобразует строку в число |
| bool cmp(<идентификатор>, <целочисленный литерал>); | Сравнивает значения переменной и литерала |

## **1.19 Ввод и вывод данных**

Вывод данных осуществляется с помощью оператора print. Допускается использование оператора print с литералами и идентификаторами.

**1.20 Точка входа**

В языке KII-2023 каждая программа должна содержать главную функцию **main**, с первой инструкции которой начнётся последовательное выполнение команд программы.

**1.21 Препроцессор**

Препроцессор в языке KII-2023 не предусмотрен.

## **1.22 Соглашения о вызовах**

В языке вызов функций происходит по соглашению о вызовах stdcall. Особенности stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

## **1.23 Объектный код**

Язык KII-2023 транслируется в язык ассемблера.

## **1.24 Классификация сообщений транслятора**

Генерируемые транслятором сообщения дают максимально полную информацию о допущенной пользователем ошибке при написании программы.

## **1.25 Контрольный пример**

|  |
| --- |
| bool function fa(uint a) {  let bool b;  let bool c;  b = cmp(a,0);  if (b) [  result true;  ]  else [  print 'number ';  print a,endl;  a=a-;  c = fa(a);  ]  result false;  };  main {  let uint a = 10;  a = 5~%8^(3%2)+&2;  print a, endl;  let char c = 'q';  print c , endl;  a = touint '125';  print a, endl;  let bool b;  b = fa(a);  print b, endl;  } |

# **2 Структура транслятора**

## **2.1 Компоненты транслятора, их назначение и принципы взаимодействия**

В языке KII-2023 исходный код транслируется в язык Assembler. Транслятор языка разделён на части, которые взаимодействуют между собой и выполняют отведённые им функции. Для того чтобы получить ассемблерный код, используется выходные данные работы лексического анализатора, а именно таблица лексем и таблица идентификаторов. Для указания выходных файлов используются входные параметры транслятора. Структура транслятора языка KII-2023 приведена на рисунке 1.

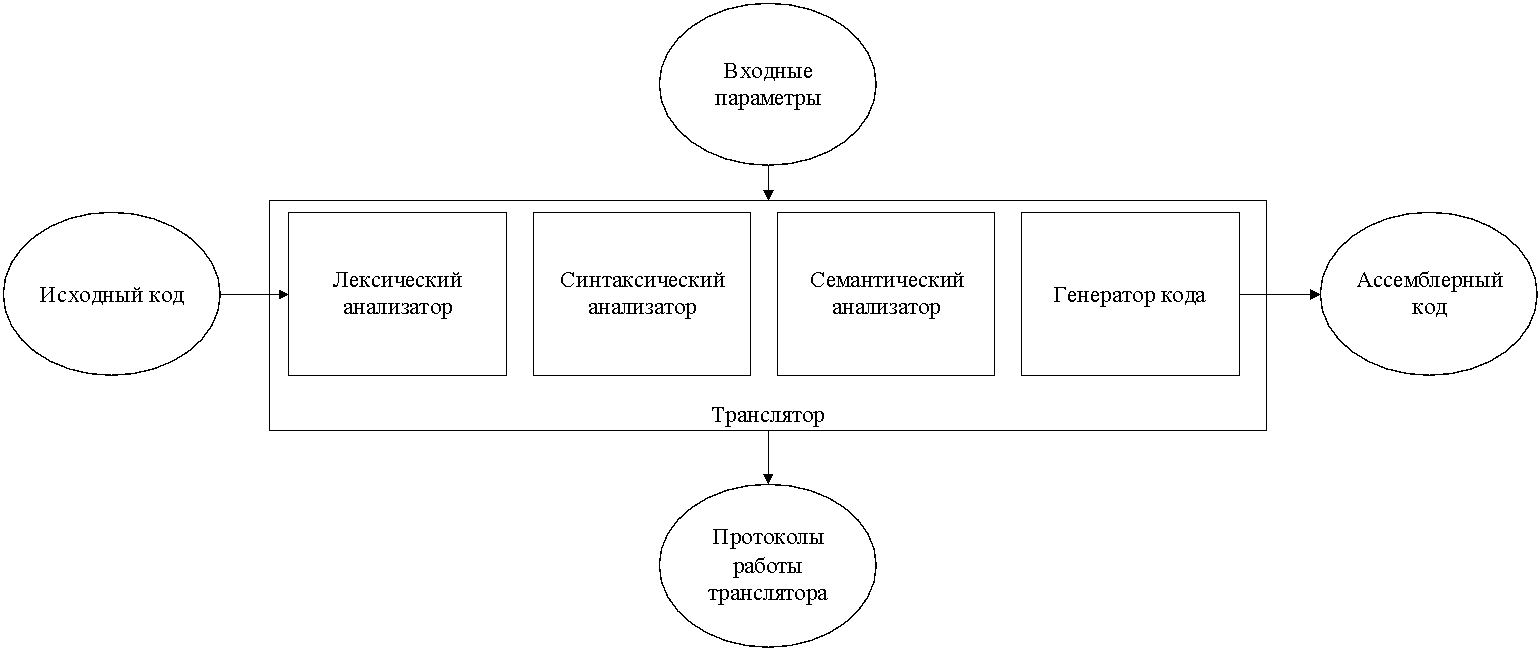


Рисунок 2.1 Структура транслятора языка программирования KII-2023

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором. На вход лексического анализатора подаётся последовательность символов входного языка. Он производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив отдельных слов. Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация. Таблица лексем и таблица идентификаторов являются входом для следующей фазы компилятора – синтаксического анализа.

Цели лексического анализатора:

− убрать все лишние пробелы;

− выполнить распознавание лексем;

− построить таблицу лексем и таблицу идентификаторов;

− при неуспешном распознавании или обнаружении некоторых ошибок во входном тексте выдать сообщение об ошибке.

