Министерство образования Российской Федерации.

Федеральное государственное общеобразовательное учреждение высшего профессионального образования

«Чувашский государственный университет имени И.Н. Ульянова»

Алатырский филиал

**Курсовая работа**

**По предмету: «Теория вычислительных процессов и структур»**

**На тему: «Проектирование компилятора»**

Выполнил студент

группы АФТ 61-05

Федин А. В.

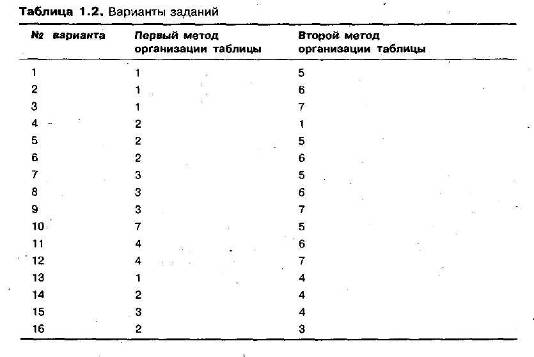
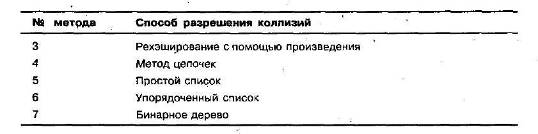
Научный руководитель:

Пичугин В.Н.

Алатырь 2009

**Задание вариант №9**

компилятор идентификатор лексический анализатор



**Задание №2**

Входной язык содержит арифметические выражения, разделенные символом; (точка с запятой). Арифметические выражения *состоят* из идентификаторов, шестнадцатеричных чисел, знака присваивания (:=), знаков операций +, -, \*, / и круглых скобок.

**Содержание**

Введение

1 Организация таблиц идентификаторов

1.1 Назначение таблиц идентификаторов

1.2 Принципы организации таблиц идентификаторов

1.3 Простейшие методы построения таблиц идентификаторов

1.4 Метод простого рехэширования с помощью произведения

2 Проектирование лексического анализатора

2.1 Назначение лексического анализатора

2.2 Таблица лексем и содержащаяся в ней информации

2.3 Построение лексических анализаторов (сканеров)

Заключение

Список использованной литературы

Приложение 1

Приложение 2

Приложение 3

# Введение

Компилятор – программный модуль, задачей которого является перевод программы, написанной на одном из языков программирования (исходный язык) в программу на язык ассемблера или язык машинных команд.

Большинство компиляторов переводят программу с некоторого высокоуровневого языка программирования в машинный код, который может быть непосредственно выполнен компьютером.

Целью данной курсовой работы является изучение составных частей, основных принципов построения и функционирования компиляторов, практическое освоение методов построения составных частей компилятора для заданного входного языка.

Курсовая работа заключается в создании отдельных частей компилятора заданного языка.

В первой части работы ставится задача разработать программу, которая получает на входе набор идентификаторов, организует таблицу по заданному методу и позволяет осуществить многократный поиск идентификатора в этой таблице. Программа должна сообщать среднее число коллизий и среднее количество сравнений, выполняемых для поиска идентификатора.

Во второй части работы требуется разработать программу, которая выполняет лексический анализ входного текста по заданной грамматике и порождает таблицу лексем с указанием их типов и значений.

# 1 Организация таблиц идентификаторов

# 

# 1.1 Назначение таблиц идентификаторов

При выполнении семантического анализа, генерации кода и оптимизации результирующей программы компилятор должен оперировать характеристиками основных элементов исходной программы - переменных, констант, функций и других лексических единиц входного языка. Эти характеристики могут быть получены компилятором на этапе синтаксического анализа входной программы (чаще всего при анализе структуры блоков описаний переменных и констант), а также дополнены на этапе подготовки к генерации кода (например при распределении памяти).

