ЛАБОРАТОРНАЯ РАБОТА № 1
РАБОТА С ТАБЛИЦЕЙ СИМВОЛОВ

Цель работы: изучить основные методы организации таблиц идентификаторов, получить представление о преимуществах и недостатках, присущих различным методам организации таблиц идентификаторов (таблиц символов или таблиц имен).

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Назначение таблицы идентификаторов

Проверка правильности семантики и генерация кода требуют знания характеристик идентификаторов, используемых в программе на исходном языке. Эти характеристики выясняются из описаний и из того, как идентификаторы используются в программе и накапливаются в ***таблице символов*** (или ***таблице идентификаторов***).

Любая таблица идентификаторов состоит из набора полей (ячеек таблицы), количество которых совпадает с количеством различных идентификаторов в исходной программе. Каждое поле содержит полную информацию об идентификаторе, которому соответствует данный элемент таблицы. Под идентификаторами подразумеваются имена констант, переменных, модулей, процедур и функций, а также формальные параметры процедур и функций. Компилятор может работать с одной или несколькими таблицам идентификаторов – их количество зависит от реализации компилятора.

Вне зависимости от реализации компилятора принцип его работы с таблицей идентификаторов остается одним и тем же – на различных фазах компиляции компилятор многократно обращается к этой таблице для поиска информации и записи новых данных. Причем поиск данных в таблице выполняется существенно чаще, чем запись в таблицу новых данных, поскольку в исходной программе использование ранее описанных идентификаторов встречается существенно чаще, чем описания новых идентификаторов, да и для каждого встреченного описания компилятор должен проверить существование такого же идентификатора, чтобы исключить дублирование их имен.

Отсюда можно сделать вывод, что таблицы идентификаторов должны быть организованы таким образом, чтобы компилятор имел возможность максимально быстрого поиска нужного ему элемента. Каждый элемент таблицы идентификаторов характеризуется уникальным именем. Даже в том случае, когда исходный язык допускает совпадающие имена идентификаторов – например, при различных областях видимости идентификаторов – в компиляторе существуют средства обеспечения уникальности их имен в пределах всей исходной программы. Поэтому поиск элемента в таблице выполняется по имени идентификатора, которое считается уникальным в пределах одной таблицы идентификаторов.

Простейшие способы организации таблицы идентификаторов

Таблица идентификаторов в виде простого линейного списка

Самый простой способ организации таблицы идентификаторов состоит в том, чтобы добавлять элементы в таблицу в порядке их поступления. В этом случае таблица идентификаторов будет представлять собой простой линейный список или массив с переменным количеством элементов. Тогда добавление нового элемента (нового идентификатора) в таблицу практически не будет требовать затрат вычислительных ресурсов – время записи нового элемента пренебрежимо мало: $T_з \sim 0$. Но поиск идентификатора в таблице в этом случае потребует последовательного сравнения искомого элемента с каждым элементом таблицы, пока не будет найден подходящий.

Для таблицы, содержащей N элементов, в среднем будет выполнено $N/2$ сравнений – тогда время поиска можно оценить как: $T_n \sim N/2$. Если N велико, то этот способ нельзя признать эффективным.

Таблица идентификаторов в виде упорядоченного списка

Поиск может быть выполнен более эффективно, если элементы таблицы упорядочены (отсортированы) согласно некоторому естественному порядку. В данном случае, когда поиск будет осуществляться по имени идентификатора, наиболее естественным будет расположить элементы таблицы в алфавитном порядке. Эффективным методом поиска в упорядоченном списке из N элементов является бинарный или логарифмический поиск. Символ S, который следует найти, сравнивается с элементом в середине таблицы: (N div 2)+1. Если они совпадают, то искомый элемент найден. Иначе, если этот элемент не является искомым (не совпадает с S), то необходимо просмотреть только блок элементов, пронумерованных от 1 до (N div 2), или блок элементов от (N div 2)+2 до N в зависимости от того, меньше искомый элемент, чем S или больше. Затем эта же операция повторяется над блоком меньшего размера. Так как на каждом шаге число элементов, которые могут содержать S, сокращается в два раза, то максимальное число сравнений равно $1+[\log_2 N]$. Тогда время поиска идентификатора можно оценить как: $T_n \sim 1+[\log_2 N]$.

Для сравнения: при N=128 бинарный поиск требует самое большее 8 сравнений, поиск в неупорядоченной таблице – в среднем 64 сравнения.

Однако для применения алгоритма бинарного поиска массив элементов должен быть упорядоченным. Это значит, что при добавлении нового идентификатора его нельзя просто добавить в конец таблицы – необходимо сначала найти место в таблице, в которое следует записать этот идентификатор, потом сдвинуть массив данных, находящихся справа от найденного места (чтобы освободить ячейку для нового элемента), и только потом записать элемент в таблицу. Таким образом, время записи нового идентификатора в таблицу можно оценить как: $T_з \sim 1+[\log_2 N]+k*N/2$, где k – это коэффициент, отражающий соотношение вычислительных затрат на операции сравнения и записи данных одного элемента таблицы.

Если бы операции поиска и добавления новых элементов в таблице идентификаторов выполнялись с одинаковой частотой, то преимущество метода бинарного поиска было бы ничтожным: сокращая время поиска с $N/2$ до $1+[\log_2 N]$ он в то же время увеличивает время добавления нового элемента с 0 до $1+[\log_2 N]+k*N/2$. Однако в таблице идентификаторов поиск элементов выполняется существенно чаще, чем добавление новых элементов в таблицу, поэтому применительно к таблице идентификаторов метод бинарного поиска является более эффективным, чем простой линейный список [2, 8, 14].

Таблица идентификаторов в форме бинарного дерева

Существует еще один простейший метод построения таблиц идентификаторов, при котором таблица имеет форму бинарного дерева. Каждый узел такого дерева представляет собой элемент таблицы, у которого может быть не более двух дочерних элементов, причем корневой узел является первым элементом таблицы. При построении дерева элементы сравниваются между собой, и в зависимости от результатов, выбирается путь в дереве.

Рассмотрим алгоритм заполнения бинарного дерева. Будем считать, что алгоритм работает с потоком входных данных, содержащим идентификаторы (в компиляторе этот поток данных порождается в процессе разбора текста исходной программы). Первый идентификатор помещается в вершину дерева. Все дальнейшие идентификаторы попадают в дерево по следующему алгоритму:

Шаг 1. Выбрать очередной идентификатор из входного потока данных. Если очередного идентификатора нет, то построение дерева закончено.

Шаг 2. Сделать текущим узлом дерева корневую вершину.

Шаг 3. Сравнить очередной идентификатор с идентификатором, содержащимся в текущем узле дерева.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, если равен – сообщить об ошибке и прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно!), иначе – перейти к шагу 7.

Шаг 5. Если у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 6.

Шаг 6. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

Шаг 7. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 8.

Шаг 8. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину правой вершиной текущего узла и вернуться к шагу 1.

На рис.1 показана последовательность построения таблицы идентификаторов в форме бинарного дерева для входного потока идентификаторов: *G D1 M EA2 AR B FE3*. На рис.1,а показана таблица с одним корневым элементом для идентификатора *G*. Теперь надо записать идентификатор *D1*. Для него выбирается левая ветвь от корня дерева, так как *D1<G* (рис.1,б). Далее запишем идентификатор *M*. Так как *M>G*, для *M* выбирается правая ветвь от узла *G* (рис.1,б). Наконец, запишем идентификатор *EA2*. Так как *EA2<G*, то идем по левой дуге от узла *G* и попадаем в узел *D1*, *EA2>D*, поэтому *EA2* попадает в правую ветвь от узла *D1* (рис.1,в). На рис.1,г изображено дерево после того, как в него также были добавлены идентификаторы *AR, B* и *FE3*.

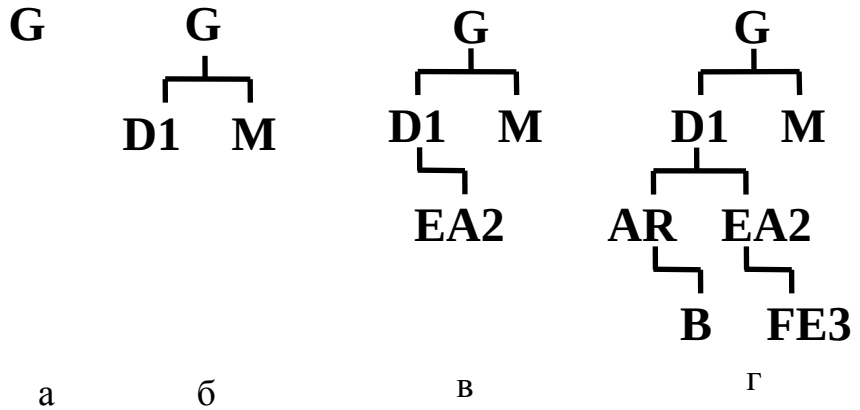


Рис. 1. Пример таблицы идентификаторов в форме бинарного дерева

Поиск нужного элемента в таблице идентификаторов в форме бинарного дерева выполняется по алгоритму, схожему с алгоритмом заполнения этого дерева:

Шаг 1. Сделать текущим узлом дерева корневую вершину.

Шаг 2. Сравнить искомый идентификатор с идентификатором, содержащимся в текущем узле дерева.

Шаг 3. Если идентификаторы совпадают, то искомый идентификатор найден, алгоритм поиска завершен, иначе надо перейти к шагу 4.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, иначе — перейти к шагу 6.

Шаг 5. Если у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершен.

Шаг 7. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершен.

Например, произведем поиск в дереве, изображенном на рис. 1, идентификатора *AR*. Берем корневую вершину *G* (она становится текущим узлом), сравниваем идентификаторы *G* и *AR*. Искомый идентификатор меньше – текущим узлом становится левая вершина *D1*. Опять сравниваем идентификаторы. Искомый идентификатор меньше – текущим узлом становится левая вершина *AR*. При следующем сравнении искомый идентификатор найден.

Если искать отсутствующий в таблице идентификатор – например, *AZ1*, то поиск опять пойдет от корневой вершины. Сравниваем идентификаторы *G* и *AZ1*. Искомый идентификатор меньше – текущим узлом становится левая вершина *D1*. Опять сравниваем идентификаторы. Искомый идентификатор меньше – текущим узлом становится левая вершина *AR*. Снова выполняем сравнение. Искомый идентификатор больше – текущим узлом становится правая вершина *B*. Сравниваем *AZ1* и *B*. Искомый идентификатор меньше, но левая вершина у узла *B* отсутствует, поэтому в данном случае искомый идентификатор не найден.

Для данного метода количество требуемых сравнений и форма получившегося дерева зависят от порядка, в котором идентификаторы поступают на вход. В среднем при большом количестве идентификаторов можно считать, что метод бинарного дерева, как и метод бинарного поиска на каждом шаге сокращает количество искомых элементов в два раза. Тогда среднее время поиска идентификатора в дереве можно оценить как: $T_п$~1+[$\log_2 N$], где N – количество идентификаторов в таблице. Но поскольку запись нового идентификатора в дерево выполняется по аналогичному алгоритму и, в отличие от упорядоченного массива, не требует сдвига данных, среднее время записи нового идентификатора в дерево также можно оценить как: $T_з$~1+[$\log_2 N$].

Как видно, среднестатистически метод организации таблицы идентификаторов в форме бинарного дерева имеет лучшие характеристики, чем метод, основанный на упорядоченном линейном списке за счёт отсутствия необходимости сдвигать уже существующие элементы таблицы при добавлении нового идентификатора. Однако метод, основанный на бинарном дереве, имеет два основных недостатка.

Первый его недостаток связан с самой формой организации таблицы – деревом. Для размещения бинарного дерева в памяти компьютера потребуются накладные расходы вычислительных ресурсов, вызванные необходимостью работы с динамической памятью при размещении элементов дерева, а также необходимостью хранения двух ссылок на дочерние элементы в каждом элементе таблицы. Этот недостаток можно считать несущественным.

Второй недостаток данного метода связан с тем, что форма дерева, а значит, и эффективность метода существенно зависят от порядка поступления идентификаторов на вход. В частности, если идентификаторы поступают на вход упорядоченно (в прямом или обратном алфавитном порядке – например, *A B C C22 D1*), построенное дерево получается полностью вырожденным в одну единственную ветвь и эффективность метода теряется. Но в реальных компьютерных программах идентификаторы поступают на вход неупорядоченно, поэтому данным недостатком можно пренебречь.

Таким образом, с учётом условий применения для ренальных входных программ метод построения таблицы идентификаторов в форме бинарного дерева достаточно эффективен. Данный метод применялся в реальных компиляторах.

Хеш-функции и хеш-адресация

Принципы работы хеш-функций

Логарифмическая зависимость времени поиска и времени заполнения таблицы идентификаторов от количества идентификаторов – это самый лучший результат, которого можно достичь за счет применения простейших методов организации таблиц. Однако в реальных исходных программах количество идентификаторов столь велико (десятки тысяч и более), что даже логарифмическую зависимость времени поиска от их числа нельзя признать удовлетворительной. Необходимы более эффективные методы поиска информации в таблице идентификаторов.

Лучших результатов можно достичь, если применить методы, связанные с использованием хеш-функций и хеш-адресации.

Хеш-функцией F называется некоторое отображение множества входных элементов **R** на множество целых неотрицательных чисел **Z**: F(r) = n, $r \in \mathbf{R}$, $n \in \mathbf{Z}$. Сам термин «хеш-функция»

происходит от английского термина «hash function» (hash — «мешать», «смешивать», «путать»). Множество допустимых входных элементов **R** называется областью определения хеш-функции. Множеством значений хеш-функции F называется подмножество **M** из множества целых неотрицательных чисел **Z**: $\mathbf{M}\subseteq\mathbf{Z}$, содержащее все возможные значения, возвращаемые функцией F: $\forall r\in\mathbf{R}$: $F(r)\in\mathbf{M}$ и $\forall m\in\mathbf{M}$: $\exists r\in\mathbf{R}$: $F(r)=m$. Процесс отображения области определения хеш-функции на множество значений называется «хешированием».

При работе с таблицей идентификаторов хеш-функция должна выполнять отображение имен идентификаторов на множество целых неотрицательных чисел. Областью определения такой хеш-функции будет множество всех возможных имен идентификаторов.

Хеш-адресация заключается в использовании значения, возвращаемого хеш-функцией, в качестве адреса ячейки некоторого массива данных. Размер массива должен соответствовать области значений используемой хеш-функции. В реальном компиляторе область значений хеш-функции не должна превышать размер доступного адресного пространства компьютера.

Метод организации таблиц идентификаторов, основанный на использовании хеш-адресации, заключается в размещении каждого элемента таблицы в ячейке, адрес которой возвращает хеш-функция, вычисленная для этого элемента. Тогда в идеальном случае для размещения любого элемента в таблице идентификаторов достаточно вычислить его хеш-функцию и обратиться к нужной ячейке массива данных. Для поиска элемента в таблице необходимо вычислить хеш-функцию для искомого элемента и проверить, не является ли заданная ею ячейка массива пустой. Если она не пуста – элемент найден, если пуста – не найден. Первоначально все ячейки массива данных должны быть пустыми.

Этот метод весьма эффективен, поскольку как время размещения элемента в таблице, так и время его поиска определяются только временем, затрачиваемым на вычисление хеш-функции, которое в общем случае несопоставимо меньше времени, необходимого на многократные сравнения элементов таблицы.

Метод имеет два очевидных недостатка. Первый из них – неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать области значений хеш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше. Второй недостаток – необходимость соответствующего разумного выбора хеш-функции.

Построение таблиц идентификаторов на основе хеш-функций

Существуют различные варианты хеш-функций. Получение результата хеш-функции – «хеширование» – обычно достигается за счет выполнения над цепочкой символов некоторых простых арифметических и/или логических операций. Самой простой хеш-функцией для символа является код внутреннего представления в компьютере литеры символа. Эту хеш-функцию можно использовать и для цепочки символов, выбирая первый ее символ. Например, если двоичное представление символа `A` есть код 00100001_2, то результатом хеширования идентификатора `ATable` будет код 00100001_2.

Хеш-функция, предложенная выше, очевидно не удовлетворительна: при использовании такой хеш-функции возникнет проблема – двум различным идентификаторам, начинающимся с одной и той же буквы, будет соответствовать одно и то же значение хеш-функции. Тогда при хеш-адресации в одну ячейку таблицы идентификаторов по одному и тому же адресу должны быть помещены два различных идентификатора, что явно невозможно. Ситуация, когда двум или более идентификаторам соответствует одно и то же значение хеш-функции, называется ***коллизией***.

Естественно, что хеш-функция, допускающая коллизии, не может быть напрямую использована для хеш-адресации в таблице идентификаторов. Достаточно получить хотя бы один случай коллизии на всем множестве идентификаторов, чтобы такой хеш-функцией нельзя было пользоваться непосредственно. Но в примере взята самая элементарная хеш-функция. А возможно ли построить хеш-функцию, которая бы полностью исключала возникновение коллизий?

Для полного исключения коллизий хеш-функция должна быть взаимно однозначной: каждому элементу из области определения хеш-функции должно соответствовать одно значение из ее множества значений, и каждому значению из множества значений этой функции должен соответствовать только один элемент из ее области определения. Тогда двум произвольным элементам из области определения хеш-функции будут всегда соответствовать два различных ее значения. Теоретически для идентификаторов такую хеш-функцию построить можно, так как и ее область определения (все возможные имена идентификаторов), и область ее значений (целые неотрицательные числа) являются бесконечными счетными множествами. Теоретически можно организовать взаимно однозначное отображение одного счетного множества на другое.

Но на практике существует ограничение, делающее создание взаимно однозначной хеш-функции для идентификаторов невозможным. Дело в том, что в реальности область значений любой хеш-функции ограничена размером доступного адресного пространства компьютера. При организации хеш-адресации значение, используемое в качестве адреса таблицы идентификаторов, не может выходить за пределы, заданные разрядностью адреса компьютера. Множество адресов любого компьютера может быть велико, но оно всегда конечно, то есть ограничено. Организовать взаимно однозначное отображение бесконечного счётного множества на конечное даже теоретически невозможно. Можно учесть, что длина принимаемой во внимание части имени идентификатора в реальных компиляторах также ограничена – обычно она лежит в пределах от 32 до 128 символов (то есть, и область определения хеш-функции конечна). Но и тогда количество элементов в конечном множестве, составляющем область определения функции, будет превышать их количество в области значений функции (количество всех возможных идентификаторов больше количества допустимых адресов в современных компьютерах). Таким образом, создать взаимно однозначную хеш-функцию практически невозможно. Следовательно, невозможно избежать возникновения коллизий.

Поэтому на практике хеш-адресацию используют не напрямую, а в сочетании с каким-либо методом, позволяющим бороться с возникающими коллизиями. Часть таких методов рассмотрена далее.

Построение таблиц идентификаторов с помощью рехеширования

Для решения проблемы коллизии можно использовать много способов. Одним из них является метод ***рехеширования*** (или расстановки). Согласно этому методу, если для элемента A адрес $h(A)$, вычисленный с помощью хеш-функции h, указывает на уже занятую ячейку, то необходимо вычислить значение функции $n_1=h_1(A)$ и проверить занятость ячейки по адресу n_1. Если и она занята, то вычисляется значение $h_2(A)$ и так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение $h_i(A)$ совпадет с $h(A)$. В последнем случае считается, что таблица идентификаторов заполнена, и места в ней больше нет – тогда выдается информация об ошибке размещения идентификатора в таблице.

Такую таблицу идентификаторов можно организовать по следующему алгоритму размещения элемента:

Шаг 1: Вычислить значение хеш-функции $n = h(A)$ для нового элемента A.

Шаг 2: Если ячейка по адресу n пустая, то поместить в нее элемент A и завершить алгоритм, иначе $i:=1$ и перейти к шагу 3.

Шаг 3: Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая, то поместить в нее элемент A и завершить алгоритм, иначе перейти к шагу 4.

Шаг 4: Если $n = n_i$, то сообщить об ошибке и завершить алгоритм, иначе $i:=i+1$ и вернуться к шагу 3.

Поиск элемента A в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1: Вычислить значение хеш-функции $n = h(A)$ для искомого элемента A.

Шаг 2: Если ячейка по адресу n пустая, то элемент не найден, алгоритм завершен, иначе сравнить имя элемента в ячейке n с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершен, иначе i:=1 и перейти к шагу 3.

Шаг 3: Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая или $n = n_i$, то элемент не найден и алгоритм завершен, иначе сравнить имя элемента в ячейке n_i с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершен, иначе i:=i+1 и повторить шаг 3.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм размещает элементы в пустых ячейках таблицы, выбирая их определенным образом. При этом элементы могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хеш-функции, что приведет к возникновению новых, дополнительных коллизий. Таким образом, количество операций, необходимых для поиска или размещения в таблице элемента, зависит от заполненности таблицы.

Для организации таблицы идентификаторов по методу рехеширования необходимо определить все хеш-функции h_i для всех i. Чаще всего функции h_i определяют как некоторую модификацию хеш-функции h. Например, самым простым методом вычисления функции $h_i(A)$ является ее организация в виде $h_i(A) = (h(A) + p_i) \bmod N_m$, где p_i — некоторое вычисляемое целое число, а N_m – максимальное значение из области значений хеш-функции h. В свою очередь, самым простым подходом здесь будет положить $p_i = i$. Тогда получаем формулу $h_i(A) = (h(A)+i) \bmod N_m$. В этом случае при совпадении значений хеш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной хеш-функцией $h(A)$. Такой метод называется *простым рехешированием*.

Этот способ нельзя признать особенно удачным — при совпадении хеш-адресов элементы в таблице начинают группироваться вокруг них, что увеличивает число необходимых сравнений при поиске и размещении. Но даже такой примитивный метод рехеширования является достаточно эффективным средством организации таблицы идентификаторов при неполном ее заполнении.

Среднее время на помещение одного элемента в таблицу и на поиск элемента в таблице можно снизить, если применить более совершенный метод рехеширования. Одним из таких методов является использование в качестве p_i для функции $h_i(A)=(h(A)+p_i) \bmod N_m$ последовательности псевдослучайных целых чисел $p_1, p_2, \ldots, p_k$. При хорошем выборе генератора псевдослучайных чисел длина последовательности k будет $k=N_m$. В качестве самого простого варианта последовательности псевдослучайных чисел можно взять вычисление p_i по формуле $p_i=(i*K)/L$, где K и L должны быть простыми числами (L выбирают максимально близко к N_m, а K берут около $L/2$).

Существуют и другие методы организации функций рехеширования $h_i(A)$, основанные на квадратичных вычислениях или, например, на вычислении по формуле: $h_i(A) = (h(A)*i) \bmod N_m$, если N_m – простое число. В целом рехеширование позволяет добиться неплохих результатов для эффективного поиска элемента в таблице (лучших, чем бинарный поиск и бинарное дерево), но эффективность метода сильно зависит от заполненности таблицы идентификаторов и качества используемой хеш-функции – чем реже возникают коллизии, тем выше эффективность метода. Как показывают расчёты, при выборе качественной хеш-функции (например, хеширование по алгоритмам MD5 или MD6) метод простого рехеширования является более эффективным, чем бинарный поиск или бинарное дерево при заполненности таблицы идентификаторов до 80%. Требование неполного заполнения таблицы ведет к неэффективному использованию объема доступной памяти.

Несмотря на указанные недостатки, метод организации таблицы идентификаторов на основе рехеширования имеет практическое применение и встречается в реализациях реальных компиляторов.

Построение таблиц идентификаторов по методу цепочек

Неполное заполнение таблицы идентификаторов при применении хеш-функций ведет к неэффективному использованию всего объема памяти, доступного компилятору. Причем объем

неиспользуемой памяти будет тем выше, чем больше информации хранится для каждого идентификатора. Этого недостатка можно избежать, если дополнить таблицу идентификаторов некоторой промежуточной хеш-таблицей.

В ячейках хеш-таблицы может храниться либо пустое значение, либо значение указателя на некоторую область памяти из основной таблицы идентификаторов. Тогда хеш-функция вычисляет адрес, по которому происходит обращение сначала к хеш-таблице, а потом уже через нее по найденному адресу – к самой таблице идентификаторов. Если соответствующая ячейка таблицы идентификаторов пуста, то ячейка хеш-таблицы будет содержать пустое значение. Тогда вовсе не обязательно иметь в самой таблице идентификаторов ячейку для каждого возможного значения хеш-функции – таблицу можно сделать динамической так, чтобы ее объем рос по мере заполнения.

Такой подход позволяет добиться двух положительных результатов: во-первых, нет необходимости заполнять пустыми значениями таблицу идентификаторов – это можно сделать только для хеш-таблицы; во-вторых, каждому идентификатору будет соответствовать строго одна ячейка в таблице идентификаторов (в ней не будет пустых неиспользуемых ячеек). Пустые ячейки будут только в хеш-таблице, и объем неиспользуемой памяти не будет зависеть от объема информации, хранимой для каждого идентификатора – для каждого значения хеш-функции будет расходоваться только память, необходимая для хранения одного указателя.

На основе этой схемы можно реализовать еще один способ организации таблиц идентификаторов с помощью хеш-функций, называемый «***метод цепочек***». Для метода цепочек в таблице идентификаторов к каждому элементу добавляется еще одно поле, в котором может содержаться ссылка на любой элемент таблицы. Первоначально это поле всегда пустое (никуда не указывает). Также для этого метода необходимо иметь одну специальную переменную, которая всегда указывает на первую свободную ячейку основной таблицы идентификаторов (первоначально – указывает на начало таблицы).

Метод цепочек работает следующим образом по следующему алгоритму:

Шаг 1: Во все ячейки хеш-таблицы поместить пустое значение, таблица идентификаторов пуста, переменная *FreePtr* (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; i:=1.

Шаг 2: Вычислить значение хеш-функции n_i для нового элемента A_i. Если ячейка хеш-таблицы по адресу n_i пустая, то поместить в нее значение переменной *FreePtr* и перейти к шагу 5; иначе перейти к шагу 3.

Шаг 3: Выбрать из хеш-таблицы адрес ячейки таблицы идентификаторов m и перейти к шагу 4.

Шаг 4: Для ячейки таблицы идентификаторов по адресу m проверить значение поля ссылки. Если оно пустое, то записать в него адрес из переменной *FreePtr* и перейти к шагу 5; иначе выбрать из поля ссылки адрес m и повторить шаг 4.

Шаг 5: Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A_i (поле ссылки должно быть пустым), в переменную *FreePtr* поместить адрес за концом добавленной ячейки. Если больше нет идентификаторов, которые надо разместить в таблице, то выполнение алгоритма закончено, иначе i:=i+1 и перейти к шагу 2.

Поиск элемента в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1: Вычислить значение хеш-функции n для искомого элемента A. Если ячейка хеш-таблицы по адресу n пустая, то элемент не найден и алгоритм завершен, иначе выбрать из хеш-таблицы адрес ячейки таблицы идентификаторов m.

Шаг 2: Сравнить имя элемента в ячейке таблицы идентификаторов по адресу m с именем искомого элемента A. Если они совпадают, то искомый элемент найден и алгоритм завершен, иначе перейти к шагу 3.

Шаг 3: Проверить значение поля ссылки в ячейке таблицы идентификаторов по адресу m. Если оно пустое, то искомый элемент не найден и алгоритм завершен; иначе выбрать из поля ссылки адрес m и перейти к шагу 2.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм размещает элементы в ячейках таблицы, связывая их друг с другом последовательно через поле ссылки. При этом элементы не могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хеш-функции. Таким образом, дополнительные коллизии не возникают. В итоге в таблице возникают своеобразные цепочки связанных элементов, откуда происходит и название данного метода – «метод цепочек».

Метод цепочек является очень эффективным средством организации таблиц идентификаторов. Среднее время на размещение одного элемента и на поиск элемента в таблице для него зависит только от среднего числа коллизий, возникающих при вычислении хеш-функции. Накладные расходы памяти, связанные с необходимостью иметь одно дополнительное поле указателя в таблице идентификаторов на каждый ее элемент, можно признать вполне оправданными. Этот метод позволяет более экономно использовать память, чем метод, основанный на рехешировании, но при этом требует организации работы с динамическими массивами данных. Поэтому метод цепочек сложнее в реализации, чем метод рехеширования.

Метод цепочек находит практическое применение в реальных компиляторах наряду с методом рехеширования. Какой из методов организации таблиц идентификаторов использовать в каждом конкретном случае решают разработчики компилятора.

Комбинированные способы построения таблиц идентификаторов

Выше в примере была рассмотрена весьма примитивная хеш-функция, которую никак нельзя назвать удовлетворительной. Хорошая хеш-функция распределяет поступающие на ее вход идентификаторы равномерно на все имеющиеся в распоряжении адреса, так что коллизии возникают не столь часто. Существует большое множество хеш-функций. Каждая из них стремится распределить адреса под идентификаторы по своему алгоритму, но, как было показано выше, идеального хеширования достичь невозможно.

То, какой конкретно метод применяется в компиляторе для организации таблиц идентификаторов, зависит от реализации компилятора. Один и тот же компилятор может иметь даже несколько разных таблиц идентификаторов, организованных на основе различных методов.

Как правило, применяются комбинированные методы. В этом случае, как и для метода цепочек, в таблице идентификаторов организуется специальное дополнительное поле ссылки. Но в отличие от метода цепочек оно имеет несколько иное значение. При отсутствии коллизий для выборки информации из таблицы используется хеш-функция, поле ссылки остается пустым. Если же возникает коллизия, то через поле ссылки организуется поиск идентификаторов, для которых значения хеш-функции совпадают по одному из рассмотренных выше методов: неупорядоченный список, упорядоченный список или же бинарное дерево. При хорошо построенной хеш-функции коллизии будут возникать редко, поэтому количество идентификаторов, для которых значения хеш-функции совпали, будет не столь велико. Тогда и время поиска одного среди них будет незначительным (в принципе, при высоком качестве хеш-функции подойдет даже перебор по неупорядоченному списку).

Такой подход имеет преимущество по сравнению с методом цепочек, поскольку не требует использования промежуточной хеш-таблицы. Недостатком метода является необходимость работы с динамически распределяемыми областями памяти. Эффективность такого метода, очевидно, в первую очередь зависит от качества применяемой хеш-функции, а во вторую — от метода организации дополнительных хранилищ данных.

Хеш-адресация – это метод, который применяется не только для организации таблиц идентификаторов в компиляторах. Данный метод нашел свое применение и в операционных системах, и в системах управления базами данных.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя.

2. Разработать алгоритм организации таблицы идентификаторов в соответствии с заданием.
3. Подготовить и защитить отчет.
4. Написать и отладить программу на ЭВМ.
5. Сдать работающую программу преподавателю.

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет по лабораторной работе должен содержать следующие разделы:

- Краткое изложение цели работы.
- Задание по лабораторной работе с описанием своего варианта.
- Схему организации хеш-таблицы (в соответствии с вариантом задания).
- Описание алгоритма поиска в хеш-таблице (в соответствии с вариантом задания).
- Выводы по проделанной работе.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое таблица символов (таблица идентификаторов) и для чего она предназначена?
2. Какая информация может храниться в таблице символов?
3. Какие операции выполняет компилятор с таблицей символов? В чем особенность работы компилятора с этой таблицей?
4. Какие цели преследуются при организации таблицы символов?
5. Какими характеристиками могут обладать идентификаторы (именованные константы, переменные)?
6. Какие существуют простейшие способы организации таблиц символов?
7. В чем заключается алгоритм логарифмического поиска? Какие преимущества он дает по сравнению с простым перебором и какие он имеет недостатки?
8. Расскажите о древовидной организации таблиц идентификаторов. В чем ее преимущества и недостатки?
9. Что такое хеш-функции и для чего они используются?
10. В чем суть хеш-адресации?
11. Расскажите о преимуществах и недостатках организации таблицы идентификаторов с помощью хеш-функции.
12. Что такое коллизия? Почему она происходит? Можно ли полностью избежать коллизий?
13. Что такое рехеширование? Какие методы рехеширования существуют?
14. В чем заключается метод цепочек?
15. Сравните между собой методы организации таблицы идентификаторов на основе рехеширования и с помощью метода цепочек.
16. Как могут быть скомбинированы различные методы организации хеш-таблиц?

