Лабораторная работа № 4
Генерация и оптимизация объектного кода

Цель работы

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

Краткие теоретические сведения

Общие принципы генерации кода

Генерация объектного кода – это перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка. Поскольку выходным языком компилятора (в отличие от транслятора) может быть только либо язык ассемблера, либо язык машинных кодов, то генерация кода порождает результирующую объектную программу на языке ассемблера или непосредственно на машинном языке (в машинных кодах).

Генерация объектного кода выполняется после того, как выполнены лексический и синтаксический анализ программы и все необходимые действия по подготовке к генерации кода: проверены семантические соглашения входного языка (семантический анализ), выполнена идентификация имен переменных и функций, распределено адресное пространство под функции и переменные и т. д.

В данной лабораторной работе используется предельно простой входной язык, поэтому нет необходимости выполнять все перечисленные преобразования. Будем считать, что все они уже выполнены. Более подробно все эти фазы компиляции описаны в [1–4, 7], а здесь речь будет идти только о самых примитивных приемах семантического анализа, которые будут проиллюстрированы на примере выполнения лабораторной работы.

Внутреннее представление программы может иметь любую структуру в зависимости от реализации компилятора, в то время как результирующая программа всегда представляет собой линейную последовательность команд. Поэтому генерация объектного кода (объектной программы) в любом случае должна выполнять действия, связанные с преобразованием сложных синтаксических структур в линейные цепочки.

Генерацию кода можно считать функцией, определенной на синтаксическом дереве, построенном в результате синтаксического анализа, и на информации, содержащейся в таблице идентификаторов. Характер отображения входной программы в последовательность команд, выполняемого генерацией, зависит от входного языка, архитектуры целевой вычислительной системы, на которую ориентирована результирующая программа, а также от качества желаемого объектного кода.

В идеале компилятор должен выполнить синтаксический анализ всей входной программы, затем провести ее семантический анализ, после чего приступать к подготовке генерации и непосредственно генерации кода. Однако такая схема работы компилятора практически почти никогда не применяется. Дело в том, что в общем случае ни один семантический анализатор и ни один компилятор не способны проанализировать и оценить смысл всей исходной программы в целом. Формальные методы анализа семантики применимы только к очень незначительной части возможных исходных программ. Поэтому у компилятора нет практической возможности порождать эквивалентную результирующую программу на основе всей исходной программы.

Как правило, компилятор выполняет генерацию результирующего кода поэтапно, на основе законченных синтаксических конструкций входной программы. Компилятор выделяет законченную синтаксическую конструкцию из текста исходной программы, порождает для нее фрагмент результирующего кода и помещает его в текст результирующей программы. Затем он переходит к следующей синтаксической конструкции. Так продолжается до тех пор, пока не будет разобрана вся исходная программа. В качестве анализируемых законченных синтаксических конструкций выступают блоки операторов, описания процедур и функций. Их конкретный состав зависит от входного языка и реализации компилятора.

Смысл (семантику) каждой такой синтаксической конструкции входного языка можно определить, исходя из ее типа, а тип определяется синтаксическим анализатором на основе грамматики входного языка. Примерами типов синтаксических конструкций могут служить операторы цикла, условные операторы, операторы выбора и т. д. Одни и те же типы синтаксических конструкций характерны для различных языков программирования, при этом они различаются синтаксисом (который задается грамматикой языка), но имеют схожий смысл (который определяется семантикой). В зависимости от типа синтаксической конструкции выполняется генерация кода результирующей программы, соответствующего данной синтаксической конструкции. Для семантически схожих конструкций различных входных языков программирования может порождаться типовой результирующий код.

Синтаксически управляемый перевод

Чтобы компилятор мог построить код результирующей программы для синтаксической конструкции входного языка, часто используется метод, называемый синтаксически управляемым переводом – СУ-переводом.

Идея СУ-перевода основана на том, что синтаксис и семантика языка взаимосвязаны. Это значит, что смысл предложения языка зависит от синтаксической структуры этого предложения. Теория синтаксически управляемого перевода была предложена американским лингвистом Ноамом Хомским. Она справедлива как для формальных языков, так и для языков естественного общения: например, смысл предложения русского языка зависит от входящих в него частей речи (подлежащего, сказуемого,

дополнений и др.) и от взаимосвязи между ними. Однако естественные языки допускают неоднозначности в грамматиках – отсюда происходят различные двусмысленные фразы, значение которых человек обычно понимает из того контекста, в котором эти фразы встречаются (и то он не всегда может это сделать). В языках программирования неоднозначности в грамматиках исключены, поэтому любое предложение языка имеет четко определенную структуру и однозначный смысл, напрямую связанный с этой структурой.

Входной язык компилятора имеет бесконечное множество допустимых предложений, поэтому невозможно задать смысл каждого предложения. Но все входные предложения строятся на основе конечного множества правил грамматики, которые всегда можно найти. Так как этих правил конечное число, то для каждого правила можно определить его семантику (значение).

Но абсолютно то же самое можно утверждать и для выходного языка компилятора. Выходной язык содержит бесконечное множество допустимых предложений, но все они строятся на основе конечного множества известных правил, каждое из которых имеет определенную семантику (смысл). Если по отношению к исходной программе компилятор выступает в роли распознавателя, то для результирующей программы он является генератором предложений выходного языка. Задача заключается в том, чтобы найти порядок правил выходного языка, по которым необходимо выполнить генерацию.

Грубо говоря, идея СУ-перевода заключается в том, что каждому правилу входного языка компилятора сопоставляется одно или несколько (или ни одного) правил выходного языка в соответствии с семантикой входных и выходных правил. То есть при сопоставлении надо выбирать правила выходного языка, которые несут тот же смысл, что и правила входного языка.

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 C(N). Код для вершины N строится путем сцепления (конкатенации) в фиксированном порядке последовательности кода C(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.

Для того чтобы построить СУ-перевод по заданному дереву синтаксического разбора, необходимо найти последовательность кода для корня дерева. Поэтому для каждой вершины дерева порождаемую цепочку кода надо выбирать таким образом, чтобы код, приписываемый корню дерева, оказался

искомым кодом для всего оператора, представленного этим деревом. В общем случае необходимо иметь единообразную интерпретацию кода C(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 оценки смысловой нагрузки для каждой вершины дерева решается разработчиком компилятора.

Возможна модель компилятора, в которой синтаксический анализ исходной программы и генерация кода результирующей программы объединены в одну фазу. Такую модель можно представить в виде компилятора, у которого операции генерации кода совмещены с операциями выполнения синтаксического разбора. Для описания компиляторов такого типа часто используется термин СУ-компиляция (синтаксически управляемая компиляция).

Схему СУ-компиляции можно реализовать не для всякого входного языка программирования. Если принцип СУ-перевода применим ко всем входным КС-языкам, то применить СУ-компиляцию оказывается не всегда возможным [1, 2, 7].

В процессе СУ-перевода и СУ-компиляции не только вырабатываются цепочки текста выходного языка, но и совершаются некоторые дополнительные действия, выполняемые самим компилятором. В общем случае схемы СУ-перевода могут предусматривать выполнение следующих действий:

• помещение в выходной поток данных машинных кодов или команд ассемблера, представляющих собой результат работы (выход) компилятора;

• выдача пользователю сообщений об обнаруженных ошибках и предупреждениях (которые должны помещаться в выходной поток, отличный от потока, используемого для команд результирующей программы);

• порождение и выполнение команд, указывающих, что некоторые действия должны быть произведены самим компилятором (например операции, выполняемые над данными, размещенными в таблице идентификаторов).

Ниже рассмотрены некоторые основные технические вопросы, позволяющие реализовать схемы СУ-перевода для данной лабораторной работы. Более подробно с механизмами СУ-перевода и СУ-компиляции можно ознакомиться в [1, 2, 7].

Способы внутреннего представления программ

Результатом работы синтаксического анализатора на основе КС-грамматики входного языка является последовательность правил грамматики, примененных для построения входной цепочки. По найденной последовательности, зная тип распознавателя, можно построить цепочку вывода или дерево вывода. В этом случае дерево вывода выступает в качестве дерева синтаксического разбора и представляет собой результат работы синтаксического анализатора в компиляторе.

Однако ни цепочка вывода, ни дерево синтаксического разбора не являются целью работы компилятора. Для полного представления о структуре разобранной синтаксической конструкции входного языка в принципе достаточно знать последовательность номеров правил грамматики, примененных для ее построения. Однако форма представления этой информации может быть различной в зависимости как от реализации самого компилятора, так и от фазы компиляции. Эта форма называется внутренним представлением программы (иногда используются также термины промежуточное представление или промежуточная программа).

Все внутренние представления программы обычно содержат в себе два принципиально различных элемента – операторы и операнды. Различия между формами внутреннего представления заключаются лишь в том, как операторы и операнды соединяются между собой. Также операторы и операнды должны отличаться друг от друга, если они встречаются в любом порядке. За различение операндов и операторов, как уже было сказано выше, отвечает разработчик компилятора, который руководствуется семантикой входного языка.

Известны следующие формы внутреннего представления программ:

• структуры связных списков, представляющие синтаксические деревья;
• многоадресный код с явно именуемым результатом (тетрады);
• многоадресный код с неявно именуемым результатом (триады);
• обратная (постфиксная) польская запись операций;
• ассемблерный код или машинные команды.

В каждом конкретном компиляторе может использоваться одна из этих форм, выбранная разработчиками. Но чаще всего компилятор не ограничивается использованием только одной формы для внутреннего представления программы.

На различных фазах компиляции могут использоваться различные формы, которые по мере выполнения проходов компилятора преобразуются одна в другую.

Некоторые компиляторы, незначительно оптимизирующие результирующий код, генерируют объектный код по мере разбора исходной программы. В этом случае применяется схема СУ-компиляции, когда фазы синтаксического разбора, семантического анализа, подготовки и генерации объектного кода совмещены в одном проходе компилятора. Тогда внутреннее представление программы существует только условно в виде последовательности шагов алгоритма разбора.