Набор характеристик, соответствующий каждому элементу исходной программы, зависит от типа этого элемента, от его смысла (семантики) и, соответственно, от той роли, которую он исполняет в исходной и результирующей программах. В каждом конкретном случае этот набор характеристик может быть свой в зависимости от синтаксиса и семантики входного языка, от архитектуры целевой вычислительной системы и от структуры компилятора. Но есть типовые характеристики, которые чаще всего присущи тем или иным элементам исходной программы. Например для переменной - это ее тип и адрес ячейки памяти, для константы - ее значение, для функции - количество и типы формальных аргументов, тип возвращаемого результата, адрес вызова кода функции.

Главной характеристикой любого элемента исходной программы является его имя. Именно с именами переменных, констант, функций и других элементов входного языка оперирует разработчик программы - поэтому и компилятор должен уметь анализировать эти элементы по их именам.

Имя каждого элемента должно быть уникальным. Многие современные языки программирования допускают совпадения (неуникальность) имен переменных, и функций в зависимости от их области видимости и других условий исходной программы.

Таким образом, задача компилятора заключается в том, чтобы хранить некоторую информацию, связанную с каждым элементом исходной программы, и иметь доступ к этой информации по имени элемента. Для решения этой задачи компилятор организует специальные хранилища данных, называемые таблицами идентификаторов, или таблицами символов. Таблица идентификаторов состоит из набора полей данных (записей), каждое из которых может соответствовать одному элементу исходной программы. Запись содержит всю необходимую компилятору информацию о данном элементе и может пополняться по мере работы компилятора. Количество записей зависит от способа организации таблицы идентификаторов, но в любом случае их не может быть меньше, чем элементов в исходной программе. В принципе, компилятор может работать не с одной, а с несколькими таблицами идентификаторов - их количество и структура зависят от реализации компилятора.

# 

# 1.2 Принципы организации таблиц идентификаторов

Компилятор пополняет записи в таблице идентификаторов по мере анализа исходной программы и обнаружения в ней новых элементов, требующих размещения в таблице. Поиск информации в таблице выполняется всякий раз, когда компилятору необходимы сведения о том или ином элементе программы. Причем следует заметить, что поиск элемента в таблице будет выполняться компилятором существенно чаще, чем помещение в нее новых элементов. Так происходит потому, что описания новых элементов в исходной программе, как правило, встречаются гораздо реже, чем эти элементы используются. Кроме того, каждому добавлению элемента в таблицу идентификаторов в любом случае будет предшествовать операция поиска - чтобы убедиться, что такого элемента в таблице нет.

На каждую операцию поиска элемента в таблице компилятор будет затрачивать время, и поскольку количество элементов в исходной программе велико (от единиц до сотен тысяч в зависимости от объема программы), это время будет существенно влиять на общее время компиляции. Поэтому таблицы идентификаторов должны быть организованы таким образом, чтобы компилятор имел возможность максимально быстро выполнять поиск нужной ему записи таблицы по имени элемента, с которым связана эта запись.

Можно выделить следующие способы организации таблиц идентификаторов:

□ простые и упорядоченные списки;

□ бинарное дерево;

□ хэш - адресация с рехэшированием;

□ хэш - адресация по методу цепочек;

□ комбинация хэш - адресации ее списком или бинарным деревом.

Далее будет дано краткое описание способа организации таблиц идентификаторов при помощи простого списка.

# 

# 1.3 Простейшие методы построения таблиц идентификаторов

В простейшем случае таблица идентификаторов представляет собой линейный неупорядоченный список, или массив, каждая ячейка которого содержит данные о соответствующем элементе таблицы. Размещение новых элементов в такой таблице выполняется путем записи информации в очередную ячейку массива или списка по мере обнаружения новых элементов в исходной программе.

Поиск нужного элемента в таблице будет в этом случае выполняться путём последовательного перебора всех элементов и сравнения их имени с именем искомого элемента, пока не будет найден элемент с таким же именем. Тогда если за единицу времени принять время, затрачиваемое компилятором на сравнение двух строк (в современных вычислительных системах такое сравнение чаще всего выполняется одной командой), то для таблицы, содержащей N элементов, в среднем будет выполнено N/2 сравнений.