ВАРИАНТЫ ЗАДАНИЙ

Для выполнения лабораторной работы требуется написать программу, которая получает на входе набор идентификаторов, организует таблицу по заданному методу и позволяет осуществить **многократный** поиск идентификатора в этой таблице. Список идентификаторов считать заданным в виде текстового файла. Длину идентификаторов можно считать ограниченной 32 символами.

Для организации таблицы используется простейшая хэш-функция, указанная в варианте задания, а при возникновении коллизий используется дополнительный метод размещения идентификаторов в памяти. Если в качестве этого метода используется дерево или список, то они должны быть связаны с элементом главной хэш-таблицы.

В каждом варианте требуется, чтобы программа сообщала среднее число коллизий и среднее количество сравнений, выполненных для поиска идентификатора. Варианты заданий приведены далее в таблице 1.

Таблица 1.

Варианты заданий для лабораторной работы №1

№	Тип хеш-функции (таблицы)	Способ разрешения коллизий
1.	Сумма кодов первой и второй букв	Бинарное дерево
2.	Сумма кодов первой и второй букв	Список с простым перебором
3.	Сумма кодов первой и второй букв	Упорядоченный список с логарифмическим поиском
4.	Сумма кодов первой и второй букв	Простое рехеширование
5.	Сумма кодов первой и второй букв	Рехеширование на основе псевдослучайных чисел
6.	Сумма кодов первой и второй букв	Метод цепочек
7.	Сумма кодов первой и последней букв	Бинарное дерево
8.	Сумма кодов первой и последней букв	Список с простым перебором
9.	Сумма кодов первой и последней букв	Упорядоченный список с логарифмическим поиском
10.	Сумма кодов первой и последней букв	Простое рехеширование
11.	Сумма кодов первой и последней букв	Рехеширование на основе псевдослучайных чисел
12.	Сумма кодов первой и последней букв	Метод цепочек
13.	Сумма кодов первой и средней букв	Бинарное дерево
14.	Сумма кодов первой и средней букв	Список с простым перебором
15.	Сумма кодов первой и средней букв	Упорядоченный список с логарифмическим поиском
16.	Сумма кодов первой и средней букв	Простое рехеширование
17.	Сумма кодов первой и средней букв	Рехеширование на основе псевдослучайных чисел
18.	Сумма кодов первой и средней букв	Метод цепочек
19.	Сумма кодов средней и последней букв	Бинарное дерево
20.	Сумма кодов средней и последней букв	Список с простым перебором
21.	Сумма кодов средней и последней букв	Упорядоченный список с логарифмическим поиском
22.	Сумма кодов средней и последней букв	Простое рехеширование
23.	Сумма кодов средней и последней букв	Рехеширование на основе псевдослучайных чисел
24.	Сумма кодов средней и последней букв	Метод цепочек

ЛАБОРАТОРНАЯ РАБОТА № 2
ПОСТРОЕНИЕ РАСПОЗНАВАТЕЛЯ ДЛЯ РЕГУЛЯРНОЙ ГРАММАТИКИ

Цель работы: изучение основных понятий теории регулярных языков и грамматик, ознакомление с назначением и принципами работы конечных автоматов (КА), получение практических навыков построения КА на основе заданной регулярной грамматики.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Функции лексического анализатора

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

Лексема (лексическая единица языка) – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц.

Результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем. 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. Кроме того, информация о некоторых типах лексем, найденных в исходной программе, должна помещаться в таблицу идентификаторов (или в одну из таблиц идентификаторов, если компилятор предусматривает различные таблицы идентификаторов для различных типов лексем).

С теоретической точки зрения лексический анализатор не является обязательной 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. Эти причины заключаются в следующем:

- 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 выкидывает всю незначащую информацию;
- для выделения в тексте и разбора лексем возможно применять простую, эффективную и теоретически хорошо проработанную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;
- сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходный 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 сканер.

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, а также ключевых (служебных) слов входного языка.

Лексический анализатор имеет дело с таким объектами, как различного рода константы и идентификаторы (к последним относятся и ключевые слова). Язык констант и идентификаторов в большинстве случаев является *регулярным* – то есть, может быть описан с помощью регулярных (праволинейных или леволинейных) грамматик [1, 2, 3, 8]. Распознавателями для регулярных языков являются конечные автоматы (КА). Существуют правила, с помощью которых для любой регулярной грамматики может быть построен недетерминированный КА, распознающий цепочки языка, заданного этой грамматикой.

Конечные автоматы

Определение конечного автомата

Как уже было сказано выше, лексемы могут быть описаны с помощью регулярных языков. Распознавателями для регулярных языков являются ***конечные автоматы*** (КА) [1, 2, 7, 10, 21]. Поэтому КА лежат в основе лексического анализа.

КА задается пятеркой:

$\mathbf{M}=(\mathbf{Q}, \mathbf{\Sigma}, \delta, q_0, \mathbf{F})$,

где:

Q - конечное множество состояний автомата;

Σ - конечное множество допустимых входных символов;

δ - заданное отображение множества $\mathbf{Q}^*\mathbf{\Sigma}$ во множество подмножеств $\mathbf{P}(\mathbf{Q})$ δ: $\mathbf{Q}^*\mathbf{\Sigma}\rightarrow\mathbf{P}(\mathbf{Q})$ (иногда δ называют *функцией переходов* автомата, которая может быть определена как: $\delta(q,a)=\mathbf{D}$, где $q\in\mathbf{Q}$ – текущее состояние КА, $a\in\mathbf{\Sigma}$ – текущий входной символ, а $\mathbf{D}\subseteq\mathbf{Q}$ – множество возможных следующих состояний КА);

$q_0 \in \mathbf{Q}$ - начальное состояние автомата;

$\mathbf{F} \subseteq \mathbf{Q}$ - множество заключительных состояний автомата.

Работа КА выполняется по тактам. На каждом очередном такте i автомат, находясь в некотором состоянии $q_i \in \mathbf{Q}$, считывает очередной символ $w \in \mathbf{\Sigma}$ из входной цепочки символов и изменяет свое состояние на $q_{i+1} \in \delta(q_i,w)$, после чего указатель в цепочке входных символов передвигается на следующий символ и начинается такт i+1. Так продолжается до тех пор, пока цепочка входных символов не закончится, либо КА попадёт в состояние, из которого нет переходов. Конец цепочки символов часто помечают особым символом ⊥. Считается также, что перед тактом 1 автомат находится в начальном состоянии q_0.

Говорят, что КА допускает цепочку $\alpha \in \mathbf{\Sigma}^*$, если в результате выполнения всех тактов работы над этой цепочкой он может оказаться в состоянии $q \in \mathbf{F}$. Язык, определяемый автоматом, является множеством всех цепочек, допускаемых данным автоматом.

Графически КА отображается нагруженным однонаправленным графом, в котором вершины представляют состояния КА, дуги отображают переходы из одного состояния в другое, а символы нагрузки (пометки) дуг соответствуют входным символам функции перехода. Если функция перехода 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'. Такой граф называют ***графом переходов*** КА.

Детерминированные и недетерминированные КА

КА называется *детерминированным*, если функция переходов этого автомата из любого состояния по любому входному символу содержит не более одного следующего состояния. В противном случае КА называется *недетерминированным*. Условие детерминированности КА можно сформулировать следующим образом: $\forall q \in \mathbf{Q}$, $\forall a \in \mathbf{\Sigma}$: либо $\delta(q,a)=\{q'\}$, либо $\delta(q,a)=\varnothing$.

Для анализа цепочки с помощью детерминированного КА достаточно подать ее на вход автомата, выполнить все такты его работы и определить, перешел ли автомат в результате работы в одно из заданных конечных состояний.

Недетерминированный КА неудобен для анализа цепочек, так как в нем могут встречаться состояния, допускающие неоднозначность, т.е. такие, из которых выходит две или более дуги, помеченных одним и тем же символом. Очевидно, что анализ цепочек и программирование работы для такого автомата – нетривиальная задача.

Доказано, что любой недетерминированный КА может быть преобразован в детерминированный так, чтобы их языки совпадали (говорят, что такие КА эквивалентны) [2, 7, 8, 21]. В отличие от обычного КА функция переходов детерминированного КА может быть определена как δ': $\mathbf{Q}^*\mathbf{\Sigma} \rightarrow \mathbf{Q}$ или $\delta(q,a)=q'$, где $q \in \mathbf{Q}$ – текущее состояние КА, $a \in \mathbf{\Sigma}$ – текущий входной символ, а $q' \in \mathbf{Q}$ – следующее состояние КА.

После построения детерминированный КА может быть минимизирован, т.е. для него может быть построен эквивалентный ему КА с минимально возможным числом состояний.

Построение КА на основе регулярной грамматики

Исходные данные для построения КА определяет регулярная грамматика, задающая язык, описывающий лексемы, которые должен уметь распознавать лексический анализатор. Таким образом, для построения лексического анализатора необходимо уметь строить КА на основе заданной регулярной грамматики. Такое построение выполняется с помощью несложного алгоритма в два этапа [2, 8, 21]:

- исходная регулярная грамматика преобразуется к автоматному виду;
- на основе автоматной грамматики строится КА.

В качестве примера рассмотрим лексический анализатор, выполняющий анализ лексем, представляющих собой целочисленные константы без знака в формате языка Си. В соответствии с требованиями языка, такие константы могут быть десятичными, восьмеричными, либо

шестнадцатеричными. Восьмеричной константой считается число, начинающееся с 0 и содержащее цифры от 0 до 7; шестнадцатеричная константа должна начинаться с последовательности символов 0x и может содержать цифры, а также буквы от A до F (будем рассматривать только прописные буквы). Остальные числа считаются десятичными (правила их записи напоминать, наверное, не стоит). Будем считать, что всякое число завершается символом конца строки ⊥.

Рассмотренные выше правила могут быть записаны в форме Бэкуса-Наура в грамматике целочисленных констант без знака языка Си (для избежания путаницы терминальные символы грамматики в правилах выделены жирным шрифтом).

G({S, G, X, H, Q, Z}, {0...9, A...F, x, ⊥}, **P**, S)

P: S→ G⊥|Z⊥|H⊥|Q⊥
G→ **1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**|G**0**|G**1**|G**2**|G**3**|G**4**|G**5**|G**6**|G**7**|G**8**|G**9**|Z**8**|Z**9**|Q**8**|Q**9**
H→ X**0**|X**1**|X**2**|X**3**|X**4**|X**5**|X**6**|X**7**|X**8**|X**9**|X**A**|X**B**|X**C**|X**D**|X**E**|X**F**|
H**0**|H**1**|H**2**|H**3**|H**4**|H**5**|H**6**|H**7**|H**8**|H**9**|H**A**|H**B**|H**C**|H**D**|H**E**|H**F**
X→ Z**x**
Q→ Z**0**|Z**1**|Z**2**|Z**3**|Z**4**|Z**5**|Z**6**|Z**7**|Q**0**|Q**1**|Q**2**|Q**3**|Q**4**|Q**5**|Q**6**|Q**7**
Z→ **0**

Данная грамматика является регулярной грамматикой (леволинейной). Она также является автоматной грамматикой, поэтому преобразование исходной грамматики к автоматному виду в данном случае не требуется. Следовательно, можно сразу построить КА.

Получим следующий КА:

M({N, Z, X, H, Q, G, S}, {0...9, A...F, x, ⊥}, δ, N, {S})

δ: δ(G,⊥)={S}, δ(Z,⊥)={S}, δ(H,⊥)={S}, δ(Q,⊥)={S},
δ(N,**1**)={G}, δ(N,**2**)={G}, δ(N,**3**)={G}, δ(N,**4**)={G}, δ(N,**5**)={G}, δ(N,**6**)={G}, δ(N,**7**)={G},
δ(N,**8**)={G}, δ(N,**9**)={G}, δ(G,**0**)={G}, δ(G,**1**)={G}, δ(G,**2**)={G}, δ(G,**3**)={G}, δ(G,**4**)={G},
δ(G,**5**)={G}, δ(G,**6**)={G}, δ(G,**7**)={G}, δ(G,**8**)={G}, δ(G,**9**)={G}, δ(Z,**8**)={G}, δ(Z,**9**)={G},
δ(Q,**8**)={G}, δ(Q,**9**)={G},
δ(X,**0**)={H}, δ(X,**1**)={H}, δ(X,**2**)={H}, δ(X,**3**)={H}, δ(X,**4**)={H}, δ(X,**5**)={H}, δ(X,**6**)={H},
δ(X,**7**)={H}, δ(X,**8**)={H}, δ(X,**9**)={H}, δ(X,**A**)={H}, δ(X,**B**)={H}, δ(X,**C**)={H}, δ(X,**D**)={H},
δ(X,**E**)={H}, δ(X,**F**)={H}, δ(H,**0**)={H}, δ(H,**1**)={H}, δ(H,**2**)={H}, δ(H,**3**)={H}, δ(H,**4**)={H},
δ(H,**5**)={H}, δ(H,**6**)={H}, δ(H,**7**)={H}, δ(H,**8**)={H}, δ(H,**9**)={H}, δ(H,**A**)={H}, δ(H,**B**)={H},
δ(H,**C**)={H}, δ(H,**D**)={H}, δ(H,**E**)={H}, δ(H,**F**)={H},
δ(Z,**x**)={X},
δ(Z,**0**)={Q}, δ(Z,**1**)={Q}, δ(Z,**2**)={Q}, δ(Z,**3**)={Q}, δ(Z,**4**)={Q}, δ(Z,**5**)={Q}, δ(Z,**6**)={Q},
δ(Z,**7**)={Q}, δ(Q,**0**)={Q}, δ(Q,**1**)={Q}, δ(Q,**2**)={Q}, δ(Q,**3**)={Q}, δ(Q,**4**)={Q}, δ(Q,**5**)={Q},
δ(Q,**6**)={Q}, δ(Q,**7**)={Q},
δ(N,**0**)={Z}

Построенный КА **M**({N,Z,X,H,Q,G,S}, {0...9,A...F,x,⊥}, δ, N, {S}) изображен на рис. 2 (без учёта дуг, показанных пунктирными линиями). Начальным состоянием данного КА является состояние *N*, множество конечных состояний содержит одно состояние *S*.

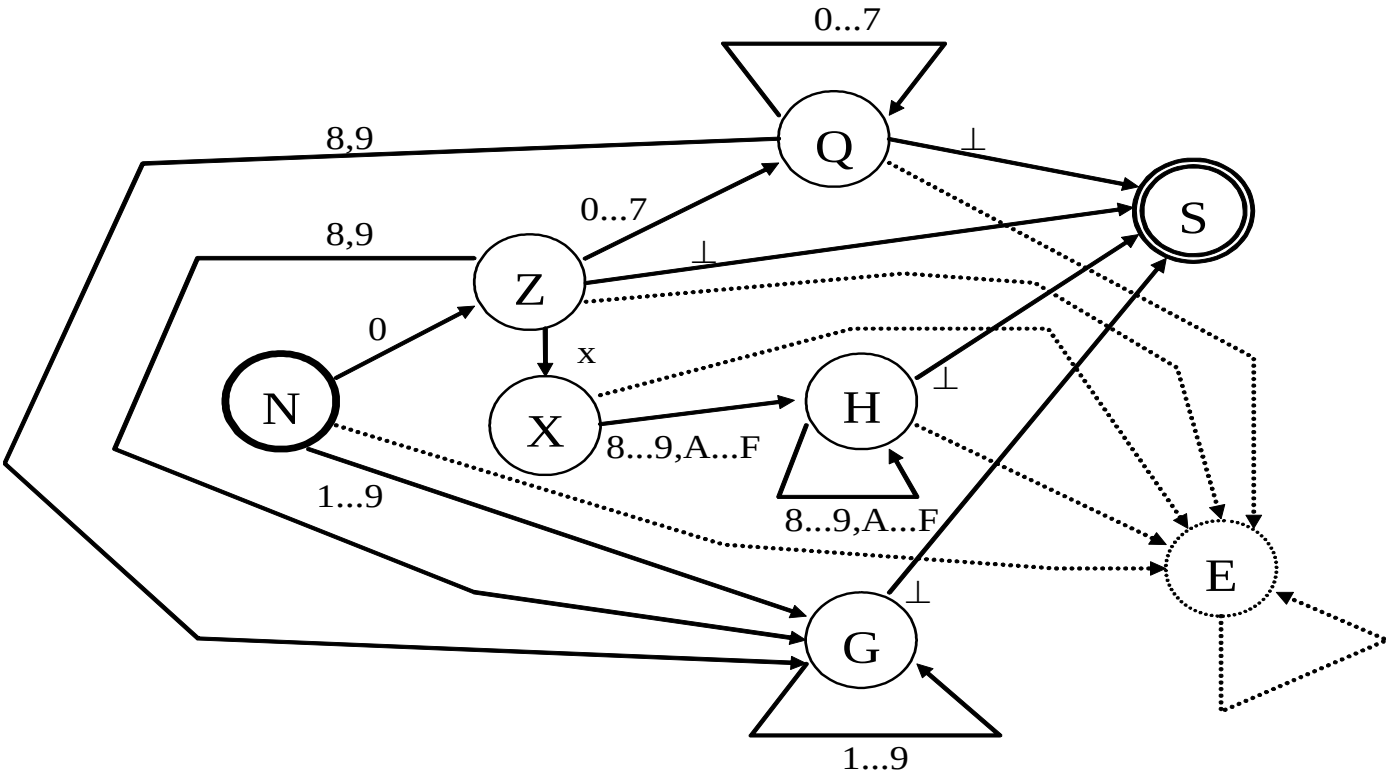


Рис. 2. Граф переходов детерминированного КА, распознающего целые числа языка Си

Очевидно, что построенный КА является детерминированным КА, но не является полностью определённым – в нём есть состояния, из которых нет переходов по одному или нескольким допустимым входным символам. Например, из состояния G нет перехода по допустимому входному символу **x**. Это может быть неудобным при программировании работы данного КА.

Чтобы данный КА стал полностью определённым, в него дополнительно введено особое состояние *E*, обозначающее ошибку в распознавании цепочки символов (на рис. 2 это состояние и идущие в него дуги обозначены пунктирными линиями). На графе автомата дуги, идущие в это состояние, не нагружены символами. По принятому соглашению они обозначают функцию перехода по любому символу, кроме тех символов, которыми уже помечены другие дуги, выходящие из той же вершины графа. Такое соглашение принято, чтобы не загромождать обозначениями граф автомата.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя. Самостоятельно выбрать используемый вид комментария.
2. Разработать регулярную грамматику лексем входного языка в соответствии с заданием с учётом выбранного вида комментариев.
3. Построить КА на основе регулярной грамматики входного языка.
4. Преобразовать КА к детерминированному виду (если необходимо).
5. Минимизировать КА (если необходимо).
6. Подготовить и защитить отчет.

Рекомендации:

1. Используемый вид комментария можно выбрать из любого языка программирования, либо придумать свой собственный вид комментария.
2. Идентификатором считается любая последовательность букв и цифр, начинающаяся с буквы (часто в идентификаторах допускается также знак подчёркивания – «_», иногда и другие символы). Для определения идентификаторов рекомендуется использовать правило грамматики: I → **б** | I**б** | I**ц**, где «**б**» обозначает любую букву, а «**ц**» - любую цифру.
3. Для упрощения структуры КА ключевые слова предлагается считать идентификаторами, а уже после их выделения выполнять проверку на совпадение найденного идентификатора с заданным перечнем ключевых слов. Более подробно об этом см. в [1, 2, 6].

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать следующие разделы:

- Краткое изложение цели работы.
- Задание по лабораторной работе с описанием своего варианта задания и выбранного вида комментария.
- Описание регулярной грамматики лексем входного языка в форме Бэкуса-Наура или в форме регулярных выражений (в соответствии с вариантом задания).
- Описание КА или граф переходов КА для распознавания лексем (в соответствии с вариантом задания).
- Выводы по проделанной работе.

Допускается оформлять единый (совместный) отчёт для лабораторных работ №2 и №3.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение цепочки символов, алфавита, языка. Что такое синтаксис и семантика языка?
2. Какие существуют методы задания языков?
3. Что такое транслятор, компилятор, интерпретатор? В чём их различие между собой?
4. Расскажите об общей структуре компилятора.
5. Что такое лексема? Расскажите, какие типы лексем существуют в языках программирования.
6. Какие функции выполняет лексический анализ в процессе компиляции?
7. Что такое грамматика? Дайте определения грамматики.
8. Как выглядит описание грамматики в форме Бэкуса-Наура.
9. Какие классы грамматик существуют?
10. Что такое регулярные грамматики? В чём их особенности?
11. Расскажите о распознавателях языка.
12. Что такое конечный автомат (КА)? Как можно задать КА?
13. Расскажите о том, как функционирует КА.
14. Дайте определение детерминированного и недетерминированного КА. Расскажите о возможности преобразования недетерминированного КА в детерминированный.
15. Объясните, как можно построить КА на основе регулярной грамматики.
16. Что такое автоматная грамматика? Как связаны между собой регулярные и автоматные грамматики.

ВАРИАНТЫ ЗАДАНИЙ

Все варианты заданий предусматривают возможность ***наличия комментариев неограниченной длины*** во входном языке. Форму организации комментариев предлагается выбрать самостоятельно.

Любые лексемы, не предусмотренные вариантом задания, встречающиеся в исходном тексте, должны трактоваться как ошибочные.

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

2. Входной язык содержит логические выражения, разделенные символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, констант **true** («истина») и **false** («ложь»), знака присваивания (**:=**), знаков операций **or**, **xor**, **and**, **not** и круглых скобок.

3. Входной язык содержит операторы условия типа **if** … **then** … **else** и **if** … **then**, разделенные символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой, знак присваивания (**:=**).

4. Входной язык содержит операторы цикла типа **for (**…**;** …**;** …**) do**, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой, знак присваивания (**:=**).

5. Входной язык содержит операторы выбора типа **case** … **of** … **end**, разделенные символом **;** (точка с запятой). Операторы выбора содержат идентификаторы, знак двоеточия (**:**), знаки операций **+**, **-**, десятичные числа с плавающей точкой, знак присваивания (**:=**).

6. Входной язык содержит операторы цикла типа **while** … **do** …, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, знаки операций **+**, **-**, десятичные числа с плавающей точкой, знак присваивания (**:=**).

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

8. Входной язык содержит логические выражения, разделенные символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, констант **0** и **1**, знака присваивания (**:=**), знаков операций **or**, **xor**, **and**, **not** и круглых скобок.

9. Входной язык содержит операторы условия типа **if** … **then** … **else** и **if** … **then**, разделенные символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**).

10. Входной язык содержит операторы цикла типа **for (**…**;** …**;** …**) do**, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**).

11. Входной язык содержит операторы выбора типа **case** … **of** … **end**, разделенные символом **;** (точка с запятой). Операторы выбора содержат идентификаторы, знак двоеточия (**:**), знаки операций **+**, **-**, римские числа, знак присваивания (**:=**).

12. Входной язык содержит операторы цикла типа **while** … **do** …, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, знаки операций **+**, **-**, римские числа, знак присваивания (**:=**).

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

14. Входной язык содержит логические выражения, разделенные символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (**:=**), знаков операций **or**, **xor**, **and**, **not** и круглых скобок.

15. Входной язык содержит операторы условия типа **if** … **then** … **else** и **if** … **then**, разделенные символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, шестнадцатеричные числа, знак присваивания (**:=**).

16. Входной язык содержит операторы цикла типа **for (**…**;** …**;** …**) do**, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, шестнадцатеричные числа, знак присваивания (**:=**).

17. Входной язык содержит операторы выбора типа **case** … **of** … **end**, разделенные символом **;** (точка с запятой). Операторы выбора содержат идентификаторы, знак двоеточия (**:**), знаки операций +, -, шестнадцатеричные числа, знак присваивания (**:=**).

18. Входной язык содержит операторы цикла типа **while** … **do** …, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, знаки операций +, -, шестнадцатеричные числа, знак присваивания (**:=**).

19. Входной язык содержит арифметические выражения, разделенные символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, символьных констант (один символ в одинарных кавычках), знака присваивания (**:=**), знаков операций +, -, *****, / и круглых скобок.

20. Входной язык содержит логические выражения, разделенные символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, символьных констант ‘T’ («истина») и ‘F’ («ложь), знака присваивания (**:=**), знаков операций **or**, **xor**, **and**, **not** и круглых скобок.

21. Входной язык содержит операторы условия типа **if** … **then** … **else** и **if** … **then**, разделенные символом **;**(точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

22. Входной язык содержит операторы цикла типа **for (**…**;** …**;** …**) do**, разделенные символом **;**(точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

23. Входной язык содержит операторы выбора типа **case** … **of** … **end**, разделенные символом **;** (точка с запятой). Операторы выбора содержат идентификаторы, знак двоеточия (**:**), знаки операций +, -, символьные константы (один символ в одинарных кавычках), знак присваивания (**:=**).

24. Входной язык содержит операторы цикла типа **while** … **do** …, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, знаки операций +, -, символьные константы (один символ в одинарных кавычках), знак присваивания (**:=**).

Примечание:

1. Римскими числами считать последовательности больших латинских букв **X**, **V** и **I**. Если в задании требуется использование римских чисел, то обычные (арабские) числа должны восприниматься как недопустимые лексемы.
2. Шестнадцатеричными числами считать последовательность цифр и символов ‘a’, ‘b’, ‘c’,’d’, ’e’ и ‘f’, начинающуюся с цифры (например: 89, 45ac9, 0abc4). Допускается использовать специальные символы для выделения шестнадцатеричных чисел (например: $0abc4 или 0x0abc4).
3. Задание по лабораторной работе №2 взаимосвязано с заданиями по лабораторным работам №3-8. Результаты, полученные при выполнении лабораторной работы №2, будут использованы в последующих работах.
4. Для уточнения состава входного языка можно посмотреть грамматику, заданную в работе №4 для соответствующего варианта.

ЛАБОРАТОРНАЯ РАБОТА № 3
ПОСТРОЕНИЕ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

Цель работы: ознакомление с назначением и принципами работы лексических анализаторов (сканеров), получение практических навыков построения сканера на примере заданного простейшего входного языка.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Проблемы при построении лексических анализаторов

Лексический анализатор (или сканер) – это часть компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы). Функции лексического анализатора были рассмотрены в предыдущей лабораторной работе №2.

Лексический анализатор имеет дело с таким объектами, как различного рода константы и идентификаторы (к последним относятся и ключевые слова). Язык констант и идентификаторов в большинстве случаев является *регулярным* – то есть, может быть описан с помощью регулярных (праволинейных или леволинейных) грамматик [1, 2, 8, 9]. 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.

Таким образом, построение распознавателя входного языка, лежащего в основе лексического анализатора, является достаточно простой задачей, которая гарантированно имеет решение для всех регулярных языков. Решение этой задачи было подробно рассмотрено в предыдущей лабораторной работе №2.

КА для каждой входной цепочки входного языка дает ответ на вопрос о том, принадлежит или нет цепочка языку, заданному этим автоматом. Однако в общем случае задача лексического анализатора несколько шире, чем просто проверка цепочки символов лексемы на соответствие входному языку. Кроме этого, он должен выполнить следующие действия:

- определить границы лексем, которые в тексте исходной программы явно не указаны;
- выполнить действия для сохранения информации об обнаруженной лексеме (или выдать сообщение об ошибке, если лексема неверна).

Проблема определения границ лексем

Выделение границ лексем является нетривиальной задачей. Ведь в тексте исходной программы лексемы никак не ограничены. В лабораторной работе №2 при построении КА был использован специальный символ (⊥), определяющий границы входной лексемы. Но в реальных программах такой символ в исходном тексте отсутствует – лексический анализатор должен сам определить, где необходимо поставить границы лексем. При этом следует учесть, что граница одной лексемы может быть в то же время началом другой лексемы. Если говорить в терминах лексического анализатора, то определение границ лексем – это выделение тех строк в общем потоке входных символов, для которых надо выполнять распознавание.

Иллюстрацией случая, когда определение границ лексемы вызывает определенные сложности, может служить пример оператора программы на языке FORTRAN: по фрагменту исходного кода `DO 10 I=1` невозможно определить тип оператора языка (а соответственно, и границы лексем). В случае `DO 10 I=1.15` это присвоение вещественной переменной `DO10I` значения константы `1.15` (пробелы в языке FORNTAN игнорируются), а в случае `DO 10 I=1,15` – это цикл с перечислением от `1` до `15` по целочисленной переменной `I` до метки `10`.

Другой иллюстрацией может служить оператор языка C, имеющий вид: `k=i+++++j;`. Существует только одна единственно верная трактовка этого оператора: `k = i++ + ++j;` (если явно пояснить ее с помощью скобок, то данная конструкция имеет вид: `k = (i++) + (++j);`).

Однако найти ее лексический анализатор может, лишь просмотрев весь оператор до конца и перебрав все варианты, причем неверные варианты могут быть обнаружены только на этапе семантического анализа (например, вариант вида `k = (i++)++ + j;` является синтаксически правильным, но семантикой языка С не допускается). Конечно, чтобы эта конструкция была в принципе допустима, входящие в нее операнды `k`, `i` и `j` должны быть описаны и должны допускать выполнение операций языка `++` и `+`.

Как видно из приведённых примеров, для определения границ лексем недостаточно только распознавателя входного языка на основе КА. Поэтому во многих компиляторах лексический и синтаксический анализаторы – это взаимосвязанные части [1, 2, 8, 10, 14].

Взаимодействие лексического и синтаксического анализаторов

В простейшем случае фазы лексического и синтаксического анализа могут выполняться компилятором последовательно. В этом случае лексический анализатор последовательно считывает текст исходной программы, анализирует его, формирует таблицу лексем, заполняет таблицу идентификаторов и заканчивает работу. Его выходные данные – таблица лексем и таблица идентификаторов – поступают на вход следующего этапа компиляции – синтаксического анализатора, который обрабатывает их уже после завершения предыдущей фазы компиляции. Такой метод взаимодействия лексического и синтаксического анализаторов называют ***последовательным***. Схема, изображающая последовательный метод взаимодействия, представлена на рис. 3.

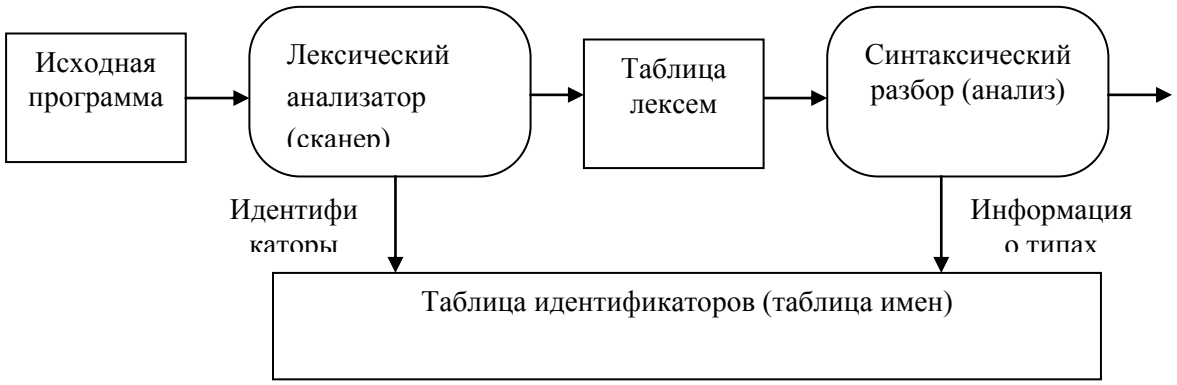


Рис. 3. Последовательное взаимодействие лексического и синтаксического анализаторов

Для некоторых языков программирования информации, имеющейся на этапе лексического анализа, может быть недостаточно для однозначного определения типа и границ очередной лексемы. Иллюстрацией такого случая может служить рассмотренный выше пример оператора программы на языке Фортран, когда по части текста программы `DO 10 I=1...` невозможно определить тип оператора (а соответственно, и границы лексем).