Синтаксический анализатор – часть компилятора, выполняющая синтаксический анализ, то есть проверку исходного кода на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией является дерево разбора

Семантический анализатор – часть транслятора, выполняющая семантический анализ, то есть проверку исходного кода на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Генератор кода – часть транслятора, выполняющая генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

## **2.2 Перечень входных параметров транслятора**

Для формирования файлов с результатами работы лексического, синтаксического и семантического анализаторов используются входные параметры транслятора, которые приведены в таблице 2.1.

Таблица 2.1 – Входные параметры транслятора языка KII-2023

|  |  |  |
| --- | --- | --- |
| Входной параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к in-файлу> | Файл с исходным кодом на языке KII-2023 , имеющий расширение .txt | Не предусмотрено |
| -log:<путь к log-файлу> | Файл журнала для вывода протоколов работы программы. | Значение по умолчанию:  <имя in-файла>.log |

## **2.3 Перечень протоколов, формируемых транслятором и их содержимое**

В ходе работы программы формируются протокол работы лексического, синтаксического анализаторов которые содержат в себе перечень протоколов работы. В таблице 2.2 приведены протоколы, формируемые транслятором и их содержимое.

Таблица 2.2 – Протоколы, формируемые транслятором языка KII-2023

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| Файл журнала, заданный параметром "-log:" | Файл с протоколом работы транслятора языка программирования KII-2023. Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа, а также результат работы алгоритма преобразования выражений к польской записи. |
| Выходной файл, c расширением ".asm" | Результат работы программы – файл, содержащий исходный код на языке ассемблера. |

# **3 Разработка лексического анализатора**

## **3.1 Структура лексического анализатора**

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором. На вход лексического анализатора подаётся исходный код входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка. Лексический анализатор производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов.

Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация.

Функции лексического анализатора:

− удаление «пустых» символов и комментариев. Если «пустые» символы и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними;

− распознавание идентификаторов и ключевых слов;

− распознавание констант;

− распознавание разделителей и знаков операций.

Структура лексического анализатора представлена на рисунке 3.1.

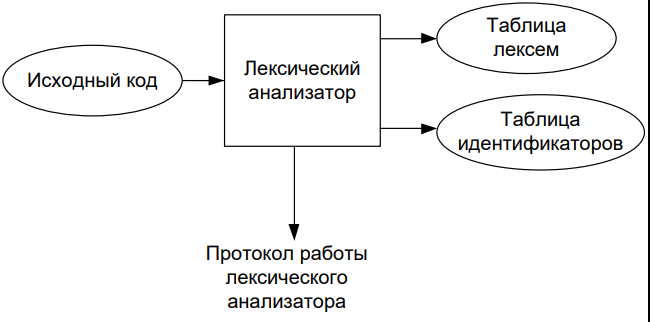


Рисунок 3.1 Структура лексического анализатора

## **3.2. Контроль входных символов**

Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. Таблица входных символов представлена на рисунке 3.2, категории входных символов представлены в таблице 3.1.

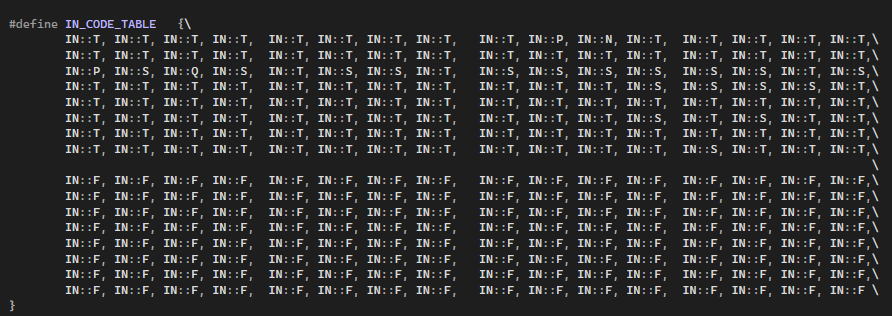


Рисунок 3.2. Таблица контроля входных символов

Таблица 3.1 Соответствие символов и их значений в таблице

|  |  |
| --- | --- |
| Значение в таблице входных символов | Символы |
| Разрешенный | T |
| Запрещенный | F |
| Игнорируемый | I |
| Сепаратор | S |

## **3.3 Удаление избыточных символов**

Избыточными символами являются символы табуляции и пробелы.

Избыточные символы удаляются на этапе разбиения исходного кода на токены.

Описание алгоритма удаления избыточных символов:

1. Посимвольно считываем файл с исходным кодом программы;

2. Встреча пробела или знака табуляции является своего рода встречей символа-сепаратора;

3. В отличие от других символов-сепараторов не записываем в очередь лексем эти символы, т.е. игнорируем.

## **3.4 Перечень ключевых слов**

Лексический анализатор преобразует исходный текст, заменяя лексические единицы лексемами для создания промежуточного представления исходной программы. Соответствие токенов и лексем приведено в таблице 3.2.

Таблица 3.2 Соответствие токенов и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| bool | b | Тип данных bool. |
| char | a | Тип данных char. |
| uint | t | Тип данных uint. |
| function | f | Объявление функции. |
| result | r | Выход из функции. |
| main | m | Главная функция. |
| let | d | Объявление переменной. |
| endl | x | Оператор перевода строки. |
| Идентификатор | i | Имя переменной/функции. |
| Литерал | l | Литерал доступного типа. |
| print | s | Оператор вывода. |
| if | q | Условный блок. |
| else | e | Условный блок. |
| ; | ; | Конец выражения. |
| , | . | Разделитель параметров. |
| [ | [ | Начало условного блока |
| ] | ] | Конец условного блока. |
| { | { | Открытие блока функции. |
| } | } | Закрытие блока функции. |

Продолжение таблицы 3.2 Соответствие токенов и сепараторов с лексемами

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Пояснение |
| ( | ( | Передача параметров в функцию, приоритет операций. |
| ) | ) | Закрытие блока для передачи параметров, приоритет операций. |
| = | = | Знак присваивания. |
| +  - | +  - | Знаки арифметических операций. |
| ~  &  % | ~  &  % | Знаки побитовых  операторов |