Время, требуемое на добавление нового элемента в таблицу (Тд), не зависит от числа элементов в таблице (N). Но если N велико, то поиск потребует значительных затрат времени. Время поиска (Ти) в такой таблице можно оценить как Ти = O(N). Поскольку именно поиск в таблице идентификаторов является наиболее часто выполняемой компилятором операцией, такой способ организации таблиц идентификаторов является неэффективным. Он применим только для самых простых компиляторов, работающих с небольшими программами.

Поиск может быть выполнен более эффективно, если элементы таблицы отсортированы (упорядочены) естественным образом. Поскольку поиск осуществляется по имени, наиболее естественным решением будет расположить элементы таблицы в прямом или обратном алфавитном порядке. Эффективным методом поиска в упорядоченном списке из N элементов является бинарный, или логарифмический; поиск.

Алгоритм логарифмического поиска заключается в следующем: искомый символ сравнивается с элементом (N+ 1)/2 в середине таблицы; если этот элемент не является искомым, то мы должны просмотреть только блок элементов, пронумерованных от 1 до (N+ 1)/2 - 1, или блок элементов от (N+ 1)/2 + 1 до N в зависимости от того, меньше или больше искомый элемент того, с которым его сравнили. Затем процесс повторяется над нужным блоком в два раза меньшего размера. Так продолжается до тех пор, пока либо искомый элемент не будет найден, либо алгоритм не дойдет до очередного блока, содержащего один или два элемента (с которыми можно выполнить прямое сравнение искомого элемента).

Так как на каждом шаге число элементов, которые могут содержать искомый элемент, сокращается в два раза, максимальное число сравнений равно 1 + log2 N. Тогда время поиска элемента в таблице идентификаторов можно оценить как Тп = O(log2 N). Для сравнения: при N=128 бинарный поиск требует самое большее 8 сравнений, а поиск в неупорядоченной таблице — в среднем 64 сравнения. Метод называют «бинарным поиском», поскольку на каждом шаге объем рассматриваемой информации сокращается в два раза, а «логарифмическим» — поскольку время, затрачиваемое на поиск нужного элемента в массиве, имеет логарифмическую зависимость от общего количества элементов в нем.

Недостатком логарифмического поиска является требование упорядочивания таблицы идентификаторов. Так как массив информации, в котором выполняется поиск, должен быть упорядочен, время его заполнения уже будет зависеть от числа элементов в массиве. Таблица идентификаторов зачастую просматривается компилятором еще до того, как она заполнена, поэтому требуется, чтобы условие упорядоченности выполнялось на всех этапах обращения к ней. Следовательно, для построения такой таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.

Если пользоваться стандартными алгоритмами, применяемыми для организации упорядоченных массивов данных, то среднее время, необходимое на помещение всех элементов в таблицу, можно оценить следующим образом:

Тд = O(N\*log2 N) + k\*O(N^2).

Здесь k - некоторый коэффициент, отражающий соотношение между временами, затрачиваемыми компьютером на выполнение операции сравнения и операции переноса данных.

При организации логарифмического поиска в таблице идентификаторов обеспечивается существенное сокращение времени поиска нужного элемента за счет увеличения времени на помещение нового элемента в таблицу. Поскольку добавление новых элементов в таблицу идентификаторов происходит существенно реже, чем обращение к ним, этот метод следует признать более эффективным, чем метод организации неупорядоченной таблицы. Однако в реальных компиляторах этот метод непосредственно также не используется, поскольку существуют более эффективные методы.

# 

# 1.4 Метод простого рехэширования с помощью произведения.

Для организации таблицы идентификаторов по методу рехэширования с помощью произведения необходимо определить все хэш-функции для всех Чаще всего функции определяют как некоторую модификацию хэш-функции *h*. Например, самым простым методом вычисления функции является ее организация в виде



где — некоторое вычисляемое целое число, а — максимальное значение из области значений хэш-функции *h*. В данной работе положим . Тогда получаем формулу



В этом случае при совпадении значений хэш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной хэш-функцией .



Блок-схема метода простого рехэширования представлена на рисунке 1.1.