Алгоритмы, предложенные для выполнения данной лабораторной работы, построены на основе использования формы внутреннего представления программы в виде триад. Поэтому далее будет рассмотрена именно эта форма внутреннего представления программы. С остальными формами можно более подробно познакомиться в [1–3, 7].

Многоадресный код с неявно именуемым результатом (триады)

Триады представляют собой запись операций в форме из трех составляющих: операция и два операнда. Например, в строковой записи триады могут иметь вид: <операция>(<операнд1>,<операнд2>). Особенностью триад является то, что 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 нумеруют для удобства указания ссылок одних триад на другие (в реализации компилятора в качестве ссылок можно использовать не номера триад, а непосредственно ссылки в виде указателей – тогда при изменении нумерации и порядка следования триад менять ссылки не требуется).

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 соответствующими знаками (при этом присваивание также является операцией), а знак ^ означает ссылку операнда одной триады на результат другой.

Триады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме триад, они вычисляются одна за другой последовательно. Каждая триада в последовательности вычисляется так: операция, заданная триадой, выполняется над операндами, а если в качестве одного из операндов (или обоих операндов) выступает ссылка на другую триаду, то берется результат вычисления той триады. Результат вычисления триады нужно сохранять во временной памяти, так как он может быть затребован последующими триадами. Если какой-то из операндов в триаде отсутствует (например, если триада представляет собой унарную операцию), то он может быть опущен или заменен пустым операндом (в зависимости от принятой формы записи и ее реализации). Порядок вычисления триад может быть изменен, но только если допустить наличие триад, целенаправленно изменяющих этот порядок (например, триады, вызывающие безусловный переход на другую триаду с заданным номером или переход на несколько шагов вперед или назад при каком-то условии).

Триады представляют собой линейную последовательность, а потому для них несложно написать тривиальный алгоритм, который будет преобразовывать последовательность триад в последовательность команд результирующей программы (либо последовательность команд ассемблера). В этом их преимущество перед синтаксическими деревьями. Однако для триад требуется также и алгоритм, отвечающий за распределение памяти, необходимой для хранения промежуточных результатов вычисления, так как временные переменные для этой цели не используются (в этом отличие триад от тетрад).

Триады не зависят от архитектуры вычислительной системы, на которую ориентирована результирующая программа. Поэтому они представляют собой машинно-независимую форму внутреннего представления программы.
Триады обладают следующими преимуществами:
• являются линейной последовательностью операций, в отличие от синтаксического дерева, и потому проще преобразуются в результирующий код;
• занимают меньше памяти, чем тетрады, дают больше возможностей по оптимизации программы, чем обратная польская запись;
• явно отражают взаимосвязь операций между собой, что делает их применение удобным, особенно при оптимизации внутреннего представления программы;
• промежуточные результаты вычисления триад могут храниться в регистрах процессора, что удобно при распределении регистров и выполнении машинно-зависимой оптимизации;
• по форме представления находятся ближе к двухадресным машинным командам, чем другие формы внутреннего представления программ, а именно эти команды более всего распространены в наборах команд большинства современных компьютеров.
Необходимость создания алгоритма, отвечающего за распределение памяти для хранения промежуточных результатов, является главным недостатком триад. Но при грамотном распределении памяти и регистров процессора этот недостаток может быть обращен на пользу разработчиками компилятора.

Схемы СУ-перевода
Ранее был описан принцип СУ-перевода, позволяющий получить линейную последовательность команд результирующей программы или внутреннего представления программы в компиляторе на основе результатов синтаксического анализа. Теперь построим вариант алгоритма генерации кода, который получает на входе дерево синтаксического разбора и создает по нему последовательность триад (далее – просто «триады») для линейного участка результирующей программы. Рассмотрим примеры схем СУ-перевода для бинарных арифметических операций. Эти схемы достаточно просты, и на их основе можно проиллюстрировать, как выполняется СУ-перевод в компиляторе при генерации кода.
Для построения триад по синтаксическому дереву может использоваться простейшая рекурсивная процедура обхода дерева. Можно использовать и другие методы обхода дерева – важно, чтобы соблюдался принцип, согласно которому нижележащие операции в дереве всегда выполняются перед вышележащими операциями (порядок выполнения операций одного уровня не важен, он не влияет на результат и зависит от порядка обхода вершин дерева).
Процедура генерации триад по синтаксическому дереву прежде всего должна определить тип узла дерева. Для бинарных арифметических операций каждый узел дерева имеет три нижележащие вершины (левая вершина – первый операнд, средняя вершина – операция и правая вершина – второй операнд).

При этом тип узла дерева соответствует типу операции, символом которой помечена средняя из нижележащих вершин. После определения типа узла процедура строит триады для узла дерева в соответствии с типом операции.
Фактически процедура генерации триад должна для каждого узла дерева выполнить конкатенацию триады, связанной с текущим узлом, и цепочек триад, связанных с нижележащими узлами. Конкатенация цепочек триад должна выполняться таким образом, чтобы триады, связанные с нижележащими узлами, выполнялись до выполнения операции, связанной с текущим узлом. Причем для арифметических операций важно, чтобы триады, связанные с первым операндом, выполнялись раньше, чем триады, связанные со вторым операндом (так как все арифметические операции при отсутствии скобок и приоритетов выполняются в порядке слева направо).
При этом возможны четыре ситуации:
• левая и правая вершины указывают на непосредственный операнд (это можно определить, если у каждой из них есть только один нижележащий узел, помеченный символом какой-то лексемы – константы или идентификатора);
• левая вершина является непосредственным операндом, а правая указывает на другую операцию;
• левая вершина указывает на другую операцию, а правая является непосредственным операндом;
• обе вершины указывают на другую операцию.
Считаем, что на вход процедуры порождения триад по синтаксическому дереву подается список, в который нужно добавлять триады, и ссылка на узел дерева, который надо обработать. Тогда процедура порождения триад для узла синтаксического дерева, связанного с бинарной арифметической операцией, может выполняться по следующему алгоритму:
1. Проверяется тип левой вершины узла. Если она – простой операнд, запоминается имя первого операнда, иначе для этой вершины рекурсивно вызывается процедура порождения триад, построенные ею триады добавляются в конец общего списка и запоминается номер последней триады из этого списка как первый операнд.
2. Проверяется тип правой вершины узла. Если она – простой операнд, запоминается имя второго операнда, иначе для этой вершины рекурсивно вызывается процедура порождения триад, построенные ею триады добавляются в конец общего списка и запоминается номер последней триады как второй операнд.
3. В соответствии с типом средней вершины в конец общего списка добавляется триада, соответствующая арифметической операции. Ее первым операндом становится операнд, запомненный на шаге 1, а вторым операндом – операнд, запомненный на шаге 2.
4. Процедура закончена.
Процедуры такого рода должен создавать разработчик компилятора, так как только он может сопоставить по смыслу узлы синтаксического дерева и соответствующие им последовательности триад. Для разных типов узлов синтаксического дерева могут быть построены разные варианты процедур,

которые будут вызывать друг друга в зависимости от принятого порядка обхода синтаксического дерева (в описанном выше варианте – рекурсивно).
В рассмотренном примере при порождении кода преднамеренно не были приняты во внимание многие вопросы, возникающие при построении реальных компиляторов. Это было сделано для упрощения примера. Например, фрагменты кода, соответствующие различным узлам дерева, принимают во внимание тип операции, но никак не учитывают тип операндов. Все эти требования ведут к тому, что в реальном компиляторе при генерации кода надо принимать во внимание очень многие особенности, зависящие от семантики входного языка и от используемой формы внутреннего представления программы. В данной лабораторной работе эти вопросы не рассматриваются.
Кроме того, в случае арифметических операций код, порождаемый для узлов синтаксического дерева, зависит только от типа операции, то есть только от текущего узла дерева. Такие схемы можно построить для многих операций, но не для всех. Иногда код, порождаемый для узла дерева, может зависеть от типа вышестоящего узла: например, код, порождаемый для операторов типа Break и Continue (которые есть в языках C, C++ и Object Pascal), зависит от того, внутри какого цикла они находятся. Тогда при рекурсивном построении кода по дереву вышестоящий узел, вызывая функцию для нижестоящего узла, должен передать ей необходимые параметры. Но код, порождаемый для вышестоящего узла, никогда не должен зависеть от нижестоящих узлов, в противном случае принцип СУ-перевода неприменим.
Далее в примере выполнения работы даются варианты схем СУ-перевода для различных конструкций входного языка, которые могут служить хорошей иллюстрацией механизма применения этого метода.

Общие принципы оптимизации кода
Как уже говорилось, в подавляющем большинстве случаев генерация кода выполняется компилятором не для всей исходной программы в целом, а последовательно для отдельных ее конструкций. Для построения результирующего кода различных синтаксических конструкций входного языка используется метод СУ-перевода. Он объединяет цепочки построенного кода по структуре дерева без учета их взаимосвязей.
Построенный таким образом код результирующей программы может содержать лишние команды и данные. Это снижает эффективность выполнения результирующей программы. В принципе, компилятор может завершить на этом генерацию кода, поскольку результирующая программа построена и является эквивалентной по смыслу (семантике) программе на входном языке. Однако эффективность результирующей программы важна для ее разработчика, поэтому большинство современных компиляторов выполняют еще один этап компиляции – оптимизацию результирующей программы (или просто «оптимизацию»), чтобы повысить ее эффективность, насколько это возможно.

Важно отметить два момента: во-первых, выделение оптимизации в отдельный этап генерации кода – это вынужденный шаг. Компилятор вынужден производить оптимизацию построенного кода, поскольку он не может выполнить семантический анализ всей входной программы в целом, оценить ее смысл и исходя из него построить результирующую программу. Во-вторых, оптимизация – это необязательный этап компиляции. Компилятор может вообще не выполнять оптимизацию, и при этом результирующая программа будет правильной, а сам компилятор будет полностью выполнять свои функции. Однако практически все современные компиляторы так или иначе выполняют оптимизацию, поскольку их разработчики стремятся завоевать хорошие позиции на рынке средств разработки программного обеспечения.

Теперь дадим определение понятию «оптимизация».

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.