Каждому выражению соответствует детерминированный конечный автомат, по которому происходит разбор данного выражения. На каждый автомат в массиве подаётся токен и с помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. Структура конечного автомата и пример графа перехода конечного автомата представлены в листингах 3.3 и 3.4 соответственно.

|  |
| --- |
| namespace FST  {  struct RELATION  {  unsigned char symbol;  short nnode;  RELATION(  unsigned char c = 0x00;  short ns = 0;  )  };  struct NODE  {  short n\_relation;  RELATION \*relations;  NODE(); |
| NODE(short n,RELATION rel, ...);  };    struct FST  {  unsigned char\* string;  short position;  short nstates;  NODE\* nodes;  short\* rstates;  FST(  unsigned char\* s,  short ns,  NODE n, ...  );  };  bool execute(FST& fst);  } |

Листинг 3.3 - Структура конечного атвомата

|  |
| --- |
| #define FST\_FUNCTION 9, \  FST::NODE(1, FST::RELATION('f', 1)),\  FST::NODE(1, FST::RELATION('u', 2)),\  FST::NODE(1, FST::RELATION('n', 3)),\  FST::NODE(1, FST::RELATION('c', 4)),\  FST::NODE(1, FST::RELATION('t', 5)),\  FST::NODE(1, FST::RELATION('i', 6)),\  FST::NODE(1, FST::RELATION('o', 7)),\  FST::NODE(1, FST::RELATION('n', 8)),\  FST::NODE() |

## Листинг 3.4 - Пример графа перехода конечного автомата

## **3.5 Основные структуры данных**

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде, и номер в таблице идентификаторов, если лексема является идентификатором. Таблица идентификаторов содержит имя идентификатора, номер в таблице лексем, тип данных, тип идентификатора и значение. Код C++ со структурой таблицы лексем представлен на листинге 3.5. Код C++ со структурой таблицы идентификаторов представлен на листинге 3.6.

|  |
| --- |
| namespace LT  {  struct Entry  {  unsigned char lexema;  int sn;  int idxTI;  };  struct LexTable {  int maxsize;  int size;  Entry\* table;  };  LexTable Create(int size);  void Add(LexTable& lextable,Entry entry);  Entry GetEntry( LexTable& lextable, int n);  void Delete(LexTable& lextable);  Entry writeEntry(  Entry& entry,  unsigned char lexema,  int indx,  int line  );  void showTable(LexTable lextable, Log::LOG& log);  }; |

Листинг 3.5 Структура таблицы лексем

|  |
| --- |
| namespace IT  {  enum IDDATATYPE { UINT = 1, CHR = 2, BOOL = 3, STRING = 4 };  enum IDTYPE { V = 1, F = 2, P = 3, L = 4, OP = 5 };  struct Entry{  int idxfirstLE;  unsigned char id[ID\_MAXSIZE];  IDDATATYPE iddatatype;  IDTYPE idtype;  short system;  union{  unsigned int vint;  struct {  int len;  unsigned char str[TI\_STR\_MAXSIZE - 1];  } vstr;  } value;  };    struct IdTable{  int maxsize;  int size;  Entry\* table;  };  IdTable Create(int size);    void Add(IdTable& idtable,Entry entry);    Entry GetEntry( IdTable& idtable,int n);    int IsId( IdTable& idtable, unsigned char id[ID\_MAXSIZE]);    void Delete(IdTable& idtable);  void showTable(IdTable& idtable, Log::LOG log);  void newShowTable(IdTable& idtable);  }; |

Листинг 3.6 Структура таблицы идентификаторов

## **3.6 Принцип обработки ошибок**

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. При возникновении сообщения, лексический анализатор игнорирует найденную ошибку и продолжает работу с исходным кодом. Перечень сообщений представлен в таблице 3.7.

Таблица 3.7 - Сообщения лексического анализатора

|  |  |
| --- | --- |
| Код ошибки | Сообщение |
| 151 | Превышена максимальная длина строки |
| 152 | Слишком большое число |

## 3.7 Структура и перечень сообщений лексического анализатора

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Если в процессе анализа находятся более трёх ошибок, то анализ останавливается.

## **3.8 Параметры лексического анализатора**

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся как в файл журнала, так и в командную строку.

## **3.9 Алгоритм лексического анализа**

* проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой 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 «**string**» представлен на рисунке 3.2, где S0 – начальное, а S6 – конечное состояние автомата.



s t r i n g

Рисунок 3.2 Пример графа переходов для цепочки string

**3.10 Контрольный пример**

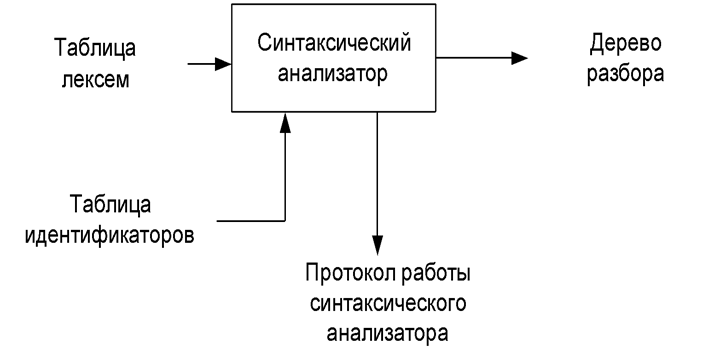
Результаты работы синтаксического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью приведены в приложении В.

# **4. Разработка синтаксического анализатора**

## **4.1 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.

Рисунок 4.1 Структура синтаксического анализатора.

## **4.2 Контекстно-свободная грамматика, описывающая синтаксис языка**

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

T – множество терминальных символов (было описано в разделе 1.2 данной пояснительной записки),

N – множество нетерминальных символов.

P – множество правил языка.

S – начальный символ грамматики, являющийся нетерминалом.

## **4.3 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, описание которых содержится в таблица 3.1 и 4.1. |
|  | Алфавит специальных магазинных символов | Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $) |
|  | Функция переходов автомата | Функция представляет из себя множество правил грамматики, описанных в таблице 4.1. |
|  | Начальное состояние автомата | Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики |
|  | Начальное состояние магазина автомата | 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 |

## **4.4 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 KII-2023. Данные структуры в приложении В.

**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. Иначе генерируется исключение.

## **4.6 Структура и перечень сообщений синтаксического анализатора**

Перечень сообщений синтаксического анализатора представлен на рисунке 4.3.