**Рис. 1.1 – Блок-схема метода простого рехэширования с помощью произведения**

*а*) – Блок-схема алгоритма простого рехэширования с помощью произведения; *б*) – Блок-схема функции поиска идентификатора;

*в*) – Блок-схема функции добавления идентификатора

# 2 Проектирование лексического анализатора

# 

# 2.1 Назначение лексического анализатора

Лексический анализатор (или сканер) – это часть-компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы) исходного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора.

Лексема (лексическая единица языка) – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т. п. Состав возможных лексем каждого конкретного языка программирования определяется синтаксисом этого языка.

С теоретической точки зрения лексический анализатор не является обязательной, необходимой частью компилятора. Его функции могут выполняться на этапе синтаксического анализа. Однако существует несколько причин, исходя из которых в состав практически всех компиляторов включают лексический анализ.

Это следующие причины:

- упрощается работа с текстом исходной программы на этапе синтаксического разбора и сокращается объем обрабатываемой информации, так как лексический анализатор структурирует поступающий на вход исходный текст программы и удаляет всю незначащую информацию;

- для выделения в тексте и разбора лексем возможно применять простую, эффективную и хорошо проработанную теоретически технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;

- лексический анализатор отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от версии входного языка - при такой конструкции компилятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой лексический анализатор.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от версии компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих пробелов, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, знаков операций, разделителей и ключевых (служебных) слов входного языка.

В большинстве компиляторов лексический и синтаксический анализаторы - это взаимосвязанные части. Где провести границу между лексическим и синтаксическим анализом, какие конструкции анализировать сканером, а какие – синтаксическим распознавателем, решает разработчик компилятора. Как правило, любой анализ стремятся выполнить на этапе лексического разбора входной программы, если он может быть там выполнен. Возможности лексического анализатора ограничены по сравнению с синтаксическим анализатором, так как в его основе лежат более простые механизмы.

# 

# 2.2 Таблица лексем и содержащаяся в ней информация

Результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем с учетом характеристик каждой лексемы. Этот перечень лексем можно представить в виде таблицы, называемой таблицей лексем. Каждой лексеме в таблице лексем соответствует некий уникальный условный код, зависящий от типа лексемы, и дополнительная служебная информация. Таблица лексем в каждой строке должна содержать информацию о виде лексемы, ее типе и, возможно, значении. Обычно структуры данных, служащие для организации такой таблицы, имеют два поля: первое - тип лексемы, второе - указатель на информацию о лексеме.

Кроме того, информация о некоторых типах лексем, найденных в исходной программе, должна помешаться в таблицу идентификаторов (или в одну из таблиц идентификаторов, если компилятор предусматривает различные таблицы идентификаторов для различных типов лексем).

Не следует путать таблицу лексем и таблицу идентификаторов - это две принципиально разные таблицы, обрабатываемые лексическим анализатором.

Таблица лексем фактически содержит весь текст исходной программы, обработанный лексическим анализатором. В нее входят все возможные типы лексем, кроме того, любая лексема может встречаться в ней любое количество раз. Таблица идентификаторов содержит только определенные типы лексем – идентификаторы и константы. В нее не попадают такие лексемы, как ключевые (служебные) слова входного языка, знаки операций и разделители. Кроме того, каждая лексема (идентификатор или константа) может встречаться в таблице идентификаторов только один раз. Также можно отметить, что лексемы в таблице лексем обязательно располагаются в том же порядке, что и в исходной программе (порядок лексем в ней не меняется), а в таблице идентификаторов лексемы располагаются в любом порядке так, чтобы обеспечить удобство поиска.

# 2.3 Построение лексических анализаторов (сканеров)

Лексический анализатор имеет дело с такими объектами, как различного рода константы и идентификаторы (к последним относятся и ключевые слова). Язык описания констант и идентификаторов в большинстве случаев является регулярным, то есть может быть описан с помощью регулярных грамматик. Распознавателями для регулярных языков являются конечные автоматы (КА). Существуют правила, с помощью которых для любой регулярной грамматики может быть построен КА, распознающий цепочки языка, заданного этой грамматикой.