В качестве показателей эффективности результирующей программы можно использовать два критерия: объем памяти, необходимый для выполнения результирующей программы, и скорость выполнения (быстродействие) программы. Далеко не всегда удается выполнить оптимизацию так, чтобы она удовлетворяла обоим этим критериям. Зачастую сокращение необходимого программе объема данных ведет к уменьшению ее быстродействия, и наоборот. Поэтому для оптимизации обычно выбирается один из упомянутых критериев. Выбор критерия оптимизации обычно выполняется в настройках компилятора.

Но даже выбрав критерий оптимизации, в общем случае практически невозможно построить код результирующей программы, который бы являлся самым коротким или самым быстрым кодом, соответствующим входной программе. Дело в том, что нет алгоритмического способа нахождения самой короткой или самой быстрой результирующей программы, эквивалентной заданной исходной программе. Эта задача в принципе неразрешима. Существуют алгоритмы, которые можно ускорять сколь угодно много раз для большого числа возможных входных данных, и при этом для других наборов входных данных они окажутся неоптимальными [1, 2]. К тому же компилятор обладает весьма ограниченными средствами анализа семантики всей входной программы в целом. Все, что можно сделать на этапе оптимизации, – это выполнить над заданной программой последовательность преобразований в надежде сделать ее более эффективной.

Чтобы оценить эффективность результирующей программы, полученной с помощью того или иного компилятора, часто прибегают к сравнению ее с эквивалентной программой (программой, реализующей тот же алгоритм), полученной из исходной программы, написанной на языке ассемблера. Лучшие оптимизирующие компиляторы могут получать результирующие

объектные программы из сложных исходных программ, написанных на языках высокого уровня, почти не уступающие по качеству программам на языке ассемблера. Обычно соотношение эффективности программ, построенных с помощью компиляторов с языков высокого уровня, и программ, построенных с помощью ассемблера, составляет 1,1–1,3. То есть объектная программа, построенная с помощью компилятора с языка высокого уровня, обычно содержит на 10–30 % больше команд, чем эквивалентная ей объектная программа, построенная с помощью ассемблера, а также выполняется на 10–30 % медленнее.

Это очень неплохие результаты, достигнутые компиляторами с языков высокого уровня, если сравнить трудозатраты на разработку программ на языке ассемблера и языке высокого уровня. Далеко не каждую программу можно реализовать на языке ассемблера в приемлемые сроки (а значит и выполнить напрямую приведенное выше сравнение можно только для узкого круга программ).

Оптимизацию можно выполнять на любой стадии генерации кода, начиная от завершения синтаксического разбора и вплоть до последнего этапа, когда порождается код результирующей программы. Если компилятор использует несколько различных форм внутреннего представления программы, то каждая из них может быть подвергнута оптимизации, причем различные формы внутреннего представления ориентированы на различные методы оптимизации [1–3, 7]. Таким образом, оптимизация в компиляторе может выполняться несколько раз на этапе генерации кода.

Принципиально различаются два основных вида оптимизирующих преобразований:

• преобразования исходной программы (в форме ее внутреннего представления в компиляторе), не зависящие от результирующего объектного языка;

• преобразования результирующей объектной программы.

Первый вид преобразований не зависит от архитектуры целевой вычислительной системы, на которой будет выполняться результирующая программа. Обычно он основан на выполнении хорошо известных и обоснованных математических и логических преобразований, производимых над внутренним представлением программы (некоторые из них будут рассмотрены ниже).

Второй вид преобразований может зависеть не только от свойств объектного языка (что очевидно), но и от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. Так, например, при оптимизации может учитываться объем кэш-памяти и методы организации конвейерных операций центрального процессора. В большинстве случаев эти преобразования сильно зависят от реализации компилятора и являются «ноу-хау» производителей компилятора. Именно этот тип оптимизирующих преобразований позволяет существенно повысить эффективность результирующего кода.

Используемые методы оптимизации ни при каких условиях не должны приводить к изменению «смысла» исходной программы (то есть к таким ситуациям, когда результат выполнения программы изменяется после ее оптимизации). Для преобразований первого вида проблем обычно не возникает. Преобразования второго вида могут вызывать сложности, поскольку не все методы оптимизации, используемые создателями компиляторов, могут быть теоретически обоснованы и доказаны для всех возможных видов исходных программ. Именно эти преобразования могут повлиять на смысл исходной программы. Поэтому у современных компиляторов существуют возможности выбора не только общего критерия оптимизации, но и отдельных методов, которые будут использоваться при выполнении оптимизации.

Нередко оптимизация ведет к тому, что смысл программы оказывается не совсем таким, каким его ожидал увидеть разработчик программы, но не по причине наличия ошибки в оптимизирующей части компилятора, а потому, что пользователь не принимал во внимание некоторые аспекты программы, связанные с оптимизацией. Например, компилятор может исключить из программы вызов некоторой функции с заранее известным результатом, но если эта функция имела «побочный эффект» – изменяла некоторые значения в глобальной памяти – смысл программы может измениться. Чаще всего это говорит о плохом стиле программирования исходной программы. Такие ошибки трудноуловимы, для их нахождения разработчику программы следует обратить внимание на предупреждения, выдаваемые семантическим анализатором, или отключить оптимизацию. Применение оптимизации также нецелесообразно в процессе отладки исходной программы.

Методы преобразования программы зависят от типов синтаксических конструкций исходного языка. Теоретически разработаны методы оптимизации для многих типовых конструкций языков программирования.

Оптимизация может выполняться для следующих типовых синтаксических конструкций:

- линейных участков программы;
- логических выражений;
- циклов;
- вызовов процедур и функций;
- других конструкций входного языка.

Во всех случаях могут использоваться как машинно-зависимые, так и машинно-независимые методы оптимизации.

В лабораторной работе используются два машинно-независимых метода оптимизации линейных участков программы. Поэтому только эти два метода будут рассмотрены далее. С другими машинно-независимыми методами оптимизации можно более подробно ознакомиться в [1, 2, 7]. Что касается машинно-зависимых методов, то они, как правило, редко упоминаются в литературе. Некоторые из них рассматриваются в технических описаниях компиляторов.

Принципы оптимизации линейных участков

Линейный участок программы – это выполняемая по порядку последовательность операций, имеющая один вход и один выход. Чаще всего линейный участок содержит последовательность вычислений, состоящих из арифметических операций и операторов присваивания значений переменным. Любая программа предусматривает выполнение вычислений и присваивания значений, поэтому линейные участки встречаются в любой программе. В реальных программах они составляют существенную часть программного кода. Поэтому для линейных участков разработан широкий спектр методов оптимизации кода.

Кроме того, характерной особенностью любого линейного участка является последовательный порядок выполнения операций, входящих в его состав. Ни одна операция в составе линейного участка программы не может быть пропущена, ни одна операция не может быть выполнена большее число раз, чем соседние с нею операции (иначе этот фрагмент программы просто не будет линейным участком). Это существенно упрощает задачу оптимизации линейных участков программ. Поскольку все операции линейного участка выполняются последовательно, их можно пронумеровать в порядке их выполнения.

Для операций, составляющих линейный участок программы, могут применяться следующие виды оптимизирующих преобразований:

• удаление бесполезных присваиваний;
• исключение избыточных вычислений (лишних операций);
• свертка операций объектного кода;
• перестановка операций;
• арифметические преобразования.

Далее рассмотрены два метода оптимизации линейных участков: исключение лишних операций и свертка объектного кода.

Свертка объектного кода

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

Внимание!

Не следует путать оптимизацию по методу свертки объектного кода с рассмотренным в лабораторной работе № 3 алгоритмом «сдвиг-свертка». Свертка объектного кода и свертка по правилам грамматики при выполнении синтаксического разбора– это принципиально разные операции!

Простейший вариант свертки – выполнение в компиляторе операций, операндами которых являются константы. Несколько более сложен процесс определения тех операций, значения которых могут быть известны в результате выполнения других операций. Для этой цели при оптимизации

линейных участков программы используется специальный алгоритм свертки объектного кода.

Алгоритм свертки для линейного участка программы работает со специальной таблицей Т, 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, поскольку в других формах представления операций (таких как тетрады или команды ассемблера) требуются дополнительные структуры, чтобы отразить связь результатов одних операций с операндами других.

Рассмотрим выполнение алгоритма свертки объектного кода для триад. Для пометки операций, не требующих порождения объектного кода, будем использовать триады специального вида С(К,0).

Алгоритм свертки триад последовательно просматривает триады линейного участка и для каждой триады делает следующее:

1. Если операнд есть переменная, которая содержится в таблице Т, то операнд заменяется на соответствующее значение константы.
2. Если операнд есть ссылка на особую триаду типа С(К,0), то операнд заменяется на значение константы К.
3. Если все операнды триады являются константами, то триада может быть свернута. Тогда данная триада выполняется и вместо нее помещается особая триада вида С(К,0), где К – константа, являющаяся результатом 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 В – константа, то А со значением константы заносится в таблицу Т (если там уже было старое значение для А, то это старое значение исключается);
• если В – не константа, то А вообще исключается из таблицы Т, если оно там есть.

Рассмотрим пример выполнения алгоритма.

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

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

Ее внутреннее 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 показан в табл. 4.1.

Таблица 4.1. 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)
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)
Видно, что результирующая последовательность триад может быть подвергнута дальнейшей оптимизации – в ней присутствуют лишние присваивания, но другие методы оптимизации выходят за рамки данной лабораторной работы (с ними можно познакомиться в [1, 2, 7]).
Алгоритм свертки объектного кода позволяет исключить из линейного участка программы операции, для которых на этапе компиляции уже известен результат. За счет этого сокращается время выполнения, а также объем кода результирующей программы.
Свертка объектного кода, в принципе, может выполняться не только для линейных участков программы. Когда операндами являются константы, логика выполнения программы значения не имеет – свертка может быть выполнена в любом случае. Если же необходимо учитывать известные значения переменных, то нужно принимать во внимание и логику выполнения результирующей программы. Поэтому для нелинейных участков программы (ветвлений и циклов) алгоритм будет более сложным, чем последовательный просмотр линейного списка триад.