В этом случае в процессе анализа текста исходной программы лексический и синтаксический анализаторы обмениваются данными между собой. Лексический анализ исходного текста в таком варианте выполняется поэтапно так, что синтаксический анализатор, выполнив разбор очередной конструкции языка, обращается к сканеру за следующей лексемой. При этом он может сообщить информацию о том, какую лексему следует ожидать. В процессе разбора может даже происходить «откат назад», чтобы выполнить анализ текста на другой основе, используя другие типы лексем. Такое взаимодействие лексического и синтаксического анализаторов называют ***параллельным***. Схема, изображающая параллельный метод взаимодействия, представлена на рис. 4.

Последовательное взаимодействие лексического и синтаксического анализаторов, представленное на рис. 3, является более эффективным. Оно проще в реализации и требует меньших вычислительных затрат, чем их параллельное взаимодействие, показанное на рис. 4.

Поэтому разработчики компиляторов стремятся использовать в компиляторе последовательное взаимодействие лексического и синтаксического анализаторов.

Но, как было сказано выше, для многих языков программирования на этапе лексического анализа может быть недостаточно информации для однозначного определения типа и границ очередной лексемы. Тогда в современных языках программирования применяется принцип выбора из всех возможных лексем лексемы наибольшей длины: очередной символ из входного потока добавляется в лексему всегда, когда он может быть туда добавлен. Когда символ не может быть добавлен в лексему, то считается, что он является границей лексемы и началом следующей лексемы.

Такой принцип не всегда позволяет правильно определить границы лексем в том случае, когда они не разделены пустыми символами. Например, приведенная выше строка языка C `k = i+++++j;` будет разбита на лексемы следующим образом: `k = i++ ++ + j;` – и это разбиение неверное. Лексический анализатор, разбирая строку из 5 знаков `+` дважды выбрал лексему наибольшей возможной длины – знак операции инкремента (увеличения значения переменной на 1) `++`, хотя это неправильно. Компилятор должен будет выдать пользователю сообщение об ошибке при том, что правильный вариант распознавания этой строки всё же существует.

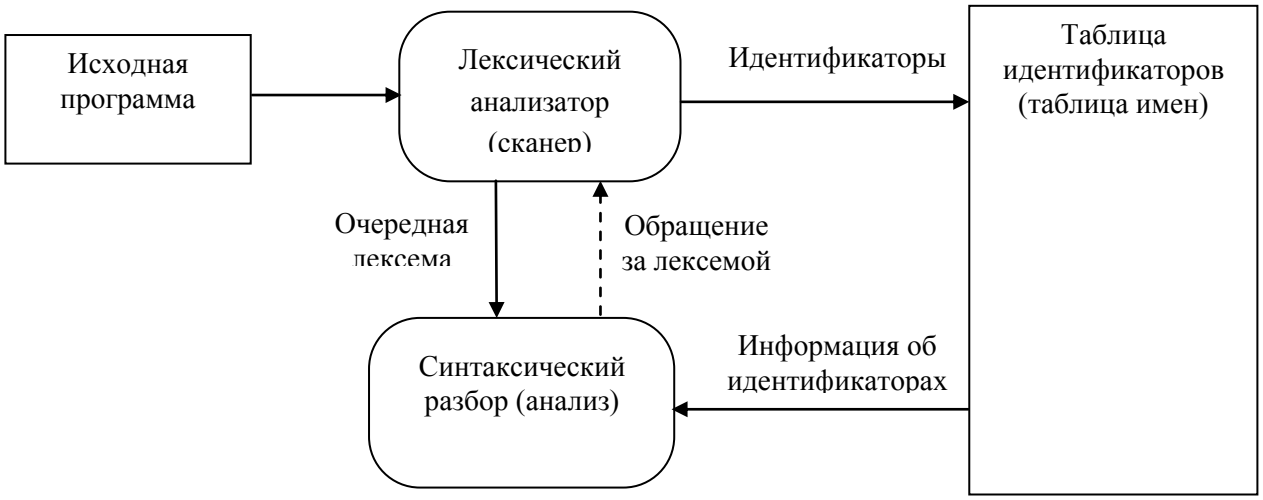


Рис. 4. Параллельное взаимодействие лексического и синтаксического анализаторов

Разработчики компиляторов сознательно идут на то, что отсекают некоторые правильные, но не вполне удобочитаемые варианты исходных программ. Попытки усложнить лексический анализатор неизбежно приведут к необходимости организации его параллельной работы с синтаксическим анализатором. А это снизит эффективность работы всего компилятора.

Не для всех входных языков такой подход возможен. Например, для рассмотренного выше примера с языка FORTRAN невозможно применить указанный метод – разница между оператором цикла и оператором присваивания слишком существенна. В таком случае приходится организовывать параллельную работу лексического и синтаксического анализаторов.

В дальнейшем будем исходить из предположения, что все лексемы могут быть однозначно выделены сканером на этапе лексического разбора – большинство современных широко распространенных языков программирования это допускают.

Результаты лексического анализа

Результатом работы лексического анализатора является заполнение двух таблиц – таблицы идентификаторов и таблицы лексем.

Таблица идентификаторов содержит информацию обо всех лексемах исходной программы, заданных её разработчиком. В неё попадают все идентификаторы – имена переменных, функций, констант и других элементов входного языка. Методы работы с таблицей идентификаторов были подробно рассмотрены в предыдущей лабораторной работе.

Таблица лексем содержит информацию обо всех лексемах исходной программы и порядке их следования. В отличие от таблицы идентификаторов в неё попадают не только идентификаторы, но и лексемы других типов – ключевые слова, константы, знаки операций и разделители. Кроме того, для таблицы лексем важен порядок следования лексем – они обязательно располагаются в ней в том же порядке, что и в исходной программе. Поэтому если в таблице идентификаторов каждая лексема может встречаться только один раз, то в таблице лексем она будет встречаться столько раз, сколько раз она присутствует в тексте исходной программы. Таблица лексем заполняется и обрабатывается последовательно по мере анализа исходной программы, поэтому, в отличие от таблицы идентификаторов, она не требует разработки специальных методов для работы с данными.

Способ представления информации в таблице лексем после выполнения лексического анализа целиком зависит от конструкции компилятора. Но в общем виде ее можно представить как таблицу, которая в каждой строчке должна содержать информацию о виде лексемы, ее типе и, возможно, значении. Обычно такая таблица имеет два столбца: первый – строка лексемы, второй – указатель на информацию о лексеме, может быть включен и третий столбец – тип лексем.

Вот пример фрагмента текста программы на языке Паскаль и соответствующей ему таблицы лексем (таблица 2):

```
...
begin
  for i:=1 to N do
    fg := fg * 0.5
...
```

Таблица 2

Таблица лексем программы

Лексема	Тип лексемы	Значение
begin	Ключевое слово	X_1
for	Ключевое слово	X_2
i	Идентификатор	i : 1
:=	Знак присваивания	
1	Целочисленная константа	1
to	Ключевое слово	X_3
N	Идентификатор	N : 2
do	Ключевое слово	X_4
fg	Идентификатор	fg : 3
:=	Знак присваивания	
fg	Идентификатор	fg : 3
*	Знак арифметической операции	
0.5	Вещественная константа	0.5

Информация, представленная в столбце «Значение» в таблице 2, условно изображает некоторые данные, связанные с каждой лексемой. Конкретный состав информации, хранимой в таблице лексем, и форма её представления определяются разработчиками компилятора.

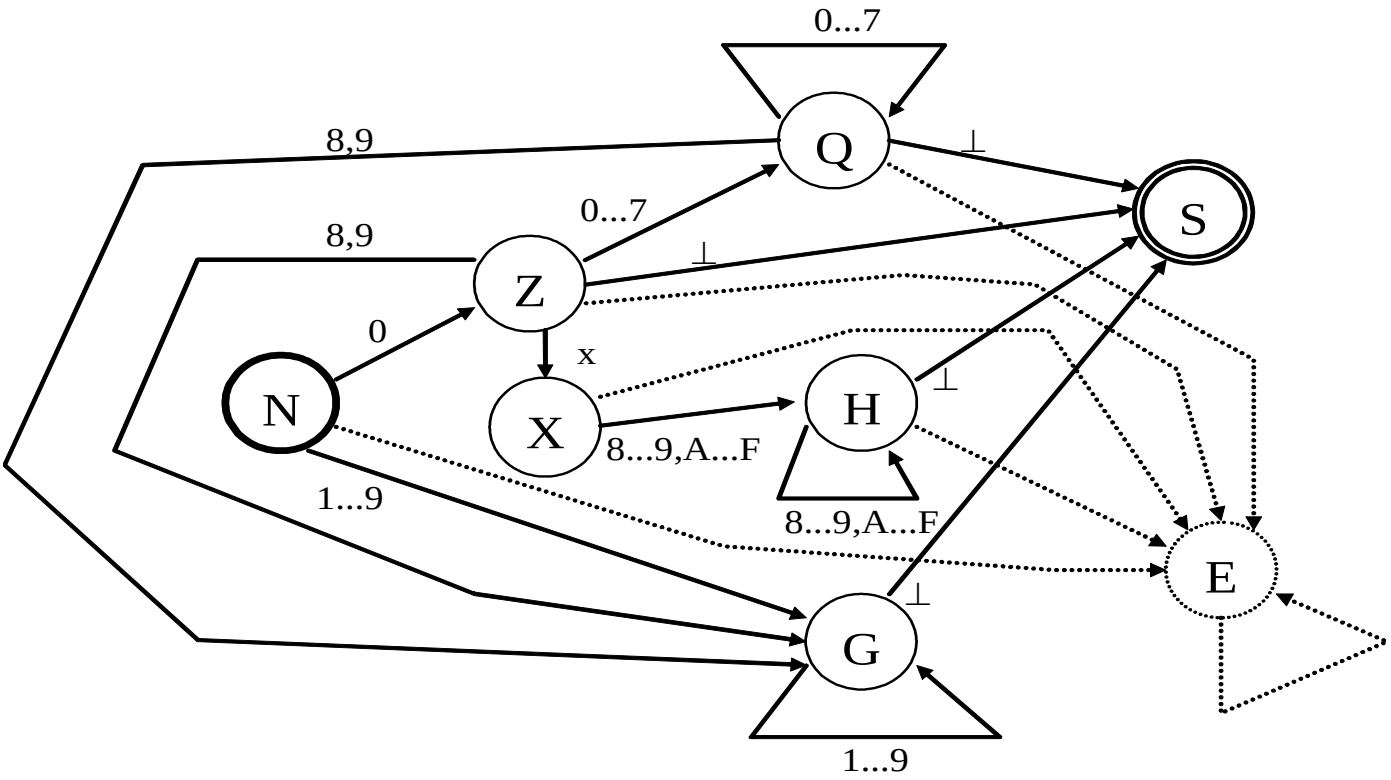


Рис. 5. Граф переходов детерминированного КА, распознающего целые числа языка Си

Для построения лексического анализатора необходимо построить КА, являющийся распознавателем входного языка лексического анализатора. Построение КА описано в лабораторной работе №2. В качестве примера возьмем КА, полученный в лабораторной работе №2 – он представлен на рис. 5.

Для удобства программирования построенный КА желательно привести к детерминированному виду и минимизировать. КА, представленный на рис. 5, является детерминированным (а потому не требует преобразования) и не нуждается в минимизации.

Далее необходимо написать функцию, отражающую функционирование построенного детерминированного КА. Чтобы запрограммировать такую функцию, достаточно иметь переменную, которая бы отображала текущее состояние автомата, а переходы автомата 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 завершению проанализировать состояние автомата. Если это одно из конечных состояний, то функция выполнена успешно, и входная цепочка принимается, если нет – то входная цепочка не принадлежит заданному языку.

Таким способом можно написать программу, моделирующую работу КА, показанного на рис. 5. Ниже приводится текст функции на языке Паскаль, его реализующей. Результат функции истинный (*True*), если входная цепочка символов принадлежит входному языку автомата. Границей цепочки считается символ с кодом 0 (*#0*), в функции он искусственно добавляется в конец цепочки.

В этой программе переменная iState отображает текущее состояние автомата, переменная i является счетчиком символов входной строки. Конечно, рассмотренная программа может быть оптимизирована (например, можно сразу же прекращать разбор по обнаружению ошибки), но в данном примере оптимизация не выполнялась, чтобы можно было четко отследить соответствие между программой и построенным автоматом.

```
type
  TAutoState = ( AUTO_N, AUTO_Z, AUTO_X,
            AUTO_Q, AUTO_H, AUTO_G,
            AUTO_ER, AUTO_S );

function RunAuto (sInput: string): Boolean;
var
  iState : TAutoState;
  i : integer;
begin
  sInput := sInput + #0;
  iState := AUTO_N;
  i := 0;
  repeat
          i := i + 1;
    case iState of
    AUTO_N:
      case sInput[i] of
        '0':   iState := AUTO_Z;
        '1'..'9': iState := AUTO_G;
        else   iState := AUTO_ER;
      end;
    AUTO_Z:
      case sInput[i] of
        '0'..'7': iState := AUTO_Q;
        '8','9':  iState := AUTO_G;
        'x':   iState := AUTO_X;
        #0:    iState := AUTO_S;
        else   iState := AUTO_ER;
      end;
    AUTO_X:
      case sInput[i] of
        '0'..'9': iState := AUTO_H;
        'A'..'F': iState := AUTO_H;
        else   iState := AUTO_ER;
      end;
    AUTO_Q:
      case sInput[i] of
        '0'..'7': iState := AUTO_Q;
        '8','9':  iState := AUTO_G;
        #0:    iState := AUTO_S;
        else   iState := AUTO_ER;
      end;
    AUTO_H:
      case sInput[i] of
        '0'..'9': iState := AUTO_H;
        'A'..'F': iState := AUTO_H;
        #0:    iState := AUTO_S;
        else   iState := AUTO_ER;
      end;
    AUTO_G:
      case sInput[i] of
        '0'..'9': iState := AUTO_G;
        #0:    iState := AUTO_S;
        else   iState := AUTO_ER;
      end;
    AUTO_ER:    iState := AUTO_ER;
    end {case};
  until (sInput[i] = #0);
  RunAuto := (iState = AUTO_S);
end; { RunAuto }
```

Построение КА вручную на основе грамматик, содержащих небольшое количество правил (до нескольких десятков), не вызывает затруднений. Однако для грамматик реальных языков построение распознавателя вручную может представлять проблему из-за значительного количества

состояний КА. Поэтому существуют средства, позволяющие автоматизировать построение лексического анализатора (например, программа LEX [2, 10, 19]).

В общем случае задача сканера несколько шире, чем просто проверка цепочки символов лексемы на соответствие ее входному языку. Сканер должен выполнить те или иные действия по запоминанию распознанной лексемы (занесение ее в таблицу лексем). Набор действий определяется реализацией компилятора. Обычно эти действия выполняются сразу же по обнаружению конца распознаваемой лексемы, поэтому их несложно вставить в соответствующие места рассмотренной выше программы-сканера (в те операторы, где обнаруживается символ *#0*).

В целом для построения лексического анализатора необходимо выполнить следующее:

- проанализировать возможные входные лексемы и выбрать способ определения границ лексем;
- исходя из выбранного способа определения границ лексем, выбрать метод взаимодействия лексического и синтаксического анализаторов;
- построить КА, составляющий основу лексического анализатора;
- привести построенный КА к детерминированному виду;
- минимизировать полученный детерминированный КА;
- создать программную функцию, реализующую работу КА;
- дополнить реализованную функцию структурами данных, используемыми для хранения информации о лексемах (таблицы лексем и таблицы идентификаторов), обеспечить заполнение этих структур в процессе функционирования КА.

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.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя.
2. Построить КА на основе регулярной грамматики (по результатам лабораторной работы №2).
3. Подготовить и защитить отчет.
4. Написать и отладить программу на ЭВМ.
5. Сдать работающую программу преподавателю.

Для выполнения лабораторной работы допускается использовать средства автоматизированного построения лексического анализатора (например, LEX).

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать следующие разделы:

- Краткое изложение цели работы.
- Задание по лабораторной работе с описанием своего варианта и используемого вида комментариев.
- Описание КА или граф КА для распознавания лексем (в соответствии с вариантом задания на основе результатов лабораторной работы №2).
- Пример анализируемого входного текста и результат работы лексического анализатора.

- Текст программы (оформляется только при необходимости по согласованию с преподавателем).
- Выводы по проделанной работе.

Если для выполнения лабораторной работы используются средства автоматизированного построения лексического анализатора, то отчёт должен содержать описание используемого программного обеспечения (либо библиотеки), а также описание всех входных данных, необходимых для построения сканера в соответствии с заданием.

Допускается оформлять единый (совместный) отчёт для лабораторных работ №2 и №3.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение цепочки символов, алфавита, языка. Что такое синтаксис и семантика языка?
2. Какие существуют методы задания языков?
3. Что такое транслятор, компилятор, интерпретатор? В чём их различие между собой?
4. Расскажите об общей структуре компилятора.
5. Что такое лексема? Расскажите, какие типы лексем существуют в языках программирования.
6. Какие функции выполняет лексический анализ в процессе компиляции?
7. Какие преимущества дает компилятору наличие лексического анализатора?
8. Расскажите о проблемах, возникающих при построении лексического анализатора.
9. Как могут быть связаны лексический и синтаксический анализаторы?
10. Что такое грамматика? Дайте определения грамматики.
11. Как выглядит описание грамматики в форме Бэкуса-Наура.
12. Какие классы грамматик существуют?
13. Что такое регулярные грамматики? В чём их особенности?
14. Что такое конечный автомат?
15. Дайте определение детерминированного и недетерминированного КА. Расскажите о возможности преобразования недетерминированного КА в детерминированный.
16. Какие проблемы необходимо решить при построении сканера на основе конечного автомата?

ВАРИАНТЫ ЗАДАНИЙ

Для выполнения лабораторной работы требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений. Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа. Наличие синтаксических ошибок проверять не требуется.

Длину идентификаторов и строковых констант можно считать ограниченной 32 символами. Программа должна допускать ***наличие комментариев неограниченной длины*** во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

Любые лексемы, не предусмотренные вариантом задания, встречающиеся в исходном тексте, должны трактоваться как ошибочные.

Варианты заданий полностью соответствуют вариантам заданий по лабораторной работе №2.

ЛАБОРАТОРНАЯ РАБОТА № 4
ПОСТРОЕНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Цель работы: изучение основных понятий теории грамматик простого и операторного предшествования, ознакомление с алгоритмами синтаксического анализа (разбора) для некоторых

классов КС-грамматик, получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

По иерархии грамматик Хомского выделяют 4 основные группы языков и описывающих их грамматик. При этом для разработки компиляторов наибольший интерес представляют регулярные и контексно-свободные (КС) грамматики и языки. Они используются при описании лексических конструкций и синтаксиса языков программирования. С помощью регулярных грамматик можно описать лексемы языка – идентификаторы, константы, служебные слова и прочие. На основе КС-грамматик строятся более сложные синтаксические конструкции – описания типов и переменных, арифметические и логические выражения, операторы, блоки, модули, и, наконец, полностью вся программа на исходном языке.

Входные цепочки регулярных языков распознаются с помощью конечных автоматов (КА). Они лежат в основе сканеров, выполняющих лексический анализ и выделение слов в тексте программы на входном языке. Результатом работы сканера является преобразование исходной программы в список или таблицу лексем. Дальнейшую ее обработку выполняет другая часть компилятора – синтаксический анализатор. Его работа основана на использовании правил КС-грамматики, описывающих конструкции исходного языка.

Взаимодействие лексического и синтаксического анализатора рассматривалось в предыдущей лабораторной работе №3, здесь же будут рассмотрены алгоритмы, лежащие в основе синтаксического анализа.

Перед синтаксическим анализатором стоят две основные задачи: проверить правильность конструкций программы, которая представляется в виде уже выделенных слов входного языка, и преобразовать ее в вид, удобный для дальнейшей семантической (смысловой) обработки и генерации кода.

Распознавание цепочек КС-языков

Анализаторы для различных типов языков

Как было сказано выше, анализатором для самого простого типа языков – регулярных языков – являются конечные автоматы (КА). Второй по сложности тип языков – КС-языки – допускает распознавание с помощью недетерминированного конечного автомата со стековой (или магазинной) памятью – МП-автомата.

Более сложные контекстно-зависимые языки – последний тип языков, тексты на которых можно распознавать с помощью ЭВМ. Они обрабатываются двусторонними недетерминированными линейно-ограниченными автоматами. Для распознавания цепочек еще более сложных языков без ограничений (языков с фразовой структурой), в общем случае, требуется универсальный вычислитель (машина Тьюринга, машина с неограниченным числом регистров и т.п.), который не может быть реализован с помощью традиционного компьютера на основе архитектуры фон-Неймана. Поэтому языки с фразовой структурой (а к ним относятся все языки естественного общения) не могут быть полноценно проанализированы на ЭВМ.

Контекстно-зависимые языки и языки с фразовой структурой при создании структур языков программирования не используются.

Распознаватели на основе МП-автоматов

МП-автомат [1, 2, 8, 10] в отличие от обычного КА имеет стек (магазин), в который можно помещать специальные "магазинные" символы (обычно это терминальные и нетерминальные символы грамматики языка). Переход МП-автомата из одного состояния в другое зависит не только от входного символа, но и от одного или нескольких верхних символов стека. Таким образом, конфигурация МП-автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки (положением указателя в цепочке) и содержимым стека.

При выполнении перехода между состояниями из стека МП-автомата удаляются верхние символы, соответствующие условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится верхушкой стека. Для МП-автомата допускаются переходы, при которых входной символ игнорируется (и, тем самым, он будет входным символом при следующем переходе). Такие переходы называются λ-переходами.

МП-автомат называется *детерминированным* (ДМП-автоматом) если для каждой возможной его конфигурации возможен только один переход в следующее состояние автомата (учитывая и λ-переходы). В противном случае, если хотя бы в одной конфигурации МП-автомата возможен переход более, чем в одно состояние, МП-автомат называется *недетерминированным*.

Если при окончании входной цепочки МП-автомат находится в одном из заданных конечных состояний, а стек пуст, цепочка считается принятой (после окончания цепочки могут быть сделаны λ-переходы). Иначе входная цепочка символов не принимается.

По КС-грамматике **G(VN**,**VT**,**P**,S), **V**=**VT**∪**VN** можно построить недетерминированный МП-автомат, который допускает (принимает) цепочки языка, заданного этой грамматикой. Он имеет только одно состояние q и следующий набор переходов:

- λ(A)÷(α) - для каждого правила грамматики A→α ∈ **P**, где A∈**VN**, α∈**V***;
- a(a)÷() - для каждого терминала a∈**VT**;

Здесь в круглых скобках показано содержимое верхушки стека. Перед скобками стоит очередной символ входной цепочки или символ λ для λ-перехода, а символ ÷ разделяет состояния автомата до и после выполнения перехода (показывает такт работы автомата). В начале работы такого МП-автомата в стеке лежит целевой символ грамматики S∈**VN**, в конце работы автомата стек должен быть пуст.

Для той же грамматики можно построить и другой МП-автомат, который также содержит одно состояние и имеет следующие переходы:

- a()÷(a) - для каждого терминала a∈**VT**;
- λ(α)÷(A) - для каждого правила грамматики A→α ∈ **P**, где A∈**VN**, α∈**V***;

Такой МП-автомат называют *расширенным МП-автоматом*, поскольку при выполнении перехода между состояниями он анализирует не один символ на верхушке стека, а цепочку символов. Для расширенного МП-автомата в начале разбора стек автомата пуст, а в конце разбора допустимой цепочки в стеке должен остаться целевой символ грамматики S∈**VN.**

Как было показано выше, для любой КС-грамматики можно построить МП-автомат, распознающий цепочки символов языка, заданного этой грамматикой. Также для любого МП-автомата всегда можно построить КС-грамматику, задающую язык цепочек, распознаваемых этим автоматом. Таким образом, тип КС-языков и тип языков, допускаемых МП-автоматами, являются эквивалентными.

Не каждый КС-язык допускает разбор с помощью детерминированного МП-автомата. Недетерминированные МП-автоматы могут распознавать более широкий класс языков, чем детерминированные. Поэтому не всегда для произвольного недетерминированного МП-автомата можно построить эквивалентный ему детерминированный МП-автомат – в этом их отличие от обычных КА.

Интерес для реализации компиляторов представляют именно детерминированные КС-языки (ДКС-языки) – собственное подмножество КС-языков, допускающее распознавание с помощью детерминированных МП-автоматов – поскольку доказано, что для любого ДКС-языка существует однозначная КС-грамматика, задающая этот язык. Однозначность грамматики – это безусловное требование для любого языка программирования. Поэтому при создании компиляторов используются именно детерминированные МП-автоматы.

Сложность синтаксического разбора. Разбор сверху вниз и снизу вверх

Программирование работы недетерминированного МП-автомата – это сложная задача. Разработан алгоритм, позволяющий для произвольной КС-грамматики определить, принадлежит ли ей заданная входная цепочка (алгоритм Кока-Янгера-Касами) [1, 2, 3, 8, 14]. Доказано, что время работы этого алгоритма пропорционально n^3, где n - длина входной цепочки. Для однозначной КС-грамматики при использовании другого алгоритма (алгоритм Эрли [1, 2, 3, 8, 14]) это время пропорционально n^2. Подобная зависимость делает эти алгоритмы требовательными к вычислительным ресурсам, а потому малопригодными для практических целей.

Два МП-автомата, построенных выше для произвольной КС-грамматики, определяют два метода разбора КС-языков: сверху вниз и снизу вверх. Сами по себе эти МП-автоматы не представляют практического интереса, так как для реальных языков программирования они будут недетерминированными, а потому неэффективными и сложными в реализации. Однако все используемые на практике синтаксические анализаторы являются модификациями описанных выше МП-автоматов, представляющими собой ДМП-автоматы, применимые для анализа различных классов КС-языков. Каждый такой ДМП-автомат имеет ограниченную применимость только для определенного класса в составе КС-языков, но при этом он является детерминированным и эффективным в реализации. Множество разработанных таким образом модификаций ДМП-автоматов покрывает существенную долю всех возможных ДКС-языков. Поэтому на практике для каждого языка программирования удается найти подходящий для него ДМП-автомат, распознающий цепочки этого языка (а чаще всего – сразу несколько подходящих ДМП-автоматов).

В первом случае, при синтаксическом разборе сверху вниз выполняется просмотр входной цепочки слева направо, а в результате получаются левосторонние выводы [1, 2, 3, 8, 10]. МП-автомат, лежащий в основе такого разбора, называется *МП-автоматом с подбором альтернатив*. Он является недетерминированным, так как нет однозначного выбора, какое правило применить для раскрытия самого левого нетерминального символа (подборе альтернативы). ДМП-автоматы, построенные на его основе, используют различные модификации, позволяющие сделать однозначный выбор правила при подборе альтернативы. Примерами таких автоматов являются разбор по методу рекурсивного спуска и распознаватель на основе LL-грамматик.

Во втором случае при синтаксическом разборе снизу вверх выполняется просмотр входной цепочки слева направо, а в результате получаются правосторонние выводы [1, 2, 3, 8, 10]. МП-автомат, лежащий в основе такого разбора, называется *МП-автоматом типа сдвиг-свертка*. Это название определено тем, сто такой разбор удобно формализовать в терминах "сдвиг" (перенос символа из входной цепочки в магазин) и "свертка" (применение к вершине магазина какого-либо правила грамматики). Для построения ДМП-автоматов на основе МП-автомата типа сдвиг-свертка необходимо на каждом шаге работы автомата однозначно определить, какое действие – сдвиг или свертка – должно выполняться, а если выполняется свертка, то для нее должно однозначно выбираться правило. Примерами таких автоматов являются распознаватели на основе LR-грамматик и грамматик предшествования.

Грамматики предшествования

Синтаксический разбор на основе грамматик предшествования

Как было сказано выше, на практике не требуется анализ цепочки произвольного КС-языка – большинство конструкций языков программирования может быть отнесено в один из классов ДКС-языков, для которых разработаны алгоритмы разбора, линейно зависящие от длины входной цепочки.

Одним из таких классов является класс грамматик предшествования. Особенность грамматик предшествования заключается в том, что для каждой упорядоченной пары символов в грамматике устанавливается отношение, называемое отношением предшествования. Обычно

используются три вида отношений предшествования, которые обозначаются следующим образом: $<\cdot$ («предшествует»), $=\cdot$ («составляет основу») и $\cdot>$ («следует»). Отношение предшествования единственно для каждой упорядоченной пары символов. При этом между какими-либо двумя символами может и не быть отношения предшествования – это значит, что они не могут находиться рядом ни в одном элементе разбора синтаксически правильной цепочки. Отношения предшествования зависят от порядка, в котором стоят символы, и в этом смысле их нельзя путать со знаками математических операций – например, если $A \cdot> B$, то не обязательно, что $B <\cdot A$ (поэтому знаки предшествования обычно помечают специальной точкой, чтобы не путать их со знаками математических операций сравнения).

Метод предшествования основан на том факте, что отношения предшествования между двумя соседними символами распознаваемой строки соответствуют трем следующим вариантам:

- $S_i <\cdot S_{i+1}$, если символ S_{i+1} - крайний левый символ некоторой основы;
- $S_i \cdot> S_{i+1}$, если символ S_i - крайний правый символ некоторой основы;
- $S_i =\cdot S_{i+1}$, если символы S_i и S_{i+1} принадлежат одной основе.

Исходя из этих соотношений, выполняется разбор входной строки на основе грамматики предшествования.

Распознаватель на основе грамматики предшествования в общем виде работает следующим образом: в процессе разбора расширенный МП-автомат сравнивает текущий символ входной цепочки с одним из символов, находящихся на верхушке стека автомата. При этом текущий входной символ считается левым символом в отношении предшествования, а символ, находящийся на верхушке стека автомата – правым символом в отношении (порядок символов в отношениях предшествования имеет принципиальное значение!). В процессе сравнения проверяется, какое из возможных отношений предшествования существует между этими двумя символами. В зависимости от найденного отношения выполняется либо сдвиг, либо свертка. При отсутствии отношения предшествования между символами распознаватель сигнализирует об ошибке.

Задача построения грамматики предшествования заключается в том, чтобы иметь возможность непротиворечивым образом определить отношения предшествования между символами грамматики. Если это возможно, то грамматика может быть отнесена к одному из видов грамматик предшествования.

Существует несколько видов грамматик предшествования. Они различаются по тому, какие отношения предшествования в них определены и между какими типами символов (терминальными или нетерминальными) могут быть установлены эти отношения. Кроме того, возможны незначительные модификации функционирования алгоритма «сдвиг-свертка» в распознавателях для таких грамматик (в основном на этапе выбора правила для выполнения свертки, когда возможны неоднозначности) [2, 3, 6].

Выделяют следующие типы грамматик предшествования:

- простого предшествования;
- расширенного предшествования;
- слабого предшествования;
- смешанной стратегии предшествования;
- операторного предшествования.

Далее будут рассмотрены ограничения на структуру правил и алгоритмы разбора для грамматик операторного предшествования.

Грамматики операторного предшествования

Грамматикой операторного предшествования называется приведенная КС-грамматика без λ-правил (е-правил), в которой все правые части продукций различны и не содержат смежных нетерминальных символов. В грамматике операторного предшествования различные порождающие правила должны иметь разные правые части. Для грамматики операторного

предшествования отношения предшествования задаются только на множестве терминальных символов.