Любой КА может быть задан с помощью пяти параметров: M(Q,∑,δ,qo,F),

где:

Q - конечное множество состояний автомата;

∑ - конечное множество допустимых входных символов (входной алфавит КА);

δ - заданное отображение множества Q\*∑ во множество подмножеств P(Q)δ: Q\*∑→ P(Q) (иногда δ называют функцией переходов автомата);

q0 Q - начальное состояние автомата;

F Q. - множество заключительных состояний автомата.

Другим способом описания КА является граф переходов – графическое представление множества состояний и функции переходов КА. Граф переходов КА – это нагруженный однонаправленный граф, в котором вершины представляют состояния КА, дуги отображают переходы из одного состояния в другое, а символы нагрузки (пометки) дуг соответствуют функции перехода КА. Если функция перехода КА предусматривает переход из состояния q в q' по нескольким символам, то между ними строится одна дуга, которая помечается всеми символами, по которым происходит переход из q в q'.

Недетерминированный КА неудобен для анализа цепочек, так как в нем могут встречаться состояния, допускающие неоднозначность, то есть такие, из которых выходит две или более дуги, помеченные одним и тем же символом. Очевидно, что программирование работы такого КА - нетривиальная задача. Для простого программирования функционирования КА M(Q,∑,δ,qo,F) он должен быть детерминированным - в каждом из возможных состояний этого КА для любого входного символа функция перехода должна содержать не более одного состояния.

Доказано, что любой недетерминированный КА может быть преобразован в детерминированный КА так, чтобы их языки совпадали (говорят, что эти КА эквивалентны).

Кроме преобразования в детерминированный КА любой КА может быть минимизирован - для него может быть построен эквивалентный ему детерминированный КА с минимально возможным количеством состояний.

Можно написать функцию, отражающую функционирование любого детерминированного КА. Чтобы запрограммировать такую функцию, достаточно иметь переменную, которая бы отображала текущее состояние КА, а переходы из одного состояния в другое на основе символов входной цепочки могут быть построены с помощью операторов выбора. Работа функции должна продолжаться до тех пор, пока не будет достигнут конец входной цепочки. Для вычисления результата функции необходимо по ее завершении проанализировать состояние КА. Если это одно из конечных состояний, то функция выполнена успешно и входная цепочка принимается, если нет, то входная цепочка не принадлежит заданному языку.

Однако в общем случае задача лексического анализатора шире, чем просто проверка цепочки символов лексемы на соответствие ее входному языку. Он должен правильно определить конец лексемы (об этом было сказано выше) и выполнить те или иные действия по запоминанию распознанной лексемы (занесение ее в таблицу лексем). Набор выполняемых действий определяется реализацией компилятора. Обычно эти действия выполняются сразу же при обнаружении конца распознаваемой лексемы.

Во входном тексте лексемы не ограничены специальными символами. Определение границ лексем — это выделение тех строк в общем потоке входных символов, для которых надо выполнять распознавание. Если границы лексем всегда определяются (а выше было принято именно такое соглашение), то их можно определить по заданным терминальным символам и по символам начала следующей лексемы. Терминальные символы - это пробелы, знаки операций, символы комментариев, а также разделители (запятые, точки е запятой и др.). Набор таких терминальных символов может варьироваться в зависимости от входного языка. Важно отметить, что знаки операций сами также являются лексемами и необходимо не пропустить их при распознавании текста.

Таким образом, алгоритм работы простейшего сканера можно описать так:

□ просматривается входной поток символов программы на исходном языке до обнаружения очередного символа, ограничивающего лексему;

□ для выбранной части входного потока выполняется функция распознавания лексемы;

□ при успешном распознавании информация о выделенной лексеме заносится в таблицу лексем, и алгоритм возвращается к первому этапу;

□ при неуспешном распознавании выдается сообщение об ошибке, а дальнейшие действия зависят от реализации сканера: либо его выполнение прекращается, либо делается попытка распознать следующую лексему (идет возврат к первому этапу алгоритма).

Работа программы-сканера продолжается до тех пор, пока не будут просмотрены все символы программы на исходном языке из входного потока.

# Заключение

В результате выполнения курсовой работы для заданного входного языка были построены отдельные части компилятора.