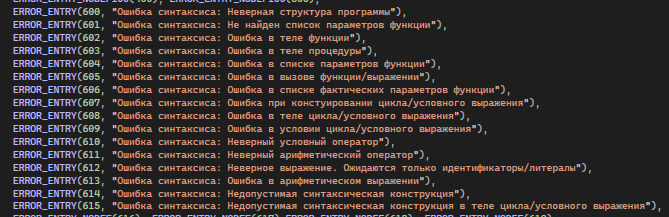


Рисунок 4.3 - Сообщения синтаксического анализатора

## **4.7. Параметры синтаксического анализатора и режимы его работы**

Входной информацией для синтаксического анализатора является таблица лексем и идентификаторов. Кроме того используется описание грамматики в форме Грейбах. Результаты работы лексического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

## **4.8. Принцип обработки ошибок**

Синтаксический анализатор выполняет разбор исходной последовательности лексем до тех пор, пока не дойдёт до конца цепочки лексем или не найдёт ошибку. Тогда анализ останавливается и выводится сообщение об ошибке (если она найдена). Если в процессе анализа находятся более трёх ошибок, то анализ останавливается.

## **4.9. Контрольный пример**

Результаты работы синтаксического разбора, а именно дерево разбора и протокол работы автомата с магазинной памятью приведены в приложении В.

# **5 Разработка семантического анализатора**

## **5.1 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 проверка на единственность точки входа, проверка на предварительное объявление переменной) осуществляются в процессе лексического анализа. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1. Структура семантического анализатора

## **5.2 Функции семантического анализатора**

Семантический анализатор выполняет проверку на основные правила языка (семантики языка), которые описаны в разделе 1.16.

## **5.3 Структура и перечень сообщений семантического анализатора**

Сообщения, формируемые семантическим анализатором, представлены в таблице 5.2.

Таблица 5.2 – Перечень сообщений семантического анализатора

|  |  |
| --- | --- |
| Код ошибки | Сообщение |
| 606 | Ошибка в параметре функции touint |
| 612 | Использование не объявленной переменной |
| 613 | Несоответствие типов при присвоении |
| 614 | Битовые операции возможны только над переменными/литералами типа uint |
| Продолжение таблицы 5.2 – Перечень сообщений семантического анализатора | |
| Код ошибки | Сообщение |
| 615 | Повторное объявление переменной |
| 615 | Повторное объявление переменной |
| 618 | Неверный возвращаемый тип функции |
| 621 | Несоответствие типов передаваемых параметров |
| 626 | Нельзя объявлять переменные в блоках if/else |

Рисунок 5.1 – Перечень сообщений семантического анализатора

## **5.4 Принцип обработки ошибок**

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Анализ останавливается после того, как будут найдены все ошибки.

## **5.5 Контрольный пример**

Соответствие примеров некоторых ошибок в исходном коде и диагностических сообщений об ошибках приведено в таблице 5.2.

Таблица 5.2. Примеры диагностики ошибок

|  |  |
| --- | --- |
| Исходный код | Текст сообщения |
| main {  let char a;  let uint a = 10;} | Повторное объявление переменной |
| main{  let uint x = 'hello';  } | Несоответствие типов при присвоении |
| Продолжение таблицы 5.2 – Примеры диагностики ошибок | |
| Исходный код | Текст сообщения |
| main{  let uint s = touint 124;  } | Ошибка в параметре функции touint |

# **6. Вычисление выражений**

# **6.1 Выражения, допускаемые языком**

В языке KII-2023 допускаются вычисления выражений беззнакового целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритеты операций

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| ( ) | 2 |
| & | 1 |
| | | 1 |
| ~ | 1 |

## **6.2 Польская запись и принцип её построения**

Все выражения языка KII-2023 преобразовываются к обратной польской записи.

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

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

## **6.3 Программная реализация обработки выражений**

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

# **7. Генерация кода**

## **7.1 Структура генератора кода**

В языке KII-2023 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода KII-2023 представлена на рисунке 7.1.

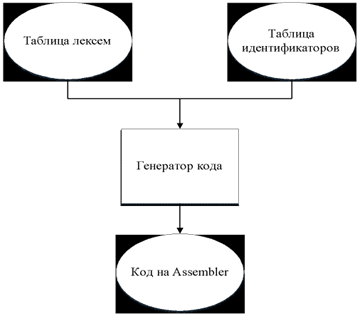


Рисунок 7.1 – Структура генератора кода

## 7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке KII-2023 и на языке ассемблера приведены в таблице 7.1.

Таблица 7.1 – Соответствия типов идентификаторов языка KII-2023 и языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке KII-2023 | Тип идентификатора на языке ассемблера | Пояснение |
| uint | DWORD | Хранит целочисленный тип данных. |
| char | DWORD | Хранит символьный тип данных |

## **7.3 Статическая библиотека**

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

Таблица 7.3 – Функции статической библиотеки

|  |  |
| --- | --- |
| Функция | Назначение |
| void ShowBool(int n, int endl) | Вывод в консоль булевого значения |
| void ShowString(unsigned char\* str, int n, int endl) | Вывод в консоль строки |
|  | Вывод в консоль символа |
| void ShowInt(unsigned int n, int endl, int system) | Вывод в консоль целочисленной переменной |
| unsigned int StringToInt(unsigned char\* str, int n) | Преобразование строки в число |
| int cmpuint(unsigned int n, unsigned int m) | Сравнение чисел |

## **7.4 Особенности алгоритма генерации кода**

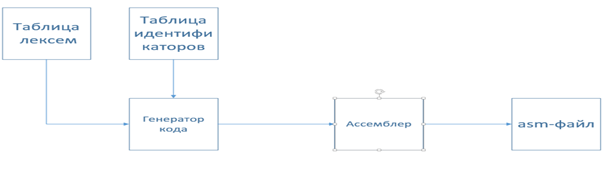
В языке KII-2023 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке

Рисунок 7.2 – Структура генератора кода

## **7.5 Входные параметры генератора кода**

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке KII-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 Д. Результат работы контрольного примера приведён на рисунке 7.2.