Отношения предшествования для грамматики операторного предшествования **G**(**VN**,**VT**,**P**,S) задаются следующим образом:

- $a \doteq b$, если и только если существует правило $U \to \alpha ab\beta \in \mathbf{P}$ или правило $U \to \alpha aCb\beta \in \mathbf{P}$, где $a,b \in \mathbf{VT}$, $U,C \in \mathbf{VN}$, $\alpha,\beta \in \mathbf{V}^*$;
- $a \lessdot b$, если и только если существует правило $U \to \alpha aC\beta \in \mathbf{P}$ и вывод $C \Rightarrow^* b\omega$ или вывод $C \Rightarrow^* Db\omega$, где $a,b \in \mathbf{VT}$, $U,C,D \in \mathbf{VN}$, $\alpha,\beta,\omega \in \mathbf{V}^*$;
- $a \gtrdot b$, если и только если существует правило $U \to \alpha Cb\beta \in \mathbf{P}$ и вывод $C \Rightarrow^* \omega a$ или вывод $C \Rightarrow^* \omega aD$, где $a,b \in \mathbf{VT}$, $U,C,D \in \mathbf{VN}$, $\alpha,\beta,\omega \in \mathbf{V}^*$.

Для удобства работы на основании отношений предшествования строят матрицу предшествования грамматики. Строки матрицы предшествования помечаются первыми (левыми) символами, а столбцы – вторыми (правыми) символами отношений предшествования. В клетки матрицы на пересечении соответствующих столбца и строки помещаются знаки отношений. При этом пустые клетки матрицы говорят о том, что между данными символами нет ни одного отношения предшествования.

Матрица предшествования для грамматики операторного предшествования содержит только терминальные символы. В этом проявляется преимущество данного вида грамматик перед другими видами грамматик предшествования, в которых для определения отношений предшествования используются как терминальные, так и нетерминальные символы – при использовании только терминальных символов существенно сокращается объем матрицы предшествования. Однако это налагает дополнительные ограничения на правила грамматики – в правилах грамматик операторного предшествования не может быть двух смежных нетерминальных символов. Это, в свою очередь, сокращает множество грамматик, языки которых могут быть проанализированы с помощью распознавателя на основе операторного предшествования без предварительного преобразования исходной грамматики.

Матрицу предшествования грамматики можно построить, опираясь непосредственно на определения отношений предшествования, но на практике ее построение удобнее выполнить следующим образом:

- построить два дополнительных множества – крайних левых и крайних правых символов относительно нетерминальных символов грамматики;
- на основе множеств крайних левых и крайних правых символов относительно нетерминальных символов грамматики построить множества крайних левых и крайних правых терминальных символов;
- с помощью множеств крайних левых и крайних правых терминальных символов относительно нетерминальных символов грамматики заполнить матрицу предшествования.

Такой подход удобен на практике, так как не требует построения выводов. Множества крайних левых и крайних правых символов относительно нетерминальных символов грамматики определяются следующим образом:

- $\mathbf{L}(U) = \{T \mid \exists\ U \Rightarrow^* T\omega\}$, $U,T \in \mathbf{V}$, $\omega \in \mathbf{V}^*$ – множество крайних левых символов относительно нетерминального символа U;
- $\mathbf{R}(U) = \{T \mid \exists\ U \Rightarrow^* \omega T\}$, $U,T \in \mathbf{V}$, $\omega \in \mathbf{V}^*$ – множество крайних правых символов относительно нетерминального символа U.

В приведённых выше определениях цепочка ω может быть любой произвольной цепочкой символов, в том числе и пустой цепочкой.

Множества **L**(U) и **R**(U) могут быть построены для каждого нетерминального символа грамматики $U \in \mathbf{VN}$ по очень простому алгоритму:

Шаг 1. Для каждого нетерминального символа U ищем все правила, содержащие U в левой части. Во множество **L**(U) включаем самый левый символ из правой части правил, а во множество **R**(U) - самый крайний символ правой части. Переходи к шагу 2.

Шаг 2. Для каждого нетерминального символа U: если множество **L**(U) содержит нетерминальные символы грамматики U',U'',..., то его надо дополнить символами, входящими в соответствующие множества **L**(U'), **L**(U''), ... и не входящими в **L**(U). Ту же операцию надо выполнить для **R**(U).

Шаг 3. Если на предыдущем шаге хотя бы одно множество **L**(U) или **R**(U) для некоторого символа грамматики изменилось, то надо вернуться к шагу 2, иначе построение закончено.

Далее необходимо построить множества крайних левых и крайних правых терминальных символов относительно нетерминального символа U - $\mathbf{L}^t(U)$ или $\mathbf{R}^t(U)$. Эти множества могут быть определены следующим образом:

- $\mathbf{L}^t(U) = \{d \mid \exists\ U \Rightarrow^* d\omega$ или $\exists\ U \Rightarrow^* Cd\omega\ \}$, где $d \in \mathbf{VT}$, $U,C \in \mathbf{VN}$, $\omega \in \mathbf{V}^*$;
- $\mathbf{R}^t(U) = \{d \mid \exists\ U \Rightarrow^* \omega d$ или $\exists\ U \Rightarrow^* \omega dC\ \}$, где $d \in \mathbf{VT}$, $U,C \in \mathbf{VN}$, $\omega \in \mathbf{V}^*$.

Тогда определения отношений операторного предшествования будут выглядеть так:

- $a =\cdot\ b$, если $\exists$ правило $U \to \alpha ab\beta \in \mathbf{P}$ или правило $U \to \alpha aCb\gamma \in \mathbf{P}$, где $a,b \in \mathbf{VT}$, $U,C \in \mathbf{VN}$, $\alpha,\beta \in \mathbf{V}^*$;
- $a <\cdot\ b$, если $\exists$ правило $U \to \alpha aC\beta \in \mathbf{P}$ и $b \in \mathbf{L}^t(C)$, где $a,b \in \mathbf{VT}$, $U,C \in \mathbf{VN}$, $\alpha,\beta \in \mathbf{V}^*$;
- $a \cdot> b$, если $\exists$ правило $U \to \alpha Cb\beta \in \mathbf{P}$ и $a \in \mathbf{R}^t(C)$, где $a,b \in \mathbf{VT}$, $U,C \in \mathbf{VN}$, $\alpha,\beta \in \mathbf{V}^*$.

В данных определениях цепочки символов α,β,ω могут быть произвольными, в том числе и пустыми цепочками.

Для нахождения множеств $\mathbf{L}^t(U)$ и $\mathbf{R}^t(U)$ используется следующий алгоритм:

Шаг 1. Для каждого нетерминального символа грамматики U строятся множества **L**(U) и **R**(U). Таким образом, множества **L**(U) и **R**(U) являются исходными данными для заполнения множеств $\mathbf{L}^t(U)$ и $\mathbf{R}^t(U)$.

Шаг 2. Для каждого нетерминального символа грамматики U ищутся правила вида $U \to d\omega$ и $U \to Cd\omega$, где $d \in \mathbf{VT}$, $C \in \mathbf{VN}$, $\omega \in \mathbf{V}^*$; терминальные символы d включаются во множество $\mathbf{L}^t(U)$. Аналогично для множества $\mathbf{R}^t(U)$ ищутся правила вида $U \to \omega d$ и $U \to \omega dC$.

Шаг 3. Просматривается множество **L**(U), в которое входят символы U',U'',... Множество $\mathbf{L}^t(U)$ дополняется символами, входящими в $\mathbf{L}^t(U')$, $\mathbf{L}^t(U'')$, ..., не входящими в $\mathbf{L}^t(U)$. Аналогичная операция выполняется и для множества $\mathbf{R}^t(U)$ на основе множества **R**(U).

После построения множеств $\mathbf{L}^t(U)$ и $\mathbf{R}^t(U)$ на основе правил грамматики и определения отношений предшествования, данных выше, заполняется матрица операторного предшествования. Для удобства работы матрицу предшествования дополняют символами $\perp_н$ и $\perp_к$, которые обозначают начало и конец входной цепочки. Для них определены следующие отношения предшествования:

$\perp_н <\cdot\ a$, $\forall\ a \in \mathbf{VT}$, если $\exists\ S \Rightarrow^* a\omega$ или $\exists\ S \Rightarrow^* Ca\omega$, где $S,C \in \mathbf{VN}$, $\omega \in \mathbf{V}^*$ или если $a \in \mathbf{L}^t(S)$;

$\perp_к \cdot> a$, $\forall\ a \in \mathbf{VT}$, если $\exists\ S \Rightarrow^* \omega a$ или $\exists\ S \Rightarrow^* \omega aC$, где $S,C \in \mathbf{VN}$, $\omega \in \mathbf{V}^*$ или если $a \in \mathbf{R}^t(S)$.

Таким образом, чтобы построить матрицу предшествования для грамматики операторного предшествования необходимо выполнить следующие действия:

1. Построить множества крайних левых и крайних правых символов грамматики – **L**(U) и **R**(U).
2. На основе построенных множеств **L**(U) и **R**(U) построить множества крайних левых и крайних правых терминальных символов грамматики – $\mathbf{L}^t(U)$ и $\mathbf{R}^t(U)$.
3. Используя построенные множества $\mathbf{L}^t(U)$ и $\mathbf{R}^t(U)$ на основе правил исходной грамматики заполнить матрицу операторного предшествования.

При заполнении матрицы предшествования может оказаться, что в одну и ту же клетку матрицы необходимо записать более одного отношения предшествования. Это означает, что исходная грамматика не является грамматикой операторного предшествования. В таком случае для анализа цепочек языка, заданного исходной грамматикой, не может использоваться распознаватель на основе операторного предшествования, либо перед построением распознавателя исходную грамматику необходимо преобразовать.

Алгоритм “сдвиг-свертка” для грамматик операторного предшествования

Алгоритм “сдвиг-свертка” для грамматики операторного предшествования. Данный алгоритм выполняется расширенным ДМП-автоматом типа «сдвиг-свертка» с одним состоянием. Отношения предшествования служат для того, чтобы определить, какое действие – сдвиг или свертка – должно выполняться на каждом шаге и однозначно выбрать правило при свертке. При определении отношений предшествования не принимаются во внимание нетерминальные символы, находящиеся на верхушке стека, и ищется ближайший к верхушке стека терминальный символ. Однако после выполнения сравнения и определения границ основы при поиске правила в грамматике нетерминальные символы следует, безусловно, принимать во внимание.

В начальном состоянии ДМП-автомата считывающая головка обозревает первый символ, в конец цепочки помещен символ $\perp_к$. Алгоритм разбора можно описать следующим образом:

Шаг 1. Поместить в верхушку стека символ начала цепочки – $\perp_н$.

Шаг 2. Сравнить символ, находящийся на вершине стека, игнорируя все нетерминальные символы, с текущим символом входной цепочки.

Шаг 3. Если имеет место отношение $<\cdot$ или $=\cdot$, то произвести перенос (поместить текущий входной символ в стек и сдвинуть входную цепочку на один символ) и вернуться к шагу 2.

Шаг 4. Если имеет место отношение $\cdot>$, то произвести свертку, то есть найти на вершине стека (игнорируя нетерминальные символы) все символы, связанные отношением $=\cdot$ (составляет основу), и заменить их левой частью соответствующего правила грамматики (при выборе правила нетерминальные символы должны учитываться). Поскольку грамматика операторного предшествования не допускает различных правил с одинаковой правой частью, выбор правила на этом шаге однозначен. Если подходящее правило найти не удается, сигнализировать об ошибке.

Шаг 5. Если отношение предшествования между символами не определено, прервать выполнение и сигнализировать об ошибке.

Шаг 6. Проверить условие завершения разбора (на верхушке стека находится целевой символ грамматики S и символ $\perp_н$, текущий входной символ – символ конца цепочки $\perp_к$). Если разбор не закончен, то вернуться к шагу 2, иначе выполнение алгоритма завершено.

Ошибка в процессе выполнения алгоритма возникает, когда невозможно выполнить очередной шаг – например, если не установлено отношение предшествования между двумя сравниваемыми символами (на шаге 2) или если не удается найти нужное правило в грамматике (на шаге 4). Тогда выполнение алгоритма немедленно прерывается.

Разбор считается законченным (алгоритм завершается), если считывающая головка автомата обозревает символ конца входной цепочки – $\perp_н$, и при этом больше не может быть выполнена свертка. Решение о принятии цепочки зависит от содержимого стека. Автомат принимает цепочку, если в результате завершения алгоритма он находится в конечном состоянии, когда в стеке находятся только целевой символ грамматики S и символ $\perp_н$, иначе входная цепочка не принимается (выдаётся сообщение об ошибке). Выполнение алгоритма может быть прервано, если на одном из его шагов возникнет ошибка. Тогда входная цепочка также не принимается.

Пример построения распознавателя для грамматики операторного предшествования.

Рассмотрим в качестве примера грамматику **G**({S,B,T,J},{**-**,**&**,^,**(**,**)**,**p**},**P**,S) (терминальные символы выделены жирным шрифтом):

P: S $\rightarrow$ **-**B (правило 1)
B $\rightarrow$ T | B**&**T (правила 2 и 3)
T $\rightarrow$ J | T^J (правила 4 и 5)
J $\rightarrow$ **(**B**)** | **p** (правила 6 и 7)

Видно, что эта грамматика удовлетворяет необходимым условиям, предъявляемым к грамматикам операторного предшествования.

Построим множества крайних левых и крайних правых символов **L**(U) и **R**(U) относительно всех нетерминальных символов грамматики. Результат построения приведен в таблице 3.

Таблица 3.

Множества крайних правых и крайних левых символов грамматики (по шагам построения)

Символ (U)	Шаг 1 (начало построения)		Последний шаг (результат)	
	L(U)	**R**(U)	**L**(U)	**R**(U)
J	**(p**	**) p**	**(p**	**) p**
T	J T	J	J T **(p**	J **) p**
B	T B	T	T B J **(p**	T J **) p**
S	-	B	-	B T J **) p**

На основе полученных множеств **L**(U) и **R**(U) построим множества крайних левых и крайних правых терминальных символов $\mathbf{L}^t$(U) и $\mathbf{R}^t$(U) относительно всех нетерминальных символов грамматики. Результат (второй и третий шаги построения) приведен в таблице 4.

Таблица 4.

Множества крайних правых и левых терминальных символов грамматики (по шагам построения)

Символ (U)	Шаг 1 (начало построения)		Последний шаг (результат)	
	$\mathbf{L}^t$(U)	$\mathbf{R}^t$(U)	$\mathbf{L}^t$(U)	$\mathbf{R}^t$(U)
J	**(p**	**) p**	**(p**	**) p**
T	^	^	^ **(p**	^ **) p**
B	**&**	**&**	**& ^ (p**	**& ^) p**
S	-	-	-	**- & ^) p**

На основе множеств $\mathbf{L}^t$(U) и $\mathbf{R}^t$(U) и правил грамматики **G** построим матрицу предшествования исходной грамматики (таблица 5).

Таблица 5.

Матрица предшествования грамматики

Символы	-	**&**	^	**(**	**)**	**p**	$\perp_к$
-		<·	<·	<·		<·	·>
&		·>	<·	<·	·>	<·	·>
^		·>	·>	<·	·>	<·	·>
(		<·	<·	<·	=·	<·	
)		·>	·>		·>		·>
p		·>	·>		·>		·>
$\perp_н$	<·						

Посмотрим, как заполняется матрица предшествования в таблице 5 на примере символа **&**. В правиле грамматики B → B**&**T (правило 3) этот символ стоит слева от нетерминального символа T. В множество $\mathbf{L}^t$(T) входят символы: **^ (p**. Ставим знак <· в клетках матрицы, соответствующих этим символам, в строке для символа **&**. В то же время в этом же правиле символ **&** стоит справа от нетерминального символа B. В множество $\mathbf{R}^t$(B) входят символы: **& ^) p**. Ставим знак ·> в клетках матрицы, соответствующим этим символам, в столбце для символа **&**. Больше символ **&** ни в каком правиле не встречается, значит, заполнение матрицы для него закончено.

Выполнив аналогичные действия для всех терминальных символов исходной грамматики, заполним всю матрицу предшествования отношениями <· и ·>. Остается добавить в неё отношения =· («составляет основу»). Для этого надо найти в исходной грамматике все правила, где два терминальных символа стоят рядом или между ними есть один нетерминальный символ. В исходной грамматике такое правило только одно: J → **(**B**)** (правило 6). В нём терминальные символы **(** и **)** стоят рядом и между ними только один нетерминальный символ (символ B). При

этом символ **(** стоит слева, а символ **)** – справа (порядок символов принципиально важен!). Тогда на пересечении строки для символа **(** и столбца для символа **)** ставим знак отношения =·.

Теперь матрица операторного предшествования, представленная в таблице 5, полностью заполнена. В ней есть пустые клетки – это не является ошибкой. Важно, что ни в одну клетку матрицы не попало более одного знака отношений предшествования – это означает, что исходная грамматика является грамматикой операторного предшествования и не требует дополнительных преобразований для анализа входных цепочек на основе данного метода.

Алгоритм разбора цепочек грамматики операторного предшествования игнорирует нетерминальные символы. Поэтому имеет смысл преобразовать исходную грамматику таким образом, чтобы оставить в ней только один нетерминальный символ. Тогда получим следующий вид правил:

P: E → -E (правило 1)
E → E | E**&**E (правила 2 и 3)
E → E | E^E (правила 4 и 5)
E → **(**E**)** | **p** (правила 6 и 7)

Это преобразование не ведет к созданию эквивалентной грамматики и выполняется только после построения всех множеств и матрицы предшествования. Само преобразование выполняется только с целью более эффективного выполнения алгоритма разбора, в который уже заложены необходимые данные о порядке применения правил при создании матрицы предшествования. Полученная в результате такого преобразования грамматика называется остовной грамматикой.

Рассмотрим работу алгоритма распознавания на примерах. Последовательность разбора будем записывать в виде последовательности конфигураций ДМП-автомата из трех составляющих: не просмотренная автоматом часть входной цепочки, содержимое стека и последовательность правил грамматики. Так как используемый ДМП-автомат имеет только одно состояние, то для определения его конфигурации достаточно двух составляющих – положения считывающей головки во входной цепочке и содержимого стека. Но последовательность номеров правил нужно запоминать, поскольку она несет полезную информацию, которая должна быть использована компилятором на последующих этапах для семантической обработки и генерации кода объектной программы. Кроме того, последовательность примененных правил делает пример более наглядным.

Будем обозначать такт автомата символом ÷. Введем также дополнительное обозначение $\div_п$, если на данном такте выполнялся перенос, и $\div_с$, если выполнялась свертка.

Последовательности разбора цепочек входных символов будут, таким образом, иметь следующий вид:

Пример 1. Входная цепочка **-p&p^(p)**.

{**-p&p^(p)**$\perp_к$; $\perp_н$; ∅} ÷$_п$ {**p&p^(p)**$\perp_к$; $\perp_н$**-**; ∅} ÷$_п$ {**&p^(p)**$\perp_к$; $\perp_н$**-p**; ∅} ÷$_с$ {**&p^(p)**$\perp_к$; $\perp_н$-E; 7} ÷$_п$
{**p^(p)**$\perp_к$; $\perp_н$-E**&**; 7} ÷$_п$ {**^(p)**$\perp_к$; $\perp_н$-E**&p**; 7} ÷$_с$ {**^(p)**$\perp_к$; $\perp_н$-E**&**E; 7,7} ÷$_п$ {**(p)**$\perp_к$; $\perp_н$-E**&**E^; 7,7} ÷$_п$
{**p)**$\perp_к$; $\perp_н$-E**&**E^**(**; 7,7} ÷$_п$ {**)**$\perp_к$; $\perp_н$-E**&**E^**(p**; 7,7} ÷$_с$ {**)**$\perp_к$; $\perp_н$-E**&**E^**(**E; 7,7,7} ÷$_п$
{$\perp_к$; $\perp_н$-E**&**E^**(**E**)**; 7,7,7} ÷$_с$ {$\perp_к$; $\perp_н$-E**&**E^E; 7,7,7,6} ÷$_с$ {$\perp_к$; $\perp_н$-E**&**E; 7,7,7,6,5} ÷$_п$
{$\perp_к$; $\perp_н$-E; 7,7,7,6,5,3} ÷$_с$ {$\perp_к$; $\perp_н$E; 7,7,7,6,5,3,1} – разбор закончен, цепочка принята.

Пример 2. Входная цепочка **-p^p(p)**.

{**-p^p(p)**$\perp_к$; $\perp_н$; ∅} ÷$_п$ {**p^p(p)**$\perp_к$; $\perp_н$**-**; ∅} ÷$_п$ {**^p(p)**$\perp_к$; $\perp_н$**-p**; ∅} ÷$_с$ {**^p(p)**$\perp_к$; $\perp_н$-E; 7} ÷$_п$
{**p(p)**$\perp_к$; $\perp_н$-E^; 7} ÷$_п$ {**(p)**$\perp_к$; $\perp_н$-E^**p**; 7} - ошибка ! (нет отношения для пары символов {**p**,**(**})

Пример 3. Входная цепочка **-p^p&p**.

{**-p^p&p**$\perp_к$; $\perp_н$; ∅} ÷$_п$ {**p^p&p**$\perp_к$; $\perp_н$**-**; ∅} ÷$_п$ {**^p&p**$\perp_к$; $\perp_н$**-p**; ∅} ÷$_с$ {**^p&p**$\perp_к$; $\perp_н$-E; 7} ÷$_п$
{**p&p**$\perp_к$; $\perp_н$-E^; 7} ÷$_п$ {**&p**$\perp_к$; $\perp_н$-E^**p**; 7} ÷$_с$ {**&p**$\perp_к$; $\perp_н$-E^E; 7,7} ÷$_с$ {**&p**$\perp_к$; $\perp_н$-E; 7,7,5} ÷$_п$
{**p**$\perp_к$; $\perp_н$-E**&**; 7,7,5} ÷$_п$ {$\perp_к$; $\perp_н$-E**&p**; 7,7,5} ÷$_с$ {$\perp_к$; $\perp_н$-E**&**E; 7,7,5,7} ÷$_с$ {$\perp_к$; $\perp_н$-E; 7,7,5,7,3} ÷$_с$
{$\perp_к$; $\perp_н$E; 7,7,5,7,3,1} – разбор закончен, цепочка принята.

Пример 4. Входная цепочка **-p&p^p**.

{**-p&p^p**$\perp_к$; $\perp_н$; ∅} ÷$_п$ {**p&p^p**$\perp_к$; $\perp_н$**-**; ∅} ÷$_п$ {**&p^p**$\perp_к$; $\perp_н$**-p**; ∅} ÷$_с$ {**&p^p**$\perp_к$; $\perp_н$-E; 7} ÷$_п$ {**p^p**$\perp_к$; $\perp_н$-E**&**; 7} ÷$_п$ {**^p**$\perp_к$; $\perp_н$-E**&p**; 7} ÷$_с$ {**^p**$\perp_к$; $\perp_н$-E**&**E; 7,7} ÷$_п$ {**p**$\perp_к$; $\perp_н$-E**&**E^; 7,7} ÷$_п$ {$\perp_к$; $\perp_н$-E**&**E^**p**; 7,7} ÷$_с$ {$\perp_к$; $\perp_н$-E**&**E^E; 7,7,7} ÷$_с$ {$\perp_к$; $\perp_н$-E**&**E; 7,7,7,5} ÷$_п$ {$\perp_к$; $\perp_н$-E; 7,7,7,5,3} ÷$_с$ {$\perp_к$; $\perp_н$E; 7,7,7,5,3,1} – разбор закончен, цепочка принята.

Два последних примера наглядно демонстрируют, что приоритет операций, установленный в грамматике, влияет на последовательность разбора и последовательность применения правил.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя.
2. Выполнить предварительные преобразования, необходимые для построения матрицы предшествования.
3. Построить матрицу предшествования для заданной грамматики.
4. Выполнить разбор простейшего примера вручную по правилам заданной грамматики.
5. Подготовить и защитить отчет.

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать следующие разделы:

- Краткое изложение цели работы.
- Задание по лабораторной работе с описанием своего варианта**.**
- Запись заданной грамматики входного языка в форме Бэкуса-Наура.
- Множества крайних правых и крайних левых символов с указанием шагов построения.
- Множества крайних правых и крайних левых терминальных символов.
- Заполненную матрицу предшествования для грамматики.
- Пример выполнения разбора простейшего предложения (по выбору).
- Выводы по проделанной работе.

Допускается оформлять единый (совместный) отчёт для лабораторных работ №4 и №5.

Допускается (по соглашению с преподавателем) использовать для выполнения лабораторной работы иной класс синтаксических анализаторов (отличный от анализаторов на основе грамматик операторного предшествования). В этом случае отчет по лабораторной работе должен включать в себя краткое описание выбранного класса синтаксических анализаторов, а вместо данных, необходимых для построения анализатора на основе грамматик операторного предшествования, он должен содержать данные, необходимые для построения анализатора из выбранного класса.

Также допускается (по соглашению с преподавателем) использовать автоматизированные средства построения синтаксических анализаторов (YACC и ему подобные). В этом случае отчет по лабораторной работе должен включать в себя описание выбранного средства автоматизированного построения синтаксических анализаторов, информацию о классе КС-грамматик, лежащих в основе данного средства, и вместо данных, необходимых для построения анализатора на основе грамматик операторного предшествования, он должен содержать данные, необходимые для автоматизированного построения анализатора.

Использование другого класса синтаксических анализаторов, а также автоматизированных средств построения синтаксических анализаторов не освобождает исполнителя лабораторной работы от необходимости знать принципы функционирования синтаксических распознавателей на основе грамматик операторного предшествования.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какую роль выполняет синтаксический анализ в процессе компиляции?

2. Расскажите о классификации грамматик. Какие типы грамматик существуют?
3. Расскажите о классификации языков. Как связаны типы грамматик и языков?
4. Что такое КС-грамматики? Расскажите об их использовании в компиляторе.
5. Расскажите о грамматиках предшествования. Какие общие черты присущи всем грамматикам предшествования?
6. Расскажите о грамматиках операторного предшествования. Какие черты характеризуют грамматики операторного предшествования?
7. Дайте определения отношений предшествования для грамматик операторного предшествования.
8. Как вычисляются отношения предшествования для грамматик операторного предшествования?
9. Расскажите алгоритм построения множеств крайних левых и крайних правых символов.
10. Расскажите алгоритм построения множеств крайних левых и крайних правых терминальных символов.
11. Расскажите о задаче разбора. Что такое распознаватель языка?
12. Расскажите об общих принципах работы распознавателя языка.
13. Какие типы распознавателей существуют? Какие из них используются в компиляторе и на каких этапах?
14. Что такое МП-автомат? Расскажите о МП-автоматах.
15. Расскажите, как работает алгоритм «сдвиг-свертка» в общем случае (с возвратами).
16. Как работает алгоритм «сдвиг-свертка» для грамматик операторного предшествования? Поясните разбор предложения входного языка на своем примере.

ВАРИАНТЫ ИСХОДНЫХ ГРАММАТИК

Ниже приведены варианты грамматик. Во всех вариантах символ S является начальным символом грамматики; S, F, T и E обозначают нетерминальные символы. Терминальные символы выделены жирным шрифтом. Вместо терминального символа **a** должны подставляться лексемы в соответствии с вариантом задания.

1. S → **a :=** F**;**
F → F**+**T | T
T → T*****E | T/E | E
E → **(**F**)** | **-(**F**)** | **a**

2. S → **a :=** F**;**
F → F **or** T | F **xor** T | T
T → T **and** E | E
E → **(**F**)** | **not (**F**)** | **a**

3. S → F**;**
F → **if** E **then** T **else** F | **if** E **then** F | **a := a**
T → **if** E **then** T **else** T | **a := a**
E → **a<a** | **a>a** | **a=a**

4. S → F**;**
F → **for** T **do** F | **a := a**
T → **(**F**;** E**;** F**)** | **(;** E**;** F**)** | **(**F**;** E**;)** | **(;** E**;)**
E → **a<a** | **a>a** | **a=a**

5. S → F**;**
F → **case** E **of** T **end** | **a :=** E
T → **a: a :=** E | T**; a: a :=** E
E → E **+ a** | E **- a** | **a**

6. S → F**;**
F → **while** T **do a :=** E | **a :=** E
T → E < E | E > E | E = E
E → E **+ a** | E **- a** | **a**

ВАРИАНТЫ ЗАДАНИЙ

Варианты заданий приведены далее в таблице 6.

Таблица 6.

Варианты заданий для лабораторной работы №4.

№ варианта	№ варианта грамматики	Допустимые лексемы входного языка
1	1	Идентификаторы, десятичные числа с плавающей точкой
2	2	Идентификаторы, константы **true** и **false**
3	3	Идентификаторы, десятичные числа с плавающей точкой
4	4	Идентификаторы, десятичные числа с плавающей точкой
5	5	Идентификаторы, десятичные числа с плавающей точкой
6	6	Идентификаторы, десятичные числа с плавающей точкой
7	1	Идентификаторы, римские числа
8	2	Идентификаторы, константы **0** и **1**
9	3	Идентификаторы, римские числа
10	4	Идентификаторы, римские числа
11	5	Идентификаторы, римские числа
12	6	Идентификаторы, римские числа
13	1	Идентификаторы, шестнадцатеричные числа
14	2	Идентификаторы, шестнадцатеричные числа
15	3	Идентификаторы, шестнадцатеричные числа
16	4	Идентификаторы, шестнадцатеричные числа
17	5	Идентификаторы, шестнадцатеричные числа
18	6	Идентификаторы, шестнадцатеричные числа
19	1	Идентификаторы, символьные константы (в одинарных кавычках)
20	2	Идентификаторы, константы 'T' и 'F'
21	3	Идентификаторы, строковые константы (в двойных кавычках)
22	4	Идентификаторы, строковые константы (в двойных кавычках)
23	5	Идентификаторы, символьные константы (в одинарных кавычках)
24	6	Идентификаторы, символьные константы (в одинарных кавычках)

Примечание: требования к представлению римских чисел и шестнадцатеричных чисел соответствуют требованиям, изложенным в задании на лабораторную работу №2.

ЛАБОРАТОРНАЯ РАБОТА № 5
ПОСТРОЕНИЕ ПРОСТЕЙШЕГО ДЕРЕВА ВЫВОДА

Цель работы: получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования, обработка и представление результатов синтаксического анализа.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Общий алгоритм работы синтаксического анализатора

В данной лабораторной работе используется синтаксический анализатор на основе грамматик операторного предшествования, рассмотренный в предыдущей лабораторной работе №4.

Такой синтаксический анализатор работает, опираясь на построенную матрицу предшествования. На его вход поступает обработанный сканером текст исходной программы. Каждый идентификатор или константа представляются для него некоторым терминальным символом (в примере в лабораторной работе №4 он обозначен как **p**, а в задании – **a**). Тогда

первым шагом общего алгоритма анализа должно являться построение таблицы лексем (что уже выполнялось в предыдущей лабораторной работе №3).

По таблице лексем и матрице предшествования выполняется разбор входной цепочки. Результатом разбора является проверка цепочки на синтаксическую правильность и для правильных цепочек – построение последовательности правил вывода (для неправильных цепочек выдается сообщение об ошибке). Получение последовательности правил – второй шаг общего алгоритма анализа.

Третий шаг этого алгоритма – построение дерева вывода на основе полученной последовательности правил. Входными данными для этого шага алгоритма являются исходная грамматика и последовательность правил вывода, полученная при выполнении предыдущего шага алгоритма.

Первые два шага общего алгоритма работы синтаксического анализатора были рассмотрены выше при выполнении лабораторных работ №3 и №4. В данной лабораторной работе будет уделено внимание последнему (третьему) шагу этого алгоритма.

Вывод. Цепочки вывода

Определение вывода

Выводом называется процесс порождения предложения языка на основе правил определяющей язык грамматики. Чтобы дать формальное определение процессу вывода, необходимо ввести еще несколько дополнительных понятий.