В первой части работы была разработана программа, которая получает на входе набор идентификаторов, организует таблицу идентификаторов методом упорядоченного списка и методом простого рехэширования с помощью произведения, позволяет осуществить многократный поиск идентификатора в этих таблицах.

Во второй части работы была написана программа, которая выполняет лексический анализ входного текста и порождает таблицу лексем с указанием их типов и значений.

Отдельные части компилятора, разработанные в данной курсовой работе, дают представление о технике и методах, лежащих в основе построения компиляторов.

# Список использованной литературы

1. Кампапиец Р.II. Манькоп Е.В., Филатов Н.Е. Системное программирование. Основы построения трансляторов: Учеб. пособие для высших и средних учебных заведений. – СПб.: КОРОНА Принт, 2000. – 256 с.
2. Системное программное обеспечение: Учебник для вузов/ А.Ю. Молчанов- СПб.: Питер, 2003.- 396 с.
3. http://trubetskoy1.narod.ru/index.html

# 

# Приложение 1

**Задание:**

Организовать таблицу идентификатором с помощью простого рехеширования с помощью произведения.

**Код программы:**

unit Unit1;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, ComCtrls, StdCtrls, ExtCtrls, Grids;

type

TForm1 = class(TForm)

OpenDialog1: TOpenDialog;

Panel1: TPanel;

GroupBox1: TGroupBox;

Button1: TButton;

Memo2: TMemo;

Button2: TButton;

Edit1: TEdit;

GroupBox2: TGroupBox;

StringGrid1: TStringGrid;

Label1: TLabel;

procedure Button1Click(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure Button2Click(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

const

hash\_min=Ord('0')+Ord('0')+Ord('0');

HASH\_MAX= Ord('z')+Ord('z')+Ord('z');

REHASH1 = 127;

REHASH2 =223;

var

Form1: TForm1;

lengtM:integer;

sName:string ;

Find,NumSTR:integer;

implementation

function VarHash(const sName:string):longint;

var

i:integer;

begin

for i:=1 to length(sname) do

begin

Result:=(Ord(sName[i])+Ord(sName[(Length(sName)+i) div 2]) \* i{HASH\_MIN}) mod (HASH\_MAX - HASH\_MIN+i) + HASH\_MiN;

if Result < HASH\_MIN then Result := HASH\_MIN;

end;

end;

{$R \*.dfm}

procedure TForm1.Button1Click(Sender: TObject);

var

fName,str:string;

i:integer;

begin

form1.OpenDialog1.Execute;

fname:=form1.OpenDialog1.FileName;

form1.Memo2.Lines.loadfromfile(fName);

form1.StringGrid1.RowCount:=memo2.Lines.Count+1;

NumSTR:=memo2.Lines.Count+1;

for i:=0 to NumSTR do

begin

//Заполнение таблицы идентификаторов

str:=memo2.Lines.Strings[i];

form1.StringGrid1.Cells[2,i+1]:=(str);

stringgrid1.Cells[0,i+1]:=inttostr(i);

stringgrid1.Cells[1,i+1]:=Inttostr(VarHash(str));

end;

end;

procedure TForm1.FormCreate(Sender: TObject);

begin

with stringgrid1 do

begin

ColCount:=3;

RowCount:=3;

cells[0,0]:='#Функции';

stringgrid1.ColWidths[1]:=110;

cells[1,0]:='Значение Функции';

stringgrid1.ColWidths[2]:=100;

cells[2,0]:='Строка';

end;

end;

procedure TForm1.Button2Click(Sender: TObject);

var i,n:integer;

begin

find:=0;

n:=VArHAsh(form1.Edit1.Text);

begin

for i:=1 to numstr do

if (strtoint(stringgrid1.Cells[1,i])=n) and (edit1.Text=stringgrid1.Cells[2,i])

then

begin

Find:=Find+1;

form1.Label1.Caption:='Найдено Элементов - '+inttostr(Find);

showmessage('Элемент '+stringgrid1.Cells[2,i]+' найден'+chr(13)+'Найдено Элементов - '+inttostr(Find));