Исключение лишних операций
Исключение избыточных вычислений (лишних операций) заключается в нахождении и удалении из объектного кода операций, которые повторно обрабатывают одни и те же операнды.
Операция линейного участка с порядковым номером i считается лишней операцией, если существует идентичная ей операция с порядковым номером j, j< i и никакой операнд, обрабатываемый операцией с порядковым номером i, не изменялся никакой другой операцией, имеющей порядковый номер между i и j.
Алгоритм исключения лишних операций просматривает операции в порядке их следования. Так же как и алгоритму свертки, алгоритму исключения

лишних операций проще всего работать с триадами, потому что они полностью отражают взаимосвязь операций.
Рассмотрим алгоритм исключения лишних операций для триад.
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-й 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-й триады;
• при обработке i-й 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 – я триада идентична j-й триаде (j<="" p="">
Алгоритм исключения лишних операций использует в своей работе триады особого вида SAME(j,O). Если такая триада встречается в позиции с номером i, то это означает, что в исходной последовательности триад 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 какой-то операнд триады 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 в просмотренной части списка триад существует идентичная j-я триада, причем j < i и dep(i) = dep(j), то текущая триада i заменяется на триаду особого вида SAME(j,O).
4. Если текущая триада есть присваивание, 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)
Видно, что в данном примере некоторые операции вычисляются дважды над одними и теми же операндами, а значит, они являются лишними и могут быть исключены. Работа алгоритма исключения лишних операций отражена в табл. 4.2.
Таблица 4.2. Пример работы алгоритма исключения лишних операций

Обрабатываемая триада i	Числа зависимости переменных				Числа зависимости триад dep(i)	Триады, полученные после выполнения алгоритма
	A	B	C	D		
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)

Теперь, если исключить триады особого вида SAME(j,O), 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 компиляторе в качестве ссылок использовать не номера триад, а непосредственно указатели на них, то изменения ссылок в таком варианте не потребуется.
Алгоритм исключения лишних операций позволяет избежать повторного выполнения одних и тех же операций над одними и теми же операндами. В результате оптимизации по этому алгоритму сокращается и время выполнения, и объем кода результирующей программы.

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 результирующей программы.
Алгоритм должен выполнить следующую последовательность действий:
• построить последовательность триад на основе дерева вывода;
• выполнить оптимизацию кода методом свертки для линейных участков результирующей программы;

• выполнить оптимизацию кода методом исключения лишних операций для линейных участков результирующей программы;
• 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.

Требования к выполнению работы

Порядок выполнения работы

Для выполнения лабораторной работы требуется написать программу, которая на основании дерева синтаксического разбора порождает объектный код и выполняет затем его оптимизацию методом свертки объектного кода и методом исключения лишних операций. В качестве исходного дерева синтаксического разбора рекомендуется взять дерево, которое порождает программа, построенная по заданию лабораторной работы № 3.

Программу рекомендуется построить из трех основных частей: первая часть – порождение дерева синтаксического разбора (по результатам лабораторной работы № 3), вторая часть – реализация алгоритма порождения объектного кода по дереву разбора и третья часть – оптимизация порожденного объектного кода (если в результирующей программе присутствуют линейные участки кода). Результатом работы должна быть построенная на основе заданного предложения грамматики программа на объектном языке или построенная последовательность триад (по согласованию с преподавателем выбирается форма представления конечного результата).

В качестве объектного языка предлагается взять язык ассемблера для процессоров типа Intel 80x86 в реальном режиме (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 лабораторной работы № 3.

1. Получить вариант задания у преподавателя.
2. Изучить алгоритм генерации объектного кода по дереву синтаксического разбора.
3. Разработать схемы СУ-перевода для операций исходного языка в соответствии с заданной грамматикой.
4. Выполнить генерацию последовательности триад вручную для выбранного простейшего примера. Проверить корректность результата.
5. Изучить и реализовать (если требуется) для заданного входного языка алгоритмы оптимизации результирующего кода методом свертки и методом исключения лишних операций.
6. Разработать алгоритм преобразования последовательности триад в заданный объектный код (по согласованию с преподавателем).
7. Подготовить и защитить отчет.
8. Написать и отладить программу на ЭВМ.
9. 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 схем СУ-перевода для операций исходного языка в соответствии с заданной грамматикой.
• Пример генерации и оптимизации последовательности триад на основе простейшей исходной программы.
• Текст программы (оформляется после выполнения программы на ЭВМ).

Основные контрольные вопросы
• Что такое транслятор, компилятор и интерпретатор? Расскажите об общей структуре компилятора.
• Как строится дерево вывода (синтаксического разбора)? Какие исходные данные необходимы для его построения?
• Какую роль выполняет генерация объектного кода? Какие данные необходимы компилятору для генерации объектного кода? Какие действия выполняет компилятор перед генерацией?
• Объясните, почему генерация объектного кода выполняется компилятором по отдельным синтаксическим конструкциям, а не для всей исходной программы в целом.
• Расскажите, что такое синтаксически управляемый перевод.
• Объясните работу алгоритма генерации последовательности триад по дереву синтаксического разбора на своем примере.
• За счет чего обеспечивается возможность генерации кода на разных объектных языках по одному и тому же дереву?

• Дайте определение понятию оптимизации программы. Для чего используется оптимизация? Каким условиям должна удовлетворять оптимизация?
• Объясните, почему генерацию программы приходится проводить в два этапа: генерация и оптимизация.
• Какие существуют методы оптимизации объектного кода?
• Что такое триады и для чего они используются? Какие еще существуют методы для представления объектных команд?
• Объясните работу алгоритма свертки. Приведите пример выполнения свертки объектного кода.
• Что такое лишняя операция? Что такое число зависимости?
• Объясните работу алгоритма исключения лишних операций. Приведите пример исключения лишних операций.

Варианты заданий
Варианты заданий соответствуют вариантам заданий для лабораторной работы № 3. Для выполнения работы рекомендуется использовать результаты, полученные в ходе выполнения лабораторных работ № 2 и 3.

Пример выполнения работы

Задание для примера
В качестве задания для примера возьмем язык, заданный КС-грамматикой G({if,then,else,a,=,or,xor,and,(,),},{S,F,_£,£), C},P,S) с правилами P:
S → F;
F → if-then T else F | if E then F | a:= E
T → if-then T else T | a:= E
E → E or D | E xor D | D
D → D and C | C
C → a | (E)
Жирным шрифтом в грамматике и в правилах выделены терминальные символы.
Этот язык уже был использован для иллюстрации выполнения лабораторных работ № 2 и № 3.
Результатом примера выполнения лабораторной работы № 4 будет генератор списка триад. Преобразование списка триад в ассемблерный код рассмотрено далее в примере выполнения курсовой работы (см. главу «Курсовая работа»).

Построение схем СУ-перевода
Все операции, которые могут присутствовать во входной программе на языке, заданном грамматикой G, по смыслу (семантике) можно разделить на следующие группы:
• логические операции (or, xor и and);
• оператор присваивания;

• полный условный оператор (if…then… else…) и неполный условный оператор (if… then…);
• операции, не несущие смысловой нагрузки, а служащие только для создания синтаксических конструкций исходной программы (в данном языке таких операций две: круглые скобки и точка с запятой).

Рассмотрим схемы СУ-перевода для всех перечисленных групп операций.

СУ-перевод для линейных операций

Линейной операцией будем называть такую операцию, для которой порождается код, представляющий собой линейный участок результирующей программы. Например, рассмотренные ранее бинарные арифметические операции (см. раздел «Краткие теоретические сведения») являются линейными.

В заданном входном языке логические операции выполняются над целыми десятичными числами как побитовые операции, то есть они также являются бинарными линейными операциями. Поэтому для них могут быть использованы те же самые схемы СУ-перевода, что были рассмотрены ранее.

Примечание.

На самом деле возможен другой вариант вычисления логических операций в том случае, когда они являются операциями булевой логики и их операндами могут быть только значения «Истина» (1) и «Ложь» (0). Здесь этот вариант не рассматривается. Более подробно о нем сказано в разделе «Курсовая работа», когда строятся схемы СУ-перевода для логических операций, а также можно обратиться к литературе [2].

СУ-перевод для оператора присваивания

Оператор присваивания также является бинарной логической операцией, поэтому для него может быть использована соответствующая схема СУ-перевода.

Отличие оператора присваивания от прочих бинарных линейных операций заключается в том, что первым операндом у него всегда должна быть переменная. Поэтому функция, строящая код для оператора присваивания, должна проверять тип первого операнда. Эта проверка представляет собой реализацию простейшего семантического анализа и в данном случае необходима, так как присваивание значений константам не отслеживается на этапе синтаксического анализа (об этом было сказано в лабораторной работе № 3).

СУ-перевод для условных операторов

Для условных операторов генерация кода должна выполняться в следующем порядке:

1. Порождается блок кода№ 1, вычисляющий логическое выражение, находящееся между лексемами if (первая нижележащая вершина) и then (третья нижележащая вершина), – для этого должна быть рекурсивно вызвана функция порождения кода для второй нижележащей вершины.

2. Порождается команда условного перехода, которая передает управление в зависимости от результата вычисления логического выражения:
• в начало блока кода № 2, если логическое выражение имеет ненулевое значение;
• в начало блока кода № 3 (для полного условного оператора) или в конец оператора (для неполного условного оператора), если логическое выражение имеет нулевое значение.
3. Порождается блок кода № 2, соответствующий операциям после лексемы then (третья нижележащая вершина), – для этого должна быть рекурсивно вызвана функция порождения кода для четвертой нижележащей вершины.
4. Для полного условного оператора порождается команда безусловного перехода в конец оператора.
5. Для полного условного оператора порождается блок кода № 3, соответствующий операциям после лексемы else (пятая нижележащая вершина), – для этого должна быть рекурсивно вызвана функция порождения кода для шестой нижележащей вершины.
Схемы СУ-перевода для полного и неполного условных операторов представлены на рис. 4.1.