Цепочка $\beta = \delta_1\gamma\delta_2$ называется *непосредственно выводимой* из цепочки $\alpha = \delta_1\omega\delta_2$ в грамматике $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$, $\mathbf{V} = \mathbf{VT}\cup\mathbf{VN}$, $\delta_1,\gamma,\delta_2 \in \mathbf{V}^*$, $\omega \in \mathbf{V}^+$, если в грамматике **G** существует правило: $\omega \to \gamma \in \mathbf{P}$. Непосредственная выводимость цепочки β из цепочки α обозначается так: $\alpha \Rightarrow \beta$. Согласно определению при непосредственном выводе $\alpha \Rightarrow \beta$ выполняется подстановка подцепочки γ вместо подцепочки ω. Иными словами, цепочка β выводима из цепочки α в том случае, если можно взять несколько символов в цепочке α, поменять их на другие символы согласно некоторому правилу грамматики и получить цепочку β.

В формальном определении непосредственной выводимости любая из цепочек δ_1 или δ_2 (а равно и обе эти цепочки) может быть пустой. В предельном случае вся цепочка α может быть заменена цепочкой β, тогда в грамматике **G** должно существовать правило: $\alpha \to \beta \in \mathbf{P}$.

Цепочка β называется *выводимой* из цепочки α (обозначается $\alpha \Rightarrow^* \beta$) в том случае, если выполняется одно из условий:

- эти две цепочки совпадают ($\alpha = \beta$);
- β непосредственно выводима из α ($\alpha \Rightarrow \beta$);
- $\exists$ цепочка γ, такая, что: γ выводима из α и β непосредственно выводима из γ ($\alpha \Rightarrow^* \gamma$ и $\gamma \Rightarrow \beta$).

Это рекурсивное определение выводимости цепочки. Суть его заключается в том, что цепочка β выводима из цепочки α, если $\alpha \Rightarrow \beta$ или же если можно построить последовательность непосредственно выводимых цепочек от α к β следующего вида: $\alpha \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_i \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \beta$, $n \geq 1$. В этой последовательности каждая последующая цепочка γ_i непосредственно выводима из предыдущей цепочки γ_{i-1}.

Такая последовательность непосредственно выводимых цепочек называется выводом или ***цепочкой вывода***. Каждый переход от одной непосредственно выводимой цепочки к следующей в цепочке вывода называется *шагом вывода*. Очевидно, что шагов вывода в цепочке вывода всегда на один больше, чем промежуточных цепочек. Если цепочка β непосредственно выводима из цепочки α: $\alpha \Rightarrow \beta$, то имеется всего один шаг вывода, если же две цепочки совпадают, то считается, что имеется ноль шагов вывода.

Если цепочка вывода из α к β содержит одну или более промежуточных цепочек (два или более шагов вывода), то она имеет специальное обозначение $\alpha \Rightarrow^+ \beta$ (говорят, что цепочка β

нетривиально выводима из цепочки α). Если количество шагов вывода известно, то его можно указать непосредственно у знака выводимости цепочек. Например, запись $\alpha \Rightarrow^4 \beta$ означает, что цепочка β выводится из цепочки α за 4 шага вывода.

Левосторонний и правосторонний выводы

Вывод называется *левосторонним*, если в нем на каждом шаге вывода правило грамматики применяется всегда к крайнему левому нетерминальному символу в цепочке. Другими словами, вывод называется левосторонним, если на каждом шаге вывода происходит подстановка цепочки символов на основании правила грамматики вместо крайнего левого нетерминального символа в исходной цепочке.

Аналогично, вывод называется *правосторонним*, если в нем на каждом шаге вывода правило грамматики применяется всегда к крайнему правому нетерминальному символу в цепочке.

Для грамматик типов 2 и 3 (КС-грамматик и регулярных грамматик) для любой сентенциальной формы всегда можно построить левосторонний или правосторонний выводы. Для грамматик других типов это не всегда возможно, так как по структуре их правил не всегда можно выполнить замену крайнего левого или крайнего правого нетерминального символа в цепочке.

Сентенциальная форма грамматики. Язык, заданный грамматикой

Вывод называется *законченным* (или *конечным*), если на основе цепочки β, полученной в результате этого вывода, нельзя больше сделать ни одного шага вывода. Иначе говоря, вывод называется законченным, если цепочка β, полученная в результате этого вывода, пустая или содержит только терминальные символы грамматики G(**VT**,**VN**,**P**,S): $\beta \in \mathbf{VT}^*$. Цепочка β, полученная в результате законченного вывода, называется *конечной* цепочкой вывода.

Цепочка символов $\alpha \in \mathbf{V}^*$ называется *сентенциальной формой* грамматики G(**VT**,**VN**,**P**,S), $\mathbf{V} = \mathbf{VT} \cup \mathbf{VN}$, если она выводима из целевого символа грамматики S: $S \Rightarrow^* \alpha$. Если цепочка $\alpha \in \mathbf{VT}^*$ получена в результате законченного вывода, то она называется *конечной сентенциальной формой*.

Язык **L**, заданный грамматикой G(**VT**,**VN**,**P**,S),– это множество всех конечных сентенциальных форм данной грамматики. Язык **L**, заданный грамматикой **G**, обозначается как L(**G**). Очевидно, что алфавитом такого языка L(**G**) будет множество терминальных символов грамматики **VT**, поскольку все конечные сентенциальные формы грамматики – это цепочки над алфавитом **VT**.

Следует помнить, что две грамматики G(**VT**,**VN**,**P**,S) и G'(**VT**',**VN**',**P**',S') называются эквивалентными, если эквивалентны заданные ими языки: L(**G**) = L(**G**'). Очевидно, что эквивалентные грамматики должны иметь, по крайней мере, пересекающиеся множества терминальных символов $\mathbf{VT} \cap \mathbf{VT}' \neq \varnothing$ (как правило, эти множества даже совпадают **VT**=**VT**'), а вот множества нетерминальных символов, правила грамматики и целевой символ у них могут кардинально отличаться.

Дерево вывода. Методы построения дерева вывода

Деревом вывода грамматики G(**VT**,**VN**,**P**,S) называется дерево (граф), которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

- каждая вершина дерева обозначается символом грамматики $A \in (\mathbf{VT} \cup \mathbf{VN} \cup \{\lambda\})$;
- корнем дерева является вершина, обозначенная целевым символом грамматики — S;
- листьями дерева (концевыми вершинами) являются вершины, обозначенные терминальными символами грамматики или символом пустой цепочки λ;

- если некоторый узел дерева обозначен нетерминальным символом A∈**VN**, а связанные с ним узлы — символами $b_1,b_2,\ldots,b_n$; n > 0, ∀n≥i>0: b_i∈(**VT**∪**VN**∪{λ}), то в грамматике G(**VT**,**VN**,**P**,S) существует правило $A \rightarrow b_1,b_2,\ldots,b_n \in$ **P**.

Из определения видно, что по структуре правил дерево вывода в указанном виде всегда можно построить только для грамматик типов 2 и 3 (контекстно-свободных и регулярных). Для грамматик других типов дерево вывода в таком виде можно построить не всегда (либо же оно будет иметь несколько иной вид).

Для того чтобы построить дерево вывода, достаточно иметь только цепочку вывода. Дерево вывода можно построить двумя способами: сверху вниз и снизу вверх. Для строго формализованного построения дерева вывода всегда удобнее пользоваться строго определенным выводом: либо левосторонним, либо правосторонним.

При построении дерева вывода сверху вниз построение начинается с целевого символа грамматики, который помещается в корень дерева. Затем в грамматике выбирается необходимое правило, и на первом шаге вывода корневой символ раскрывается на несколько символов первого уровня. На втором шаге среди всех концевых вершин дерева выбирается крайняя (крайняя левая – для левостороннего вывода, крайняя правая – для правостороннего) вершина, обозначенная нетерминальным символом, для этой вершины выбирается нужное правило грамматики, и она раскрывается на несколько вершин следующего уровня. Построение дерева заканчивается, когда все концевые вершины обозначены терминальными символами, в противном случае надо вернуться ко второму шагу и продолжить построение.

Построение дерева вывода снизу вверх начинается с листьев дерева. В качестве листьев выбираются терминальные символы конечной цепочки вывода, которые на первом шаге построения образуют последний уровень (слой) дерева. Построение дерева идет по слоям. На втором шаге построения в грамматике выбирается правило, правая часть которого соответствует крайним символам в слое дерева (крайним правым символам при правостороннем выводе и крайним левым – при левостороннем). Выбранные вершины слоя соединяются с новой вершиной, которая выбирается из левой части правила. Новая вершина попадает в слой дерева вместо выбранных вершин. Построение дерева закончено, если достигнута корневая вершина (обозначенная целевым символом), а иначе надо вернуться ко второму шагу и повторить его над полученным слоем дерева.

Поскольку все известные языки программирования имеют нотацию записи «слева – направо», компилятор также всегда читает входную программу слева на право (и сверху вниз, если программа разбита на несколько строк). Поэтому на практике построение дерева вывода удобнее выполнять методом «сверху вниз». Тогда для левостороннего вывода полученную в результате синтаксического анализа последовательность правил нужно применять в порядке «слева – направо», а для правостороннего вывода – в порядке «справа – налево».

По последовательности правил, полученной в результате выполнения лабораторной работы №4, легко строится цепочка вывода и дерево вывода. При построении дерева следует учитывать, что алгоритм разбора на основе грамматик операторного предшествования порождает правосторонний вывод, поэтому на каждом шаге на основе правила в цепочке следует заменять крайний правый нетерминальный символ (нижний правый в дереве). Это третий шаг общего алгоритма анализа.

Цепочки вывода для двух из рассмотренных в лабораторной работе №4 примеров будут иметь следующий вид:

Пример 3. Входная цепочка **-p^p&p**.

E → **-**E → **-**E**&**E → **-**E**&p** → **-**E^E**&p** → **-**E^**p&p** → **-p^p&p**

Пример 4. Входная цепочка **-p&p^p**.

E → **-**E → **-**E**&**E → **-**E**&**E^E → **-**E**&**E^**p** → **-**E**&p^p** → **-p&p^p**

Деревья вывода для этих двух примеров приведены на рис. 6.

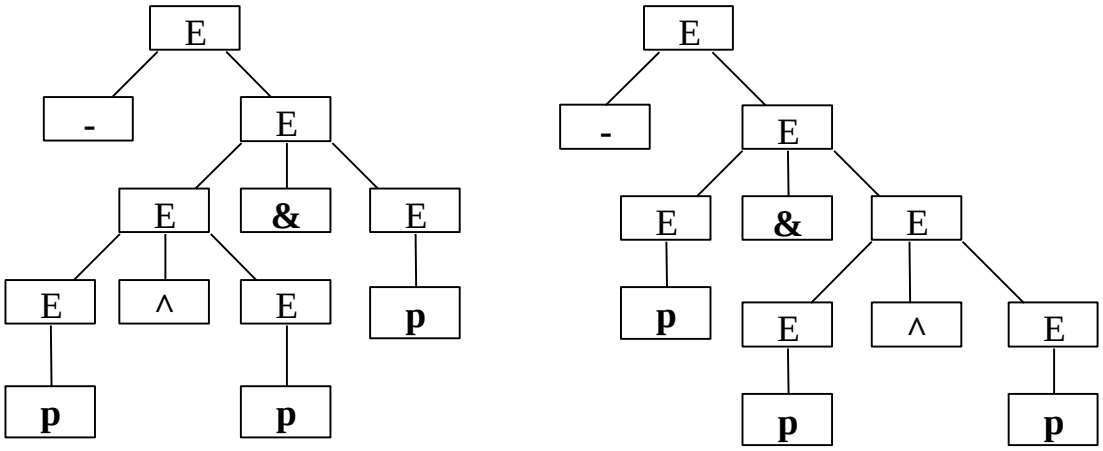


Рис. 6. Деревья вывода для цепочек из примеров 3 и 4 соответственно.

После построения дерева остается заменить терминальные символы (**p** или **a**) грамматики на соответствующие константы и идентификаторы из таблицы лексем. Построенное таким образом дерево и будет деревом синтаксического разбора предложения грамматики.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя.
2. Построить матрицу предшествования для заданной грамматики (по результатам лабораторной работы №4).
3. Выполнить построение дерева вывода для простейшего примера вручную по правилам заданной грамматики.
4. Подготовить и защитить отчет.
5. Написать и отладить программу на ЭВМ.
6. Сдать работающую программу преподавателю.

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать следующие разделы:

- Краткое изложение цели работы.
- Задание по лабораторной работе с описанием своего варианта.
- Запись заданной грамматики входного языка в форме Бэкуса-Наура.
- Пример выполнения разбора простейшего предложения (по результатам предыдущей лабораторной работы №4).
- Пример построения дерева вывода.
- Текст программы (оформляется только при необходимости по согласованию с преподавателем).
- Выводы по проделанной работе.

Требования к оформлению отчета полностью соответствуют требованиям, изложенным в задании на лабораторную работу №4.

Допускается оформлять единый (совместный) отчёт для лабораторных работ №4 и №5.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение алфавита, языка.
2. Какие способы определения языков существуют.
3. Что такое грамматика? Дайте определение грамматики.

4. Какую роль выполняет синтаксический анализ в процессе компиляции?
5. Дайте определение непосредственной выводимости цепочки символов.
6. Что такое вывод? Дайте определение цепочке вывода.
7. Дайте определение левостороннего и правостороннего вывода.
8. Что такое дерево вывода?
9. Какие методы построения дерева вывода существуют?
10. Что такое КС-грамматики? Расскажите об их использовании в компиляторе.
11. Поясните правила построения дерева вывода грамматики.
12. Что такое грамматика операторного предшествования?
13. Как вычисляются отношения для грамматик операторного предшествования?
14. Расскажите о задаче разбора. Что такое распознаватель языка?
15. Расскажите об общих принципах работы распознавателя языка.
16. Поясните построение дерева вывода на своем примере

ВАРИАНТЫ ЗАДАНИЙ

Для выполнения лабораторной работы требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием, порождает таблицу лексем и выполняет синтаксический разбор текста по заданной грамматике.

Текст на входном языке задается в виде символьного (текстового) файла. Допускается исходить из условия, что текст содержат не более одного предложения входного языка. Длину идентификаторов и строковых констант можно считать ограниченной 32 символами. Программа должна допускать ***наличие комментариев неограниченной длины*** во входном файле. Форму организации комментариев предлагается выбрать самостоятельно. При наличии лексических или синтаксических ошибок в исходном тексте программа должна выдавать сообщения об ошибках и корректно завершать работу.

Рекомендуется программу разбить на две составные части: лексический анализ и синтаксический анализ (построение цепочки вывода). Для построения лексического анализатора предлагается использовать результаты предыдущих лабораторных работ №2 и №3. Лексический анализатор должен выделять в тексте лексемы языка и менять их на терминальный символ грамматики (который в задании обозначен как “**a**”). Полученная после лексического анализа цепочка далее должна рассматриваться в соответствии с алгоритмом синтаксического разбора.

Варианты заданий полностью соответствуют вариантам, указанным в задании на лабораторную работу №4.

ЛАБОРАТОРНАЯ РАБОТА № 6
ГЕНЕРАЦИЯ ОБЪЕКТНОГО КОДА

Цель работы: изучение основных принципов генерации компилятором объектного кода, выполнение генерации объектного кода программы на основе результатов синтаксического анализа для заданного входного языка.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Генерация объектного кода. Синтаксически управляемый перевод

Генерация объектного кода - это перевод компилятором внутреннего представления исходной программы в линейную последовательность команд результирующей объектной программы на языке ассемблера или непосредственно на машинном языке (машинных кодах).

Генерация объектного кода выполняется после того, как выполнен синтаксический анализ программы и все необходимые действия по подготовке к генерации кода: распределено адресное

пространство под функции и переменные, проверено соответствие имен и типов переменных, констант и функций в синтаксических конструкциях исходной программы и т.д.

Чтобы компилятор мог построить код результирующей программы для всей синтаксической конструкции исходной программы, часто используется метод, называемый синтаксически управляемым переводом — ***СУ-переводом***. Идея СУ-перевода основана на том, что синтаксис и семантика языка взаимосвязаны, при этом семантика (то есть смысл) каждого предложения языка определяется синтаксисом. Это значит, что смысл предложения языка непосредственно зависит от синтаксической структуры этого предложения. Принцип СУ-перевода заключается в том, что синтаксис результирующей программы совпадает со структурой исходной программы, но при этом каждому правилу входного языка компилятора сопоставляется одно или несколько (или ни одного) правил выходного языка в соответствии с семантикой входных и выходных правил.

СУ-перевод – это основной метод порождения кода результирующей программы на основании результатов синтаксического анализа. Для удобства понимания сути метода можно считать, что результат синтаксического анализа представлен в виде дерева синтаксического анализа, хотя в реальных компиляторах это не всегда так.

Суть принципа СУ-перевода заключается в следующем: с каждой вершиной дерева синтаксического разбора N связывается цепочка некоторого промежуточного кода С(N). Код для вершины N строится путем сцепления (конкатенации) в фиксированном порядке последовательности кода С(N) и последовательностей кодов, связанных со всеми вершинами, являющимися прямыми потомками N. В свою очередь, для построения последовательностей кода прямых потомков вершины N потребуется найти последовательности кода для их потомков – потомков второго уровня вершины N – и т. д. Процесс перевода идет, таким образом, снизу вверх в строго установленном порядке, определяемом структурой дерева.

В общем случае необходимо иметь единообразную интерпретацию кода С(N), которая бы встречалась во всех ситуациях, где присутствует вершина N. В принципе, эта задача может оказаться нетривиальной, так как требует оценки смысла (семантики) каждой вершины дерева. При применении СУ-перевода задача интерпретации кода для каждой вершины дерева решается разработчиками компилятора.

Для того чтобы построить СУ-перевод по заданному дереву синтаксического разбора, необходимо найти последовательность кода для корня дерева. Если используемая для каждой вершины дерева N последовательность кодов С(N) построена в кодах результирующей программы, то полученная в результате последовательность кодов и будет искомым кодом результирующей программы. Если в процессе СУ-перевода используется некоторый внутренний промежуточный код компилятора, тогда после выполнения СУ-перевода требуется дополнительный алгоритм преобразования промежуточного кода в код результирующей программы, зависящий от того, какой промежуточный код используется.

СУ-перевод достаточно прост в реализации и, самое главное, позволяет компилятору строить результирующую программу без полноценного анализа семантики исходной программы. Как известно, компиляторы не имеют возможности выполнять анализ семантики исходных программ по причине существующих принципиальных ограничений, связанных с тем, что практически все языки программирования более сложны, чем КС-языки и кроме формализованных синтаксических ограничений, как правило, содержат семантические ограничения, зачастую задаваемые в форме неформализованных описаний. По этой причине метод СУ-перевода является основным методом порождения результирующего кода в современных компиляторах.

Метод СУ-перевода имеет ограничения, проистекающие из самой идеи этого метода: он применим только в том случае, если каждому элементу исходной программы N может быть однозначно сопоставлена цепочка некоторого кода С(N). Это сопоставление не должно зависеть от того, в каком месте исходной программы встречается элемент N – оно должно определяться только синтаксической структурой этого элемента. Это не ограничивает применимость метода для всех КС-языков, в том числе и для синтаксических конструкций языков программирования, поскольку

для них отсутствует контекстная зависимость элементов языка – действительно, в языке программирования смысл какого-либо оператора, функции, процедуры или другого элемента исходной программы не зависит от того, в каком месте программы он встречается. Однако по этой причине СУ-перевод нельзя применять для других, более сложных типов языков – КЗ-языков и, тем более, языков с фразовой структурой. Например, выполнить качественный перевод с одного естественного языка на другой с помощью СУ-перевода невозможно, поскольку смысл слов естественного языка существенно зависит от контекста.

Второй недостаток СУ-перевода также связан с тем, что данный метод не учитывает контекстную зависимость: поскольку фрагмент кода C(N), порождаемый для каждого элемента исходной программы N, не зависит от местоположения элемента, он не учитывает возможную зависимость соседних элементов между собой по данным или по выполняемым операциям. В результате код результирующей программы, порождаемый с помощью СУ-перевода, получается неэффективным.

Поэтому в компиляторах фаза генерации кода, как правило, состоит из двух этапов: генерации и оптимизации кода.

Алгоритм генерации объектного кода по дереву вывода

Формы внутреннего представления программ

Внутреннее представление программы – это структура данных, используемая компилятором для хранения информации об исходной программе после выполнения ее анализа. Внутреннее представление программы должно хранить информацию о содержании исходной программы и о ее структуре.

Возможны различные формы внутреннего представления синтаксических конструкций исходной программы в компиляторе. На этапе синтаксического разбора часто используется форма, именуемая деревом вывода (методы его построения рассматривались в предыдущих лабораторных работах). Но формы представления, используемые на этапах синтаксического анализа, оказываются неудобными в работе при генерации и оптимизации объектного кода. Поэтому в процессе генерации и оптимизации объектного кода внутреннее представление программы преобразуется в одну из соответствующих форм записи.

Примерами таких форм записи являются:

- обратная польская запись операций;
- тетрады операций;
- триады операций;
- команды ассемблера.

Перечисленные примеры не исчерпывают все возможные формы внутреннего представления программ, используемые в компиляторах. То, какие формы внутреннего представления использовать и на каких этапах компиляции, решают разработчики компилятора. В одном компиляторе на разных этапах может использоваться несколько различных форм внутреннего представления.

Обратная польская запись – это постфиксная запись операций (операторы записываются после операндов). Преимуществом ее является то, что все операции записываются непосредственно в порядке их выполнения, не используются скобки и приоритет операций. Она чрезвычайно эффективна в тех случаях, когда для вычислений используется стек.

Тетрады (или трехадресный код) представляют собой запись операций в форме из четырех составляющих: *<операция>(<операнд1>,<операнд2>,<результат>)*. Тетрады используются редко, так как требуют больше памяти для своего представления, чем триады, в явном виде не отражают взаимосвязи операций и, кроме того, плохо отображаются в команды ассемблера и машинные коды, так как в наборах команд большинства современных машин не встречаются операции с тремя операндами.

Триады (или двухадресный код) представляют собой запись операций в форме из трех составляющих: *<операция>(<операнд1>,<операнд2>)*, при этом один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады. Поэтому триады при записи последовательно номеруют для удобства указания ссылок одних триад на другие.

Например, выражение *A := B*C + D - B*10*, записанное в виде триад будет иметь вид:

1) ***** (B, C)
2) + (^1, D)
3) ***** (B, 10)
4) - (^2, ^3)
5) **:=** (A, ^4)

Здесь операции обозначены соответствующим знаком (при этом присвоение также является операцией), а знак ^ означает ссылку операнда одной триады на результат другой.

Команды ассемблера удобны тем, что при их использовании внутреннее представление программы полностью соответствует объектному коду и сложные преобразования не требуются. Однако использование команд ассемблера требует дополнительных структур для отображения их взаимосвязи. Кроме того, внутреннее представление программы получается зависимым от результирующего кода, а это значит, что при ориентации компилятора на другой результирующий код потребуется перестраивать как само внутреннее представление программы, так и методы его обработки в алгоритмах оптимизации (при использовании триад или тетрад этого не требуется).

Построение внутреннего представления программы по дереву вывода

Для построения внутреннего представления объектного кода (в дальнейшем - просто кода) по дереву вывода может использоваться простейшая рекурсивная процедура. Эта процедура, прежде всего, должна определить тип узла дерева – он соответствует типу операции, символ которой находится в листе дерева для текущего узла. Этот лист является средним листом узла дерева для бинарных операций и крайним левым листом – для унарных операций. После определения типа процедура строит код для узла дерева в соответствии с типом операции. Если все узлы следующего уровня для текущего узла есть листья дерева, то в код включаются операнды, соответствующие этим листьям, и получившийся код становится результатом выполнения процедуры. Иначе процедура должна рекурсивно вызвать сама себя для генерации кода нижележащих узлов дерева и результат выполнения включить в свой порожденный код.

Поэтому для построения внутреннего представления объектного кода по дереву вывода в первую очередь необходимо разработать формы представления объектного кода для четырех случаев, соответствующих видам текущего узла дерева вывода:

- оба нижележащих узла дерева - листья (терминальные символы грамматики);
- только левый нижележащий узел является листом дерева;
- только правый нижележащий узел является листом дерева:
- оба нижележащих узла не являются листьями дерева.

Рассмотрим построение двух видов внутреннего представления по дереву вывода:

- построение ассемблерного кода по дереву вывода;
- построение списка триад по дереву вывода.

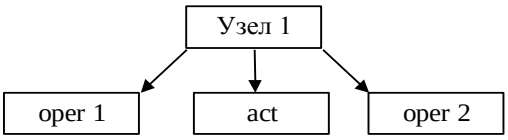
Построение ассемблерного кода по дереву вывода

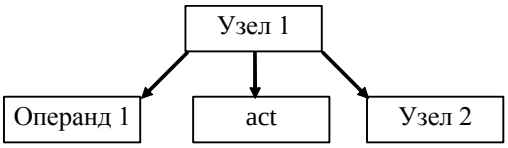
В качестве языка ассемблера возьмем язык ассемблера процессоров типа Intel 80x86. При этом будем считать, что операнды могут быть помещены в 16-разрядные регистры процессора и в коде результирующей объектной программы могут использоваться регистры AX (аккумулятор) и DX (регистр данных), а также стек для хранения промежуточных результатов.

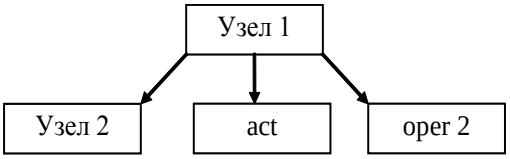
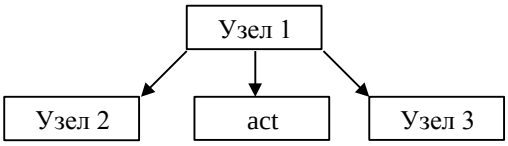
Тогда четырем формам текущего узла дерева будут соответствовать следующие фрагменты кода на языке ассемблера (таблица 7):

Таблица 7.

Преобразование типовых узлов дерева вывода в код на языке ассемблера

Вид узла дерева	Результирующий код	Примечание
	mov ax,oper1 act ax,oper2	act - команда соответствующей операции oper1,oper2 - операнды (листья дерева)

	Code(Узел 2)
	mov dx,ax
	mov ax,oper1
	act ax,dx
		Узел 2 - нижележащий узел (не лист!) дерева
		Code(Узел 2) - код, порождаемый процедурой для нижележащего узла

	Code(Узел 2) act ax,oper2	Code(Узел 2) - код, порождаемый процедурой для нижележащего узла
	Code(Узел 2) push ax Code(Узел 3) mov dx,ax pop ax act ax,dx	Code(Узел 2) - код, порождаемый процедурой для нижележащего узла Code(Узел 3) - код, порождаемый процедурой для нижележащего узла push и pop - команды сохранения результатов в стеке и извлечения результатов из стека

Рассмотрим пример дерева вывода для выражения *A := B*C + D - B*10* на рис. 7 и соответствующий ему фрагмент кода на языке ассемблера, построенный по описанным выше правилам (обратите внимание, что для операции присваивания используется отдельный код, не подпадающий под общие правила):

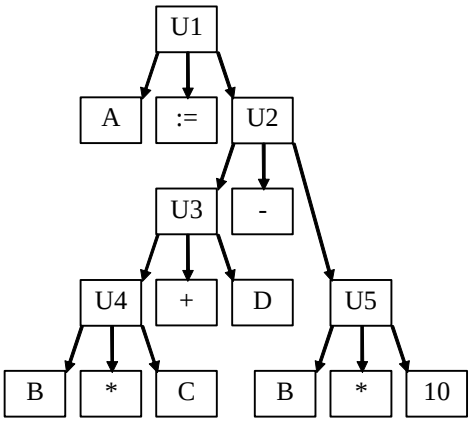


Рис. 7. Дерево вывода для арифметического выражения.

```
Шаг1: Code(U2)
      mov A,ax      ;операция присваивания
Шаг 2:      Code(U3)
      push ax
      Code(U5)
      mov dx,ax
      pop ax
      sub ax,dx
      mov A,ax      ;операция присваивания
Шаг 3:      Code(U4)
      add ax,D
      push ax
      Code(U5)
      mov dx,ax
      pop ax
      sub ax,dx
      mov A,ax      ;операция присваивания
Шаг 4:      mov ax,B
      mul ax,C
      add ax,D
      push ax
      Code(U5)
      mov dx,ax
      pop ax
      sub ax,dx
      mov A,ax      ;операция присваивания
Шаг 5:      mov ax,B
      mul ax,C
      add ax,D
      push ax
      mov ax,B
      mul ax,10
      mov dx,ax
      pop ax
      sub ax,dx
      mov A,ax      ;операция присваивания
```

Полученный объектный код на языке ассемблера, очевидно, может быть оптимизирован, однако для его обработки требуются специальные (ориентированные именно на данный язык ассемблера) методы и структуры, учитывающие взаимосвязь операций. Кроме того, ориентация на определенный язык ассемблера сводит на нет универсальность метода. Так в приведенном примере используется команда mul, которая в ранних версиях процессоров фирмы Intel имеет ограничения на типы операндов, а следовательно, в универсальном компиляторе не может быть использована так, как в данном примере. Это потребует, чтобы генерация кода для узлов дерева шла в зависимости не только от операндов, но и от типа операции (даже в приведенном примере такую зависимость пришлось установить для операции присваивания).

Обычно такие проблемы решаются таким образом, что вместо команд непосредственно языка ассемблера используются команды некоторого близкого к нему промежуточного псевдокода. Большинство этих команд один в один отображаются затем в команды языка ассемблера, другие же однозначно преобразуются в фиксированную последовательность команд.

Построение списка триад по дереву вывода.

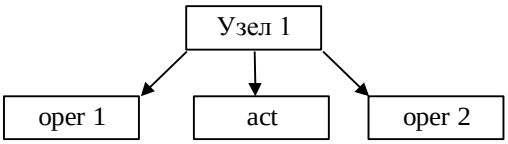
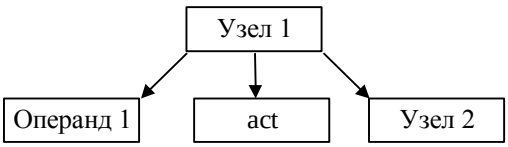
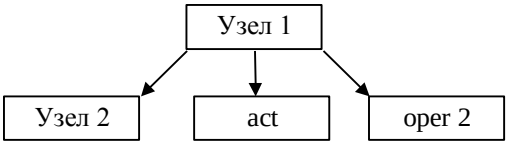
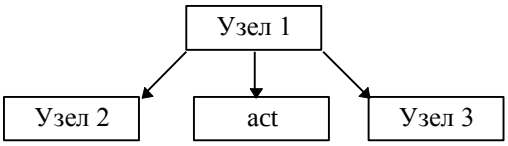
Триады являются универсальной, машинно-независимой формой внутреннего представления в компиляторе результирующей объектной программы, а потому не требуют оговорки дополнительных условий при генерации кода. Триады взаимосвязаны между собой,

поэтому для установки корректной взаимосвязи процедура генерации кода должна получать также текущий номер i очередной триады.

Тогда четырем формам текущего узла дерева будут соответствовать последовательности триад объектного кода (таблица 8):

Таблица 8.