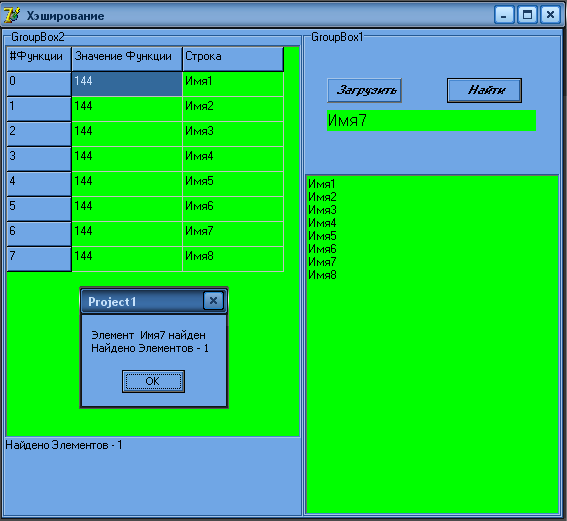
end;

end;

end;

end.

**Результат выполнения:**



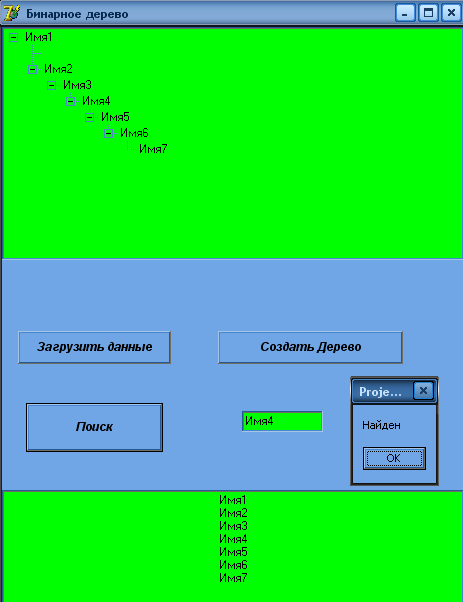
# Приложение 2

**Задание:**

Организовать таблицу идентификатором с помощью бинарного дерева.

**Код программы:**

**Результат выполнения:**



# Приложение 3

**Задание:**

Создать лексический анализатор арифметических выражений.

**Код программы:**

unit Program2;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, XPMan, ComCtrls, Buttons, Grids;

type

TForm1 = class(TForm)

Memo1: TMemo;

Button1: TButton;

ListBox1: TListBox;

Button2: TButton;

PageControl1: TPageControl;

TabSheet1: TTabSheet;

TabSheet2: TTabSheet;

GroupBox1: TGroupBox;

Edit1: TEdit;

Button3: TButton;

Button4: TButton;

OpenDialog1: TOpenDialog;

Button5: TButton;

Button6: TButton;

StringGrid1: TStringGrid;

StringGrid2: TStringGrid;

procedure Button1Click(Sender: TObject);

procedure Button2Click(Sender: TObject);

procedure Button3Click(Sender: TObject);

procedure Button4Click(Sender: TObject);

procedure Button5Click(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure Button6Click(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

//====================================

function StringToWords(T: string; Mode: Short; List: Tstrings = nil): integer;

var

i, z: integer;

s: string;

c: Char;

procedure Check;

begin

if (s > '') and (List <> nil) then

begin

List.Add(S);

z := z + 1;

end

else if (s > '') and (List = nil) then z := z + 1;

s := '';

end;

begin

i := 0;

z := 0;

s := '';

if t > '' then

begin

while i <= Length(t) + 1 do

begin

c := t[i];

case Mode of

0: {русские и английские слова}

if (c in ['a'..'z']) or (c in ['A'..'Z']) or (c in ['а'..'я']) or

(c in ['А'..'Я']) and (c <> ' ') then

s := s + c

else

Check;

1: {только русские слова}

if (c in ['а'..'я']) or (c in ['А'..'Я']) and (c <> ' ') then

s := s + c

else

Check;