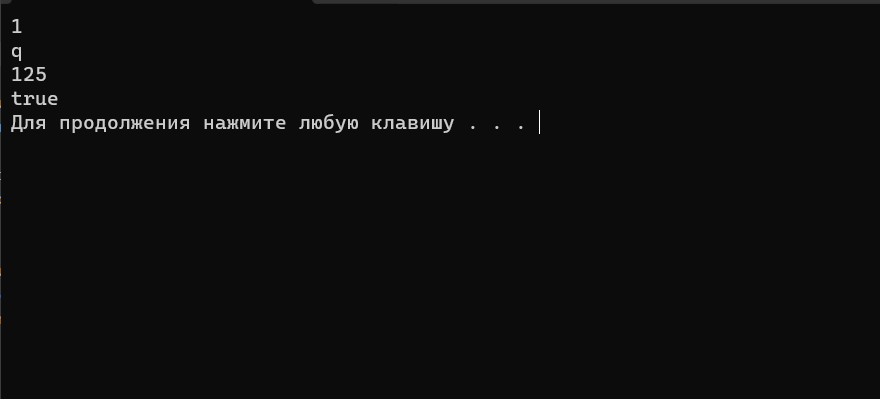


Рисунок 7.2 Результат работы программы на языке KII-2023

# **8. Тестирование транслятора**

## **8.1 Тестирование проверки на допустимость символов**

В языке KII-2023 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 - Тестирование проверки на допустимость символов

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main {print 'п'} | Ошибка лексики: Недопустимый символ в исходном файле(-in) |

## **8.2 Тестирование лексического анализатора**

На этапе лексического анализа в языке KII-2023 могут возникнуть ошибки, описанные в пункте 3.7. Результаты тестирования лексического анализатора показаны в таблице 8.2.

Таблица 8.2 - Тестирование лексического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main  {new uint x11;} | Ошибка лексики: Неизвестная последовательность символов |

## **8.3 Тестирование синтаксического анализатора**

На этапе синтаксического анализа в языке KII-2023 могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 - Тестирование синтаксического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main{ new uint x == 10;} | Ошибка синтаксиса: Неверное выражение. Ожидаются только идентификаторы/литералы |

## **8.4 Тестирование семантического анализатора**

Семантический анализ в языке KII-2023 содержит множество проверок по семантическим правилам, описанным в пункте 1.16. Итоги тестирования семантического анализатора на корректное обнаружение семантических ошибок приведены в таблице 8.4.

Таблица 8.4 - Тестирование семантического анализатора

|  |  |
| --- | --- |
| Исходный код | Диагностическое сообщение |
| main { let int a = 24; } | Ошибка семантики: В объявлении отсутствует ключевое слово |
| main{new uint t = "1"} | Ошибка семантики: Типы данных в выражении не совпадают |
| main{}  main{} | Ошибка семантики: Обнаружено несколько точек входа main |
| main{new uint t;  new str t;} | Ошибка семантики: Попытка переопределения идентификатора |
| main{new uint x=99999999999999999;} | Ошибка семантики: Недопустимый целочисленный литерал |

# **Заключение**

В ходе выполнения курсовой работы был разработан транслятор и генератор кода для языка программирования KII-2023 со всеми необходимыми компонентами. Таким образом, были выполнены основные задачи данной курсовой работы:

1. Сформулирована спецификация языка KII-2023;
2. Разработаны конечные автоматы и важные алгоритмы на их основе для эффективной работы лексического анализатора;
3. Осуществлена программная реализация лексического анализатора, распознающего допустимые цепочки спроектированного языка;
4. Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
5. Осуществлена программная реализация синтаксического анализатора;
6. Разработан семантический анализатор, осуществляющий проверку используемых инструкций на соответствие логическим правилам;
7. Разработан транслятор кода на язык ассемблера;
8. Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка KII-2023 включает:

1. 2 типа данных;
2. Поддержка операторов вывода и перевода строки;
3. Возможность вызова функций стандартной библиотеки;
4. Наличие 4 арифметических операторов для вычисления выражений;
5. Поддержка функций, процедур, операторов цикла и условия;
6. 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.

# Список использованных источников

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

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

# **Приложение А**

Листинг 1 – Исходный код программы на языка KII-2023

|  |
| --- |
| bool function fa(uint a) {  let bool b;  let bool c;  b = cmp(a,0);  if (b) [  result true;  ]  else [  print 'number ';  print a,endl;  a=a-;  c = fa(a);  ]  result false;  };  main {  let uint a = 10;  a = 1&101;  print a,endl;  let char c = 'q';  print c , endl;  a = touint '125';  print a, endl;  let bool b;  a = 0;  b = fa(a);  print b, endl;    } |

# **Приложение Б**

Листинг 1 Таблица идентификаторов контрольного примера

|  |
| --- |
| № | Идентификатор | Тип данных | Тип идентификатора | Индекс в ТЛ | Значение  ---------------------------------------------------------------------------------------  0000 | fa | bool | функция | 2 | -  0001 | faa | uint | параметр | 5 | -  0002 | fab | bool | переменная | 10 | 3435973836  0003 | fac | bool | переменная | 14 | 3435973836  0004 | L1 | uint | литерал | 22 | 0  0005 | L2 | uint | литерал | 31 | 1  0006 | L3 | string | литерал | 37 | [7]"number "  0007 | - | - | оператор | 47 | -  0008 | maina | uint | переменная | 66 | 0  0009 | L4 | uint | литерал | 68 | 10  0010 | & | - | оператор | 73 | -  0011 | L5 | uint | литерал | 74 | 101  0012 | mainc | char | переменная | 83 | [0]""  0013 | L6 | string | литерал | 85 | [1]"q"  0014 | L7 | string | литерал | 95 | [3]"125"  0015 | mainb | bool | переменная | 104 | 3  --------------------------------------------------------------------------------------- |