Преобразование типовых узлов дерева вывода в последовательность триад

Вид узла дерева	Результирующий код	Примечание
	i) act (oper1,oper2)	act - тип триады oper1,oper2 - операнды (листья дерева вывода)
	i) Code(Узел 2,i) i+j) act(oper1,^i+j-1)	Узел 2 - нижележащий узел дерева вывода Code(Узел 2,i) - последовательность триад, порождаемая для Узла2, начиная с триады с номером i j - количество триад, порождаемых для Узла2
	i) Code(Узел 2,i) i+j) act(^i+j-1,oper2)	Узел 2 - нижележащий узел дерева вывода Code(Узел 2,i) - последовательность триад, порождаемая для Узла2, начиная с триады с номером i j - количество триад, порождаемых для Узла2
	i) Code(Узел 2,i) i+j) Code(Узел 3,i+j) i+j+k) act(^i+j-1,^i+j+k-1)	Узел 2, Узел 3 - нижележащие узлы дерева вывода Code(Узел 2,i) - последовательность триад, порождаемая для Узла2, начиная с триады с номером i j - количество триад, порождаемых для Узла2 Code(Узел 3,i+j) - последовательность триад, порождаемая для Узла3, начиная с триады с номером i+j k - количество триад, порождаемых для Узла3

Рассмотрим тот же пример дерева вывода для выражения *A := B*C + D - B*10* на рис. 8 и соответствующую ему последовательность триад:

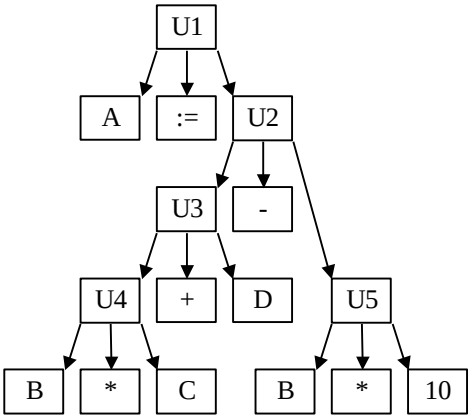


Рис. 8. Дерево вывода для арифметического выражения.

Шаг 1:
1) Code(U2,1)
i) :=(A,^i-1)

Шаг 2:
1) Code(U3,1)
j) Code(U5,j)
i-1) -(^j-1,^i-2)
i) :=(A,^i-1)

Шаг 3:
1) Code(U4,1)
k) +(^k-1,D)
j) Code(U5,j)
i-1) -(^j-1,^i-2)
i) :=(A,^i-1)

Шаг 4:
1) *(B,C)
2) +(^1,D)
3) Code(U5,3)
i-1) -(^j-1,^i-2)
i) :=(A,^i-1)

Шаг 5:
1) *(B,C)
2) +(^1,D)
3) *(B,10)
4) -(^2,^3)
5) :=(A,^4)

Следует обратить внимание, что в данном алгоритме последовательные номера триад (а следовательно, и ссылки на них) устанавливаются не сразу. Это не имеет значение при рекурсивной организации процедуры, но при другом способе обхода дерева вывода в программе генерации кода лучше увязывать триады между собой именно по ссылке (указателю), а не по порядковому номеру.

Для триад разработаны универсальные (машинно-независимые) алгоритмы оптимизации кода. После их выполнения (оптимизации внутреннего представления) триады могут быть преобразованы в команды на языке ассемблера.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя.
2. Изучить алгоритм генерации объектного кода по дереву синтаксического разбора.
3. Разработать фрагменты объектного кода, реализующие в виде триад простейшие операции в заданной грамматике.
4. Выполнить генерацию объектного кода вручную для выбранного простейшего примера. Проверить корректность результата.
5. Подготовить и защитить отчет.
6. Написать и отладить программу на ЭВМ.
7. Сдать работающую программу преподавателю.

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать следующие разделы:

- Задание по лабораторной работе (согласно своему варианту).
- Краткое изложение цели работы.
- Запись заданной грамматики входного языка в форме Бэкуса-Наура.
- Фрагменты объектного кода в виде триад для операций заданной грамматики.
- Простейший пример генерации кода по дереву синтаксического разбора.
- Текст программы (оформляется только при необходимости по согласованию с преподавателем).
- Выводы по проделанной работе

Допускается оформлять единый (совместный) отчёт для лабораторных работ №6 и №7.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое транслятор, компилятор и интерпретатор? В чем их отличия?
2. Расскажите об общей структуре компилятора.
3. Как строится дерево вывода (синтаксического разбора)? Какие исходные данные необходимы для его построения?
4. Какие действия выполняет компилятор при подготовке к генерации объектного кода?
5. Какую роль выполняет генерация объектного кода в процессе компиляции?
6. Может ли компилятор полноценно анализировать смысл (семантику) исходной программы? Обоснуйте свой ответ.
7. Расскажите, что такое синтаксически управляемый перевод.
8. Обоснуйте границы применимости метода синтаксически управляемого перевода.
9. Объясните преимущества и недостатки метода синтаксически управляемого перевода.
10. Объясните, кто и как устанавливает зависимость между элементами исходной программы и фрагментами кода результирующей программы. Является ли эта зависимость однозначной?
11. За счет чего обеспечивается возможность генерации кода на разных объектных языках по одному и тому же дереву?
12. Объясните работу алгоритма генерации объектного кода по дереву синтаксического разбора.
13. Какие данные необходимы компилятору для генерации объектного кода? Какие действия выполняет компилятор перед генерацией?
14. Какие формы внутреннего представления программ в компиляторе существуют?
15. Что такое триады и для чего они используются?
16. Является ли результирующая программа, построенная компилятором, оптимальной? Обоснуйте свой ответ.

ВАРИАНТЫ ЗАДАНИЙ

Для выполнения лабораторной работы требуется написать программу, которая на основании дерева синтаксического разбора порождает объектный код. В качестве исходного дерева синтаксического разбора рекомендуется взять дерево, которое порождает программа, построенная по заданию предыдущей лабораторной работы №5.

Программу рекомендуется построить из двух основных частей: первая часть – порождение дерева синтаксического разбора (по результатам лабораторной работы №5), вторая часть – реализация алгоритма порождения объектного кода по дереву разбора.

Результатам работы должна быть построенная на основании заданного предложения грамматики программа на объектном языке. В качестве объектного языка предлагается взять триады (возможен выбор другого объектного языка по согласованию с преподавателем). Все встречающиеся в исходной программе идентификаторы считать простыми скалярными переменными, не требующими выполнения преобразования типов. Ограничения на длину идентификаторов и констант соответствуют требованиям предыдущих лабораторных работ.

Варианты заданий полностью соответствуют вариантам заданий для лабораторной работы №4.

Для выполнения работы рекомендуется использовать результаты, полученные в ходе выполнения работ №3, №4 и №5.

ЛАБОРАТОРНАЯ РАБОТА № 7
ОПТИМИЗАЦИЯ ОБЪЕКТНОГО КОДА

Цель работы: изучение основных принципов оптимизации компилятором объектного кода для линейного участка программы, ознакомление с методами оптимизации результирующего объектного кода с помощью методов свертки объектного кода и исключения лишних операций.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Оптимизация. Определение и основные принципы оптимизации программ

Оптимизация программы – это обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе с целью получения более эффективной результирующей объектной программы. Оптимизация выполняется на этапах подготовки к генерации и непосредственно при генерации объектного кода.

Как уже было сказано выше в лабораторной работе №6, оптимизация является вынужденной мерой, с помощью которой компилятор компенсирует недостаток используемого им метода порождения кода результирующей программы – СУ-перевода. Вызвано это тем, что компилятор не имеет возможности анализировать смысл исходной программы, поэтому вынужден использовать СУ-перевод. Использование СУ-перевода, в свою очередь, ведет к необходимости применения оптимизации.

Конечной целью оптимизации является увеличение эффективности результирующей программы, порождаемой компилятором. Для оценки эффективности результирующей программы существует два основных критерия – быстродействие (скорость выполнения программы) и потребность в ресурсах (количество вычислительных ресурсов – прежде всего, оперативной памяти – необходимых для выполнения программы). Многие методы оптимизации позволяют повысить эффективность программы по обоим критериям. Но существуют методы оптимизации, которые повышают эффективность по одному критерию одновременно снижая ее по другому. Так, например, возможна оптимизация с минимизацией размера программы, которая ведет к снижению скорости ее выполнения. Поэтому у многих современных компиляторов существуют возможности выбора приоритетного критерия оптимизации, на основе которого оценивается эффективность результирующей объектной программы.

При выполнении любой оптимизации не требуется изменять текст программы на исходном языке – оптимизацию выполняет компилятор, а не разработчик программы.

Оптимизация не является обязательным этапом компиляции – в принципе, компилятор может совсем не выполнять оптимизацию. Однако большинство существующих компиляторов в той или иной степени используют оптимизацию. Лучшие оптимизирующие компиляторы могут получать объектные программы из сложных исходных программ, написанных на языках высокого уровня, сопоставимые по качеству с аналогичными по смыслу программами на языке ассемблера. Временные и трудовые затраты на создание такой программы существенно меньше, чем при ее реализации на ассемблере.

Все эти преимущества говорят в пользу применения оптимизации. Единственным, но существенным недостатком оптимизации является необходимость тщательной ее проработки при создании компилятора. Используемые методы оптимизации ни при каких условиях не должны приводить к изменению «смысла», т.е. семантики результирующей программы (т.е. к таким ситуациям, когда результат выполнения программы изменяется после ее оптимизации). К сожалению, не все методы оптимизации, используемые создателями компиляторов, могут быть теоретически обоснованы и доказаны для всех возможных видов исходных программ. Поэтому большинство компиляторов предусматривает возможность отключать те или иные из возможных

методов оптимизации, либо полностью отключать оптимизацию результирующей программы. Применение оптимизации также нецелесообразно в процессе отладки исходной программы.

Различаются две основные категории оптимизирующих преобразований:

- преобразования программы в форме ее внутреннего представления в компиляторе, не зависящие от результирующего объектного языка;
- преобразования результирующей объектной программы.

Последний тип преобразований может зависеть не только от свойств объектного языка (что очевидно), но и от архитектуры целевой вычислительной системы, на которой будет выполняться результирующая программа. Так, например, при оптимизации может учитываться объем кэш-памяти и методы организации конвейерных операций центрального процессора. Этот тип преобразований здесь рассматриваться не будет. Во многих случаях эти преобразования являются «ноу-хау» производителей компиляторов и при этом строго ориентированы на определенные архитектуры вычислительных машин.

Оптимизация различных элементов программы. Линейный участок программы

Используемые методы оптимизации программы зависят от типов синтаксических конструкций исходного языка. Теоретически разработаны и применяются на практике методы оптимизации для многих типовых конструкций языков программирования. Но далее будут рассмотрены только методы оптимизации линейных участков – они встречаются в любой программе и составляют существенную часть программного кода.

Линейный участок программы - это выполняемая по порядку последовательность операций, имеющая один вход и один выход. Чаще всего линейный участок содержит последовательность арифметических операций, логических операций, адресной арифметики и операторов присвоения значений переменным.

Линейные участки обязательно встречаются в любой программе, поэтому их оптимизация играет существенную роль в общей оптимизации кода.

В свою очередь, для линейных участков разработано множество различных методов оптимизации, но в данной лабораторной работе рассматриваются и используются только два из них – свертка объектного кода и исключение лишних операций.

В предыдущей лабораторной работе №6 были рассмотрены некоторые формы внутреннего представления программ. Здесь для оптимизации будет использоваться представление линейных участков программы в форме триад [2, 8, 10, 15].

Для триад разработаны универсальные (машинно-независимые) алгоритмы оптимизации кода. После их выполнения (оптимизации внутреннего представления) триады могут быть преобразованы в команды на языке ассемблера.

Оптимизация объектного кода методом свертки

Свертка объектного кода - это выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны. Поэтому нет необходимости многократно выполнять их в самой результирующей программе – вполне достаточно один раз выполнить их при ее компиляции.

Простейший вариант свертки – выполнение компилятором операций, операндами которых являются константы. Несколько более сложен процесс определения тех операций, значения которых могут быть известны в результате выполнения других операций. Для этого служит специальный алгоритм свертки.

Алгоритм свертки работает со специальной таблицей **T**, которая содержит пары *<переменная>*-*<константа>* для всех переменных, значения которых уже известны. Кроме того, алгоритм свертки помечает те операции во внутреннем представлении программы, для которых в результате свертки уже не требуется генерация кода. Так как при выполнении алгоритма свертки учитывается взаимосвязь операций, то удобной формой представления для него являются триады,

так как в других формах представления операций (таких как тетрады или команды ассемблера) требуются дополнительные структуры, чтобы отразить связь результатов одних операций с операндами других.

Алгоритм свертки объектного кода для триад последовательно просматривает триады линейного списка и для каждой триады делает следующее:

1. Если операнд есть переменная, которая содержится в таблице **T**, то операнд заменяется на соответствующее значение константы.

2. Если операнд есть ссылка на особую триаду типа *C(K,0)*, то операнд заменяется на значение константы *K*.

3. Если все операнды триады являются константами, то триада может быть свернута. Тогда данная триада выполняется и вместо нее помещается особая триада вида *C(K,0)*, где *K* - константа, результат выполнения свернутой триады. (При генерации кода для особой триады объектный код не порождается, а потому она в дальнейшем может быть просто исключена).

4. Если триада является присваиванием типа `A:=B`, тогда:

- если `B` - константа, то `A` со значением константы заносится в таблицу **T** (если там уже было старое значение для `A`, то это старое значение исключается).
- если `B` - не константа, то `A` вообще исключается из таблицы **T**, если оно там есть.

Очевидно, что свертка объектного кода оптимизирует программу сразу по двум критериям. Скорость выполнения программы увеличивается за счет того, что часть операций выполняется на этапе компиляции, а не при выполнении результирующей программы, а объем результирующей программы (а, следовательно, и объем оперативной памяти, требуемой для ее выполнения) сокращается за счет исключения из нее операций, выполненных на этапе компиляции.

Очевидно также, что время, затраченное на компиляцию, при использовании свертки объектного кода увеличивается за счет выполнения на этапе компиляции части операций результирующей программы. Это время точно соответствует тому времени, которое потребовалось бы результирующей программе на выполнение тех же самых операций. Однако и тут свертка объектного кода дает преимущество: во-первых, компиляция любой программы выполняется только один раз, а сама программа может выполняться многократно, что в любом случае дает выигрыш во времени выполнения. Во-вторых, фрагменты линейных участков кода, оптимизируемые с помощью свертки объектного кода, могут встречаться в многократно выполняемых частях программы – процедурах, функциях, операторах цикла – тогда выигрыш во времени выполнения будет еще более существенным.

Рассмотрим пример выполнения алгоритма свертки объектного кода.

Пусть фрагмент исходной программы (записанной на языке типа Паскаль) имеет вид:

```
I := 1 + 1;
I := 3;
J := 6*I + I;
```

Ее внутренне представление в форме триад будет иметь вид:

1) `+ (1,1)`
2) `:= (I, ^1)`
3) `:= (I, 3)`
4) `* (6, I)`
5) `+ (^4, I)`
6) `:= (J, ^5)`

Процесс выполнения алгоритма свертки можно отразить в таблице 9:

Таблица 9.

Пример работы алгоритма свертки

Триада	Шаг 1	Шаг 2	Шаг 3	Шаг 4	Шаг 5	Шаг 6
1	`C (2, 0)`	`C (2, 0)`	`C (2, 0)`	`C (2, 0)`	`C (2, 0)`	`C (2, 0)`

Триада	Шаг 1	Шаг 2	Шаг 3	Шаг 4	Шаг 5	Шаг 6
2	:= (I, ^1)	:= (I, 2)	:= (I, 2)	:= (I, 2)	:= (I, 2)	:= (I, 2)
3	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)	:= (I, 3)
4	* (6, I)	* (6, I)	* (6, I)	C (18, 0)	C (18, 0)	C (18, 0)
5	+ (^4, I)	+ (^4, I)	+ (^4, I)	+ (^4, I)	C (21, 0)	C (21, 0)
6	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, ^5)	:= (J, 21)
T	(,)	(I, 2)	(I, 3)	(I, 3)	(I, 3)	(I, 3) (J, 21)

Если исключить особые триады типа *C(K,0)* (которые не порождают объектного кода), то в результате выполнения свертки получим следующую последовательность триад:

1) := (I, 2)
2) := (I, 3)
3) := (J, 21)

Оптимизация объектного кода методом исключения лишних операций

Определим понятие ***лишней операции***. Операция линейного участка с порядковым номером *i* считается лишней, если существует более ранняя идентичная ей операция с порядковым номером *j* (*i*<*j*) и никакой операнд, от которого зависит операция *i*, не изменялся никакой третьей операций, имеющей порядковый номер между *i* и *j*. Очевидно, что лишние операции могут быть исключены из результирующей программы.

Исключение лишних операций, также как и свертка объектного кода, оптимизирует результирующую программу сразу по двум критериям. Скорость выполнения результирующей программы увеличивается, так как сокращается количество выполняемых операций, а объем результирующей программы (а, следовательно, и объем оперативной памяти, требуемой для ее выполнения) тоже сокращается за счет исключения операций из нее.

Рассмотрим алгоритм исключения лишних операций, обрабатывающий внутреннее представление программы в форме триад. Этот алгоритм исключения лишних операций просматривает операции в порядке их следования. Также как и алгоритму свертки, алгоритму исключения лишних операций проще всего работать с триадами, потому что они в явном виде отражают взаимосвязь операций.

Чтобы следить за внутренней зависимостью переменных и триад алгоритм присваивает им некоторые значения, называемые числами зависимости, по следующим правилам:

- изначально для каждой переменной ее число зависимости равно 0, так как в начале работы программы значение переменной не зависит ни от какой триады;
- после обработки *i*-ой триады, в которой переменной *A* присваивается некоторое значение, число зависимости *A* (dep(*A*)) получает значение *i*, так как значение *A* теперь зависит от данной *i*-ой триады;
- при обработке *i*-ой триады ее число зависимости (dep(*i*)) принимается равным значению: 1+(максимальное из чисел зависимости операндов).

Таким образом, при использовании чисел зависимости триад и переменных можно утверждать, что если *i*-ая триада идентична *j*-ой триаде (*j*<*i*), то *i*-ая триада считается лишней в том и только в том случае, когда dep(*i*)=dep(*j*).

Алгоритм исключения лишних операций использует в своей работе также особого вида триады *SAME(j,0)*, которые означают, что некоторая триада *i* идентична триаде *j*.

Алгоритм исключения лишних операций последовательно просматривает триады линейного участка. Он состоит из следующих шагов, выполняемых для каждой триады:

1. Если операнд ссылается на особую триаду вида *SAME(j,0)*, то он заменяется на ссылку на триаду с номером *j* (^*j*).

2. Вычисляется число зависимости текущей триады с номером *i*, исходя из чисел зависимости ее операндов.

3. Если существует идентичная *j*-ая триада, причем *j*<*i* и dep(*i*)=dep(*j*), то текущая триада *i* заменяется на триаду особого вида *SAME(j,0)*.

4. Если текущая триада есть присвоение, то вычисляется число зависимости соответствующей переменной.

Рассмотрим работу алгоритма на примере:

```
D := D + C*B;
A := D + C*B;
C := D + C*B;
```

Этому фрагменту программы будет соответствовать следующая последовательность триад:

1) `* (C, B)`
2) `+ (D, ^1)`
3) `:= (D, ^2)`
4) `* (C, B)`
5) `+ (D, ^4)`
6) `:= (A, ^5)`
7) `* (C, B)`
8) `+ (D, ^7)`
9) `:= (C, ^8)`

Работу алгоритма отобразим в таблице 10.

Теперь, если исключить триады особого вида SAME(j,0), то в результате выполнения алгоритма получим следующую последовательность триад:

1) `* (C, B)`
2) `+ (D, ^1)`
3) `:= (D, ^2)`
4) `+ (D, ^1)`
5) `:= (A, ^4)`
6) `:= (C, ^4)`

Обратите внимание, что в итоговой последовательности изменилась нумерация триад и номера в ссылках одних триад на другие. Если в программе компилятора в качестве ссылок использовать не номера триад, а непосредственно указатели на них, то изменение ссылок в таком варианте не потребуется.

Таблица 10.

Пример работы алгоритма исключения лишних операций.

Обрабатываемая триада *i*	Числа зависимости переменных A	B	C	D	Числа зависимости триад dep(*i*)	Триады, полученные после выполнения алгоритма
1) * (C, B)	0	0	0	0	1	1) * (C, B)
2) + (D, ^1)	0	0	0	0	2	2) + (D, ^1)
3) := (D, ^2)	0	0	0	3	3	3) := (D, ^2)
4) * (C, B)	0	0	0	3	1	4) SAME (1, 0)
5) + (D, ^4)	0	0	0	3	4	5) + (D, ^1)
6) := (A, ^5)	6	0	0	3	5	6) := (A, ^5)
7) * (C, B)	6	0	0	3	1	7) SAME (1, 0)
8) + (D, ^7)	6	0	0	3	4	8) SAME (5, 0)
9) := (C, ^8)	6	0	9	3	5	9) := (C, ^5)

Общий алгоритм генерации и оптимизации объектного кода

Теперь рассмотрим общий вариант алгоритма генерации кода, который получает на входе дерево вывода (построенное в результате синтаксического разбора) и создает по нему фрагмент объектного кода линейного участка результирующей программы.

Алгоритм должен выполнить следующую последовательность действий:

- построить последовательность триад на основе дерева вывода;
- выполнить оптимизацию кода методом свертки;
- выполнить оптимизацию кода методом исключения лишних операций;
- преобразовать последовательность триад в последовательность команд на языке ассемблера (полученная последовательность команд и будет результатом выполнения алгоритма).

Алгоритм преобразования триад в команды языка ассемблера - это единственная машинно-зависимая часть общего алгоритма. При преобразовании компилятора для работы с другим результирующим объектным кодом потребуется изменить только эту часть, при этом все алгоритмы оптимизации и внутреннее представление программы останутся неизменными.

В данной работе алгоритм преобразования триад в команды языка ассемблера предлагается разработать самостоятельно. В тривиальном виде такой алгоритм заменяет каждую триаду на последовательность соответствующих команд, а результат ее выполнения запоминается во временной переменной с некоторым именем (например, `TMPi`, где *i* - номер триады). Тогда вместо ссылки на эту триаду в другой триаде будет подставлено значение этой переменной. Однако алгоритм может предусматривать и оптимизацию временных переменных.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя.
2. Выполнить генерацию объектного кода вручную для выбранного простейшего примера (по результатам лабораторной работы №6).
3. Изучить алгоритмы оптимизации результирующего кода методом свертки и методом исключения лишних операций.
4. Выполнить вручную оптимизацию простейшего примера объектного кода методом свертки и методом исключения лишних операций.
5. Подготовить и защитить отчет.
6. Написать и отладить программу на ЭВМ.
7. Сдать работающую программу преподавателю.

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать следующие разделы:

- Задание по лабораторной работе (согласно своему варианту).
- Краткое изложение цели работы.
- Запись заданной грамматики входного языка в форме Бэкуса-Наура.
- Простейший пример генерации кода по дереву синтаксического разбора (по результатам лабораторной работы №6).
- Пример оптимизации кода, построенного по дереву синтаксического разбора.
- Текст программы (оформляется только при необходимости по согласованию с преподавателем).
- Выводы по проделанной работе.

Допускается оформлять единый (совместный) отчёт для лабораторных работ №6 и №7.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Расскажите об общей структуре компилятора.
2. Как строится дерево вывода (синтаксического разбора)? Какие исходные данные необходимы для его построения?
3. Объясните работу алгоритма генерации объектного кода по дереву синтаксического разбора.
4. Расскажите, что такое синтаксически управляемый перевод.

5. За счет чего обеспечивается возможность генерации кода на разных объектных языках по одному и тому же дереву?
6. Какую роль выполняет генерация объектного кода в процессе компиляции?
7. Какие данные необходимы компилятору для генерации объектного кода? Какие действия выполняет компилятор перед генерацией?
8. Какие формы внутреннего представления программы существуют?
9. Что такое триады и для чего они используются?
10. Дайте определение понятию оптимизации программы. Для чего используется оптимизация?
11. Объясните, почему генерацию программы приходится проводить в два этапа: генерация и оптимизация.
12. Какие существуют методы оптимизации объектного кода?
13. Объясните работу алгоритма свертки.
14. Что такое лишняя операция? Дайте определение лишней операции.
15. Что такое “число зависимости”? Для чего оно используется и как вычисляется?
16. Объясните работу алгоритма исключения лишних операций на основе чисел зависимости.

ВАРИАНТЫ ЗАДАНИЙ

Для выполнения лабораторной работы требуется написать программу, которая на основании дерева синтаксического разбора порождает объектный код и выполняет затем его оптимизацию. В качестве исходных данных рекомендуется взять результат выполнения предыдущей лабораторной работы №6.

Программу рекомендуется построить из трех основных частей: первая часть – порождение дерева синтаксического разбора (по результатам лабораторной работы №5), вторая часть – реализация алгоритма порождения объектного кода по дереву разбора (по результатам лабораторной работы №6), и третья часть – оптимизация порожденного объектного кода. Результатам работы должна быть построенная на основании заданного предложения грамматики программа на объектном языке. В качестве объектного языка предлагается взять триады или язык ассемблера для процессоров типа Intel 80x86 в реальном режиме - выбранный язык должен совпадать с языком представления результата в предыдущей лабораторной работе. Все встречающиеся в исходной программе идентификаторы считать простыми скалярными переменными, не требующими выполнения преобразования типов. Ограничения на длину идентификаторов и констант соответствуют требованиям предыдущей лабораторной работы.

Варианты заданий полностью соответствуют вариантам заданий для лабораторных работ №4-№6.

Для выполнения работы рекомендуется использовать результаты, полученные в ходе выполнения лабораторных работ №3, №4, №5 и №6.

ЛАБОРАТОРНАЯ РАБОТА № 8
ПОСТРОЕНИЕ РАСПОЗНАВАТЕЛЯ ДЛЯ АНАЛИЗА ОПИСАНИЙ ТИПОВ ДАННЫХ И ПЕРЕМЕННЫХ

Цель работы: построение распознавателя исходного текста программы, содержащего описания типов данных, структур данных и переменных.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В качестве распознавателя текста программы, содержащего описания типов данных и переменных, рекомендуется использовать распознаватель на основе грамматик операторного предшествования, рассмотренный в лабораторной работе №4. Основой функционирования данного класса распознавателей является матрица операторного предшествования.

Чтобы построить матрицу операторного предшествования для грамматики операторного предшествования необходимо выполнить следующие действия:

1. Построить множества крайних левых и крайних правых символов грамматики – **L**(U) и **R**(U).
2. На основе построенных множеств **L**(U) и **R**(U) построить множества крайних левых и крайних правых терминальных символов грамматики – $\mathbf{L}^t$(U) и $\mathbf{R}^t$(U).
3. Используя построенные множества $\mathbf{L}^t$(U) и $\mathbf{R}^t$(U) на основе правил исходной грамматики заполнить матрицу операторного предшествования.

Построение матрицы операторного предшествования и алгоритм функционирования распознавателя на основании грамматик операторного предшествования подробно рассмотрены в лабораторной работе №4.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя.
2. Выполнить предварительные преобразования, необходимые для построения матрицы предшествования.
3. Построить матрицу предшествования для заданной грамматики.
4. Выполнить разбор простейшего примера вручную по правилам заданной грамматики.
5. Подготовить и защитить отчет.
6. Написать и отладить программу на ЭВМ.
7. Сдать работающую программу преподавателю.

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать следующие разделы:

- Краткое изложение цели работы.
- Задание по лабораторной работе с описанием своего варианта.
- Запись заданной грамматики входного языка в форме Бэкуса-Наура.
- Множества крайних правых и крайних левых символов с указанием шагов построения.
- Множества крайних правых и крайних левых терминальных символов.
- Заполненную матрицу предшествования для грамматики.
- Пример выполнения разбора простейшего предложения (по выбору).
- Текст программы (оформляется только при необходимости по согласованию с преподавателем).
- Выводы по проделанной работе.

Допускается оформлять единый (совместный) отчёт для лабораторных работ №8 и №9.

Допускается (по соглашению с преподавателем) использовать для выполнения лабораторной работы иной класс синтаксических анализаторов (отличный от анализаторов на

основе грамматик операторного предшествования). В этом случае отчет по лабораторной работе должен включать в себя краткое описание выбранного класса синтаксических анализаторов, а вместо данных, необходимых для построения анализатора на основе грамматик операторного предшествования, он должен содержать данные, необходимые для построения анализатора из выбранного класса.

Также допускается (по соглашению с преподавателем) использовать автоматизированные средства построения синтаксических анализаторов (YACC и ему подобные). В этом случае отчет по лабораторной работе должен включать в себя описание выбранного средства автоматизированного построения синтаксических анализаторов, информацию о классе КС-грамматик, лежащих в основе данного средства, и вместо данных, необходимых для построения анализатора на основе грамматик операторного предшествования, он должен содержать данные, необходимые для автоматизированного построения анализатора.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какую роль выполняет синтаксический анализ в процессе компиляции?
2. Что такое распределение памяти? Какую роль оно играет в процессе компиляции?
3. Для каких элементов входной программы выполняется распределение памяти? Какие исходные данные требуются для распределения памяти?
4. Что такое КС-грамматики? Расскажите об их использовании в компиляторе.
5. Расскажите о грамматиках предшествования. Какие общие черты присущи всем грамматикам предшествования?
6. Расскажите о грамматиках операторного предшествования. Какие черты характеризуют грамматики операторного предшествования?
7. Дайте определения отношений предшествования для грамматик операторного предшествования.
8. Как вычисляются отношения предшествования для грамматик операторного предшествования?
9. Расскажите алгоритм построения множеств крайних левых и крайних правых символов.
10. Расскажите алгоритм построения множеств крайних левых и крайних правых терминальных символов.
11. Расскажите о задаче разбора. Что такое распознаватель языка?
12. Расскажите об общих принципах работы распознавателя языка.
13. Какие типы распознавателей существуют? Какие из них используются в компиляторе и на каких этапах?
14. Что такое МП-автомат? Расскажите о МП-автоматах.
15. Расскажите, как работает алгоритм «сдвиг-свертка» в общем случае (с возвратами).
16. Как работает алгоритм «сдвиг-свертка» для грамматик операторного предшествования? Поясните разбор предложения входного языка на своем примере.

ВАРИАНТЫ ИСХОДНЫХ ГРАММАТИК

Во всех вариантах используется одна и та же грамматика входного языка. Входная программа представляет собой последовательность двух блоков: первый блок – описание типов, начинающееся с ключевого слова **type**, второй блок – описание переменных, начинающееся с ключевого слова **var**. Описание типов и описание переменных выполняется в стиле языка Pascal.

Необходимо разобрать описания всех типов, рассчитать объем для каждого типа данных на основе известных алгоритмов и размеров скалярных типов данных, указанных в задании. Расчет надо выполнить в двух случаях: с учетом кратности распределения памяти и без него. Затем на основе рассчитанных размеров типов данных необходимо рассчитать объем памяти, занимаемой всеми описанными переменными (расчет также должен выполняться в двух вариантах: с учетом

кратности распределения памяти и без него). Результатом выполнения программы должны быть две величины: размер памяти, требуемой для всех описанных переменных, с учетом кратности распределения и тот же самый размер без учета кратности распределения.

Во всех вариантах символ S является начальным символом грамматики; L, T, R, V, K, D, F и E обозначают нетерминальные символы. Терминальные символы выделены жирным шрифтом. Терминальный символ **c** соответствует одному из двух скалярных типов, указанных в задании. Терминальный символ **t** соответствует одному из типов данных, который может быть описан в секции **type**, а терминальный символ **a** соответствует переменным, которые могут быть описаны в секции **var**. Терминальный символ **d** соответствует любой целочисленной константе. Грамматики в заданиях отличаются только правилами для терминальных символов D, F и E, которые описывают вариант допустимых типов данных – возможны три варианта: структура данных, союз (запись с вариантами) или массив.

Кроме расчета объема памяти программа, построенная на основе задания, должна выполнять синтаксическую проверку и элементарный семантический контроль входной программы – в том случае, если встречается тип данных, не описанный в секции **type**, должно выдаваться сообщение об ошибке.