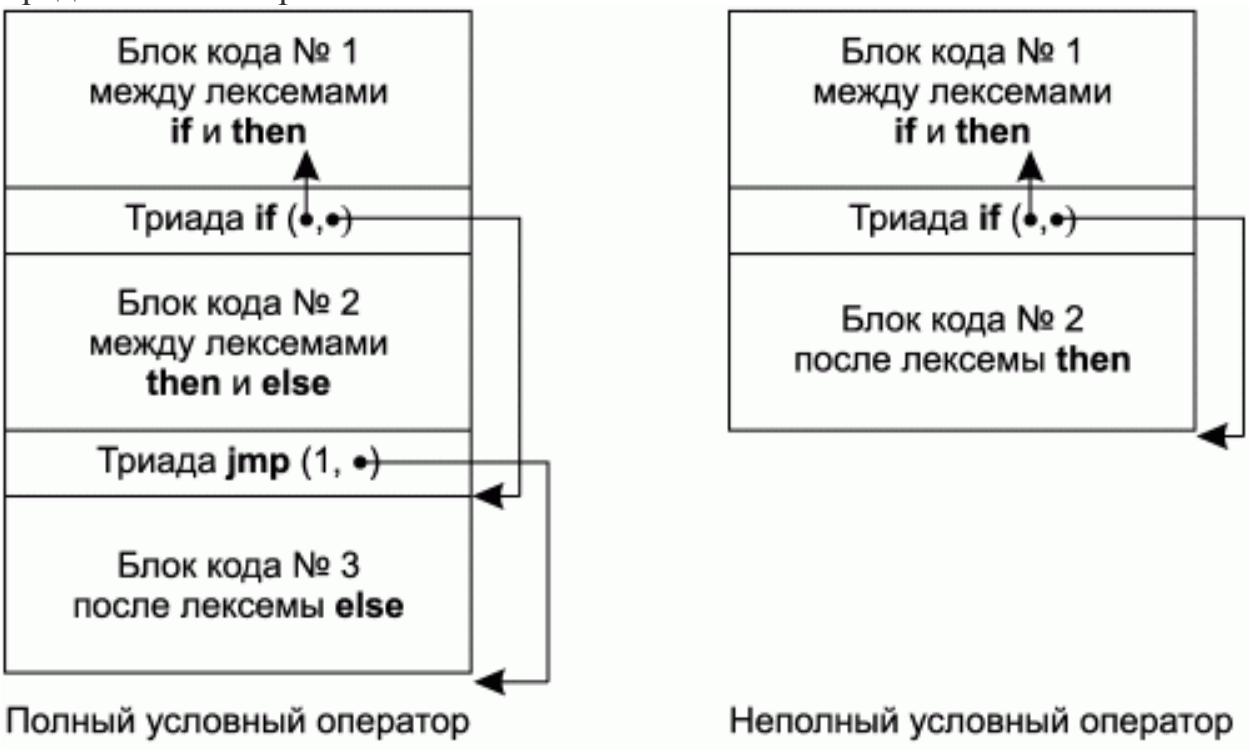

Рис. 4.1. Схемы СУ-перевода для условных операторов.
Для того чтобы реализовать эти схемы, необходимы два типа триад: триада условного перехода и триада безусловного перехода.
Эти два типа триад реализуются следующим образом:
• IF(<операнд1>,<операнд2>) – триада условного перехода;
• JMP(1,<операнд2>) – триада безусловного перехода.
У триады IF первый операнд может быть переменной, константой или ссылкой на другую триаду, второй операнд – всегда ссылка на другую триаду. Триада

IF передает управление на триаду, указанную вторым операндом, если первый операнд равен нулю, иначе управление передается на следующую триаду.
У триады JMP первый операнд не имеет значения (для определенности он всегда будет равен 1), второй операнд – всегда ссылка на другую триаду. Триада JMP всегда передает управление на триаду, указанную вторым операндом.
СУ-перевод для семантически ненагруженных конструкций
Операции, которые не несут никакой смысловой нагрузки, не требуют построения результирующего кода. Для них не требуется строить схемы СУ-перевода.
Тем не менее функция генерации списка триад должна обрабатывать и эти операции.
Они должны обрабатываться следующим образом:
• для вершины, у которой первая нижележащая вершина – открывающая скобка, вторая нижележащая вершина – узел дерева (не концевая вершина) и третья нижележащая вершина – закрывающая скобка, должна рекурсивно вызываться функция порождения кода для второй нижележащей вершины;
• для вершины, у которой первая нижележащая вершина – узел дерева (не концевая вершина) и вторая нижележащая вершина – точка с запятой, должна рекурсивно вызываться функция порождения кода для первой нижележащей вершины.

Пример генерации списка триад
Возьмем в качестве примера входную цепочку:
if a and b or a and b and 345 then a:= 5 or 4 and 7;
В результате лексического и синтаксического разбора этой входной цепочки будет построено дерево синтаксического разбора, приведенное на рис. 4.2.
Этому дереву будет соответствовать следующая последовательность триад:
1: and (a, b)
2: and (a, b)
3: and (^2, 345)
4: or (^1, ^3)
5: if (^4, ^9)
6: and (4, 7)
7: or (5, ^6)
8::= (a, ^7)
9:…
В этой последовательности два линейных участка: от триады 1 до триады 5 и от триады 6 до триады 9.
После оптимизации методом свертки объектного кода получим последовательность триад:
1: and (a, b)
2: and (a, b)
3: and (^2, 345)
4: or (^1, ^3)

5: if (^4, ^9)
6: C (4, 0)
7: C (5, 0)
8::= (a, 5)
9:…
Если удалить триады типа C, то эта последовательность примет следующий вид:
1: and (a, b)
2: and (a, b)
3: and (^2, 345)
4: or (^1, ^3)
5: if (^4, ^7)
6::= (a, 5)
7:…

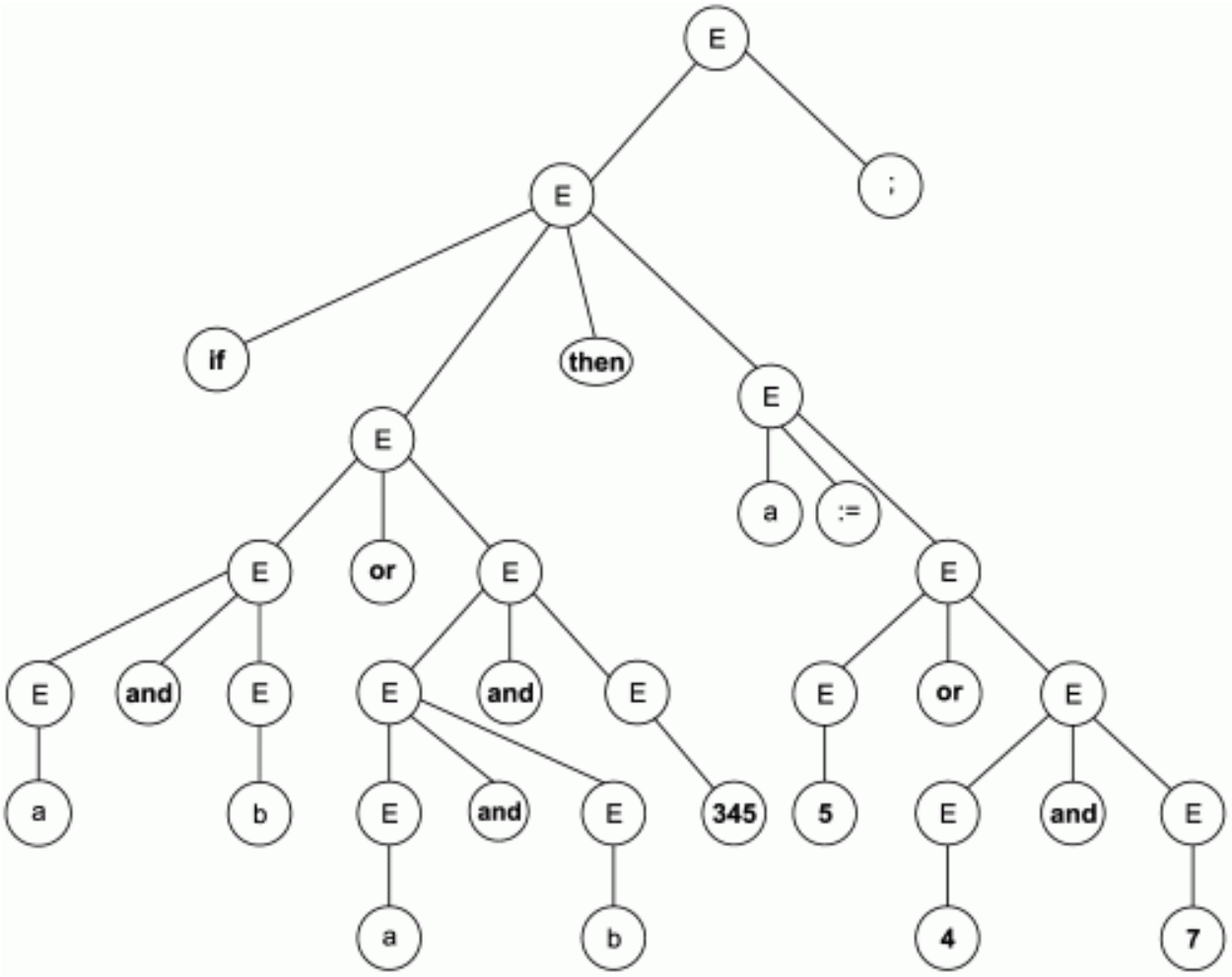

Рис. 4.2. Дерево синтаксического разбора цепочки «if a and b or a and b and 345 then a:= 5 or 4 and 7;».
После оптимизации методом исключения лишних операций получим последовательность триад:
1: and (a, b)
2: same (^1, 0)
3: and (^1, 345)
4: or (^1, ^3)

5: if (^4, ^7)
6::= (a, 5)
7:…
Если удалить триады типа same, то эта последовательность примет следующий вид:
1: and (a, b)
2: and (^1, 345)
3: or (^1, ^2)
4: if (^3, ^6)
5::= (a, 5)
6:…
После применения оптимизации получаем последовательность из пяти триад. Это на 37,5 % меньше, чем в исходной без применения оптимизации последовательности, состоявшей из восьми триад. Следовательно, объем результирующего кода и время его выполнения в данном случае сократятся примерно на 37,5 % (слово «примерно» указано здесь потому, что разные триады могут порождать различное количество команд в результирующем коде, а потому соотношения между количеством триад и между количеством команд объектного кода могут немного различаться).
Можно еще обратить внимание на то, что алгоритм оптимизации методом исключения лишних операций не учитывает особенности выполнения логических и арифметических операций. Методами булевой алгебры последовательность операций «a and b or a and b and 345» можно преобразовать в «a and b» точно так же, как последовательность операций «a-b + a-b-345» – в «a-b-346», что было бы эффективней, чем варианты, которые строит алгоритм оптимизации методом исключения лишних операций. Но для таких преобразований нужны алгоритмы, ориентированные на особенности выполнения логических и арифметических операций [1, 2, 7].