2: {только английские слова}

if (c in ['a'..'z']) or (c in ['A'..'Z']) or (c in ['0'..'9']) or

(c in ['+','-','\*','/']) or (c in [':','=','(',')','.','\_',';','%']) and (c <> ' ') then

s := s + c

else

Check;

end;

i := i + 1;

end;

end;

result := z;

end;

//====================================

procedure TForm1.Button1Click(Sender: TObject);

var i, j, v: Integer;

c: Char;

begin

for i := 0 to Memo1.Lines.Count - 1 do

begin

StringToWords(Memo1.Lines.Strings[i], 2, ListBox1.Items);

end;

v := 1;

for i := 0 to ListBox1.Items.Count - 1 do

for j := 0 to StringGrid2.RowCount - 1 do

begin

if ListBox1.Items.Strings[i] = StringGrid2.Cells[0,j] then

begin

StringGrid1.RowCount := StringGrid1.RowCount + 1;

StringGrid1.Cells[0,v] := IntToStr(v);

StringGrid1.Cells[1,v] := StringGrid2.Cells[0,j];

StringGrid1.Cells[2,v] := StringGrid2.Cells[1,j];

v := v + 1;

Break;

end

else if j = StringGrid2.RowCount - 1 then

begin

StringGrid1.RowCount := StringGrid1.RowCount + 1;

StringGrid1.Cells[0,v] := IntToStr(v);

StringGrid1.Cells[1,v] := ListBox1.Items.Strings[i];

c := ListBox1.Items.Strings[i][1];

if (c in ['0'..'9']) then

StringGrid1.Cells[2,v] := 'Числовое значение'

else if ((c in ['%'])) then StringGrid1.Cells[2,v] := 'Символьная константа'

else StringGrid1.Cells[2,v] := 'Переменная';

v := v + 1;

end;

end;

end;

procedure TForm1.Button2Click(Sender: TObject);

begin

Memo1.Clear;

ListBox1.Clear;

end;

procedure TForm1.Button3Click(Sender: TObject);

begin

if OpenDialog1.Execute then

begin

Memo1.Lines.LoadFromFile(OpenDialog1.FileName);

Edit1.Text := OpenDialog1.FileName;

end;

end;

procedure TForm1.Button4Click(Sender: TObject);

begin

if FileExists(Edit1.Text) then

Memo1.Lines.LoadFromFile(Edit1.Text)

else MessageBox(Handle, 'Файл не найден.', 'Внимание', MB\_ICONINFORMATION);

end;

procedure TForm1.Button5Click(Sender: TObject);

begin

if Button5.Caption = '>' then

begin

Form1.Width := 854;

Button5.Caption := '<';

end

else

begin

Form1.Width := 680;

Button5.Caption := '>';

end;

end;

procedure TForm1.FormCreate(Sender: TObject);

begin

StringGrid1.Cells[0,0] := '№ п/п';

StringGrid1.Cells[1,0] := 'Лексема';

StringGrid1.Cells[2,0] := 'Значение';

StringGrid2.Cells[0,0] := '+';

StringGrid2.Cells[1,0] := 'Операция сложения';

StringGrid2.Cells[0,1] := '-';

StringGrid2.Cells[1,1] := 'Операция вычитания';

StringGrid2.Cells[0,2] := '\*';

StringGrid2.Cells[1,2] := 'Операция умножения';

StringGrid2.Cells[0,3] := '/';

StringGrid2.Cells[1,3] := 'Операция деления';

StringGrid2.Cells[0,4] := '(';

StringGrid2.Cells[1,4] := 'Открывающая скобка';

StringGrid2.Cells[0,5] := ')';

StringGrid2.Cells[1,5] := 'Закрывающая скобка';

StringGrid2.Cells[0,6] := ':-';

StringGrid2.Cells[1,6] := 'Операция присваивания';

StringGrid2.Cells[0,6] := ';';

StringGrid2.Cells[1,6] := 'Разделитель';

end;

procedure TForm1.Button6Click(Sender: TObject);

var i: Integer;

begin

for i := 1 to StringGrid1.RowCount - 1 do

StringGrid1.Rows[i].Clear;

StringGrid1.RowCount := 2;

end;

end.

**Результат выполнения:**