Входная грамматика описаний типов для программы описывается следующими правилами:

1. S → **type** L **var** R
L → T**;** | T**;**L
T → **t=c** | **t=**D
R → V**;** | V**;**R
V → K**:t** | K**:c** | K**:**D
K → **a** | K**,a**
D → **record** F **end**
F → E**;** | E**;**F
E → K**:c** | K**:t**

2. S → **type** L **var** R
L → T**;** | T**;**L
T → **t=c** | **t=**D
R → V**;** | V**;**R
V → K**:t** | K**:c** | K**:**D
K → **a** | K**,a**
D → **union** F **end**
F → E**;** | E**;**F
E → K**:c** | K**:t**

3. S → **type** L **var** R
L → T**;** | T**;**L
T → **t=c** | **t=**D
R → V**;** | V**;**R
V → **t:**K | **c:**K | D**:**K
K → **a** | K**,a**
D → **struct (**F**)**
F → E**;** | E**;**F
E → **c:**K | **t:**K

4. S → **type** L **var** R
L → T**;** | T**;**L
T → **t=c** | **t=**D
R → V**;** | V**;**R
V → K**:t** | K**:c** | K**:**D
K → **a** | K**,a**
D → **array [**F**] of c** | **array [**F**] of t**
F → E | F**,**E
E → **d..d**

5. S → **type** L **var** R
L → T**;** | T**;**L
T → **t=c** | **t=**D
R → V**;** | V**;**R
V → **t:**K | **c:**K | D**:**K
K → **a** | K**,a**
D → **union (**F**;)**
F → E | F**,**E
E → **c:**K | **t:**K

6. S → **type** L **var** R
L → T**;** | T**;**L
T → **t=c** | **t=**D
R → V**;** | V**;**R
V → **t:**K | **c:**K | D**:**K
K → **a** | K**,a**
D → **array c (**F**)** | **array t (**F**)**
F → E | F**,**E
E → **d**

ВАРИАНТЫ ЗАДАНИЙ

Для выполнения лабораторной работы требуется написать программу, которая анализирует текст входной программы, содержащий описания типов данных и переменных. Результатом работы программы является проверка корректности исходного описания и подготовка внутренней структуры данных для вычисления объема памяти, необходимой для размещения переменных, определенных во входном тексте (результат работы программы будет далее использован в лабораторной работе №9).

Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, если структура входной программы не соответствует заданию. Если в исходном тексте встречаются типы данных или структуры данных, не предусмотренные заданием, программа должна сигнализировать об ошибке.

Длину идентификаторов и строковых констант можно считать ограниченной 32 символами. Программа должна допускать наличие комментариев неограниченной длины во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

Таблица 11.

Варианты заданий для лабораторной работы №8

№ варианта	№ варианта грамматики	Скалярные типы (размер в байтах)	Кратность распределения памяти	Кратность элементов структур
1	1	byte (1 байт), word (2 байта)	2	Да
2	2	byte (1 байт), word (2 байта)	2	Да
3	3	byte (1 байт), word (2 байта)	2	Да
4	4	char (1 байт), integer (4 байта)	2	Да
5	5	char (1 байт), integer (4 байта)	2	Да
6	6	char (1 байт), integer (4 байта)	2	Да
7	1	byte (1 байт), real (6 байт)	4	Нет
8	2	byte (1 байт), real (6 байт)	4	Нет
9	3	byte (1 байт), real (6 байт)	4	Нет
10	4	char (1 байт), double (8 байт)	4	Нет
11	5	char (1 байт), double (8 байт)	4	Нет
12	6	char (1 байт), double (8 байт)	4	Нет
13	1	byte (1 байт), extended (10 байт)	8	Да
14	2	byte (1 байт), extended (10 байт)	8	Да
15	3	byte (1 байт), extended (10 байт)	8	Да
16	4	integer (4 байта), real (6 байт)	8	Да
17	5	integer (4 байта), real (6 байт)	8	Да
18	6	integer (4 байта), real (6 байт)	8	Да
19	1	byte (1 байт), real (6 байт)	2	Нет
20	2	byte (1 байт), real (6 байт)	2	Нет
21	3	byte (1 байт), real (6 байт)	2	Нет
22	4	char (1 байт), double (8 байт)	2	Нет
23	5	char (1 байт), double (8 байт)	2	Нет
24	6	char (1 байт), double (8 байт)	2	Нет

ЛАБОРАТОРНАЯ РАБОТА № 9
ОПРЕДЕЛЕНИЯ ОБЪЕМА ПАМЯТИ ДЛЯ СТРУКТУР ДАННЫХ

Цель работы: изучение основных принципов распределения памяти, ознакомление с алгоритмами расчета объема памяти, занимаемой простыми и составными структурами данных, получение практических навыков создания простейшего анализатора для расчета объема памяти, занимаемого заданной структурой данных.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Принципы распределения памяти

Распределение памяти — это процесс, который ставит в соответствие лексическим единицам исходной программы адрес, размер и атрибуты области памяти, необходимой для этой лексической единицы. *Область памяти* — это блок ячеек памяти, выделяемый для данных, каким-

то образом объединенных логически. Логика таких объединений задается семантикой исходного языка.

Распределение памяти работает с лексическими единицами языка — переменными, константами, функциями и т. п. — и с информацией об этих единицах, полученной на этапах лексического и синтаксического анализа. Как правило, исходными данными для процесса распределения памяти в компиляторе служат таблица идентификаторов, построенная лексическим анализатором, и декларативная часть программы («область описаний»), полученная в результате синтаксического анализа. Не во всех языках программирования декларативная часть программы присутствует явно, некоторые языки предусматривают дополнительные семантические правила для описания констант и переменных; кроме того, перед распределением памяти надо выполнить идентификацию лексических единиц языка. Поэтому распределение памяти выполняется уже после семантического анализа текста исходной программы.

Процесс распределения памяти в современных компиляторах, как правило, работает с относительными, а не абсолютными адресами ячеек памяти. Распределение памяти выполняется перед генерацией кода результирующей программы, потому что его результаты должны быть использованы в процессе генерации кода.

Каждую область памяти можно классифицировать по двум параметрам: в зависимости от ее роли в результирующей программе и в зависимости от способа ее распределения в ходе выполнения результирующей программы.

По роли области памяти в результирующей программе она бывает *глобальная* или *локальная*.

По способам распределения область памяти бывает *статическая* или *динамическая*. Динамическая память, в свою очередь, может распределяться либо разработчиком исходной программы (по командам разработчика), либо компилятором (автоматически).

Классификация областей памяти представлена на рис. 9.

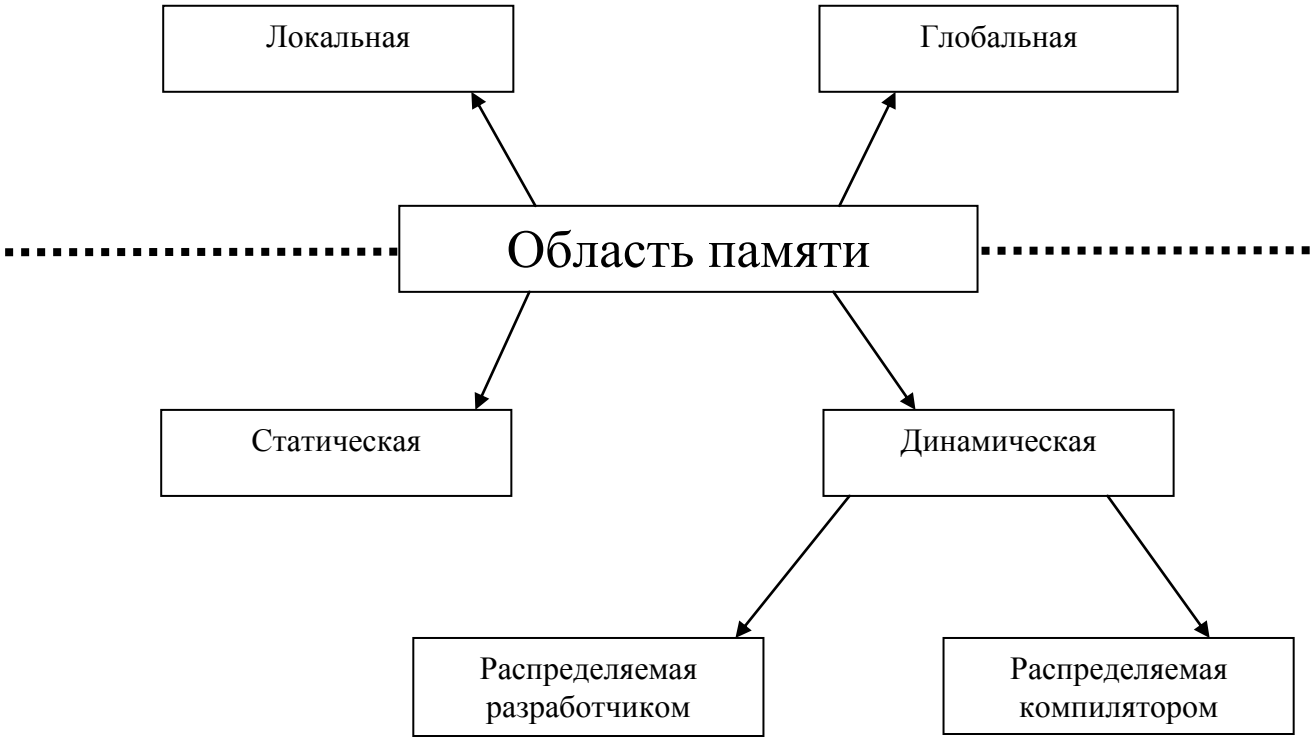


Рис. 9. Область памяти в зависимости от ее роли и способа распределения

Далеко не все лексические единицы языка требуют для себя выделения памяти. То, под какие элементы языка нужно выделять области памяти, а под какие нет, определяется исключительно реализацией компилятора и архитектурой целевой вычислительной системы. Так, целочисленные константы можно разместить в статической памяти, а можно непосредственно в тексте результирующей программы (это позволяют практически все современные вычислительные системы), то же самое относится и к константам с плавающей точкой, но их размещение в тексте программы допустимо не всегда. Кроме того, в целях экономии памяти, занимаемой результирующей программой, под различные элементы языка компилятор может выделить одни и те же ячейки памяти. Например, в одной и той же области памяти могут быть размещены одинаковые строковые константы или две различные локальные переменные, которые никогда не используются одновременно.

Виды переменных и областей памяти

Распределение памяти для переменных скалярных типов

Во всех языках программирования существует понятие так называемых «базовых типов данных», которые также называют основными или *скалярными* типами. Размер области памяти, необходимый для лексической единицы скалярного типа, считается заранее известным. Он определяется семантикой языка и архитектурой целевой вычислительной системы.

Например, в языке программирования C базовыми типами данных являются типы `char`, `int`, `long int` и т. п. (реально этих типов, конечно, больше, чем три), а в языке программирования Pascal — типы `byte`, `char`, `word`, `integer` и т. п.. Размер базового типа `int` в языке C для архитектуры компьютера на базе 16-разрядных процессоров составляет 2 байта, а для 32-разрядных процессоров — 4 байта. Разработчики исходной программы на этом языке, конечно, могут узнать данную информацию, но если ее использовать в исходной программе напрямую, то такая программа будет жестко привязана к конкретной архитектуре целевой вычислительной системы. Чтобы исключить эту зависимость, лучше использовать механизм определения размера памяти для типа данных, предоставляемый языком программирования — в языке C это функция `sizeof`.

Распределение памяти для сложных структур данных

Для более сложных структур данных используются правила распределения памяти, определяемые семантикой (смыслом) этих структур. Эти правила достаточно просты и, в принципе, одинаковы во всех языках программирования.

Вот правила распределения памяти под основные виды структур данных:

- для массивов — произведение числа элементов в массиве на размер памяти для одного элемента (то же правило применимо и для строк, но во многих языках строки содержат еще и дополнительную служебную информацию фиксированного объема);
- для структур (записей с именованными полями) — сумма размеров памяти по всем полям структуры;
- для объединений (союзов, общих областей, записей с вариантами) — размер максимального поля в объединении;
- для реализации объектов (классов) — размер памяти для структуры с такими же именованными полями плюс память под служебную информацию объектно-ориентированного языка (как правило, фиксированного объема).

Формулы для вычисления объема памяти можно записать следующим образом:

для массивов: $V_{мас} = \prod_{i=1,n} (m_i) * V_{эл}$,

где n — размерность массива, m_i — количество элементов i-ой размерности, $V_{эл}$ — объем памяти для одного элемента;

для структур: $V_{стр} = \mathbf{\Sigma}_{i=1,n} V_{поля_i}$,

где n — общее количество полей в структуре, $V_{поля_i}$ — объем памяти для i-го поля структуры;

для объединений (союзов): $V_{союз} = \mathbf{max}_{i=1,n} V_{поля_i}$,

где n — общее количество полей в объединении, $V_{поля_i}$ — объем памяти для i-го поля объединения.

Для более сложных структур данных входного языка объем памяти, отводимой под эти структуры данных, вычисляется рекурсивно. Например, если имеется массив структур, то при вычислении объема отводимой под этот массив памяти для вычисления объема памяти, необходимой для одного элемента массива, будет вызвана процедура вычисления памяти структуры. Такой подход определения объема занимаемой памяти очень удобен, если декларативная часть языка представлена в виде дерева типов. Тогда для вычисления объема памяти, занимаемой типом из каждой вершины дерева, нужно вычислить объем памяти для всех потомков этой вершины, а потом применить формулу, связанную непосредственно с самой вершиной (этот механизм подобен механизму СУ-перевода, применяемому при генерации кода). Как раз такого типа древовидные конструкции строит синтаксический анализ для декларативной части языка.

Например, если рассмотреть фрагмент текста входной программы на языке Pascal:

```
...
type
  arr1 = array[1..10,1..20] of integer;

  rec1 = record
    v1: word;
    a1: arr1;
  end;

  rec2 = record
    v1: byte;
    case (v2: integer) of
    0: v3: byte;
    1: v4: word;
    3: v5: integer;
  end;

  arr2 = array[1..100] of rec2;

var
  aa1: arr1;
  vv1,vv2,vv3,vv4: byte;
  rr1: rec1;
  aa2arr2;
...
```

Тогда, если известно, что размер скалярных типов данных `byte`, `word` и `integer` составляет для целевой вычислительной системы 1, 2 и 4 байта соответственно, то можно посчитать, что размер типа `arr1` будет составлять V_1 = 10*20*4 = 800 байт, размер типа `rec1` составит V_2 = 2+800 = 802 байта, размер типа `rec2` составит V_3 = 1+4+Max(1,2,4) = 9 байт, а размер типа `arr2` составит V_4 = 100*9 = 900 байт. Тогда для размещения переменной `aa1` потребуется 800 байт памяти, для размещения переменных `vv1,vv2,vv3,vv4` потребуется 4 байта памяти, для переменной `rr1` – 802 байта, а для переменной `aa2` – еще 900 байт. Таким образом, для размещения в памяти всей описанной структуры данных потребуется выделить V = 2506 байт памяти.

Выравнивание границ областей памяти

Говоря об объеме памяти, занимаемой различными лексическими единицами и структурами данных языка, следует упомянуть еще один момент – выравнивание границ областей памяти, отводимых для различных лексических единиц.

Архитектура многих современных вычислительных систем предусматривает, что обработка данных выполняется более эффективно, если адрес, по которому выбираются данные, кратен определенному числу байт (как правило, это 2, 4, 8 или 16 байт). Современные компиляторы учитывают особенности целевых вычислительных систем. При распределении данных они могут размещать области памяти под лексические единицы наиболее оптимальным образом. Поскольку не всегда размер памяти, отводимой под лексическую единицу, кратен указанному числу байт, то в общем объеме памяти, отводимой под результирующую программу, могут появляться неиспользуемые области.

Например, если мы имеем описание переменных на языке C:

```
static char c1, c2, c3;
```

то, зная, что под одну переменную типа `char` отводится 1 байт памяти, можем ожидать, что для описанных выше переменных потребуется всего 3 байта памяти. Однако если кратность адресов для доступа к памяти установлена 4 байта, то под эти переменные будет отведено в целом 12 байт памяти, из которых 9 не будут использоваться.

Выравнивание границ областей памяти возможно как по границам структур данных (и представляющих их лексем), так и внутри самих структур данных для составляющих их элементов. Например, если мы имеем описание массивов на языке Pascal:

a1,a2: `array`[1..3] of char;

то при условии, что под одну переменную типа `char` отводится 1 байт памяти и кратность адресов установлена 4 байта, в случае, когда выравнивание границ областей памяти выполняется только по границам самих областей, под описанные выше переменные потребуется 8 байт памяти (по 4 байта на каждый массив). Если же выравнивание границ областей памяти выполняется и для составляющих элементов (в данном случае – элементов массивов), то под эти же переменные потребуется 24 байта памяти (по 4 байта на один элемент массива – и в целом 12 байт на каждый массив).

Как правило, разработчику исходной программы не нужно знать, каким образом компилятор распределяет адреса под отводимые области памяти. Чаще всего компилятор сам выбирает оптимальный метод, и, изменяя границы выделенных областей, он всегда корректно осуществляет доступ к ним. Вопрос об этом может встать, если с данными программы, написанной на одном языке, работают программы, написанные на другом языке программирования (чаще всего, на языке ассемблера), реже такие проблемы возникают при использовании двух различных компиляторов с одного и того же входного языка. Большинство компиляторов позволяют пользователю в этом случае самому указать, использовать или нет кратные адреса и какую границу кратности установить (если это вообще возможно с точки зрения архитектуры целевой вычислительной системы).

Если вернуться к рассмотренному выше примеру фрагмента программы на языке Pascal, то, если предположить, что кратность размещения данных в целевой вычислительной системе составляет 4 байта и выравнивание границ памяти выполняется только по границам структур, можно рассчитать новые значения объема памяти, требуемой для размещения описанных в этом фрагменте переменных.

Размер типа `arr1` будет составлять V_1 = 10*20*4 = 800 байт. Размер типа `rec1` составит V_2 = 2+800 = 802 байта, но с учетом кратности (4 байта) для его размещения потребуется V_2' = 4+800 = 804 байта. Размер типа `rec2` V_3 = 1+4+Max(1,2,4) = 9 байт с учетом кратности составит V_3' = 12 байт, а размер типа `arr2` с учетом кратности составит V_4' = 100*12 = 1200 байт. Тогда для

размещения переменной `aa1` потребуется 800 байт памяти, для размещения переменных `vv1,vv2,vv3,vv4` с учетом кратности потребуется 4*4 = 16 байт памяти, для переменной `rr1` – 804 байта, а для переменной `aa2` – еще 1200 байт. Таким образом, для размещения в памяти всей описанной структуры данных потребуется выделить $V' = 2820$ байт памяти.

При тех же условиях, но с выравниванием границ памяти по элементам структур, необходимые объемы памяти в данном случае не изменятся. Размер типа `arr1` будет составлять $V_1 = 10*20*4 = 800$ байт. Размер типа `rec1` с учетом кратности составит $V_2'' = 4+800 = 804$ байта. Размер типа `rec2` с учетом кратности составит $V_3' = 4+4+\mathrm{Max}(4,4,4) = 12$ байт, а размер типа `arr2` с учетом кратности составит $V_4' = 100*12 = 1200$ байт.

Видно, что объем требуемой памяти в обоих случаях увеличился примерно на 12.5%, но за счет этого увеличивается эффективность обращения к данным и, следовательно, скорость выполнения результирующей программы.

Виды областей памяти. Статическое и динамическое связывание

Глобальная и локальная память

Глобальная область памяти — это область памяти, которая выделяется один раз при инициализации результирующей программы и действует все время выполнения результирующей программы.

Как правило, глобальная область памяти может быть доступна из любой части исходной программы, но многие языки программирования позволяют налагать синтаксические и семантические ограничения на доступность даже для глобальных областей памяти. При этом сами области памяти и связанные с ними лексические единицы остаются глобальными, ограничения налагаются только на возможность их использования в тексте исходной программы (и эти ограничения, в принципе, возможно обойти).

Локальная область памяти — это область памяти, которая выделяется в начале выполнения некоторого фрагмента результирующей программы (блока, функции, процедуры или оператора) и может быть освобождена по завершении выполнения данного фрагмента.

Доступ к локальной области памяти всегда запрещен за пределами того фрагмента программы, в котором она выделяется. Это определяется как синтаксическими и семантическими правилами языка, так и кодом результирующей программы. Даже если удастся обойти ограничения, налагаемые входным языком, использование таких областей памяти вне их области видимости приведет к катастрофическим последствиям для результирующей программы.

Распределение памяти на локальные и глобальные области целиком определяется семантикой языка исходной программы. Только зная смысл синтаксических конструкций исходного языка можно четко сказать, какая из них будет отнесена в глобальную область памяти, а какая — в локальную.

Рассмотрим для примера фрагмент текста модуля программы на языке Pascal:

```
...
const
    Global_1 = 1;
    Global_2 : integer = 2;
var
    Global_I : integer;
...
function Test (Param: integer): pointer;
const
    Local_1 = 1;
    Local_2 : integer = 2;
var
    Local_I : integer;
begin
  ...
end;
```

Согласно семантике языка Pascal, переменная `Global_I` является глобальной переменной языка и размещается в глобальной области памяти, константа `Global_1` также является глобальной, но язык не требует, чтобы компилятор обязательно размещал ее в памяти — значение константы может быть непосредственно подставлено в код результирующей программы там, где она используется. Типизированная константа `Global_2` является глобальной, но в отличие от константы `Global_1`, семантика языка предполагает, что эта константа обязательно будет размещена в памяти, а не в коде программы. Доступность идентификаторов `Global_I`, `Global_2` и `Global_1` из других модулей зависит от того, где и как они описаны. Например, для компилятора Borland Pascal переменные и константы, описанные в заголовке модулей, доступны из других модулей программы, а переменные и константы, описанные в теле модулей, недоступны, хотя и те и другие являются глобальными.

Параметр `Param` функции `Test`, переменная `Local_I` и константа `Local_1` являются локальными элементами этой функции. Они не доступны вне пределов данной функции и располагаются в локальной области памяти. Типизированная константа `Local_2` представляет собой очень интересный элемент программы. С одной стороны, она не доступна вне пределов функции `Test` и потому является ее локальным элементом. С другой стороны, как типизированная константа она будет располагаться в глобальной области памяти, хотя ниоткуда извне не может быть доступна (аналогичными конструкциями языка C, например, являются локальные переменные функций, описанные как `static`).

Семантические особенности языка должны учитывать создатели компилятора, когда разрабатывают модуль распределения памяти. Безусловно, это должен знать и разработчик исходной программы, чтобы не допускать семантических (смысловых) ошибок — этот тип ошибок сложнее всего поддается обнаружению.

Статическая и динамическая память

Статическая область памяти — это область памяти, размер которой известен на этапе компиляции.

Поскольку для статической области памяти известен ее размер, компилятор всегда может выделить эту область памяти и связать ее с соответствующим элементом программы. Поэтому для статической области памяти компилятор порождает некоторый адрес (как правило, это относительный адрес в программе). Статические области памяти обрабатываются компилятором

самым простейшим образом, поскольку напрямую связаны со своим адресом. В этом случае говорят о *статическом связывании* области памяти и лексической единицы входного языка.

Динамическая область памяти — это область памяти, размер которой на этапе компиляции программы не известен.

Размер динамической области памяти будет известен только в процессе выполнения результирующей программы. Поэтому для динамической области памяти компилятор не может непосредственно выделить адрес — для нее он порождает фрагмент кода, который отвечает за распределение памяти (ее выделение и освобождение). Как правило, с динамическими областями памяти связаны многие операции с указателями и с экземплярами объектов (классов) в объектно-ориентированных языках программирования. При использовании динамических областей памяти говорят о *динамическом связывании* области памяти и лексической единицы входного языка.

Динамические области памяти, в свою очередь, можно разделить на динамические области памяти, выделяемые пользователем, и динамические области памяти, выделяемые непосредственно компилятором.

Динамические области памяти, выделяемые пользователем, появляются в тех случаях, когда разработчик исходной программы явно использует в тексте программы функции, связанные с распределением памяти (примером таких функций являются `New` и `Dispose` в языке Pascal, `malloc` и `free` в языке C, `new` и `delete` в языке C++ и другие). Функции распределения памяти могут использовать либо напрямую средства ОС, либо средства исходного языка (которые, в конце концов, все равно основаны на средствах ОС). В этом случае за своевременное выделение и освобождение памяти отвечает сам разработчик, а не компилятор. Компилятор должен только построить код вызова соответствующих функций и сохранения результата — в принципе, для него работа с динамической памятью пользователя ничем не отличается от работы с любыми другими функциями и дополнительных сложностей не вызывает.

Другое дело — динамические области памяти, выделяемые компилятором. Они появляются тогда, когда пользователь использует типы данных, операции над которыми предполагают перераспределение памяти, не присутствующее в явном виде в тексте исходной программы. Примерами таких типов данных могут служить строки в некоторых языках программирования, динамические массивы и, конечно, многие операции над экземплярами объектов (классов) в объектно-ориентированных языках (наиболее характерный пример — тип данных `string` в Borland Delphi). В этом случае сам компилятор отвечает за порождение кода, который будет всегда обеспечивать своевременное выделение памяти под элементы программы и за освобождение ее по мере использования.

Как статические, так и динамические области памяти сами по себе могут быть глобальными или локальными.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя.
2. Изучить алгоритм расчета требуемого объема памяти для структур данных входной программы, соответствующих варианту задания.
3. Определить, как заданная кратность распределения памяти будет влиять на выделяемый объем памяти.
4. Реализовать алгоритм расчета объема требуемой памяти на основе результатов синтаксического анализа, полученных в ходе выполнения лабораторной работы №8.
5. Выполнить вручную расчет требуемого объема памяти для выбранного простейшего примера. Проверить корректность результата.
6. Подготовить и защитить отчет.
7. Написать и отладить программу на ЭВМ.
8. Сдать работающую программу преподавателю.

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать следующие разделы:

- Задание по лабораторной работе.
- Краткое изложение цели работы.
- Запись заданной грамматики входного языка в форме Бэкуса-Наура.
- Описание алгоритма расчета объема памяти для структуры данных, указанной в задании.
- Пример выполнения расчета объема памяти для простейшей входной программы (по выбору).
- Текст программы (оформляется только при необходимости по согласованию с преподавателем).
- Выводы по проделанной работе

Допускается оформлять единый (совместный) отчёт для лабораторных работ №8 и №9.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Расскажите об общей структуре компилятора.
2. Что такое подготовка к генерации объектного кода? Когда она выполняется?
3. Что такое распределение памяти? Какую роль оно играет в процессе компиляции?
4. Для каких элементов входной программы выполняется распределение памяти?
5. Что такое относительные адреса? В какой момент и как относительные адреса преобразуются в адреса виртуальные и физические?
6. Какие типы областей памяти бывают в зависимости от роли и способа распределения? Как они могут сочетаться между собой?
7. Что такое локальная и глобальная память?
8. Как распределяется память для аргументов и локальных переменных процедур и функций?
9. Что такое динамическая и статическая память? Какие особенности можно указать для динамической памяти, распределяемой компилятором?
10. Приведите примеры локальных и глобальных, статических и динамических элементов памяти на основе известного Вам языка программирования.
11. Что такое скалярные типы данных? От чего зависит объем памяти, выделяемой для скалярных типов данных?
12. Как рассчитывается объем памяти для сложных структур данных?
13. Что такое «кратность распределения памяти»? Для чего она используется и как влияет на объем выделяемой памяти?
14. Как влияет кратность распределения памяти на объем памяти, требуемой для различных структур данных? Приведите примеры.
15. Что такое дисплей памяти процедуры или функции?
16. Расскажите о стековой организации дисплея памяти.

ВАРИАНТЫ ЗАДАНИЙ

Для выполнения лабораторной работы требуется написать программу, которая анализирует текст входной программы, содержащий описания типов данных и переменных, и рассчитывает объем памяти, требуемый для размещения всех переменных, описанных во входной программе. Для анализа исходной программы рекомендуется использовать результаты выполнения лабораторной работы №8.

Все переменные в исходной программе считаются статическими глобальными переменными. Результатом работы программы является значение требуемого объема памяти в байтах с учетом фрагментации памяти и без учета ее (объем памяти для скалярных типов данных и коэффициент фрагментации даются в задании). Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, если структура входной программы не соответствует заданию.

Длину идентификаторов и строковых констант считать ограниченной 32 символами. Программа должна допускать наличие комментариев неограниченной длины во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

Варианты заданий полностью соответствуют вариантам заданий для лабораторной работы №8.

ЛАБОРАТОРНАЯ РАБОТА № 10
РАЗРАБОТКА ПРОСТЕЙШЕГО ПРИЛОЖЕНИЯ НА ОСНОВЕ ТЕХНОЛОГИИ «КЛИЕНТ-СЕРВЕР»

Цель работы: Изучить основные принципы взаимодействия приложений, разработанных в архитектуре «клиент-сервер», реализовать простейшее клиентское приложение, осуществляющее доступ к базе данных по технологии ODBC (или другой технологии взаимодействия с базами данных), изучить основные принципы работы клиентского приложения с API ODBC и с другими технологиями доступа к базам данных.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Методы организации распределенных вычислений

Распространение динамически загружаемых библиотек и ресурсов пользовательского интерфейса программ привело к ситуации, когда прикладные программы стали представлять собой не единый программный модуль, а набор взаимосвязанных компонентов. Причем не все компоненты создавались теми же разработчиками, что и сама прикладная программа. Некоторые из них входили в состав ОС, другие поставлялись сторонними разработчиками, которые очень часто могли быть никак не связаны с разработчиками прикладной программы.

При этом любую прикладную программу (приложение) можно условно разделить на четыре основных уровня обработки данных:

- пользовательский интерфейс, отвечающий за отображение данных, представление их пользователю и взаимодействие с ним;
- бизнес-логики, реализующий специализированную логику обработки и преобразования данных, характерную для данной прикладной программы;
- БД, отвечающий за типовые операции получения, хранения, защиты и архивации данных, разделение доступа к ним;
- файловых операций, обеспечивающий работу прикладной программы с файловой системой ОС.

Каждый из этих уровней представляет собой логически цельную составляющую единой прикладной программы. При распределенной обработке данных каждая составляющая может выполняться отдельно (при условии сохранения взаимосвязи между всеми составляющими). В принципе, каждый из этих уровней можно далее разбить на подуровни и получить больше составляющих, общее число которых ничем принципиально не ограничено. Но с другой стороны, все составляющие любой прикладной программы можно разделить всего на две крупные части.

Первая из них обеспечивает нижний уровень работы приложения, отвечает за методы хранения, резервирования, доступа и разделения данных. Вторая организует верхний уровень работы приложения, включает логику обработки данных и интерфейс пользователя. Первая часть обеспечивает два нижних уровня обработки данных. Она, как правило, представляет собой набор компонент, входящих в состав ОС, либо созданных крупными сторонними фирмами-разработчиками, никак не связанными с созданием прикладной программы. Вторая часть обеспечивает реализацию двух верхних уровней обработки данных. Она включает в себя алгоритмы, логику и интерфейс пользователя, созданные разработчиками прикладной программы.

Таким образом, сложилось понятие приложения, построенного на основе архитектуры клиент-сервер. Первая часть в этой архитектуре стала носить название сервер данных, а вторая — клиент или клиентское приложение. При этом первая (серверная) часть приложения обеспечивает обработку данных на уровне файлов и БД. Для работы ее компонентов зачастую требуется наличие высокопроизводительной вычислительной системы. Вторая (клиентская) часть приложения, получая данные от сервера данных, обеспечивает их обработку и отображение в интерфейсе пользователя. По командам клиентской части сервер данных выполняет их добавление, обновление и удаление. Требования к вычислительным ресурсам, необходимым для выполнения компонент второй части, обычно существенно ниже, чем для первой.

На начальном этапе развития архитектуры клиент-сервер доступ к данным сервера осуществлялся на уровне файлов, а разделение доступа к файлам обеспечивалось средствами ОС. Сервер данных выполнял только примитивные процедуры хранения, копирования и защиты файлов от несанкционированного доступа. Такие приложения иногда называют приложениями, построенными по принципу "файл-сервер". В общем виде схема их работы представлена на рис. 10.