Реализация генератора списка триад
Разбиение на модули
Так же, как и для лабораторных работ № 2 и 3, модули, реализующие генератор списка триад, в лабораторной работе № 4 разделены на две группы:
• модули, программный код которых не зависит от входного языка;
• модули, программный код которых зависит от входного языка.
В первую группу входят модули:
• Triads – описывает структуры данных для представления триад;
• TrdOpt – реализует два алгоритма оптимизации: методом свертки объектного кода и методом исключения лишних операций;
• FormLab4 – описывает интерфейс с пользователем.
Во вторую группу входят модули:
• TrdType – описывает допустимые типы триад и их текстовое представление;
• TrdMake – строит список триад на основе дерева синтаксического разбора;
• TrdCal c – обеспечивает вычисление значений для триад разных типов при свертке объектного кода.

Такое разбиение на модули позволяет использовать те же самые структуры данных для организации нового генератора списка триад при изменении входного языка.
Кроме этих модулей для реализации лабораторной работы № 4 используются следующие программные модули:
• TblElem и FncTree – позволяют работать с комбинированной таблицей идентификаторов (созданы при выполнении лабораторной работы № 1);
• LexType, LexElem, и LexAuto – обеспечивают работу лексического распознавателя (созданы при выполнении лабораторной работы № 2);
• SyntRule и SyntSymb – обеспечивают работу синтаксического распознавателя (созданы при выполнении лабораторной работы № 3).
Кратко опишем содержание программных модулей, используемых для организации генератора списка триад.
Модуль описания допустимых типов триад
Модуль TrdType содержит структуры данных, которые описывают допустимые типы триад.
Он содержит следующие важные типы данных и переменные:
• TTriadType – перечисление всех возможных типов триад;
• TriadStr – массив строковых обозначений для всех типов триад;
• TriaD1ineSet – множество тех триад, которые являются линейными операциями (оно важно для оптимизации и для порождения кода).
Модуль описания структур данных для триад
Модуль Triads содержит структуры данных, которые описывают триады и список триад. Эти структуры зависят от реализации компилятора, но не зависят от входного языка.
Он содержит следующие важные структуры данных:
• TOperand – описывает операнд триады;
• TTriad – описывает триаду и все связанные с нею данные;
• TTriaD1ist – описывает список триад.
Структура TOperand описывает операнд триады. Она содержит следующие данные:
• OpType – тип операнда, который может принимать три значения:
– OPC0NST – константа;
– OPVAR – переменная (идентификатор);
– OPLINK – ссылка на другую триаду;
• и дополнительную информацию по операнду:
– ConstVal – значение (для константы);
– VarLink – ссылка на таблицу идентификаторов (для переменной);
– TriadNum – номер триады (для ссылки на триаду).
Один из вопросов, который необходимо было решить при реализации операндов триад, состоял в следующем: что использовать для описания ссылки на триаду – непосредственно ссылку на тип данных (указатель) или номер триады в списке?
Оба варианта имеют свои преимущества и недостатки:

• при использовании указателя легче осуществлять доступ к триаде (не надо выбирать ее из списка), не надо менять указатели при перемещении триад в списке, но при удалении любой триады из списка нужно корректно менять все указатели на эту триаду, какие только есть;
• при использовании номера триады легче порождать список триад по дереву разбора, но при любом перемещении и удалении триад из списка нужно пересчитывать все номера.

Какой вариант выбрать, решает разработчик компилятора. В данном случае автор выбрал второй вариант (номер триады, а не указатель на нее), поскольку наглядная иллюстрация алгоритмов оптимизации требует удаления триад, а перестановка указателей при каждом удалении намного сложнее, чем изменение номеров (этот недостаток оказался решающим). Но поскольку в реальном компиляторе не нужно иллюстрировать работу алгоритмов оптимизации выводом списка триад (достаточно просто не порождать код для триад с типами C и same), в этом случае указатели, по мнению автора, были бы предпочтительнее.

Структура TTriad описывает триаду и все связанные с ней данные. Она содержит следующие поля данных:
• TriadType – тип триады (один из перечисленных в типе TTriadType в модуле TrdType);
• Operands – массив операндов триады (из двух операндов типа TOperand);
• Info – дополнительная информация о триаде для алгоритмов оптимизации;
• IsLinked – флаг, сигнализирующий о том, что на триаду имеется ссылка из другой триады, обеспечивающей передачу управления (типа IF или JMP).

Для хранения дополнительной информации можно было использовать один из двух подходов: хранить ее непосредственно в самой триаде или хранить внутри триады только ссылку (указатель), а саму дополнительную информацию размещать во внешней структуре данных.

Этот вопрос уже возникал при выборе метода хранения информации при организации таблиц идентификаторов в лабораторной работе № 1. Тогда было отдано предпочтение второму варианту, поскольку характер и размер хранимой информации для каждого идентификатора был неизвестен.

В данном случае известно, что для каждой триады потребуется хранить информацию, обрабатываемую двумя алгоритмами оптимизации – алгоритмом свертки объектного кода и алгоритмом исключения лишних операций. Оба эти алгоритма работают со значениями, которые могут принимать триады – для заданного входного языка это целые десятичные числа. Для их хранения достаточно одного целочисленного поля (два алгоритма никогда не выполняются одновременно, а потому могут использовать одно и то же поле данных). Поэтому тут выбран первый вариант и хранимая информация включена непосредственно в структуру данных триады в виде поля Info.

Флаг наличия ссылки важен для определения границ линейных участков программы при оптимизации: если на какую-то триаду есть ссылка из триад типа IF или JMP, значит, на нее может быть передано управление. Такая триада

является возможной точкой входа участка программы, а потому – границей линейного участка.
Кроме перечисленных данных структура TTriad содержит следующие процедуры и функции:
• конструктор Create для создания триады;
• функцию проверки совпадения двух триад IsEqual;
• функцию MakeString, формирующую строковое представление триады для отображения триад на экране;
• функции, процедуры и свойства для доступа к данным триады.
Нужно обратить внимание, что функция проверки совпадения двух триад IsEqual считает триады эквивалентными, если они имеют один тип и одинаковые операнды. Эта функция нужна для выполнения алгоритма исключения лишних операций – она проверяет первое условие того, что операция является лишней, то есть имеется ли совпадающая с ней операция. Второе условие (что ни один из операндов не изменялся между двумя операциями) проверяется с помощью чисел зависимости.
Структура данных TTriaD1ist описывает список триад и методы работы с ним. Как и некоторые списки, рассмотренные ранее (в лабораторных работах № 2 и 3), она построена на основе динамического массива типа TList из библиотеки VCL системы программирования Delphi 5. В этой структуре нет никаких данных (используются только данные, унаследованные от класса TList), но с ней связаны методы, необходимые для работы со списком триад:
• функция очистки списка триад (Clear) и деструктор для освобождения памяти при удалении списка триад (Destroy);
• функция записи списка триад в текстовом представлении в список строк для отображения списка триад на экране (WriteToList);
• функция удаления триады из списка (DelTriad);
• функция GetTriad и свойство Triads для доступа к триадам в списке по их порядковому номеру.
Следует отметить, что функция записи списка триад в список строк (WriteToList) последовательно вызывает функцию MakeString для записи в список строк каждой триады из списка триад. Функция удаления триады из списка (DelTriad) освобождает память, занятую удаляемой триадой, а кроме того, следит за тем, чтобы при удалении триады флаг метки (IsLinked) от удаляемой триады был корректно переставлен на следующую по списку триаду.
Кроме трех перечисленных структур данных в модуле Triads описана также функция DelTriadTypes, которая выполняет удаление из списка триад всех триад заданного типа. Эта функция необходима только для наглядной иллюстрации работы алгоритмов оптимизации. Для этого надо удалять из списка триад триады с типами C и same, которые не порождают результирующего кода.
Удаление триад из списка можно выполнить в виде двух вложенных циклов:
• первый обеспечивает просмотр всего списка триад;

• второй обеспечивает изменение номеров всех ссылок и всех последующих триад в списке при удалении какой-либо триады.

Тогда среднее количество просмотров списка триад можно оценить как N + K-N-N, где N – количество триад в списке, К – средний процент удаляемых триад. При хорошей оптимизации, когда К велико, время работы функции удаления триад из списка будет квадратично зависеть от количества триад. При увеличении объема результирующей программы (при росте N) это время будет существенно возрастать.

Поэтому функция удаления триад из списка реализована другим путем. Она выполняет два просмотра списка триад:

1. На первом просмотре подсчитывается количество удаляемых триад и для каждой триады запоминается, на какую величину изменится ее номер при удалении.
2. На втором просмотре удаляются те триады, которые должны быть удалены, а для остальных номера и ссылки меняются на величину, запомненную при первом просмотре.

При такой реализации функции количество просмотров списка триад всегда будет равно 2N и обеспечит линейную зависимость времени выполнения функции от количества триад. Правда, в таком случае функция потребует еще дополнительно N ячеек памяти для хранения изменений индексов каждой триады, но это оправдано существенным выигрышем во времени ее выполнения.

Модуль построения списка триад по дереву синтаксического разбора

Модуль TrdMake содержит функцию, которая строит список триад на основе дерева синтаксического разбора. Эта функция работает с типами триад, описанными в модуле TrdType, и со структурами данных, описанными в модуле Triads. Дерево синтаксического разбора описано структурами данных из модуля SyntSymb, который был создан при выполнении лабораторной работы № 3. Функция построения списка триад на основе синтаксического дерева зависит от входного языка, а потому вынесена в отдельный модуль.