Листинг 2 Таблица лексем после контрольного примера

|  |
| --- |
| 01 bfi[0](ti[1]){  02 dbi[2];  03 dbi[3];  04 i[2]=n(i[1],l[4]);  05 q(i[2])[  06 rl[5];  07 ]  08 e[  09 sl[6];  10 si[1],x;  11 i[1]=i[1]~;  12 i[3]=i[0](i[1]);  13 ]  14 rl[4];  15 };  16 m{ |

|  |
| --- |
| 17 dti[8]=l[9];  18 i[8]=l[5]v[10]l[11];  19 si[8],x;  20 dai[12]=l[13];  21 si[12],x;  22 i[8]=hl[14];  23 si[8],x;  24 dbi[15];  25 i[8]=l[4];  26 i[15]=i[0](i[8]);  27 si[15],x;  29 } |

# **Приложение В**

Листинг 1 Грамматика языка KII –2023

|  |
| --- |
| Greibach greibach(  NS('S'),  TS('$'),  9,  Rule(  NS('S'),  GRB\_ERROR\_SERIES + 0,  22,  Rule::Chain(6, TS('m'), TS('{'), NS('N'), TS('}'), TS(';'), NS('S')),  Rule::Chain(14, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';')),  Rule::Chain(14, TS('a'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('a'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';')),  Rule::Chain(14, TS('b'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('b'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';')),  Rule::Chain(14, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('t'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';')),  Rule::Chain(14, TS('a'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('a'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';')),  Rule::Chain(14, TS('b'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('b'), TS('f'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';')),  Rule::Chain(5, TS('m'), TS('{'), NS('N'), TS('}'), TS(';')),  Rule::Chain(5, TS('m'), TS('{'), TS('}'), TS(';'), NS('S')),  Rule::Chain(13, TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(12, TS('t'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';')),  Rule::Chain(13, TS('a'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';'), NS('S')), |

|  |
| --- |
| Rule::Chain(12, TS('a'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'),  TS(';')),  Rule::Chain(13, TS('b'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';'), NS('S')),  Rule::Chain(12, TS('b'), TS('f'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('C'), TS(';'), TS('}'), TS(';')),  Rule::Chain(4, TS('m'), TS('{'), TS('}'), TS(';'))  ),  Rule(  NS('N'),  GRB\_ERROR\_SERIES + 1,  29,  Rule::Chain(5, TS('d'), TS('t'), TS('i'), TS(';'), NS('N')),  Rule::Chain(5, TS('d'), TS('b'), TS('i'), TS(';'), NS('N')),  Rule::Chain(5, TS('d'), TS('a'), TS('i'), TS(';'), NS('N')),  Rule::Chain(7, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(7, TS('d'), TS('b'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(7, TS('d'), TS('a'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(6, TS('d'), TS('t'), TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(6, TS('d'), TS('b'), TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(6, TS('d'), TS('a'), TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(9, TS('i'), TS('='), TS('n'), TS('('), NS('C'), TS(','), NS('C'), TS(')'), TS(';')),  Rule::Chain(10, TS('i'), TS('='), TS('n'), TS('('), NS('C'), TS(','), NS('C'), TS(')'), TS(';'), NS('N')),  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(5, TS('i'), TS('='), TS('h'), NS('C'), TS(';')),  Rule::Chain(6, TS('i'), TS('='), TS('h'), NS('C'), TS(';'), NS('N')),  Rule::Chain(6 ,TS('i'), TS('='), TS('o'), NS('C'), TS(';'), NS('N')),  Rule::Chain(5, TS('i'), TS('='), TS('o'), NS('C'), TS(';')),  Rule::Chain(2, TS('q'), NS('Q')),  Rule::Chain(4, TS('d'), TS('t'), TS('i'), TS(';')),  Rule::Chain(4, TS('d'), TS('b'), TS('i'), TS(';')),  Rule::Chain(4, TS('d'), TS('a'), TS('i'), TS(';')),  Rule::Chain(3, TS('r'), NS('C'), TS(';')),  Rule::Chain(3, TS('r'), NS('C'), TS(';'), NS('N')),  Rule::Chain(6, TS('c'), TS('i'), TS(','), TS('l'), TS(';'), NS('N')),  Rule::Chain(5, TS('c'), TS('i'), TS(','), TS('l'), TS(';')),  Rule::Chain(4, TS('s'), NS('C'), TS(';'), NS('N')),  Rule::Chain(3, TS('s'), NS('C'), TS(';')),  Rule::Chain(6, TS('s'), NS('C'), TS(','), TS('x'), TS(';'), NS('N')),  Rule::Chain(5, TS('s'), NS('C'), TS(','), TS('x'), TS(';'))  ), |