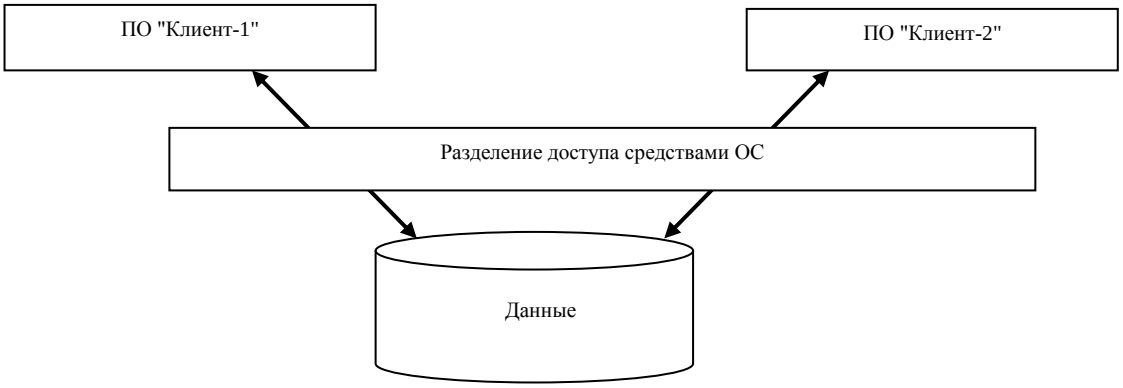


Рис. 10. Схема доступа к данным для приложений типа "файл-сервер"

Видно, что в этом случае серверная часть выполняет только один, самый нижний уровень обработки данных, остальные уровни реализуются на клиентской части. Поэтому приложения типа "файл-сервер" можно только весьма условно отнести к приложениям, построенным в архитектуре "клиент-сервер". Хотя в них присутствует серверная компонента, ее функции, в основном, выполняются в ОС.

Главными недостатками приложений, построенных по принципу "файл-сервер", являются:

- высокая нагрузка на клиентскую часть, и как следствие, высокие требования к вычислительным ресурсам клиентской части;
- невозможность эффективно разделить доступ к данным при их одновременном использовании несколькими пользователями;
- невозможность организовать защиту данных иначе, как на уровне доступа ОС;
- высокая нагрузка на сеть для передачи файлов, если сервер данных и клиентское приложение работают на разных компьютерах в составе сети.

Как правило, разработчики, создающие ПО в архитектуре клиент-сервер, разрабатывают именно клиентскую часть. Но тогда с одним сервером данных работают несколько различных клиентских приложений (иначе нет никакого смысла создавать ПО в данной архитектуре).

Проще всего полноценное приложение в архитектуре клиент-сервер возможно получить, если использовать БД для хранения данных и СУБД для доступа к данным. Схема работы такого приложения представлена на рис. 11.

Здесь серверная компонента реализует два уровня обработки данных – файловые операции и работу с БД. Все функции по хранению данных, доступу к ним, защите и резервному

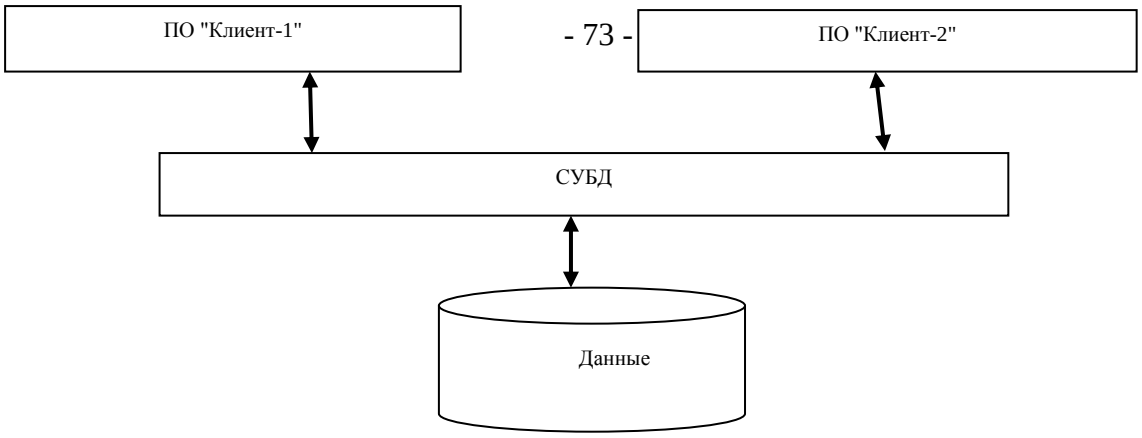


Рис. 11. Схема доступа к данным для приложений в архитектуре клиент-сервер

копированию данных в такой схеме реализует СУБД на сервере. Такой подход освобождает клиентские рабочие места от реализации этих функций и снижает требования к ним. Клиентское приложение не работает непосредственно с файлами и данными, оно обращается к СУБД с запросами на получение (просмотр) данных, их добавление, модификацию и удаление. При работе сервера данных и клиентского приложения на разных компьютерах в составе сети через сеть будут передаваться не все данные из БД, а только запросы клиента и ответы СУБД на них. Таким образом, в архитектуре клиент-сервер снижается нагрузка на сеть при обмене данными.

Общение с сервером СУБД происходит на языке структурированных запросов SQL (Structured Query Language). Базовый набор языка стандартизован ANSI. Самая распространенная редакция ANSI SQL92. SQL предназначен для построения запросов и манипуляции данными и структурами данных. У него нет ни переменных, ни меток, ни циклов, ни всего прочего, с чем привык работать нормальный программист. Надо четко представлять себе, что SQL оговаривает способ передачи данных в клиентскую программу, но никак не оговаривает то, как эти данные должны в клиентской программе обрабатываться и представляться пользователю.

Естественно, что базовый стандарт не может предусмотреть все потребности пользователей, поэтому многие фирмы производители СУБД предлагают свои собственные и часто непереносимые расширения SQL. Например, Oracle и IBM имеют собственные расширения оператора SELECT, которое позволяет эффективно разворачивать в горизонтальное дерево иерархически упорядоченные данные. Количество расширений может исчисляться десятками для сервера СУБД от одной фирмы.

Технология ODBC

Одним из наиболее распространенных средств, позволяющих унифицировать организацию взаимодействия с различными СУБД, является интерфейс ODBC. ODBC - Open Database Connectivity это интерфейс доступа к базам данных в среде Windows. Доступ к БД осуществляется при помощи специального ODBC драйвера, который транслирует запросы к БД на язык, поддерживаемый конкретной СУБД.

Для установления соединения с БД технология ODBC использует ODBC драйверы и источники данных, которые позволяют настроиться на сеанс конкретного пользователя СУБД. Для этого источник содержит имя пользователя и его пароль, а также, при необходимости, другую информацию, требуемую для присоединения к СУБД. ODBC драйвер представляет собой динамически загружаемую библиотеку, которая может использоваться приложением для получения доступа к конкретному серверу данных. Каждому типу СУБД соответствует свой ODBC драйвер.

Взаимодействие клиентской части ПО с СУБД осуществляется при помощи набора системных вызовов, которые выполняются по отношению к источнику данных. Источник данных взаимодействует с драйвером ODBC, транслирует ему запросы клиента и получает ответы, а тот в

свою очередь обращается к СУБД. При необходимости работать с типом СУБД достаточно указать другой тип ODBC драйвера в источнике данных, при этом не требуется изменять клиентскую часть ПО.

Такая двухступенчатая схема взаимодействия клиента с сервером данных обеспечивает независимость клиентской части от типа СУБД на сервере данных, но в целом снижает эффективность работы приложения, поскольку запрос, посланный клиентом, дважды передается различным библиотекам функций прежде, чем попадет на сервер. Решение вопроса о выборе способа взаимодействия с сервером данных остается за разработчиком клиентской части ПО и зависит от предъявляемых к нему требований.

ODBC реализует интерфейс доступа к разным SQL совместимым базам данных.

Клиент
|
ODBC
|
Диспетчер драйверов ODBC
|
драйвер БД
|
БД

Идея заключается в том, что приложение может получать доступ к совершенно разным базам данных, не меняя при этом код. Схема взаимодействия приложения с элементами ODBC представлена на рис. 12.

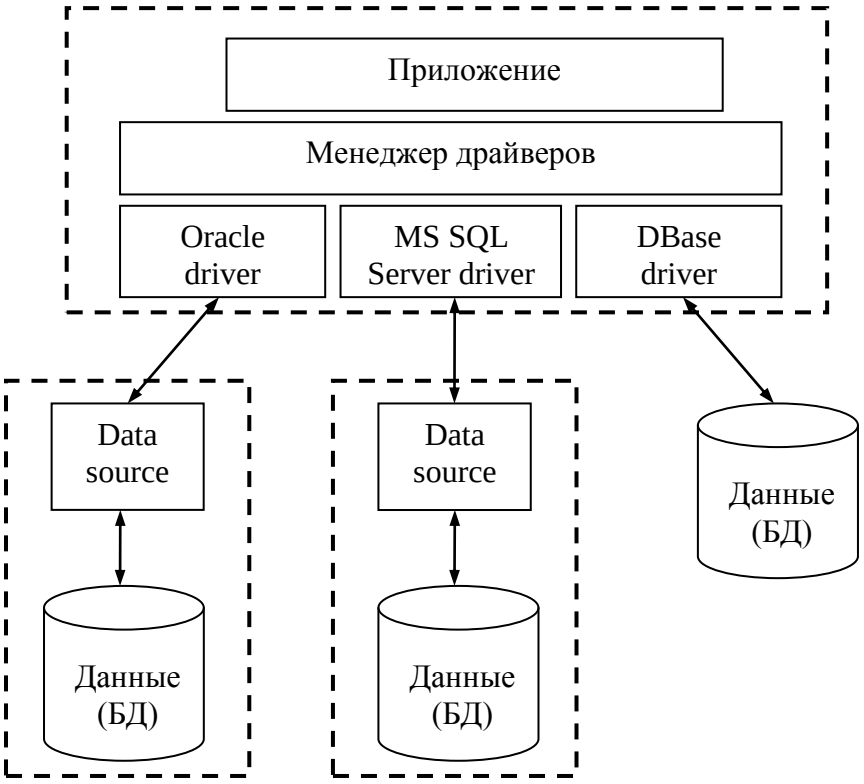


Рис. 12. Взаимодействие ODBC с различными источниками

Архитектура, по которой строится ODBC, легко наращиваемая. Для добавления нового типа БД нужно лишь написать драйвер и зарегистрировать его.

Основные преимущества ODBC:

- API функции одинаковые и не зависят от поставщика. (API - Application Programming Interface, интерфейс прикладной программы, посредством которого приложение получает доступ к операционной системе и другим сервисам. Использование API позволяет одинаковым образом осуществлять обработку файлов, вывод на принтер, передачу сообщений и выполнение других операций).
- SQL операторы могут быть сгенерированы на любой стадии при компиляции или выполнении.
- Данные принимаются в программу в едином формате.

Физически ODBC представляет собой набор динамических библиотек (DLL – Dynamic Link Library), которые обслуживают подключение и работу с конкретным типом базы данных. При запросе на подключение к определенной, заранее описанной базе "активизируется" определенная DLL - драйвер этого типа БД. Обращение к определенной базе данных происходит по имени так называемого источника данных ODBC или DSN.

DSN - Data Source Name - именованный источник данных ODBC. Источник данных содержит данные и сведения о подключении, необходимые для доступа к данным. Примерами источников данных могут служить базы данных Microsoft Access, Microsoft SQL Server, электронная таблица и текстовый файл. К примерам сведений о подключении относятся: папка на сервере, имя базы данных, сетевое имя, пароль, а также различные параметры драйвера ODBC, описывающие способ подключения к источнику данных.

Диспетчер использует информацию, связанную с именем для доступа к БД. С именем связана следующая информация:

- Местонахождение;
- Тип драйвера;
- Другие обязательные параметры.

Можно представить DSN как своего рода объявление БД на данном компьютере.

Существует три типа имен DSN:

- Пользовательский;
- Системный;
- Файловый.

В первом случае информация хранится в реестре Windows и привязана к конкретному пользователю (т.е. находится в области видимости только одного пользователя HKEY_CURRENT_USER\Software\ODBC\ODBC.INI). Во втором случае к конкретному компьютеру и каждый пользователь имеет доступ к данному источнику (HKEY_LOCAL_MACHINE\Software\ODBC\ODBC.INI). В последнем случае информация хранится в файле, что облегчает перенос проекта с компьютера на компьютер. Это просто текстовый файл с расширением *.dsn, обычно в папке C:\Program Files\Common Files\ODBC\Data Sources.

Первые два типа источников данных особенно полезны, когда требуется обеспечить дополнительную защиту, поскольку, таким образом, гарантируется, что источник данных может просматриваться только зарегистрированными пользователями и не может быть скопирован удаленным пользователем на другой компьютер.

Управление источниками данных ODBC (да и вообще настройкой всей системы ODBC) осуществляется с помощью специальной программы - ODBC-администратора. В Windows 9x - это исполняемый файл odbcad32.exe, который лежит в каталоге Windows\System. Запускать его можно напрямую, либо через Панель управления (значок "Источники данных ODBC (32-бит)"). В Windows 2000, XP - исполняемый файл odbcad32.exe лежит в каталоге WinNT\System32, а запускать его можно через Панель управления -> Администрирование -> Источники данных ODBC.

Технологии OLE DB и ADO

Технологии OLE DB и ADO были созданы компанией Microsoft и получили распространение в среде ОС типа Windows. Основой для их создания, в свою очередь, послужила технология OLE (Object Linking and Embedding — связывание и внедрение объектов), которая первоначально создавалась и развивалась в ОС типа Windows для обеспечения взаимосвязи между прикладными программами (в первую очередь — для связи между собой приложений Microsoft Office). После того, как в основу OLE была положена принципиально новая технология COM (Component Object Model — Модель многокомпонентных объектов), стало возможным использовать OLE для связи любых типов программ.

В результате применения технологии OLE к решению задачи организации взаимодействия клиентской и серверной частей приложения была получена новая технология — OLE DB (OLE for DataBases — OLE для баз данных).

С точки зрения Microsoft, OLE DB представляет собой следующую ступень развития технологии ODBC. Обе технологии образуют относительно независимый программный слой, использующий однотипные интерфейсы прикладных программ (API) для доступа к разным типам СУБД. Их работа обеспечивается программными модулями, которые учитывают специфические особенности каждой СУБД. В OLE DB полностью реализован принцип универсального доступа к разнотипным СУБД. Наибольшие различия технологий ODBC и OLE DB проявляются в использовании некоторых основных терминов и в окружающем контексте.

Технология ADO родилась в результате объединения OLE DB с еще одной технологией, созданной Microsoft —ActiveX. Отсюда происходит и название технологии ADO — ActiveX Data Object (объекты данных ActiveX).

С точки зрения разработчика клиентской части приложения в архитектуре «клиент-сервер» ADO представляет собой прикладной объектный интерфейс более высокого уровня, чем OLE DB. Этот интерфейс упрощает доступ к средствам OLE DB разработчикам, использующим языки высокого уровня. В настоящее время разработчик этих технологий — компания Microsoft — выпустила уже несколько версий библиотек ADO и продолжает развивать эту технологию.

Однако развитие этих технологий ограничено, прежде всего, тем, что они ориентированы в основном только на вычислительные системы, построенные на базе ОС типа Windows. И хотя появляются версии ADO для других типов ОС, переносимость приложений, созданных на основе этой технологии, на вычислительные системы, не использующие Windows, остается проблемой.

Способы организации взаимодействия клиентской и серверной частей приложений на основе архитектуры «клиент-сервер» не ограничиваются только перечисленными технологиями. Сложно даже перечислить их все в рамках данного учебного пособия, а тем более подробно описать. Более подробно об организации приложений на основе архитектуры «клиент-сервер» можно узнать в [2, 19, 26, 28, 29, 31, 35].

Поддержка технологий доступа к СУБД в системах программирования

Появление и развитие различных технологий доступа клиентской части приложений к серверу данных не осталось без внимания разработчиков систем программирования (особенно если учесть, что такие создатели новых технологий, как компания Microsoft, являются одновременно и разработчиками систем программирования). Поэтому на рынке систем программирования появились системы, поддерживающие основные существующие технологии доступа клиента к серверу данных.

Если разработчик клиентской части приложения останавливает свой выбор на одной из существующих стандартных технологий, то в этом случае он не только избегает зависимости создаваемого приложения от типа СУБД, но и получает возможность выбора среди систем программирования, доступных на рынке. Это обусловлено тем, что практически все современные системы программирования поддерживают такие распространенные технологии как ODBC или ADO и предоставляют разработчикам инструменты и библиотеки, снижающие трудоемкость

создания клиентов на основе таких технологий. В дальнейшем возможности созданного приложения будут ограничены только возможностями выбранной технологии: например, технология ADO имеет широкие возможности, но ее применение ограничено только вычислительными системами, построенными на базе ОС типа Windows.

Таким образом, выбирая стандартную технологию, разработчик получает выигрыш в выборе СУБД и средств разработки, но проигрывает в производительности, так как универсальные методы доступа к СУБД менее эффективны, чем специализированные.

Если же разработчик клиентской части приложения выберет для взаимосвязи с сервером данных специализированную технологию, ориентированную на определенный тип СУБД, то он, безусловно, получит более высокую скорость обмена данными между сервером и клиентом. Однако в этом случае он будет ограничен и в выборе типа СУБД, и в выборе средств разработки. Первое утверждение очевидно, так как специализированный метод доступа ориентирован на определенный тип СУБД (а зачастую — и на определенную версию СУБД). Второе утверждение основано на том, что специализированные методы доступа к СУБД либо совсем не поддерживаются системами программирования от других производителей, либо имеют ограниченные инструменты поддержки, снижающие эффективность разработки. В этом случае лучше всего использовать систему программирования, созданную тем же производителем, что и СУБД (в настоящее время ведущие производители СУБД предлагают и свои средств разработки).

Выигрыш в эффективности доступа клиента к серверу данных при использовании специализированных методов обусловлен еще одним немаловажным фактором: все производители СУБД заинтересованы в создании как можно большего количества клиентских приложений, ориентированных именно на их СУБД. Поэтому все полезные для клиентов нововведения появляются в первую очередь именно в специализированных средствах доступа и только потом уже, если это возможно, переносятся в средства обеспечивающие поддержку стандартных технологий.

Каждый из описанных путей создания клиентского приложения в архитектуре «клиент-сервер» не лишен своих преимуществ и недостатков. Выбор одного из них остается за разработчиком и зависит от тех условий, для которых создается то или иное приложение.

Проблемы и недостатки архитектуры «клиент-сервер»

Приложения, созданные на основе архитектуры «клиент-сервер», получили ряд преимуществ по сравнению с приложениями, не использующими распределенных вычислений, а также приложениями, созданными на основе архитектуры «файл-сервер». Тот факт, что основные производители систем программирования включили в свои системы средства поддержки разработки приложений в этой архитектуре, обусловил широкое распространение таких приложений на рынке программных средств.

В то же время, сама по себе архитектура «клиент-сервер» не лишена некоторых недостатков:

- функции управления данными возложены на сервер данных, но обработка данных (прикладная логика или бизнес-логика) по-прежнему выполняется клиентами, что не позволяет существенно снизить требования к ним, если логика приложения предусматривает достаточно сложные манипуляции с данными;
- при необходимости изменить или дополнить логику обработки данных необходимо выполнить обновление клиентских приложений на всех рабочих местах, что может быть достаточно трудоемко;
- если необходимо изменить только внешний вид интерфейса пользователя (отображение данных), но оставить неизменной логику обработки данных, при этом чаще всего требуется заново создать и установить на рабочем месте новый вариант клиентской части системы;

- при использовании мощной промышленной СУБД требуется наличие лицензии на подключение к СУБД для каждого рабочего места, где установлена клиентская часть программного обеспечения.

Наличие хранимых процедур и других мощных средств, расширяющих возможности серверов данных, позволяет перенести на них так же и значительную долю функций прикладной логики. Это несколько нивелирует большинство указанных недостатков архитектуры «клиент-сервер», но не позволяет полностью избавится от них по следующим причинам:

- внутренние языки СУБД, используемые для создания хранимых процедур и других подобных средств, хотя и сравнимы в настоящее время по своим возможностям с языками современных систем программирования, но всё же уступают им — поэтому не все функции прикладной логики могут быть реализованы на сервере данных столь же эффективно, как на клиенте;
- хранимые процедуры, триггеры и другие средства СУБД, созданные на ее внутреннем языке, после предварительной компиляции сохраняются во внутреннем коде СУБД (но не в машинных кодах), а во время исполнения интерпретируются СУБД — поэтому они проигрывают в производительности результирующим программам, созданным в машинных кодах.

При разработке приложений в архитектуре «клиент-сервер» вовсе не следует стремиться все функции прикладной логики возложить на сервер данных. Следует помнить, что СУБД интерпретирует код, а потому серверу всегда потребуется больше ресурсов для выполнения тех же функций, которые на клиенте выполняются в машинных кодах. Разумное распределение функций между клиентской и серверной частью приложения зависит от многих факторов, в том числе от количества клиентов, мощности клиентских и серверного компьютеров и др.

Еще одним средством, которое может расширить возможности приложений, построенных на основе архитектуры «клиент-сервер», является использование терминального доступа в сочетании с возможностями этой архитектуры. В этом случае компьютер-клиент, на котором выполняется клиентская часть приложения, одновременно становится терминальным сервером, к которому могут подключаться компьютеры-терминалы. Тогда вся логика организации пользовательского интерфейса выполняется компьютерами-терминалами, прикладная логика приложения выполняется на терминальном сервере (который одновременно является клиентом), а логика обработки данных и файловые операции выполняются на сервере данных. Поскольку клиентов может быть несколько, то и терминальных серверов в такой структуре может быть несколько. Тогда пользователь компьютера-терминала либо подключается только к заданному терминальному серверу, либо имеет возможность выбрать один из доступных терминальных серверов для подключения.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Получить вариант задания у преподавателя.
2. Разработать структуру БД в соответствии с полученным заданием, выбрать тип используемой СУБД (по согласованию с преподавателем).
3. Создать БД в соответствии с разработанной структурой. При необходимости реализовать хранимые процедуры, триггеры и другие средства обработки данных на сервере.
4. Изучить принципы построения приложений на основе архитектуры «клиент-сервер». Выбрать технологию для взаимодействия с серверной частью.
5. Выбрать систему программирования для разработки клиентской части приложения (по согласованию с преподавателем).
6. Используя выбранную технологию взаимодействия с серверной частью, с помощью выбранной системы программирования разработать клиентскую часть приложения.
7. Написать и отладить программу на ЭВМ.
8. Подготовить и защитить отчет.
9. Продемонстрировать разработанное приложение преподавателю.

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет по лабораторной работе должен содержать следующие разделы:

- Краткое изложение цели работы.
- Задание по лабораторной работе.
- Схему организации БД (БД должна содержать не менее трех взаимосвязанных таблиц).
- Описание выбранной технологии взаимодействия клиентской и серверной частей приложения.
- Текст программы (оформляется при необходимости по согласованию с преподавателем).
- Выводы по проделанной работе.

ОСНОВНЫЕ КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите уровни обработки данных.
2. Какие варианты архитектур для организации распределенных вычислений существуют?
3. Расскажите об архитектуре распределенных вычислений типа «файл-сервер».
4. Что такое «файл-сервер»? Назовите особенности его реализации.
5. Расскажите о преимуществах и недостатках архитектуры распределенных вычислений типа «файл-сервер».
6. Расскажите об архитектуре распределенных вычислений типа «клиент-сервер».
7. Назовите основные особенности архитектуры «клиент-сервер»
8. Какие функции может выполнять сервер в архитектуре «клиент-сервер»?
9. Какие существуют методы работы с базами данных?
10. Дайте характеристику интерфейса ODBC.
11. Какие источники данных могут быть использованы ODBC?
12. Как осуществляется работа программы при использовании ODBC?
13. Расскажите о преимуществах и недостатках архитектуры распределенных вычислений типа «клиент-сервер».
14. Расскажите о методах борьбы с недостатками архитектуры распределенных вычислений типа «файл-сервер».
15. Расскажите о трехуровневой и многоуровневой архитектуре распределенных вычислений.
16. Расскажите о преимуществах и недостатках многоуровневой архитектуры распределенных вычислений.

ВАРИАНТЫ ЗАДАНИЙ

Общие требования к разрабатываемому приложению:

1. Приложение должно реализовывать простейшую прикладную функцию в соответствии с заданием. Конкретный состав выполняемых функций разрабатывается студентами самостоятельно, при необходимости согласовывается с преподавателем.
2. Функции приложения должны выполняться на основе взаимодействия с БД. Состав таблиц и структура БД разрабатывается студентами самостоятельно, согласовывается с преподавателем. Как правило, БД должна содержать 3-5 взаимосвязанных таблиц.
3. Приложение должно работать в двух режимах: административном режиме и режиме пользователя. В административном режиме должно быть доступно больше функций по управлению данными (состав функций для каждого режима указан в варианте задания). Два режима работы приложения должны быть реализованы либо в виде двух программных модулей, либо в виде двух режимов работы одного программного модуля – в этом случае выбор режима работы должен выполняться на основе ввода имени и пароля пользователя при запуске программного модуля.
4. Интерфейс клиентской части приложения должен быть простым, понятным и ориентированным на пользователя. В частности, при выборе данных для взаимосвязанных

таблиц выбор должен осуществляться из связанной таблицы с наглядным отображением значащих полей, поиск должен осуществляться по заданным пользователем критериям, при возникновении ошибок, связанных с обработкой данных, должны выдаваться соответствующие сообщения (не сообщения сервера взаимодействия с СУБД!).

Таблица 12.

Варианты заданий для лабораторной работы №10

№	Функции приложения	Функции администратора	Функции пользователя
1.	Библиотека	Ввод и редактирование данных об имеющихся книгах.	Библиотекарь: выдача книг и получение книг от читателей.
2.	Библиотека	Библиотекарь: обработка поступивших заявок, выдача книг и получение книг от читателей.	Читатель: поиск информации о наличии книг, заявка на получение найденной книги.
3.	Театральная касса	Ввод и редактирование данных о спектаклях и наличии билетов.	Кассир: продажа и возврат билетов.
4.	Театральная касса	Кассир: обработка поступивших заказов, продажа и возврат билетов.	Зритель: поиск информации о спектаклях и о наличии билетов на них, заказ билета на выбранный спектакль.
5.	Железнодорожная касса	Ввод и редактирование данных о расписании и маршрутах движения поездов.	Кассир: продажа и возврат билетов.
6.	Железнодорожная касса	Кассир: обработка поступивших заказов, продажа и возврат билетов.	Пассажир: поиск информации о возможности проезда до станции назначения и о наличии мест, заказ билета на определенный поезд.
7.	Автосалон	Ввод и редактирование данных об автомобилях, их комплектации и сроках поставки.	Продавец: заключение договоров о покупке автомобиля, заказ автомобиля нужной комплектации.
8.	Автосалон	Продавец: обработка заявки, заключение договоров о покупке автомобиля, заказ автомобиля нужной комплектации.	Покупатель: поиск информации об автомобиле нужной комплектации, формирование заявки на автомобиль.
9.	Мебельная фабрика	Ввод и редактирование информации о конструкции мебели.	Изготовитель: ввод информации о поступлении комплектующих на склад и об изготовлении мебели.
10.	Мебельная фабрика	Изготовитель: ввод информации о поступлении комплектующих на склад и об изготовлении мебели, продажа изготовленной мебели.	Покупатель: поиск наличия необходимой мебели, заказ и приобретение мебели по заказу.
11.	Стадион	Формирование расписания матчей.	Кассир: продажа билетов и возврат проданных билетов.
12.	Стадион	Кассир: обработка заказов, продажа билетов и возврат проданных билетов.	Зритель: поиск информации об интересующем матче, заказ билетов на выбранный матч.

По согласованию с преподавателем, допускается объединять однотипные по функциям задания (например, варианты 1 и 2) для выполнения их группой студентов (2-3 человека). В этом случае разрабатывается одна БД и несколько приложений (в соответствии с вариантами задания).

Допускается выполнение студентами своих собственных вариантов работы (по выбору) при условии предварительного согласования с преподавателем своего варианта задания и функций разрабатываемого приложения.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

ОСНОВНАЯ ЛИТЕРАТУРА

1. Карпов Ю. Г. Теория и технология программирования. Основы построения трансляторов. – СПб.: БХВ-Петербург, 2012 – 272 с.
2. Молчанов А. Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. — СПб.: Питер, 2010 — 400 с.
3. Ахо А., Лам М., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. – М.: Издательский дом «Вильямс», 2008 – 1184 с.
4. Свердлов С.З. Языки программирования и методы трансляции: учеб. пособие. — СПб.: Питер, 2007 — 400 с.
5. Теория языков программирования и методы трансляции: методические указания к выполнению практических занятий – СПб.: ГОУ ВПО «СПб ГУАП», 2007 – 27 с.
6. Молчанов А.Ю. Системное программное обеспечение. Лабораторный практикум. – СПб.: Питер, 2005 – 284 с.
7. Карпов Ю. Г. Теория и технология программирования. Основы построения трансляторов. – СПб.: БХВ-Петербург, 2005 – 272 с.
8. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты: Пер. с англ. — М.: Издательский дом «Вильямс», 2003 — 768 с.
9. Робин Хантер Основные концепции компиляторов – М.: Издательский дом «Вильямс», 2002 – 256 с.
10. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение – СПб.: Питер, 2001 (2002) - 736 с.
11. Коровинский В.В., Жаков В.И., Фильчаков В.В. Синтаксический анализ и генерация кода – СПб.: ГААП, 1993.
12. Бржезовский А.В., Корсакова Н.В., Фильчаков В.В. Лексический и синтаксический анализ. Формальные языки и грамматики – Л.: ЛИАП, 1990.
13. Льюис Ф. и др. Теоретические основы построения компиляторов - М.: Мир, 1979.
14. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции – М.: Мир, 1978, т.1.
15. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции - М.: Мир, 1978, т.2.
16. Грис Д. Конструирование компиляторов для цифровых вычислительных машин - М.: Мир, 1975.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

17. Головин И. Г. Языки и методы программирования /И. Г. Головин, И. А. Волкова – М.: Издательский центр «Академия», 2012 – 304 с.
18. Олифер В.Г., Олифер Н.А. Сетевые операционные системы: учебник для вузов. — СПб.: Питер, 2008 — 672 с.
19. Таненбаум Э. Современные операционные системы. 2-е изд. — СПб.: Питер, 2007 — 1038 с.
20. Гордеев А.В. Операционные системы: учебник для вузов. — СПб.: Питер, 2004 — 416 с.
21. Карпов Ю.Г. Теория автоматов: Учебник для вузов — СПб.: Питер, 2003 — 208 с.
22. Карпова Т.С. Базы данных: модели, разработка, реализация. — СПб.: Питер, 2001 — 304 с.
23. Корсакова Н.В., Пятлина Е.О. Фильчаков В.В. Структуры данных - Л.: ЛИАП, 1986.
24. http://www.fi.ru/~mill
25. http://www.codegear.com
26. http://www.corba.ru/

27. http://www.perl.org/
28. http://www.php.net/
29. http://www.microsoft.com/rus/msdn/vs/default.mspx
30. http://msdn.microsoft.com/ru-ru/aa496123.aspx
31. http://www.citforum.ru/programming/32less/les44.shtml
32. http://www.citforum.ru/cfin/prcorpsys/index.shtml
33. http://www.visual.2000.ru/develop/ms-vb/cp9907-2/msado1-1.htm
34. http://www.interface.ru/fset.asp?Url=/borland/com_dcom.htm
35. http://www.citforum.ru/programming/middleware/midas_4.shtml