Модуль содержит одну функцию, доступную извне, – MakeTriaD1ist. Входными данными этой функции являются:

• symbTop – ссылка на корень синтаксического дерева, по которому строится список триад;
• listTriad – список, в который должны быть записаны построенные триады.

Результатом выполнения функции является пустая ссылка, если при построении списка триад не было обнаружено семантических ошибок, или же ссылка на лексему, возле которой обнаружена семантическая ошибка, если такая ошибка обнаружена. Генератор списка триад обнаруживает один вид семантических ошибок – присваивание значения константе.

Функция MakeTriaD1ist выполняет построение списка триад, добавляет в конец списка триад завершающую триаду типа NOP (No Operation – Нет операции), чтобы корректно обрабатывать ссылки на конец списка триад, а также обеспечивает расстановку флагов IsLinked для всех триад в списке.

Функция MakeTriaD1ist построена на основе внутренней функции модуля TrdMake – MakeTriaD1istNOP, которая и выполняет главные действия по порождению списка триад. Эта функция обрабатывает те же входные данные и имеет такой же результат выполнения, что и функция MakeTriaD1ist.
Функция MakeTriaD1istNOP реализует схемы СУ-перевода, которые были рассмотрены выше. Выбор схемы СУ-перевода происходит по номеру правила остовной грамматики G', взятого из текущего нетерминального символа дерева:
• для правил 2 и 5 – схема полного условного оператора;
• для правила 3 – схема неполного условного оператора;
• для правил 4 и 6 – схема оператора присваивания;
• для правил 7, 8 и 10 – схема для бинарных линейных операций;
• для правила 13 – схема для скобок;
• в остальных случаях – схема для точки с запятой.
Функция MakeTriaD1istNOP содержит две вспомогательные функции:
• функцию MakeOperand для порождения кода, связанного с дочерним узлом дерева (одним из операндов);
• функцию MakeOperation, реализующую схему СУ-перевода для бинарных линейных операций в зависимости от типа операции.
Для построения кода для нижележащих нетерминальных символов по дереву функция MakeTriaD1istNOP рекурсивно вызывает сама себя. Этот вызов реализован в функции MakeOperand, если нижележащий узел является нетерминальным символом, а также напрямую для узлов, связанных со скобками и с точкой с запятой (как было рассмотрено ранее при построении схем СУ-перевода).
Модуль вычисления значений триад на этапе компиляции
Модуль TrdCalc содержит функцию, которая вызывается, когда необходимо вычислить значение триады на этапе компиляции. Эта функция нужна для алгоритма оптимизации методом свертки объектного кода. Она зависит от типов триад, которые зависят от входного языка, поэтому вынесена в отдельный модуль.
Модуль содержит одну-единственную функцию CalcTriad, которая предельно проста и в комментариях не нуждается.
Модуль, реализующий алгоритмы оптимизации
Модуль TrdOpt реализует два алгоритма оптимизации списка триад:
• методом свертки объектного кода;
• методом исключения лишних операций.
Алгоритмы, реализованные в модуле TrdOpt, в общем случае не зависят от входного языка, однако они обрабатывают триады типа «присваивание» (в данной реализации – TRDASSIGN). Кроме того, границы линейных участков, на которых работают эти алгоритмы, зависят от триад условного и безусловного перехода (в данной реализации – TRDIF и TRDJMP). Сами алгоритмы требуют для себя триад специального типа, которые в данном случае реализованы как TRDC и TRDSAME.

В итоге реализация алгоритмов оптимизации зависит от следующих типов триад:
• триад присваивания;
• триад условного и безусловного перехода;
• триад специальных типов.
В общем случае эти типы триад и их реализация зависят от входного языка (кроме триад специальных типов, которые разработчик компилятора может реализовать по своему усмотрению). Но поскольку сложно представить себе язык программирования, в котором не было бы операций присваивания, условных и безусловных переходов, можно считать, что в такой реализации модуль TrdOpt от входного языка не зависит.
Функция вычисления значений триад при свертке объектного кода, которая имеет явную зависимость от входного языка, вынесена в отдельный модуль (модуль TrdCalc, функция CalcTriad).
Кроме функций, реализующих алгоритмы оптимизации, модуль TrdOpt содержит две структуры данных:
• TConstInfo – для хранения информации о значениях переменных;
• TDepInfo – для хранения информации о числах зависимости переменных.
Обе эти структуры построены на основе структуры TAddVarInfo, описанной в модуле TblElem (этот модуль был создан при выполнении лабораторной работы № 1), и предназначены для хранения информации, связанной с переменной в таблице идентификаторов.
Структура TConstInfo хранит информацию о значении переменной, если оно известно. Она используется в алгоритме оптимизации методом свертки объектного кода.
Структура TDepInfo хранит информацию о числе зависимости переменной. Она используется в алгоритме оптимизации методом исключения лишних операций.
Каждая из этих структур имеет одно поле, которое и предназначено для хранения информации. Для доступа к этому полю используются виртуальные функции и связанные с ними свойства, которые переопределяют функции и свойства типа данных TAddVarInfo.
Эти структуры данных создаются по мере выполнения соответствующих алгоритмов и уничтожаются после завершения их выполнения.
Теперь можно сравнить два подхода к хранению дополнительной информации:
1. Хранение информации внутри структур данных (реализовано для триад).
2. Хранение внутри структур данных только ссылок (указателей), а самой информации – во внешних структурах.
Первый подход имеет следующие преимущества:
• доступ к хранимой информации осуществлять проще и быстрее;
• нет необходимости работать с динамической памятью, выделять и освобождать ее по мере надобности.
В то же время первый подход имеет ряд недостатков:

• при хранении разнородной информации оперативная память расходуется неэффективно, будут появляться неиспользуемые поля данных на разных стадиях компиляции;
• обеспечивается меньшая гибкость в обработке информации.
Второй подход имеет следующие преимущества:
• можно хранить разнородную информацию в зависимости от потребностей на каждой стадии компиляции;
• оперативная память расходуется только на хранение необходимой информации и только тогда, когда она действительно используется;
• обеспечивается более гибкая обработка информации (например, легко реализуется понятие «отсутствие данных» в алгоритме оптимизации методом свертки объектного кода через пустую ссылку nil).
Но и он имеет ряд недостатков:
• использование ссылок увеличивает время доступа к хранимой информации, что может быть важно при обработке компилятором больших объемов данных;
• использование ссылок требует работы с динамической памятью, выделения и освобождения памяти по мере использования информации, что расходует время и ресурсы ОС.
Какой подход выбрать в каждом конкретном случае, решает разработчик компилятора, принимая во внимание их достоинства и недостатки. Здесь проиллюстрирована реализация обоих подходов: первого – для идентификаторов (переменных) в лабораторных работах № 1 и 4, второго – для триад в лабораторной работе № 4. Почему были выбраны именно эти подходы, было описано ранее и для переменных, и для триад.
Алгоритмы оптимизации реализованы в модуле TrdOpt в виде двух процедур:
• OptimizeConst – для оптимизации методом свертки объектного кода;
• OptimizeSame – для оптимизации методом исключения лишних операций.
Обе процедуры принимают на вход один параметр – список триад. Все необходимые операции выполняются над этим списком, поэтому результатом их работы будет тот же самый список, в котором некоторые триады изменены, а другие заменены на триады специального вида:
• C (TRDC) – при оптимизации методом свертки объектного кода;
• Same (TRDSAME) – при оптимизации методом исключения лишних операций.
Триады специального вида можно удалить из общего списка триад с помощью функции удаления триад заданного типа (DelTriadTypes), которая была описана в модуле Triads. В принципе, нет необходимости выполнять это, так как на порождаемый объектный код эта операция никак не влияет – триады специального вида не порождают никакого кода, но для иллюстрации работы алгоритмов оптимизации такая операция полезна.
Процедуры OptimizeConst иOptimizeSame реализуют алгоритмы оптимизации, которые были описаны в разделе «Краткие теоретические сведения», поэтому в дополнительных пояснениях не нуждаются.

Можно отметить только, что для хранения информации, связанной с переменными (значения переменных и числа зависимости переменных), эти процедуры используют непосредственно таблицу идентификаторов. И в этом случае проявляются преимущества того, что в триадах в качестве ссылки на переменную используется именно ссылка на таблицу идентификаторов, а не на имя переменной. Эффективность прямого обращения в таблицу за требуемым значением намного выше, чем поиск переменной по ее имени. Это справедливо для любых операций, выполняемых компилятором на этапах подготовки к генерации кода, генерации кода и оптимизации.

Текст программы генератора списка триад

Кроме перечисленных модулей необходим еще модуль, обеспечивающий интерфейс с пользователем. Этот модуль (FormLab4) реализует графическое окно TLab4Form на основе класса TForm библиотеки VCL и включает в себя две составляющие:

• файл программного кода (файл FormLab4.pas);

• файл описания ресурсов пользовательского интерфейса (файл FormLab4.dfm).

Модуль FormLab4 построен на основе модуля FormLab3, который использовался для реализации интерфейса с пользователем в лабораторной работе № 3. Он содержит все данные, управляющие и интерфейсные элементы, которые были использованы в лабораторных работах № 2 и 3. Такой подход оправдан, поскольку первым этапом лабораторной работы № 4 является лексический анализ, который выполняется модулями, созданными для лабораторной работы № 2, а вторым этапом – синтаксический анализ, который выполняется модулями, созданными для лабораторной работы № 3.

Кроме данных, используемых для выполнения лексического и синтаксического анализа так, как это было описано в лабораторных работах № 2 и 3, модуль содержит поле listTriad, которое представляет собой список триад. Этот список инициализируется при создании интерфейсной формы и уничтожается при ее закрытии. Он также очищается всякий раз, когда запускаются процедуры лексического и синтаксического анализа.