|  |
| --- |
| Rule(  NS('Q'),  GRB\_ERROR\_SERIES + 7,  8,  Rule::Chain(11, TS('('), TS('i'), TS(')'), TS('['), NS('N'), TS(']'), TS('e'), TS('['), NS('N'), TS(']'), NS('N')),  Rule::Chain(10, TS('('), TS('i'), TS(')'), TS('['), NS('N'), TS(']'), TS('e'), TS('['), NS('N'), TS(']')),  Rule::Chain(9, TS('('), TS('i'), TS(')'), TS('['), TS(']'), TS('e'), TS('['), TS(']'), NS('N')),  Rule::Chain(8, TS('('), TS('i'), TS(')'), TS('['), TS(']'), TS('e'), TS('['), TS(']')),  Rule::Chain(10, TS('('), TS('i'), TS(')'), TS('['), TS(']'), TS('e'), TS('['), NS('N'), TS(']'), NS('N')),  Rule::Chain(9, TS('('), TS('i'), TS(')'), TS('['), TS(']'), TS('e'), TS('['), NS('N'), TS(']')),  Rule::Chain(10, TS('('), TS('i'), TS(')'), TS('['), NS('N'), TS(']'), TS('e'), TS('['), TS(']'), NS('N')),  Rule::Chain(9, TS('('), TS('i'), TS(')'), TS('['), NS('N'), TS(']'), TS('e'), TS('['), TS(']'))  ),  Rule(  NS('C'),  GRB\_ERROR\_SERIES + 9,  2,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l'))  ),  Rule(  NS('D'),  GRB\_ERROR\_SERIES + 2,  4,  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l')),  Rule::Chain(2, TS('i'), NS('M')),  Rule::Chain(2, TS('l'), NS('M'))  ),  Rule(  NS('E'),  GRB\_ERROR\_SERIES + 2,  9,  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(  NS('F'),  GRB\_ERROR\_SERIES + 3,  6,  Rule::Chain(2, TS('t'), TS('i')),  Rule::Chain(2, TS('b'), TS('i')),  Rule::Chain(2, TS('a'), TS('i')),  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F')),  Rule::Chain(4, TS('b'), TS('i'), TS(','), NS('F')),  Rule::Chain(4, TS('a'), TS('i'), TS(','), NS('F'))  ),  Rule(  NS('W'),  GRB\_ERROR\_SERIES + 4,  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  4,  Rule::Chain(2, TS('v'), NS('E')),  Rule::Chain(2, TS('~'), NS('M')),  Rule::Chain(1, TS('~')),  Rule::Chain(3, TS('v'), NS('E'), NS('M'))  )  ); |

Листинг 2 Структура магазинного автомата

|  |
| --- |
| struct Mfst{  enum RC\_STEP {  NS\_OK,  NS\_NORULE,  NS\_NORULECHAIN,  NS\_ERROR,  TS\_OK,  TS\_NOK,  LENTA\_END,  SURPRISE  };  struct MfstDiagnosis{  short lenta\_position;  RC\_STEP rc\_step;  short nrule; |

|  |
| --- |
| short nrule\_chain;  MfstDiagnosis();  MfstDiagnosis(  short plenta\_position,  RC\_STEP prt\_step,  short pnrule,  short pnrule\_chain  );  } diagnosis[MFST\_DIAGN\_NUMBER];  GRBALPHABET\* lenta;  short lenta\_position;  short nrule;  short nrulechain;  short lenta\_size;  GRB::Greibach grebach;  Lex::LEX lex;  MFSTSTACK st;  std::stack<MfstState> storestate;  Mfst();  Mfst(  Lex::LEX plex,  GRB::Greibach pgrebach  );  char\* getCSt(char\* buf);  char\* getCLenta(char\* buf, short pos, short n = 25);  char\* getDiagnosis(short n, char\* buf);  bool savestate(Log::LOG log);  bool reststate(Log::LOG log);  bool push\_chain(  GRB::Rule::Chain chain  );  RC\_STEP step(Log::LOG log);  bool start(Log::LOG log);  bool savediagnosis(  RC\_STEP pprc\_step  );  void printrules(Log::LOG log);  struct Deducation  {  short size;  short\* nrules;  short\* nrulechains;  Deducation() { size = 0; nrules = 0; nrulechains = 0; };  } deducation;  bool savededucation();  }; |

Листинг 3 Структура грамматики Грейбах

|  |
| --- |
| struct Greibach {  short size;  GRBALPHABET startN;  GRBALPHABET stbottomT;  Rule\* rules;  Greibach() { short size = 0; startN = 0; stbottomT = 0; rules = 0; };  Greibach(  GRBALPHABET pstartN,  GRBALPHABET pstbottomT,  short psize,  Rule r,  );  short getRule(  GRBALPHABET pnn,  Rule& prule  );  Rule getRule(short n);  }; |

Листинг 4 Разбор исходного кода синтаксическим анализатором

|  |
| --- |
| 0---: S->bfi(F){NrC;};S---bfi(ti){dbi;dbi;i=n(i,l);-----S$------------------  0---: SAVESTATE:----------1  0---: -------------------bfi(ti){dbi;dbi;i=n(i,l);-----bfi(F){NrC;};S$-----  1---: -------------------fi(ti){dbi;dbi;i=n(i,l);q-----fi(F){NrC;};S$------  2---: -------------------i(ti){dbi;dbi;i=n(i,l);q(-----i(F){NrC;};S$-------  3---: -------------------(ti){dbi;dbi;i=n(i,l);q(i-----(F){NrC;};S$--------  4---: -------------------ti){dbi;dbi;i=n(i,l);q(i)-----F){NrC;};S$---------  5---: F->ti---------------ti){dbi;dbi;i=n(i,l);q(i)-----F){NrC;};S$---------  5---: SAVESTATE:----------2  5---: -------------------ti){dbi;dbi;i=n(i,l);q(i)-----ti){NrC;};S$--------  6---: -------------------i){dbi;dbi;i=n(i,l);q(i)[-----i){NrC;};S$---------  7---: -------------------){dbi;dbi;i=n(i,l);q(i)[r-----){NrC;};S$----------  8---: -------------------{dbi;dbi;i=n(i,l);q(i)[rl-----{NrC;};S$-----------  9---: -------------------dbi;dbi;i=n(i,l);q(i)[rl;-----NrC;};S$------------  10--: N->dti;N------------dbi;dbi;i=n(i,l);q(i)[rl;-----NrC;};S$------------  10--: SAVESTATE:----------3  10--: -------------------dbi;dbi;i=n(i,l);q(i)[rl;-----dti;NrC;};S$-------- |

# **Приложение Г**

Листинг 1 Программная реализация механизма преобразования в ПОЛИЗ

|  |
| --- |
| LT::LexTable PolishNotation(int lextable\_pos, LT::LexTable& lextable, IT::IdTable& idtable) {  stack <LT::Entry> op;  stack <LT::Entry> res;  LT::Entry end;  bool flag = true;  end.lexema = '|';  op.push(end);  bool func = false;  int k = lextable\_pos, j = 0;  LT::LexTable str = LT::Create(LT\_MAXSIZE);  LT::LexTable buff = LT::Create(LT\_MAXSIZE);  while (lextable.table[k].lexema!= LEX\_SEMICOLON)  {  str.table[j] = lextable.table[k];  k++;  j++;  }  str.table[j].lexema = '|';  for (int i = 0; i < j+1; i++) {  if (str.table[i].lexema == LEX\_ID) {  if (idtable.table[str.table[i].idxTI].idtype == IT::F) {  func = true;  }  }  if (func) {  if (str.table[i].lexema == LEX\_LITERAL || str.table[i].lexema == LEX\_ID) {  res.push(str.table[i]);  } |

|  |
| --- |
| else if (str.table[i].lexema == LEX\_RIGHTTHESIS) {  LT::Entry entryLT = writeEntry(entryLT, '@', LT\_TI\_NULLIDX, 0);  res.push(entryLT);  func = false;  }  }  else if (str.table[i].lexema == LEX\_LITERAL || str.table[i].lexema == LEX\_ID) {  res.push(str.table[i]);  }  else if (str.table[i].lexema == LEX\_OPERATOR) {  if (idtable.table[(str.table[i].idxTI)].id[0] == LEX\_OR) {  if (op.top().lexema == '|' || op.top().lexema == LEX\_LEFTTHESIS)  op.push(str.table[i]);  else {  res.push(op.top());  op.pop();  i--;  }  }  else if (idtable.table[(str.table[i].idxTI)].id[0] == LEX\_AND || idtable.table[(str.table[i].idxTI)].id[0] == LEX\_XOR) {  if (op.top().lexema == LEX\_LEFTTHESIS)  op.push(str.table[i]);  else if (op.top().lexema == '|')  op.push(str.table[i]);  else if (op.top().lexema == LEX\_OPERATOR)  if (idtable.table[op.top().idxTI].id[0] == LEX\_OR)  op.push(str.table[i]);  else {  res.push(op.top());  op.pop();  i--;  }  }  } |

|  |
| --- |
| else if (str.table[i].lexema == LEX\_NOT) {  res.push(str.table[i]);  }  else if (str.table[i].lexema == LEX\_LEFTTHESIS)  op.push(str.table[i]);  else if (str.table[i].lexema == LEX\_RIGHTTHESIS)  if (op.top().lexema == '|') {  flag = !flag;  break;  }  else if (op.top().lexema == LEX\_LEFTTHESIS)  op.pop();  else {  res.push(op.top());  op.pop();  i--;  }  else if (str.table[i].lexema == '|') {  if (op.top().lexema == '|')  break;  else if (str.table[i].lexema == LEX\_LEFTTHESIS) {  flag = !flag;  break;  }  else {  res.push(op.top());  op.pop();  i--;  }  }  }  j = 0;  if (flag)  while (res.size()) {  buff.table[j] = res.top();  res.pop();  j++;  }  else  cout << "Error";  buff.size = j;  return buff;  } |

# **Приложение Д**

Листинг 1 Результат генерации кода контрольного примера в Ассемблере

|  |
| --- |
| .586  .model flat, stdcall    includelib libucrt.lib  includelib kernel32.lib  includelib Lib.lib    ShowInt PROTO : DWORD, : DWORD, : DWORD  ShowChar PROTO : DWORD, : DWORD  CharToInt PROTO : DWORD, : DWORD  ShowString PROTO : DWORD, : DWORD, : DWORD  ShowBool PROTO : DWORD, : DWORD  inv PROTO : DWORD  StringToInt PROTO : DWORD, : DWORD  translate PROTO : DWORD, : DWORD, : DWORD  cmpstr PROTO : DWORD, : DWORD  cmpuint PROTO : DWORD, : DWORD  ExitProcess PROTO : DWORD  stop PROTO  funcfa PROTO :DWORD  .stack 4096    .const  L1 DWORD 0  systemL1 DWORD 10  L2 DWORD 1  systemL2 DWORD 10  L3 DWORD 'n','u','m','b','e','r',' '  L4 DWORD 10  systemL4 DWORD 10  L5 DWORD 5  systemL5 DWORD 10  L6 DWORD 8  systemL6 DWORD 10 |

|  |
| --- |
| L7 DWORD 3  systemL7 DWORD 10  L8 DWORD 2  systemL8 DWORD 10  L9 DWORD 'q'  L10 DWORD '1','2','5'    .data  systemfaa DWORD 10  fab DWORD 0  fac DWORD 0  maina DWORD 0  systemmaina DWORD 10  mainc DWORD 0  mainb DWORD 0  .CODE  funcfa PROC faa :DWORD  push faa  push L1  call cmpuint  push eax  pop eax  mov fab,eax  mov eax ,fab  cmp eax,0  je else0  push L2  pop eax  RET  call go0  else0:  mov esi, offset L3  mov ecx, LENGTHOF L3  mov ebx,ecx  push 0  push ebx  push esi  call ShowString  push systemfaa  push 1 |
| push faa  call ShowInt  push faa  pop eax  sub eax,1  cmp eax, -1  je exit  push eax  pop eax  mov faa,eax  push faa  call funcfa  push eax  pop eax  mov fac,eax  go0:  push L1  pop eax  RET  funcfa ENDP  MAIN:  push L4  pop eax  mov maina,eax  push L5  call inv  push eax  push L6  push L7  push L8  pop eax  pop ebx  or eax,ebx  push eax  pop eax  add eax,1  push eax  pop eax  pop ebx  xor eax,ebx  push eax  push L8 |
| pop eax  pop ebx  and eax,ebx  push eax  pop eax  pop ebx  or eax,ebx  push eax  pop eax  mov maina,eax  push systemmaina  push 1  push maina  call ShowInt  push L9  pop eax  mov mainc,eax  push 1  push mainc  call ShowChar  mov esi, offset L10  mov ecx, LENGTHOF L10  mov ebx,ecx  push ebx  push esi  call StringToInt  cmp eax, -1  je exit  mov maina,eax  push systemmaina  push 1  push maina  call ShowInt  push L1  pop eax  mov maina,eax  push maina  call funcfa  push eax  pop eax |
| mov mainb,eax  push 1  push mainb  call ShowBool  call stop  exit:  push 0  call ExitProcess  END MAIN |