Кроме органов управления, использованных в лабораторной работе № 3, интерфейсная форма, описанная в модуле FormLab4, содержит органы управления для генератора списка триад лабораторной работы № 4:

• в многостраничной вкладке (PageControll) появилась новая закладка (Sheet-Triad) под названием «Триады»;

• на закладке SheetTriad расположены интерфейсные элементы для вывода и просмотра списков триад (группа с заголовком и список строк для отображения каждого списка триад):

GroupTriadAll и ListTriadAll – для отображения полного списка триад, построенного до применения алгоритмов оптимизации;

GroupTriadConst и ListTriadConst – для отображения списка триад, построенного после оптимизации методом свертки объектного кода;

GroupTriadSame и ListTriadSame – для отображения списка триад, построенного после оптимизации методом исключения лишних операций.
• на той же закладке SheetTriad расположены два сплиттера для управления размерами списков триад;
• на первой закладке SheetFi 1 e («Исходный файл») появились два дополнительных органа управления – флажки с двумя состояниями («пусто» или «отмечено»):
CheckDelC – при установке этого флажка триады типа C удаляются из списка триад после выполнения оптимизации методом свертки объектного кода;
CheckDelSame – при установке этого флажка триады типа same удаляются из списка триад после выполнения оптимизации методом исключения лишних операций.
Внешний вид новой закладки интерфейсной формы TLab4Form приведен на рис. 4.3.

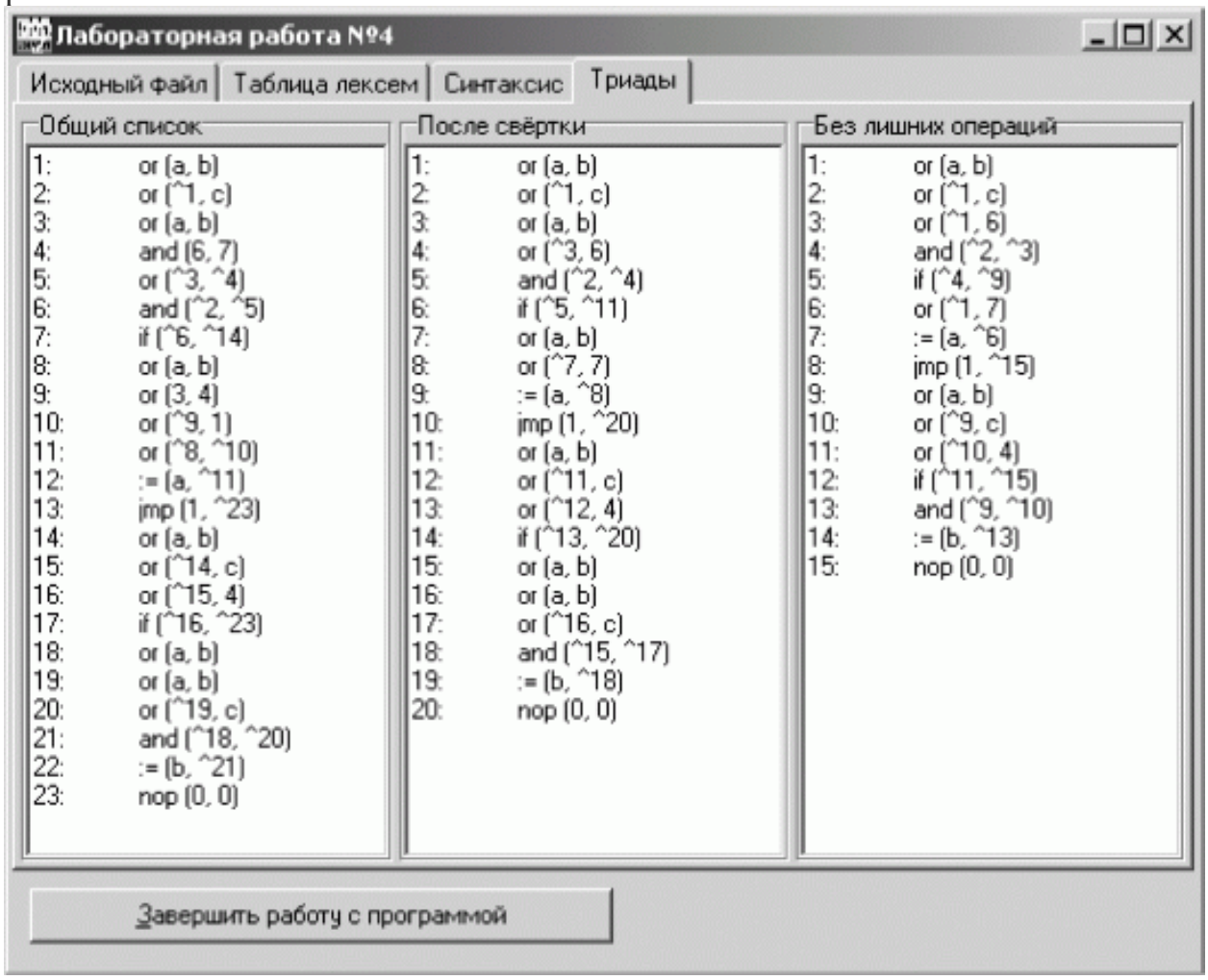

Рис. 4.3. Внешний вид четвертой закладки интерфейсной формы для лабораторной работы № 4.

Чтение содержимого входного файла организовано точно так же, как в лабораторной работе № 2.

После чтения файла выполняется лексический анализ, как это было описано в лабораторной работе № 2, а затем, при успешном выполнении лексического

анализа, синтаксический анализ, как это было описано в лабораторной работе № 3.
Если синтаксический анализ выполнен успешно, полученная в результате его выполнения переменная symbRes указывает на корень построенного синтаксического дерева. Тогда, после того как синтаксическое дерево отобразится на экране с помощью функции MakeTree, вызывается функция построения списка триад по синтаксическому дереву MakeTriaD1ist (из модуля TrdMake). Список триад запоминается в список listTriad, а результат выполнения функции – во временную переменную lexTmp.
Если переменная lexTmp после построения списка триад содержит непустую ссылку на лексему, это значит, что исходная программа содержит семантическую ошибку. Лексема, на которую указывает lexTmp, определяет место, где обнаружена ошибка. В этом случае список строк позиционируется на место ошибки и пользователю выдается соответствующее сообщение.
Иначе, если переменная lexTmp после построения списка триад содержит пустую ссылку (nil), это значит, что построение списка триад выполнено без ошибок, и список listTriad содержит все построенные триады в порядке их следования. Список триад отображается на экране в списке строк ListTriadAll, после чего выполняется оптимизация методом свертки объектного кода – вызывается процедура OptimizeConst. Если установлен флажок CheckDel_C, то после оптимизации методом свертки объектного кода из списка триад удаляются триады типа C (вызывается функция DelTriadTypes с параметром TRD_CONST), после чего список триад отображается в списке строк ListTriadConst. Затем выполняется оптимизация методом исключения лишних операций – вызывается процедура OptimizeSame. Если установлен флажок CheckDelSame, то после оптимизации методом исключения лишних операций из списка триад удаляются триады типа same (вызывается функция DelTriadTypes с параметром TRD_SAME), после чего список триад отображается в списке строк ListTriadSame.
Полный текст программного кода модуля интерфейса с пользователем и описание ресурсов пользовательского интерфейса можно найти в архиве, который располагается на веб-сайте издательства, в файлах FormLab4.pas и FormLab4.dfm соответственно.
Полный текст всех программных модулей, реализующих рассмотренный пример для лабораторной работы № 4, можно найти в архиве, располагающемся на вебсайте издательства, в подкаталогах LABS и COMMON (в подкаталог COMMON вынесены те программные модули, исходный текст которых не зависит от входного языка и задания по лабораторной работе). Главным файлом проекта является файл LAB4.DPR в подкаталоге LABS. Кроме того, текст модуля Triads приведен в листинге П3.10, а текст модуля TrdOpt – в листинге П3.11 в приложении 3.

Выводы по проделанной работе

В результате лабораторной работы № 4 построен генератор списка триад, порождающий триады для логических операций, оператора присваивания и

условного оператора. Генератор списка триад обнаруживает семантические ошибки, связанные с присваиванием значений константам (когда первый операнд оператора присваивания – константа). При наличии одной ошибки пользователю выдается сообщение с указанием местоположения ошибки. При наличии нескольких ошибок обнаруживается только первая из них, и дальнейший анализ исходного текста прекращается.

Построенный генератор также выполняет оптимизацию списка триад методом свертки объектного кода и исключения лишних операций, что позволяет сократить объем результирующего списка триад и время выполнения объектного кода, который может быть построен на его основе. После выполнения оптимизации генератор списка триад может удалять из списка триады специального вида C и same в зависимости от настроек, сделанных пользователем.

Построенный при выполнении данной лабораторной работы генератор списка триад входит в состав компилятора, в который также входят: лексический анализатор, построенный при выполнении лабораторной работы № 2, и синтаксический анализатор, построенный при выполнении лабораторной работы № 3. Этот компилятор получает на вход исходную программу в соответствии с заданной грамматикой и порождает результирующую программу в виде списка триад.

Компилятор позволяет обнаруживать следующие однократные ошибки:

• любые лексические ошибки (неправильные лексемы);

• любые синтаксические ошибки (несоответствие исходной программы синтаксису заданного входного языка);

• семантические ошибки типа «присваивание значения константе».

При обнаружении ошибки пользователю выдается сообщение о типе ошибки (лексическая, синтаксическая или семантическая) и о местонахождении ошибки в тексте исходной программы. Дальнейший анализ типа обнаруженной ошибки не производится. При наличии нескольких ошибок в исходной программе обнаруживается только первая из них.

В результате выполнения лабораторных работ № 1–4 построен компилятор, выполняющий обработку исходной программы за пять проходов:

1. Лексический анализ исходного текста и построение таблицы лексем.
2. Синтаксический анализ по таблице лексем и построение дерева синтаксического разбора.
3. Построение списка триад по дереву синтаксического разбора.
4. Оптимизация списка триад методом свертки объектного кода.
5. Оптимизация списка триад методом исключения лишних операций.

На каждом проходе компилятора исходными данными являются результаты, полученные при выполнении предыдущего прохода.

Количество проходов построенного компилятора может быть существенно сокращено, поскольку все операции выполняются последовательно, независимо друг от друга, однако это не входит в задачу выполненных лабораторных работ.